HAUFE.
Ausgabe 02/2016

ACCESS
IM UNTERNEHMEN

TICKETSYSTEM

Importieren Sie Kundenanfragen in eine Datenbank und verarbeiten Sie diese komfortabel mit unserer Ticketverwaltung (ab S. 67).

In diesem Heft:

API-PROGRAMMIERUNG

Lernen Sie die Geheimnisse der API-Programmierung unter Access und VBA kennen!

SEITE 56

CSV-VERKNÜPFUNGEN

Legen Sie Verknüpfungen zu CSV-Dateien an und pflegen Sie diese bequem per VBA.

SEITE 2

STANDARDVERZEICHNISSE

Greifen Sie flexibel auf Systemverzeichnisse, Datenbankverzeichnisse und mehr zu.

SEITE 51

Wunschkonzert

Im Vorfeld dieser Ausgabe haben wir Sie nach Ihren Wünschen gefragt. Welche Themen interessieren Sie? Mit welchen Lösungen können wir Sie beglücken? Bei welchen kleinen Problemen können wir Ihnen mit einer Antwort in Form eines Beitrags helfen? Das Feedback war phänomenal: Sie haben uns mit über hundert Vorschlägen mehr als genug Material für diese und die folgenden Ausgaben geliefert. Vielen Dank dafür!

Dementsprechend können Sie, liebe Leser und Leserinnen, stolz von sich behaupten: Wir haben den Inhalt der aktuellen Ausgabe entscheidend mitgestaltet. Im Mittelpunkt steht dabei die Lösung, die wir ab S. 67 unter dem Titel **Ticketsystem, Teil 1** finden. Gleich mehrere Leser haben uns gefragt, ob wir nicht einmal eine Lösung vorstellen können, mit der man eingehende Kundenanfragen aufnehmen und verarbeiten kann. Dies haben wir aufgenommen und uns überlegt, wie eine solche Lösung für einen Benutzer von Access und den übrigen Office-Komponenten aussehen könnte. Damit stellte sich zunächst die Frage: Wie erreichen uns solche Anfragen eigentlich? Meist geschieht dies per E-Mail. Deshalb konzentrieren wir uns auch darauf, E-Mail-Anfragen auf komfortable Weise aus Outlook in die Access-Lösung namens Ticketsystem zu importieren.

Dabei wollen wir die Daten gleich in einen Ticket-Datensatz überführen. Dazu haben wir uns ein pfiffiges Formular ausgedacht, dass alle offenen E-Mails anzeigt und uns ermöglicht, auf dieser Basis gleich zwei Informationen zu speichern: erstens den Kunden, der die Anfrage gestellt hat, und zweitens die Anfrage selbst in Form eines Tickets.

Damit Sie den wichtigen Teil der E-Mail-Anfrage in den Ticket-Datensatz überführen können, stellen wir im Beitrag **Markierung automatich kopieren** (ab S. 16) eine Technik vor, mit der Sie den zu kopierenden Inhalt einfach nur markieren müssen – egal, ob mit der Maus oder mit der Tastatur. Der markierte Bereich wird dann wie von Geisterhand in das Ziel-Textfeld kopiert.

Damit die E-Mail-Inhalte dazu möglichst einfach erfasst werden können, sollen diese in ausreichend großen Textfeldern dargestellt werden. Nun kommen E-Mails mal in reiner Textform, mal im HTML-Format an. Das Formular soll beide Inhalte anzeigen, aber aus Platzgründen kann nur einer von beiden ausreichend groß dargestellt werden. Der Beitrag **Textfelder mit flexibler Höhe** (ab S. 12) zeigt, wie Sie Textfelder abwechselnd vergrößert darstellen, indem Sie einfach das zu vergrößernde Textfeld anklicken.

Zu einem Ticket soll der Benutzer eine oder mehrere Notizen anlegen können. Damit die Notizen auch alle übersichtlich dargestellt werden, verwendet man üblicherweise die Datenblatt- oder Endlosformularansicht. Beide passen ihre Höhe jedoch nicht an die Inhalte an. Wenn Sie die Höhe also so einstellen, dass auch lange Texte vollständig dargestellt werden, nehmen kurze Texte den gleichen Platz ein und hinterlassen einen großen leeren Bereich. Dieses Problem haben wir im Beitrag **HTML-Liste mit Access-Daten** ab S. 27 gelöst: Dort zeigen wir, wie Sie das Webbrowser-Steuerelement von Access nutzen, um die Inhalte in einer HTML-Tabelle darzustellen – und dies bei flexibler, vom jeweiligen Inhalt abhängiger Zellenhöhe. Als Bonbon geben wir noch dazu, wie Sie etwa per Doppelklick auf einen der Texte ein Formular zum Bearbeiten desselben öffnen können.

Im Bereich VBA und Programmiertechnik widmen wir uns grundlegend dem Thema API-Programmierung. Windows bietet mit der API eine Schnittstelle, die Ihnen alle Möglichkeiten bei der Windows-Programmierung eröffnet. Dabei können Sie die verfügbaren Funktionen allerdings nicht so einfach wie Funktionen aus den eingebundenen Bibliotheken nutzen, sondern müssen diese zuvor auf bestimmte Art verfügbar machen. Wie dies gelingt, zeigt der Beitrag **Windows-API per VBA nutzen** ab S. 56.

Schließlich liefern wir mit dem Beitrag **CSV-Verknüpfungen pflegen** ab S. 2 noch die Grundlagen zum Erstellen von Verknüpfungen mit CSV-Dateien und erläutern die Techniken, um solche Verknüpfungen per VBA auf dem aktuellen Stand zu halten.

Und nun: Viel Spaß beim Lesen!

Ihr Michael Forster

CSV-Verknüpfungen pflegen

Wenn Sie Dateien im .csv-Format in Ihre Datenbank eingebunden haben, kommt es gelegentlich vor, dass diese verschoben, umbenannt oder gelöscht werden. Dieser Beitrag zeigt nicht nur, wie Sie .csv-Dateien per VBA einbinden können, sondern auch, wie Sie erkennen, ob die Dateien noch an Ort und Stelle sind und diese gegebenenfalls mithilfe des Benutzers wieder einbinden.

Als Beispiel-Datei zum Verknüpfen verwenden wir einfach eine im **.csv**-Format exportierte Tabelle der Südsturm-Datenbank, in diesem Fall **tblArtikel**. Dazu nutzen wir den Export-Assistenten, den Sie über den Ribbon-Eintrag **Externe Daten|Exportieren|Textdatei** aufrufen. Hier stellen Sie beispielsweise ein, dass die Datei mit Spaltenüberschriften gespeichert werden soll. Außerdem öffnen Sie mit einem Klick auf die Schaltfläche **Erweitert...** den Dialog mit den Exportspezifikationen.

Speichern Sie diese per Mausklick auf die Schaltfläche **Speichern unter...** und geben Sie im Dialog **Import/Export-Spezifikation speichern** den Namen **tblArtikel_CSV** ein (s. Bild 1).

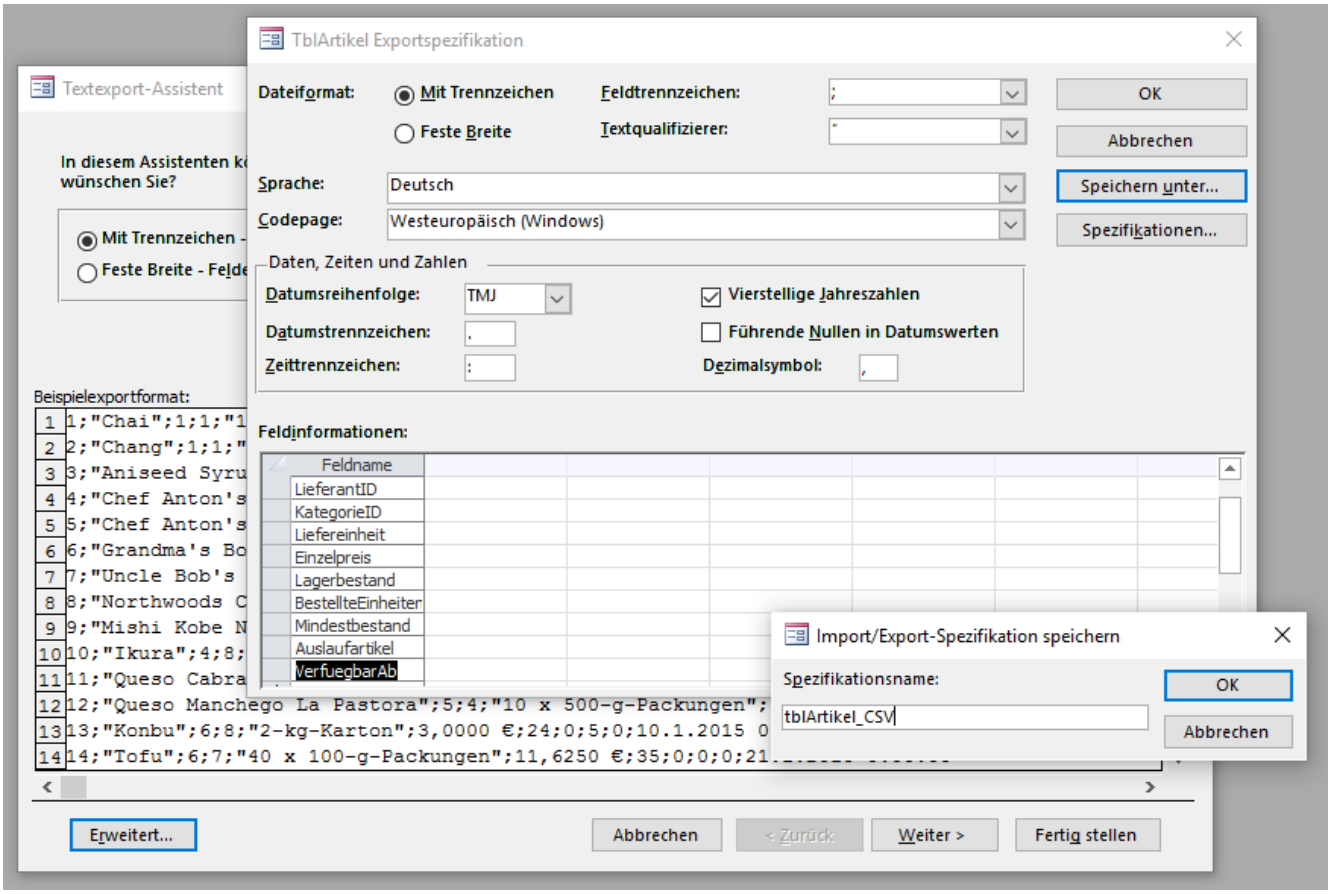

Bild 1: Export der Beispieltabelle, die später verknüpft werden soll

Nun wollen wir eine Verknüpfung zu der soeben in eine CSV-Datei exportierten Tabelle mit dem dafür vorgesehenen Assistenten erstellen (Ribbon-Eintrag **Externe Daten|Importieren und Verknüpfen|Textdatei**). Egal, ob wir dies mit Access 2013 oder 2016 erledigen wollten: Es gab immer eine Fehlermeldung wie die aus Bild 2.

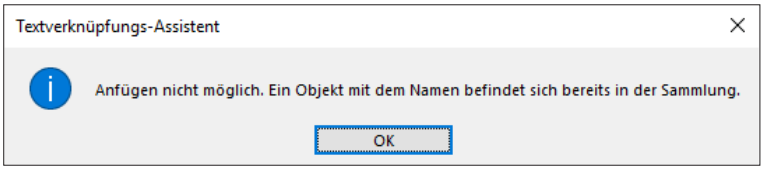

Bild 2: Fehlermeldung beim Versuch, eine Verknüpfung zu erstellen

Diese erscheint auch unabhängig davon, ob wir beim Verknüpfen die Textdateispezifikation auswählen oder nicht. In beiden Fällen legte Access jedoch die Verknüpfung in der Liste der Tabellen an (s. Bild 3).

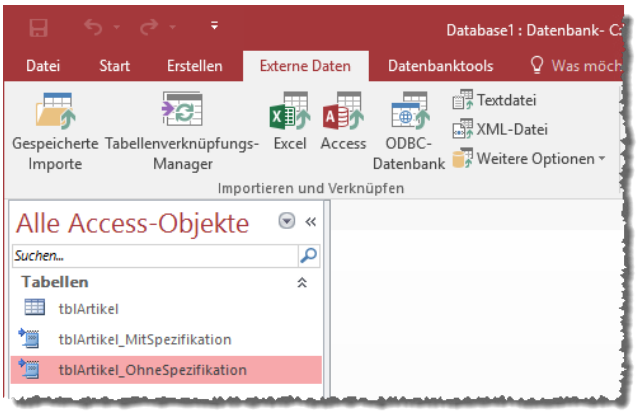

Bild 3: Access erstellt die Verknüpfungen trotz Fehlermeldung.

Wenn Sie keine Textdateispezifikation auswählen, liefert Access jedoch noch zusätzlich die Fehlermeldung aus Bild 4. Anscheinend sucht Access hier nach der Textdateispezifikation, deren Name standardmäßig beim Anlegen einer solchen vorgeschlagen wird – diese ist aber nicht in der Datenbank gespeichert (die Textdateispezfikationen finden Sie übrigens in den Systemtabellen **MSysIMEXSpecs** und **MSysIMEXColumns**).

In einer weiteren Systemtabelle namens **MSysObjects** entdecken wir, dass auch beim Verknüpfen ohne Textdateispezifikation eine solche angegeben wird – in diesem Fall als Wert der Eigenschaft **DSN** mit dem Wert **TblArtikel Verknüpfungsspezifikation** (s. Bild 5).

Es scheint sich hier also um einen Fehler des Import-Assistenten zu handeln. Das ist aber kein Problem, denn wir streben ja in diesem Beitrag ohnehin eine VBA-gesteuerte Methode zum Verknüpfen der CSV-Dateien an.

Verschieben der CSV-Datei

Wenn Sie nun einmal die CSV-Datei verschieben und die darauf basierende Verknüpfung öffnen möchten, findet Access die angegebene Datei natürlich nicht mehr und

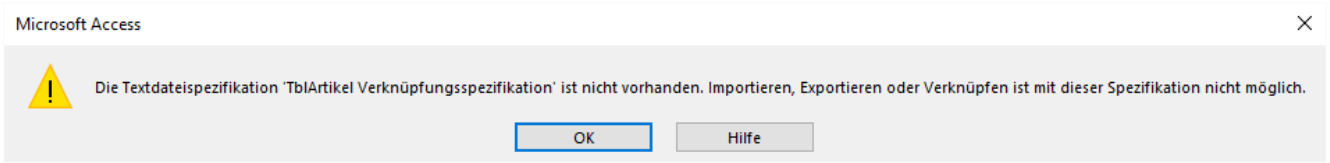

Bild 4: Fehlermeldung wegen einer fehlenden Textdateispezifikation

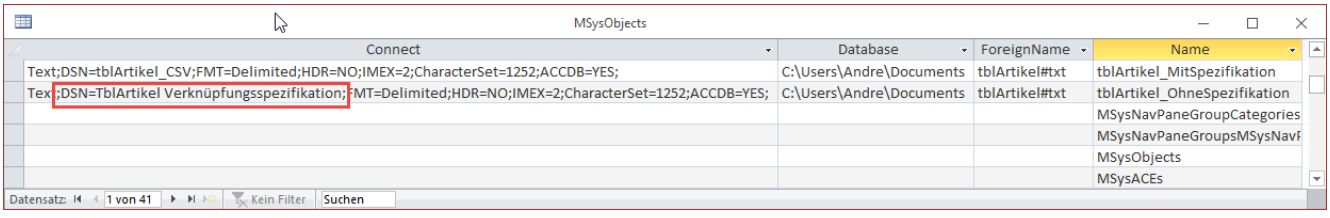

Bild 5: Die Importspezifikation wird in der Systemtabelle **MSysObjects** als Teil der **Connect**-Zeichenfolge angegeben.

gibt eine entsprechende Fehlermeldung aus. Es gibt auch keine einfache Möglichkeit, den Speicherort der verknüpften Datei an die veränderten Gegebenheiten anzupassen – Sie müssen die Verknüpfung erneuern.

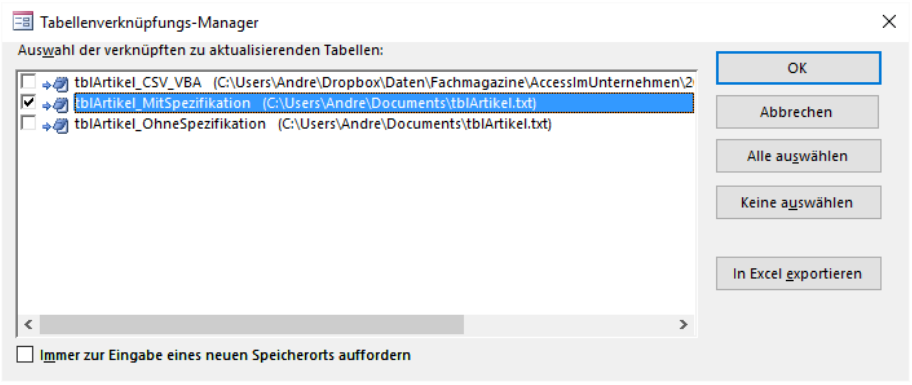

Bild 6: Der Tabellenverknüpfungs-Manager

Dazu können Sie diese entweder löschen und komplett neu erstellen oder Sie nutzen den **Tabellenverknüpfungs-Manager**. Diesen öffnen Sie über den Ribbon-Eintrag **Externe Daten|Importieren und Verknüpfen|Verknüpfungs-Manager**. Hier wählen Sie die zu aktualisierende Verknüpfung aus (s. Bild 6).

Nach einem Klick auf die Schaltfläche **OK** bietet Access mit einem **Datei auswählen**-Dialog die Möglichkeit, die Quelldatei der Verknüpfung erneut auszuwählen (s. Bild 7). Nach einer Erfolgsmeldung können Sie nun wieder auf die aktualisierte Verknüpfung zugreifen.

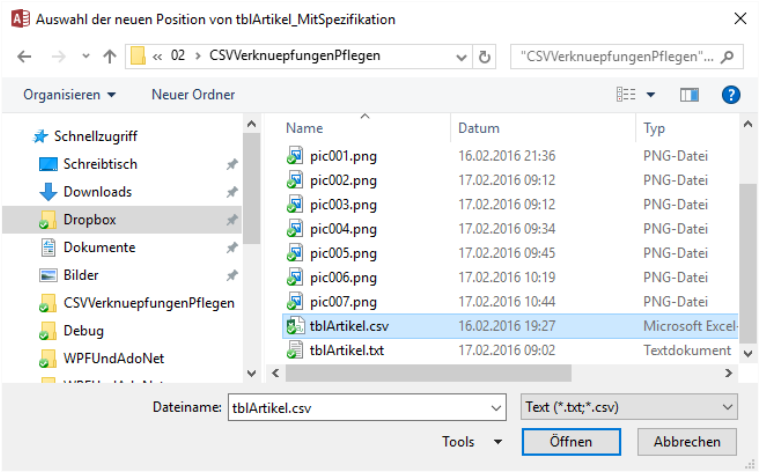

Bild 7: Auswahl des neuen Speicherorts der Quelldatei einer Verknüpfung

Dies funktioniert übrigens nur, wenn sich der Speicherort der Datei geändert hat. Sollten Sie den Dateinamen oder die Dateiendung angepasst haben, können Sie mit der bereits angelegten Verknüpfung nicht mehr auf diese Datei zugreifen – der Dateiname ist fest verankert, lediglich der Pfad ist änderbar.

Übrigens: Die Option **Immer zur Eingabe eines neuen Speicherort auffordern** im **Tabellenverknüpfungs-Manager** ist etwas ungeschickt benannt: Man könnte meinen, dass bei folgenden Problemen, die durch das Fehlen der Datei am angegebenen Ort auftreten, direkt ein **Datei auswählen**-Dialog erscheint.

Das ist aber nicht so: Die Option ist vielmehr so gemeint, dass Sie, wenn Sie mehr als eine Verknüpfung aktualisieren wollen, für jede einzelne den Speicherort angeben müssen.

Wie auch immer: Sie möchten die Verknüpfung, wenn Sie beispielsweise abwechselnd an Ihrem Arbeitsplatz oder am heimischen Rechner eine Anwendung entwickeln, nicht immer manuell aktualisieren. Das Gleiche gilt natürlich für den Kunden, der meist originelle Ideen hat, um den Speicherort der verknüpften Dateien zu variieren.

Außerdem wollen wir so flexibel sein, auch einmal eine Datei mit einer anderen Bezeichnung als beim erstmaligen Erstellen der Verknüpfung angegeben zu verwenden. Also prüfen wir nun die Möglichkeiten, die Verknüpfung möglichst automatisch zu aktualisieren.

