HAUFE.
Ausgabe 04/2025

ACCESS
IM UNTERNEHMEN

ICONS IN ACCESS

Platzieren Sie Icons in Ribbons, Kontextmenüs, Formularen und im TreeView mit unserem vereinfachten System (Seite 12)

In diesem Heft:

SUCHEN UND ERSETZEN OPTMIERT

Nutzen Sie unsere neue Suchfunktion mit Gedächtnis für Suchbegriffe und Ersetzungen.

SEITE 32, 51

ACCESS-ANWENDUNG MIT UPDATE-FUNKTION

Prüfen Sie gleich beim Start, ob es ein Update zur Datenbank gibt, und installieren Sie es direkt!

SEITE 63

FORMULAR-TIMER IM DETAIL

Erfahren Sie alles, was Sie zur Verwendung von Formular-Timern wissen müssen.

SEITE 2

Mehr Glanz für Access-Anwendungen

Wussten Sie, dass kleine Icons oft einen großen Unterschied machen können? In vielen Access-Anwendungen liegt der Fokus traditionell auf Funktionalität — und das ist grundsätzlich auch richtig so. Schließlich sollen Ihre Daten zuverlässig verarbeitet, ausgewertet und präsentiert werden. Doch gerade in einer Zeit, in der Benutzer immer stärker an moderne, visuell ansprechende Oberflächen gewöhnt sind, spielt das Design eine entscheidende Rolle. Eine Anwendung wirkt sofort professioneller und intuitiver, wenn Sie durchgängig klare Symbole und einheitliche Icons verwenden.

Genau hier setzt unser Schwerpunktartikel dieser Ausgabe an: **Icons im Ribbon, Kontextmenü, Formular und TreeView verwenden** (ab Seite 12). Wir zeigen Ihnen, wie Sie Icons zentral in einer Ressourcentabelle speichern und anschließend in allen wichtigen Bereichen Ihrer Access-Anwendung einsetzen.

Ob im Ribbon, in eigenen Kontextmenüs, auf Formularschaltflächen oder in einem TreeView-Steuerelement — Schritt für Schritt erfahren Sie, wie Sie Ihre Oberfläche moderner und professioneller gestalten. Dabei lernen Sie, wie Sie die Icons technisch einbinden, aus der Tabelle laden und korrekt zuweisen. So verleihen Sie Ihren Anwendungen ein einheitliches, ansprechendes Erscheinungsbild.

Natürlich bietet diese Ausgabe noch viele weitere spannende Themen: Gleich zu Beginn erwartet Sie der Beitrag **Formular-Timer in Access mit VBA** (ab Seite 2). Damit können Sie Ihre Anwendungen dynamischer gestalten, zum Beispiel um Statusmeldungen automatisch zu aktualisieren oder Aktionen zeitgesteuert auszuführen.

In **Tabellen und Abfragen im Unterformular anzeigen** (ab Seite 6) erfahren Sie, wie Sie nicht nur fest definierte Formulare, sondern auch verschiedene Tabellen und Abfragen flexibel in Unterformularen darstellen können.

Ein weiteres praxisnahes Thema ist **Abfragen mit Unterabfragen als Datenquelle verwenden** (ab Seite 54). Hier lernen Sie, wie Sie Abfragen mit verschachtelten Unterabfragen effektiv als Grundlage für Formulare oder Berichte nutzen können, um komplexe Datenbeziehungen sauber aufzubereiten.

Der Beitrag **Datenblätter in Endlosformularen anzeigen** (ab Seite 58) zeigt Ihnen, wie Sie Datenblätter direkt in Endlosformularen integrieren und damit kombinierte Ansichten schaffen, die dem Benutzer mehr Kontext und Übersicht bieten.

Ein besonderes Highlight ist **Automatisches Update per Startdatenbank** (ab Seite 63). Diese Lösung stellt sicher, dass alle Benutzer stets mit der aktuellen Frontend-Version arbeiten, ohne dass Sie manuell eingreifen müssen.

Zum Abschluss finden Sie den Tipp **Add-In-Verzeichnis mit WizHook ermitteln** (ab Seite 74). Damit identifizieren Sie schnell den Speicherort Ihrer Access-Add-Ins — ein oft unterschätzter, aber sehr nützlicher Trick, der in der Praxis viel Zeit sparen kann.

Wir wünschen Ihnen viel Freude bei der Lektüre dieser Ausgabe und zahlreiche neue Impulse, wie Sie Ihre Anwendungen nicht nur funktional, sondern auch visuell und konzeptionell weiterentwickeln können. Lassen Sie sich inspirieren und bringen Sie Ihre Access-Projekte auf ein neues Niveau!

Herzlichst, Ihr André Minhorst

Formular-Timer in Access mit VBA

Formular-Timer in Access können ganz unterschiedliche Aufgaben übernehmen. Die beiden Hauptkategorien sind Benutzerinteraktion und das Umgehen von Einschränkungen durch das System. Bei der Benutzerinteraktion geht es darum, in den gewünschten Zeitintervallen Meldungen anzuzeigen, Informationen abzufragen et cetera. Bei den Systemeinschränkungen haben wir es mit Herausforderungen zu tun, bei denen wir zuerst eine asynchrone Aktion auslösen, die Daten für eine weitere Aktion liefern soll. Diese wird dann aber zu schnell ausgeführt, als dass die Daten bereits vorhanden sind. In diesem Fall können wir mit der asynchronen Aktion einen Formular-Timer starten, der die folgende Aktion erst nach einem ausreichend großen Zeitintervall auslöst, sodass alle Aktionen die notwendigen Daten zur Verfügung haben. In diesem Artikel zeigen wir, wie genau ein Formular-Timer mit VBA programmiert wird: Wie wir das Zeitintervall festlegen, wie wir diesen starten und auch, wie wir ihn nach der Erledigung wieder ausschalten.

Grundlagen zum Formular-Timer

Der Formular-Timer von Access basiert auf den beiden Eigenschaften **TimerInterval** und **OnTimer**. Mit **TimerInterval** legen wir fest, in welchen Zeitintervallen das Ereignis **OnTimer** ausgelöst wird. Der Wert für **TimerInterval** wird in Millisekunden angegeben.

Wenn wir also wollen, dass das **Timer**-Ereignis alle fünf Sekunden ausgelöst wird, setzen Sie **TimerInterval** auf 5000. Das Ereignis **OnTimer** enthält dann den VBA-Code, der bei jedem Intervall ausgeführt werden soll.

Timer über die Eigenschaften einstellen

Die Eigenschaften **TimerInterval** und **OnTimer** können wir entweder direkt im Eigenschaftenblatt des Formulars einstellen oder per VBA zur Laufzeit setzen. Im Eigenschaftenblatt geben wir bei **Zeitgeberintervall** die Millisekunden an und bei **Bei Zeitgeber** wählen wir den Eintrag **[Ereignisprozedur]** aus (siehe Bild 1).

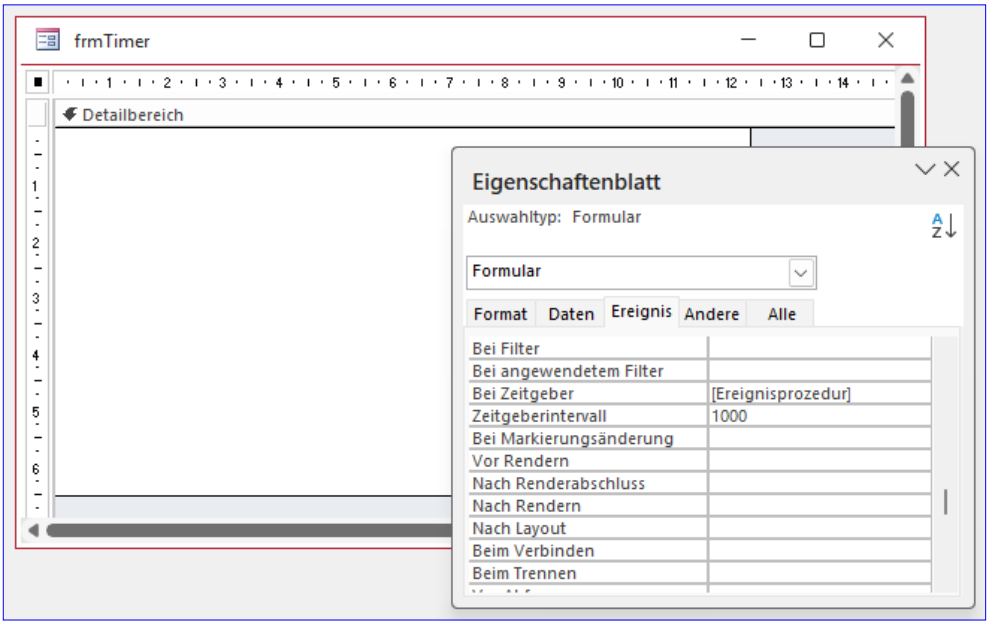

Bild 1: Einstellungen für einen Formular-Timer

Beim Einstellen der Eigenschaft **Zeitgeberintervall** haben wir gesehen, dass diese standardmäßig den Wert **0** enthält. Das bedeutet, dass der Zeitgeber nicht ausgelöst wird.

Nun klicken wir für die Eigenschaft **Bei Zeitgeber** die Schaltfläche mit den drei Punkten an. Im nun erscheinenden VBA-Editor fügen wir in diese Prozedur die folgenden Anweisungen ein:

```
Private Sub Form_Timer()
    i = i + 1
    Me.Caption = "Zähler Zeitgeber: " & i
End Sub
```

Die Variable **i** deklarieren wir ebenfalls in diesem Klassenmodul:

```
Dim i As Long
```

Wenn wir dieses Formular nun öffnen, erhalten wir in der Titelleiste jede Sekunde ein Update mit dem jeweiligen Wert der Variablen **i** (siehe Bild 2).

Timer per VBA starten und stoppen

Wenn Sie den Timer per VBA steuern möchten, können Sie die Eigenschaft **TimerInterval** direkt zur Laufzeit festlegen. Möchten Sie den Timer stoppen, setzen Sie **TimerInterval** einfach auf **0**.

Um ihn wieder zu starten, können wir die Eigenschaft **TimerInterval** einfach wieder auf den gewünschten Wert in Millisekunden einstellen.

Wir wollen dies mit zwei Schaltflächen erledigen. Die erste Schaltfläche heißt **cmdTimerStoppen** und soll den Timer anhalten. Die zweite heißt **cmdTimerFortsetzen** und soll den Timer wieder starten.

Für das Ereignis **Beim Klicken** dieser Schaltflächen hinterlegen wir jeweils eine Ereignisprozedur.

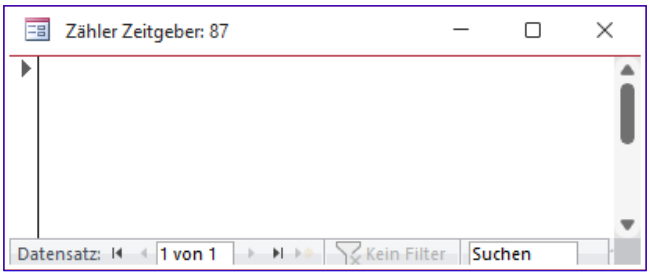

Bild 2: Formular mit Timer

Die Ereignisprozedur für **cmdTimerStoppen** lautet wie folgt und stellt die Eigenschaft **Zeitgeberintervall** auf **0** ein:

```
Private Sub cmdTimerStoppen_Click()
    Me.TimerInterval = 0
End Sub
```

Für **cmdTimerFortsetzen** hinterlegen wir die folgende Prozedur, die einfach das Zeitgeberintervall wieder auf **1.000** einstellt:

```
Private Sub cmdTimerFortsetzen_Click()
    Me.TimerInterval = 1000
End Sub
```

Damit erhalten wir das Ergebnis aus Bild 3.

Timer zurücksetzen

Wir wollen den Timer auch noch zurücksetzen können. Dazu fügen wir eine dritte Schaltfläche namens **cmd-**

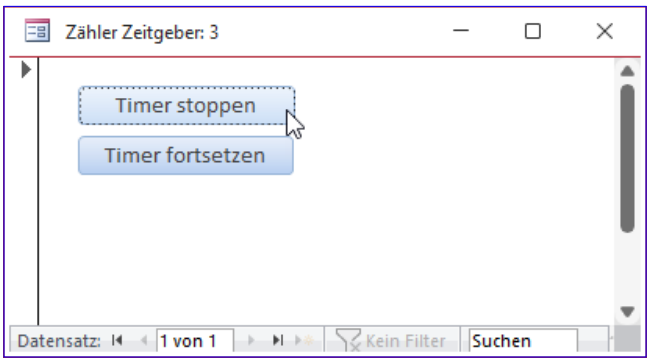

Bild 3: Zwei Schaltflächen zum Stoppen und Starten des Zeitgebers

Tabellen und Abfragen im Unterformular anzeigen

Unterformulare in der Datenblattansicht nutzen wir normalerweise, wenn wir die Daten einer festen Datenquelle passend zu den Daten im Hauptformular anzeigen wollen – zum Beispiel für Kunden im Hauptformular und ihre Bestellungen im Unterformular. Manchmal möchte man aber auch einfach mal schnell Daten aus beliebigen Tabellen einsehen. Warum dann nicht einfach die Tabelle oder Abfrage in der Datenblattansicht anzeigen? Weil wir dann im Gegensatz zum Haupt-/Unterformular viel weniger Möglichkeiten haben, was zum Beispiel den Einsatz von VBA angeht. Wenn wir uns etwa die Lösung aus dem Beitrag Markierte Datensätze aus dem Datenblatt einlesen (www.access-im-unternehmen.de/1548) anschauen, können wir dort per VBA die aktuell markierten Datensätze auslesen. Das ist bei Tabellen oder Abfragen, die wir alleinstehend öffnen, nicht möglich. Deshalb zeigen wir in diesem Beitrag zunächst, wie wir beliebige Tabellen oder Abfragen komfortabel über ein Kombinationsfeld auswählen und dann im Unterformular einblenden können. Anschließend bauen wir noch die Lösung aus dem oben genannten Beitrag ein.

Tabellen und Abfragen im Unterformular

In Microsoft Access gibt es schon lange die Möglichkeit, für ein Unterformular über die Eigenschaft **Herkunftsobjekt** nicht nur Formulare, sondern auch Tabellen oder Abfragen direkt in der Datenblattansicht anzuzeigen.

Für Formulare selbst gibt es diese Eigenschaft nicht, diese wird nur für das Unterformular-Steuerelement bereitgestellt.

Das können wir leicht testen, indem wir einem neuen, leeren Formular ein Unterformular-Steuerelement hinzufügen. Klicken wir dieses an, zeigt das Eigenschaftenblatt im Register **Daten** die Eigenschaft **Herkunftsobjekt** an, mit dem wir neben den Formularen auch alle vorhandenen Tabellen und Abfragen auswählen können (siehe Bild 1).

Um zwischen den verschiedenen Objekttypen unterscheiden zu können, wird diesen noch die Objektbezeichnung vorangestellt, zum Beispiel Formular, Tabelle oder Abfrage. Danach folgt, durch einen Punkt getrennt, der eigentliche Name des Objekts.

