

ACCESS

IM UNTERNEHMEN

BELEGUNGSPLAN

Lernen Sie einen Belegungsplan mit Kalenderansicht kennen (ab S. 44).



In diesem Heft:

TREEVIEW-ERSATZ

Verhindern Sie Update-Probleme mit dem MSCOMCTL-TreeView durch ein alternatives Treeview-Steurelement

KALENDER- STEUERELEMENTE

Lernen Sie verschiedene Arten von Kalendersteuerelementen kennen.

NOTIZEN NACH KUNDE

Erweitern Sie eine Kundenverwaltung um eine Funktion zum übersichtlichen Speichern von Notizen.

SEITE 2

SEITE 15

SEITE 56

Von Ast zu Ast

Für viele Anwendungen ist das TreeView-Steuerelement ein existenzieller Bestandteil. Es zeigt die Daten mehrerer hierarchisch verknüpfter Tabellen an oder auch die Daten aus einer reflexiv verknüpften Tabelle. Daher ist es wichtig, dass dieses Steuerelement wie alle anderen eingebauten und von Microsoft gelieferten Elemente einwandfrei funktionieren. Leider ist das nicht durchgängig der Fall, was sowohl für Entwickler als auch für die Benutzer einer Software unangenehm ist.



Auch wenn die Entwicklung rund um das TreeView-Steuerelement aus der MSCOMCTL-Bibliothek recht aufwändig ist, wenn man die Daten aus den Tabellen einer Anwendung darin anzeigen möchte, nutzen viele Entwickler dieses Steuerelement. Der Grund für den hohen Aufwand ist die fehlende Datenbindung. Während Sie Formulare und Berichte sowie Steuerelemente wie Textfelder, Listfelder oder Kombinationsfelder ganz einfach durch die Angabe einer Tabelle oder Abfrage mit Daten füllen können, müssen Sie dies beim TreeView-Steuerelement (und übrigens auch beim ListView) komplett per Code erledigen. Dazu durchlaufen Sie die entsprechenden Datensätze und fügen jedes einzelne Element einzeln hinzu.

Wer so viel Arbeit in die Programmierung der Benutzeroberfläche steckt, möchte sich natürlich auch darauf verlassen können, dass diese durchgängig funktioniert. Leider ist es in letzter Zeit immer wieder dazu gekommen, dass Microsoft Updates für verschiedene Office-Versionen geliefert hat, welche die Funktion der Bibliothek **MSCOMCTL.ocx**, die das TreeView und andere Steuerelemente liefert, komprimieren. Das heißt, dass plötzlich bei allen betroffenen Anwendern Fehlermeldungen auftauchen, wenn er eines der mit dem TreeView-Steuerelement ausgestatteten Formulare öffnet. Daten zeigt das TreeView-Steuerelement dann natürlich auch nicht mehr an.


Es gibt zwar ein paar Tricks und Kniffe, mit denen man das TreeView-Steuerelement wieder zum Laufen bringt. Dazu gehört das Einspielen einer älteren Version der Datei **MSCOMCTL.ocx**, das Rückgängigmachen des Updates et cetera. Das ist noch gangbar, wenn es sich nur um die

eigene Entwicklermaschine handelt, aber wenn Sie mehrere zig oder hundert Kunden haben, welche eine Ihrer Anwendungen nutzen und die plötzlich nicht mehr damit arbeiten können, erzeugt das eine Menge Stress, Arbeit und somit auch Kosten.

Wer sich vor einer solchen Situation schützen möchte, hat zwei Möglichkeiten: Entweder er greift auf eine Bibliothek von einem Drittanbieter zu, die dann allerdings separat installiert werden muss und ihrerseits Probleme machen kann. Oder er nutzt die Lösung des Anbieters **picoware**, die wir in der aktuellen Ausgabe in einem Praxisbeitrag vorstellen. **picoware** hat ein Treeview programmiert, das allein durch Bordmittel realisiert wird – in diesem Fall durch die geschickte Programmierung eines Unterformulars in der Endlosansicht.

Dieses Treeview ist nicht nur updatesicher, sondern auch die Programmierung macht auch eine Menge mehr Spaß als die des MSCOMCTL-TreeViews: Sie können es beispielsweise allein durch die Angabe einer geeigneten SQL-Anweisung mit Daten füllen. Der kleine Haken: Auch die Programmierung dieses Tools kostet Zeit und Arbeit und somit ist das picoware-Treeview nicht kostenlos. Aber schauen Sie sich doch erstmal unseren Beitrag ab S. 2 an, wo Sie auch einen Gutscheincode für einen satten Rabatt finden.

Und nun: Viel Spaß beim Lesen!



Ihr André Minhorst

Treeview ohne MSCOMCTL

Im Juli/August 2017 war es wieder mal soweit: Microsoft hat ein Update für einige Office-Varianten geliefert, das die TreeView-Steuer-elemente in den Anwendungen auf den betroffenen Rechnern lahmgelegt hat. Zwar gibt es ein paar Wochen später immer einen Patch oder ein weiteres Update zur Behebung des Fehlers, aber wer kurzfristig handeln muss, darf manuell an der Datei MSCOMCTL.ocx und/oder der Registry herumfuschen. Das kann man natürlich keinem Kunden zumuten, daher zeigen wir eine mögliche Lösung.

Normalerweise beschreiben wir in **Access im Unternehmen** keine Tools von Drittanbietern, die kostenpflichtig sind. Im Falle des Treeviews scheint es jedoch sinnvoll, eine Ausnahme zu machen. Thomas Pfoch hat mit seiner Firma **picoware** einen **TreeView**-Ersatz programmiert, der komplett ohne zusätzliche Dateien wie **.ocx**- oder **.dll**-Dateien auskommt.

Er verwendet stattdessen ein Unterformular, das er geschickt aufbohrt, um Daten nach den Vorgaben des Benutzers anzuzeigen und diese – wie es für ein Treeview typisch ist – für die untergeordneten Ebenen nach rechts einzurücken.

Wenn Sie diese Erweiterung nutzen, müssen Sie lediglich einige Objekte aus der Beispieldatenbank, die Sie nach dem Erwerb des Pakets erhalten, in die Zieldatenbank kopieren. Darunter befindet sich auch ein Formular, das Sie als Unterformular in das Zielformular einfügen.

Im Code, der beim Laden des Zielformulars ausgelöst wird, tragen Sie dann die Anweisungen ein, welche die anzuzeigenden Daten definieren.

Der picoware-Treeview kostet 299,- EUR (inkl. 19% MwSt.) pro Entwicklerlizenz. Das heißt, dass Sie für jeden Entwickler, der das Tools einsetzt, eine Lizenz benötigen. Dieser Entwickler kann allerdings beliebig viele Anwendungen mit den Treeview-Funktionen ausstatten und weitergeben. Gemessen an dem Ärger, den die re-

gelmäßigen Updates der Datei **MSCOMCTL.ocx** mit sich bringen, ist dies nach unserem Ermessen eine sinnvolle Investition.

Hinzu kommt, dass Sie als Leser von **Access im Unternehmen** im Shop unter www.amvshop.de 20% Rabatt erhalten, und zwar mit dem Gutscheincode **aiu-treeview**.

Vorteile des picoware-Treeviews

Zu den Vorteilen des picoware-Treeviews gehören die folgenden:

- Es ist keine externe Komponente mehr erforderlich, die durch Updates kompromittiert werden könnte.
- Es funktioniert nicht nur unter 32bit-Versionen von Access, sondern auch unter 64bit.
- Die Elemente können per spezieller SQL-Abfrage und somit mit wesentlich weniger Code zugewiesen werden.

Nachteile des picoware-Treeviews

Durch die Anpassung an die Eigenarten einer Datenbank mit ihren in Tabellen gespeicherten Daten und durch die Verwendung eines Unterformulars zur Realisierung Treeviews ergeben sich auch ein paar Nachteile:

- Bestehende MSCOMCTL-TreeViews lassen sich nicht ohne weiteres in das picoware-Treeview umwandeln.

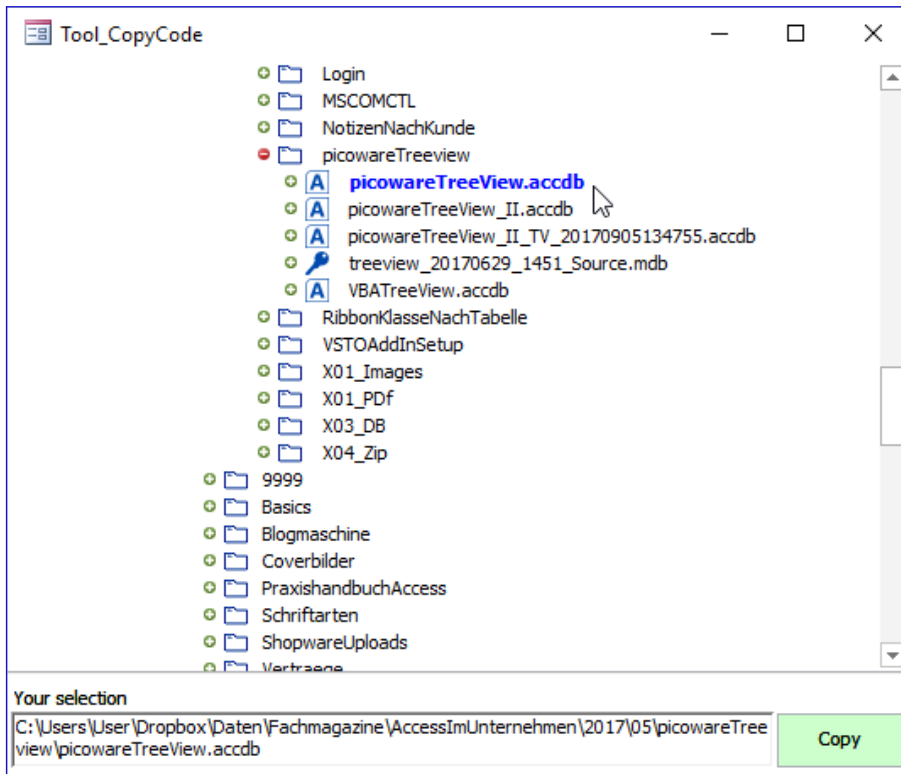


Bild 1: Auswahl der Zieldatenbank für die Datenbankobjekte

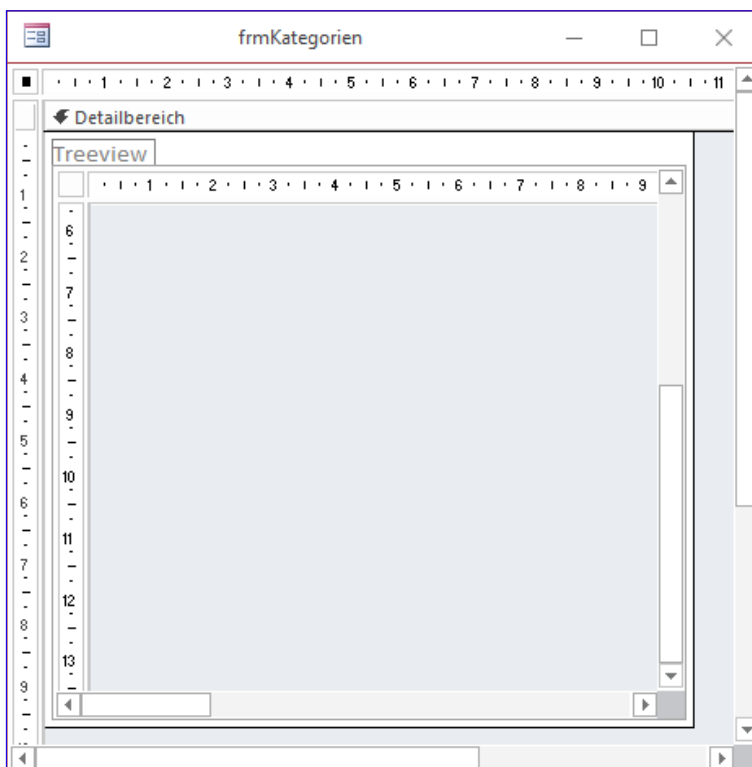


Bild 2: Einfügen des Unterformulars und Anpassen der Größe

- Durch die Verwendung eines Unterformulars kommt es beim Aktualisieren von Daten und anderen Vorgängen manchmal zum Neuaufbau des Treeviews.

Beispieldatenbank

Wenn Sie nicht sicher sind, ob das Tool das Richtige für Sie ist, können Sie unter dem Link amvshop.de/access-tools/322/picoware-treeview Beispieldatenbanken für verschiedene Access-Versionen herunterladen, welche die Möglichkeiten des Tools demonstrieren. Auf den folgenden Seiten zeigen wir Ihnen, wie Sie das **picoware-Treeview** programmieren.

picoware-Treeview-Elemente hinzufügen

Nach dem Erwerb und dem Download erhalten Sie eine Datenbank etwa namens **tree-view_20170629_1451_Source.mdb**. Diese enthält alle Objekte, die Sie für den Einsatz in eigenen Datenbanken benötigen.

Sie brauchen sich keine Mühe zu machen, die benötigten Objekte selbst in die Zieldatenbank zu ziehen. Die Datenbank aus dem Download enthält ein Tool, das Ihnen diese Arbeit abnimmt. Dazu öffnen Sie das Formular namens **Tool_CopyCode**. Dieses zeigt eine Treeview-Ansicht der Verzeichnis- und Dateistruktur auf Ihrer Festplatte an, wobei das Verzeichnis der Quelldatenbank geöffnet wird (s. Bild 1).

Wählen Sie hier die Zieldatenbank aus und klicken Sie auf die Schaltfläche **Copy**. Dies kopiert nun alle notwendigen Datenbankobjekte in die

ausgewählte Datenbank. Falls nötig, halten Sie dabei die Umschalttaste gedrückt.

Erstes Beispiel: Kategorien auflisten

Im ersten Beispiel wollen wir einfach die Daten der Tabelle **tblKategorien** im Treeview anzeigen. Dazu erstellen Sie ein neues Formular namens **frmKategorien**, öffnen es in der Entwurfsansicht und ziehen das Formular **USys_pTV_TreeView** als Unterformular in den Entwurf. Stellen Sie den Namen des Unterformular-Steuerelements auf **sfmTreeView** ein.

Passen Sie außerdem die Größe des Unterformulars an, das initial in der vollen Breite eingefügt wird (s. Bild 2). Stellen Sie außerdem die Eigenschaften **Bildlaufleisten**, **Datensatzmarkierer**, **Navigationsschaltflächen** und **Trennlinien** auf den Wert **Nein** ein – das Hauptformular selbst zeigt keine Daten an, sodass wir diese Elemente nicht benötigen.

Ein erster Wechsel in die Formularansicht liefert natürlich noch kein sinnvolles Ergebnis – kein Wunder, denn wir füllen das Treeview ja auch noch nicht mit Daten (s. Bild 3). Das ändern wir allerdings in den folgenden Schritten.

Dazu fügen Sie dem Klassenmodul des Formulars **frmKategorien**, das Sie durch Einstellen der Eigenschaft **Enthält Modul** des Formulars anlegen, eine Objektvariable hinzu, welche das Unterformular zur Anzeige des Treeviews referenzieren soll:

```
Public WithEvents objTreeView As 7
    Form_USys_pTV_TreeView
```

Die Deklaration enthält das Schlüsselwort **WithEvents**, damit wir innerhalb der Klasse auch die Ereignisse dieses Formulars beziehungsweise der Formulkasse **Form_USys_pTV_TreeView** implementieren können.

Damit wir das Unterformular zur Anzeige des Treeviews füllen können, weisen wir der Objektvariablen **objTree-**

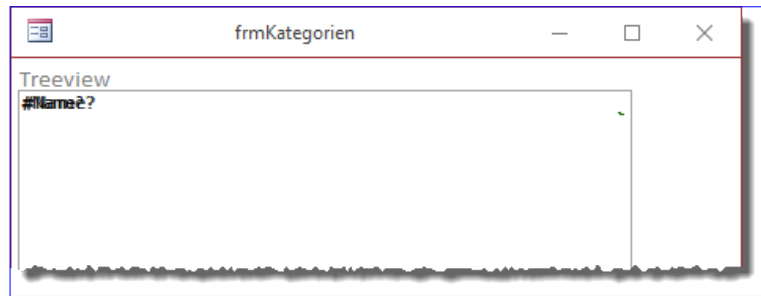


Bild 3: Erster Wechsel in die Formularansicht

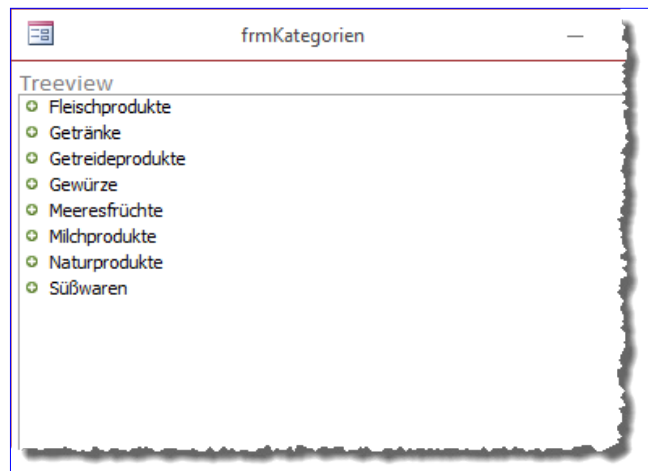


Bild 4: Die Kategorien im picoware-Treeview

View das im Unterformularsteuerelement **sfmTreeView** gespeicherte Formular zu, und zwar in der Prozedur, die durch das Ereignis **Beim Öffnen** des Hauptformulars ausgelöst wird:

```
Private Sub Form_Open(Cancel As Integer)
    Set objTreeView = Me!TreeView.Form
End Sub
```

Treeview mit Daten füllen

Damit wäre das Treeview-Unterformular schon einmal referenziert. Nun wollen wir diese noch mit den gewünschten Daten füllen, in diesem Falle den Namen der Kategorien aus der Tabelle **tblKategorien**.