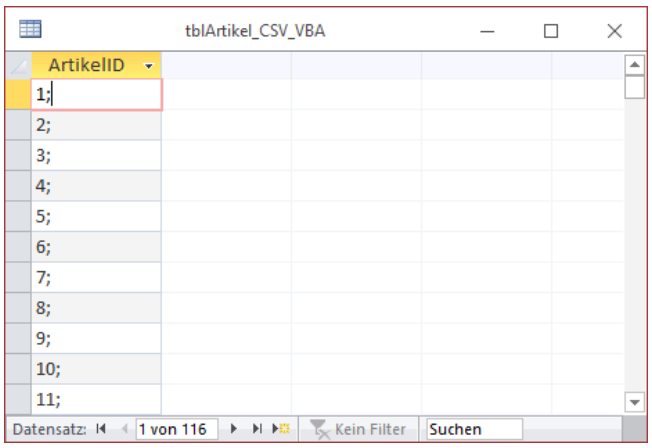

Bild 8: Das Ergebnis eines Imports ohne Textdateispezifikation

Verknüpfung per VBA herstellen

Nun schauen wir uns an, wie wir die Verknüpfung per VBA herstellen und ob auch hier die Probleme mit der Textdateispezifikation auftreten. Dazu nutzen wir den VBA-Befehl **DoCmd.TransferText**.

Der erste Aufruf sieht wie folgt aus:

```
DoCmd.TransferText acLinkDelim, , "tblArtikel_CSV_VBA",
CurrentProject.Path & "\tblArtikel.csv", True
```

Die Parameter haben dabei die folgenden Bedeutung:

- **TransferType**: Der Wert **acLinkDelim** legt fest, dass eine Verknüpfung einer Textdatei mit einem festgelegten Delimiter zum Einsatz kommt.
- **SpecificationName**: Bleibt im ersten Anlauf leer, da wir zunächst keine Textdateispezifikation verwenden möchten.
- **TableName**: Legt den Namen der Tabelle fest, hier **"tblArtikel_CSV_VBA"**
- **FileName**: Gibt den Dateinamen der zu verknüpfenden Datei an, in diesem Fall mit **CurrentProject.Path & "\tblArtikel.csv"** die Datei **tblArtikel.csv** im Verzeichnis der Datenbankdatei.
- **HasFieldNames**: Legt mit dem Wert **True** fest, dass die erste Zeile beim Verknüpfen als Spaltenüberschriften interpretiert wird und nicht als Werte.

Das Ergebnis in der Tabelle **MSysObjects** sieht zunächst gut aus: Hier finden wir nun keine Angabe einer Textdateispezifikation mehr unter dem Wert **DSN** des Feldes **Connect** vor:

```
Text;DSN=;FMT=Delimited;HDR=YES;IMEX=2;ACCDB=YES;
```

Dummerweise wirkt sich dies negativ auf die Erstellung der Verknüpfung aus. Wenn Sie den neuen Eintrag **tblArtikel_CSV_VBA** per Doppelklick öffnen, erscheint zwar keine Fehlermeldung, aber die angezeigte Tabelle liefert nur die erste Spalte mit den Werten des Feldes **ArtikelID** (s. Bild 8) – und diese auch noch jeweils mit abschließendem Semikolon.

Es sieht also so aus, als ob wir doch eine Textdateispezifikation angeben müssen. Da die Methode **TransferText** aber einen Parameter zur Übergabe des Namens einer solchen anbietet, probieren wir dies gleich einmal aus:

```
DoCmd.TransferText acLinkDelim, "tblArtikel_CSV", "tblArtikel_CSV_VBA_MitSpezifikation", CurrentProject.Path & "\tblArtikel.txt", True
```

Diese Variante gelingt auf Anhieb – die Verknüpfung landet unter dem Namen **tblArtikel_CSV_VBA_MitSpezifikation** im Navigationsbereich der Datenbank. Wir können die Arbeit, die wir vorher manuell durch Durchlaufen des Assistenten erledigt haben, auch per VBA ausführen. Mit einem Unterschied: Die Textdateispezifikation müssen wir zunächst einmal mit dem Assistenten erstellen.

Verknüpfung per VBA aktualisieren

Was geschieht nun mit der Verknüpfung, wenn wir diese erneut mit dem gleichen VBA-Befehl anlegen? In diesem Fall erstellt die Methode die gleiche Verknüpfung nochmals, aber behält die bereits vorhandene Verknüpfung bei.

Textfelder mit flexibler Höhe

Es ist seit jeher ein Wunsch der Access-Entwickler, Steuerelemente zum Beispiel dynamisch an die Größe des umgebenden Formulars anzupassen. Microsoft hat dem Rechnung getragen, indem es mit der Version 2007 die beiden Eigenschaften »Horizontaler Anker« und »Vertikaler Anker« für Steuerelemente hinzugefügt hat. Damit lässt sich allerdings immer nur ein Steuerelement in der horizontalen und der vertikalen Ebene in der Größe anpassen – doch was ist, wenn das Formular mehrere Steuerelemente besitzt, die abwechselnd größer dargestellt werden sollen? Dieser Beitrag zeigt, wie dies gelingt.

Textfelder verankern

Das Verankern von Textfeldern gibt es in Access ja schon immer – allerdings nur das Verankern am linken und oberen Rand. Seit Access 2007 können Sie in zwei speziellen Eigenschaften namens **Horizontaler Anker** und **Vertikaler Anker** die drei Werte **Links**, **Rechts** und **Beide** beziehungsweise **Oben**, **Unten** und **Beide** auswählen. Wenn Sie also etwa wünschen, dass ein Textfeld wie das aus Bild 1 beim Vergrößern des Formulars ebenfalls vergrößert wird, müssen Sie nur die beiden Eigenschaften auf den Wert **Beide** einstellen.

Wenn Sie das Formular dann allerdings ohne weitere Anpassungen vergrößern, treten die zwei Probleme wie in Bild 2 auf: Das Bezeichnungsfeld des Textfeldes wird nach rechts und nach unten verschoben und die Schaltfläche **OK** wird allmählich vom Textfeld überdeckt.

Es reicht also niemals aus, einfach nur die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** des betroffenen Steuerelements einzustellen. Für die Steuerelemente, die sich rechts oder unter dem zu vergrößernden Steuerelement befinden, müssen Sie die Eigenschaft **Horizontaler Anker** auf **Rechts** und die Eigenschaft **Vertikaler Anker** auf **Unten** einstellen.

Außerdem passt Access automatisch die Eigenschaften von Bezeichnungsfeldern an, sofern dem zu vergrößernde Steuerelement eines zugewiesen ist. In unserem Falle gehört ein Bezeichnungsfeld dazu, das für die Eigenschaft **Horizontaler Anker** nun den

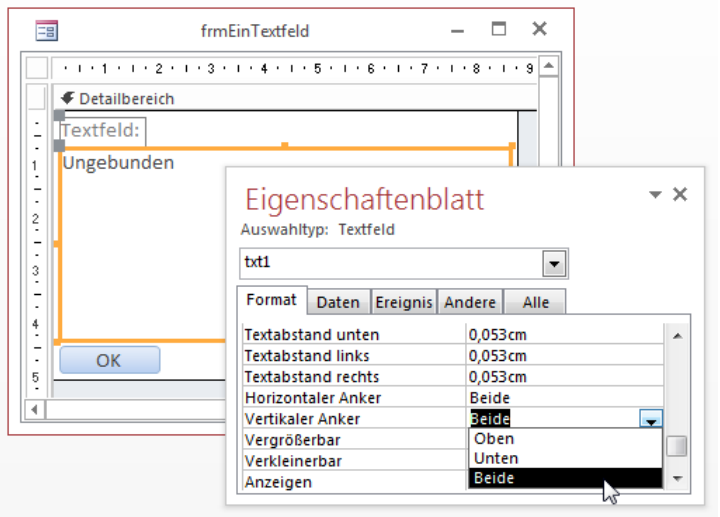

Bild 1: Einstellen der Anker für ein Textfeld

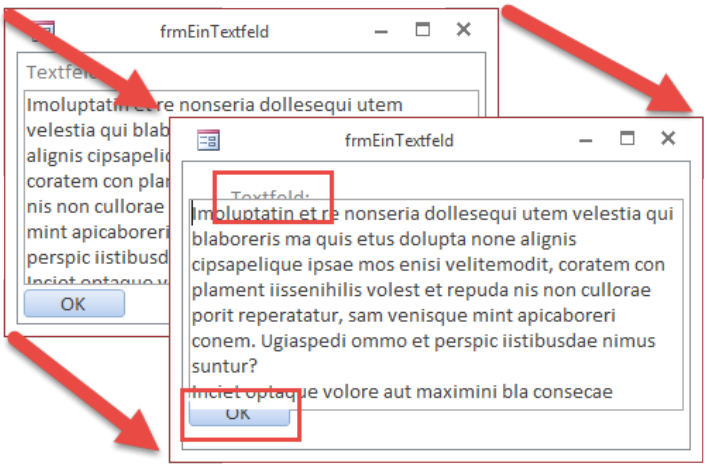

Bild 2: Probleme beim Vergrößern

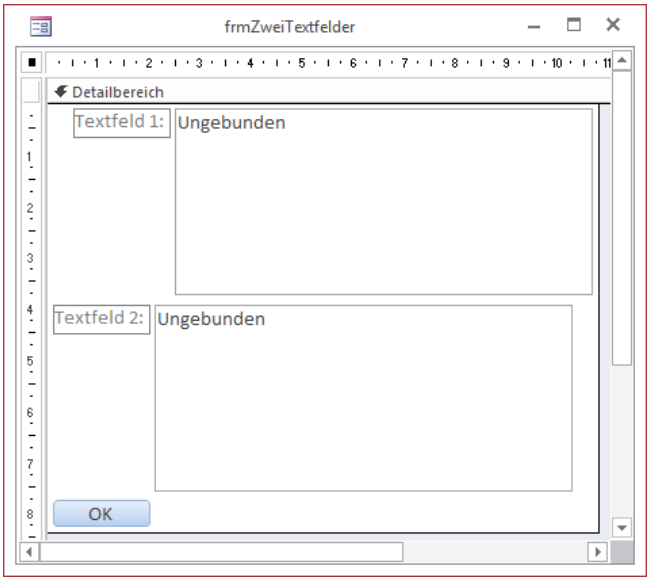

Bild 3: Zwei Textfelder mit gleichen Anker-Einstellungen

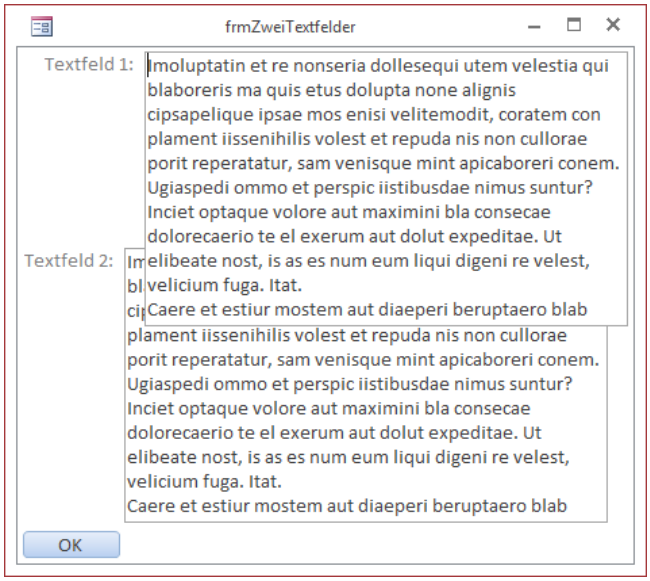

Bild 4: Beim Vergrößern überlappen sich die Textfelder.

Wert **Rechts** statt wie zuvor **Links** und für die Eigenschaft **Vertikaler Anker** nun den Wert **Unten** statt **Oben** aufweist.

Also müssen Sie sowohl die rechts und unter den zu vergrößernden Steuerelementen befindlichen Steuerelemente anpassen als auch die automatisch erfolgten Anpassungen der Bezeichnungsfelder der zu vergrößernden Steuerelemente wieder rückgängig machen.

Zwei Textfelder verankern

Wenn ein Formular zwei Textfelder enthält, deren Größe sich ändern soll, stellen wir einfach für beide die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** auf **Beide** ein (s. Bild 3).

Das Ergebnis ist allerdings nicht wie gewünscht: Beide Textfelder vergrößern sich zwar wie beabsichtigt nach rechts, aber nach unten erfolgt eine Überlappung – das obere Textfeld legt sich über das untere (s. Bild 4).

Damit steht fest: Wir können tatsächlich nur eine Reihe von Steuerelementen in der horizontalen und vertikalen Ebene vergrößern lassen. Zwei nebeneinander liegende Steuerelemente lassen sich ebenso wenig beide horizontal vergrößern wie zwei untereinander liegende Steuerelemente in der vertikalen Ebene.

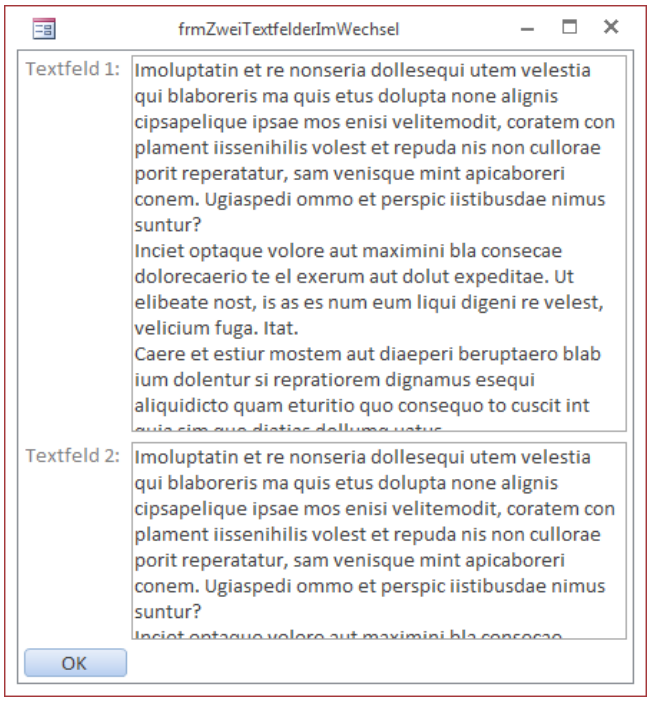

Bild 5: Wenn nur ein Textfeld oben und unten verankert wird, klappt es.

Markierung automatisch kopieren

Manchmal möchten Sie vielleicht schnell den kompletten Inhalt oder auch Ausschnitte von einem Textfeld in ein anderes übernehmen. Dann markieren Sie diesen, kopieren ihn, beispielsweise mit der Tastenkombinatino Strg + C, und fügen ihn mit Strg + V in das Zieltextfeld ein. Das ist für den Alltag schnell genug, so Sie denn die Tastenkombinationen aus dem Stegreif beherrschen (das ist längst nicht immer der Fall!). Wenn Sie jedoch häufiger Inhalte von Textfeld A nach Textfeld B übertragen wollen, gibt es einen viel eleganteren Weg. Schauen Sie selbst!

Mit dem Beispiel des vorliegenden Beitrags wollen wir zeigen, wie einfach es ist, den in einem ersten Textfeld markierten Text in ein zweites Textfeld zu kopieren. Dabei soll es zwei auslösende Ereignisse geben. Das erste und einfachere ist die Markierung mit der Maus. Diese beginnt, indem der Benutzer den Start der Markierung durch Herunterdrücken der Maustaste an der entsprechenden Stelle selektiert. Dann bewegt er den Mauszeiger bei gedrückter Maustaste zu der Stelle, die das Ende der Markierung festlegt. Um den Inhalt in das andere Textfeld zu kopieren, muss der Benutzer nun nur noch die Maustaste loslassen. Natürlich gelingt dies auch andersherum: Man kann auch zuerst das Ende der Markierung auswählen und dann weiter vorn an der Startposition die Maustaste loslassen.

Die zweite Variante soll die Markierung mit der Tastatur ermöglichen. Hier positioniert der Benutzer die Einfügemarke vor die erste Position des zu markierenden Textes – sei es mit der Maus oder mit den Cursortasten.

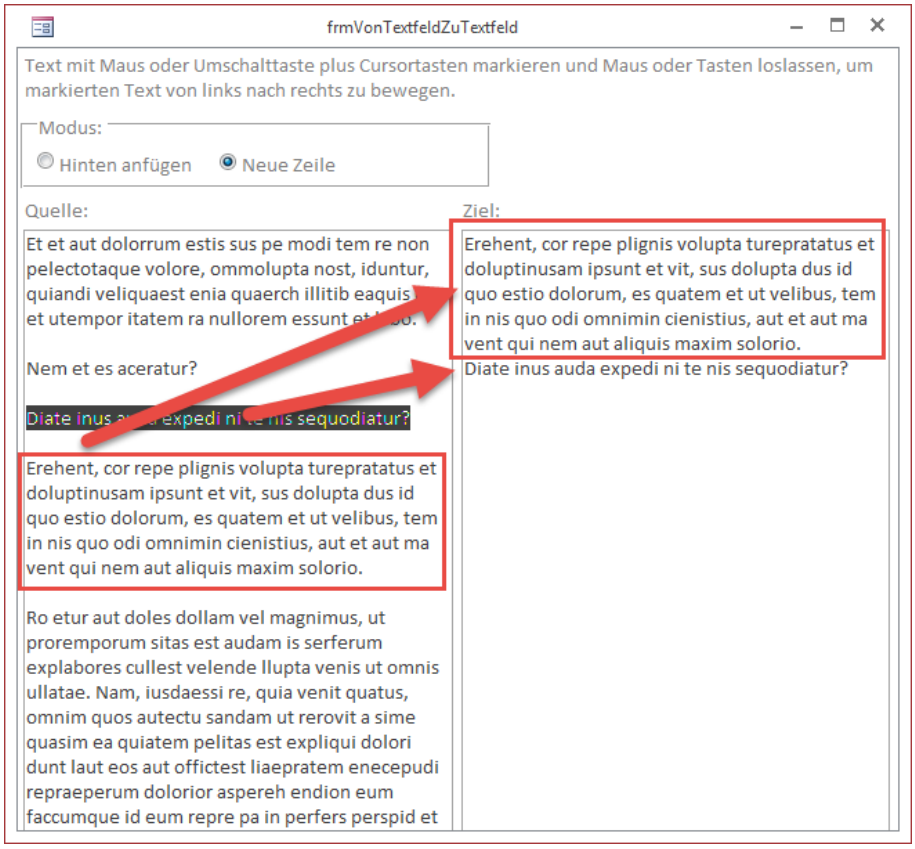

Bild 1: Kopieren von einem Textfeld zum nächsten

Dann erweitert er die Markierung durch Bewegung der Cursortasten bei gedrückter Umschalttaste (für zeichenweisen Vorschub) oder bei zusätzlich gedrückter **Strg**-Taste (um gleich ganze Wörter zur Auswahl hinzuzufügen). Wenn die gewünschte Auswahl erreicht wurde, lässt er einfach die Umschalttaste los und der markierte Text wird kopiert.

Dies sieht beispielsweise wie in Bild 1 aus. Hier haben wir zunächst den vierten Abschnitt des Quelltextes markiert und so zum Zielfeld hinzugefügt. Dann haben

wir den dritten Absatz markiert. Dadurch, dass die Option **Neue Zeile** ausgewählt ist, landet der markierte Text in der nächsten Zeile hinter dem bereits im Zielfeld vorhandenen Text.