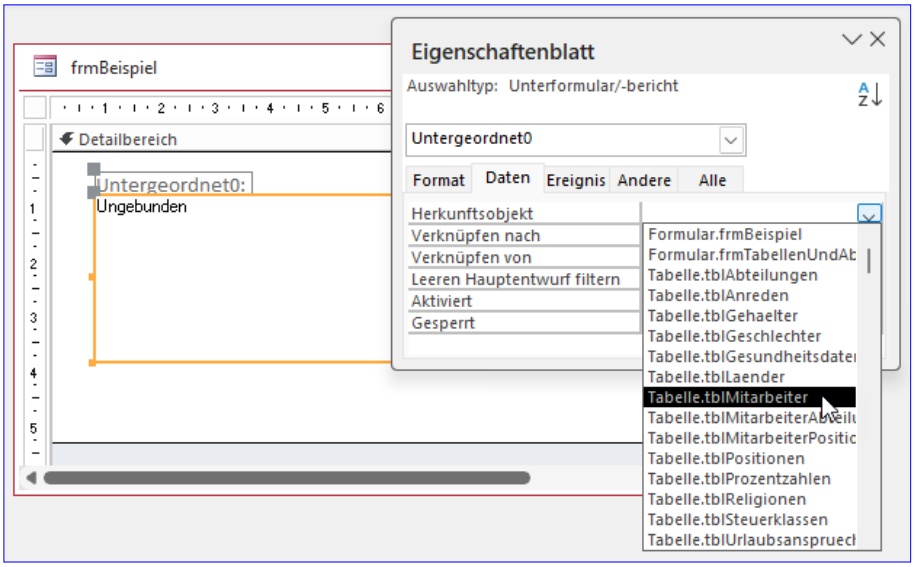

Bild 1: Klassische Auswahl der Tabelle oder Abfrage für ein Unterformular

Wählen wir hier beispielsweise den Eintrag **Tabelle.tblMitarbeiter** aus, sehen wir den Namen dieses Eintrags anschließend auch im Unterformular-Steuerelement (siehe Bild 2).

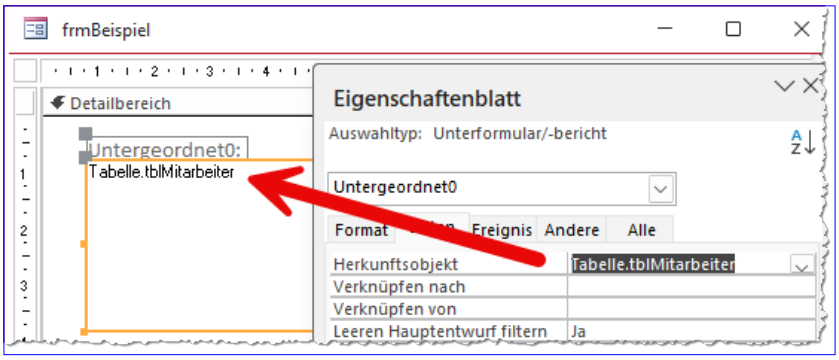

Bild 2: Das gewählte Objekt wird im Unterformular angezeigt.

Im Gegensatz zur Verwendung eines Formulars als Unterformular, das nach dem Schließen und erneutem Öffnen des Formulars in der Entwurfsansicht angezeigt wird, sehen wir bei Verwendung einer Tabelle oder Abfrage nach wie vor nur die Bezeichnung.

Wenn wir jedoch in die Formularansicht wechseln, erscheint die Datenblattansicht der gewählten Tabelle oder Abfrage im Unterformular (siehe Bild 3).

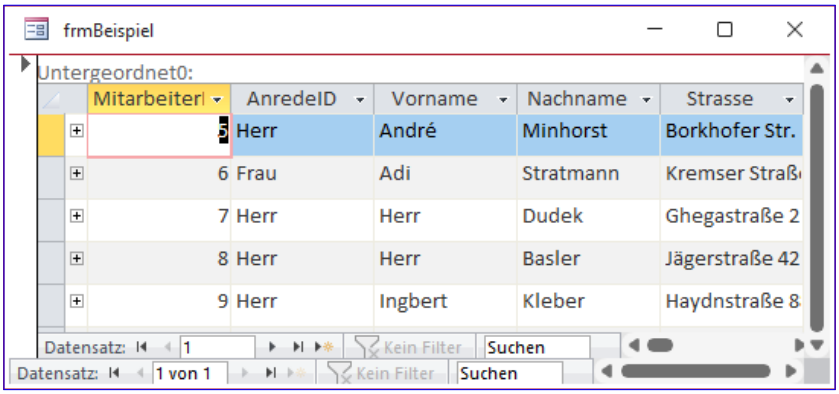

Bild 3: Eine Tabelle im Unterformular

Tabelle oder Abfrage komfortabel auswählen

Wir erstellen nun ein neues Formular, dem wir zusätzlich zu dem Unterformular-Steuerelement namens **sfm** noch ein Kombinationsfeld namens **cboTabellenUndAbfragen** hinzufügen (siehe Bild 4).

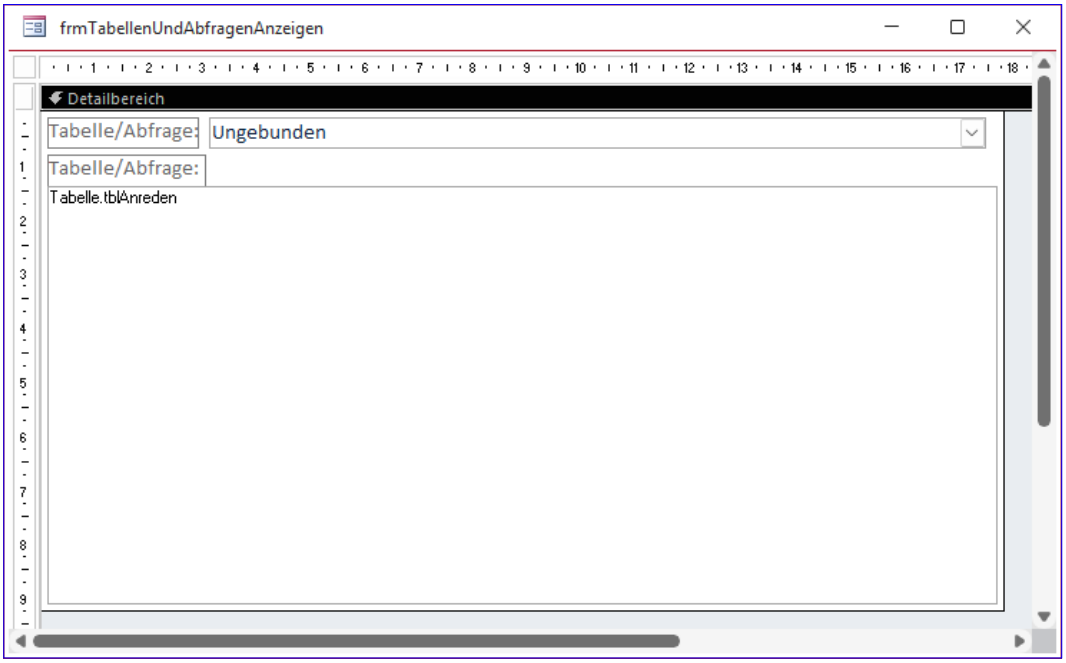

Bild 4: Aufbau unseres Beispielformulars

Dieses Kombinationsfeld soll alle Tabellen und Abfragen der Datenbank zur Auswahl anbieten, die wir auch im Navigationsbereich von Access sehen können.

Abfrage mit allen Tabellen und Abfragen

Dazu erstellen wir eine Abfrage als Datensatzherkunft des Kombinationsfeldes, welche die Systemtabelle **MSysObjekts** als Quelle verwendet (siehe Bild 5).

Um diese Tabelle in den Abfrageentwurf ziehen zu können, müssen wir in den Optionen des Navigationsbereichs die Option **Systemobjekte anzeigen** aktivieren.

Danach fügen wir diese Tabelle zum Abfrageentwurf hinzu und ziehen die beiden Felder **Name** und **Type** in das Entwurfsraster.

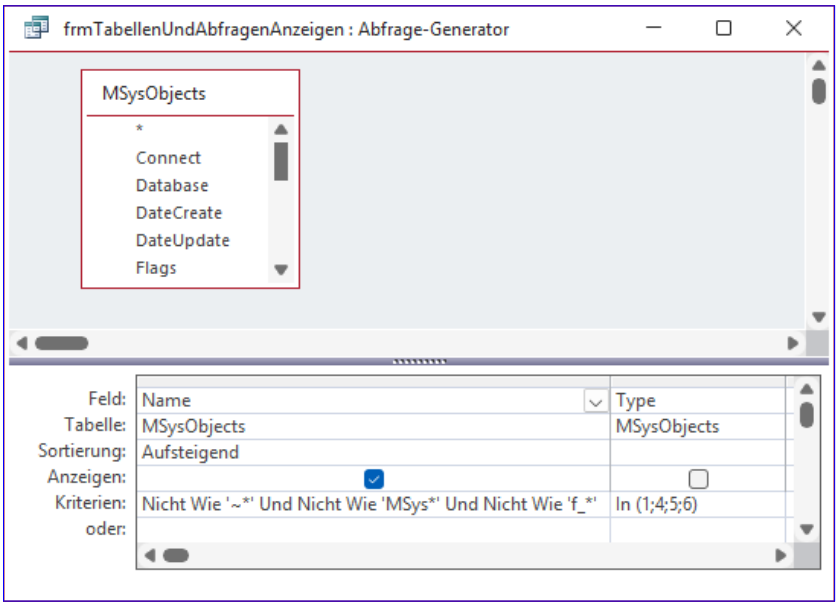

Bild 5: Abfrage, die alle Tabellen und Abfragen liefert

Das Feld **Type** verwenden wir, um nur die Tabellen und Abfragen der Datenbank anzuzeigen. Tabellen haben einen der Typen **1**, **4** oder **6** (für lokale Tabellen, ODBC-Tabellen oder andere verknüpfte Datenquellen), Abfragen werden dem Typ **5** zugeordnet.

Aber auch die Tabellen und Abfragen müssen wir noch weiter filtern, um nur die relevanten Objekte zu erhalten. Wir möchten zum Beispiel keine Systemtabellen sehen. Außerdem sollen keine temporären Objekte sichtbar sein. Dazu stellen wir in der Zeile **Kriterium** für das Feld **Name** den folgenden Ausdruck ein:

```
Nicht Wie '~*' Und Nicht
Wie 'MSys*' Und Nicht Wie
'f_*'
```

Schließlich sortieren wir das Feld **Name** noch in aufsteigender Reihenfolge.

Nach dem Wechsel in die Formularansicht zeigt das Kombinationsfeld nun alle gewünschten Elemente der Datenbank an (siehe Bild 6).

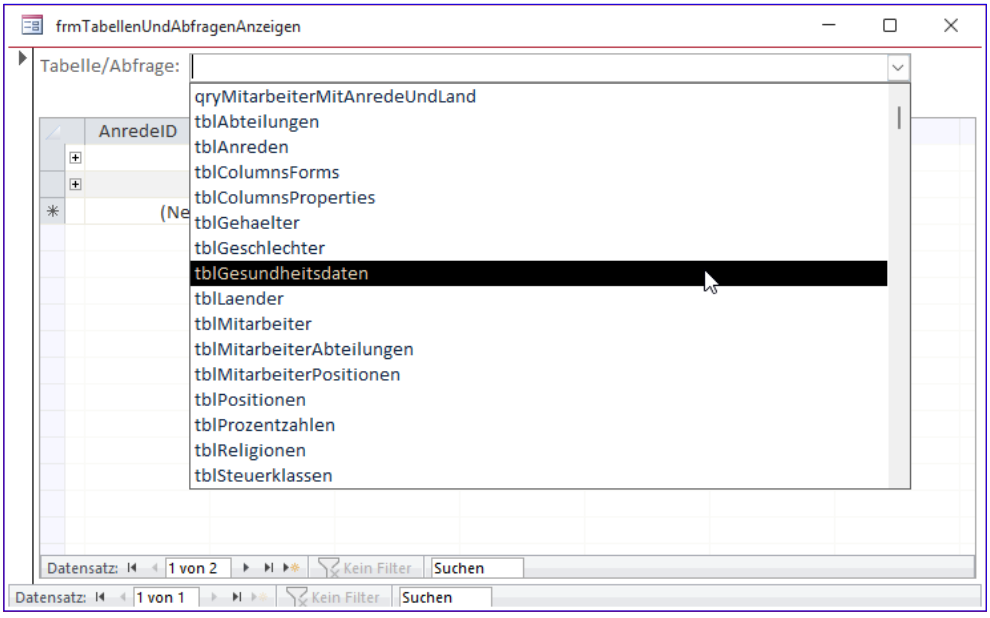

Bild 6: Das Kombinationsfeld zeigt nun alle gewünschten Tabellen und Abfragen an.

Icons in Ribbon, Kontextmenü, Formular und TreeView

Seit Jahren gibt es im Web einige Module, die das Handling von Bilddateien als Icons in Ribbons, Kontextmenüs, TreeViews oder als Formular-Icon erlauben. Diese sind meist umfangreich und für Laien kaum verständlich. Wir haben die verwendeten Techniken einmal ein wenig einfacher gestaltet, sodass man nicht mehr so viel Code benötigt – und vor allem weniger API-Funktionen. In diesem Artikel lernen Sie, wie Sie Ribbon-Einträge und Kontextmenü-Befehle mit Icons versehen. Als Bonus gibt es eine hilfreiche Funktion, mit der wir schnell viele Icons in die Access-Datenbank übertragen können, statt diese Schritt-für-Schritt einzulesen.

Theorie zu Bilddateien für Icons und Ribbons

Seit der Version 2010 speichert Microsoft Access Bilder, die aus dem Dateisystem ausgewählt und Steuerelementen wie dem Bildsteuerelement oder der Schaltfläche zugeordnet werden, in einer Systemtabelle der Datenbank namens **MSysResources**.

Nachdem wir einem Formular im Entwurf über den Ribbonbefehl **Bild einfügen** ein Bild hinzugefügt haben, ist dieses anschließend ständig verfügbar (siehe Bild 1).

Die Bilddateien werden intern versteckt gespeichert, nämlich in der bereits erwähnten Tabelle **MSysResources**.

Diese sieht nach dem Hinzufügen einiger Bilder wie in Bild 2 aus.

All diese Bilder können wir allerdings auch anderweitig nutzen, wenn sie sich einmal in der Datenbank befinden – zum Beispiel, indem wir sie per VBA auslesen und in Ribbons oder Kontextmenüs anzeigen.