Beim **MSCOMCTL.ocx**-TreeView hätten Sie jedes Element einzeln mit der **Add**-Methode hinzufügen müssen. Hier spielt das speziell auf die Anzeige von Daten aus Tabellen

oder Abfragen ausgelegte pico-ware-Treeview seine Stärken aus. Sie müssen lediglich eine SQL-Anweisung mit einem bestimmten Format definieren, welche die anzuzeigenden Daten liefert, und diese mit der **AddSQL**-Methode dem in der Variablen **objTreeView** gespeicherten Treeview zuweisen. Das sieht dann beispielsweise wie in Listing 1 aus.

Ein Wechsel zur Formularansicht liefert bereits ein akzeptables Ergebnis (s. Bild 4) – und das mit einer sehr überschaubaren Anzahl von Codezeilen!

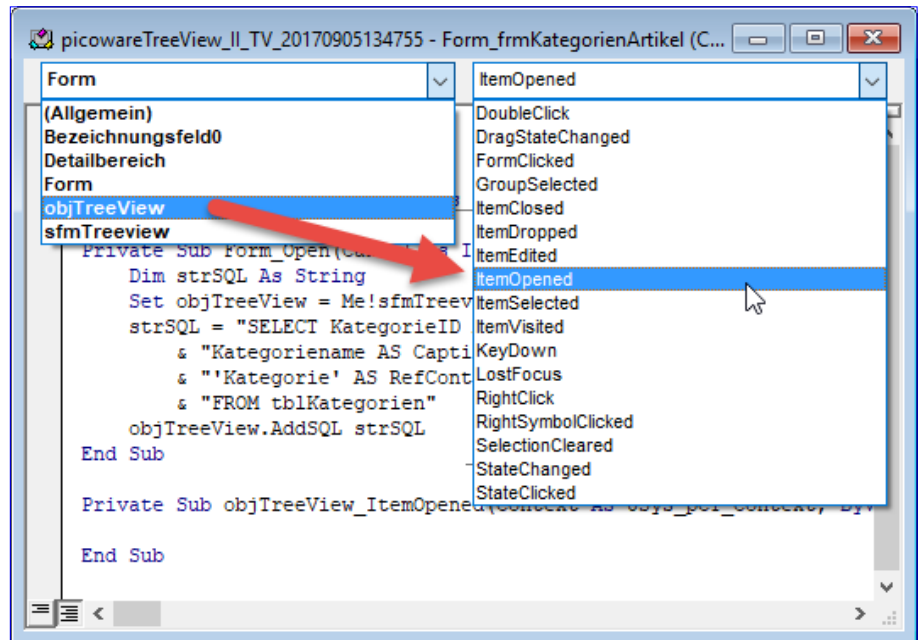


Bild 5: Implementieren einer Ereignisprozedur des picoware-Treeviews

Erläuterung des SQL-Ausdrucks

Der zum Füllen des Treeviews verwendete SQL-Ausdruck sieht zusammengefasst wie folgt aus:

```
SELECT KategorieID AS Reference,
Kategorienname AS Caption,
'Kategorie' AS RefContext
FROM tblKategorien
```

Hier erkennen Sie drei Felder, die jeweils mit einem **ALIAS**-Namen ausgestattet wurden. Dies dient dem Zweck, dass das Treeview beim Anzeigen der Daten genau weiß, welches Feld die Daten für welchen Zwecke enthält. Wir verwenden die folgenden drei **ALIAS**-Bezeichnungen:

- **Reference:** Das hier genannte Feld, in diesem Fall **KategorieID**, wird nicht angezeigt, sondern als Referenz verwendet, wenn die untergeordnete Ebene Daten aufnehmen soll, die über ein Fremdschlüsselfeld, das ebenfalls speziell gekennzeichnet wird, entsprechend verknüpfte Daten enthält.
- **Caption:** Das mit dem **ALIAS** namens **Caption** versehene Feld enthält den im Treeview anzuzeigenden Wert, in diesem Fall den Inhalt des Feldes **Kategorienname**.
- **RefContext:** Dieses Feld nimmt eine Zeichenkette auf, mit der Sie den Typ des daraus generierten Elements im TreeView definieren – in diesem Fall **Kategorie**.

```
Private Sub Form_Open(Cancel As Integer)
    Dim strSQL As String
    Set objTreeView = Me!treeview.Form
    strSQL = "SELECT KategorieID AS Reference, Kategorienname AS Caption, 'Kategorie' AS RefContext " _
        & "FROM tblKategorien"
    objTreeView.AddSQL strSQL
End Sub
```

Listing 1: Anzeigen der Kategorien beim Öffnen des Formulars

```
Private Sub objTreeView_ItemOpened(Context As USys_pCT_Context, ByVal RefPath As String)
    Dim strSQL As String
    Dim lngReference As Long
    lngReference = Context.SettingLong("Reference", -1)
    strSQL = "Select ArtikelID As Reference, 'Artikel' As RefContext, Artikelname As Caption
            FROM tblArtikel WHERE KategorieID = " & lngReference

    objTreeView.AddSQL strSQL, RefPath
End Sub
```

Listing 2: Ereignisprozedur, welche die Artikel für die angeklickte Kategorie einblendet

Im folgenden Beispiel werden Sie sehen, wie Sie die hierfür angegebene Zeichenkette nutzen können.

Artikel zu Kategorien hinzufügen

Nun gehen wir einen Schritt weiter und fügen in einer zweiten Ebene noch die zu einer jeden Kategorie gehörenden Artikel hinzu. Dazu kopieren Sie das soeben erstellte Formular und fügen es unter dem Namen **frmKategorienArtikel** erneut in die Datenbank ein (**frmKategorie** markieren, **Strg + C**, **Strg + V**, neuen Namen eingeben).

Am Entwurf des Formulars brauchen Sie keine Änderungen vorzunehmen, wir kümmern uns nur um den Code.

Wie zuvor soll beim Öffnen des Formulars die Liste der Kategorien erscheinen. Beim Klick auf eines der Plus-Zeichen sollen die zur jeweiligen Kategorie gehörenden Artikel im Treeview angezeigt werden.

Dazu implementieren wir ein Ereignis der Klasse **Form_USys_pTV_TreeView**. Zu diesem Zweck wechseln Sie zum VBA-Fenster des Klassenmoduls des Formulars **frmKategorienArtikel**, wählen im linken Kombinationsfeld des Fensters den Eintrag **objTreeView** und im rechten Kombinationsfeld den Eintrag **ItemOpened** aus (s. Bild 5).

Dies legt die folgende Ereignisprozedur an:

```
Private Sub objTreeView_ItemOpened(Context As
    USys_pCT_Context, ByVal RefPath As String)

End Sub
```

Die Ereignisprozedur wird immer ausgelöst, wenn der Benutzer auf einen der Einträge im Treeview klickt. Dies können Sie ausprobieren, indem Sie einen Haltepunkt für die erste Zeile der Prozedur festlegen und auf eine der Kategorien im Beispielformular **frmKategorienArtikel** klicken.

Nun wollen wir die Zeilen hinzufügen, die dafür sorgen, dass beim Anklicken des Plus-Zeichens einer Kategorie die untergeordneten Artikel im Treeview erscheinen. Diese Prozedur sieht wie in Listing 2 aus.

Eine zusätzliche Zeile deklariert eine Variable namens **lngReference** mit dem Datentyp **Long**, eine weitere füllt diese mit einem Wert, den der Ausdruck **Context.SettingLong("Reference", -1)** liefert. **Context** ist einer der beiden Parameter, der beim Auslösen der Ereignisprozedur übergeben wird.

Dieser liefert einige Informationen rund um das angeklickte Element und auch einige mögliche Methoden. In diesem Fall verwenden wir die Funktion **SettingLong**, um den Wert des Attributs **Reference** zu ermitteln. Dabei handelt es sich um den Wert des angeklickten Elements, der in der SQL-Anweisung, welche dieses Element definiert, für das Feld mit dem **ALIAS**-Namen **Reference** übergeben wurde – in diesem Fall also der Wert des Feldes **KategorieID** für den angeklickten Eintrag.

Diesen Wert speichern wir in der Variablen **lngReference** und nutzen ihn in der nachfolgend zusammengestellten SQL-Anweisung als Vergleichswert des Kriteriums. Für die

KategorieID mit dem Wert **5** sieht die SQL-Abfrage etwa so aus:

```
Select ArtikelID As Reference,
'Artikel' As RefContext,
Artikelname As Caption
FROM tblArtikel
WHERE KategorieID = 5
```

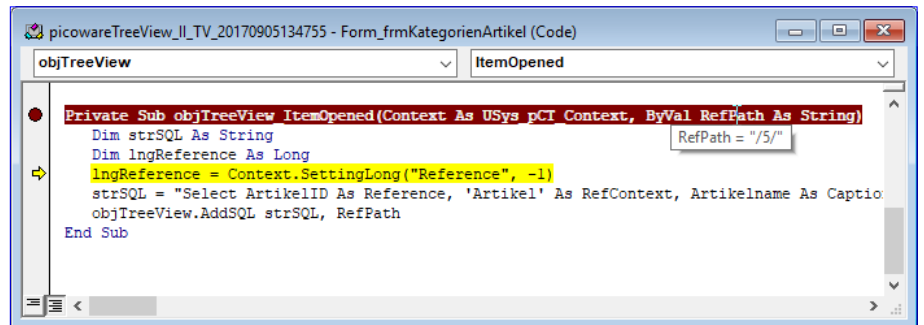


Bild 6: Ermitteln eines der Parameter der Ereignisprozedur

Danach folgt dann auch für die Elemente dieser zweiten Ebene der Aufruf der Methode **AddSQL** des Objekts aus **objTreeView**. Diesmal verwenden wir allerdings auch den zweiten Parameter dieser Methode und übergeben dieser den unveränderten Wert des zweiten Parameters der Ereignisprozedur namens **RefPath**.

Was enthält dieser Parameter? Dies erfahren wir wieder, indem wir einen Haltepunkt setzen und dann nach dem Abarbeiten der ersten Zeile mit der Maus über den Parameter fahren (s. Bild 6).

Es handelt sich also um die Zeichenfolge **/5/**, wobei der Wert **5** wieder dem Referenzwert des auslösenden Elements entspricht. Diesen übergeben wir der Methode **AddSQL**, damit diese weiß, an welches Element sie die neuen Elemente anfügen soll.

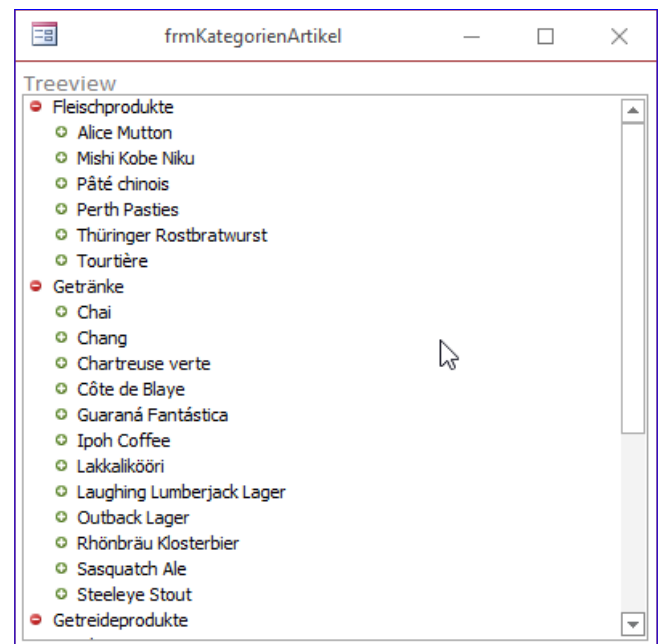


Bild 7: Treeview mit zwei Ebenen

```
Private Sub objTreeView_ItemOpened(Context As USys_pCT_Context, ByVal RefPath As String)
    Dim strSQL As String
    Dim lngReference As Long
    Dim strRefContext As String
    strRefContext = Context.SettingNz("RefContext", "Root")
    Select Case strRefContext
        Case "Kategorie"
            lngReference = Context.SettingLong("Reference", -1)
            strSQL = "Select ArtikelID As Reference, 'Artikel' As RefContext, Artikelname As Caption
                    FROM tblArtikel WHERE KategorieID = " & lngReference

            objTreeView.AddSQL strSQL, RefPath
        End Select
    End Sub
```

Listing 3: Neue Version der Ereignisprozedur, welche die Artikel für die angeklickte Kategorie einblendet

Kalendersteuerelement, Teil 1

Zur Ein- oder Ausgabe eines Datums macht sich ein geeignetes Kalendersteuerelement im Formular besser, als ein schnödes Textfeld. Das kann fest im Formular integriert sein, oder als Popup zur Auswahl erscheinen. In Buchungssystemen im Web sind solche Kalendersteuerelemente allgegenwärtig. Auch Access wurde mit der Version 2007 ein solches Popup-Element spendiert, welches sich aber leider in keiner Weise steuern lässt. Grund genug also, um sich nach Alternativen umzuschauen.

Existierende Kalendersteuerelemente

Einst war ein Kalendersteuerelement in Form der **ActiveX-Datei mscal.ocx** optionaler Bestandteil der **Office-Installation**. Das hat sich seit einiger Zeit geändert. Man ist nunmehr allein auf das Popup angewiesen, das bei Datumsfeldern zur Auswahl erscheint, wenn in das zugehörige Textfeld geklickt wird. Eine dauerhafte Ansicht des Kalenders ist somit verwehrt.

Bild 1 zeigt das alte Kalender-ActiveX-Steuerelement links oben im Formular. Aus irgendeinem Grund befand es sich, möglicherweise aus älteren Office-Installationen, noch bei uns in englischer Version im System. Seine Darstellung lässt sich im Formularentwurf oder auch per VBA steuern. Das betrifft die Schriftarten und die Farbgebung. Es löst bei einigen Aktionen ein Ereignis aus und spiegelt in der Eigenschaft **Value** das markierte Datum wieder. Sollte auf ihrem System die ActiveX-Datei nicht installiert sein, so meldet Access beim Aufruf des Formulars der Beispieldatenbank den etwas eigenartigen Fehler **In diesem Formular befindet sich kein Steuerelement**.

Entfernen Sie in diesem Fall im Entwurf des Formulars **frmCalendarTest** einfach den Platzhalter des Steuerelements.

Da von diesem Fall auszugehen ist, fragt sich, wie mit anderen Mitteln eine ähnliche Darstellung und Funktion zu erreichen wäre. Testweise haben wir ein Listview (rechts unten in der Abbildung) und ein **Microsoft Listview Control** (links unten) als Kalender zu gestalten versucht. Mit einigen Zeilen Code und den entsprechenden Einstellungen für die Steuerelemente gelingt dies auch. Der inakzeptable Nachteil der Lösungen besteht darin, dass sich in diesen Steuerelementen nur ganze Zeilen markieren lassen, nicht aber einzelne Zellen. Eine Auswahl per

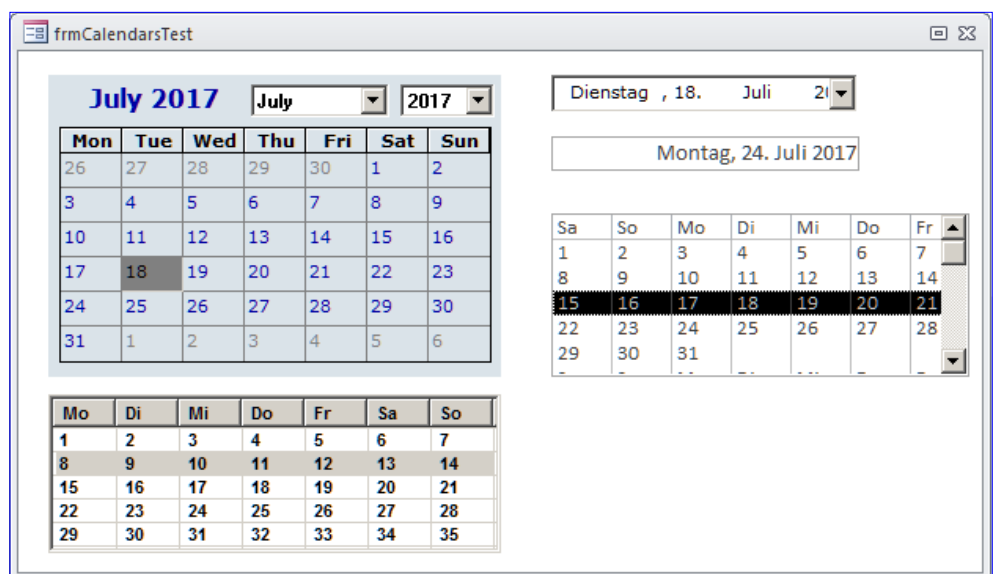


Bild 1: Demo einiger Datums- und Kalendersteuerelemente im Testformular

Maus scheidet deshalb aus. Derlei ließe sich also lediglich zur Anzeige der Tage eines Monats verwenden. Doch das brauchen Sie wohl höchst selten. Bleiben noch zwei Alternativen: das Microsoft **DateTime Picker Control** und die Access-Textbox mit Datumsformatierung. Ersteres zeigt sich im Formular rechts oben. Es entspringt der ActiveX-Datei **mscomctl2.ocx**, die früher ebenfalls mit Office installiert wurde (s. Bild 2).