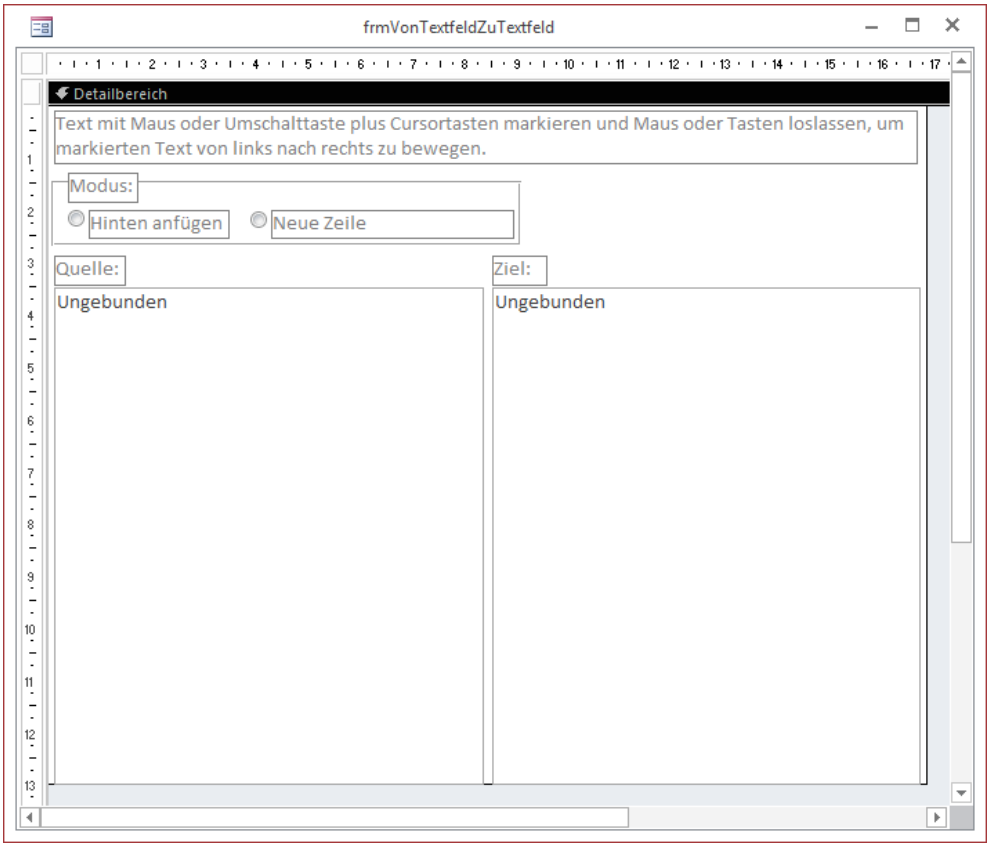

Bild 2: Entwurf des Formulars zum Kopieren von Inhalten per einfacher Markierung

Aufbau des Formulars

Der Versuchsaufbau sieht in der Entwurfsansicht wie in Bild 2 aus. Das Formular enthält hauptsächlich zwei Textfelder namens **txtQuelle** und **txtZiel**, von denen das Textfeld **txtQuelle** beim Laden Text anzeigen soll, der durch verschiedene Aktionen in das zweite Textfeld **txtZiel** kopiert werden soll.

Um genauer festzulegen, wie der Text an den bereits in **txtZiel** enthaltenen Text angefügt werden soll, gibt es noch eine Optionsgruppe namens **ogrModus**. Diese bietet zwei mögliche Werte an: **Hinten anfügen** oder **Neue Zeile**.

Damit unser Beispielformular beim Öffnen einige Beispielsätze anzeigt, füllen wir dieses in der Ereignisprozedur, die durch das Ereignis **Beim Laden** des Formulars ausgelöst wird, mit etwas Platzhaltertext.

Dann setzen wir den Fokus auf das Textfeld **txtQuelle** und heben eine eventuell durch die Einstellung für den Eintritt in das Textfeld vorgesehene Markierung auf, indem wir die Eigenschaft **SelLength** auf den Wert **0** einstellen:

```
Private Sub Form_Load()
    Dim strTemp As String
    strTemp = "Et et aut ..." & vbCrLf & vbCrLf
    strTemp = strTemp & "Diate inus ..." & vbCrLf & vbCrLf
    strTemp = strTemp & "Erehent, c..." & vbCrLf & vbCrLf
    strTemp = strTemp & "Ro etur aut..." & vbCrLf & vbCrLf
    strTemp = strTemp & "consed maio..."
    Me!txtQuelle = strTemp
    Me!txtQuelle.SetFocus
    Me!txtQuelle.SelLength = 0
End Sub
```

Kopieren per Tastatur

Das Kopieren nach der Markierung eines Bereichs unter Zuhilfenahme der Umschalttaste (zur Markierung einzelner Buchstaben) beziehungsweise der Umschalttaste plus der **Strg**-Taste (zur Markierung kompletter Wörter) erfolgt nach dem Loslassen der zuvor betätigten Tasten.

Dies löst das Ereignis **Bei Taste** auf des Textfeldes **txtQuelle** aus (s. Listing 1).

Standardwerte per bedingter Formatierung

Es geschieht gelegentlich, dass Sie einen Standardwert aus mehreren Werten festlegen müssen – beispielsweise, wenn ein Benutzer mehrere E-Mail-Adressen hat, Sie aber eine davon als Standardadresse für ausgehende Mails definieren möchten. Wenn die E-Mail-Adressen in einem Unterformular in der Datenblattansicht angezeigt werden, bietet sich die bedingte Formatierung an, um die jeweils aktive E-Mail-Adresse zu markieren. Wie dies aussehen kann, zeigt der vorliegende Beitrag.

Das Formular, das wir in diesem Beitrag beschreiben, soll am Ende so aussehen wie in Bild 1. Es zeigt im Hauptformular einen Datensatz der Tabelle **tblKunden** an. Das Unterformular steuert die Datensätze der Tabelle **tblEMailAdressen** bei, die über das Feld **KundeID** mit dem jeweiligen Kundendatensatz verknüpft sind. Das Formular soll folgende Aktionen unterstützen:

- Wenn der Benutzer doppelt auf eine E-Mail-Adresse klickt, soll diese als Standardadresse festgelegt und entsprechend farbig hinterlegt werden.
- Wenn der Benutzer einen neuen Kunden eingibt und dessen E-Mail-Adresse hinzugefügt hat, soll diese automatisch als Standard-E-Mail-Adresse markiert werden.
- Wenn der Benutzer eine E-Mail-Adresse eines Kunden löscht und es ist nur noch eine E-Mail-Adresse übrig, soll diese als Standard-E-Mail-Adresse markiert werden.
- Wenn der Benutzer die Standard-E-Mail-Adresse eines Kunden löscht und es ist noch mehr als eine E-Mail-Adresse vorhanden, soll die erste verfügbare E-Mail-Adresse als Standard gekennzeichnet werden – begleitet von einem Meldungsfenster, das den Benutzer darauf hinweist, welche E-Mail-Adresse nun als Standard festgelegt wurde.

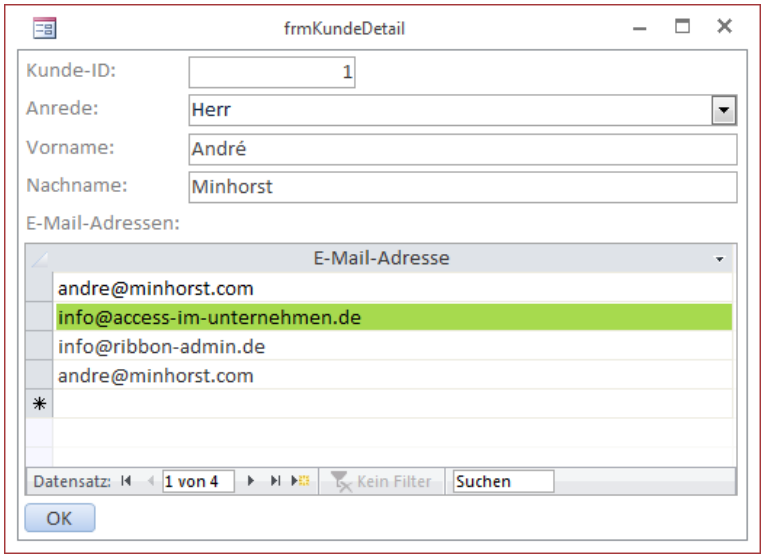

Bild 1: Formular mit markierter Standard-E-Mail-Adresse

Datenmodell

Die Beispieldatenbank enthält drei Tabellen, deren Felder und Beziehungen Sie Bild 2 entnehmen können. Die Tabelle **tblKunden** enthält einige wesentliche Kundendaten wie **AnredeID**, **Vorname** und **Nachname**. Das Feld **AnredeID** wird dabei aus der Tabelle **tblAnreden** gespeist, welche die passenden Anreden zu den Primärschlüsselwerten liefert. Die Tabelle **tblEMailAdressen** enthält neben dem Primärschlüsselfeld **EMailAdresseID** noch das Feld **EMailAdresse**, welche die eigentliche E-Mail-Adresse aufnimmt, das Feld **KundeID**, mit dem die Adresse einem Kunden zugewiesen wird und das Feld **Standard**, das die Standard-E-Mail-Adresse eines Kunden definiert.

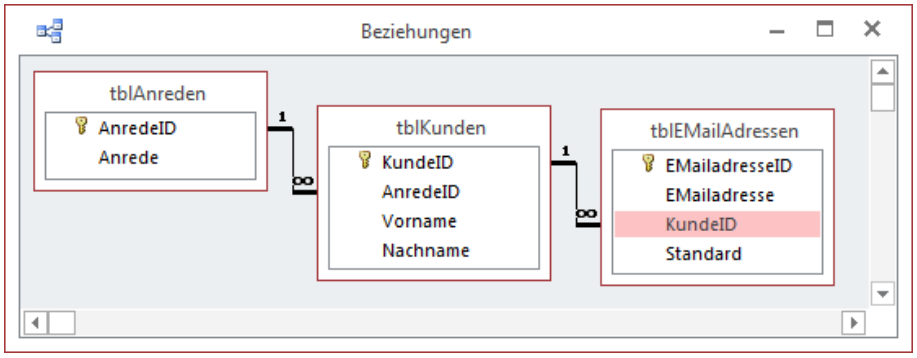

Bild 2: Das Datenmodell der Beispieldatenbank

Während man früher vielleicht nur eine E-Mail-Adresse je Kunde gespeichert hat, reicht dies heute nicht mehr aus – wenn Sie etwa eingehende E-Mails eines Kunden einem Kunden zuordnen wollen, müssen Sie davon ausgehen, dass dieser nicht immer die gleiche Absender-E-Mail-Adresse verwendet. Also benötigen wir eine Möglichkeit, je Kunde beliebig viele E-Mail-Adressen zu speichern – ohne außer Acht zu lassen, dass wir möglichst immer die gleiche E-Mail-Adresse für eigene Anschreiben nutzen wollen (außer natürlich bei Antworten an den Kunden – die sollten möglichst an die Absenderadresse geschickt werden).

Wie auch immer: Die einzige Möglichkeit, einem Kunden beliebig viele E-Mail-Adressen zuzuweisen, besteht in einer eigenen Tabelle für diesen Zweck. Dabei muss diese, wie unsere Tabelle **tblEMailAdressen**, mit dem eigentlichen Kundendatensatz aus der Tabelle **tblKunden** verknüpft sein.

Um nun noch festlegen zu können, welche der E-Mail-Adressen für eigene Anschreiben verwendet werden soll, nutzen wir ein **Ja/Nein**-Feld namens **Standard**. Dieses sollte natürlich für die E-Mail-Adressen zu einem Kundendatensatz nur jeweils einmal mit dem Wert **Ja** gefüllt sein.

Dies regeln wir in der vorliegenden Lösung allein über die Benutzeroberfläche, aber es ist auch möglich, dazu ein Datenmakro zu nutzen. Dies ist jedoch ein Thema für einen anderen Beitrag.

Beispielformulare

Wir benötigen insgesamt zwei Formulare. Das Hauptformular heißt **frmKundeDetail** und verwendet die Tabelle **tblKunden** als Datenherkunft. Da dieses Formular üblicherweise zum Anlegen oder zum Bearbeiten eines Kunden-Datensatzes geöffnet wird, verzichten wir auf die Darstellung von **Datensatzmarkierer**, **Navigationsschaltflächen**, **Bildlaufleisten** und **Trennlinien**, indem wir die gleichnamigen Eigenschaften auf den Wert **Nein** einstellen. Dafür weisen wir der Eigenschaft **Automatisch zentrieren** den Wert **Ja** zu.

Ziehen Sie dann die vier Felder **KundeID**, **AnredeID**, **Vorname** und **Nachname** aus der Feldliste in den Formularentwurf und ändern Sie die Beschriftungen und die Anordnung der Steuerelemente etwa wie in Bild 3.

Nun kümmern wir uns um das Unterformular. Dieses soll den Namen **sfmEMailAdressen** erhalten und die Tabelle **tblEMailAdressen** als Datenherkunft nutzen. Ziehen Sie die Felder **EMailAdresse** sowie **Standard** in den Detailbereich des Formularentwurfs (s. Bild 4). Die Anordnung spielt in diesem Fall keine Rolle, da wir die E-

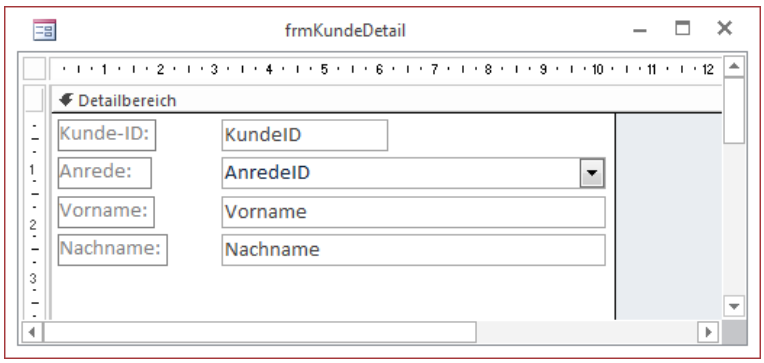

Bild 3: Erster Entwurf des Formulars **frmKundeDetail**

Mail-Adressen ohnehin in der Datenblattansicht darstellen wollen. Allerdings sollten Sie die Beschriftung des Feldes **EMailAdresse** noch in **E-Mail-Adresse:** ändern, da diese ja als Spaltenüberschrift für die Datenblattansicht zum Einsatz kommt. Damit das Formular in der Datenblattansicht angezeigt wird, stellen Sie noch die Eigenschaft **Standardansicht** auf **Datenblatt** ein.

Außerdem stellen Sie den Namen des Kontrollkästchens, das an das Feld **Standard** gebunden ist, auf **chkStandard** ein. Wozu dies nötig ist, erfahren Sie weiter unten.

Nun schließen Sie das Formular **sfmEMailAdressen** und ziehen es aus dem Navigationsbereich heraus in den Entwurf des Formulars **frmKundeDetail** hinein. Nach der Anpassung der Breite des Unterformulars an die Breite der darüber befindlichen Steuerelemente sieht der Entwurf nun etwa wie in Bild 5 aus.

Hier erkennen Sie auch bereits die Schaltfläche **cmdOK**, mit deren Hilfe der Benutzer das Formular schließen kann. Es löst die folgende Ereignisprozedur aus:

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

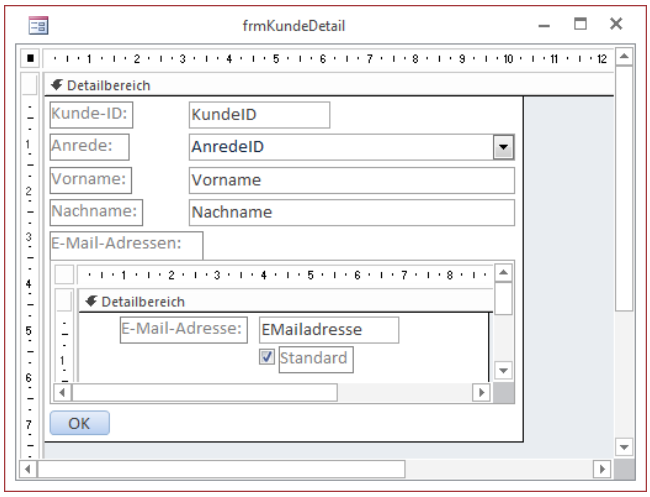

Bild 5: Haupt- und Unterformular in der Entwurfsansicht

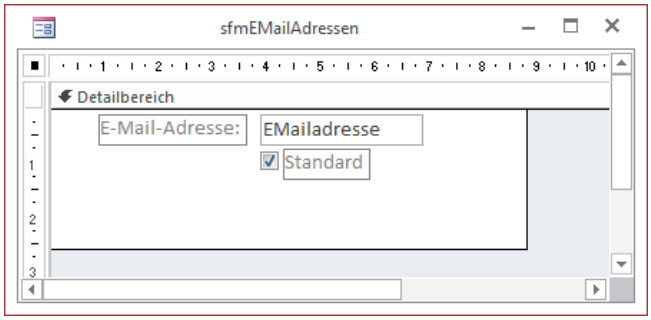

Bild 4: Entwurfsansicht des Formulars **sfmEMailAdressen**

Verankern

Das Unterformular hat zu Beginn eine Höhe, welche die Anzeige von etwa drei E-Mail-Adressen zulässt – was bei den meisten Kunden ausreichend sein sollte. Hat dennoch einmal ein Kunde mehr E-Mail-Adressen, soll der Benutzer diese auf einen Blick einsehen können, ohne durch die Datensätze des Unterformulars scrollen zu müssen. Er soll dafür einfach nur das Formular vergrößern, während sich die Höhe des Unterformulars entsprechend anpasst. Dazu stellen Sie die Eigenschaft **Vertikaler Anker** des Unterformular-Steuerelements auf **Beide** ein. Beachten Sie, dass sich damit die Eigenschaft **Vertikaler Anker** des Bezeichnungsfeldes des Unterformulars automatisch auf **Unten** ändert. Dies sollten Sie wieder auf **Oben** einstellen, damit das Bezeichnungsfeld beim Vergrößern nicht hinter dem Unterformular verschwindet.

Datenblatt optimieren

Nun müssen wir noch die Ansicht des Datenblatts optimieren. Wenn Sie nämlich nun in die Formularansicht des Hauptformulars wechseln, werden die E-Mail-Adressen nicht in der optimalen Spaltenbreite angezeigt. Außerdem erscheint hier auch noch das Feld **Standard**, das wir eigentlich nicht sehen wollen – schließlich soll die aktuelle Standard-E-Mail-Adresse farbig hinterlegt werden.

Darum kümmern wir uns in zwei Schritten: Als Erstes markieren Sie den Spaltenkopf der Spalte **Standard** und wählen nach einem Rechtsklick den Kontextmenü-Eintrag **Felder ausblenden** aus (s. Bild 6). Dies sorgt dafür, dass die Spalte **Standard** verschwindet. Anschließend ziehen

Sie die Spalte mit der Spaltenüberschrift **E-Mail-Adresse** auf die komplette Breite des Unterformulars.

Die Ansicht ist nun in Ordnung, also können wir uns an die wichtigste Aufgabe begeben: Der aktuelle Standarddatensatz im Unterformular soll farbig hervorgehoben werden und wir wollen per Code die eingangs erwähnten Funktionen umsetzen, die für das Ändern des Standarddatensatzes nötig sind.

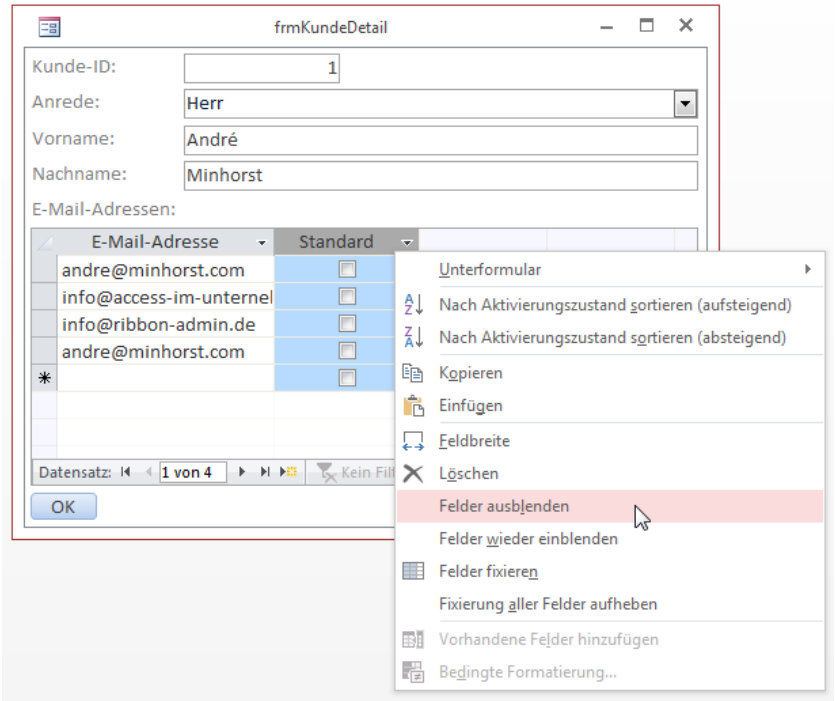

Bild 6: Die Spalten des Unterformulars müssen noch angepasst werden.

Aktive Standard-E-Mail-Adresse markieren

Hier kommt die bedingte Formatierung ins Spiel. Wir wollen dafür sorgen, dass das Feld **EMailAdresse** für den Datensatz, dessen Feld **Standard** den Wert **Wahr** (**True**/**-1**) enthält, farbig hinterlegt wird.

Dazu öffnen Sie das Formular in der Formularansicht und klicken auf das mit der bedingten Formatierung auszustattende Textfeld. Wählen Sie dann im Ribbon den Eintrag **Datenblatt|Formatierung|Bedingte Formatierung** aus. Es erscheint der Dialog aus Bild 7. Hier klicken Sie nun auf die Schaltfläche **Neue Regel**, um eine neue bedingte Formatierung zu erstellen.