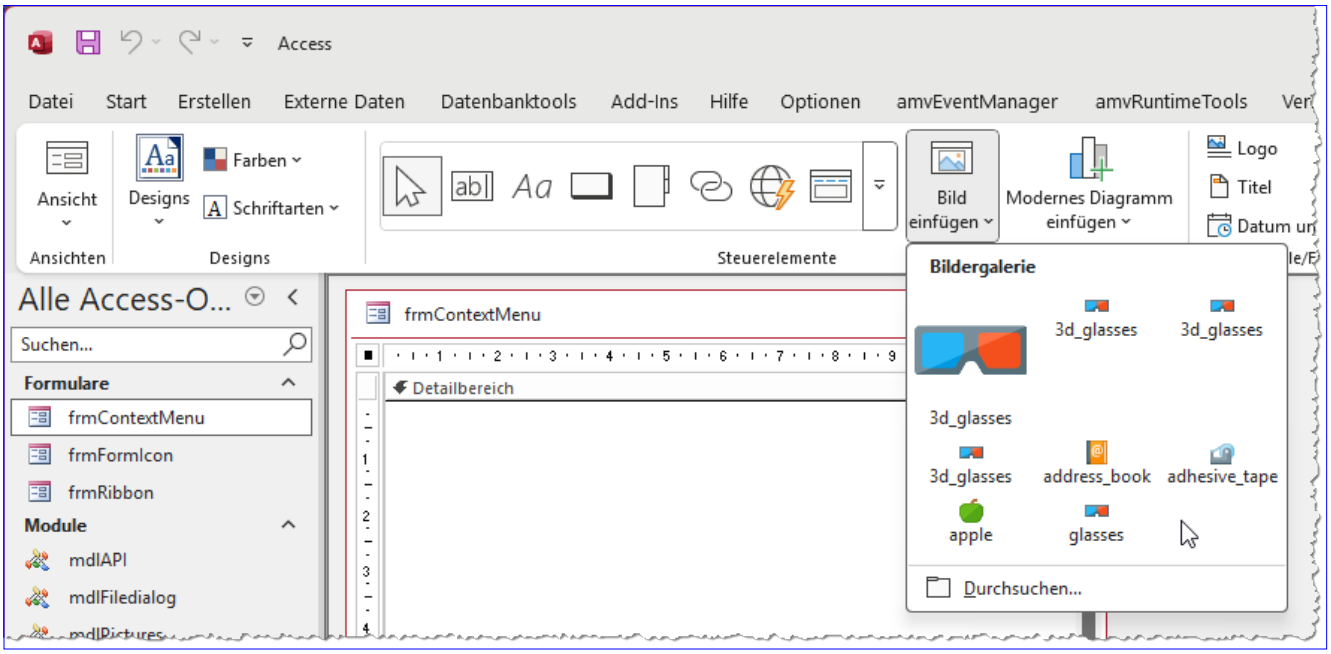

Bild 1: Bilder sind nach dem Hinzufügen ständig in der aktuellen Datenbank verfügbar.

Vorbereitung: Icons in der Datenbank speichern

Der Dialog zum Hinzufügen von Bildern zur Datenbank über den Befehl **Bild einfügen** hat einen Nachteil, der bereits aus der Beschriftung dieses Befehls ersichtlich wird – man kann immer nur ein Bild pro Vorgang hinzufügen. Wenn man einen Satz von Bildern hat, die man immer wieder benötigt, wird das schnell aufwendig.

Wir stellen also in diesem Beitrag eine Funktion vor, mit der wir gleich mehrere Bilder im Dateidialog auswählen und zur Datenbank hinzufügen können.

MSysResources

	Extension	Id	Name	Type
(1)	thmx	1	Office Theme	thmx
(1)	png	2	apple	img
(1)	png	3	3d_glasses	img
(1)	ico	4	test	icon
(1)	png	7	glasses	img
(1)	png	8	3d_glasses	img
(1)	png	9	3d_glasses	img
(1)	png	10	3d_glasses	img
(1)	png	11	address_book	img
(1)	png	12	adhesive_tape	img
(0)		(Neu)		

Datensatz: 1 von 10 Kein Filter Suchen

Bild 2: Tabelle zum Speichern der Bilder in der aktuellen Datenbank

Bilder in Ribbon und Kontextmenü

Während wir die Bilder aus der Tabelle **MSysResources** zu Bildsteuerelementen oder Schaltflächen hinzufügen können, indem wir diese einfach für die Eigenschaft **Bild** auswählen (siehe Bild 3), ist das bei den Schaltflächen und anderen Elementen in Schaltflächen im Kontextmenüs nicht so einfach möglich. Diese erwarten ein per VBA zugewiesenes Element des Typs **StdPicture**.

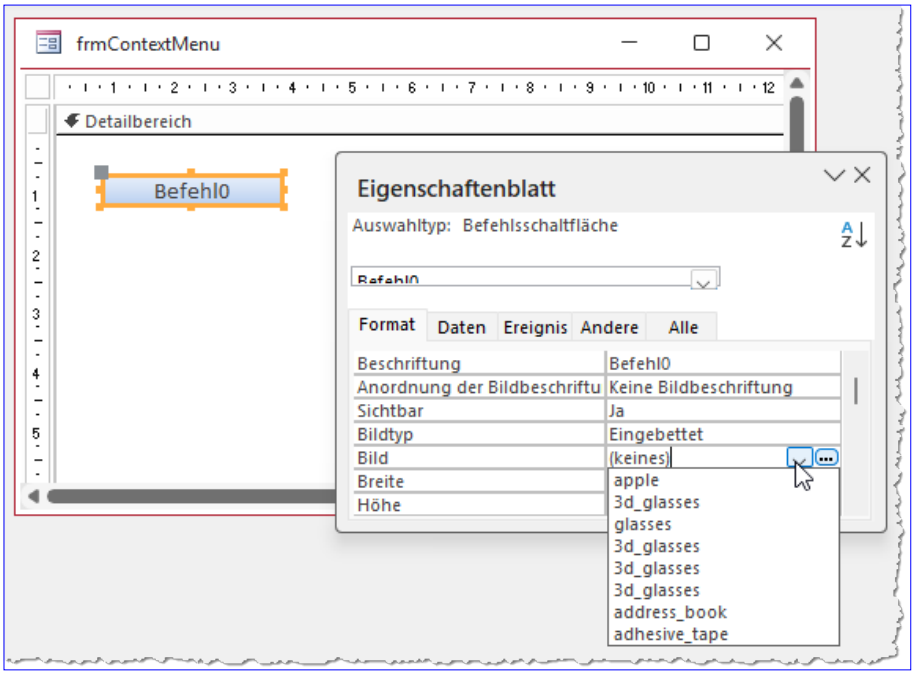

Bild 3: Die Bilder stehen in der Bild-Eigenschaft zur Verfügung

Ein Bild aus der Tabelle **MSysResources** auszuwählen und in ein **StdPicture**-Element umzuwandeln ist kein Hexenwerk, aber dazu sind einige Codezeilen nötig und eine Reihe API-Funktionen. Im zweiten Abschnitt dieses Beitrags werden wir eine Vorgehensweise vorstellen, die wesentlich kompakter als die üblicherweise verwendeten und auch in diesem Magazin eingeführten Techniken sind.

Außerdem zeigen wir, wie diese Elemente dann den entsprechenden Steuerelementen zugewiesen werden.

Schnelles Einlesen von vielen Icons

Zuerst schauen wir uns die Funktion zum schnellen Einlesen von mehreren Bilddateien an.

Diese erfordert zunächst eine Funktion, mit der wir die einzulesenden Dateien per Dateidialog auswählen können. Außerdem benötigen wir eine Routine, welche die ausgewählten Dateien in das Anlagefeld der Tabelle **MSysResources** schreibt. Diese beiden Routinen führen wir in

einer aufrufenden Prozedur namens **amvFileToResources** zusammen.

Die Funktion zum Anzeigen des Dateidialogs basiert auf der **FileDialog**-Klasse der Office-Bibliothek, auf die wir noch einen Verweis setzen müssen (siehe Bild 4).

Diese benötigen wir auch später noch, wenn wir ein Kontextmenü mit Icons erstellen wollen.

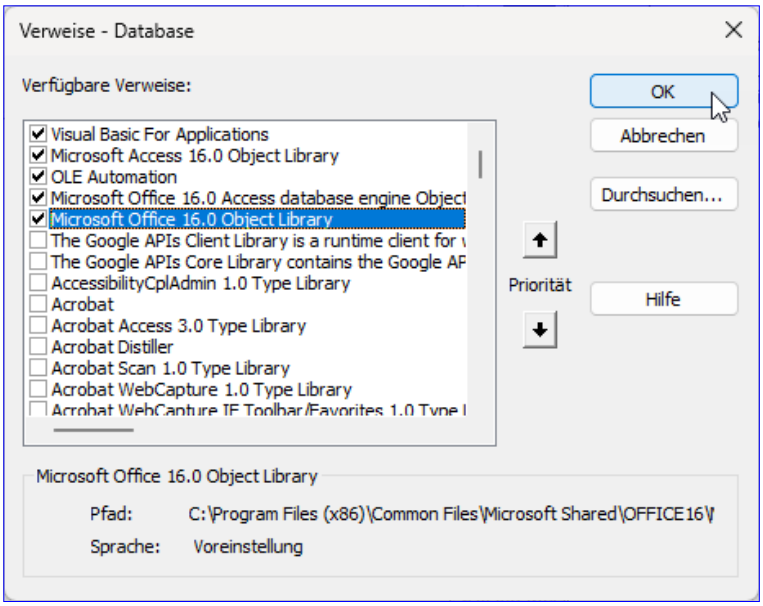

Bild 4: Hinzufügen des Verweises auf die Office-Bibliothek

Steuerung des Einlesevorgangs der Bilddateien

Die Prozedur **amvFileToResources** steuert unseren Vorgang:

```
Public Sub amvFilesToResources()
    Dim strPaths() As String
    Dim strPathlist As String
    Dim i As Integer

    strPathlist = ChooseFiles
    strPaths() = Split(strPathlist, "|")
    For i = LBound(strPaths) To UBound(strPaths)
        Debug.Print strPaths(i)
        amvAddImageToMSysResources strPaths(i)
    Next i
End Sub
```

Sie ruft zunächst die Funktion **amvChooseFiles** auf, die eine Zeichenkette mit den durch das Pipe-Zeichen (|) getrennten einzelnen Pfaden der einzulesenden Dateien zurückgibt.

Diese Liste speichern wir in der Variablen **strPathlist**. Den Inhalt dieser Variablen teilen wir mit der **Split**-Funktion in ein Array auf, das wir in der Variablen **strPaths()** speichern.

Die Elemente dieses Arrays durchlaufen wir nachfolgend in einer **For...Next**-Schleife. In dieser Schleife rufen wir für jedes Element einmal die Funktion **amvAddImageToMSysResources** auf, um die jeweilige Datei in einen neuen Datensatz der Tabelle **MSysResources** zu schreiben.

Die Funktion amvChooseFiles

Die Funktion zum Auswählen der Dateien nutzt den Dateiauswahldialog der Office-Bibliothek, um dem Benutzer eine flexible Mehrfachauswahl von Dateien zu ermöglichen, und gibt die Pfade der ausgewählten Dateien als durch das Pipe-Zeichen getrennte Zeichenkette zurück (siehe Listing 1). Gleichzeitig merkt sich die Funktion den beim letzten Vorgang verwendeten Ordnerpfad und schlägt diesen beim nächsten Aufruf erneut vor, was eine erhebliche Erleichterung bei wiederholter Arbeit mit denselben Ordnern darstellt.

Zu Beginn der Funktion werden die nötigen Variablen deklariert. Eine dieser Variablen speichert das Initialverzeichnis für den Dateiauswahldialog. Dieser Pfad wird mithilfe der Funktion amv**GetAppSetting** (siehe Modul **MDL_AMV_Registry**) aus dem Bereich der Registry für Anwendungseinstellungen ausgelesen. Ist dort kein Pfad hinterlegt, etwa weil die Funktion zum ersten Mal verwendet wird oder die Einstellung gelöscht wurde, verwendet die Funktion standardmäßig das Verzeichnis der aktu-

```
Public Function amvChooseFiles()
    Dim objFileDialog As Office.FileDialog
    Dim strTemp As String
    Dim varFilename As Variant
    Dim strInitialFilename As String
    strInitialFilename = amvGetAppSetting("InitialFilename")
    If Len(strInitialFilename) = 0 Then
        strInitialFilename = CurrentProject.Path & "\"
    End If
    Set objFileDialog = Application.FileDialog(msoFileDialogOpen)
    With objFileDialog
        .Title = "Bilddateien auswählen"
        .ButtonName = "Auswählen"
        .AllowMultiSelect = True
        .InitialFileName = strInitialFilename
        .Filters.Clear
        .Filters.Add "PNG-Dateien", "*.png"
        .Filters.Add "ICO-Dateien (.ico)", "*.ico"
        If .Show = True Then
            If Not .SelectedItems.Count = 0 Then
                strInitialFilename = .SelectedItems(1)
                strInitialFilename = Left(strInitialFilename, InStrRev(strInitialFilename, "\"))
                For Each varFilename In .SelectedItems
                    strTemp = strTemp & "|" & varFilename
                Next varFilename
                If Not Len(strTemp) = 0 Then
                    strTemp = Mid(strTemp, 2)
                End If
            End If
        End If
    End With
    amvSaveAppSetting "InitialFilename", strInitialFilename
    amvChooseFiles = strTemp
End Function
```

Listing 1: Ermitteln der einzulesenden Bilddateien per Dateidialog

ellen Datenbankdatei. Das bedeutet, dass der Benutzer beim ersten Aufruf im Ordner der Datenbank zu arbeiten beginnt, bei weiteren Aufrufen jedoch direkt im zuletzt verwendeten Verzeichnis landet.

Anschließend wird ein Dialogobjekt vom Typ **FileDialog** mit dem Typ **msoFileDialogOpen** zum Auswählen von Dateien erzeugt. Der Dialog wird so konfiguriert, dass mehrere Dateien auf einmal ausgewählt werden können. Er erhält einen beschreibenden Titel, einen angepassten Beschriftungstext für die Auswahlschaltfläche sowie zwei Filtereinträge: einen für PNG-Dateien und einen für ICO-Dateien.

Zeigt der Benutzer nun den Dialog an und bestätigt die Auswahl, wird überprüft, ob überhaupt Dateien ausgewählt wurden. Ist dies der Fall, wird aus dem Pfad der ersten Datei der zugehörige Ordner extrahiert. Dieser

Ordner wird später als neue Anwendungseinstellung gespeichert, damit er beim nächsten Aufruf der Funktion wieder als Ausgangspunkt dient. Danach werden alle ausgewählten Dateipfade nacheinander in einer Zeichenkette gesammelt. Zwischen den einzelnen Dateipfaden wird das Pipe-Zeichen als Trenner eingefügt. Abschließend wird der führende Trenner entfernt, sodass die Rückgabe sauber formatiert ist.

Die Funktion speichert dann den aktualisierten Startordner mithilfe der Prozedur **amvSaveAppSetting**, ebenfalls im Modul **MDL_AMV_Registry** zu finden, und gibt schließlich die Zeichenkette mit den Pfaden der ausgewählten Dateien zurück.

Prozedur zum Speichern der Bilder in der Tabelle MSysResources

Nach der Auswahl folgt die Prozedur **amvAddImageToMSysResources**. Diese deklariert ein **Database**-Objekt, zwei Recordsets für die Tabelle **MSysResources** und die darin enthaltene interne Tabelle zum Speichern der Attachments und ein **Field2**-Objekt zum Laden der Dateien in das Zielfeld (siehe Listing 2).

Die Prozedur erwartet den Pfad der einzulesenden Datei als Parameter. Sie ermittelt zunächst die Dateiendung und den Dateinamen sowie den Dateinamen ohne Dateiendung und trägt diese Daten in die Variablen **strExtension**, **strFilename** und **strFilenameWithoutExtensions** ein.

Dann erstellt sie eine Referenz zum aktuellen **Database**-Objekt und öffnet darüber die Tabelle **MSysResources** als Recordset.

Hier fügt sie mit **AddNew** einen neuen Datensatz hinzu und trägt die Dateiendung in das Feld **Extension** sowie den Dateinamen ohne Dateiendung in das Feld **Name** ein. Als Type wird immer **img** eingetragen.

Nun folgt der spannende Teil. Das zweite Recordset wird mit dem Wert **Value** des Feldes **Data** der Tabelle **MSysResources** gefüllt, also mit der internen Tabelle zum Speichern der Attachments.