Wir erwähnen es nur der Vollständigkeit halber, denn außer einer abweichenden Gestalt bietet es gegenüber der dem Popup von Access (s. Bild 3) keine besonderen Vorteile. Nur die Schriftarten und Farben können hier zusätzlich eingestellt werden. Die Funktionalität ist bei beiden jedoch gleich.

Normalerweise öffnet sich das Kalenderelement einer Textbox mit Datumsformatierung dann, wenn Sie auf das Symbol rechts neben dem Textfeld klicken. Es gibt aber eine Anweisung, die die Anzeige des Kalender-Popups erzwingt. Möchten Sie etwa, dass es sofort beim Eintritt in das Feld erscheint, oder auch beim Klicken in das Textfeld, so schreiben Sie die folgende Zeile in die entsprechenden Ereignisprozeduren:

```
Private Sub txtDate_Click()
    RunCommand acCmdShowDatePicker
End Sub
```

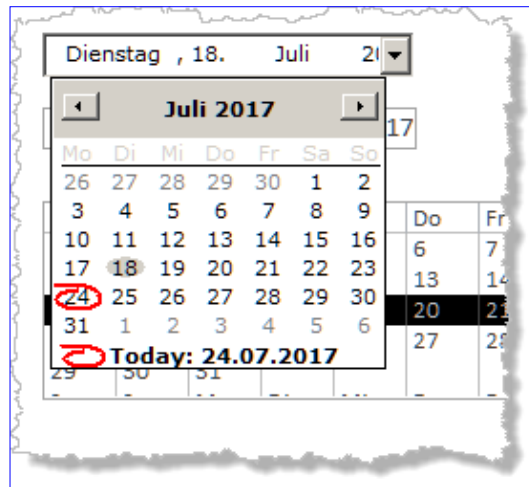


Bild 2: Das Microsoft **DateTime-Picker**- ActiveX-Steuer-element

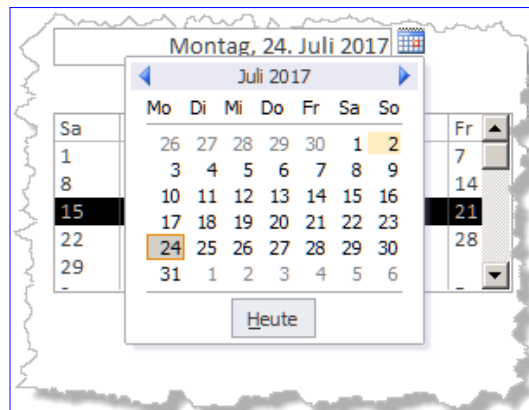


Bild 3: Das Popup-Element zur Datumsauswahl bei Access-Textboxen

```
Private Sub txtDate_Enter()
    RunCommand acCmdShowDatePicker
End Sub
```

Diese **RunCommand**-Anweisung ermöglicht das Öffnen des Popups per Code, falls die Datums-Textbox gerade den Fokus besitzt. Andernfalls ereignet sich eine Fehlermeldung. Verwenden Sie sie deshalb besser nur in den Ereignissen der Textbox selbst.

Sicher finden Sie auch noch weitere Fremdsteuerelemente in Form von ActiveX-Dateien bei Drittanbietern. Es gilt jedoch die Maxime, wegen der möglichen Registrierungsprobleme solche ActiveX-Komponenten nur dann einzusetzen, wenn es absolut notwendig ist. Und das ist hier nicht gegeben, denn ein Kalendersteuerelement lässt sich gut auch mit Access-Bordmitteln selbst erstellen!

Kalender im Eigenbau

Natürlich könnte man einfach 31

Steuerelemente, etwa Buttons oder Textboxen, in einem Formular so anordnen, dass diese einen Kalender ergeben. Um auf eine Datumsauswahl zu reagieren, bräuchte es dafür dann eben auch 31 Ereignisprozeduren. Das wäre zwar recht unkompliziert, ist aber wenig elegant.

Dabei reicht es im Prinzip ja für die Tage der Woche nur sieben Steuerelemente im Detailbereich zu platzieren und diese Wochen untereinander zu wiederholen. Damit sind wir allerdings auf eine Tabelle angewiesen, die die Datensätze für alle Wochen eines Monats bereitstellt. Nur eine Tabelle oder Abfrage als Datensatzherkunft kann bewirken, dass sich der Detailbereich wiederholt.

Bevor es Schritt für Schritt an die Entwicklung des Kalenderformulars geht, sollte noch definiert werden, was der Ausdruck **Steuerelement** in unserem Zusammenhang bedeutet. Ein wirklich neues Steuerelement kann nun mal mit Access nicht erzeugt werden. Sie können lediglich aus bestehenden Elementen eine neue Funktionsgruppe erstellen, die den Anschein eines einzelnen Steuerelements erweckt. Damit das Ganze wieder verwendbar ist, sollte diese Funktionsgruppe möglichst in einem Formular ohne weitere Abhängigkeiten untergebracht werden, welches Sie später als Unterformular in andere einsetzen. Dieses Unterformular stellt dann quasi ein Pseudo-Steuerelement dar. In der Beispieldatenbank nennt es sich **sfrmCalendar**. Das Präfix **s** steht für **Subform**.

Wochentabelle für einen Monat

Die Basistabelle **tblCalendar** für das Kalenderformular in Bild 4 weist lediglich die sieben Datenfelder **D1** bis **D7** von Typ **Long** für die einzelnen Tage der Woche auf. Für die Namen der Felder kann man eher nicht Kürzel für Wochentage, wie **Mo** bis **So**, verwenden, da der Erste eines Monats eben selten ein Montag ist.

Die Nummerierung in den Namen der Felder ermöglicht später den gezielten Zugriff auf die Spalten der Tabelle. Ein Blick auf Bild 5 zeigt, was dahinter steckt.

Die Datensätze füllt nämlich eine VBA-Routine des Formulars zur Laufzeit unter Angabe des gewünschten Jahrs und Monats. Sie ordnet die Zahlen für die Tage so an, dass sich links immer die Montage befinden. Der Erste des Monats wiederum soll in der ersten Zeile auftauchen. Ist das kein Montag, so sind links von diesem Feld die



Bild 4: Die Tabelle **tblCalendar** als Basis für den Kalender im Eigenbau

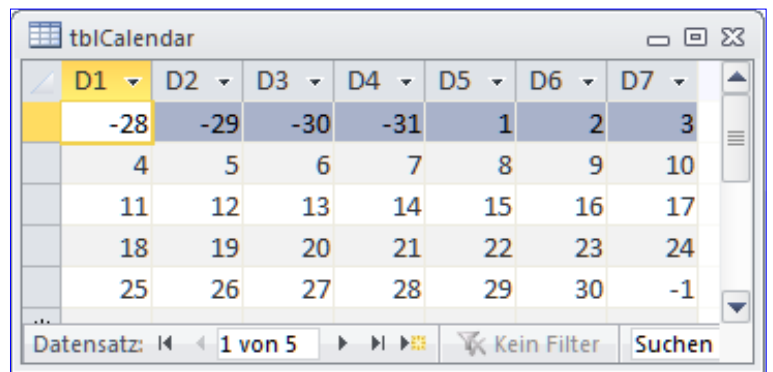


Bild 5: Die gefüllte Tabelle **tblCalendar** im Datenblatt

Tage des Vormonats einzusetzen. Hier sind das der **28.** bis **31.**. Ähnliches gilt für den Letzten des Monats, welcher sich im letzten Datensatz befinden soll, aber in der Regel kein Sonntag ist. Folglich ergeben sich in der letzten Zeile rechts vom Monatsletzten meist noch weitere Tage des Folgemonats. Je nach Monat und Jahr sind in der Tabelle mindestens vier, maximal sechs, Datensätze zu erzeugen.

Meist sind die Tage, welche nicht zum angezeigten Monat gehören, also jene des Vor- und Folgemonats, in Kalendern ausgegraut dargestellt. Um dieses Feature auch unserem Kalender zu spendieren, verwenden wir für die Textfelder im Formular eine **Bedingte Formatierung**. Denn per VBA-Code kann zwar die Hintergrundfarbe einer Textbox über die Eigenschaft **BackgroundColor** gesteuert werden, doch das betrifft dann sämtliche Zellen einer Spalte, die

sich von diesem Datenfeld ableiten, nicht jedoch einzelne Zellen.

Um **Bedingte Formatierung** kommt man hier also nicht herum. Da diese einen Vergleichsausdruck für die Steuerung des Formats verwendet, benötigen wir irgendein Indiz, welches die nicht zum Monat gehörigen Tage definiert. Und das ist hier das Minuszeichen. Alle auszugrauernden Tage haben einen negativen Wert. Damit ist auch klar,

weshalb hier keine Datumstypen für die Felder der Tabelle zum Einsatz kommen, denn deren Werte können nicht negativ sein. Ein **Long**- oder ein **Integer**-Wert reichen zur Kennzeichnung aus.

Der direkte Bezug des Kalenders zu einer Tabelle bringt allerdings einen Nachteil mit sich. Benötigen Sie etwa zwei Kalendersteuerelemente in Ihrem Hauptformular, so wären diese ohne weiteres Zutun beide an dieselbe

```
Private m_Table As String

Property Get Table() As String
    Table = m_Table
End Property
Property Let Table(ByVal Value As String)
    m_Table = Value
    FillCalendar
End Property

Private Sub FillCalendar()
    Dim rs As DAO.Recordset
    Dim i As Long, j As Long
    Dim n As Long
    Dim StartDate As Date
    Me.RecordSource = m_Table
    n = Weekday(DateSerial(m_Year, m_Month, 1), vbMonday)
    StartDate = DateSerial(m_Year, m_Month, 1) - n
    CurrentDb.Execute "DELETE FROM " & m_Table
    Set rs = CurrentDb.OpenRecordset("SELECT * FROM " & m_Table, dbOpenDynaset)
    For j = 0 To 5
        rs.AddNew
        For i = 0 To 6
            StartDate = StartDate + 1
            n = VBA.Month(StartDate)
            rs.Fields("D" & CStr(1 + i)).Value = Day(StartDate) * IIf(n <> m_Month, -1, 1)
        Next i
        rs.Update
        If n <> m_Month Then Exit For
    Next j
    rs.Close
    Me.Requery
End Sub
```

Listing 1: Füllen einer Kalendertabelle über VBA-Code

Tabelle gebunden und zeigten in der Folge auch die gleichen Daten an. In diesem Fall erstellen Sie eine Kopie der Tabelle und bezeichnen diese etwa mit **tblCalendar2**. Die folgend beschriebene Routine zum Füllen der Tabelle kann unterschiedliche Tabellen berücksichtigen.

Füllen der Tabelle per VBA

Die dafür verantwortliche Routine nennt sich **FillCalendar** und ist in Listing 1 abgebildet, wobei hier einige Teile entfernt wurden, die nicht unmittelbar zum Erzeugen der Datensätze gehören.

Der Name der zu füllenden Tabelle steht im Kopf in der Eigenschaftsvariablen **m_Table** des Formularmoduls. Dieser Variablen muss also erst ein Wert zugewiesen werden, was die **Property-Let**-Prozedur **Table** übernimmt:

```
frm.Table = "tblCalendar"
```

Erst dann kann die eigentliche Routine aufgerufen werden. Sie weist dem Formular selbst zunächst als Datensatzherkunft (**RecordSource**) die nun in **m_Table** stehende Tabelle zu. Das Formular ist im Entwurf also noch nicht zwingend an eine Tabelle gebunden, sondern das geschieht hier zu Laufzeit.

Nun benötigen wir jenes Datum, welches in der linken oberen Ecke des Kalenders steht. Über **Property**-Prozeduren (hier nicht im Listing), wurde in den Variablen **m_Month** und **m_Year** der gewünschte Monat und das Jahr für den Kalender abgespeichert. Den Ersten dieses Monats erhalten Sie über die VBA-Funktion **DateSerial**:

```
DateSerial(m_Year, m_Month, 1)
```

Das ist indessen noch nicht das Datum, welches links oben steht. Um zu ermitteln, wie viele Tage des Vormonats zu berücksichtigen sind, kommt die VBA-Funktion **Weekday** zum Einsatz. Sie gibt eine Zahl zurück, die den Wochentag symbolisiert. Die **1** entspricht dabei **Montag**, die **7** dem **Sonntag**. Nun muss vom Ersten des Monats le-

diglich diese Zahl subtrahiert werden, und schon steht das Datum links oben fest. Es wird der Variablen **StartDate** vom Type **Date** zugewiesen. Nach diesen Vorarbeiten kann die Tabelle **m_Table** über zwei verschachtelte Schleifen mit Daten versehen werden.

Doch vorher muss die Tabelle noch über die **Execute**-Anweisung und den **SQL-DELETE**-Ausdruck geleert werden. Anschließend öffnet ein Recordset **rs** die Datensätze zum Beschreiben.

Die Zählervariable für die Schleife zum Hinzufügen von Datensätzen ist **j**. Da maximal sechs Zeilen im Kalender stehen können, ist ihr Bereich auf **0** bis **5** eingestellt. Nach jedem Durchlauf wird per **AddNew** ein neuer Datensatz erzeugt. Für den Zugriff auf die Feldwerte eines Datensatzes gibt es dann die folgende Schleife auf den Zähler **i**, deren Bereich sich für die einzelnen Wochentage von **0** bis **6** erstreckt. In dieser wird fortlaufend der Wert des Kalenderdatums in **StartDate** um eins erhöht. Das ist statthaft, weil ein **Date**-Type imgrunde ein **Double**-Wert ist, wobei die Nachkommastellen die Tageszeit angeben, die Vorkommastellen die Tage. Also führt Addition von **1** zum nächsten Tag.

Der Zugriff auf die Datenfelder geschieht namentlich über das Präfix **D** und den Zähler **i**, zu dem noch **1** addiert werden muss, weil die Felder von **1** bis **7** nummeriert sind, nicht von **0** bis **6**. Der Wert eines Felds errechnet sich aus dem Tagesanteil des Datums, welchen die VBA-Funktion **Day** zurückgibt.

Im Prinzip wäre es das auch schon, wollten wir nicht die Tage außerhalb des Zielmonats mit einem Minuszeichen versehen werden. Um jene zu eruieren, wird in der Variablen **n** der Monatsanteil des Datums über die Funktion **Month()** zwischengespeichert. Weicht dieser Wert vom Zielmonat ab (**n <> m_Month**), so greift die **IIf**-Bedingungsfunktion (entspricht **Wenn()** in Abfragen), die entweder eine **-1** oder eine **1** als Ergebnis zeitigt. Und das ist eben der Multiplikator für den Tageswert.

Da die äußere Schleife immer von **0** bis **5** zählt, würden auch immer sechs Zeilen im Kalender stehen. Um das etwa für den Monat **Februar** zu verhindern, vergleicht die Zeile nach der inneren Schleife die Monate abermals und verlässt die äußere, sobald der Folgemonat erreicht ist.

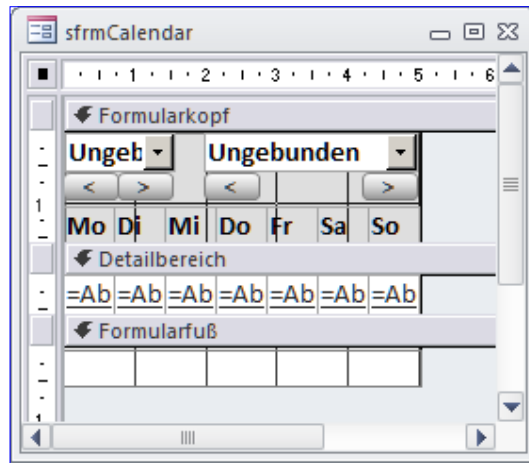


Bild 6: Entwurfsansicht des Endlosformulars **sfrmCalendar**

Damit ist das Werk vollbracht. Die **Requery**-Anweisung auf das Formular (**Me**) baut die Ansicht auf Grundlage der neuen Datensätze nun neu auf.

Entwurf des Kalenderformulars

Neben den eigentlichen Tagen des Kalenders im Detailbereich soll das Formular noch in den Spaltenköpfen die Wochentage anzeigen. Außerdem dienen zwei Kombinationsfelder im Formularkopf der Auswahl des gewünschten Jahres und Monats. Als zusätzliche Navigations-elemente können die Werte dieser Comboboxen über darunter liegende Buttons jeweils um eins vor oder zurückgeschaltet werden. Bild 6 demonstriert den Aufbau.

Die Spaltenüberschriften sind durch Labels realisiert, die hier mit festen Wochentagen versehen sind. Im Detailbereich gibt es sieben Textboxen, die anfänglich an die Datenfelder **D1** bis **D7** der Tabelle **tblCalendar** gebunden waren. Das aber

muss umgangen werden, weil sonst ja negative Tageswerte erschienen. Der Ausdruck **Abs** aber macht aus negativen Zahlen positive, lässt positive aber unverändert. Also ist der **Steuerelementinhalt** der ersten beiden Textboxen dieser:

```
= Abs([D1])
= Abs([D2])
...
```

Damit werden die Tage korrekt ausgegeben. Fehlt nur noch die graue Hinterlegung der nicht zum Monat gehörigen Tage des Kalenders über die **Bedingte Formatierung**.