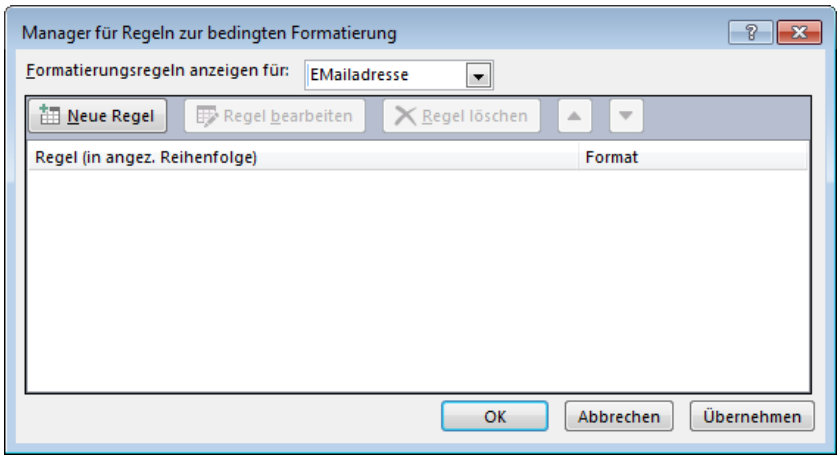

Bild 7: Anlegen einer neuen bedingten Formatierung

Nun erscheint der Dialog **Neue Formatierungsregel** (s. Bild 8). Hier ändern Sie nun zunächst den Wert des Kombinationsfeldes unter dem Text **Nur Zellen formatieren, für die gilt:** von **Feldwert ist** in **Ausdruck ist**. Die bedingte Formatierung soll angewendet werden, wenn das Feld **Standard** des aktuellen Datensatzes den Wert **True** enthält. Dies lässt sich leider nicht direkt mit **[Standard] = True** formulieren, weshalb wir weiter oben den Namen des Kontrollkästchens auf **chkStandard** geändert haben. Erst dann funktionierte die bedingte Formatierung mit dem Ausdruck **[chkStandard] = True** (interessanterweise konnten wir bei Experimenten danach den Namen des Kontrollkästchens wieder von **chkStandard** in **Standard** zurückändern, während die bedingte Formatierung

HTML-Liste mit Access-Daten

Die Datenblatt-Ansicht und das Endlosformular sind gute Helfer, wenn es um die Anzeige mehrerer Datensätze einer Tabelle oder Abfrage geht. Allerdings haben beide einen gravierenden Nachteil: Sie zeigen alle Datensätze immer in der gleichen Höhe an. Wenn also ein Datensatz etwa eine Notiz enthält, die nur eine Zeile lang ist, eine andere aber zehn Zeilen, müssen Sie die Formularhöhe nach dem längsten Inhalt auslegen. Dadurch verlieren Sie wiederum eine Menge Platz, da die Höhe ja auch für kurze Inhalte beansprucht wird. Dieser Beitrag zeigt, wie Sie das Problem mithilfe von HTML und dem Webbrowser-Steuerelement beheben.

Beispieltabelle

Als Beispiel für die Anzeige und Bearbeitung von Daten in einem Webbrowser-Steuerelement verwenden wir die Tabelle **tblNotizen**. Diese enthält neben dem Primärschlüsselfeld **NotizID** noch die beiden Felder **Notiz** mit dem eigentlichen Inhalt (ausgelegt als Memo-Feld, also mit ausreichend Platz) sowie ein Feld namens **AngelegtAm**.

Webbrowser-Steuerelement einbauen

Das Formular mit dem ersten Beispiel soll **frmNotizen** heißen. Dass das Formular seine Daten in einem Webbrowser-Steuerelement anzeigen soll und keine eigenen Datensätze, können Sie die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**, **Bildlaufleisten** und **Trennlinien** auf den Wert **Nein** einstellen.

Fügen Sie dem Formular dann das Webbrowser-Steuerelement hinzu und nennen Sie es **ctlWebbrowser** (s. Bild 1).

Stellen Sie seine Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** jeweils auf den Wet **Beide** ein. Auf diese Weise passt es seine Größe automatisch an die Größe des Formulars an.

Bevor wir richtig loslegen, fügen Sie dem VBA-Projekt der Datenbank noch einen Verweis auf die Bibliothek **Microsoft HTML Object Library** hinzu (s. Bild 2). Den **Verweise**-Dialog des VBA-Editors öffnen Sie mit dem Menübefehl **Extras|Verweise**.

Diese Bibliothek steuert die Elemente für die VBA-Programmierung des HTML-Dokuments bei – wir können damit vollständige Internetseiten nicht nur definieren, sondern auch steuern, also beispielsweise auf die Ereignisse der einzelnen Elemente der HTML-Seite reagieren. Dies ist

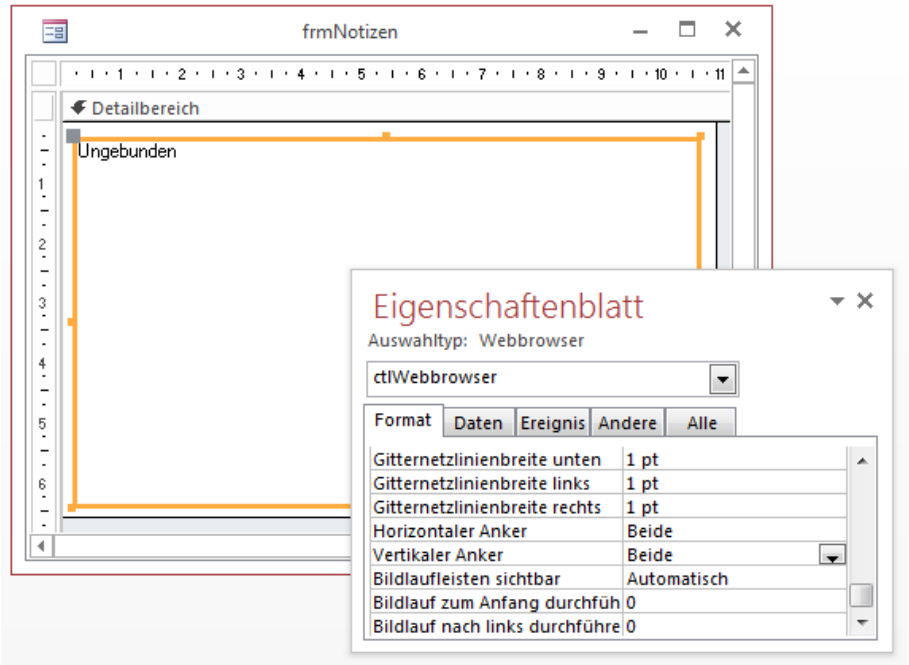

Bild 1: Einfügen und Anpassen des Webbrowser-Steuerelements

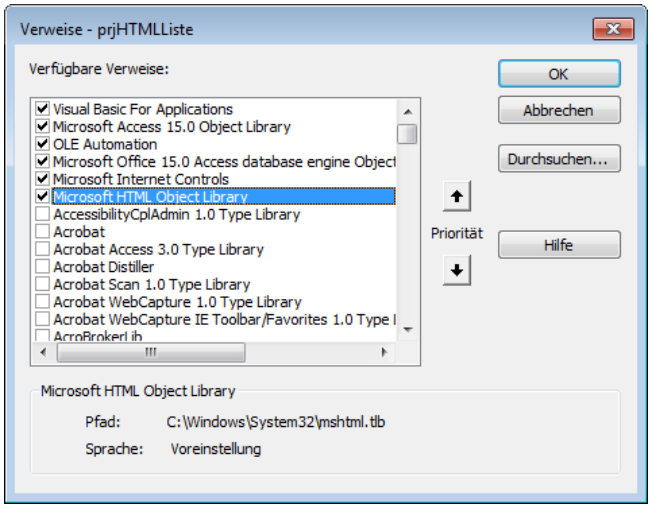

Bild 2: Zusätzlicher Verweis auf die Bibliothek **Microsoft HTML Object Library**

wichtig, wenn wir etwa durch einen Mausklick auf einen Datensatz ein Detailformular zum Bearbeiten der Daten dieses Datensatzes aufrufen möchten.

Bevor wir mit der Programmierung des Formulars und des Webbrowser-Steuerelements beginnen, wollen wir noch die Tabelle **tblNotizen** mit einigen Datensätzen füllen. Die Tabelle sieht anschließend wie in Bild 3 aus.

Programmierung des Webbrowser-Steuerelements

Das Webbrowser-Steuerelement soll die Daten der Tabelle **tblNotizen** anzeigen und bei Betätigen der Taste **F5** wie ein herkömmliches Formular seine Daten aktualisieren. Wenn Sie also das Formular geöffnet haben und dann beispielsweise im Hintergrund einen Datensatz direkt in der Tabelle verändern, soll die Taste **F5** zur Anzeige der geänderten Daten im Webbrowser-Steuerelement führen.

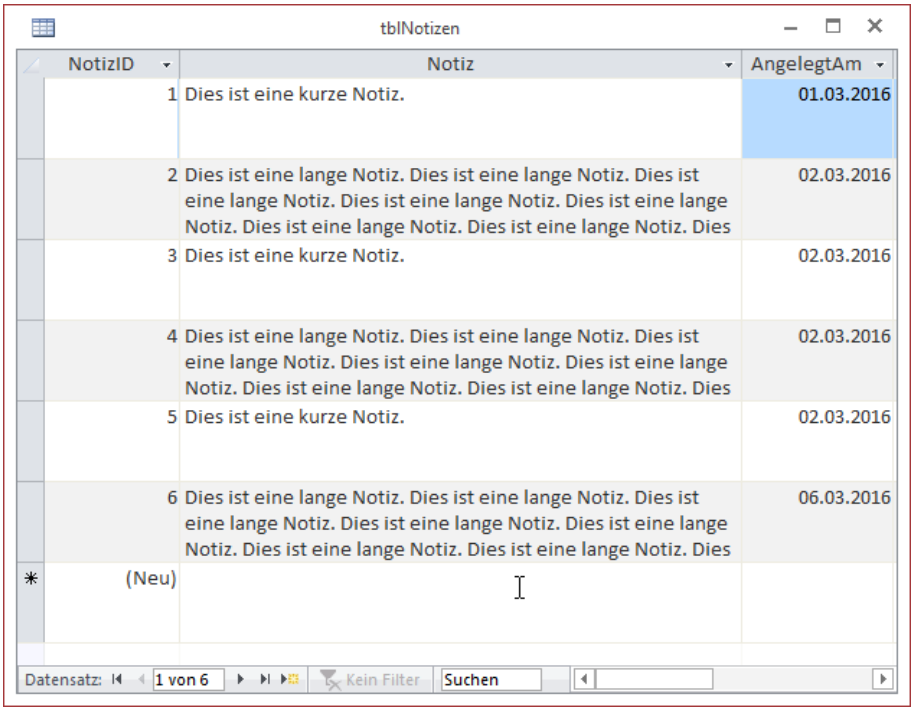

Bild 3: Beispieldaten für die Tabelle **tblNotizen**

Damit wir auf alle Ereignisse rund um das Webbrowser-Steuerelement und das enthaltene Dokument in Form der Implementierung geeigneter Ereignisprozeduren reagieren können, deklarieren wir sowohl für das Webbrowser-Steuerelement als auch für das darin angezeigte Dokument je eine Variable mit dem Schlüsselwort **WithEvents**:

```
Private WithEvents objWebbrowser As WebBrowser
Private WithEvents objDocument As MSHTML.HTMLDocument
```

Beim Laden des Formulars sind einige vorbereitende Aktionen durchzuführen. Diese implementieren wir in Form einer Ereignisprozedur, die durch das Ereignis **Beim Laden** des Formulars ausgelöst wird. Diese Prozedur sieht wie folgt aus:

```
Private Sub Form_Load()
    Set objWebbrowser = Me!ctlWebbrowser.Object
    WebbrowserLeeren
    objWebbrowser.Document.Write "<p></p>"
End Sub
```

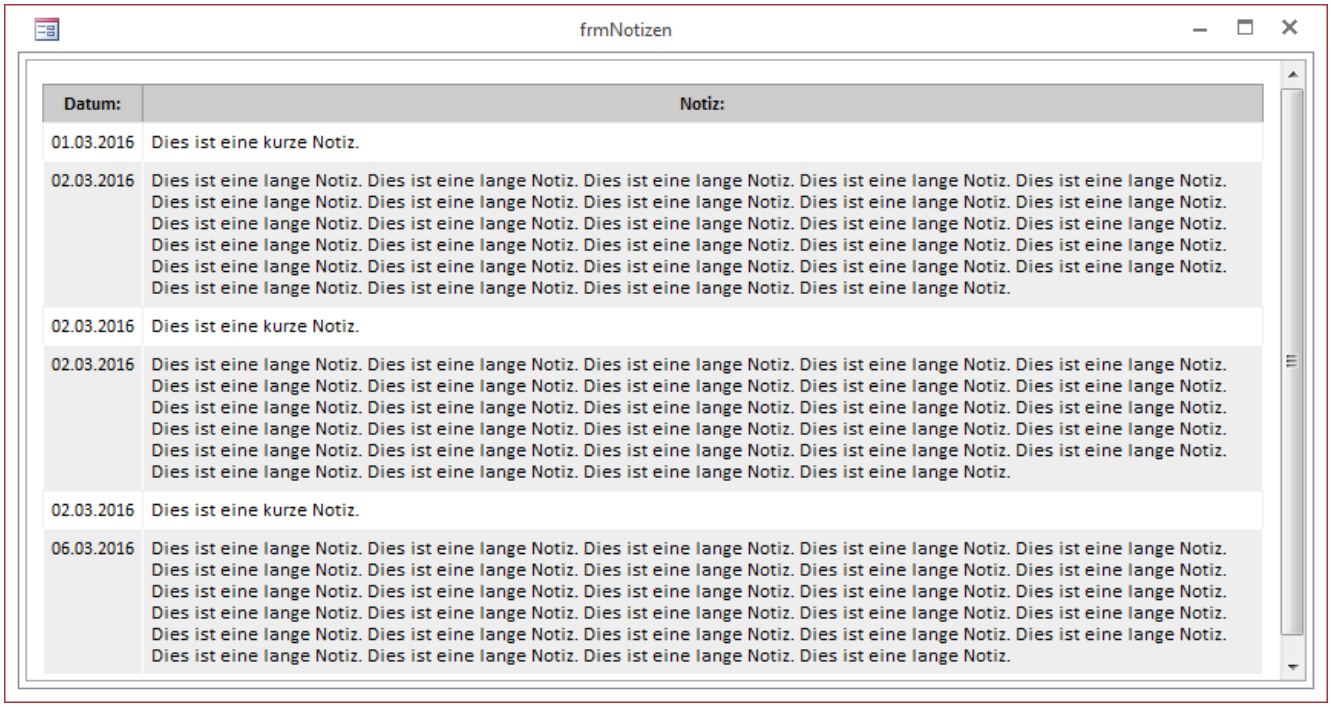

Bild 4: Anzeige der Daten im Webbrowser-Steuerelement

Die erste Anweisung setzt die Objektvariable **objWebbrowser** auf das im Webbrowser-Steuerelement enthaltene Objekt. Die zweite ruft eine Prozedur namens **WebbrowserLeeren** auf, die das Steuerelement mit einer leeren HTML-Seite füllt und erst zur aufrufenden Prozedur zurückwechselt, wenn dieser Vorgang ausgeführt wurde. Dazu durchläuft die Prozedur eine **Do While**-Schleife, deren Abbruchbedingung die Eigenschaft **ReadyState** des Webbrowser-Steuerelements mit dem Wert **READYSTATE_COMPLETE** vergleicht:

```
Private Sub WebbrowserLeeren()
    objWebbrowser.Navigate "about:blank"
    Do While Not objWebbrowser.ReadyState = _
            READYSTATE_COMPLETE
        DoEvents
    Loop
End Sub
```

Diese Bedingung wird erfüllt, wenn die Seite von der angegebenen Adresse, hier **about:blank** für eine leere Seite, vollständig geladen wurde. Danach fügt die Prozedur **Form_Load** noch eine einzelne, leere Zeile zum Dokument hinzu (**objWebbrowser.Document.Write " "**).

Daten im Webbrowser-Steuerelement anzeigen

Das waren bereits die Vorbereitungen und wir können mit dem Füllen der Seite mit den eigentlichen Inhalten beginnen, damit diese wie in Bild 4 im Webbrowser-Steuerelement erscheinen. Aber wann sollen diese nun geladen werden – und welches Ereignis löst diesen Vorgang aus? Da wir, wie weiter oben beschrieben, den Inhalt des Webbrowser-Steuerelements beim Betätigen von **F5** ebenfalls aktualisieren wollen, wäre die durch das Ereignis **Beim Anzeigen** ausgelöste Prozedur der richtige Ort. Also fügen wir dort den Aufruf der Routine zum Anzeigen der Daten ein:

```
Private Sub Form_Current()
    DatenAnzeigen
    Debug.Print objDocument.documentElement.innerHTML
End Sub
```

```
Private Sub DatenAnzeigen()
    Dim objTable As MSHTML.HTMLTable
    Dim objRow As MSHTML.HTMLTableRow
    Dim objCell As MSHTML.HTMLTableCell
    Dim strBackgroundColor As String
    Dim strBorderColor As String
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set objWebbrowser = Me!ctlWebbrowser.Object
    Set objDocument = objWebbrowser.Document
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblNotizen", dbOpenDynaset)
    WebbrowserLeeren
    objWebbrowser.Document.Clear
    Set objTable = objDocument.createElement("Table")
    objDocument.Body.appendChild objTable
    objTable.Style.borderCollapse = "collapse"
    Set objRow = objTable.insertRow
    Set objCell = objRow.insertCell
    With objCell
        FormatHeader objCell
        .innerText = "Datum:"
    End With
    Set objCell = objRow.insertCell
    With objCell
        FormatHeader objCell
        .innerText = "Notiz:"
    End With
    Do While Not rst.EOF
        If rst.AbsolutePosition Mod 2 = 0 Then
            strBackgroundColor = "#FFFFFF"
            strBorderColor = "#EEEEEE"
        Else
            strBackgroundColor = "EEEEEE"
            strBorderColor = "#FFFFFF"
        End If
        Set objRow = objTable.insertRow
        Set objCell = objRow.insertCell
        With objCell
            FormatCell objCell, strBackgroundColor, strBorderColor
            .ID = rst!NotizID
            .innerText = rst!AngelegtAm
        End With
        Set objCell = objRow.insertCell
        With objCell
            FormatCell objCell, strBackgroundColor, strBorderColor
            .ID = rst!NotizID
            .innerText = rst!Notiz
        End With
        rst.MoveNext
    Loop
End Sub
```

Listing 1: Füllen des Webbrowser-Steuerelements mit den Daten der Tabelle **tblNotizen**

Außerdem hinterlegen wir dort noch einen Befehl, der nach Füllen des Webbrowser-Steuerelements einmal den HTML-Inhalt des angezeigten Dokuments im Direktfenster ausgibt – einfach, damit Sie einmal kontrollieren können, wie der erzeugte Code aussieht. Mit dem Kontextmenübefehl **Quellcode anzeigen** des Webbrowser-Steuerelements erhalten Sie nämlich lediglich den Text **<html></html>**, was wenig hilfreich ist.

Die Prozedur **DatenAnzeigen** finden Sie in Listing 1. Sie deklariert zunächste einige Elemente, welche die im Webbrowser-Steuerelement angezeigten Objekte repräsentieren. Als Erstes benötigen wir, um die Struktur einer Tabelle abzubilden, ein Objekt des Typs **HTMLTable**, das wir mit der Variablen **objTable** deklarieren. Außerdem benötigen wir ein **HTMLTableRow**-Objekt (**objRow**) und ein **HTMLTableCell**-Objekt (**objCell**).

Daneben müssen wir natürlich auf die aktuelle Datenbank sowie die Tabelle **tblNotizen** zugreifen, daher deklarieren wir ein entsprechendes Objekt des Typs **Database** (**db**) und eines des Typs **Recordset** (**rst**).

Das Webbrowser-Steuerelement referenzieren wir hier nur aus dem Grund erneut mit der Vari-

ablen **objWebbrowser**, weil sich dies bei Experimenten und den dabei auftretenden unbehandelten Fehlern schon einmal leert und dies dann zu einem weiteren Fehler führt. Dann füllen wir die Variable **objDocument** mit einem Verweis auf das im Webbrowser angezeigte Dokument (**objWebbrowser.Document**).

Die Vorbereitungen für den Zugriff auf die Tabelle **tblNotizen** enthalten das Füllen der Variablen **db** mit dem **Database**-Objekt der aktuellen Datenbank mit **CurrentDb**, und die **OpenRecordset**-Methode versieht die Variable **rst** mit einem Recordset auf Basis der Tabelle **tblNotizen**.

Danach ruft die Prozedur erneut die Routine **WebbrowserLeeren** auf und leert auch noch das **Document**-Objekt mit der **Clear**-Methode.