Auch hier fügen wir mit **AddNew** einen neuen Datensatz ein und referenzieren dann das Feld **FileData** mit der Variablen **fldFileData**. Die hier verwendete **Field2**-Klasse liefert gegenüber der herkömmlichen Field-Klasse noch die Methode **LoadFromFile**, die das Einlesen einer Datei in das Feld ermöglicht. Diese nutzen wir, um die angegebene Datei in das Anlagefeld zu laden. Zusätzlich stellen wir die Eigenschaft **Filename** auf den Dateinamen ein, diesmal mit Dateierweiterung, und speichern den neu hinzugefügten Datensatz der internen Tabelle mit der **Update**-Methode.

Die gleiche Methode rufen wir auch noch für das übergeordnete Recordset auf und schließen dann die geöffneten Objekte und geben die Objektvariablen frei.

Vorbereiten der Tabelle MSysResources mit einigen Bilddateien

Rufen wir die Prozedur **amvFileToResources** nun auf, erscheint der Dateidialog, mit dem wir die gewünschten Bilddateien auswählen können. Die ausgewählten Dateien werden ohne Rücksicht auf eventuelle Dubletten in die Tabelle **MSysResources** geschrieben.

Nachdem wir dies erledigt haben, sind einige Dateien in dieser Tabelle gelandet, die wir für die folgenden Techniken als Icons für das Ribbon und für Kontextmenüs nutzen können.

Bilder im Ribbon anzeigen

Wenn wir Bilder im Ribbon anzeigen wollen, benötigen wir zweierlei:

- Ein Element im Ribbon, dessen **image**-Attribut mit dem Namen des zu verwendenden Bildes aus der Tabelle **MSysResources** gefüllt ist. Dabei verwenden wir den Namen ohne Dateiendung, so wie er im Feld **Name** der Tabelle eingetragen ist.

```
Public Sub amvAddImageToMSysResources(ByVal strPath As String)
    Dim db As DAO.Database
    Dim rstResources As DAO.Recordset
    Dim rstAttachments As DAO.Recordset2
    Dim fldFileData As DAO.Field2
    Dim strExtension As String
    Dim strFilename As String
    Dim strFilenameWithoutExtension As String

    strExtension = Mid(strPath, InStrRev(strPath, ".") + 1)
    strFilename = Mid(strPath, InStrRev(strPath, "\") + 1)
    strFilenameWithoutExtension = Left(strFilename, InStrRev(strFilename, ".") - 1)
    Set db = CurrentDb
    Set rstResources = db.OpenRecordset("SELECT * FROM MSysResources", dbOpenDynaset)
    rstResources.AddNew

    rstResources!Extension = strExtension
    rstResources!Name = strFilenameWithoutExtension
    rstResources!Type = "img"

    Set rstAttachments = rstResources.Fields("Data").Value
    rstAttachments.AddNew
    Set fldFileData = rstAttachments.Fields("FileData")
    fldFileData.LoadFromFile strPath
    rstAttachments.FileName = strFilename
    rstAttachments.Update
    rstAttachments.Close
    Set rstAttachments = Nothing

    rstResources.Update
    rstResources.Close
    Set rstResources = Nothing
    Set db = Nothing
End Sub
```

Listing 2: Die Prozedur **amvAddImageToMSysResources**

- Außerdem benötigen wir im Element **customUI** das Attribut **loadImages**, das den Namen der Funktion enthält, das beim Anzeigen der jeweiligen Elemente zum Einlesen der Bilder genutzt werden soll.

Eine der einfachsten möglichen Ribbondefinitionen für diesen Zweck haben wir bereits in die für diesen Fall vorbereitete Tabelle **USysRibbons** eingetragen (siehe Bild 5). Das Attribut des **button**-Elements enthält den Wert **apple**, was dazu führt, dass ein Bild namens **apple** angezeigt werden soll. Das Attribut **loadImage** ist mit dem Wert **loadImage** ausgestattet, was bedeutet, dass wir eine gleichnamige Funktion zum Einlesen der Bilder hinterlegen müssen.

Funktionen für die Automation von Ribbons hinterlegen wir traditionell in einem Modul namens **mdlRibbons**. In diesem Fall fügen wir dort die folgende Funktion ein:

USysRibbons

ID	RibbonName	RibbonXML
37	Einfache Buttons	<customUI xmlns="http://schemas.microsoft.com/office/2009/07/customui" loadImage="loadImage"> <ribbon startFromScratch="true"> <tabs> <tab id="tab" label="Tab"> <group id="grp" label="Gruppe"> <button id="btn" image="apple" label="Beispielbutton mit Bild" /> </group> </tab> </tabs> </ribbon> </customUI>

Bild 5: Einfachste mögliche Ribbondefinition zum Anzeigen eines benutzerdefinierten Bildes

```
Public Sub loadImage(control, ByRef image)
    Set image = amvGetImageFromResourcesByName(control)
End Sub
```

Sie ruft lediglich eine weitere Funktion namens **amvGetImageFromResourcesByName** auf, die den Wert des Parameters **control** übergeben bekommt. Diese Funktion liefert das Bild in Form einer Variable des Typs **StdPicture** zurück, die schließlich an die Ribbondefinition zurückgegeben werden kann.

Im Gegensatz zu Prozeduren, die zum Beispiel beim Anklicken eines Ribbonbuttons ausgelöst werden, liefert **control** hier kein Objekt des Typs **IRibbonControl**, sondern lediglich den Wert des Attributs **image** des jeweils mit einer Bilddatei zu füllenden Steuerelements.

```
Public Function amvGetImageFromResourcesByName(strName As Variant) As StdPicture
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim lngID As Long

    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT ID FROM MSysResources WHERE Name = '" & strName & "'", dbOpenSnapshot)

    Select Case rst.RecordCount
        Case 1
            lngID = rst.Fields(0)
            Set amvGetImageFromResourcesByName = amvGetImageFromResources(lngID)
        Case 0
            MsgBox "Bild '" & strName & "' nicht in der Tabelle MSysResources gefunden."
        Case Else
            MsgBox "Mehr als ein Bild namens '" & strName & "' in der Tabelle MSysResources vorhanden."
    End Select
End Function
```

Listing 3: Die Funktion **amvGetImageFromResources**

Besserer »Suchen und Ersetzen«-Dialog: Grundgerüst

Der Dialog zum Suchen und Ersetzen für die Suche nach Daten in der Datenblattansicht ist bereits sehr praktisch. Allerdings hatte ein Leser die Anforderung, dass er öfter mit dem Dialog nach den gleichen Ausdrücken sucht, um diese zu ersetzen. Der Dialog hat zwar immer die zuletzt verwendeten zu suchenden und zu ersetzenden Ausdrücke gespeichert und in den Auswahlfeldern angeboten, aber wenn er die Datenbank geschlossen und erneut geöffnet hat, war die Arbeit verloren und die Auswahlfelder waren wieder leer. Also bauen wir diesen Dialog einfach nach und erweitern diesen um eine Funktion, welche die zuletzt verwendeten Einträge dauerhaft speichert und diese nach dem Schließen und Öffnen der Anwendung wieder bereitstellt.

Worum geht es?

Der Suchen und Ersetzen-Dialog, den wir durch einen besseren Dialog ersetzen wollen, kann über den Ribbon-Eintrag **Start|Suchen|Suchen** geöffnet werden (siehe Bild 1).

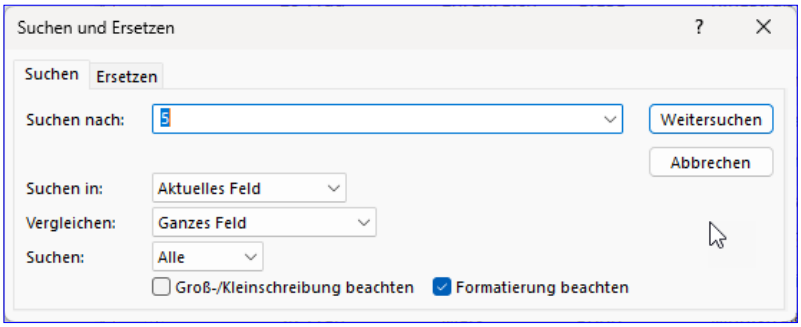

Bild 2: Der **Suchen und Ersetzen**-Dialog

Dies öffnet den Dialog mit aktiviertem **Suchen**-Registerreiter (siehe Bild 2). Hier können wir nacheinander nach verschiedenen Begriffen suchen. Die zuletzt verwendeten Begriffe werden beim Aufklappen der Auswahl unter **Suchen nach:** direkt bereitgestellt.

Wir können direkt von diesem Dialog aus zur Registerseite **Ersetzen** wechseln. Diese enthält lediglich einige zusätzliche Elemente wie zum Beispiel das Feld zum Eingeben des zu ersetzenden Ausdrucks (siehe Bild 3). Außerdem finden wir nun Schaltflächen zum Ersetzen der aktuellen Fundstelle und aller Fundstellen.

Der Dialog kann außerdem mit der Taste **Strg + F** geöffnet werden.

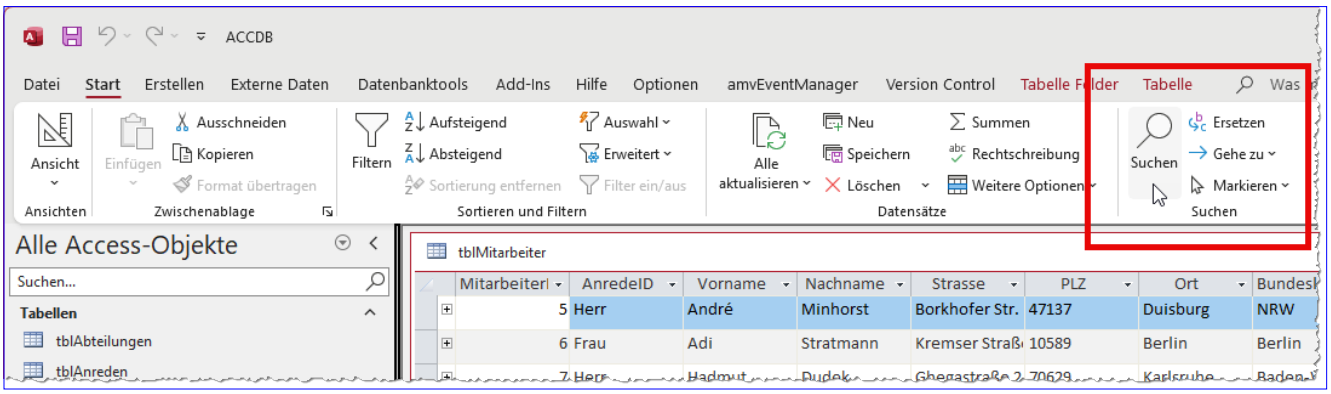

Bild 1: Aufruf des **Suchen und Ersetzen**-Dialogs

Option »Suchen in«

Der Dialog erlaubt mit **Suchen in** das Festlegen des Suchziels:

- **Aktuelles Feld**: Durchsucht nur das aktuelle Steuerelement. Bei Datenblatt- oder Endlosansichten bezieht sich das auf das gleiche Steuerelement für jeden Datensatz.

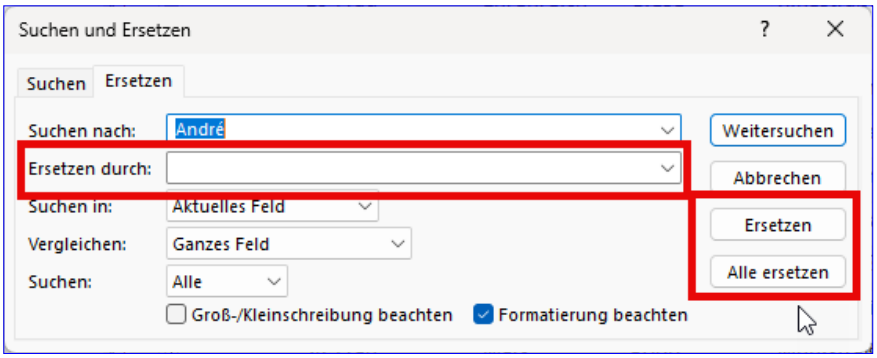

Bild 3: Registerseite **Ersetzen** des **Suchen und Ersetzen**-Dialogs

- **Aktuelles Dokument**: Durchsucht alle Steuerelemente und Datensätze der aktuellen Ansicht. Dabei wird zuerst Steuerelement für Steuerelement des aktuellen Datensatzes durchsucht und dann in gleicher Weise die folgenden Datensätze.

Option »Vergleichen«

Mit **Vergleichen** können wir angeben, welcher Teil der Felder untersucht werden soll: **Teil des Feldinhalts**, **Ganzes Feld** oder **Anfang des Feldinhalts**.

Option »Suchen«

Unter **Suchen** geben wir an, welchen Bereich wir ausgehend vom aktuell markierten Steuerelement untersuchen wollen:

- **Alle**: Durchsucht von oben nach unten und beginnt von vorn, wenn sich noch Elemente vor dem aktuellen Steuerelement befinden.
- **Aufwärts**: Sucht von der aktuellen Position bis zum Start des Dokuments.
- **Abwärts**: Sucht von der aktuellen Position bis zum Ende des Dokuments.

Option »Groß-/Kleinschreibung«

Mit **Groß-/Kleinschreibung** können wir angeben, ob wir Groß- und Kleinschreibung berücksichtigen wollen.

Option »Formatierung beachten«

Schließlich gibt es noch die Option **Formatierung beachten**. Diese ist zum Beispiel bei Datumsfeldern wichtig, wenn man eine andere Formatierung angegeben hat als das System verwendet. Wenn wir also Datumsangaben eigentlich im Format **tt.mm.jjjj** speichern (zum Beispiel **21.06.2025**), aber für das Tabellenfeld das Format **tt-mm-jjjj** festgelegt haben,

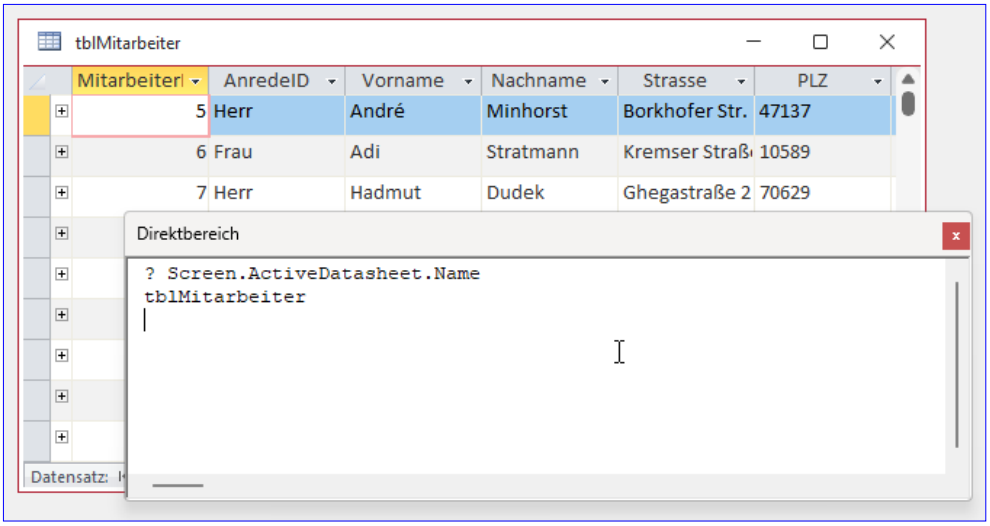

Bild 4: Zugriff per VBA auf das aktuelle Datenblatt einer Tabelle

wird das Datum als **21-06-2025** angezeigt.

Eine Suche nach **21.06.** liefert nun in jedem Fall das gesuchte Datum. Wenn wir jedoch nach **21-06-** suchen und **Formatierung beachten** ist deaktiviert, finden wir das Datum nicht.