Bedingte Formatierung der Tageswerte

Klicken Sie auf die erste Textbox im Detailbereich rechts und wählen im Kontextmenü den Eintrag **Bedingte Formatierung...** Das ruft einen Dialog, wie in Bild 7, auf den Plan. Hier sind zunächst eine oder mehrere Bedingungs-

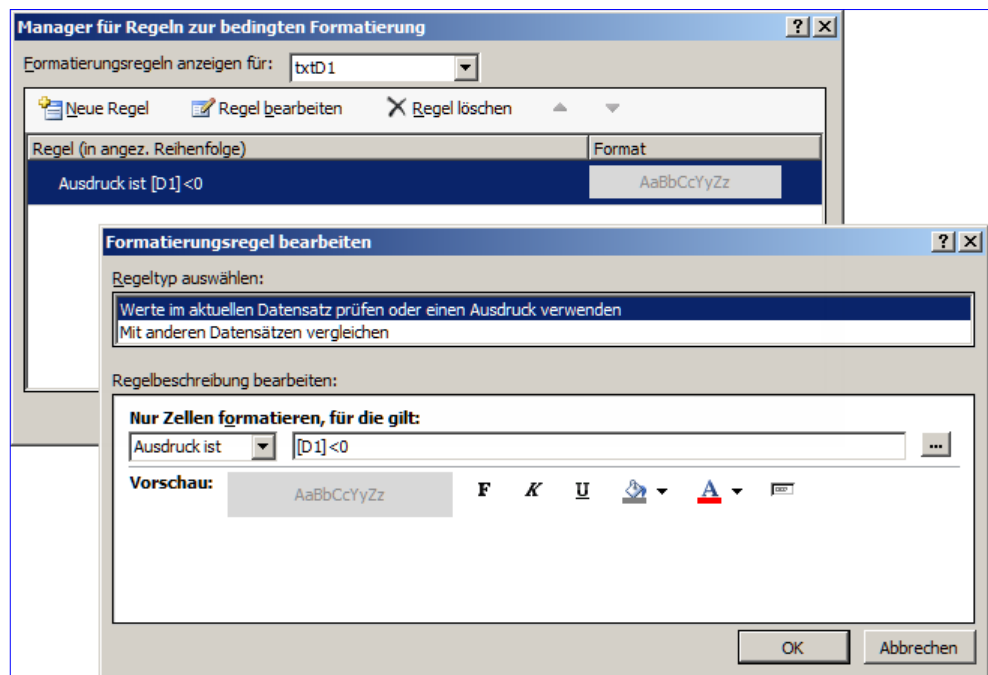


Bild 7: Dialoge zur Bedingten Formatierung der Datumtextfelder

Kalendersteuerelement, Teil 2

Das im ersten Teil dieser Ausgabe beschriebene Kalendersteuerelement eignet sich vornehmlich zur Auswahl eines Datums. Benötigen Sie aber eine Übersicht, wie die Termine des Outlook-Kalenders mit der Markierung von Datumsbereichen, etwa zur Darstellung Ihrer Urlaubsplanung, so sind die Anforderungen ganz andere. Auch für diesen Zweck stellen wir ein Pseudo-Steuerelement vor, das allein mit Access-Bordmitteln realisiert ist.

Terminkalender

Einsatzbereiche für solche kalendarischen Übersichten gibt es viele. Feiertage können darin eingetragen werden, Geburtstage und andere Jubilaren, die Buchung von Ferienwohnungen oder Autovermietungen, die Einsatzplanung von Mitarbeitern, oder Sie markieren hier Ihre Urlaubstage. Dabei geht es weniger um die Auswahl von Terminen, als um deren Darstellung. Bild 1 zeigt bereits, wie unser Beispiel aussieht.

Natürlich bauen wir hier keinen komplexen Terminkalender nach, wie den von Outlook. Das Kalendersteuerelement **sfrmCalendarMonths** ist eng an die einfache Version des Auswahlkalenders im ersten Teil dieser Ausgabe angelehnt, wie seine linke Seite schon vermuten

lässt. Es dient in erster Linie der Visualisierung bereits vorliegender Termindaten. Beschreiben wir zunächst kurz seinen Aufbau.

Grundsätzlich werden hier drei Monate mit ihren Tagen angezeigt. Die Auswahl der Monate geschieht mit dem Kombinationsfeld links, die den mittleren Monat einstellt. Die übrigen Navigationselemente, wie die Jahres-Combo oder die Buttons zum Weiterschalten der Monate gleichen in ihrer Funktion genau der des einfachen Auswahlkalenders. Zusätzlich ist die Kalenderübersicht noch mit einer Titelzeile oben versehen, die Sie mit beliebigem Inhalt füllen können. Das eigentliche Feature aber sind die rot unterlegten Zellen, die durch die dem Steuerelement über eine öffentliche Funktion zugewiesene Datumsbereiche

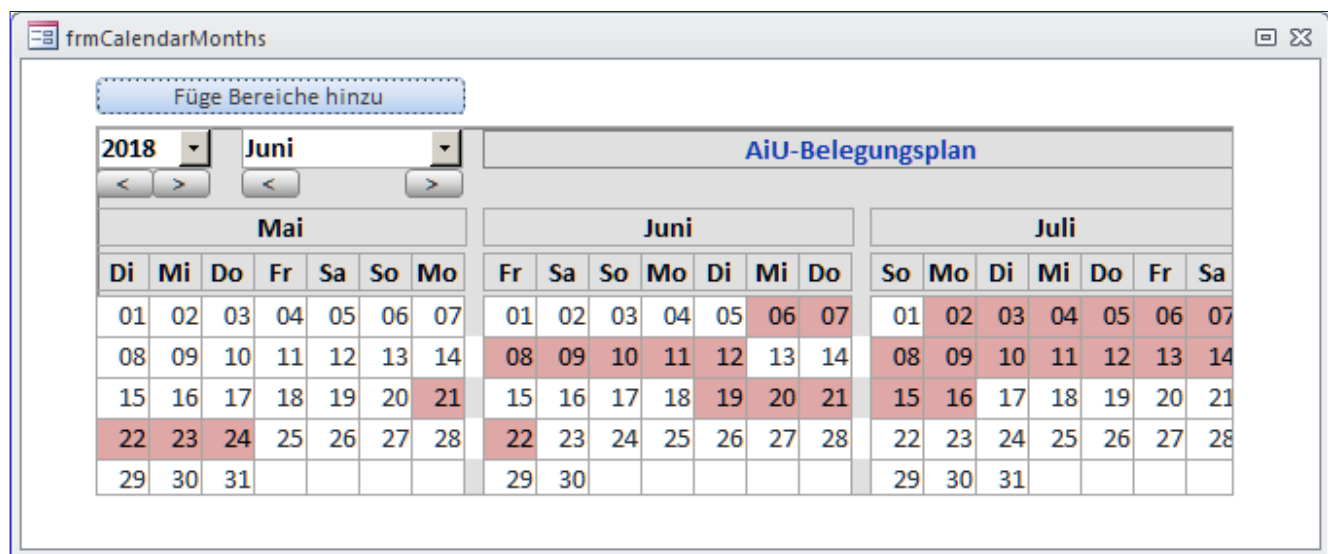


Bild 1: Demo des Unterformularsteuerelements Monatskalender nach Einbau in ein Hauptformular mit Markierungen

zustande kommen. Mehr gibt das Steuerelement in dieser Version nicht her. Es reagiert auch nicht auf Klicks in die Zellen. Das wäre eine Eigenschaft, die Sie mit dem Wissen aus dem ersten Teil dieser Ausgabe zum Auswahlkalender leicht selbst nachtragen könnten.

| Feldname | Felddatentyp | Beschreibung |
|----------|--------------|--------------|
| D11 | Zahl | |
| D21 | Zahl | |
| D31 | Zahl | |
| D41 | Zahl | |
| D51 | Zahl | |
| D61 | Zahl | |
| D71 | Zahl | |
| D12 | Zahl | |

Bild 2: Ausschnitt der Monatskalendertabelle in der Entwurfsansicht

Basistabelle

Ähnlich, wie beim Auswahlkalender, kommt für die Darstellung der Tage eine Tabelle zum Einsatz, deren Felder dann von 21 Textboxen im Endformular ausgegeben werden. Hier reichen sieben Felder natürlich nicht aus. Für jede Spalte des Dreimonatskalenders muss ein eigenes Feld her, wenn nicht etwa eine umfassende Kreuztabellenabfrage verwendet werden soll.

Die Felder der Tabelle **tblCalendarMonths** sind ebenfalls vom Zahlentyp **Long**. Die Nummerierung geschieht über das Präfix **D**, gefolgt von der Nummer des Wochentags und der Position des Monats im Kalender. **D62** bezeichnet also etwa den sechsten Tag (**Sonnabend**) und die **2** darin den mittleren Monat. Bild 2 verdeutlicht dies ausschnittsweise. Auch diese Tabelle wird vom Formular selbst mit Datensätzen gefüllt.

Allerdings speichern wir hier nicht jeweils die Tage der Monate ab, da diese ja mehrfach vorkommen –pro Monat einmal. Stattdessen nimmt ein Feld jeweils tatsächlich einen Datumswert auf. Warum ein Datum in einem **Long**-Feld? Sie ahnen es möglicherweise: Auch hier möchten wir **Bedingte Formatierung** zur farblichen Unterscheidung der Zellen benutzen und nehmen wieder negative

Werte als Indiz für eine Markierung. Schauen Sie auf die gefüllte Tabelle in Bild 3.

Ein negativer Wert führt später zur roten Hervorhebung der Zelle. Der Absolutwert einer Zahl stellt das Datum dar. Wir erwähnten bereits, dass ein Access- oder VBA-Datum tatsächlich ein **Double**-Wert ist, dessen ganzzahliger Anteil den Tag angibt. Und diesen kann auch ein **Long**-Wert aufnehmen. Beispiel:

```
? Now()                -> Datum: 21.07.2017 08:33
? CDb1(Now())          -> Double: 42937,35625
? CLng(CDb1(Now()))    -> Long: 42937
? CDate(42938)         -> Datum: 21.07.2017
```

Benötigen Sie also den Zeitanteil eines Datums nicht, so reicht zu dessen Speicherung ein **Long**-Wert.

Die Tatsache, dass hier tatsächlich Datumswerte abgespeichert sind, macht es überdies später leichter, sie beim Klick auf die Textboxen zu ermitteln. Den Steuerelementinhalt unterziehen Sie dazu lediglich der Funktion **CDate()**. Eine Berechnung aus der Zellenposition, wie beim Aus-

| D11 | D21 | D31 | D41 | D51 | D61 | D71 | D12 | D22 | D32 | D42 | D52 | D62 | D72 | D13 | D23 | D33 | D43 | D53 | D63 | D73 |
|--------|--------|--------|-------|-------|-------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 43221 | 43222 | 43223 | 43224 | 43225 | 43226 | 43227 | 43252 | 43253 | 43254 | 43255 | 43256 | -43257 | -43258 | 43282 | -43283 | -43284 | -43285 | -43286 | -43287 | -43288 |
| 43228 | 43229 | 43230 | 43231 | 43232 | 43233 | 43234 | -43259 | -43260 | -43261 | -43262 | -43263 | 43264 | 43265 | -43289 | -43290 | -43291 | -43292 | -43293 | -43294 | -43295 |
| 43235 | 43236 | 43237 | 43238 | 43239 | 43240 | -43241 | 43266 | 43267 | 43268 | 43269 | -43270 | -43271 | -43272 | -43296 | -43297 | 43298 | 43299 | 43300 | 43301 | 43302 |
| -43242 | -43243 | -43244 | 43245 | 43246 | 43247 | 43248 | -43273 | 43274 | 43275 | 43276 | 43277 | 43278 | 43279 | 43303 | 43304 | 43305 | 43306 | 43307 | 43308 | 43309 |
| 43249 | 43250 | 43251 | | | | | 43280 | 43281 | | | | | | 43310 | 43311 | 43312 | | | | |

Bild 3: So präsentiert sich die Tabelle **tblCalendarMonths**, nachdem sie über das Formular mit Daten gefüllt wurde.

wahlkalender, ist hier nicht nötig. Schauen wir im Folgenden an, wie die Daten der Tabelle generiert werden. Auch hier nennt sich die verantwortliche Prozedur **FillCalendar**.

Daten der Basistabelle erzeugen

Die Prozedur **FillCalendar** wird im Formularmodul immer dann aufgerufen, wenn sich der Monat oder Tag ändern. Das kann durch Auswahl in den entsprechenden Kombinationsfeldern geschehen, durch Betätigung der Buttons zum Weiterschalten, oder durch Zuweisung an die Eigenschaftsprozeduren **Year** und **Month**. Listing 1 zeigt wieder eine gekürzte Version, in der nur die für die Erzeugung der Datensätze relevanten Teile abgebildet sind.

Zunächst wird in **StartDate** der erste Tag des linken Monats ermittelt. Dazu wird vom Monat der **Member-Variablen m_Month eins** abgezogen und das Datum mit **DateSerial** berechnet. **EndDate** wiederum speichert den letzten Tag des rechten Monats. Da die Anzahl der Tage eines Monats variabel ist, kann für den **Tag**-Parameter in **DateSerial** kein Wert angegeben werden. Man behilft sich in diesem Fall mit dem Folgemonat (**m_Month + 2**) und dem Tag **0**. Das entspricht dem ersten Tag des Monats minus eins.

Die **Execute**-Anweisung leert die Tabelle und **OpenRecordset** öffnet eine beschreibbare Datensatzgruppe auf

sie. Die folgende Schleifenkonstruktion weicht von der des Auswahlkalenders ab. Wir benötigen hier drei ineinander verschachtelte Schleifen. Die äußere auf die Zählervariable **n** stellt die Zeilen der Kalender dar. Diese entspricht dann auch der Anzahl der Datensätze der Tabelle, weshalb der Durchlauf auch mit einem **AddNew** beginnt. Die nächste Schleife mit dem Zähler **i** betrifft die drei Monate, die innerste Schleife mit dem Zähler **j** deren Wochentage.

Zur Berechnung des Datums eines Datenfeldes wird hier **StartDate** nicht einfach fortlaufend erhöht. Stattdessen werden zu **StartDate** zweimal Werte über die **DateAdd**-Funktion addiert und das Ergebnis in der **Date**-Variablen **DTmp**

```
Private Sub FillCalendar()  
    Dim rs As DAO.Recordset  
    Dim i As Long, j As Long, n As Long, f As Long  
    Dim StartDate As Date, EndDate As Date  
    Dim MaxDate As Date, DTmp As Date  
  
    StartDate = DateSerial(m_Year, m_Month - 1, 1)  
    EndDate = DateSerial(m_Year, m_Month + 2, 0)  
  
    CurrentDb.Execute "DELETE FROM tblCalendarMonths"  
    Set rs = CurrentDb.OpenRecordset("tblCalendarMonths", dbOpenDynaset)  
    For n = 0 To 4  
        rs.AddNew  
        For i = 1 To 3  
            MaxDate = DateSerial(m_Year, m_Month + i - 1, 0)  
            For j = 1 To 7  
                DTmp = DateAdd("m", i - 1, StartDate)  
                DTmp = DateAdd("d", n * 7 + j - 1, DTmp)  
                If DTmp > MaxDate Then Exit For  
                If IsInRanges(DTmp) Then f = -1 Else f = 1  
                rs.Fields("D" & CStr(j) & CStr(i)).Value = f * CLng(DTmp)  
            Next j  
        Next i  
        rs.Update  
    Next n  
  
    rs.Close  
    Me.Requery  
End Sub
```

Listing 1: Befüllen der Tabelle **tblCalendarMonths**

zwischen gespeichert. Die erste **DateAdd**-Funktion addiert die Nummer des Monats aus dem Monatsschleifenzähler **i**, wobei der Ausdruck **m** der Funktion erst sagt, dass Monate zu addieren sind. Das zweite **DateAdd** verwendet Tag-Werte (Ausdruck **d**) und berechnet über den Wochentag in **j** und das Siebenfache der betreffenden Woche **n** einen weiteren Offset. Im Prinzip könnte dieser Wert aus **DTmp** dann schon einem Tabellenfeld zugewiesen werden.

Da jedoch die Schleifendurchläufe dazu führen können, dass über die Addition der Tage ein über einen Monat hinausreichendes Datum ermittelt würde, gibt es eine Abbruchbedingung für die innere Schleife. Ist **DTmp** größer, als der letzte Tag des Monats, so wird die Schleife verlassen. Dieser letzte Tag ist in der Variablen **MaxDate** gespeichert und wird nach dem Beginn der zweiten Schleife jeweils neu berechnet.

Schließlich sind noch die Zeiträume zu berücksichtigen, die im Kalendersteuerelement rot zu unterlegen sind und durch negative Datumswerte repräsentiert werden. Die Funktion **IsInRanges**, auf welche wir noch zu sprechen kommen, wird dazu befragt.

Befindet sich das Schleifendatum **DTmp** innerhalb eines der verabreichten Zeiträume, so gibt die Funktion **True** zurück. In diesem Fall nimmt die Variable **f** den Wert **-1** an, andernfalls **1**. Und **f** ist dann wieder der Multiplikator für den Datumswert, der dem Feld des Recordsets zugewiesen wird. Der Name des Felds ergibt sich aus dem Präfix **D** und den Zählvariablen **j** und **i** über String-Verkettung.