Schließlich beginnt das eigentliche Füllen des Dokuments, zunächst mit einem **Table**-Element. Das Element erstellt die Prozedur mit der **createElement**-Methode und dem Namen des zu erstellenden Objekts als Parameter (Table). Dieses Element fügt die **appendChild**-Methode des **Body**-Elements des Dokuments dann in den Body des Dokuments ein. Der HTML-Code sieht an dieser Stelle wie folgt aus:

```
<HEAD></HEAD>
<BODY>
    <TABLE></TABLE>
</BODY>
```

Nun stellen wir die Eigenschaft **Style.borderCollapse** auf **collapse** ein, was bewirkt, dass die Rahmen der verschiedenen Zellen der Tabelle miteinander verschmelzen und nicht jede Zelle einen eigenen Rahmen erhält.

Nun kommt die Prozedur zu den Zeilen und Spalten, zuerst mit der Kopfzeile, welche die Spaltenüberschriften enthalten und anders formatiert sein soll. Diese fügen wir mit der **insertRow**-Methode des **HTMLTable**-Objekts aus **objTable** ein und referenzieren es mit der Variablen **objRow**. Wir benötigen zwei Spalten, eine für das Feld **AngelegtAm** und eine für das Feld **Notiz**. Die Zelle der Kopfzeile für das Feld **AngelegtAm** fügt die Prozedur mit der Methode **insertCell** des soeben erzeugten Elements **objRow** an und speichert den Verweis darauf in der Variablen **objCell**. Für dieses Objekt rufen wir einmal die Routine **FormatHeader** auf und übergeben dieser den Verweis auf die Zelle, dann fügen wir der Eigenschaft **innerText** den Ausdruck **Datum:** hinzu, also den Text für die Spaltenüberschrift.

Die Routine **FormatHeader** nimmt den Verweis auf die zu formatierende Zelle entgegen und weist dann verschiedenen Eigenschaften des **Style**-Objekts der Zelle die gewünschten Werte zu. **BorderColor** und **BackgroundColor** erhalten die Schrift- und die Hintergrundfarbe, **BorderWidth** die Rahmendicke, **BorderStyle** die Rahmenart (hier **solid** für einen durchgezogenen Rahmen).

Die horizontale Ausrichtung stellt die Eigenschaft **TextAlign** mit **center** als zentriert ein, die vertikale Ausrichtung übernimmt die Eigenschaft **verticalAlign** mit dem Wert **top**. **FontSize**, **FontFamily** und **FontWeight** definieren das Aussehen der Schrift. **Padding** legt schließlich fest, dass der Text einen Abstand von **5** zum linken, oberen, rechten und unteren Rand erhalten soll:

```
Private Sub FormatHeader(objCell As HTMLTableCell)
    With objCell.Style
        .BorderColor = "#999999"
        .BackgroundColor = "#CCCCCC"
        .BorderWidth = "1px"
        .BorderStyle = "solid"
        .TextAlign = "center"
        .VerticalAlign = "top"
        .FontSize = "9pt"
        .FontFamily = "Calibri"
        .FontWeight = "bold"
        .Padding = "5,5,5,5"
    End With
End Sub
```

Die Prozedur **DatenAnzeigen** führt dann den gleichen Vorgang für die Spaltenüberschrift mit dem Text **Notiz:** durch und ruft auch hier wieder die Routine **FormatHeader** auf.

Danach folgen die eigentlichen Daten, welche die Prozedur in einer **Do While**-Schleife über alle Datensätze des Recordsets **rst** durchläuft. Dabei trifft die Prozedur zunächst die Vorbereitung für die Anzeige der einzelnen Zeilen mit wechselnden Hintergrundfarben. Dies betrifft nicht nur die Hintergrundfarbe, sondern auch die Rahmenfarbe, weshalb wir die zu verwendenden Farben in zwei Variablen namens **strBackgroundColor** und **strBorderColor** speichern. Hier stehen die beiden Farben **#FFFFFF** (weiß) und **#EEEEEE** (hellgrau) zur Auswahl, wobei diese wechselseitig zum Einsatz kommen sollen (also weißer Hintergrund/hellgrauer Rahmen und hellgrauer Hintergrund/weißer Rahmen).

```
Private Sub FormatCell(objCell As HTMLTableCell, Optional strBackgroundColor _
        As String, Optional strBorderColor As String)
    With objCell.Style
        If Len(strBackgroundColor) > 0 Then
            .backgroundColor = strBackgroundColor
        Else
            .backgroundColor = "#FFFFFF"
        End If
        If Len(strBorderColor) > 0 Then
            .BorderColor = strBorderColor
        Else
            .BorderColor = "#CCCCCC"
        End If
        .BorderWidth = "1px"
        .BorderStyle = "solid"
        .TextAlign = "left"
        .FontSize = "9pt"
        .verticalAlign = "top"
        .fontFamily = "Calibri"
        .padding = "5,5,5,5"
    End With
End Sub
```

Listing 2: Formatieren der Zellen mit den Daten der Tabelle **tblNotizen**

Ob die erste oder die zweite Variante gewählt werden soll, ermitteln wir mit dem Ausdruck **rst.AbsolutePosition Mod 2 = 0**. Dies ergibt bei geraden Werten für **rst.AbsolutePosition** (also der aktuellen Position des Datensatzzeigers mit dem Wert **0** für den ersten Datensatz) den Wert **True** und bei ungeraden Werten den Wert **False**, sodass die beiden Variablen **strBackgroundColor** und **strBorderColor** abwechselnd mit den Werten **#FFFFFF** und **#EEEEEE** gefüllt werden.

Nach dem Anlegen der Zeile für den aktuellen Datensatz mit der **insertRow**-Methode und dem Erstellen der Zelle für das Anlagedatum mit **insertCell** ruft die Prozedur diesmal die Routine **FormatCell** auf. Diese finden Sie in Listing 2. Die Routine erwartet nicht nur einen Verweis auf die zu formatierende Zelle, sondern optional noch Werte für die beiden Parameter **strBackgroundColor** und **strBorderColor**. Sie prüft zuerst, ob der Parameter **strBackgroundColor** gefüllt ist. Falls ja, erhält die Eigenschaft **BackgroundColor** den Wert dieses Parameters, anderenfalls den Standardwert **#FFFFFF**. Das Gleiche erledigt die Prozedur für den Parameter **strBorderColor** und die Eigenschaft **BorderColor** der Zelle. Schließlich stellt sie die übrigen Eigenschaften der Zelle ein, die Sie bereits weiter oben kennen gelernt haben (mit kleinen Unterschieden – zum Beispiel soll die Schrift nicht fett dargestellt werden, dafür aber links ausgerichtet).

Die Eigenschaft **innerText** füllt die Prozedur mit dem Wert des Feldes **AngelegtAm**. Außerdem schreibt sie den Primärschlüsselwert des aktuellen Datensatzes in die Eigenschaft **ID** der Zelle – dies ist eine vorbereitende Maßnahme für weitere Funktionen. Das Erstellen und

Füllen der Zelle mit dem Inhalt des Feldes **Notiz** verläuft analog zum Füllen der Zelle mit dem Anlagedatum. Diese Schritte wiederholt die Prozedur für alle Datensätze des Recordsets **rst** und füllt so die Tabelle mit den gewünschten Daten.

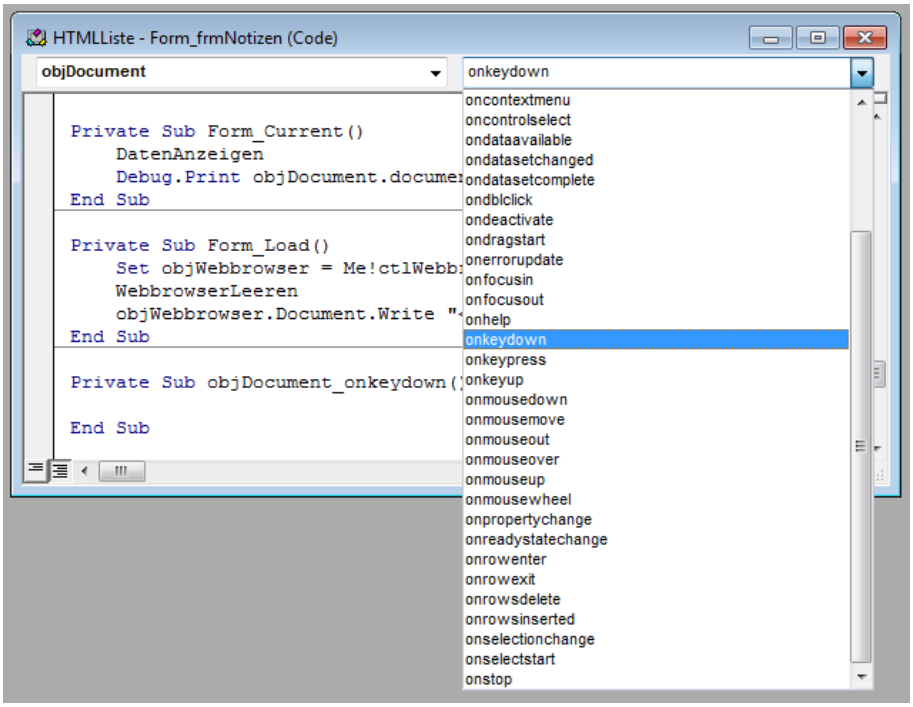

Bild 5: Anlegen eines Ereignisses für das Objekt **objDocument**

Auf die Taste F5 reagieren

Wenn der Benutzer die Taste **F5** betätigt, sollen die im Webbrowser-Steuerelement angezeigten Daten aktualisiert werden. Dies ist der erste Fall, in dem wir ein Ereignis des **HTMLDocument**-Objekts implementieren wollen. Es handelt sich um das Ereignis **Bei Taste ab**, das Sie anlegen, indem Sie im Codefenster des Klassenmoduls des Formulars mit dem linken Kombinationsfeld den Eintrag **objDocument** und mit dem rechten den Wert **onkeydown** auswählen (s. Bild 5). Die nun im Codefenster erscheinende Prozedur **objDocument_onkeydown** füllen Sie wie folgt:

```
Private Sub objDocument_onkeydown()
    Dim Cancel As Boolean
    Cancel = False
    With objDocument.parentWindow.event
        If .keyCode = 116 Then
            Cancel = True 'F5
        End If
        If Cancel Then
            DatenAnzeigen
        End If
    End With
End Sub
```

Die Prozedur prüft, ob der Benutzer die Taste **F5** gedrückt hat, was dem Wert **116** für den Wert **keyCode** des Objekts **objDocument.parentWindow.event** entspricht. In diesem Fall ruft die Prozedur erneut die Routine **DatenAnzeigen** auf und füllt das Webbrowser-Steuerelement erneut.

Der mit dieser Prozedur erzeugte HTML-Quellcode des im Webbrowser-Steuerelement angezeigten Dokuments sieht nun in gekürzter Fassung wie in Listing 3 aus.

Erweiterung: Anzeigen eines Detailformulars

Nun hilft uns die Anzeige der reinen Daten insofern weiter, als dass wir diese nun ohne unnötige Zwischenräume präsentiert bekommen, die durch die unterschiedlich langen Inhalte des Feldes Notiz auftreten. Allerdings können wir im Datenblatt als auch im Endlosformular immer noch die Daten bearbeiten. Diese Möglichkeit wollen wir auch hier anbieten – allerdings in etwas anderer Form, nämlich über ein Detailformular, das beim Anklicken eines bestimmten Bereichs des zu bearbeitenden Datensatzes geöffnet wird. Das Detailformular ist ein einfaches Formular, das einen Datensatz der zugrunde liegenden Tabelle zum Bearbeiten öffnet.

```
<HEAD></HEAD>
<BODY>
  <TABLE style="BORDER-COLLAPSE: collapse">
    <TBODY>
      <TR>
        <TD style="BORDER-BOTTOM: #999999 1px solid; TEXT-ALIGN: center; BORDER-LEFT: #999999 1px solid;
          PADDING-BOTTOM: 5px; BACKGROUND-COLOR: #cccccc; PADDING-LEFT: 5px; PADDING-RIGHT: 5px;
          FONT-FAMILY: Calibri; FONT-SIZE: 9pt; VERTICAL-ALIGN: top; BORDER-TOP: #999999 1px solid;
          FONT-WEIGHT: bold; BORDER-RIGHT: #999999 1px solid; PADDING-TOP: 5px">Datum:</TD>
        <TD style="..." id=1>Notiz:</TD>
      </TR>
      <TR>
        <TD style="..." id=1>01.03.2016</TD>
        <TD style="..." id=1>Dies ist eine kurze Notiz.</TD></TR>
      <TR>
        <TD style="..." id=2>02.03.2016</TD>
        <TD style="..." id=2>Dies ist eine lange Notiz. Dies ist eine lange Notiz. ... Dies ist eine lange Notiz. </TD>
      </TR>
      <TR>
        <TD style="..." id=3>02.03.2016</TD>
        <TD style="..." id=3>Dies ist eine kurze Notiz.</TD>
      </TR>
      <TR>
        <TD style="..." id=4 >02.03.2016</TD>
        <TD style="..." id=4>Dies ist eine lange Notiz. Dies ist eine lange Notiz. ... Dies ist eine lange Notiz. </TD>
      </TR>
      <TR>
        <TD style="..." id=5 >02.03.2016</TD>
        <TD style="..." id=5>Dies ist eine kurze Notiz.</TD>
      </TR>
      <TR>
        <TD style="..." id=6 >06.03.2016</TD>
        <TD style="..." id=6>Dies ist eine lange Notiz. Dies ist eine lange Notiz. ... Dies ist eine lange Notiz. </TD>
      </TR>
    </TBODY>
  </TABLE>
</BODY>
```

Listing 3: Quellcode der erzeugten HTML-Seite

Nun müssen wir uns nur noch entscheiden, wie der Benutzer das Detailformular mit dem gewünschten Datensatz öffnen soll.

Am intuitivsten wäre wohl eine kleine Schaltfläche mit einem Bearbeiten-Symbol für jeden einzelnen Datensatz. Zusätzlich können wir dem Benutzer auch noch ermöglichen, den zu bearbeitenden Datensatz durch einen Doppelklick auf eine der Zellen des betroffenen Datensatzes im Detailformular anzuzeigen. Außerdem benötigen wir natürlich noch eine Schaltfläche, mit der wir das Detailformular zum Anlegen eines neuen Datensatzes öffnen können, und eine Schaltfläche je Datensatz, über die der Benutzer den Datensatz löschen kann.

HTML-Ansicht mit Aktion füllen

Wir haben ja bereits die beiden Objekte **HTMLTableRow** und **HTMLTableCell** mit den Objektvariablen **objRow** und **objCell** referenziert. Diese Variablen können wir auch mit dem Schlüsselwort **WithEvents** versehen und damit dafür sorgen, dass wir deren Ereignisse unter VBA implementieren können. Wir können dann so beispielsweise ein Ereignis behandeln, das durch einen Doppelklick auf ein **HTMLTableCell**-Objekt ausgelöst wird. Oder wir fügen der Tabelle noch ein oder zwei Spalten hinzu, die Icons beziehungsweise Schaltflächen zum Ausführen von Aktionen wie dem Bearbeiten oder Löschen aufnehmen.

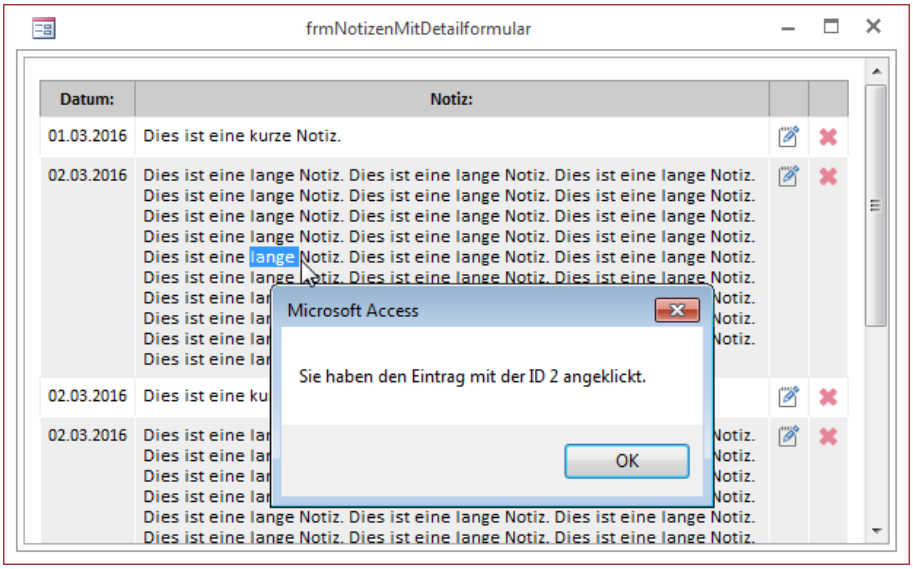

Bild 6: Anzeigen eines Meldungsfensters beim Doppelklick auf einen Eintrag

Im Detail sieht das so aus wie in Bild 6. Hier haben wir etwa durch einen Doppelklick auf eines der **HTMLTableCell**-Objekte ein Ereignis ausgelöst, für das wir eine **MsgBox**-Anweisung hinterlegt haben. Die Schaltflächen rechts lösen beim einfachen Anklicken Ereignisprozeduren aus.

Nun wäre es einfach, eine einzelne Zelle mit einer mit dem Schlüsselwort **WithEvents** gekennzeichneten Variablen zu referenzieren und dafür eine Ereignisprozedur zu hinterlegen. Wir haben aber bereits in dieser einfachen Darstellung in jeder Zeile zwei Zellen, die wir mit Doppelklick-Ereignissen ausstatten müssten – und noch je zwei weitere Icons, die bei einem einfachen Mausklick ihre Ereignisse auslösen sollen. Und da wir diese Menge Objektvariablen auch noch für jeden einzelnen Datensatz benötigen, obwohl wir noch nicht einmal wissen, um wie viele Datensätze es sich hier handelt, können wir kaum mit statischem Code arbeiten: Wir müssen zur Laufzeit beim Füllen des Webbrowser-Steuerelements für jedes zu berücksichtigende Steuerelement ein Objekt auf Basis einer Klasse erstellen, die alle notwendigen Eigenschaften aufnimmt und die Ereignisprozeduren enthält. Diese sollen nach dem Erstellen in einer Collection landen, damit sie nicht im Nirwana verschwinden. Dieses **Collection**-Objekt deklarieren wir wie folgt im Kopf des Klassenmoduls des Formulars (die fortgeschrittene Version finden Sie im Formular **frmNotizenMitDetailformular**):

```
Dim colObjects As Collection
```

Da nun einige Zusatzaufgaben zu erledigen sind, haben wir die Prozedur **DatenAnzeigen** stark angepasst (s. Listing 4). Die Prozedur deklariert weitgehend die gleichen Variablen, einige fallen aber weg. Dann initialisiert sie das **Collection**-Objekt **colObjects**, welches alle Wrapperklassen für die Zellen und Icons der Tabelle aufnehmen soll. Die Definition der Tabellenzeile mit den Spaltenüberschriften erfolgt wieder wie in der vorherigen Version. Innerhalb der **Do While**-Schleife zum Durchlaufen der Datensätze der Tabelle **tblNotizen** ermittelt die Prozedur weiterhin die Hintergrund- und Rahmenfarben für die einzelnen Zeilen. Dann legt sie das **HTMLTableRow**-Objekt an und fügt das erste **HTMLTableCell**-Objekt ein. Die **FormatCell**-Methode haben wir hier etwas erweitert, denn wir übergeben dieser mit dem letzten Parameter auch gleich den anzuzeigen-

```
Private Sub DatenAnzeigen()
    Dim objTable As MSHTML.HTMLTable
    Dim objRow As MSHTML.HTMLTableRow
    Dim objCell As MSHTML.HTMLTableCell
    Dim objImage As MSHTML.HTMLImg
    Dim strBackgroundColor As String
    Dim strBorderColor As String
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set colObjects = New Collection
    Set objDocument = objWebbrowser.Document
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblNotizen", dbOpenDynaset)
    WebbrowserLeeren
    objWebbrowser.Document.Clear
    Set objTable = objDocument.createElement("Table")
    objDocument.Body.appendChild objTable
    objTable.Style.borderCollapse = "collapse"
    Set objRow = objTable.insertRow
    FormatHeader objRow.insertCell, "Datum:"
    FormatHeader objRow.insertCell, "Notiz:"
    FormatHeader objRow.insertCell
    FormatHeader objRow.insertCell
    Do While Not rst.EOF
        If rst.AbsolutePosition Mod 2 = 0 Then
            strBackgroundColor = "#FFFFFF"
            strBorderColor = "#EEEEEE"
        Else
            strBackgroundColor = "EEEEEE"
            strBorderColor = "#FFFFFF"
        End If
        Set objRow = objTable.insertRow
        Set objCell = objRow.insertCell
        FormatCell objCell, strBackgroundColor, strBorderColor, rst!AngelegtAm
        colObjects.Add CreateCellWrapper(objCell, rst!NotizID, Me)
        Set objCell = objRow.insertCell
        FormatCell objCell, strBackgroundColor, strBorderColor, rst!Notiz
        colObjects.Add CreateCellWrapper(objCell, rst!NotizID, Me)
        Set objCell = objRow.insertCell
        FormatCell objCell, strBackgroundColor, strBorderColor
        Set objImage = AddImage(objDocument, objCell, CurrentProject.Path & "\edit.png", 16, 16)
        colObjects.Add CreateImageWrapper(objImage, rst!NotizID, "edit", Me)
        Set objCell = objRow.insertCell
        FormatCell objCell, strBackgroundColor, strBorderColor
        Set objImage = AddImage(objDocument, objCell, CurrentProject.Path & "\delete.png", 16, 16)
        colObjects.Add CreateImageWrapper(objImage, rst!NotizID, "delete", Me)
        rst.MoveNext
    Loop
End Sub
```

Listing 4: Neue Version der Prozedur **DatenAnzeigen**

Benutzerdefinierte Outlook-Eigenschaften

Wenn Sie Outlook mit automatischen Funktionen ausstatten, die einen Zugriff auf eine Datenbankanwendung erfordern (in unserem Fall meist auf eine Access-Datenbank), dann müssen Sie irgendwo den Pfad der Datenbankdatei hinterlegen. Dies könnte man vorübergehend statisch in einem Code-Modul erledigen, aber für professionelle Anwendungszwecke sollte dieser Pfad nicht nur geändert werden können, sondern auch an einem anderen Ort gespeichert werden.