Erst wenn wir Formatierung beachten aktivieren, liefert auch **21-06-** das gewünschte Ergebnis.

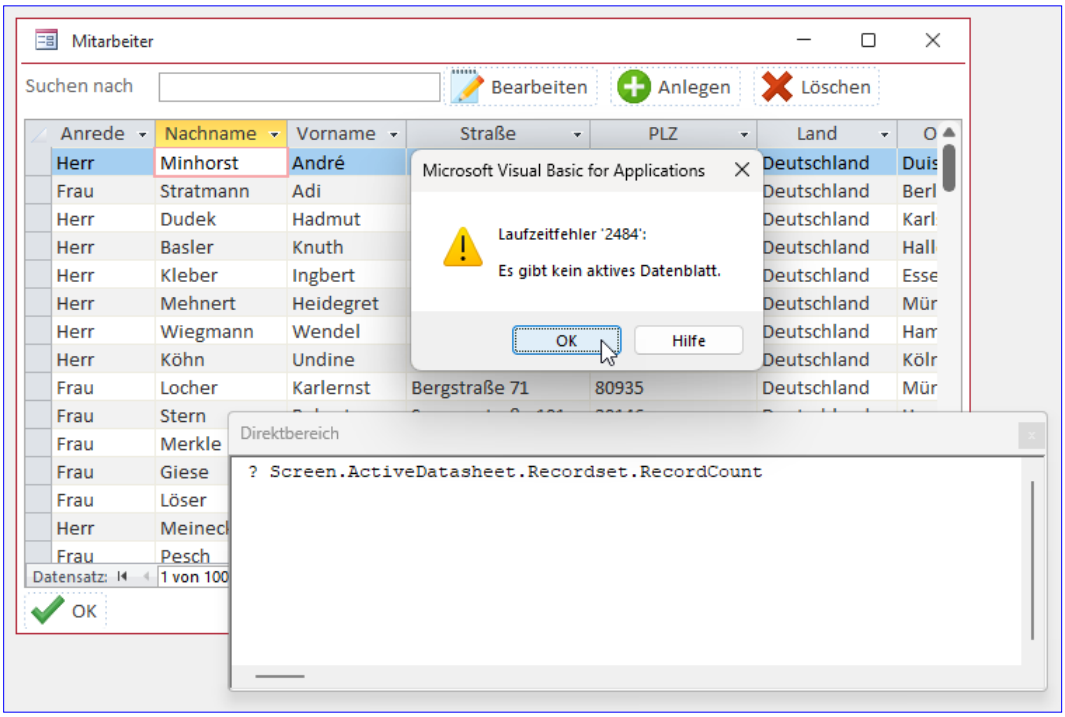

Bild 5: Fehler beim Zugriff per VBA auf das aktuelle Datenblatt im Unterformular

Schaltfläche »Weitersuchen«

Mit der Schaltfläche **Weitersuchen** starten wir den Suchvorgang nach dem nächsten Vorkommen. Klicken wir diese wiederholt an, durchläuft Access alle Fundstellen.

Schaltfläche »Ersetzen«

Diese Schaltfläche funktioniert nur, wenn zuvor mit **Weitersuchen** eine Fundstelle markiert wurde. Das Anklicken führt dann zur gewünschten Ersetzung und markiert automatisch die nächste Fundstelle, sofern noch ein weiteres Vorkommen vorhanden ist.

So können wir durch wiederholtes Betätigen dieser Schaltfläche alle Vorkommen des gesuchten Begriffes ersetzen.

Schaltfläche »Alle ersetzen«

Diese Schaltfläche ersetzt automatisch alle Vorkommen im angegebenen Bereich. Dazu muss nicht zuvor eine Fundstelle markiert worden sein.

Wichtig: Lassen sich die grundlegenden Funktionen reproduzieren?

Bevor wir Zeit investieren, den **Suchen und Ersetzen**-Dialog original nachzubilden und diesen um die gewünschten Funktionen zu erweitern, wollen wir erst einmal prüfen, ob sich das grundlegende Verhalten überhaupt reproduzieren lässt.

Um per VBA-Code zum Beispiel nach bestimmten Inhalten im Datenblatt zu suchen, müssen wir das Recordset des Datenblatts referenzieren. Die Frage lautet: Gelingt das überhaupt per Code? Haben jegliche Datenblätter ein Recordset, auf das wir zugreifen können – egal, ob es die Datenblattansicht einer Tabelle, einer Abfrage oder auch eines Formulars ist? Beim Formular wissen wir, dass wir auf das Recordset der Datenblattansicht zugreifen können.

Hier referenzieren wir das Formular über die **Forms**-Auflistung und greifen dann über die **Recordset**-Eigenschaft auf das Recordset zu. Aber wie soll das mit Tabellen und

Abfragen funktionieren? Immerhin gibt es keine **Table**- oder **Query**-Auflistungen, die wir analog zur **Forms**-Auflistung nutzen könnten.

Aber benötigen wir überhaupt eine Auflistung, um auf das aktuelle Objekt und sein Recordset zugreifen zu können?

Schnelle Experimente zeigen: Nein, das ist nicht der Fall. Wie können nämlich die Elemente der **Screen**-Klasse nutzen.

Per Screen-Klasse auf das aktuelle Datenblatt zugreifen

Wir schauen uns dies zuerst anhand einer Tabelle an, die wir in der Datenblattansicht geöffnet haben.

Öffnen wir parallel dazu den Direktbereich des VBA-Editors und geben dort den Ausdruck **Screen.ActiveDatasheet.Name** ein, erhalten wir tatsächlich den Namen des aktuellen Datenblatts respektive der zugrunde liegenden Tabelle (siehe Bild 4).

Also probieren wir auch gleich aus, ob wir damit auch auf ein entsprechendes **Recordset**-Objekt zugreifen können. Auch das gelingt:

```
? Screen.ActiveDatasheet.Recordset.RecordCount
 100
```

Das Gleiche funktioniert auch, wenn wir eine Abfrage in der Datenblattansicht öffnen.

Zugriff auf das Datenblatt eines Formulars

Bei Formularen müssen wir differenzieren. Wir können hier Formulare vorfinden, die ihre Daten direkt in der Datenblattansicht anzeigen oder solche, die Unterformulare für die Datenblattansicht verwenden.

Bei Formularen, die ihre Daten direkt in der Datenblattansicht anzeigen, können wir ebenfalls mit **Screen.ActiveDatasheet** arbeiten.

Bei Formularen, die ein Unterformular in der Datenblattansicht anzeigen, gelingt dies nicht so einfach, wie Bild 5 zeigt. Hier erhalten wir eine Fehlermeldung.

Wir können auch nicht mit der folgenden Anweisung darauf zugreifen – hier erhalten wir lediglich das Hauptformular:

```
? Screen.ActiveForm.Name
frmMitarbeiterUebersicht
```

Wenn sich der Fokus jedoch in einem der Steuerelemente im Unterformular befindet, können wir am einfachsten über folgenden Ausdruck darauf zugreifen:

```
? Screen.ActiveControl.Parent.Name
sfmMitarbeiterUebersicht
```

Mehr als nur Datenblatt

Allerdings funktioniert die eingebaute Suchen- und Ersetzen-Funktion nicht nur in Datenblättern, sondern eigentlich in allen Formularen, die überhaupt mindestens ein Steuerelement enthalten, das durchsuchbar ist – also zum Beispiel ein Textfeld. Wir wollen in diesem Beispiel allerdings nur eine **Suchen und Ersetzen**-Funktion für Datenblätter erstellen.

Weitere Erweiterungsmöglichkeiten

Wenn wir schon dabei sind, den **Suchen und Ersetzen**-Dialog zu verbessern, können wir auch gleich noch weitere Ergänzungen hinzufügen. Zum Beispiel könnten wir den Dialog so programmieren, dass er beim Öffnen gleich den Wert im **Suchen**-Feld anzeigt, der im aktuellen Steuerelement mit dem Fokus enthalten ist – vorausgesetzt, dieses Feld enthält einen Wert beziehungsweise kann einen Wert enthalten.

Ersetzen von Werten

Beim Ersetzen von Werten benötigen wir ein wenig mehr Code. Wir müssen aus den Parametern des Dialogs eine kleine Prozedur erstellen, die eine **UPDATE**-Anweisung

zusammenbaut und ausführt. Dies schauen wir uns später im Detail an.

»Suchen und Ersetzen«-Formular nachbauen

Nachdem die technischen Voraussetzungen geklärt sind, bauen wir das **Suchen und Ersetzen**-Formular originalgetreu nach.

Dazu erstellen wir ein neues, leeres Formular und speichern es unter dem Namen **frmSuchenUndErsetzen**. Um es später vom eingebauten **Suchen**-Dialog unterscheiden zu können, tragen wir für Beschriftung den Text **Suchen und Ersetzen PRO** ein.

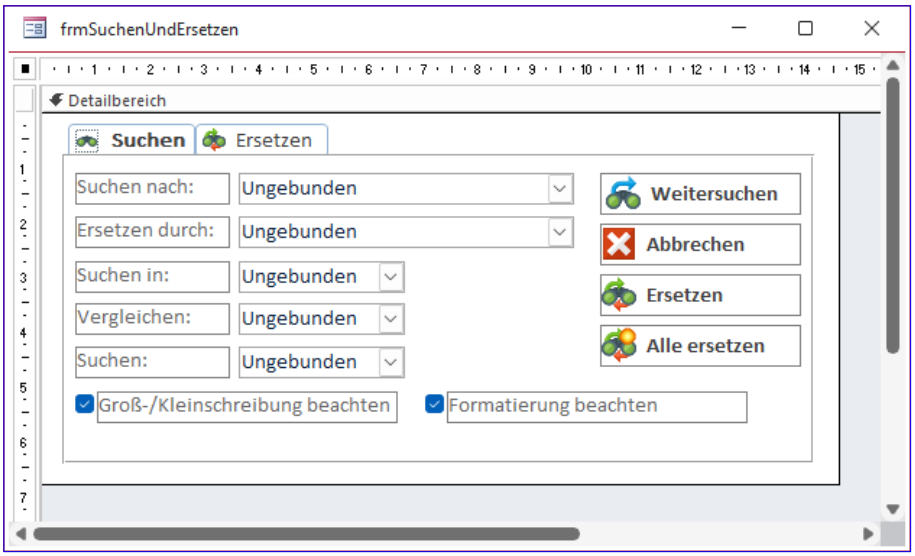

Bild 6: Entwurf des neuen **Suchen und Ersetzen**-Dialogs

Dann fügen wir erst einmal die Steuerelemente hinzu und legen ihre Namen fest.

Wichtig: Wir fügen kein echtes Registersteuerelement hinzu, denn eigentlich benötigen wir kein Registersteuerelement.

Wir haben nicht geprüft, ob Microsoft für den **Suchen und Ersetzen**-Dialog tatsächlich ein Registersteuerelement verwendet und ob dort die auf beiden Registerseiten vorkommenden Steuerelemente jeweils zwei Mal angelegt wurden. Wir werden das jedenfalls nicht tun und stattdessen ein Fake-Registersteuerelement bauen.

Dazu verwenden wir ein weißes Rechteck und zwei Schaltflächen, die wir über diesem Rechteck platzieren – genau so, dass der untere Rand der Schaltflächen etwas unter dem Rechteck verschwindet. Dazu müssen wir zuerst die Schaltflächen anlegen und dann das Rechteck-Steuerelement. In der Z-Reihenfolge werden immer die zuletzt angelegten Steuerelemente über die übrigen gelegt.

Dementsprechend fügen wir auch die übrigen Steuerelemente erst hinzu, wenn wir das Rechteck angelegt haben, und ordnen diese wie in Bild 6 an.

Die Steuerelemente versehen wir mit den folgenden Namen:

- Registertag **Suchen**: **regSuchen**
- Register-Schaltfläche **Ersetzen**: **regErsetzen**
- Kombinationsfeld **Suchen nach**: **cboSuchenNach**
- Kombinationsfeld **Ersetzen durch**: **cboErsetzenDurch**
- Kombinationsfeld **Suchen in**: **cboSuchenIn**
- Kombinationsfeld **Vergleichen**: **cboVergleichen**
- Kombinationsfeld **Suchen**: **cboSuchen**
- Kontrollkästchen **Groß-/Kleinschreibung beachten**: **chkGrossKleinschreibungBeachten**
- Kontrollkästchen **FormatierungBeachten**: **chkFormatierungBeachten**

- Schaltfläche **Weitersuchen**: **cmdWeitersuchen**
- Schaltfläche **Abbrechen**: **cmdAbbrechen**
- Schaltfläche **Ersetzen**: **cmdErsetzen**
- Schaltfläche **Alle ersetzen**: **cmdAlleErsetzen**

Für die Schaltflächen fügen wir passende Icons hinzu, indem wir die jeweilige Schaltfläche markieren und dann den Ribbonbefehl **Formularentwurf|Steuerlemente|Bild einfügen** nutzen.

Den **Hintergrund** der Schaltflächen stellen wir auf **Transparent** ein, die **Anordnung der Bildbeschriftung** auf **Rechts** und die **Schriftbreite** für alle außer der nicht markierten Registerschaltfläche auf **Fett**.

Außerdem fügen wir vor der Eigenschaft **Beschriftung** der Schaltflächen jeweils zwei Leerzeichen ein, damit die Beschriftung nicht zu nah an den Icons erscheint.

Da das Formular selbst keine Datensatzquelle hat, stellen wir die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen** und **Bildlaufleisten** auf **Nein** und **Automatisch zentrieren** auf **Ja** ein.

Schaltflächen zum Entfernen von Such- und Ersetzungstexten

Die beiden Kombinationsfelder **Suchen nach** und **Ersetzen durch** sollen die bisher eingegebenen Texte anzeigen.

Damit der Benutzer auch mal einen der eingegebenen Texte entfernen kann, wollen wir außerdem noch zwei Schaltflächen zum Löschen des aktuell ausgewählten Eintrags hinzufügen.

Diese erhalten die Namen **cmdSuchenNachLoeschen** und **cmdErsetzenDurchLoeschen**. Danach sieht der Entwurf wie in Bild 7 aus. Die Funktionalität dieser Schaltflächen fügen wir später hinzu.

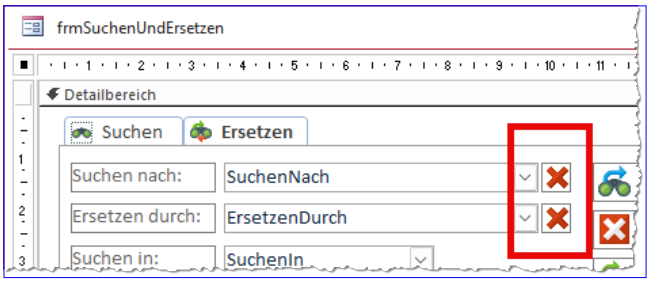

Bild 7: Schaltflächen zum Leeren

Bei Eingabetaste automatisch suchen

Wenn der Benutzer die Eingabetaste betätigt, soll die Schaltfläche **cmdWeitersuchen** ausgelöst werden. Dazu stellen wir die Eigenschaft **Standard** für diese Schaltfläche auf **Ja** ein.