Zeiträume zuweisen

Dem Kalendersteuerelement können beliebig viele Datumsbereiche hinzugefügt werden, die dann in der Ansicht rot markiert werden. Sie können dabei auch außerhalb des dargestellten Bereichs liegen. Die Prozedur **AddRange** (s. Listing 2) bewerkstelligt deren Speicherung.

```
Private Type TRange
    StartDate As Date
    EndDate As Date
End Type

Private arrRanges() As TRange

Public Sub AddRange(StartDate As Date, EndDate As Date)
    Dim n As Long

    On Error Resume Next
    n = UBound(arrRanges)
    If Err.Number <> 0 Then
        n = 0
    Else
        n = n + 1
    End If
    On Error GoTo 0
    ReDim Preserve arrRanges(n)
    arrRanges(n).StartDate = StartDate
    arrRanges(n).EndDate = EndDate

    FillCalendar
End Sub
```

Listing 2: Hinzufügen eines Zeitraums über **AddRange**

Als Parameter geben Sie das Startdatum des gewünschten Bereichs in **StartDate** an und das Enddatum in **EndDate**. Soll es nur ein Tag sein, dann müssen beide Werte identisch sein. Das Wertepaar speichert die Routine in einem modulweit gültigen **Array arrRanges** des benutzerdefinierten Typs **TRange**, der im Kopf des Moduls deklariert ist. Hier muss erst per **UBound** ermittelt werden, wie viele Elemente das Array bereits enthält. Ist noch kein Element vorhanden, so ist das Array noch nicht initialisiert, was bei **UBound** zu einem Fehler führen würde. Deshalb ist die Fehlerbehandlung eingangs per **On Error Resume Next** außer Kraft gesetzt. Die Variable **n** nimmt die Anzahl der Elemente entgegen und erhöht sie um **eins**.

Nun kann das Array mit **ReDim** neu dimensioniert werden, wobei das Schlüsselwort **Preserve** angibt, dass sein Inhalt dabei nicht verloren gehen soll. Im **n-ten** Element das Arrays werden schließlich Start- und Enddatum abgespei-

chert. Der abschließende Aufruf von **FillCalendar** führt dazu, dass der Zeitraum im Kalender auch sofort dargestellt wird.

Möchten Sie die Zeiträume modifizieren, so löschen Sie sie mit der Prozedur **DeleteRanges** alle auf einen Schlag und fügen die neuen wieder hinzu. Das Löschen einzelner Datumsbereiche erlaubt das Modul nicht.

```
Public Sub DeleteRanges()
    Erase arrRanges
    FillCalendar
End Sub
```

Die angesprochene Funktion **IsInRanges** ermittelt dann, ob ein bestimmtes Datum sich innerhalb der Zeiträume des angelegten Arrays befindet (s. Listing 3). **D** ist hier der Datumsparameter, den Sie der Funktion übergeben. Alle Zeiträume werden in einer Schleife durchlaufen und **StartDate**, wie **EndDate** eines Elements, mit ihm vergli-

```
Private Function IsInRanges(D As Date) As Boolean
    Dim i As Long, n As Long

    On Error Resume Next
    n = UBound(arrRanges)
    If Err.Number <> 0 Then
        On Error GoTo 0
        Exit Function
    End If
    On Error GoTo 0

    For i = 0 To n
        If (D >= arrRanges(i).StartDate) And (D <= arrRanges(i).EndDate) Then
            IsInRanges = True
            Exit For
        End If
    Next i
End Function
```

Listing 3: Die Funktion ermittelt, ob sich das Datum **D** innerhalb der gespeicherten Datumsbereiche befindet.

chen. Sobald das für ein Bereichselement zutrifft, erhält der Rückgabewert der Funktion **True** und die Schleife wird verlassen.

Entwurf des Steuerelementformulars

Die Entwurfsansicht des Formulars **sfrmCalendarMonths** (s. Bild 4) kommt ähnlich daher, wie die des Auswahlkalenders. Der Unterschied ist, dass hier natürlich mehr Textboxen im Detailbereich untergebracht sind, drei Labels

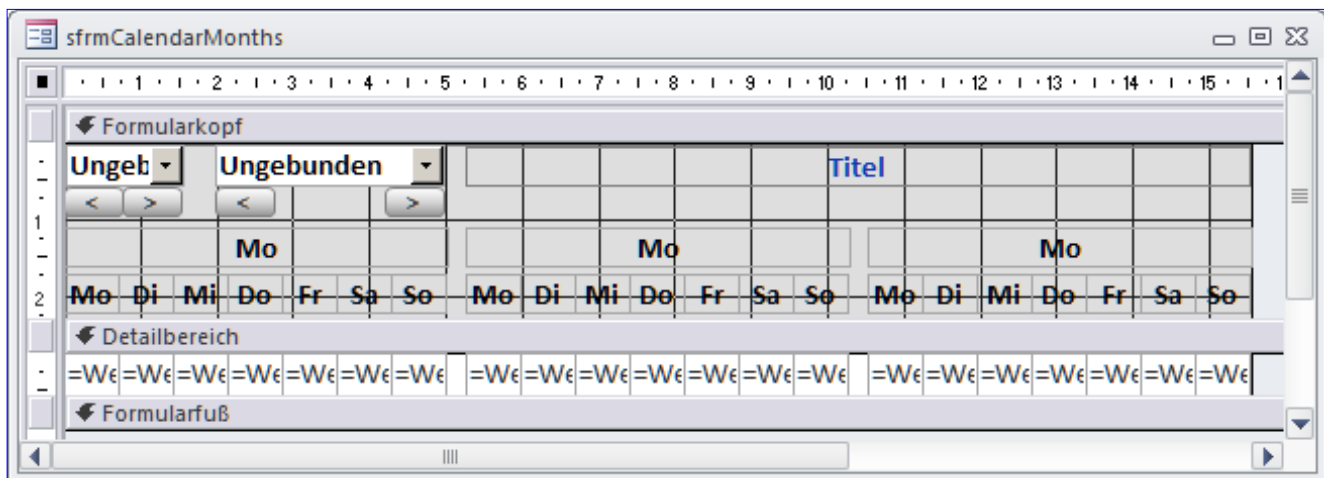


Bild 4: Das Formular **sfrmCalendarMonths** gleich in der Entwurfsansicht dem des Beitrags zum einfachen Kalender.

Setup für ein VSTO-AddIn

Im Beitrag "Effizienz-Hacks: Mailabruf erschweren" haben wir gezeigt, wie Sie mit Visual Studio 2015 Community Edition ein VSTO-Add-In erstellen, das in Outlook integriert wird. Diesmal wollen wir dem Projekt ein Setup hinzufügen, mit dem Sie das Add-In auch an andere Rechner weitergeben können.

Voraussetzungen

Voraussetzung ist ein installiertes Visual Studio 2015 in der Community Edition sowie die aktuellsten Erweiterungen, die auch den für das Hinzufügen eines Setup-Projekts benötigten Erweiterungen mit sich bringt. Ausgangspunkt für dieses Projekt ist das im Beitrag **Effizienz-Hacks: Mailabruf erschweren** (www.access-im-unternehmen.de/****) erstellte Outlook-VSTO-Projekt.

Setup-Projekt hinzufügen

Bevor wir das Setup-Projekt hinzufügen, eine kurze Erläuterung des geplanten Aufbaus: Im oben genannten Beitrag haben wir nur eine einfache Lösung programmiert, die aus einem einzelnen Projekt besteht.

Auch ein einzelnes Projekt befindet sich jedoch in einer Projektmappe. Hier wird der Begriff der Projektmappe nun wichtig, denn wir fügen dieser nun ein zweites Projekt hinzu – eben das Setup-Projekt. Deshalb verwenden Sie auch nicht einfach den Befehl **DateiNeulProjekt...**, wie Sie es sonst

tun, sondern markieren den Eintrag der Projektmappe im Projektmappen-Explorer, klicken mit der rechten Maustaste darauf und wählen aus dem Kontextmenü den Eintrag **HinzufügenNeues Projekt...** aus (s. Bild 1).

Daraufhin erscheint der Dialog **Neues Projekt** hinzufügen. Hier navigieren Sie zum Bereich **Andere ProjekttypenVisual Studio Installer** (wenn Sie diesen nicht finden, prüfen Sie, ob Sie die aktuellsten Visual Studio-Erweiterungen installiert haben) und wählen den Eintrag **Setup Project** aus. Geben Sie als Namen den Wert **OutlookAddIn_Setup** ein (s. Bild 2).

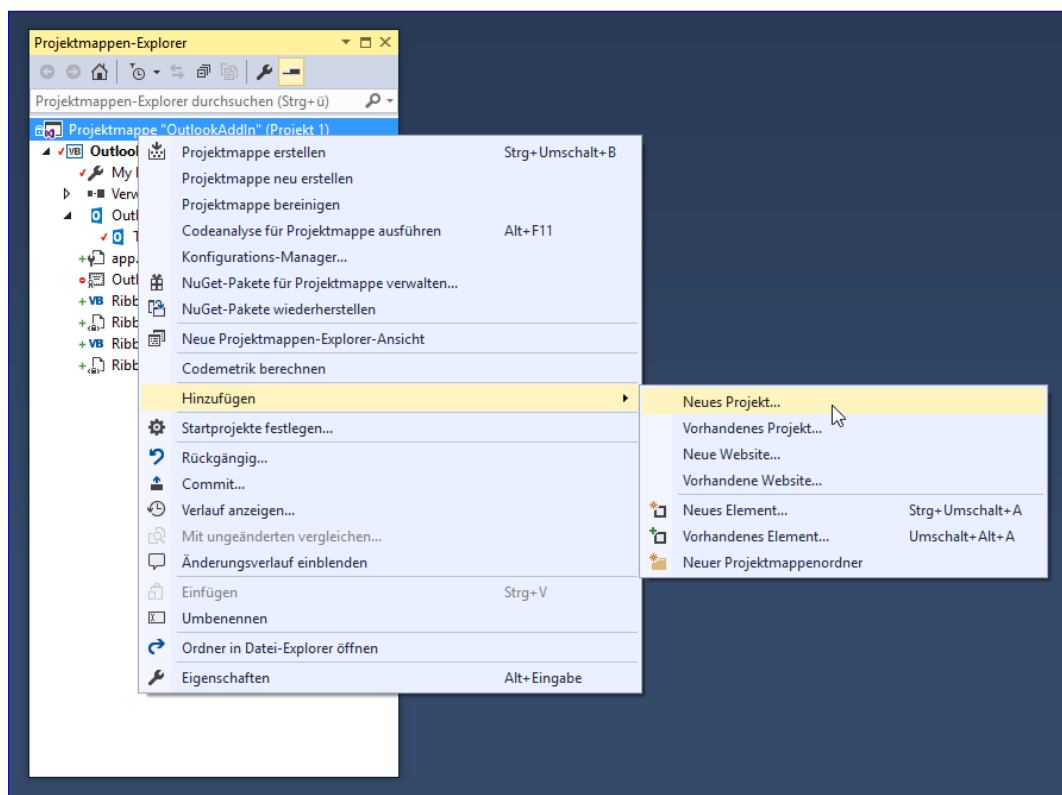


Bild 1: Hinzufügen eines neuen Projekts zu einer Projektmappe

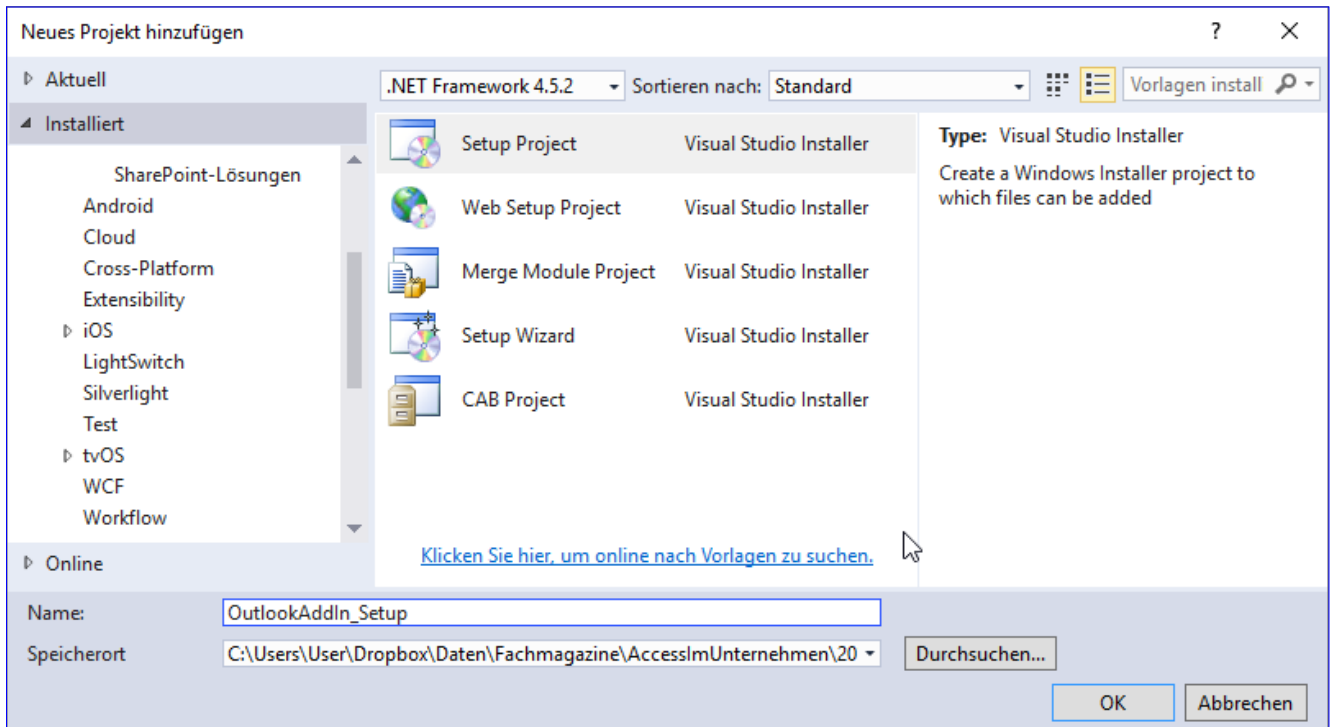


Bild 2: Auswählen des Projekttyps

Projektausgabe hinzufügen

Nach einem Klick auf **OK** wird das neue Projekt als Unterprojekt der Projektmappe angelegt.

Außerdem erscheint im mittleren Bereich von Visual Studio das Element **File System (OutlookAddIn_Setup)**.

Klicken Sie hier mit der rechten Maustaste auf den Eintrag **Application Folder** und wählen Sie aus dem Kontextmenü den Befehl **AddProjektAusgabe...** aus (s. Bild 3).

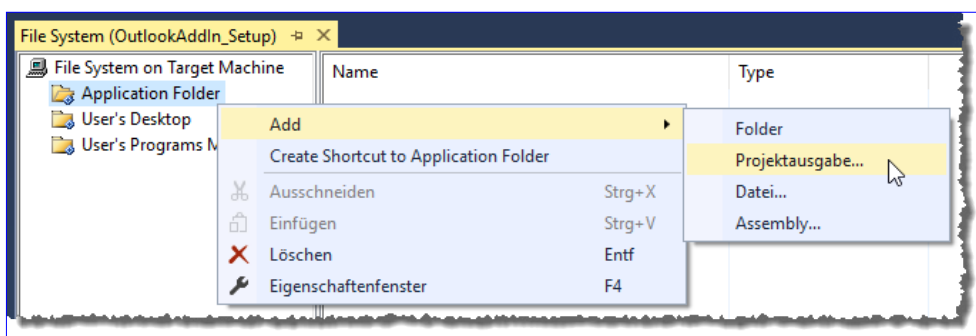


Bild 3: Hinzufügen der Projektausgabe

Nun erscheint der Dialog **Projektausgabegruppe hinzufügen**, wo Sie den Eintrag **Primäre Ausgabe** hinzufügen und auf **OK** klicken (s. Bild 4). Damit fügen Sie dem Setup-Projekt im Projektmappen-Explorer einen Eintrag **Primäre Ausgabe from Outlook_AddIn** hinzu. Außerdem werden einige Einträge unter **Detected Dependencies** angezeigt, die beim Setup berücksichtigt werden sollen.

Weitere Dateien hinzufügen

Damit legen Sie nun fest, welche Dateien alle im Setup landen. Die Option **Primäre Ausgabe** fügt jedoch nur die im Standardfall benötigten Dateien hinzu.

Wenn Sie ein Add-In weitergeben wollen, benötigen Sie einige weitere Dateien. Welche Dateien Sie überhaupt benötigen, können Sie abgleichen, wenn

Sie sich ansehen, welche Dateien beim Debuggen des Add-In-Projekts in den Ordner `.../bin/debug` des Projekts geschrieben werden (s. Bild 5).

also noch die fehlenden Dateien hinzufügen – in diesem Fall also die folgenden Dateien:

Im Setup landen jedoch nur die Dateien, die im Bereich **FileSystem** von **OutlookAddIn_Setup** aufgeführt werden.

Welche Dateien unter **Primäre Ausgabe ...** zu verstehen sind, erfahren Sie, wenn Sie den Kontextmenü-Eintrag **Ausgaben von Primäre Ausgaben from OutlookAddIn (Active)** auswählen – in diesem Fall die `.dll` plus eine `.config`-Datei (s. Bild 6).

Vor dem ersten Erstellen und Testen des Setups sollten wir

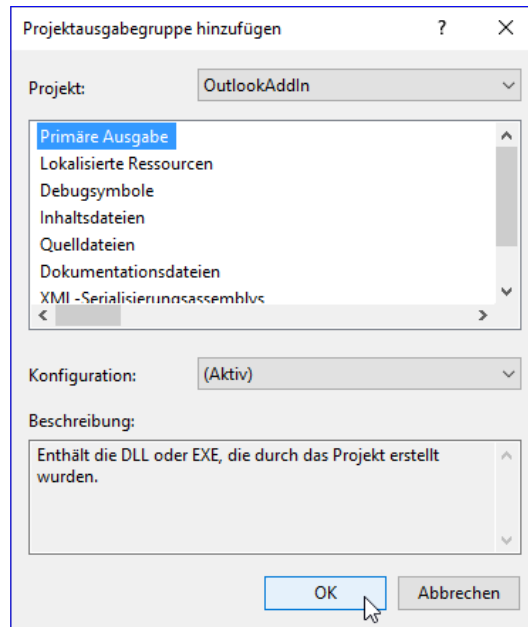


Bild 4: Auswahl der primären Ausgabe

- OutlookAddIn.xml
- OutlookAddIn.vsto
- OutlookAddIn.pdb
- OutlookAddIn.dll.manifest

Diese vier Dateien ziehen Sie einfach aus dem Ausgabeverzeichnis zum Debuggen in die Liste der in das Setup einzubindenden Dateien.