In einigen Beiträgen in Access im Unternehmen haben wir bereits von Outlook aus auf eine Access-Datenbank zugegriffen – zum Beispiel in der Projektzeiterfassung. Hier haben wir den Pfad dann schlicht in eine Konstante eingetragen, wie in folgendem Beispiel:

```
Public Const cStrDB As String = "C:\Users\AndreMinhorst\
Dropbox\Daten\Accessprojekte\Projektzeiterfassung\Projekt-
zeiterfassung.accdb"
```

Dies funktioniert allerdings nur so lange, wie die Datenbankdatei sich auch an dem angegebenen Ort befindet. Verschieben Sie diese Datei, tritt beim Zugriff auf diese Datei ein Fehler auf. Gleiches gilt natürlich, wenn ein Leser des entsprechenden Beitrags den Code einfach in sein eigenes Outlook-Modul kopiert und diesen ausgeführt hat – die Wahrscheinlichkeit, dass die Datenbankdatei beim Leser genau im gleichen Verzeichnis liegt wie beim Autor, ist relativ gering (wenn auch nicht gleich Null). Dennoch möchte ich hier eine professionellere Technik nutzen und den Pfad zur Datenbankdatei erstens an einem anderen Ort unterbringen und diesen zweitens vor dem Zugriff prüfen, damit der Benutzer den Pfad gegebenenfalls manuell selbst aktualisieren kann.

Benutzerdefinierte Einstellung speichern

Zum Speichern einer benutzerdefinierten Einstellung (und zuerst zum Anlegen einer solchen Eigenschaft) verwenden wir einen der Ordner von Outlook, der zum Profil des aktuell angemeldeten Benutzers gehört. Dabei treffen wir in der Regel mit dem Ordner **Posteingang** den richtigen Ordner. Für einen solchen Ordner können Sie ein sogenanntes **StorageItem**-Element anlegen. Dieses Element wird automatisch erstellt, wenn Sie es erstmalig mit der Methode **GetStorage** eines **Folder**-Objekts abrufen. Dabei übergeben Sie mit dem ersten Parameter den Namen des **StorageItem**-Elements und mit dem zweiten die Art, wie es identifiziert werden soll. Für ein **StorageItem**-

```
Public Sub EigenschaftSetzen(strElement As String, strName As String, strWert As String)
    Dim objMAPI As Outlook.NameSpace
    Dim objFolder As Outlook.Folder
    Dim objStorageItem As Outlook.StorageItem
    Dim objUserProperty As Outlook.UserProperty
    Set objMAPI = Outlook.GetNamespace("MAPI")
    Set objFolder = objMAPI.GetDefaultFolder(olFolderInbox)
    Set objStorageItem = objFolder.GetStorage(strElement, olIdentifyBySubject)
    Set objUserProperty = objStorageItem.UserProperties.Add(strName, olText)
    objUserProperty.Value = strWert
    objStorageItem.Save
End Sub
```

Listing 1: Funktion zum Erstellen und Schreiben einer benutzerdefinierten Eigenschaft

Objekt können Sie über die Auflistung **UserProperties** auf benutzerdefinierte Eigenschaften zugreifen. Wir verarbeiten diese Erkenntnisse in einer Funktion namens **EigenschaftSetzen**, die Sie in Listing 1 finden. Die Funktion erwartet drei Parameter:

- **strElement**: Name des StorageItem-Elements
- **strName**: Name der benutzerdefinierten Eigenschaft
- **strWert**: Wert der benutzerdefinierten Eigenschaft

Die Funktion erstellt zunächst ein **NameSpace**-Objekt namens **objMapi** und füllt es mit dem MAPI-Namespace. Dann referenziert es mit der Variablen **objFolder** den standardmäßig als Posteingang definierten Ordner. Nun ruft es die **GetStorage**-Methode des **Folder**-Objekts auf und übergibt diesem den in **strElement** gespeicherten Namen für das **StorageItem**-Element.

Außerdem legt es mit dem Wert **olIdentifyBySubject** für den zweiten Parameter fest, dass der erste Parameter den Namen des Elements enthält. Dieser Aufruf kann keinen Fehler verursachen, da das **StorageItem**-Element, sofern noch nicht vorhanden, mit diesem Aufruf erstellt wird.

Die folgende Anweisung nutzt dann die **Add**-Methode der **UserProperties**-Auflistung des **StorageItem**-Objekts, um eine neue benutzerdefinierte Eigenschaft zum **StorageItem**-Objekt hinzuzufügen. Dabei übergibt sie mit dem ersten Parameter den Namen der benutzerdefinierten Eigenschaft und mit dem zweiten den Datentyp, in diesem Fall **olText** für eine Texteigenschaft. Die neue Eigenschaft referenziert die Prozedur mit der Variablen **objUserProperty**. Diese Objektvariable liefert dann mit der Eigenschaft **Value** die Möglichkeit, den Wert festzulegen – in diesem Fall mit dem Wert des Prozedurparameters **strWert**. Schließlich speichert die Prozedur die neue benutzerdefinierte Variable mit der **Save**-Methode des **StorageItem**-Elements. Um einen Datenbankpfad in der benutzerdefinierten Eigenschaft **Datenbankpfad** im **StorageItem**-Element **Ticketsystem** zu speichern, verwenden Sie etwa den folgenden Aufruf:

```
EigenschaftSetzen "Ticketsystem", "Datenbankpfad", _
                  "c:\Test\Beispieldatenbank.mdb"
```

Benutzerdefinierte Einstellung abrufen

Wenn Sie den Pfad zur Datei, beispielsweise zu einer Access-Datenbank, auf diese Weise mit dem Outlook-Ordner gespeichert haben, wollen Sie diesen auch bei

```
Public Function EigenschaftEinlesen(strElement As String, strName As String) As String
    Dim objMAPI As Outlook.NameSpace
    Dim objFolder As Outlook.Folder
    Dim objStorageItem As Outlook.StorageItem
    Dim objUserProperty As Outlook.UserProperty
    Set objMAPI = Outlook.GetNamespace("MAPI")
    Set objFolder = objMAPI.GetDefaultFolder(olFolderInbox)
    Set objStorageItem = objFolder.GetStorage(strElement, olIdentifyBySubject)
    Set objUserProperty = objStorageItem.UserProperties.Item(strName)
    If objUserProperty Is Nothing Then
        EigenschaftEinlesen = ""
    Else
        EigenschaftEinlesen = objUserProperty.Value
    End If
End Function
```

Listing 2: Funktion zum Einlesen einer benutzerdefinierten Eigenschaft

Gelegenheit, etwa beim Start von Outlook, einlesen und verwenden. Dies erledigen wir mit der Funktion **EigenschaftEinlesen** aus Listing 2.

Die Funktion erwartet zwei Parameter:

- **strElement**: Name des **StorageItem**-Elements
- **strName**: Name der Eigenschaft

Die Funktion referenziert wieder das MAPI-Namespace-Objekt mit der Variablen **objMAPI** und den Posteingangsordner mit der Variablen **objFolder**. Dann greift sie wieder mit der **GetStorage**-Methode auf das **StorageItem**-Element zu, das den mit dem Parameter **strElement** übergebenen Namen besitzt. Auch hier kann wieder kein Fehler auftreten, da das Objekt, soweit noch nicht vorhanden, in diesem Moment angelegt wird. Probleme können lediglich auftauchen, wenn die Funktion über die **Item**-Methode mit dem Namen der Eigenschaft als Parameter auf das gesuchte Element der **UserProperties**-Auflistung zugreift.

Sollte die Eigenschaft nämlich noch nicht angelegt worden sein, führt das anschließende Abrufen des Eigenschaftswertes mit der **Value**-Eigenschaft zu einem Fehler.

Dies können wir jedoch leicht verhindern, indem wir mit **objUserProperty Is Nothing** prüfen, ob die benutzerdefinierte Eigenschaft bereits angelegt wurde oder nicht. Nur in ersterem Fall tragen wir den Wert dieser Eigenschaft in die Rückgabevariable der Funktion ein, sonst eine leere Zeichenfolge.

Letzteres kann auch wegfallen, da der Standardwert für die String-Variable ohnehin eine leere Zeichenkette ist, aber der Übersichtlichkeit halber haben wir diese beiden Zeilen im Code gelassen. Der folgende Aufruf würde nun beispielsweise den in der benutzerdefinierten Eigenschaft **Datenbankpfad** im **StorageItem**-Objekt **Ticketsystem** enthaltenen Wert im Direktbereich ausgeben:

```
Debug.Print EigenschaftEinlesen ("Ticketsystem", _
                                 "Datenbankpfad")
```

Pfad manuell aktualisieren

Nun wollen wir noch Beispielcode entwickeln, der prüft, ob die in der Variablen gespeicherte Datenbankdatei tatsächlich vorhanden ist. Ist dies nicht der Fall, soll dieser gleich einen **Datei öffnen**-Dialog anzeigen, mit dem der Benutzer den aktuellen Ort der Datenbankdatei auswählen kann. Dazu nutzen wir eine universell nutzbare Funktion

```
Public Function DatenbankpfadHolen(strElement As String, strName As String) As String
    Dim strDatenbankpfad As String
    Dim bolVorhanden As Boolean
    strDatenbankpfad = EigenschaftEinlesen(strElement, strName)
    If Not Len(strDatenbankpfad) = 0 Then
        If Not Len(Dir(strDatenbankpfad)) = 0 Then
            bolVorhanden = True
        End If
    End If
    If Not bolVorhanden Then
        strDatenbankpfad = OpenFileName("", "Datenbankpfad auswählen", "Access-DB (*.mdb;*.accdb)")
        EigenschaftSetzen strElement, strName, strDatenbankpfad
    End If
    DatenbankpfadHolen = strDatenbankpfad
End Function
```

Listing 3: Diese Funktion liefert den in einer benutzerdefinierten Eigenschaft gespeicherten Pfad oder zeigt einen Datei öffnen-Dialog an.

Standardverzeichnisse per VBA ermitteln

Es gibt eine ganze Reihe von Verzeichnissen, die es auf jedem Rechner gibt. Manche davon sollten eigentlich immer gleich lauten, aber durch die Installation von Windows oder Software auf Laufwerken mit anderen Laufwerkbuchstaben als C: gibt es hier gelegentlich Unterschiede. In manchen Fällen sorgt auch die Betriebssystemversion für Unterschiede, beispielsweise bei 32bit- gegenüber 64bit-Systemen. Und ganz sicher unterscheiden sich die Verzeichnisse, die dem jeweiligen Benutzer gehören. Dennoch wollen Sie früher oder später einmal dynamisch auf ein bestimmtes Verzeichnis zugreifen wie etwa das Verzeichnis der eigenen Dateien eines Benutzers oder das Add-In-Verzeichnis von Access. Dieser Beitrag zeigt, wie Sie solche Verzeichnisse ermitteln.

Verzeichnisse, Dateinamen und Pfade

Vorab eine kleine Begriffsklärung: Wir sprechen nachfolgend von Verzeichnis, wenn es sich um einen Ort handelt, an dem man Dateien speichern kann:

```
c:\Verzeichnis
```

Ein Dateiname ist der reine Dateiname ohne Verzeichnis:

```
Beispieldatenbank.mdb
```

Der Pfad fasst Verzeichnis und Dateiname zusammen, wobei beide noch durch das Backslash-Zeichen voneinander getrennt werden:

```
c:\Verzeichnis\Beispieldatenbank.mdb
```

Verzeichnis der aktuellen Datenbankdatei

Die aktuelle Datenbank wollen Sie zum Beispiel ermitteln, wenn Sie auf Dateien zugreifen wollen, die Sie im Kontext der aktuellen Datenbankanwendung direkt im gleichen oder in einem untergeordneten Verzeichnis der Datenbank gespeichert haben. Früher, als es noch kein **CurrentProject**-Objekt gab, hat man das Verzeichnis einer Datenbank über den Umweg der Eigenschaft **Name** des Objekts **CurrentDb** ermittelt. **CurrentDb.Name** lieferte den kompletten Pfad, also beispielsweise den folgenden Ausdruck:

```
c:\Verzeichnis\Beispieldatenbank.mdb
```

Den Dateinamen konnte man mithilfe der Funktion **Dir** ermitteln, die lediglich den Namen einer Datei bei Angabe des kompletten Pfades zurücklieferte:

```
? Dir("c:\Verzeichnis\Beispieldatenbank.mdb")
  Beispieldatenbank.mdb
```

Davon ausgehend, dass der Dateiname nicht als Teil eines der Verzeichnisse verwendet wurde, die den Speicherort der Datei festlegten, ließ sich so durch geschickten Einsatz von Zeichenkettenfunktionen auch das Verzeichnis extrahieren:

```
Left(CurrentDb.Name, Len(CurrentDb.Name) -
Len(Dir(CurrentDb.Name)))
```

Dies ermittelte im letzten Teil zunächst die Länge des Dateinamens. Dann wird die Länge des Verzeichnisses ohne Dateiname ermittelt. Das Ergebnis dient als zweiter Parameter der **Left**-Funktion, die nur die dadurch angegebene Anzahl Zeichen des im ersten Parameter übergebenen Ausdrucks zurückgibt.

Ab Access 2000 änderte sich dies zum Glück: Das **CurrentProject**-Objekt wurde eingeführt und lieferte einige Eigenschaften, mit denen sich Dateiname, Verzeichnis und

Pfad separat ermitteln ließen – nämlich **Name**, **Path** und **FullName**:

```
? CurrentProject.Name
  StandardverzeichnissePerVBA.accdb
? CurrentProject.Path
  C:\...\StandardverzeichnissePerVBA
? CurrentProject.FullName
  C:\...\StandardverzeichnissePerVBA\
                        StandardverzeichnissePerVBA.accdb
```

Verzeichnis der Datei MSAccess.exe

Das Verzeichnis der ausführbaren Datei **MSAccess.exe**, mit der die aktuell geöffnete Datenbankdatei gestartet wurde, finden Sie mit dem folgenden Ausdruck – hier etwa für die 32bit-Version unter Office 15:

```
? SysCmd(acSysCmdAccessDir)
  C:\Program Files (x86)\Microsoft Office\Office15\
```

Verzeichnis einer Add-In-Datenbank

Wenn Sie ein Add-In nutzen und innerhalb des Add-Ins auf die aktuell geöffnete Datenbank zugreifen möchten, gelingt dies mit den weiter oben vorgestellten Eigenschaften **CurrentDb.Name** beziehungsweise **CurrentProject.Path**, **CurrentProject.Name** und **CurrentProject.FullName**. Aber was, wenn Sie auf die Add-In-Datenbank zugreifen wollen – beispielsweise, um ihren Datenbankpfad zu ermitteln und in diesem Verzeichnis Dateien zu speichern?

Dann nutzen Sie statt der Objekte **CurrentDb** und **CurrentProject** einfach analog **CodeDb** und **CodeProject**. Wenn Sie die Add-In-Datenbank direkt unter Access öffnen (und nicht über das Add-In-Menü), sind die Objekte **CurrentDb** und **CodeDb** beziehungsweise **CurrentProject** und **CodeProject** identisch.

Weitere Verzeichnisse per API ermitteln

Die API-Funktion **SHGetKnownFolderPath** ermöglicht es, eine ganze Reihe von Verzeichnissen zu ermitteln. Dabei übergeben Sie dieser lediglich verschiedene Werte, die dem jeweiligen Verzeichnis entsprechen. Die API-Funktion deklarieren Sie wie folgt in einem Standardmodul:

```
Private Declare Function SHGetKnownFolderPath Lib
"shell32" (rfid As Any, ByVal dwFlags As Long, ByVal hTo-
ken As Long, ppszPath As Long) As Long
```

Wir benötigen aber noch weitere API-Deklarationen:

```
Private Declare Function CLSIDFromString Lib "ole32" (By-
Val lpszGuid As Long, pGuid As Any) As Long
Private Declare Sub CopyMemory Lib "kernel32" Alias "Rtl-
MoveMemory" (pDest As Any, pSrc As Any, ByVal ByteLen As
Long)
Private Declare Sub CoTaskMemFree Lib "ole32" (ByVal hMem
As Long)
Private Declare Function lstrlenW Lib "kernel32" (ByVal
ptr As Long) As Long
```

Außerdem verwendet die nachfolgend beschriebene Funktion ein **Type**-Element namens **GUID** zum Speichern der einzelnen Elemente einer GUID:

```
Private Type GUID
   Data1 As Long
   Data2 As Integer
   Data3 As Integer
   Data4(7) As Byte
End Type
```

Damit wir die Kennung des gesuchten Verzeichnisses, das beispielsweise **{D20BEEC4-5CA8-4905-AE3B-BF251E-A09B53}** lautet, nicht auf diese Weise eingeben müssen, sondern dies einfacher erledigen können, haben wir eine Enumeration mit je einem Element pro möglichem Verzeichnis erstellt. Diese sieht wie in Listing 1 aus.

Leider kann man den Elementen einer solchen Enumeration (Fortsetzung in Listing 2) nur Zahlenwerte zuweisen, aber keine Zeichenketten. Deshalb benötigen wir eine Hilfsfunktion, welche die GUID zu der jeweiligen Konstanten liefert.

```
Public Enum FolderID
    FOLDERID_NetworkFolder
    FOLDERID_ComputerFolder
    FOLDERID_InternetFolder
    FOLDERID_ControlPanelFolder
    FOLDERID_PrintersFolder
    FOLDERID_SyncManagerFolder
    FOLDERID_SyncSetupFolder
    FOLDERID_ConflictFolder
    FOLDERID_SyncResultsFolder
    FOLDERID_RecycleBinFolder
    FOLDERID_ConnectionsFolder
    FOLDERID_Fonts
    FOLDERID_Desktop
    FOLDERID_Startup
    FOLDERID_Programs
    FOLDERID_StartMenu
    FOLDERID_Recent
    FOLDERID_SendTo
    FOLDERID_Documents
    FOLDERID_Favorites
    FOLDERID_NetHood
    FOLDERID_PrintHood
    FOLDERID_Templates
    FOLDERID_CommonStartup
    FOLDERID_CommonPrograms
    FOLDERID_CommonStartMenu
    FOLDERID_PublicDesktop
    FOLDERID_ProgramData
    FOLDERID_CommonTemplates
    FOLDERID_PublicDocuments
    FOLDERID_RoamingAppData
    FOLDERID_LocalAppData
    FOLDERID_LocalAppDataLow
    FOLDERID_InternetCache
    FOLDERID_Cookies
    FOLDERID_History
    FOLDERID_System
    FOLDERID_SystemX86
    FOLDERID_Windows
    FOLDERID_Profile
    FOLDERID_Pictures
    FOLDERID_ProgramFilesX86
    FOLDERID_ProgramFilesCommonX86
    FOLDERID_ProgramFilesX64
    ...
```

Listing 1: Auflistung der Konstanten für die Ordner

```
    ...
    FOLDERID_ProgramFilesCommonX64
    FOLDERID_ProgramFiles
    FOLDERID_ProgramFilesCommon
    FOLDERID_AdminTools
    FOLDERID_CommonAdminTools
    FOLDERID_Music
    FOLDERID_Videos
    FOLDERID_PublicPictures
    FOLDERID_PublicMusic
    FOLDERID_PublicVideos
    FOLDERID_ResourceDir
    FOLDERID_LocalizedResourcesDir
    FOLDERID_CommonOEMLinks
    FOLDERID_CDBurning
    FOLDERID_UserProfiles
    FOLDERID_Playlists
    FOLDERID_SamplePlaylists
    FOLDERID_SampleMusic
    FOLDERID_SamplePictures
    FOLDERID_SampleVideos
    FOLDERID_PhotoAlbums
    FOLDERID_Public
    FOLDERID_ChangeRemovePrograms
    FOLDERID_AppUpdates
    FOLDERID_AddNewPrograms
    FOLDERID_Downloads
    FOLDERID_PublicDownloads
    FOLDERID_SavedSearches
    FOLDERID_QuickLaunch
    FOLDERID_Contacts
    FOLDERID_SidebarParts
    FOLDERID_SidebarDefaultParts
    FOLDERID_TreeProperties
    FOLDERID_PublicGameTasks
    FOLDERID_GameTasks
    FOLDERID_SavedGames
    FOLDERID_Games
    FOLDERID_RecordedTV
    FOLDERID_SEARCH_MAPI
    FOLDERID_SEARCH_CSC
    FOLDERID_Links
    FOLDERID_UsersFiles
    FOLDERID_SearchHome
    FOLDERID_OriginalImages
End Enum
```

Listing 2: Auflistung der Konstanten für die Ordner, Fortsetzung

Windows-API per VBA nutzen

Access stellt ausreichend Bordmittel zur Programmierung von Datenbankanwendungen zur Verfügung. Dazu gehören die Benutzerschnittstellen zur Gestaltung der Datenbankobjekte wie Tabellen, Abfragen, Formulare und Berichte sowie die Möglichkeit, Abläufe ereignisgesteuert per Makro oder VBA zu automatisieren. Manchmal reichen die vorhandenen Befehle aber nicht aus. In dem Fall greifen Sie auf die Prozeduren der Windows-API (Application Programming Interface) zu. Mit API-Prozeduren können Sie Ihre Anwendungen alles machen lassen, was Windows auch kann.