Leider funktioniert das nicht, wenn der Fokus auf einer der übrigen Schaltflächen liegt. Dann wird die jeweils aktive Schaltfläche ausgeführt. Also müssen wir für die übrigen Tasten noch das Ereignis **Bei Taste Ab** definieren und hier prüfen, ob der Benutzer für diese die Eingabetaste betätigt hat. Ist das der Fall, soll die Ereignisprozedur der Schaltfläche **cmdWeitersuchen** ausgelöst werden. Dazu hinterlegen wir zunächst die entsprechende Ereignisprozedur, die wir allerdings erst einmal mit einer Dummy-Meldung füllen:

```
Private Sub cmdWeitersuchen_Click()
    MsgBox "Weitersuchen"
End Sub
```

Für die übrigen Schaltflächen legen wir nun jeweils eine Prozedur für das Ereignis **Bei Taste Ab** an, die wir wie folgt füllen:

```
Private Sub regSuchen_KeyDown(KeyCode As Integer, _
        Shift As Integer)
    Select Case KeyCode
        Case 13
            KeyCode = 0
            Call cmdWeitersuchen_Click
    End Select
```

```
End Sub
```

Damit führt das Betätigen der Eingabetaste nun immer die Taste **cmdWeitersuchen** aus.

Bei Escape-Taste automatisch schließen

Betätigt der Benutzer die **Escape**-Taste, soll der **Suchen und Ersetzen**-Dialog geschlossen werden. Dazu stellen wir die Eigenschaft **Abbrechen** für diese Schaltfläche auf **Ja** ein.

Die Ereignisprozedur für diese Schaltfläche können wir gleich anlegen. Sie lautet:

```
Private Sub cmdAbbrechen_Click()
    DoCmd.Close acForm, Me.Name
End Sub
```

Kombinationsfelder füllen

Nun füllen wir die Kombinationsfelder, zunächst die drei unteren – diese nehmen jeweils eine Wertliste auf.

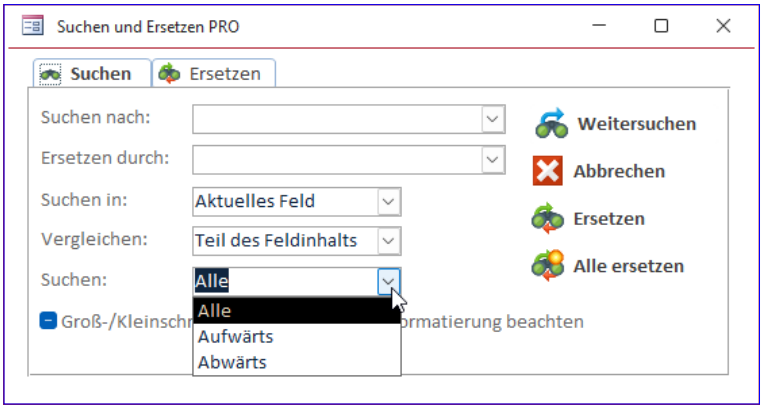

Bild 9: Kombinationsfeld in Aktion

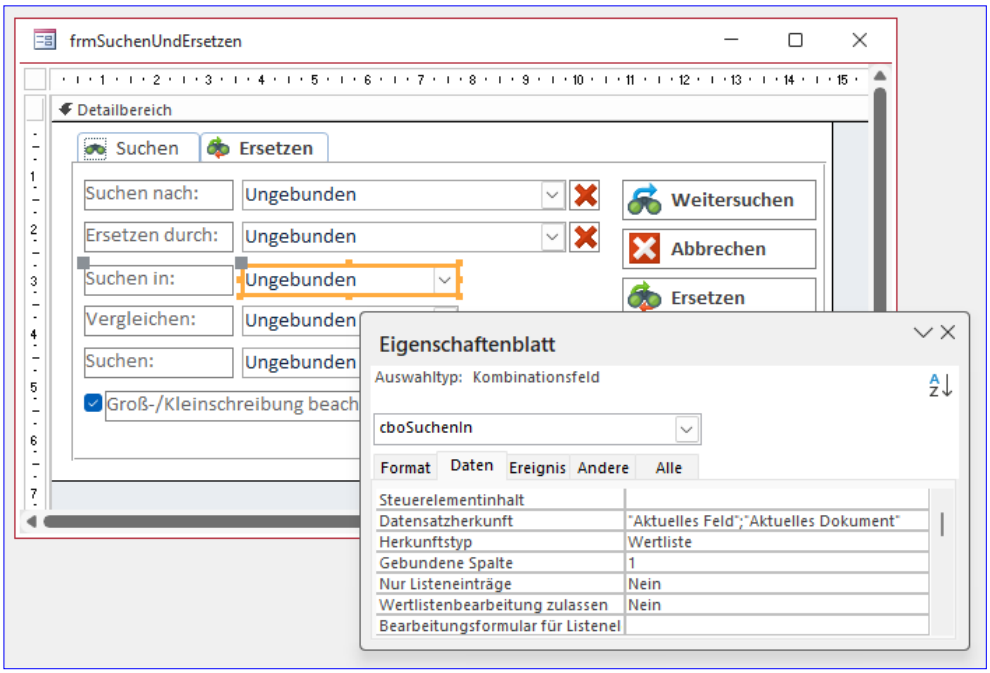

Bild 8: Einstellen der Werte für die Kombinationsfelder

Das Kombinationsfeld **cboSuchenIn** erhält diese Werteliste:

```
"Aktuelles Feld";"Aktuelles Dokument"
```

Damit diese angezeigt wird, stellen wir außerdem die Eigenschaft **Herkunftsart** auf **Wertliste** ein. Außerdem wollen wir verhindern, dass der Benutzer die Wertliste bearbeiten kann und legen daher noch den Wert **Nein** für die Eigenschaft **Wertlistenbearbeitung zulassen** fest (siehe Bild 8).

Die Einstellungen für die weiteren beiden Kombinationsfelder sind weitgehend identisch. Lediglich die Datensatzherkünfte unterscheiden sich. Die für das Kombinationsfeld **cboVergleichen** lautet:

```
"Teil des Feldinhalts";"Ganzes Feld";"Anfang des Feldinhalts"
```

Und für **cboSuchen** verwenden wir:

Suchen und Ersetzen mit COM-Add-In

Im Beitrag »Besserer Suchen und Ersetzen-Dialog« haben wir den Dialog zum Suchen und Ersetzen in Access-Formularen nachgebaut und um einige Features ergänzt. Nun wollen wir unseren selbst gebauten Dialog noch für jede Access-Anwendung verfügbar machen. Was liegt da näher, als diesen über das Ribbon aufzurufen? Um eine solche Erweiterung für jede geöffnete Anwendung verfügbar zu machen und so flexibel wie möglich zu sein, verwenden wir dazu ein COM-Add-In. Dieses enthält lediglich die Erweiterung des Ribbons um die benötigten Befehlsschaltflächen sowie den Code, um von dort aus den »Suchen und Ersetzen«-Dialog zu starten.

COM-Add-Ins erstellen mit twinBASIC

Unsere erste Wahl für das komfortable Erstellen von COM-Add-Ins für Microsoft Access ist twinBASIC. Diese Entwicklungsumgebung ist in der 32-Bit-Version kostenlos und 64-Bit-DLLs können ebenfalls damit erstellt werden – in der kostenlosen Version erscheint hier allerdings ein Splash-Screen des Herstellers.

Den jeweils aktuellen Release kann man hier herunterladen:

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

Hier klickt man auf den gewünschten Release und lädt beispielsweise für den Release mit der Nummer 784 die Datei **twinBASIC_IDE_BETA_784.zip** herunter.

Den Inhalt entpackt man in ein neues, leeres Verzeichnis. Es ist keine weitere Installation notwendig, wir brauchen einfach nur die Datei **twinBASIC.exe** zu starten. Nach dem Start erscheint ein Startdialog, in dem wir zur Registerseite Samples wechseln. Hier finden wir den Eintrag **Sample 5. MYCOMAddIn**, den wir doppelt anklicken (siehe Bild 1).

Danach erscheint ein weiterer Dialog namens **twinBASIC New Project Options**. Hier passen wir den Projektnamen

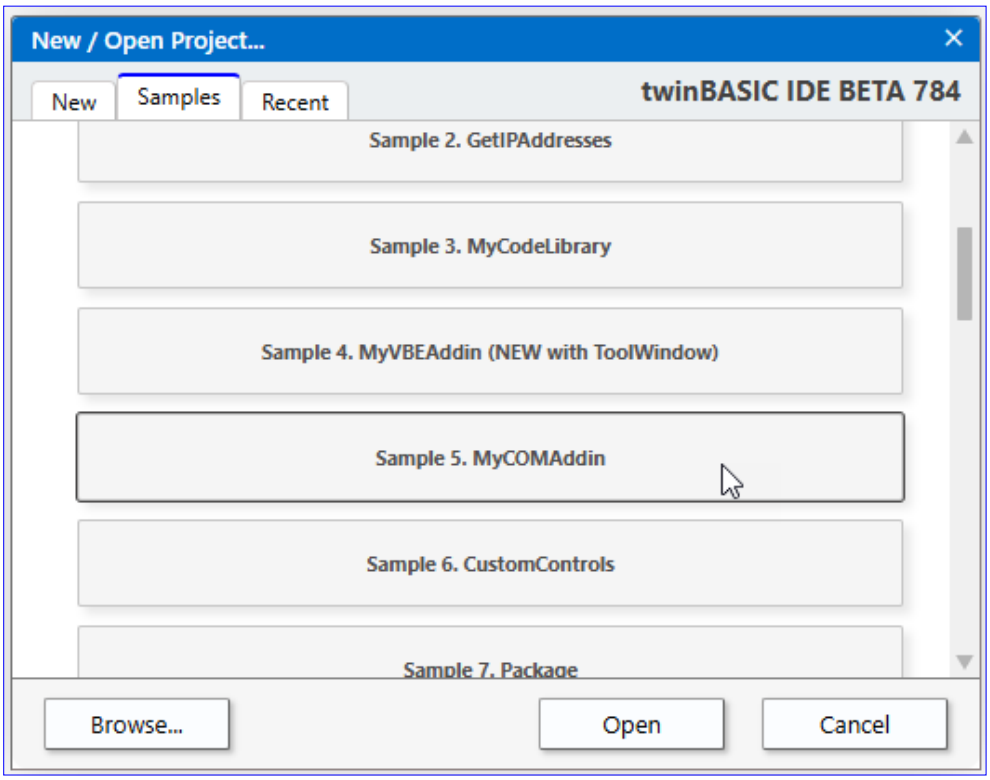

Bild 1: Anlegen eines COM-Add-In-Projekts mit Beispiel

an, in diesem Fall auf **amvSuchenUndErsetzen** (siehe Bild 2).

twinBASIC erstellt nun in wenigen Augenblicken das neue Projekt und öffnet das Modul **MyCOMAddin.twin**.

Verweise hinzufügen

Da wir ein COM-Add-In erstellen wollen, fügen wir diesem zunächst einen Verweis auf die Objektbibliothek von Microsoft Access hinzu.

Dazu öffnen wir den **References**-Bereich mit dem Menübefehl **Project|References**... (siehe Bild 3).

Dort angekommen, wechseln wir zur Registerseite **Available COM References** und geben im Suchbereich **microsoft access** ein. Dies bringt den Verweis auf die Bibliothek **Microsoft Access 16.0 Object Library** zum Vorschein, die wir