Zielordner festlegen

Nun ist es an der Zeit, festzulegen, in welchem Ordner auf dem Zielsystem die Dateien für das Add-In landen sollen. Dazu wechseln Sie wieder zum Projektmappen-Explorer und klicken mit der rechten Maustaste auf das Element **OutlookAddIn_Setup**.

Hier wählen Sie den Befehl **ViewDateisystem** aus. Dies zeigt keinen neuen Dialog oder Bereich an, sondern ändert lediglich den Inhalt des **Eigenschaften**-Bereichs auf **Application Folder File Installation Properties**.

Hier geben Sie unter **DefaultLocation** den folgenden Wert ein (s. Bild 7):

```
[LocalAppDataFolder][Manufacturer]\[ProductName]
```

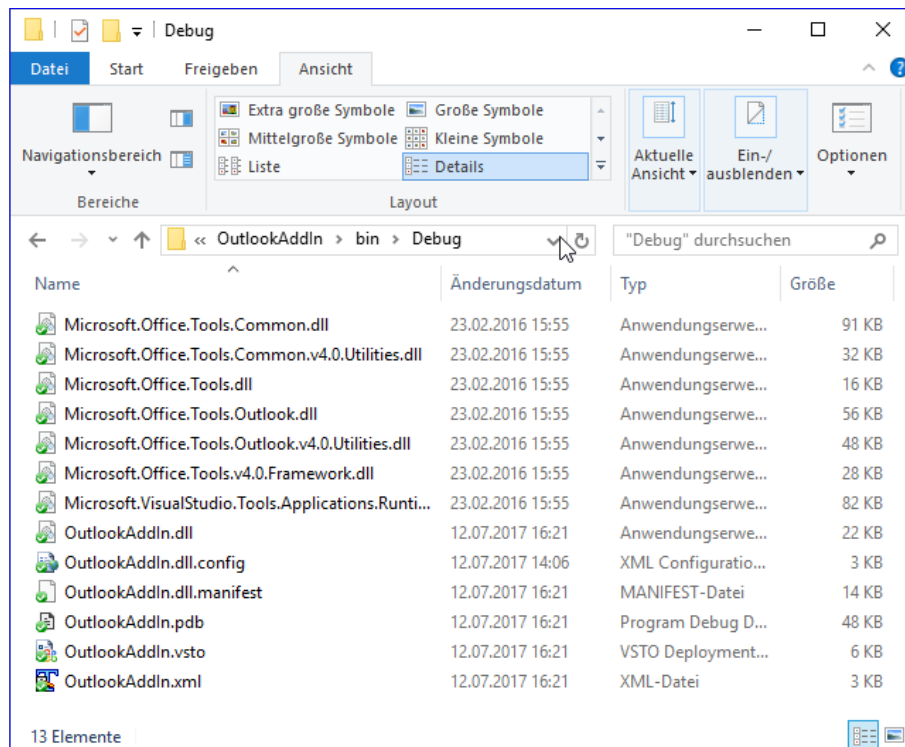


Bild 5: Dateien, die beim Debuggen im Zielordner landen

Belegungsplan mit Kalender

Bei der Suche nach sinnvollen Anwendungszwecken für die Kalendersteuerelemente dieser Ausgabe fiel die Entscheidung für ein Beispiel schwer. Wir demonstrieren deren Einsatz nun an einem fingierten Belegungsplan für Ferienwohnungen, der zudem das Übersichtskalenderelement noch um das eine oder andere Feature erweitert.

Wohnobjekte und Vermietungszeiträume

Buchen Sie im Web eine Ferienwohnung, so werden Sie häufig mit Übersichtskalendern konfrontiert, die die noch freien Zeiträume eines Objekts abbilden. Das ist im Prinzip ein idealer Anwendungsfall für das dreimonatige Kalendersteuerelement dieser Ausgabe. Der Teufel liegt allerdings, wie wir noch sehen werden, im Detail. Ohne Modifikation des Steuerelements treffen Sie auf einige Probleme, die es im Folgenden zu lösen gilt.

Die Buchung des Objekts ist das eine, die Übersicht für den Vermieter das andere. Bild 1 zeigt zunächst die Version für den Vermieter. Seine Buchungsdatenbank enthält alle Daten zu seinen Objekten, den Kunden und den gebuchten Zeiträumen. Die bekommt er nun im Formular **frmBelegung** präsentiert.

Im Kombinationsfeld oben kann er ein Objekt auswählen, dessen Vermietungszeiträume sich danach im Kalender unten abbilden. Mit den Navigationselementen kann er sich dabei durch beliebige Jahre und Monate bewegen. Die Hintergrundfarben symbolisieren verschiedenen Belegungsarten.

Grüne Felder bedeuten, dass diese Tage für das Objekt noch nicht belegt sind, rote, dass diese Tage vermietet sind. Möglicherweise würde das bereits ausreichen, doch es gibt auch noch anders geartete

Termine. Zur Reinigung oder Wartung der Objekte etwa fallen unter Umständen zusätzliche Tage an. Eventuell nutzt der Vermieter das Objekt auch für den Eigenbedarf. Auch sonst kann das Objekt aus bestimmten Gründen für die Vermietung gesperrt sein. All diese Typen von Belegung sollen sich im Kalender durch eine entsprechende Farbgebung unterscheiden lassen.

Datenmodell

Im Prinzip gibt es drei Einheiten, die es zu verknüpfen gilt. Einmal sind da die Wohnungen, die mindestens durch Bezeichnungen und Adressen repräsentiert werden.

Zum anderen gibt es Kunden, die durch Namen und Adresdaten festgelegt sind. Beide kommen über die Belegungszeiträume zueinander. Eine Belegung benötigt daher einen Verweis auf einen Kunden, einen auf eine Wohnung und dann noch den Mietzeitraum, welcher durch das Datum des Beginns und das des Endes der Buchung bestimmt ist. Somit kommt es zu drei Tabellen, die sich

| Dezember | | | | | | | Januar | | | | | | | Februar | | | | | | |
|----------|----|----|----|----|----|----|--------|----|----|----|----|----|----|---------|----|----|----|----|----|----|
| Do | Fr | Sa | So | Mo | Di | Mi | So | Mo | Di | Mi | Do | Fr | Sa | Mi | Do | Fr | Sa | So | Mo | Di |
| 01 | 02 | 03 | 04 | 05 | 06 | 07 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
| 08 | 09 | 10 | 11 | 12 | 13 | 14 | 08 | 09 | 10 | 11 | 12 | 13 | 14 | 08 | 09 | 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 15 | 16 | 17 | 18 | 19 | 20 | 21 |
| 22 | 23 | 24 | 25 | 26 | 27 | 28 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
| 29 | 30 | 31 | | | | | 29 | 30 | 31 | | | | | | | | | | | |

Bild 1: Der Belegungsplan für Ferienwohnungen enthält das dreimonatige Kalendersteuerelement mit mehrfachen Bedingungen Formatierungen zur Hervorhebung der Zeiträume

in der Beispieldatenbank **tblKunden**, **tblWohnungen** und **tblBelegung** nennen. Sie sind miteinander über indizierte Schlüsselfelder verknüpft, wie in Bild 2.

Diese drei Haupttabellen weisen zusätzlich je ein Feld **Vermerk** auf, in das optional zu jedem Datensatz eine beliebige Notiz oder Information eingetragen werden kann. Das Feld **IDTypBeleg** in **tblBelegung** verweist außerdem auf eine Nachschlagetabelle **tblBelegungstyp**, in der die Arten der Belegung festgehalten sind.

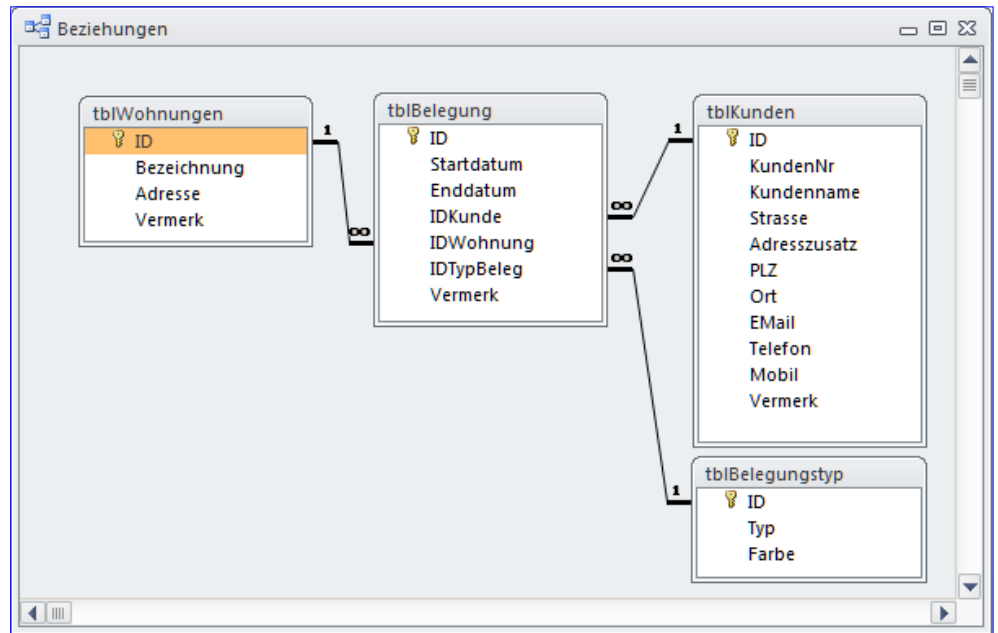


Bild 2: Das grundlegende Datenmodell der Beispieldatenbank zum Belegungsplan

Beginnen wir mit der Tabelle für die Wohnungen (s. Bild 3). Außer der **ID** vom **Long**-Typ **Autowert** enthält sie nur die

Bezeichnung des Objekts und die **Adresse** in Form eines **Memo**-Felds.

| Feldname | Felddatentyp | Beschreibung |
|-------------|--------------|--------------|
| ID | AutoWert | |
| Bezeichnung | Text | |
| Adresse | Memo | |
| Vermerk | Memo | |

Bild 3: Die Tabelle **tblWohnungen** im Entwurf

Normalerweise würde man die Adresse in mehrere Felder aufteilen, für unseren Zweck lassen wir es aber bei dieser einfachen Lösung. Die Kundentabelle (s. Bild 4) kennen Sie so oder ähnlich schon aus anderen Zusammenhängen. Sie weist hier keinerlei Besonderheiten auf.

| Feldname | Felddatentyp | Beschreibung |
|--------------|--------------|--------------|
| ID | AutoWert | |
| KundenNr | Text | |
| Kundenname | Text | |
| Strasse | Text | |
| Adresszusatz | Text | |
| PLZ | Text | |
| Ort | Text | |
| EMail | Text | |
| Telefon | Text | |
| Mobil | Text | |
| Vermerk | Memo | |

Bild 4: Die Tabelle **tblKunden** in der Entwurfsansicht

In der Belegungstabelle (s. Bild 5) gibt es zunächst **Start-** und **Enddatum** eines Mietzeitraums. Dann verweist das Feld **IDKunde** auf die **ID** eines Kundendatensatzes. Genauso verweist wiederum **IDWohnung** auf die **ID** eines Wohnungsdatensatzes. Vervollständigt wird sie mit dem Feld **IDTypBeleg**, das eine Zahl zur Art der Belegung abspeichert, welche aus der Tabelle **tblBelegungstyp** stammt.

Alle Beziehungen sind mit **Referenzieller Integrität** ausgestattet, was bedeutet, dass die

Tabelle **tblBelegung** keine Datensätze annimmt, bei denen die Verweis-IDs in den Fremdtabellen nicht existieren. Außerdem ist für die Beziehungen **Aktualisierungs- und Löschweitergabe** eingestellt.

Damit führt das Löschen eines Kunden oder einer Wohnung automatisch auch zum Löschen aller beteiligten Belegungen, die Verweise auf jene besitzen. Das macht übrigens erforderlich, dass die Felder **IDKunde** und **IDWohnung** indiziert sind und die **ID**-Felder der Haupttabellen jeweils den Primärschlüssel stellen.

Die möglichen Belegungsarten sind in der Tabelle **tblBelegungstyp** (s. Bild 6) untergebracht. Der Standard wäre der Datensatz mit dem **ID-Wert 2**, was **Vermietet** bedeutet. Vielleicht fallen Ihnen noch weitere Typen ein.

Wir sind auch keine Wohnungsmakler und wissen nicht, ob die Einträge **Gesperrt** oder **Unbestimmt** einen tieferen Sinn besitzen. Das Feld **Farbe** (Datentyp **Text**) nimmt weiter in hexadezimaler Form einen **RGB**-Farbwert an, der später für den Hintergrund der Kalenderzellen über

| Feldname | Felddatentyp |
|------------|---------------|
| ID | AutoWert |
| Startdatum | Datum/Uhrzeit |
| Enddatum | Datum/Uhrzeit |
| IDKunde | Zahl |
| IDWohnung | Zahl |
| IDTypBeleg | Zahl |
| Vermerk | Memo |

Bild 5: Tabelle **tblBelegung** im Entwurf

| ID | Typ | Farbe |
|----|-----------------------|--------|
| 1 | Gesperrt | FFA0A0 |
| 2 | Vermietung | A0A0FF |
| 3 | Reinigung und Wartung | FFE0A0 |
| 4 | Eigenbedarf | FFA0FF |
| 5 | (Unbestimmt) | A0A0A0 |

Bild 6: Die möglichen Belegungsarten der Tabelle **tblBelegungstyp** in der Datenblattansicht

Bedingte Formatierung erhalten soll. Das Feld findet nur interne Verwendung. Sein Inhalt kann von Ihnen nach Belieben geändert werden.

Sehen Sie sich einige Datensätze der Tabelle **tblBelegung** in Bild 7 an. Die Felder **IDKunde**, **IDWohnung** und **IDTypBeleg** enthalten eigentlich verweisende **Long**-Werte, die aber in der Datenblattansicht nicht erscheinen, weil diese Felder als **Nachschlagfelder** mit Kombinationsfeldern als Steuerelement ausgeführt sind.

Exemplarisch zeigt Bild 8 das für die Spalte **IDKunde**. Der **Herkunftstyp** ist hier auf Ta-

belle/Abfrage eingestellt und als Quelle dient die Tabelle **tblKunden**, aus der die erste Spalte (**ID**) angebunden werden soll. Anzeigen aber soll das Feld den Namen des Kunden, welcher sich im dritten Feld befindet. Darum ist **Spaltenanzahl** auch auf **3** gesetzt.

Da auch wirklich nur der Name auftauchen soll, müssen zusätzlich die ersten beiden **Spaltenbreiten** des Kombinationsfelds auf **0 cm** eingestellt sein. Um möglichst viele Kunden bei der **DropDown**-Auswahl berücksichtigen zu

| ID | Startdatum | Enddatum | IDKunde | IDWohnung | IDTypBeleg | Vermerk |
|-----|------------|------------|-----------------------|----------------------------|--------------|---------|
| 820 | 02.01.2016 | 07.01.2016 | Access im Unternehmen | Pension Löwenwirt Zi2 | Eigenbedarf | |
| 872 | 06.01.2016 | 20.01.2016 | Steinert, Adolf | Pension Löwenwirt Zi1 | Vermietung | |
| 793 | 21.01.2016 | 27.01.2016 | Access im Unternehmen | Pension Zum Hirschen | (Unbestimmt) | |
| 899 | 22.01.2016 | 05.02.2016 | Steinert, Adolf | Stadtzimmer Panketal | Vermietung | |
| 853 | 20.02.2016 | 06.03.2016 | Bläsing, Hans Peter | Ferienwohnung Grundlsee | Vermietung | |
| 885 | 20.02.2016 | 27.02.2016 | Büttner, Sabine | Pension Zum Hirschen | Vermietung | |
| 873 | 26.02.2016 | 01.03.2016 | Schmidt, Elisabeth | Pension Löwenwirt Zi1 | Vermietung | |
| 821 | 08.03.2016 | 11.03.2016 | Access im Unternehmen | Pension Löwenwirt Zi2 | Eigenbedarf | |
| 807 | 09.03.2016 | 16.03.2016 | Access im Unternehmen | Stadtzimmer 1 Leopoldplatz | (Unbestimmt) | |

Bild 7: Die Tabelle **tblBelegung** enthält die drei Spalten **IDKunde**, **IDWohnung** und **IDTypBeleg** als Nachschlagfelder

können, steht die **Zeilenanzahl**, abweichend vom Access-Standard **16**, auf **30**.

Damit ist das eigentliche Datenmodell der Datenbank hinreichend erläutert. Für unseren speziellen Anwendungsfall ist nun aber noch die für das Kalendersteuerelement verantwortliche Tabelle **tblCalendarMonths** zu verändern. Zur Erinnerung: Die enthält 21 Felder vom Typ Zahl (**Long**), in denen Datumswerte abgespeichert werden.

Negative Werte bestimmten dabei jene Bereiche des Kalenders, die farblich hinterlegt werden sollen.

Nun haben wir es bei unserem Belegungsplan aber nicht nur mit einer möglichen Farbe zu tun, sondern, je nach Art der Belegung, mit mehreren.