Die Windows-API verfügt über eine sehr große Anzahl von Befehlen. Der vorliegende Beitrag kann Ihnen aber aus Platzgründen nur einen kleinen Einblick in die Möglichkeiten der API-Programmierung geben. Nach einer kurzen Einführung, in der die wichtigsten Begriffe und die technischen Grundlagen erklärt werden, folgt der Praxisteil mit einigen Beispielen. Die Beispiele sind so ausgewählt, dass Sie sie gut in Ihre eigenen Anwendungen übernehmen können.

VBA und API

Das Datenbanksystem **Microsoft Access** ist Windows-kompatibel. Daraus ergeben sich für den Entwickler große Vorteile. Er kann mit Access Datenbanken erstellen, welche die Eigenschaften von Windows wie selbstverständlich ausnutzen. Dabei muss der Entwickler sich nicht damit abmühen, das Aussehen von Formularen, Berichten und den darin enthaltenen Steuerelementen zu programmieren. Access stellt Werkzeuge zur Verfügung, mit denen die Erstellung solcher Objekte zum Kinderspiel wird. Der Entwickler muss lediglich die verschiedenen Ereignisse per VBA programmieren, mit denen die Anwendung auf die unterschiedlichen Benutzereingaben reagiert.

Streng genommen handelt es sich bei den Befehlen nicht nur um VBA-Befehle. VBA selbst enthält nur die Befehle, die von den Microsoft-Anwendungen wie zum Beispiel Word, Excel und eben Access genutzt werden können. Um eine Datenbank zu programmieren, benötigen Sie darüber hinaus mindestens noch die Bibliotheken **Microsoft Access x.0 Object Library** und eine Datenzugriffsbibliothek wie in den neueren Versionen von Access die Bibliothek **Microsoft Office x.0 Access Database Engine Object Library**. Die beiden Bibliotheken enthalten die Befehle zur Steuerung der Datenbankobjekte von Access und zum Zugriff auf die in der Datenbank gespeicherten Daten.

Windows selbst kennt über die Befehle der drei genannten Bibliotheken hinaus noch viel mehr Befehle. Diese sogenannten API-Befehle können Sie auch von Access aus aufrufen. In den folgenden Abschnitten erfahren Sie, wie Sie unterschiedliche Befehle der Windows-API aufrufen können und welche Vorraussetzungen dazu erfüllt sein müssen.

Die Prozedurbibliotheken von Windows

Die wesentlichen zum Betrieb von Windows notwendigen Prozeduren sind auf einige wenige Bibliotheken verteilt. Die folgenden drei sind die wichtigsten: **Kernel32.dll**, **User32.dll** und **GDI32.dll**.

Die Bibliothek **Kernel32.dll** macht den eigentlichen Kern von Windows aus. Zu den Aufgaben der Prozeduren dieser Bibliothek gehören die Auswertung von Tastatureingaben, die Ausgabe auf dem Bildschirm, die Verwaltung der laufenden Anwendungen, die Verwaltung des Dateisystems und andere, die aber für den Einsatz im Zusammenhang mit Access eine untergeordnete Bedeutung haben.

Die Prozeduren der Dynamic Link Library **User32** dienen hauptsächlich der Bereitstellung der Benutzerschnittstelle, das heißt der Fenster, des Mauszeigers, des Ribbons und

der Menüs und so weiter. Das Öffnen, Schließen und Verschieben der Fenster gehört zu den Aufgaben der **User32.dll**.

Das Graphics Device Interface (**GDI32.dll**) dient der Ausgabe von grafischen Elementen wie Grafiken, Text, Farben et cetera. Es dient praktisch als Schnittstelle zwischen dem Benutzer und dem Ausgabegerät. Als Ausgabegeräte kommen dabei alle möglichen Geräte infrage – sowohl unterschiedliche Monitore als auch Drucker und andere Ausgabegeräte. Windows stellt für alle möglichen Formate einen Treiber zur Verfügung.

Neben den genannten Bibliotheken gibt es noch einige weitere, wie etwa die **COMDLG32.dll**, die unter anderem die Dialoge zum Öffnen und Speichern von Dateien und zur Auswahl von Farben und von Zeichensätzen zur Verfügung stellt.

Wie hier zu erkennen ist, sind die zum Betrieb von Windows notwendigen Prozeduren nach Aufgaben geordnet auf unterschiedliche Bibliotheken aufgeteilt. Durch diese modulare Bauweise ist Windows leicht erweiterbar. Dementsprechend sind im Laufe der Zeit noch einige weitere Bibliotheken hinzugekommen. Sie stellen zum Beispiel neue Steuerelemente zur Verfügung (**COMCTL32.dll**), ermöglichen den Austausch von E-Mail (**MAPI32.dll**) oder dienen zur Verwaltung von Netzwerken (**NETAPI32.dll**).

Wo finde ich die API-Prozedur für meine Zwecke?

Ohne entsprechende Dokumentation der Prozeduren der Windows-API ist es sehr schwer, bestimmte Aufgaben mit Hilfe der API zu lösen. Sie müssen also zunächst eine geeignete API-Prozedur für Ihre Aufgabe finden. Das zweite Problem ist, dass die meisten Prozeduren der API-Bibliotheken in C programmiert sind. Dementsprechend sind sie auch für den Aufruf von C-Programmen ausgelegt. Vor allem wegen der unterschiedlichen Variablentypen in Visual Basic und in C kann der Aufruf einer solchen Funktion manchmal Probleme bereiten. Es gibt aber genügend Quellen, die den Aufruf von API-Funktionen aus Visual Basic heraus beschreiben. Neben diversen Fachzeitschriften, in denen immer wieder Beiträge zum Thema API auftauchen, soll an dieser Stelle auf ein Buch hingewiesen werden, das zwar leider englischsprachig ist, aber von vielen als API-Bibel bezeichnet wird: Dan Appleman's „Visual Basic Programmer's Guide to the Win32 API". Wenn Sie also Gefallen an der Integration von API-Funktionen in Ihre Anwendungen finden, erfahren Sie dort mehr.

Deklaration von API-Funktionen

Sie können API-Funktionen nicht wie gewöhnliche VBA-Prozeduren aufrufen. Sie müssen die gewünschte Prozedur zunächst deklarieren. Strenggenommen werden die VBA-Befehle und die Befehle zur Verwaltung der Datenbankobjekte auch deklariert – allerdings funktioniert das auf eine andere Weise und wird normalerweise direkt beim Öffnen von Access erledigt. Die Deklaration dieser und anderer Bibliotheken nehmen Sie vor, indem Sie mit einem Verweis direkt die ganze Bibliothek verfügbar machen. Um einen solchen Verweis zu setzen, öffnen Sie ein beliebiges Modul in der Entwurfsansicht und wählen den Menübefehl **Extras|Verweise**. Im Dialog **Verweise** (s. Bild 1) können Sie durch Markieren der Kontrollkästchen Verweise auf die einzelnen Bibliotheken setzen.

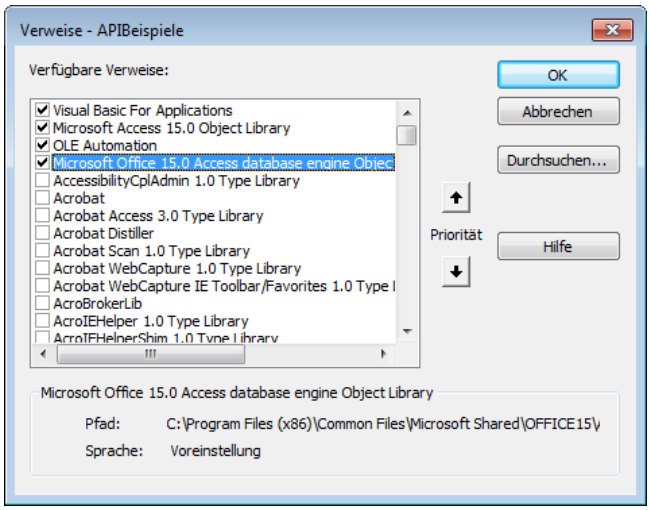

Bild 1: Die Funktionen einiger Bibliotheken können per Verweis referenziert werden.

Die Deklaration von API-Prozeduren ist nicht ganz so einfach. Sie müssen jede API-Prozedur separat deklarieren. Bevor Sie mit dem ersten Beispiel beginnen, soll hier kurz die Syntax der Deklaration einer API-Prozedur erläutert werden.

Syntax der Deklaration einer API-Prozedur

Allgemein sieht die Syntax folgendermaßen aus:

```
[Public|Private] Declare Function|Sub name Lib "libname" _
[Alias "aliasname"]([argumentlist])] [As type]
```

Mit dem ersten Parameter, **Public** oder **Private**, geben Sie an, ob die Funktion nur innerhalb des Moduls Gültigkeit hat, oder ob sie anwendungsweit aufgerufen werden kann. Lassen Sie den ersten Parameter weg, hat die Funktion anwendungsweite Gültigkeit.

Die **Declare**-Anweisung weist darauf hin, dass eine externe Funktion deklariert wird.

Als nächstes Schlüsselwort geben Sie wie bei der Programmierung einer VBA-Prozedur entweder das Schlüsselwort **Function** oder **Sub** an – je nachdem, ob die Prozedur einen oder mehrere Werte zurückgibt oder nicht. In der Regel bekommen Sie es aber mit Funktionen zu tun.

Unter **Name** geben Sie den Namen der API-Prozedur an. Unter dem hier angegebenen Namen rufen Sie die Prozedur von Ihren eigenen Prozeduren aus auf. In der Regel ist das auch der Name, unter dem die Prozedur in der jeweiligen DLL abgelegt ist. Es gibt aber manche API-Prozeduren, deren Name nicht den VBA-Konventionen für Prozedurnamen entspricht. In dem Fall wird der eigentliche Name der API-Prozedur hinter dem Schlüsselwort Alias als **aliasname** angegeben. Zum Aufruf der API-Prozedur benutzen Sie aber die unter **Name** angegebene Bezeichnung.

Nach dem Schlüsselwort **Lib** folgt der Name der Bibliothek, in der die Prozedur abgelegt ist. Es gibt drei unterschiedliche Arten, die Bibliothek anzugeben. Bei den drei Bibliotheken **Kernel32**, **User32** und **GDI32** reicht die Angabe des Namens ohne Dateinamenendung. Bei allen anderen Bibliotheken geben Sie die Dateinamenendung an. Manchmal müssen Sie zusätzlich den Pfad der Bibliothek angeben. Das ist der Fall, wenn sich die Bibliothek nicht in den folgenden Verzeichnissen befindet:

- Anwendungsverzeichnis
- aktuelles Verzeichnis
- Windows-Systemverzeichnis
- Windows-Verzeichnis
- Verzeichnisse, die in der Umgebungsvariablen **Path** angegeben sind

Für den Parameter **ArgumentList** geben Sie die Argumente an, die beim Aufruf der Prozedur übergeben werden. Die Übergabe von Argumenten kann auf zwei Arten erfolgen. Welche der beiden Arten zu wählen ist, gibt die jeweilige Prozedur vor. Entweder Sie übergeben den Parameter als Wert und setzen dem Parameter das Schlüsselwort **ByVal** voran, oder Sie übergeben die Speicheradresse des Parameters. Geben Sie dann entweder das Schlüsselwort **ByRef** oder gar kein Schlüsselwort an.

Übergeben Sie den Parameter als Wert, erhält die DLL nur eine Kopie des Parameters. Nachdem der Parameter bearbeitet wurde, wird die Kopie über das Original geschrieben. Wenn Sie hingegen eine Referenz übergeben, also einen Verweis auf die Adresse des Parameters, kann die DLL direkt auf den Parameter zugreifen.

Wann man welche Art der Parameterübergabe wählt und warum, wird erst deutlich, wenn man sich eingehender mit den unterschiedlichen Variablenarten in Visual Basic und C beschäftigt. In den später vorgestellten Beispielen wird die Art der Parameterübergabe jeweils angegeben.

Zu der Parameterübergabe in weiteren API-Prozeduren erfahren Sie mehr in der einschlägigen Fachliteratur.

Hinter dem Schlüsselwort **As** geben Sie schließlich den Datentyp an. Mehr über die in API-Aufrufen verwendeten Datentypen erfahren Sie im nächsten Abschnitt.

Datentypen beim Aufruf von API-Prozeduren

Die zu verwendenden Datentypen werden ähnlich wie die Art der Parameterübergabe durch die aufzurufende Prozedur festgelegt. Üblicherweise finden hier die in VB üblichen Datentypen Verwendung.

Eine Ausnahme ist der Datentyp **Any**. Er wird nur in Zusammenhang mit dem Aufruf von API-Prozeduren eingesetzt. Ein Parameter des Datentyps **Any** kann unterschiedliche Datentypen annehmen, zum Beispiel **String** und **Long**. Wenn Ihnen ein solcher Datentyp begegnet, ist meist ein weiterer Parameter nicht weit, in dem der für den Parameter gültige Datentyp übergeben wird.

Eine weitere Ausnahme sind benutzerdefinierte Datentypen. Parameter eines benutzerdefinierten Datentyps bestehen meist aus mehreren Parametern, die sich auf ein einziges Objekt beziehen. Solche Datentypen müssen im Deklarationsteil eines Moduls deklariert werden. Anschließend müssen Sie den Namen des benutzerdefinierten Datentyps der gewünschten Variablen zuweisen.

Wo deklariere ich API-Prozeduren?

Es gibt zwei sinnvolle Möglichkeiten, API-Prozeduren zu deklarieren. Entweder Sie nehmen die Deklaration aller API-Funktionen und benutzerdefinierten Datentypen in einem eigens dafür angelegten Modul vor. Auf diese Weise ist es ab einer gewissen Menge Deklarationen leichter, die Übersicht zu behalten. Wenn Sie hingegen wissen, dass Sie eine bestimmte API-Funktion nur in einer Prozedur aufrufen, sollten Sie die Deklaration und die aufrufende Prozedur in einem Modul unterbringen. So können Sie die Komponenten im Paket kopieren und in anderen Anwendungen verwenden.

Praxisbeispiele zur Benutzung der Windows-API mit Access

Leider sind die Kenntnisse einiger theoretischer Grundlagen bei der Benutzung der Windows-API unumgänglich. Da die API weit über 1.000 Prozeduren kennt, können Sie sich vorstellen, dass nicht alles Wichtige in den vorangegangenen Abschnitten untergebracht werden konnte. In den folgenden Beispielen zum praktischen Umgang mit den Prozeduren der Windows-API werden sich deshalb immer wieder kleine Abschnitte mit den theoretischen Hintergründen beschäftigen.

Da der vorliegende Beitrag nicht alle API-Prozeduren vorstellen kann, finden Sie hier nur eine Auswahl von Prozeduren.

Name des Computers und des Benutzers ermitteln

Die API-Funktionen zur Bestimmung des Computernamens und des aktuellen Benutzers des Computers sollen als erstes Beispiel dienen. Die entsprechenden API-Prozeduren heißen **GetComputerName** und **GetUserName**.

Sie müssen die Prozeduren vor dem Aufruf in geeigneter Weise deklarieren. Damit Sie die im Rahmen des vorliegenden Beitrags erstellten Prozeduren mit ihren API-Aufrufen gut in Ihre eigenen Datenbanklösungen integrieren können, legen Sie jeweils ein neues Modul für jedes Beispiel an.

Wenn Sie ein neues Standardmodul angelegt haben, speichern Sie es unter dem Namen **mdlSystemInformation**. Anschließend deklarieren Sie die erste der beiden Funktionen wie folgt:

```
Declare Function GetComputerName Lib "kernel32" _
    Alias "GetComputerNameA" (ByVal lpBuffer As _
    String, nSize As Long) As Long
```

Der Deklaration können Sie entnehmen, dass die Funktion **GetComputerName** zur Dynamic Link Library **Kernel32**

gehört und dort unter dem Namen **GetComputerNameA** bekannt ist. Die Funktion hat zwei Parameter, die Zeichenkette **lpBuffer** wird als Wert übergeben, während die Länge der Zeichenkette **nSize** als Referenz übergeben wird. Der Rückgabewert hat den Datentyp **Long**.

```
Public Function Computername() As String
    Dim strComputerName As String
    Dim lngAnzahlZeichen As Long
    Dim lngTemp As Long
    lngAnzahlZeichen = 256
    strComputerName = Space$(lngAnzahlZeichen)
    lngTemp = GetComputerName(strComputerName, lngAnzahlZeichen)
    Computername = Trim$(strComputerName)
End Function
```

Listing 1: Funktion zum Ermitteln des Computernamens per API

Der Rückgabewert enthält bei vielen Funktionen nicht die Informationen, die Sie eigentlich abrufen möchten. In der vorliegenden Funktion **GetComputerName** wird etwa der Zahlenwert **1** zurückgegeben, wenn es einen Computernamen gibt, und der Wert **0**, wenn kein Computername gefunden wurde.

```
Public Function Benutzername() As String
    Dim strBenutzername As String
    Dim lngAnzahlZeichen As Long
    lngAnzahlZeichen = 256
    strBenutzername = Space$(lngAnzahlZeichen)
    GetUserName strBenutzername, lngAnzahlZeichen
    Benutzername = Trim$(strBenutzername)
End Function
```

Listing 2: Funktion zum Ermitteln des Benutzernamens per API

Die gewünschte Information packt die API-Funktion in die Parameter. Sie weist dem Parameter **lpBuffer** den Namen des Computers als String und dem Parameter **nSize** die Länge der übergebenen Zeichenkette als **Long** zu.

Es ist sehr wichtig, dass Sie **String**-Parametern vor der Übergabe an eine DLL eine Zeichenkette zuweisen, die mehr Zeichen enthält als der erwartete Wert haben kann. Ist der zurückgegebene String länger als der übergebene String, löst das einen Fehler aus.

In der Funktion aus Listing 1 können Sie die API-Funktion nun endlich einsetzen.

Zunächst deklarieren Sie dort die benötigten Variablen für den zu übergebenden Text sowie die Länge des Textes.

Der Variablen **lngTemp** weisen Sie den Rückgabewert der Funktion zu. Dem Parameter **strComputerName** weisen Sie mit der **Space$**-Funktion einen String mit 256 Leerzeichen zu. Nach dem Aufruf der Funktion benutzen Sie die Funktion **Trim$**, um die überflüssigen Leerzeichen des zurückgegebenen Wertes zu entfernen.