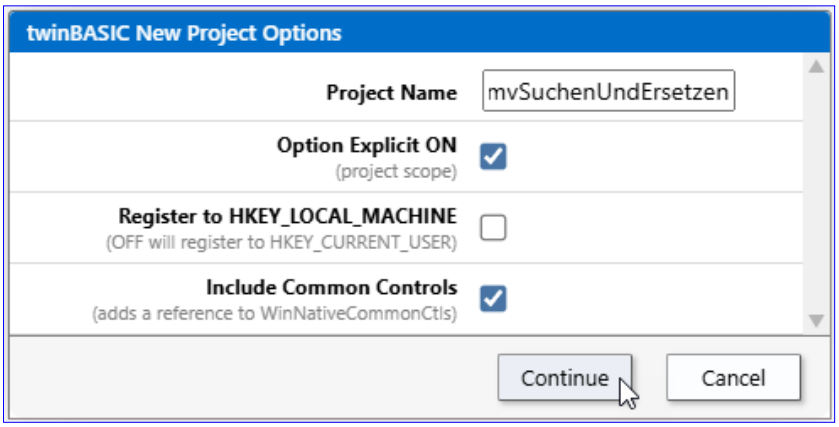

Bild 2: Angeben des Projektnamens

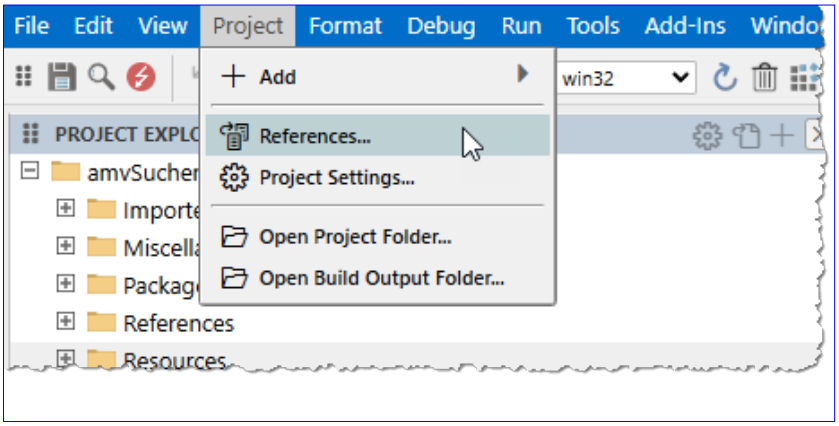

Bild 3: Öffnen der Referenzen

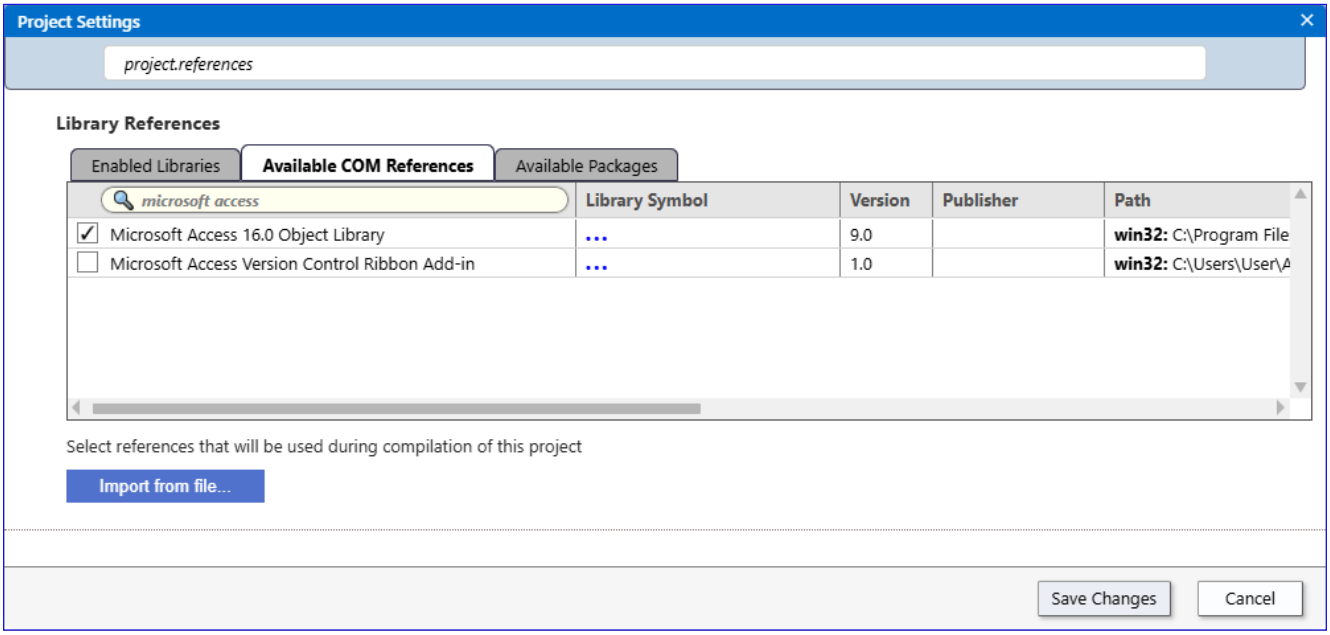

Bild 4: Hinzufügen eines Verweises auf die Access-Bibliothek

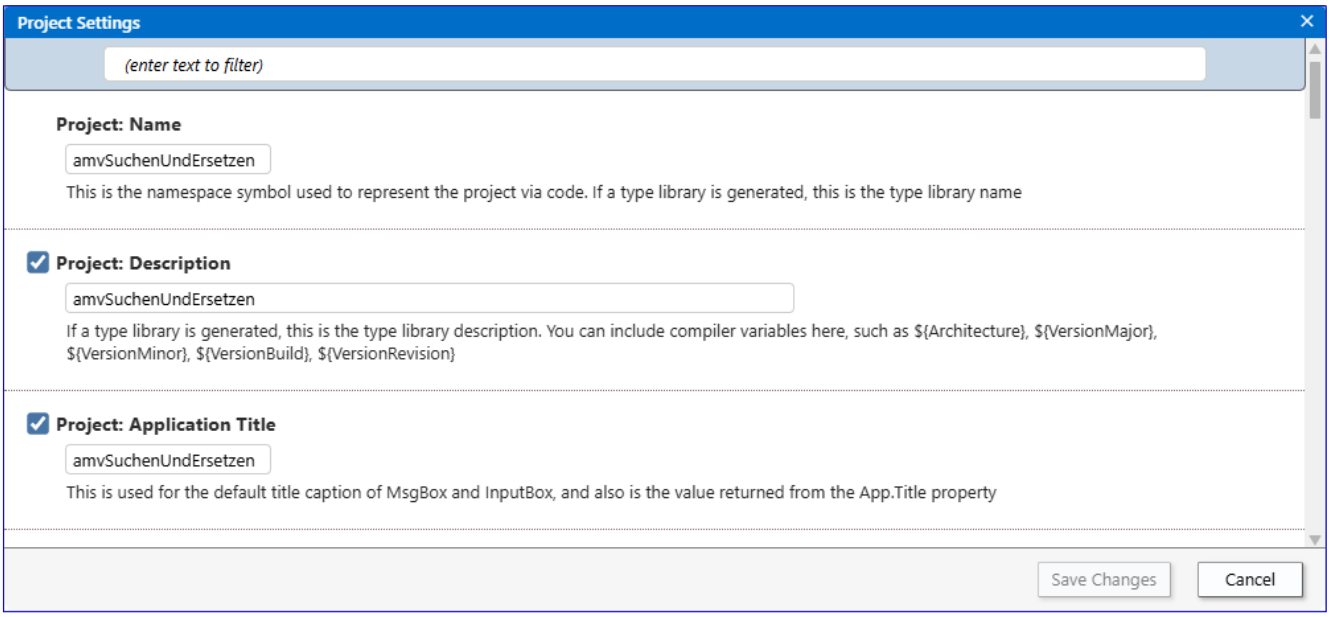

Bild 5: Einstellen von **Name**, **Description** und **Application Title**

nun aktivieren (siehe Bild 4). Anschließend klicken wir auf **Save Changes**.

Einstellungen anpassen

Danach öffnen wir mit **Project|Project Settings** weitere Einstellungen, wo wir jeweils den Namen unseres Projekts eingeben können (siehe Bild 5).

.twin-Datei umbenennen

Nun klicken wir mit der rechten Maustaste links auf den Eintrag **Sources|MyCOMAddin.twin** und wählen aus dem Kontextmenü den Befehl **Rename** aus (siehe Bild 6). Wir können den Namen von **MyCOMAddin.twin** nun bearbeiten und ändern diesen auf **SuchenUndErsetzen.twin**.

Achtung: Die Datei muss zu diesem Zweck geschlossen sein.

Prozeduren zum Registrieren des COM-Add-Ins

Wie sehen neben der frisch umbenannten Datei **SuchenUndErsetzen.twin** noch eine weitere Datei namens **DllRegistration.twin**. Wenn wir diese öffnen, sehen wir verschiedene Elemente, die wir durch den Inhalt aus Listing 1 ersetzen.

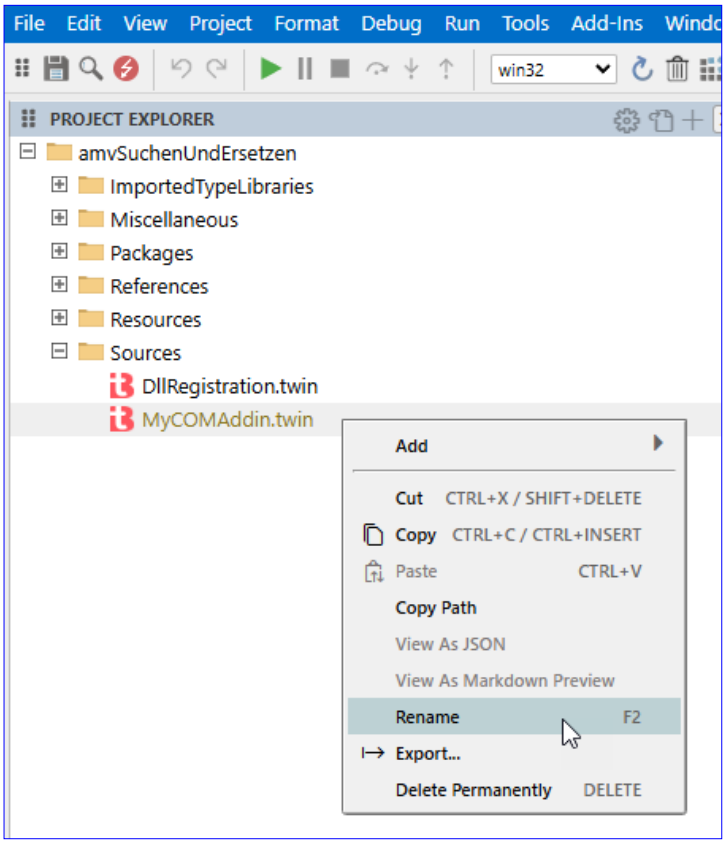

Bild 6: Umbenennen der Code-Datei

Ganz oben finden wir einige Konstanten, die unter anderem mit dem Projektnamen gefüllt werden (**AddinProjectName**) oder mit dem Namen der zu verwendenden Klasse (**SuchenUndErsetzen**).

Darunter finden wir zwei Funktionen. Diese Funktionen werden ausgeführt, wenn wir das COM-Add-In registrieren oder die Registrierung aufheben wollen, zum Beispiel über die Kommandozeile oder auch beim Registrieren über die COM-Add-Ins in den Access-Optionen.

Die erste Funktion **DllRegisterServer** nimmt die für den Einsatz notwendigen Einträge in der Registry vor, die zweite Funktion **DllUnregisterServer** entfernt diese wieder.

Code der COM-DLL

Nun müssen wir nur noch den Code erstellen, mit dem wir die gewünschten Einträge zum Ribbon hinzufügen und dafür sorgen, dass das COM-Add-In überhaupt ausgeführt wird, wenn wir Access starten.

```
Module DllRegistration
    Const AddinProjectName As String = VBA.Compilation.CurrentProjectName
    Const AddinClassName As String = "SuchenUndErsetzen"
    Const AddinQualifiedClassName As String = AddinProjectName & "." & AddinClassName
    Const RootRegistryFolder As String = "HKCU\SOFTWARE\Microsoft\Office\Access\Addins\" & AddinQualifiedClassName & "\"
    Public Function DllRegisterServer() As Boolean
        On Error GoTo RegError
        Dim wscript As Object = CreateObject("wscript.shell")
        wscript.RegWrite RootRegistryFolder & "FriendlyName", AddinProjectName, "REG_SZ"
        wscript.RegWrite RootRegistryFolder & "Description", AddinProjectName, "REG_SZ"
        wscript.RegWrite RootRegistryFolder & "LoadBehavior", 3, "REG_DWORD"
        Return True
    RegError:
        MsgBox "DllRegisterServer -- An error occured trying to write to the system registry:" & vbCrLf & _
                Err.Description & " (" & Hex(Err.Number) & ")"
        Return False
    End Function
    Public Function DllUnregisterServer() As Boolean
        On Error GoTo RegError
        Dim wscript As Object = CreateObject("wscript.shell")
        wscript.RegDelete RootRegistryFolder & "FriendlyName"
        wscript.RegDelete RootRegistryFolder & "Description"
        wscript.RegDelete RootRegistryFolder & "LoadBehavior"
        wscript.RegDelete RootRegistryFolder
        Return True
    RegError:
        MsgBox "DllUnregisterServer -- An error occured trying to delete from the system registry:" & vbCrLf & _
                Err.Description & " (" & Hex(Err.Number) & ")"
        Return False
    End Function
End Module
```

Listing 1: Prozeduren zum Registrieren des COM-Add-Ins

Dazu benötigen wir die Schnittstelle **IDTExtensibility**, die wir durch Hinzufügen mit dem Schlüsselwort **Implements** zur Klasse **SuchenUndErsetzen** bekannt geben.

Außerdem benötigen wir noch eine zweite Schnittstelle, damit wir über eine Funktion die Ribbondefinition übergeben können. Diese machen wir durch Anhängen von Implements **IRibbonExtensibility** und dem zusätzlichen Hinweis **[WithDispatchForwarding]** für die Klasse bekannt.

IDTExensibility2-Schnittstelle implementieren

Durch das Implementieren diese Schnittstellen müssen wir die dadurch definierten Methoden, Ereignisse und Eigenschaften ebenfalls im Code implementieren. Bei der Schnittstelle **IDTExtensibility2** bedeutet dies, dass wir einige Ereignisprozeduren anlegen müssen, die beim Laden des COM-Add-Ins durch Access ausgelöst werden, andere werden zu anderen Zeitpunkten aktiviert. Die wichtigste für uns ist die Ereignisprozedur **OnConnection**, die beim Start von Access ausgelöst wird und mit der Access uns mit dem Parameter **Application** einen Verweis auf die aktuelle Access-Instanz übergibt. Diesen Verweis speichern wir in der Variablen **objAccess**, damit wir später darüber auf die Access-Instanz zugreifen können. Das ist in diesem Fall streng genommen gar nicht nötig, aber wir führen diesen Code immer mit, weil er für die meisten COM-Add-Ins, die direkt Aktionen mit dem Access-Objekt ausführen sollen, nötig ist.

Diesen Teil des Codes finden Sie in Listing 2.

Ribbon erweitern

In unserem Fall wollen wir das COM-Add-In nur nutzen, um dem Ribbon von Access einige neue Elemente hinzuzufügen. Dazu implementieren wir die Funktion **GetCustomUI** der Schnittstelle **IRibbonUI**. Diese wird von Access aus aufgerufen und wir bekommen so die Gelegenheit, Access eine Anpassung des Ribbons zu übergeben. Wir tun das wie in Listing 3, wo wir die passende Ribbon-Definition in der Variablen **strXML** zusammenstellen.

```
[ClassId("DC8E7154-FAC8-43B7-88E5-A93527539222")]
Class SuchenUndErsetzen
    Implements IDTExtensibility2
    [WithDispatchForwarding]
    Implements IRibbonExtensibility
    Private objAccess As Access.Application
    Private objRibbon As IRibbonUI
    Sub OnConnection(ByVal Application As Object, ByVal ConnectMode As ext_ConnectMode, ByVal AddInInst As Object, _
            ByRef custom As Variant()) Implements IDTExtensibility2.OnConnection
        Set objAccess = Application
    End Sub
    Sub OnDisconnection(ByVal RemoveMode As ext_DisconnectMode, ByRef custom As Variant()) _
            Implements IDTExtensibility2.OnDisconnection
    End Sub
    Sub OnAddInsUpdate(ByRef custom As Variant()) Implements IDTExtensibility2.OnAddInsUpdate
    End Sub
    Sub OnStartupComplete(ByRef custom As Variant()) Implements IDTExtensibility2.OnStartupComplete
    End Sub
    Sub OnBeginShutdown(ByRef custom As Variant()) Implements IDTExtensibility2.OnBeginShutdown
    End Sub
    ...
```

Listing 2: Code des COM-Add-Ins, Teil 1

Automatisches Update per Startdatenbank

Access-Datenbankanwendungen sind bekanntlich niemals fertig. Das ist an sich kein Nachteil, sondern stellt eher einen Vorteil dar: Im Gegensatz zu fertigen Anwendungen, bei denen Anpassungen entweder gar nicht möglich sind oder viel Geld kosten, kann man Access-Datenbanken einfach selbst um neue Funktionen erweitern. Spannend wird das, wenn die Datenbank von mehreren Nutzern eingesetzt wird. Dann stellt sich die Frage: Wie bekomme ich die neue Version einer Datenbank am einfachsten auf die unterschiedlichen Arbeitsplätze? Das gelingt auf verschiedene Arten, und in diesem Beitrag schauen wir uns eine an, mit der das Update vom Arbeitsplatz des Benutzers aus getriggert werden soll.

Wenn wir eine klassische Konstellation vorliegen haben, bei der jeder Benutzer einen eigenen Rechner hat, der an das Firmennetzwerk angeschlossen ist, ist die nachfolgende beschriebene Vorgehensweise perfekt geeignet. In diesem Fall agieren wir also als Entwickler in oder für das entsprechende Unternehmen. Es gibt noch einen anderen Fall, den wir in einem weiteren Beitrag besprechen wollen – hier ist der Entwickler selbständig und verteilt seine Anwendung an Anwender aus verschiedenen Unternehmen.

Im ersten Fall würden wir die neue Version der Anwendung auf einem Server im Unternehmen platzieren, von wo aus diese dann heruntergeladen werden kann. Im zweiten Fall steht die Datenbankdatei im Internet zum Download bereit.

Normalerweise startet der Benutzer seine Kopie der Datenbankanwendung, indem er eine entsprechende Verknüpfung aufruft oder doppelt auf die Datenbankdatei klickt.

Wir wollen diesen Ablauf so ändern, dass er nicht direkt die eigentliche Datenbankanwendung startet, sondern dass die Verknüpfung eine Startdatenbank öffnet. Diese prüft dann, ob es eine neue Version der eigentlichen Datenbank gibt und lädt diese gegebenenfalls herunter und kopiert sie über die vorhandene Version der Anwendung.

Anschließend soll die Startdatenbank die eigentliche Datenbank starten. Wenn keine neue Version vorliegt, soll die Startdatenbank einfach die vorhandene Anwendung aufrufen.

Code automatisch aufrufen beim Start der Startdatenbank

Die erste Technik, die wir benötigen, ist ein automatischer Aufruf der Prozeduren zum Prüfen auf Updates und der weiteren Schritte beim Start der Startdatenbank.