Folglich muss ein Datenfeld dieser Tabelle mehr Informationen hergeben, als nur das Datum und ein Vorzeichen.

| Allgemein | Nachschlagen |
|-------------------------|------------------|
| Steuerelement anzeigen | Kombinationsfeld |
| Herkunftstyp | Tabelle/Abfrage |
| Datensatzherkunft | tblKunden |
| Gebundene Spalte | 1 |
| Spaltenanzahl | 3 |
| Spaltenüberschriften | Nein |
| Spaltenbreiten | 0cm;0cm |
| Zeilenanzahl | 30 |
| Listenbreite | Automatisch |
| Nur Listeneinträge | Ja |
| Mehrere Werte zulassen | Nein |
| Wertlistenbearbeitung z | Nein |
| Bearbeitungsformular fü | |
| Nur Datensatzherkunft s | Nein |

Bild 8: Das Feld **IDKunde** in **tblBelegung** ist ein Nachschlagefeldlegungstyp in der Datenblattansicht

| D11 | D21 | D31 | D41 | D51 | D61 | D71 | D12 | D22 |
|---------|---------|---------|---------|---------|---------|---------|---------|-------|
| 42826_2 | 42827_2 | 42828_2 | 42829_2 | 42830_2 | 42831_2 | 42832_2 | 42856_0 | 42857 |
| 42833_0 | 42834_0 | 42835_3 | 42836_3 | 42837_3 | 42838_3 | 42839_3 | 42863_0 | 42864 |
| 42840_3 | 42841_0 | 42842_0 | 42843_0 | 42844_0 | 42845_0 | 42846_0 | 42870_0 | 42871 |
| 42847_0 | 42848_0 | 42849_0 | 42850_0 | 42851_0 | 42852_0 | 42853_0 | 42877_0 | 42878 |
| 42854_0 | 42855_0 | | | | | | 42884_0 | 42885 |
| * | | | | | | | | |

Bild 9: Die Tabelle **tblCalendarMonths** kombiniert in ihren Feldern das Datum und den Belegtyp eines Kalendereintrags in einer Textform

| ID | Bezeichnung | Adresse | Vermerk |
|----|----------------------------|------------------------------------|---------------------------|
| 1 | Ferienwohnung Alpenblick | Am Alatsee 1, 87629 Füssen | 3-4 Personen |
| 3 | Ferienwohnung Grundlsee | Bräuhof 7, 8993 Grundlsee, AT | 4-6 Personen |
| 4 | Ferienwohnung Rössl | Lohbauerstr. 5, 88316 Isny | 3 Personen |
| 2 | Ferienwohnung Wiesenglück | Obertalstraße 24, 79263 Simonswald | Bad renovierungsbedürftig |
| 5 | Pension Löwenwirt Zi1 | Geretsrieder Str. 21, 88132 Lindau | |
| 10 | Pension Löwenwirt Zi2 | Geretsrieder Str. 21, 88132 Lindau | |
| 11 | Pension Löwenwirt Zi3 | Geretsrieder Str. 21, 88132 Lindau | |
| 6 | Pension Zum Hirschen | Mühlenweg 19, 79859 Schluchsee | |
| 8 | Stadtzimmer 1 Leopoldplatz | Hüttenstr. 54, 10572 Berlin | Nur AirBNB |
| 9 | Stadtzimmer 2 Leopoldplatz | Hüttenstr. 54, 10572 Berlin | Nur AirBNB |
| 7 | Stadtzimmer Panketal | Waldstr. 13, 70823 Möhringen | Nur AirBNB |
| * | (Neu) | | |

Bild 10: Die Tabelle **tblWohnungen** speichert alle vermiet- und belegbaren Objekte

```
Private Sub FillCalendar()  
    (...)  
    Set rs = CurrentDb.OpenRecordset("tblCalendarMonths", dbOpenDynaset)  
    For n = 0 To 4  
        rs.AddNew  
        For i = 1 To 3  
            MaxDate = DateSerial(m_Year, m_Month + i - 1, 0)  
            For j = 1 To 7  
                DTmp = DateAdd("m", i - 1, StartDate)  
                DTmp = DateAdd("d", n * 7 + j - 1, DTmp)  
                If DTmp > MaxDate Then Exit For  
                lType = IsInRanges(DTmp)  
                rs.Fields("D" & CStr(j) & CStr(i)).Value = CLng(DTmp) & "_" & CStr(lType)  
            Next j  
        Next i  
        rs.Update  
    Next n  
    (...)  
End Sub
```

Listing 1: Neues Befüllen der Tabelle **tblCalendarMonths**

Deshalb änderten wir alle Felder auf den Typ **Text**. Der mögliche Inhalt der Felder geht aus der Datenblattansicht in Bild 9 hervor, nachdem sich die Tabelle über das Formular **sfrmCalendarMonths** füllte.

Das eigentliche Datum steht nun an vorderster Stelle und ist immer 5 Zeichen lang. Danach schließt sich als visueller Trenner ein Unterstrich an, auf den der Typ der Belegung als Ziffer folgt. Es ist logisch, dass diese Änderung der Tabelle dann auch Modifikationen der beteiligten Routinen im Formular, wie der Prozedur **FillCalendar**, erfordert. Dazu später mehr.

Kunden und Wohnungsobjekte

Für die Kunden und Wohnungen der Datenbank gibt es keine Anzeige- oder Eingabeformulare, da wir uns komplett auf die Kalenderansichten konzentrieren. Diese müssten Sie gegebenenfalls selbst nachrüsten. Die Datensätze sind bei den Kunden aus den hinlänglich bekannten Demo-Adresstabellen von **Access Basics** importiert. Die Wohnobjekte sind natürlich fingiert. Bild 10 zeigt alle in die Beispieldatenbank eingebauten.

Neben verlockenden Ferienwohnungen erfanden wir noch Einzelzimmer und Pensionszimmer. Die **Pension Zum Hirschen** kann nur ein Zimmer anbieten.

Die Einträge in der Spalte **Vermerk** zeigen, wofür Sie intern für den Vermieter verwendet werden könnte. Wollten Sie die Sache professionell gestalten, so enthielte diese Tabelle noch weit mehr Felder. So etwa zu Ausstattungsmerkmalen oder in einem **Anlagefeld** Fotos des Interieurs.

Änderungen am Kalendersteuerelement **sfrmCalendarMonths**

Nachdem die Tabelle **tblCalendarMonths** nun **String**-, statt **Long**-Werte erwartet, muss die Routine **FillCalendar** zum Füllen der Tabelle komplett überarbeitet werden. In Listing 1 ist nur jener Teil abgebildet, der sich gegenüber der Ursprungsprozedur verändert hat.

Dabei ist der Grundaufbau mit den drei ineinander verschachtelten Schleifen gleich geblieben. Ebenso die Berechnung der Datumswerte in der Variablen **DTmp** und die Abbruchbedingung beim Vergleich mit **MaxDate**.

```

Private Type TRange
    StartDate As Date
    EndDate As Date
    Type As Long
End Type

Private Function IsInRanges(D As Date) As Long
    Dim i As Long, n As Long
    On Error Resume Next
    n = UBound(arrRanges)
    If Err.Number <> 0 Then
        On Error GoTo 0: Exit Function
    End If
    On Error GoTo 0
    For i = 0 To n
        If (D >= arrRanges(i).StartDate) And (D <= arrRanges(i).EndDate) Then
            IsInRanges = arrRanges(i).Type: Exit For
        End If
    Next i
End Function

```

Listing 2: Auch die Hilfsfunktion **IsInRanges** ändert sich

```

Public Sub AddRange(StartDate As Date, EndDate As Date, Typ As Long)
    (...)
    ReDim Preserve arrRanges(n)
    arrRanges(n).StartDate = StartDate
    arrRanges(n).EndDate = EndDate
    arrRanges(n).Type = Typ
    (...)
End Sub

```

Listing 3: Prozedur zum Zuweisen eines Zeitraums

```

Sub SetCtlSource()
    Dim frm As Access.Form
    Dim ctl As Access.TextBox
    Dim S As String

    Set frm = Screen.ActiveForm
    For Each ctl In frm.Section(acDetail).Controls
        S = "[" & Mid(ctl.Name, 4) & "]"
        ctl.ControlSource = "=IIF(IsNull(" & S & "),"",",Day(Left(" & S & ",5)))"
    Next ctl
End Sub

```

Listing 4: Automatisiertes Setzen der Textbox-Ausdrücke

Neu hingegen ist die Speicherung des Belegungstyps in der Long-Variablen **IType**, der aus der ebenfalls modifizierten Routine **IsInRanges** (s. Listing 2) stammt. Diese gibt nun nicht mehr einfach **True** oder **False** zurück, sondern eine Zahl, die der **ID** in der Tabelle **tblBelegungsarten** entspricht. Einem Datenfeld von **tblCalendarMonths** wird schließlich der per **CLng** erhaltene Integer-Wert des Datums **DTmp** zugewiesen, gefolgt vom Unterstrich und dem Inhalt der soeben ermittelten Variablen **IType**.

Das Array **arrRanges** des Moduls, welches die Zeiträume aufnimmt, basiert nun auf dem erweiterten Typ **TRange** im Modulkopf. Neben **StartDate** und **EndDate** weist dieser Typ nun zusätzlich das Element **Type** aus. Beim Zuweisen eines Zeitraums über die Methode **AddRange** müssen Sie nun auch den Belegungstyp angeben (s. Listing 3).

Ermittelt die Funktion **IsInRanges**, dass sich das übergebene Datum **D** innerhalb der abgespeicherten Zeiträume befindet, so gibt sie nun die Zahl aus **arrRanges(i).Type** zurück.

Mehr gibt es zum Füllen der Tabelle **tblCalendarMonths** nicht zu sagen. Was nun noch aussteht, sind Änderungen an den Textboxen des Detailbereichs, denn weder

Notizen nach Kunde

In Heft 2/2016 haben wir gezeigt, wie Sie Notizen in einer Art Endlosformular anzeigen können – und zwar nicht mit konstanter Höhe, wie im Access-Endlosformular üblich, sondern mit einer an den Inhalt angepassten Höhe. Außerdem ließen sich damit Texte im Richtext-Format eingeben! Diesmal bauen wir eine Erweiterung, mit der Sie die Notizen einem Kunden zuordnen und diese wieder abrufen können. Außerdem fügen wir noch weitere praktische Funktionen hinzu, zum Beispiel zum direkten Bearbeiten der Notiz per Doppelklick.

Datenmodell erweitern

In der Lösung aus dem Beitrag **HTML-Liste mit Access-Daten** (www.access-im-unternehmen.de/1029) haben wir nur die Tabelle **tblNotizen** beziehungsweise **tblNotizenRichtext** verwendet, um die Notizen einzugeben (im vorliegenden Beitrag konzentrieren wir uns auf die Version mit Richtext). Nun wollen wir das Datenmodell so erweitern, dass es auch noch eine Kunden-Tabelle samt Anreden enthält. Der Tabelle **tblNotizen_Richtext** fügen wir nun ein Fremdschlüsselfeld namens **KundeID** hinzu, über das wir die Tabelle mit dem Primärschlüsselfeld **KundeID** der Tabelle **tblKunden** verknüpfen. Das Datenmodell soll danach wie in Bild 1 aussehen.

Den Entwurf der Tabelle **tblNotizen_Richtext** ändern wir dazu wie in Bild 2. Hier fügen wir das Fremdschlüsselfeld **KundeID** hinzu. Sie können die Beziehung über den Datentyp **Nachschlagefeld** herstellen, können im Anschluss aber die Eigenschaft **Steuerelement anzeigen** unter **Nachschlagen** auf Textfeld einstellen – das Feld **KundeID** wird nicht in Zusammenhang mit der Notiz angezeigt werden,

somit brauchen wir über das Nachschlagefeld auch nicht die Vorbereitung für das Anlegen von Kombinationsfel-

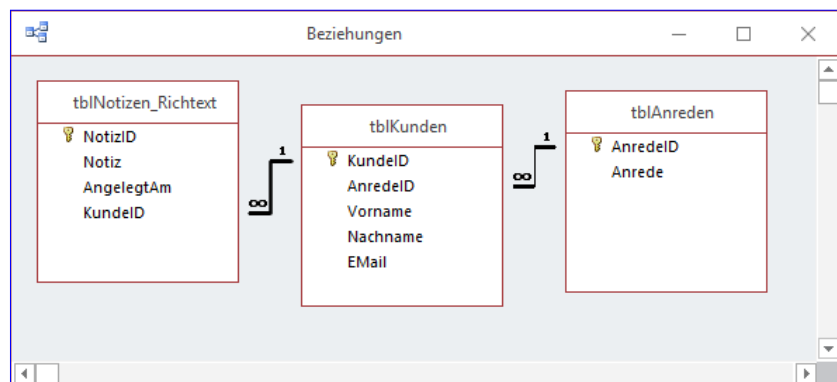


Bild 1: Angepasstes Datenmodell mit einer Beziehung zwischen Kunden und Notizen

| Feldname | Felddatentyp | Beschreibung (optional) |
|------------|---------------|--|
| NotizID | AutoWert | Primärschlüsselfeld der Tabelle |
| Notiz | Langer Text | Inhalt der Notiz im Richtext-Format |
| AngelegtAm | Datum/Uhrzeit | Anlagedatum |
| KundeID | Zahl | Fremdschlüsselfeld zur Tabelle tblKunden |

Bild 2: Tabelle **tblNotizen** mit Fremdschlüsselfeld

dem mit den gleichen Eigenschaften vorzubereiten.

Kundenformular

Als Nächstes bauen wir ein Formular als kombinierte Kundenübersicht/Detailansicht. Dazu fügen wir dem neuen Formular namens **frmKunden** im oberen Bereich ein Listenfeld hinzu, das die folgende Abfrage als Datenherkunft verwendet:

```
SELECT tb1Kunden.KundeID, [Nachname] &
", " & [Vorname] AS Kunde FROM tb1Kunden
ORDER BY [Nachname] & ", " & [Vorname];
```

Damit das Listenfeld nur den Nachnamen und den Vornamen des Kunden anzeigt und nicht den Primärschlüsselwert, stellen wir die Eigenschaft **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** ein. Das Formular soll die Auswahl des Kunden über das Listenfeld ermöglichen und dann die Daten des Kunden als gebundene Steuerelemente im Formular anzeigen. Dadurch benötigen wir die üblichen Formularelemente zur Navigation in den Datensätzen des Formulars nicht und stellen die Eigenschaften **Datensatzmarkierer**, **Navigationsschaltflächen**,

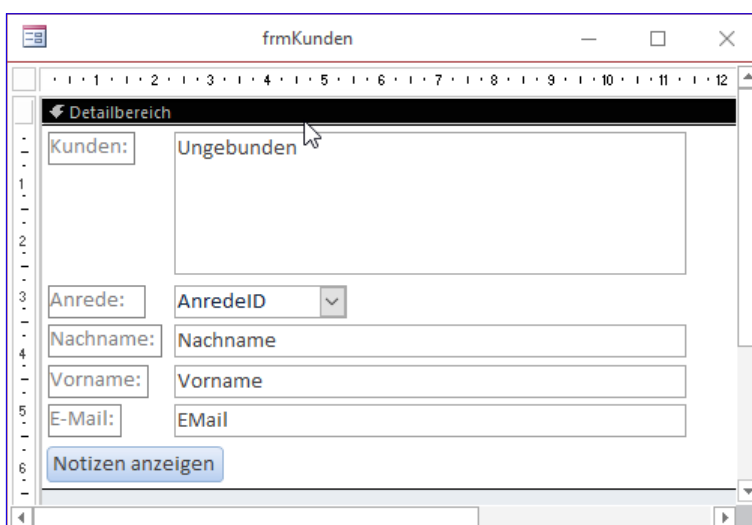


Bild 4: Entwurf des Kunden-Formulars

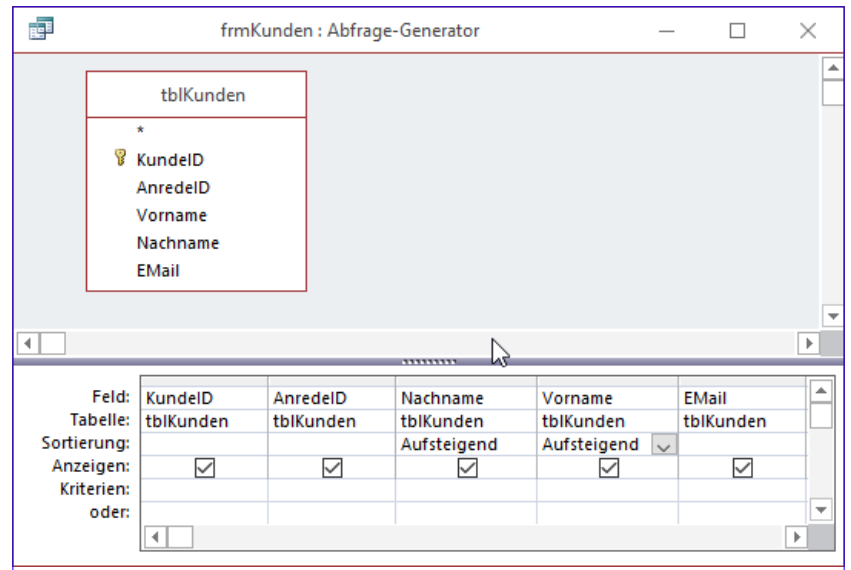


Bild 3: Datenherkunft des Kunden-Formulars

Trennlinien und **Bildlaufleisten** auf **Nein** ein. Danach binden Sie das Formular an eine Abfrage auf Basis der **tblKunden**, die alle Felder der Tabelle enthält, die Datensätze aber zuerst nach dem Nachnamen und dann nach dem Vornamen sortiert (s. Bild 3).