Testen Sie die Funktion mit Hilfe des Testfenster (zu öffnen mit **Strg + G**). Lassen Sie sich einfach den Computernamen in einem Meldungsfenster ausgeben:

```
? Computername
WIN7OFF13IND
```

Benutzername per API ermitteln

Die Ausgabe des Benutzernamens können Sie in ähnlicher Weise vornehmen. Die Deklaration des entsprechenden API-Aufrufs lautet:

```
Declare Function GetUserName Lib "advapi32.dll" _
    Alias "GetUserNameA" (ByVal lpBuffer As String, _
    nSize As Long) As Long
```

Die Vorgehensweise zum Ermitteln des Namens des aktuell eingeloggten Benutzers ist prinzipiell mit dem Ermitteln des Computernamens identisch. Die Funktion zum Aufruf der API-Prozedur **GetUserNameA** ist allerdings etwas anders gestaltet, um Sie auf eine Beson-

```
Public Function FenstertitelErmitteln()
    Dim hWnd As Long
    Dim strFenstertitel As String
    Dim lngAnzahlZeichen As Integer
    lngAnzahlZeichen = 256
    strFenstertitel = Space$(lngAnzahlZeichen)
    hWnd = GetActiveWindow()
    GetWindowText hWnd, strFenstertitel, 255
    FenstertitelErmitteln = Left$(strFenstertitel, lngAnzahlZeichen)
End Function
```

Listing 3: Funktion zum Ermitteln des Fenstertitels des aktiven Fensters

derheit beim Aufruf von API-Funktionen aufmerksam zu machen (s. Listing 2).

Wie Sie erkennen können, gibt es in der Funktion keine Variable namens **lngTemp**, um den Rückgabewert der API-Funktion zu speichern. Die Funktion wird auch gar nicht in Form einer Wertzuweisung aufgerufen, wie es üblicherweise bei Funktionen der Fall ist, sondern wie eine Prozedur. Da es in den meisten Fällen nicht auf den Rückgabewert ankommt, sondern auf die geänderten Parameter, kann man sich hier die Deklaration einer Variablen und damit Zeit und Speicherplatz sparen.

Die beiden Funktionen können Sie nun zum Beispiel benutzen, um den Anwender beim Start der Datenbank mit seinem Namen zu begrüßen.

Zugriff auf Fenster und andere Objekte per Handle

Den Begriff **Handle** werden Sie sicher bereits einmal gehört haben. Ein Handle ist eine Möglichkeit, Objekte unter Windows eindeutig zu kennzeichnen. Dadurch, dass Windows mitunter Objekte aus dem Arbeitsspeicher temporär auf die Festplatte oder in andere Speicherbereiche verbannt, sind die Objekte teilweise nicht mehr unter der angegebenen Speicheradresse anzutreffen.

Dies erscheint zunächst wie ein Windows-internes Problem – da Sie aber per API Windows direkt steuern, erweisen sich Handles für den Entwickler als sehr nützlich. Der Inhalt eines Handles, also der Verweis auf ein Objekt, wird immer im Arbeitsspeicher gehalten – auch wenn das Objekt selbst zwischenzeitlich auf die Festplatte verbannt wurde.

Wenn Sie also per API auf ein Objekt zugreifen möchten, ermitteln Sie zunächst dessen Handle. Zur Ermittlung eines Handles gibt es unterschiedliche Möglichkeiten, von denen einige später vorgestellt werden. Wenn Sie das Handle einmal kennen, können Sie über das Handle auf das entsprechende Objekt zugreifen.

Deutlicher wird dies an einem Beispiel. Mit der Funktion **GetWindowText** soll der Titel des aktuellen Access-Hauptfensters ermittelt werden. Die API-Prozedur hat die folgende Deklaration:

```
Declare Function GetWindowText Lib "user32" Alias _
    "GetWindowTextA" (ByVal hWnd As Long, ByVal _
    lpString As String, ByVal cch As Long) As Long
```

Der erste Eintrag in der Parameterliste ist der Parameter **hWnd**. Für ihn müssen Sie das Handle des Fensters übergeben, dessen Fenstertext Sie ermitteln möchten. Sie müssen also zunächst das Handle bestimmen. Wenn Sie die Funktion von Access aus aufrufen, ist das gewünschte Fenster das aktuelle Fenster. Das Handle des aktuellen Fensters ermitteln Sie mit der wie folgt deklarierten API-Prozedur:

```
Declare Function GetActiveWindow Lib "user32" () As Long
```

Sie können nun beide API-Funktionen in einer Funktion verwenden (s. Listing 3).

Die Funktion **FenstertitelErmitteln** legt wie gewohnt zunächst die an die API-Funktion zu übergebenden Parameter fest. Dabei wird das Handle auf das aktuelle

Ticketsystem, Teil 1

Wer Access-Anwendungen entwickelt, erledigt dies in der Regel für seine Kunden oder Benutzer. Ob es nun einer, wenige oder viele Benutzer sind: Früher oder später meldet sich der eine oder andere mit Fehlermeldungen oder Verbesserungswünschen. Davon ausgehend, dass Sie diese auch berücksichtigen möchten, ist hier ein funktionierendes System zur Erfassung und Abarbeitung der Anforderungen gefragt, gegebenenfalls auch für mehrere Benutzer. Wir gehen in diesem Fall von einer reinen E-Mail-Schnittstelle aus und wollen dazu Microsoft Outlook nutzen.

Aufgaben des Ticketsystems

Das Ticketsystem soll eine ganze Reihe von Aufgaben erleichtern. Die erste Aufgabe ist, die per E-Mail eingehenden Anfragen halbautomatisch zu erfassen. Halbautomatisch deshalb, weil eine vollautomatische Erfassung voraussetzen würde, dass alle derartigen Anfragen zumindest an eine eigens dafür vorgesehene E-Mail-Adresse gerichtet sind – was nicht unbedingt immer der Fall ist, wie etwa beim Autor dieser Zeilen.

E-Mails, die eine Anfrage enthalten, die mit dem Ticket-System verarbeitet werden soll, soll der Benutzer in Outlook in einen speziell dafür vorgesehenen Outlook-Ordner verschieben. Diesen statten wir natürlich per VBA-Code so aus, dass die E-Mail dann automatisch weiterverarbeitet wird und in der Datenbank des Ticketsystems landet.

E-Mails, die Bedingungen erfüllen, die auf eine Support-Anfrage hindeuten, können natürlich automatisch per Outlook-Regel erfasst und in den dafür vorgesehenen Outlook-Ordner verschoben werden, von wo aus die Mail wie oben beschrieben weiterverarbeitet wird.

Wenn eine E-Mail in Form eines Tickets im Ticketsystem angelegt wurde, sollen verschiedene Schritte erfolgen: Zum Beispiel muss die E-Mail so ausgewertet werden, dass sich ein Kunde daraus ableiten lässt. Die Kunden speichert das Ticketsystem in einer eigenen Tabelle, wobei wir uns hier auf die Daten Anrede, Vorname und Nachname beschränken wollen – und natürlich die E-Mail-Adresse. Hier wird es interessant, denn ein Kunde kann durchaus mehr als eine E-Mail-Adresse verwenden. Also müssen wir die E-Mail-Adressen separat in einer eigenen Tabelle verwalten.

Sobald einmal ein Kundendatensatz für eine E-Mail-Adresse angelegt wurde, sollen eingehende E-Mails mit dieser Absenderadresse automatisch dem Kunden zugeordnet werden.

Bei der Bearbeitung eines Tickets können verschiedene Aufgaben anfallen:

- das Erstellen einer Notiz,
- das Erstellen einer Aufgabe oder
- das Beantworten der E-Mail.

Notizen haben lediglich informativen Charakter und sollen dafür sorgen, dass Informationen nicht verloren gehen. Aufgaben sollen sortiert und bearbeitet werden können, damit man sieht, welche Teilschritte zum Fertigstellen eines Tickets nötig und welche bereits erledigt sind. E-Mail-Antworten enthalten beispielsweise Rückfragen oder aber die Mitteilung, ob und wie das Problem gelöst werden konnte. Die jeweils erste Antwort zu einem Ticket soll im Betreff mit einer eindeutigen ID ausgestattet werden, die der Kunde in seiner Antwort weiterverwenden soll. Auf diese Weise braucht man nicht mehr die E-Mail-Adresse der Antworten heranzuziehen, um die Antwort zuzuordnen, sondern kann diese direkt dem Ticket zuweisen.

Um auch dem Autor dieses Beitrags den Alltag zu erleichtern, wollen wir für die Antworten auf die Kundenanfragen auch einige Textbausteine vorsehen, die häufiger vorkommende Anfragen beantworten. Diese Textbausteine sollen natürlich mit Platzhaltern versehen und mit den Daten des Kunden gefüllt werden.

Und wenn Sie schon Kundenanfragen annehmen und diese mit einem Ticketsystem verwalten wollen, können Sie auch gleich die Reaktionszeiten tracken und entsprechende Statistiken ausgeben lassen.

Den Rahmen eines einzigen Beitrags sprengt diese Lösung bei Weitem, daher finden Sie die einzelnen Elemente auf verschiedene Beiträge aufgeteilt. Wir verweisen von diesem Hauptbeitrag aus auf weiterführende Beiträge.

E-Mails und Tickets

Den Ausgangspunkt für ein Ticket bildet eine E-Mail-Anfrage (diese kann beispielsweise direkt per Mail oder auch über ein Kontaktformular erstellt worden sein – wichtig ist, dass diese als E-Mail in Outlook eingeht und dort im entsprechenden Ordner für die Verarbeitung von E-Mail-Anfragen landet).

Die E-Mail-Anfrage liefert uns einige Daten: die Absender-E-Mail-Adresse, gegebenenfalls den Vor- und den Nachnamen des Kunden sowie einen Text, der die Beschreibung des Problems enthält. Aus diesen Informationen möchten wir folgende Aktionen ableiten:

- Erstellen eines Kundendatensatzes, sofern dieser noch nicht vorhanden ist, oder

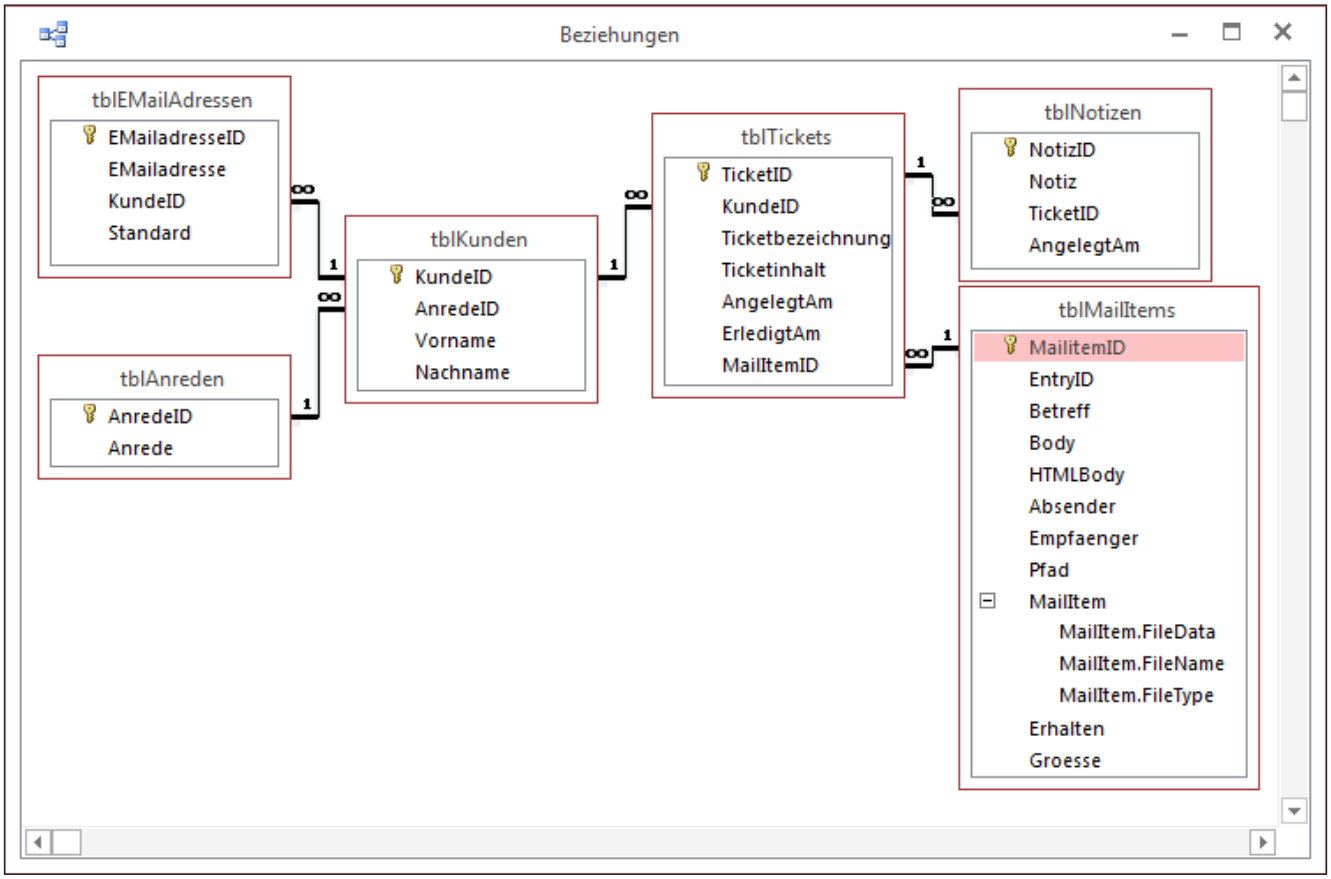

Bild 1: Datenmodell für den ersten Teil der Beitragsreihe

- Auswählen des Kundendatensatzes, welcher der E-Mail-Adresse des Absenders zugeordnet werden kann,

- Anlegen eines Tickets als neuen Datensatz in einer Ticket-Tabelle – inklusive Verweis auf die Ursprungs-E-Mail, den Kundendatensatz sowie mit einer Bezeichnung und dem Inhalt des Tickets.

Datenmodell

Das Datenmodell sieht bis hierher wie in Bild 1 aus. Die Tabelle **tblKunden** ist über das Feld **AnredeID** mit der Tabelle **tblAnreden** verknüpft. Jedem Kunden können beliebig viele E-Mail-Adressen zugeordnet werden, also gibt es eine Tabelle namens **tblEMailAdressen**, die über das Fremdschlüsselfeld **KundeID** mit der Tabelle **tblKunden** verknüpft ist.

Zu jedem Kunden kann es natürlich auch beliebig viele Tickets geben. Daher ist die Tabelle **tblTickets** ebenfalls über ein Fremdschlüsselfeld, wieder namens **KundeID**, mit der Tabelle **tblKunden** verknüpft.

Diese Tabelle enthält neben dem Primärschlüsselfeld und dem Fremdschlüsselfeld **KundeID** noch folgende weitere Felder:

- **Ticketbezeichnung**: Nimmt die Bezeichnung des Tickets auf.

- **Ticketinhalt**: Ist ein Memofeld mit Formatierung **Rich-Text**, um den Inhalt des Tickets zu beschreiben.

- **AngelegtAm**: Datum, an dem das Ticket angelegt wurde.

- **ErledigtAm**: Datum, an dem das Ticket abgeschlossen wurde.

- **MailItemID**: Fremdschlüsselfeld zu einem Eintrag der Tabelle **tblMailItems**. Speichert den Bezug zu der E-Mail, auf deren Basis das Ticket entstanden ist.

Die Tabelle **tblMailItems** kennen Sie vielleicht bereits: Wir haben diese im Beitrag **Outlook-Mails nach Empfang archivieren** (**www.access-im-unternehmen.de/1007**) vorgestellt. Wir werden auch die dort erläuterten Techniken zum halbautomatischen Einlesen von E-Mails in diese Tabelle in abgewandelter Form verwenden.

Fehlt noch die Tabelle **tblNotizen**, die wir in diesem Teil noch nicht erläutern werden, aber die zum Speichern von Notizen zum jeweiligen Ticket verwendet wird. Im Beitrag **HTML-Liste mit Access-Daten** (**www.access-im-unternehmen.de/1029**) erfahren Sie schon einmal, wie die Notizen später dargestellt werden sollen.

Kunden und E-Mail-Adressen

Wir schauen uns zuerst das Formular an, mit dem wir die Kunden und deren E-Mail-Adressen verwalten. Dieses sieht im Entwurf wie in Bild 2 aus und verwendet die Tabelle **tblKunden** als Datenherkunft. Da das Formular nur jeweils einen Datensatz zum Bearbeiten anzeigen soll beziehungsweise zum Anlegen eines neuen Kunden verwendet werden soll, stellen Sie die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**, **Trennlinien** und **Bildlaufleisten** auf den Wert **Nein** ein. Außerdem erhält **Automatisch zentrieren** den Wert **Ja**.

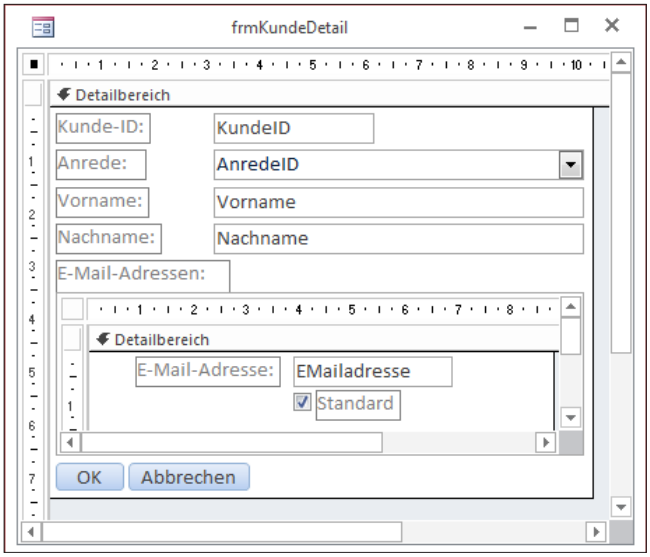

Bild 2: Entwurf des Formulars **frmKundeDetail**

Die E-Mail-Adressen zum jeweiligen Kunden steuert das Unterformular **sfmEMailAdressen** bei. Als Datenherkunft dient hier die Tabelle **tblEMailAdressen**. Das Formular soll seine Daten in der Datenblattansicht darstellen, also verwenden Sie für die Eigenschaft **Standardansicht** den Wert **Datenblatt**.

Wenn Sie das Unterformular **sfmEMailAdressen** aus dem Navigationsbereich in den Entwurf des Hauptformulars **frmKundeDetail** ziehen, sollte Access die Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformular-Steuerelements automatisch jeweils auf den Wert **KundeID** einstellen – falls nicht, holen Sie dies manuell nach.

Beim Öffnen der Kundendetails

Das Formular **frmKundeDetails** soll entweder zum Anlegen eines neuen Datensatzes oder zum Bearbeiten eines vorhandenen Datensatzes geöffnet werden.

Im Falle des Anlegens sollen gleich ein paar Standardwerte gesetzt werden; außerdem gehen wir hier davon aus, dass der Benutzer gleich die E-Mail-Adresse des zu erstellenden Benutzers übergibt – und zwar mit **OpenArgs**-Parameter.

Der Aufruf würde dann etwa so aussehen:

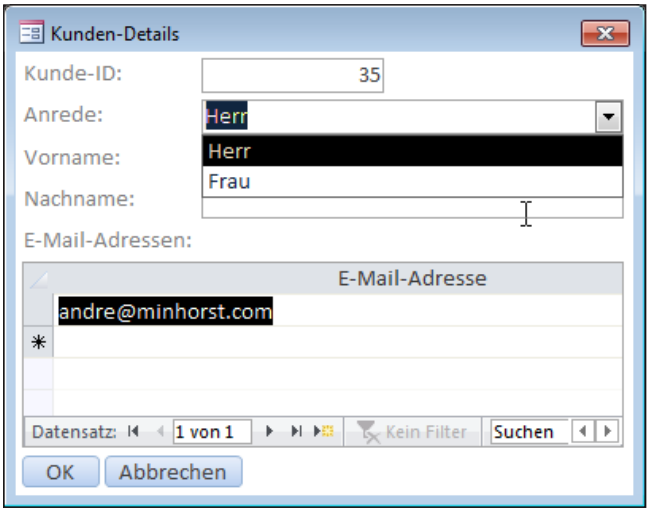

Bild 3: Das Formular **frmKundeDetail** beim Anlegen eines neuen Kunden

```
Private Sub Form_Load()
    Select Case Me.DefaultEditing
        Case 1
            If Not Len(Nz(Me.OpenArgs, "")) = 0 Then
                Me!sfmEMailAdressen.Form!txtEMailadresse.DefaultValue = Chr(34) & Me.OpenArgs & Chr(34)
            Else
                Me!sfmEMailAdressen.Form!txtEMailadresse.DefaultValue = Chr(34) & "<E-Mail-Adresse eingeben>" & Chr(34)
            End If
            Me!AnredeID.SetFocus
            Me!AnredeID.DefaultValue = 1
            Me.Dirty = True
            Me!sfmEMailAdressen.SetFocus
            Me!sfmEMailAdressen.Form!txtEMailadresse.SetFocus
            Me!sfmEMailAdressen.Form.Dirty = True
            Me!sfmEMailAdressen.Form!txtEMailadresse.DefaultValue = Chr(34) & Chr(34)
            Me!AnredeID.SetFocus
            Me!AnredeID.Dropdown
    End Select
End Sub
```

Listing 1: Laden des Formulars **frmKundeDetail**