Das erledigen wir zuverlässig mit einem **AutoExec**-Makro. Wir legen also ein neues Makro an und speichern dieses

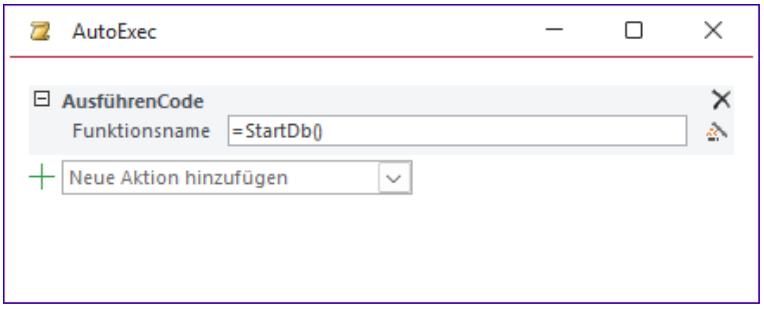

Bild 1: AutoExec-Makro, das beim Start automatisch ausgeführt wird

unter dem Namen **AutoExec**. Dieses wird beim Starten von Access automatisch aufgerufen.

Dem Makro weisen wir eine einzige Makroaktion zu. Diese heißt **AusführenCode** und erhält für den Parameter **Funktionsname** den Wert **=StartDb()**. Das Makro sieht in der Entwurfsansicht wie in Bild 1 aus.

Die aufzurufende Funktion gestalten wir erst einmal wie folgt:

```
Public Function StartDb()
    MsgBox "Start"
End Function
```

Das Speichern aller Änderungen und das Schließen und erneute Öffnen der Anwendung sollte nun die Anwendung öffnen und das Meldungsfenster anzeigen.

Erkennen einer neuen Version

Bevor wir uns die Techniken ansehen, mit denen wir eine Datenbank durch eine neuere Version ersetzen, benötigen wir erst einmal die Merkmale, mit denen wir eine neue Version erkennen können.

Üblicherweise verwendet man hier eine Versionsnummer, die auf unterschiedliche Arten gestaltet sein kann. Am einfachsten, auch für den hier vorgesehenen Zweck, ist ein einfacher **Long**-Wert für die Version. Damit können wir schnell und einfach vergleichen: Ist der Versionswert der aktuellen Version kleiner als der entsprechenden Version der Datenbank auf dem Serverlaufwerk?

Gelegentlich verwendet man auch umfangreichere Versionsnummern wie **1.0.0.4**. Auch das können wir umsetzen, allerdings ist der Aufwand minimal höher, da wir Stelle für Stelle vergleichen müssen.

Wenn man davon ausgeht, dass es maximal eine neue Version pro Tag gibt, könnten wir auch einfach das Datum der jeweiligen Version heranziehen. Dann spielt es keine Rolle, ob wir nur einen einfachen **Long**-Wert als Versionsnummer verwenden oder einen Ausdruck wie **1.0.0.4**.

Wir wollen an dieser Stelle Komplexität herausnehmen und verwenden einfach ein Datumsfeld als Kriterium für die Notwendigkeit, die Anwendung mit einer neueren Version zu aktualisieren. Zusätzlich wollen wir die Versionsnummer in einem weiteren Feld speichern, dem wir den Datentyp **Kurzer Text** geben.

Die folgende Tabelle **tblVersion** legen wir in der jeweiligen Produktdatenbank an.

Die Tabelle **tblVersion** sieht in der Entwurfsansicht so aus wie in Bild 2. Hier kann man bei Bedarf noch weitere Felder hinzufügen, zum Beispiel zur Angabe von weiteren Informationen zu dieser Version wie den Neuerungen oder Fehlerkorrekturen, die in dieser Version enthalten sind.

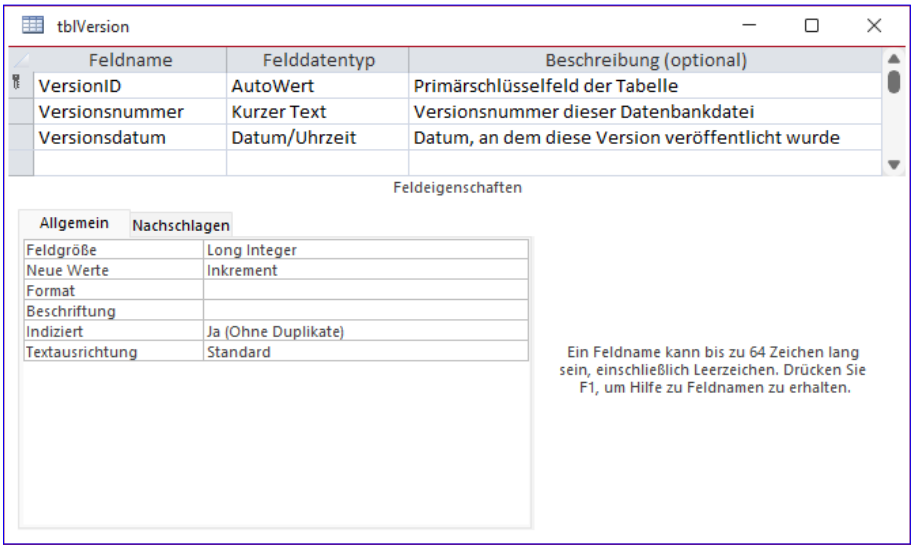

Bild 2: Tabelle zum Speichern der Versionsinformationen in der Entwurfansicht

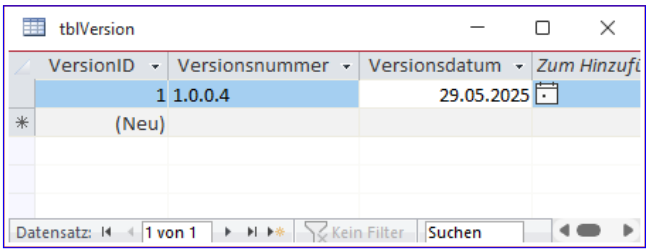

Bild 3: Tabelle mit der Versionsnummer

Wechseln wir in dieser Tabelle zur Datenblattansicht, tragen wir beispielsweise Informationen wie in Bild 3 ein.

Für die Versionsnummer können wir nun ein beliebiges Format verwenden. Wir sollten nur sicherstellen, dass diese korrekt erhöht werden, damit der Benutzer nicht plötzlich eine neue Version mit einer niedrigeren Versionsnummer vorfindet.

Standort der Startdatenbank und der Produktivdatenbank

Der Einfachheit halber sollten die Startdatenbank und die Produktivdatenbank immer im gleichen Verzeichnis liegen.

So können wir von der Startdatenbank ganz einfach den Pfad zur Produktivdatenbank ermitteln, indem wir mit **CurrentProject.Path** den Pfad der beiden Datenbankdateien ermitteln und dann den Namen der Produktivdatenbank anhängen.

Die jeweils neuere Version, durch welche die aktuell auf dem Clientrechner vorliegende Version ersetzt werden soll, wird logischerweise nicht im gleichen Verzeichnis liegen, sondern an einer zentral zugänglichen Stelle auf einem Server im Unternehmen oder auch in der Cloud. Hier müssen wir daher den kompletten Pfad in der Startdatenbank speichern.

Insgesamt benötigen wir also zwei Felder in einer Konfigurationstabelle, die wir der Startdatenbank hinzufügen. Diese Tabelle heißt **tblKonfiguration** und enthält den Entwurf aus Bild 4.

Das erste Feld namens **DatenbanknameClient** nimmt den Namen der aktuellen Version der Datenbankdatei im aktuellen Verzeichnis der Startdatenbank auf. Das zweite Feld heißt **DatenbankpfadServer** und enthält den vollständigen Pfad der zu prüfenden, gegebenenfalls aktuelleren Version der Datenbankdatei. Beide haben den Datentyp **Kurzer Text**.

Gegebenenfalls kommt es vor, dass die 255 Zeichen eines Feldes mit dem Datentyp **Kurzer Text** nicht ausreichen, um den vollständigen Pfad zu speichern. In diesem Fall haben wir zwei Möglichkeiten:

- Wir teilen das Feld **DatenbankpfadServer** auf zwei Felder auf, die das Verzeichnis und den Dateinamen speichern.
- Oder wir verwenden einfach den Felddatentyp **Langer Text**.

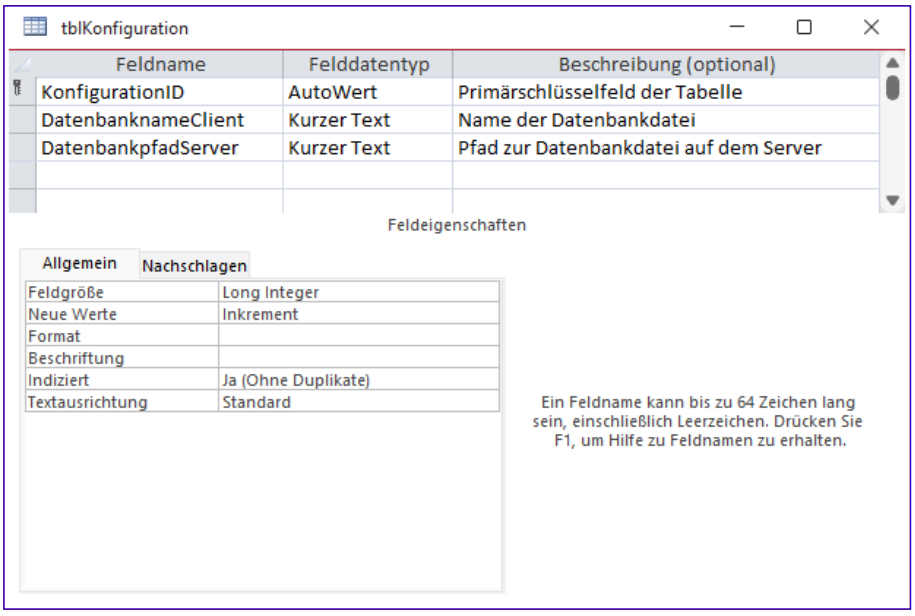

Bild 4: Tabelle zum Speichern der Konfiguration der Startdatenbank

Grundstruktur der Aktualisierung

Bevor wir uns an die Details begeben, mit denen wir die verschiedenen Bedingungen prüfen und Aktionen durchführen, bauen wir eine Grundstruktur für den Vorgang auf.

Diese finden wir in Listing 1.

```
Public Function DatenbankAktualisieren(ByRef strErgebnis As String, ByRef bolFehler As Boolean) As Boolean
    Dim strPfadClient As String
    Dim strPfadServer As String

    strPfadClient = CurrentProject.Path & "\" & DLookup("DatenbanknameClient", "tblKonfiguration")
    strPfadServer = DLookup("DatenbankpfadServer", "tblKonfiguration")

    If Not DateiVorhanden(strPfadClient) Then
        strErgebnis = "Client-Datei fehlt: " & strPfadClient
        bolFehler = True
        Exit Function
    End If
    If Not DateiVorhanden(strPfadServer) Then
        strErgebnis = "Server-Datei fehlt: " & strPfadServer
        bolFehler = True
        Exit Function
    End If
    If IstDatenbankGeoeffnet(strPfadClient) Then
        strErgebnis = "Client-Datenbank ist geöffnet."
        bolFehler = True
        Exit Function
    End If
    If IstDatenbankGeoeffnet(strPfadServer) Then
        strErgebnis = "Server-Datenbank ist geöffnet."
        bolFehler = True
        Exit Function
    End If

    If NeuereVersionVorhanden(strPfadClient, strPfadServer, strErgebnis) Then
        If KopiereServerdateiAufClient(strPfadClient, strPfadServer, strErgebnis) Then
            strErgebnis = "Datenbank erfolgreich aktualisiert." & vbCrLf & strErgebnis
            DatenbankAktualisieren = True
        Else
            strErgebnis = "Fehler beim Aktualisieren: " & strErgebnis
            bolFehler = True
        End If
    Else
        strErgebnis = "Es konnte keine neue Version gefunden werden." & vbCrLf & strErgebnis
    End If

End Function
```

Listing 1: Hauptfunktion zum Aktualisieren einer Datenbankdatei

Die Funktion liefert als Ergebnis den Wert **True** oder **False** und außerdem mit dem Parameter **strErgebnis** eine Meldung mit weiteren Details. Da **strErgebnis** sowohl im Fehlerfall als auch bei Aktualisierung/Nicht-Aktualisierung ohne Fehler geliefert wird, verwenden wir mit **bolFehler** einen weiteren Parameter, der angibt, ob ein Fehler aufgetreten ist. **bolFehler** wird immer auf **True** eingestellt, wenn ein Ereignis eintritt, das die Prüfung der Versionen oder die Aktualisierung an sich verhindert.

Zu Beginn deklariert die Funktion einige Variablen für die Versionsdaten der beiden Datenbankdateien und ihre Pfade.

Danach lesen wir aus der Tabelle **tblKonfiguration** den Pfad der zu prüfenden aktuellen Datenbankdatei auf dem Clientrechner aus. Dazu stellen wir den Pfad aus dem aktuellen Pfad der Startdatenbank und dem Namen der Datenbank aus der Tabelle zusammen und speichern diesen in der Variablen **strClientpfad**.

Das Gleiche erledigen wir anschließend für den Pfad der Datenbank auf dem Server, die gegebenenfalls die Version auf dem Client ersetzen soll. Dieser Pfad landet in der Variablen **strServerpfad**.

Danach folgt eine Reihe **If...Then**-Bedingungen. Die erste prüft, ob die Datei aus **strClientpfad** überhaupt vorhanden ist. Dazu nutzen wir die Hilfsfunktion **DateiVorhanden**, die wir weiter unten beschreiben.

Ist die Daten nicht vorhanden, endet die Funktion hier mit **Exit Function**, nachdem sie für **strErgebnis** eine entsprechende Rückmeldung eingetragen hat. Anderenfalls wird die Funktion fortgesetzt.

Die nachfolgende Bedingung ist identisch aufgebaut und prüft das Vorhandensein der zu prüfenden Version der Datenbank auf dem Server. Auch hier gilt: Ist diese Datenbank nicht vorhanden, endet die Funktion mit einer entsprechenden Meldung.

Damit ist nun sichergestellt, dass beide betroffenen Datenbankdateien vorhanden sind.

Nun folgt eine weitere wichtige Prüfung für die beiden Datenbankdateien. Diese untersucht, ob diese nicht geöffnet sind. Das ist zumindest bei der aktuellen Clientdatenbank essenziell, da diese sonst nicht durch eine eventuell vorhandene neue Version ersetzt werden kann.

Diese Prüfung nehmen wir in zwei aufeinanderfolgenden **If...Then**-Bedingungen vor, die jeweils die Funktion **IstDatenbankGeoeffnet** nutzen.

Liefern beide Prüfungen den Wert **True**, geht es weiter, ansonsten endet die Funktion hier wiederum mit entsprechenden Einträgen für den Parameter **strErgebnis**.

Sind die Grundvoraussetzungen geschaffen, können wir endlich mit der Funktion **NeuereVersionVorhanden** prüfen, ob die unter **strServerPfad** gespeicherte Datenbankdatei in einer neueren Version vorhanden ist als die unter **strClientPfad** gespeicherte.

Auch diese Funktion beschreiben wir weiter unten im Detail. Erwartungsgemäß führt ein negatives Ergebnis zum Abbruch der Funktion mit entsprechendem Eintrag für **strErgebnis**.

Schließlich sind alle Bedingungen erfüllt und die Routine ruft die Funktion **DatenbankAktualisiert** auf. Dieser übergeben wir die Pfade zu den beiden Datenbanken. Sie soll das eigentliche Ersetzen der alten durch die neue Version erledigen.

Nachfolgend beschreiben wir nun die verwendeten Funktionen.

Prüfen, ob Dateien vorhanden sind

Die Funktion **DateiVorhanden** nimmt mit dem Parameter **strPfad** den Pfad der Datei entgegen, deren Vorhandensein wir prüfen wollen.