Anschließend ziehen die relevanten Felder wie in Bild 4 aus der Feldliste in den Detailbereich des Formularentwurfs. Damit das Formular den jeweils im Listenfeld markierten Datensatz anzeigt, sind ein paar Zeilen Code notwendig. Da wir sowohl für das Listenfeld als auch für das Formular eine Datenherkunft beziehungsweise Datensatzherkunft verwenden, welche die gleichen Datensätze nach identischer Sortierung liefert, entspricht der im Formular angezeigte Datensatz automatisch dem zuerst angezeigten Datensatz im Listenfeld. Diesen wollen wir zur Verdeutlichung auch gleich noch markieren. Dazu hinterlegen Sie für das Ereignis **Beim Laden** des Formulars die folgende Ereignisprozedur:

```
Private Sub Form_Load()
    Me!lstKunden = Me!lstKunden.ItemData(0)
End Sub
```

Die einzige Anweisung dieser Prozedur markiert den ersten Eintrag des Listenfeldes. Danach fügen wir noch eine Prozedur hinzu, die durch das Ändern der Auswahl der im Listinfeld angezeigten Einträge ausgelöst wird:

```
Private Sub 1stKunden_AfterUpdate()
    Me.Recordset.FindFirst "KundeID = " & Me!1stKunden
End Sub
```

Dies stellt den Inhalt des Formulars auf den aktuell im Listinfeld markierten Eintrag ein.

Schließlich fügen Sie noch eine Schaltfläche namens **cmdNotizenAnzeigen** zum Formular hinzu. Bevor wir die Ereignisprozedur für die **Beim Klicken**-Eigenschaft dieser Schaltfläche definieren, wollen wir zunächst das Formular zur Anzeige der Notizen anpassen.

frmNotizen_Richtext anpassen

Bei der Anpassung dieses Formulars setzen wir auf dem Formular **frmNotizenMitDetailFormular_Rechtext** auf, das wir in dem oben angegebenen Beitrag erstellt haben. Wir benennen es der Einfachheit halber allerdings in **frm-Notizen_Richtext** um.

Der HTML-Code für die Darstellung der Notizen in dem Webbrowser-Steuerelement aus Bild 5 befindet sich in der Prozedur **DatenAnzeigen** im Klassenmodul des Formulars. Hier füllen wir eine Recordset-Variable mit den anzuzeigenden Daten, derzeit mit der folgenden Abfrage:

```
Private Sub DatenAnzeigen()
    ...
    Set rst = db.OpenRecordset("SELECT * FROM 7
        tblNotizen_Richtext", dbOpenDynaset)
    ...
End Sub
```

Diese Abfrage lädt aktuell alle Notizen und stellt diese zu einem HTML-Dokument zusammen, das dann im Webbrowser-Steuerelement angezeigt wird. Diese Abfrage

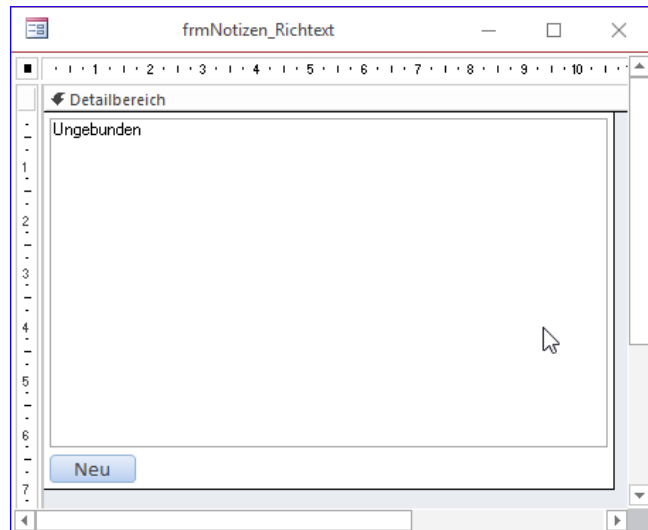


Bild 5: Entwurf des Formulars zur Anzeige der Notizen zu einem Kunden

müssen wir nun so modifizieren, dass nicht mehr alle Datensätze der Tabelle **tblNotizen_Richtext**, sondern nur noch die zu dem jeweiligen Kunden angezeigt werden, für den das Formular geöffnet wurde.

Die Anpassung der Zeile mit der Abfrage ist theoretisch leicht – wir müssen ja nur eine **WHERE**-Bedingung in der Form **WHERE KundeID = <KundeID>** hinzufügen. Dabei müssen wir allein dafür sorgen, dass die ID des jeweiligen Kunden vom aufrufenden Formular in das aufgerufene Formular gelangt und als Variable verfügbar ist. Das ist der ideale Anwendungsfall für den Parameter **OpenArgs** beim Aufrufen eines Formulars mit der Methode **DoCmd.OpenForm**. Die Prozedur **cmdNotizenAnzeigen** des Formulars **frmKunden** können wir also wie folgt füllen:

```
Private Sub cmdNotizenAnzeigen_Click()
    DoCmd.OpenForm "frmNotizen_Richtext",
        WindowMode:=acDialog, OpenArgs:=Me!KundeID
End Sub
```

Dies übergibt den aktuellen Wert des Feldes **KundeID** an das aufgerufene Formular und kann dort über die Eigenschaft **OpenArgs** abgerufen werden. Das heißt, dass wir die oben erwähnte Zeile zum Initialisieren des Recordsets wie folgt ändern müssen:

```
Private Sub DatenAnzeigen()
    ...
    Set rst = db.OpenRecordset("SELECT * FROM 7
        tblNotizen_Richtext WHERE KundeID = " 7
        & Me.OpenArgs, dbOpenDynaset)
    ...
End Sub
```

Erster Test des Notizformulars

Zeigen wir das Formular **frmKunden** nun an und klicken für einen beliebigen Kunden auf die Schaltfläche **cmd-NotizenAnzeigen**, erscheint das Notiz-Formular wie in Bild 6. Natürlich sind noch für keinen Kunden Notizen vorhanden, denn wir haben ja noch keine angelegt. Die Anzeige des Notizen-Formulars für einen Kunden ohne Notizen sieht allerdings nicht so schön aus – da müssen wir später nachbessern.

Neue Notiz anlegen

Zunächst wollen wir aber einmal eine neue Notiz anlegen. Dies haben wir in der Lösung, auf der dieser Beitrag aufbaut, mit dem Formular **frmNotizDetail_Richtext** erledigt, das wir mit der folgenden Prozedur aufgerufen haben:

```
Private Sub cmdNeu_Click()
    DoCmd.OpenForm "frmNotizDetail_Richtext", 7
        DataMode:=acFormAdd, WindowMode:=acDialog
    DatenAnzeigen
End Sub
```

Diese Prozedur zeigt das Formular **frmNotizDetail_Richtext** an. Nach der Eingabe wird die Routine **DatenAnzeigen** aufgerufen, welche die HTML-Liste mit den Notizen aktualisieren soll. Wenn wir uns jedoch die **DoCmd.OpenForm**-Methode ansehen, wird schnell klar, dass hier etwas fehlt: Der Primärschlüsselwert für den aktuellen Kunden wird hier nämlich übergeben. Dadurch können wir im aufgerufenen Formular natürlich auf keinen Fall eine Notiz im Kontext des gewünschten Kunden anlegen. Also ändern wir den Aufruf wie folgt:

```
DoCmd.OpenForm "frmNotizDetail_Richtext", 7
        DataMode:=acFormAdd, WindowMode:=acDialog, 7
        OpenArgs:=Me.OpenArgs
```

Wir geben also das vom ersten Formular weitergegebene Öffnungsargument gleich noch an das nächste Formular weiter. Dort müssen wir es natürlich adäquat verarbeiten.

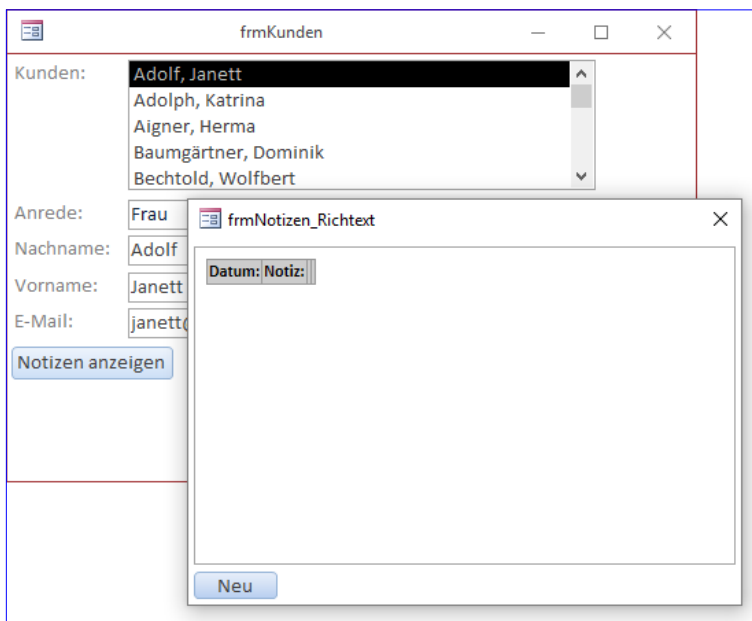


Bild 6: Anzeigen der Notizen zu einem Kunden

Kunde beim Anlegen einer neuen Notiz berücksichtigen

Die Tabelle **tblNotizen_Richtext** haben wir ja mit dem Feld **KundeID** ausgestattet, damit wir zu jeder Notiz speichern können, zu welchem Kunden diese gehört. Diesen Wert müssen wir beim Anlegen einer neuen Notiz natürlich auch füllen. Dazu ziehen wir zunächst das Feld **KundeID** aus der Feldliste in den Entwurf des Formulars (s. Bild 7). Sie können das Feld ruhig an beliebiger Stelle positionieren, denn wir benötigen es nur, um den Standardwert festzulegen – der Benutzer braucht dieses Feld später nicht zu sehen. Zu Testzwecken lassen wir es jedoch zunächst noch eingeblendet.

Kundenverwaltung mit Ribbon, Teil II

Im Beitrag »Ribbonklassen« (www.access-im-unternehmen.de/1069) haben wir Klassen für die Anzeige von Ribbons und den enthaltenen Steuerelementen eingeführt, die wir in »Kundenverwaltung mit Ribbon, Teil I« (www.access-im-unternehmen.de/1091) in eine Anwendung eingebaut haben. Ein Leser fragte nun, ob man damit nicht auch benutzerabhängige Ribbons anzeigen kann. Natürlich geht das – wie es funktioniert, zeigt der vorliegende Beitrag. Darüber hinaus erweitern wir unser Beispiel noch um ein paar neue Elemente.

Benutzerdefinierte Ribbon-Einträge

Im ersten Beispiel wollen wir das Ribbon, das mit dem Formular **frmKundenebersicht** angezeigt wird, in Abhängigkeit vom angemeldeten Benutzer die Schaltfläche zum Löschen eines Kunden aktiviert oder deaktiviert.

Zu diesem Zweck wollen wir nun keine Benutzerverwaltung zur Anwendung hinzufügen, sondern zeigen einfach beim Öffnen der Anwendung ein Meldungsfenster an, das den Benutzer fragt, ob er ein Administrator ist oder nicht.

Nur Administratoren sollen Kunden löschen dürfen. Das Ergebnis dieser Abfrage speichern wir dann in einer temporären Variablen (**TempVar**).

Dazu ändern wir die beim Öffnen durch das **AutoExec**-Makro aufgerufene Prozedur wie in Listing 1. Mit der **MsgBox**-Funktion ermitteln wir den Wert für die lokale **Boolean**-Variable **bolAdministrator**, deren Inhalt wir dann in die neu zu erstellende **TempVar**-Variable mit dem Namen **IstAdministrator** schreiben.

In der Code behind-Klasse des Formulars **frmKundenebersicht** ist eine kleine Änderung am Code notwendig. Sie müssen lediglich die Prozedur **Form_Load**, die beim Laden des Formulars ausgelöst wird, wie folgt erweitern:

```
Private Sub Form_Load()  
    CreateRibbon  
    If TempVars("IstAdministrator") = True Then  
        btnKundeLoeschen.Enabled = True  
    Else  
        btnKundeLoeschen.Enabled = False  
    End If  
End Sub
```

Die Prozedur prüft nach dem Erstellen des Ribbons durch die Prozedur **CreateRibbon**, ob die **TempVars**-Variable **IstAdministrator** den Wert **True** enthält. Falls ja, wird die Eigenschaft **Enabled** des Objekts **btnKundeLoeschen** auf **True** eingestellt, sonst auf den Wert **False**.

Die kürzere Variante dieses Konstrukts sieht so aus:

```
Public Function RibbonLaden()  
    Dim bolAdministrator As Boolean  
    bolAdministrator = MsgBox("Sind Sie ein Administrator (Ja) oder ein normaler Benutzer (Nein)?", vbYesNo) = vbYes  
    TempVars.Add "IstAdministrator", bolAdministrator  
    Startribbon.CreateRibbon  
End Function
```

Listing 1: Speichern des Benutzer-Status beim Öffnen der Anwendung

```
Private Sub Form_Load()
    CreateRibbon
    btnKundeLoeschen.Enabled = TempVars("IstAdministrator")
End Sub
```

Wenn Sie beim Öffnen der Anwendung nun die Meldung mit der Frage nach Administrator oder Benutzer mit **Benutzer** beantworten, sieht das Formular **frmKundenübersicht** wie in Bild 1 aus – mit deaktivierter Schaltfläche **Kunde löschen**.

Warum können Sie die Eigenschaft **Enabled** nicht gleich beim Definieren auf **False** einstellen? In unserem Beispiel geschieht Folgendes: Sie erstellen das Ribbon mit seinen Tabs, Groups und Buttons. Für die Buttons wird dabei, soweit nicht anders angegeben, die **get...-Version** eines Attributs angegeben. Sie können also beispielsweise zum Festlegen eines Bildes, das sich nicht ändern soll, die **image**-Eigenschaft nutzen:

```
With btnKundeLoeschen
    .Label = "Kunde löschen"
    .Size = msoRibbonControlSizeLarge
```

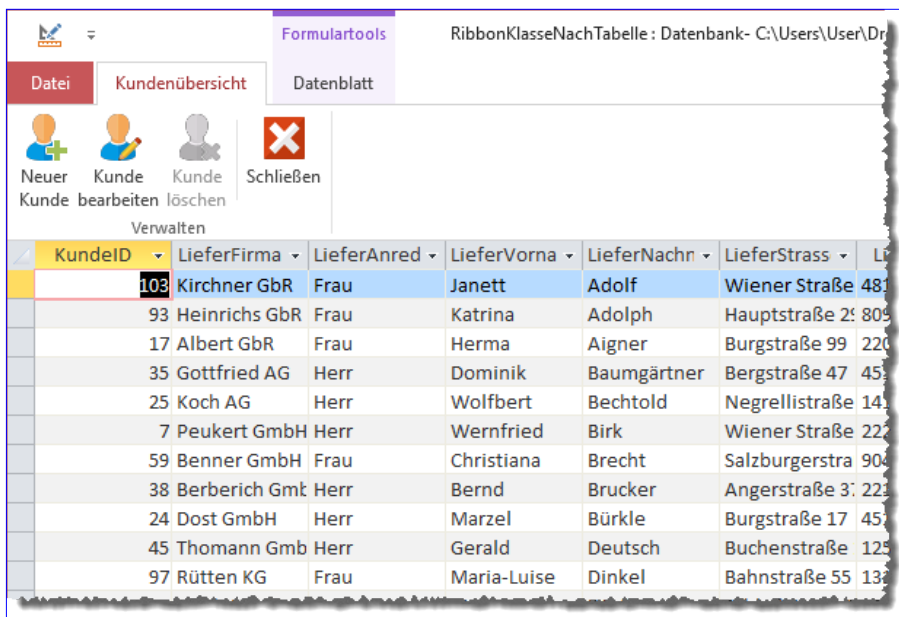


Bild 1: Deaktivierte Löschen-Schaltfläche

```
.Image = "user_delete"
End With
```

Dies resultiert dann in der folgenden Definition:

```
<button label="Kunde löschen" id="ribKundenuebersicht_
btnKundeLoeschen" size="large" image="user_delete"
onAction="OnAction" enabled="true" visible="true" />
```

Oder Sie wollen das Bild zur Laufzeit dynamisch zuweisen. Dann stellen Sie einfach noch keinen festen Wert für die Eigenschaft **Image** ein:

```
With btnKundeLoeschen
    .Label = "Kunde löschen"
    .Size = msoRibbonControlSizeLarge
End With
```

In diesem Fall erzeugen die Ribbon-Klassen den folgenden XML-Code für das Element:

```
<button label="Kunde löschen" id="ribKundenuebersicht_
btnKundeLoeschen" size="large" getImage="GetImage"
onAction="OnAction"
getEnabled="GetEnabled"
setVisible="GetVisible" />
```

Sie sehen, dass hier nicht mehr die Eigenschaft **image** verwendet wird, sondern die Eigenschaft **getImage**. Damit wird das Bild auch nicht mehr direkt festgelegt, sondern Sie können es zu einem späteren Zeitpunkt über die Eigenschaft **Image** der Objektvariablen für das Element einstellen.

Wichtig ist also der Zeitpunkt, zu dem Sie die Eigenschaft **Image** füllen – vor dem Erstellen der Ribbon-Definition durch die Me-

thode **GetRibbon** der Klasse **Ribbons** oder erst danach. Vorher wird das Image über das Attribut **image** fest eingebunden, nachher wird es über die Callback-Methode **getImage** ermittelt.

Das Gleiche gilt auch für die beiden Eigenschaft **Visible** und **Enabled**. Deshalb dürfen Sie, wenn Sie **Visible** oder **Enabled** später dynamisch setzen wollen, die entsprechende Eigenschaft nicht vor dem Erstellen der Ribbon-Definition setzen, sondern erst danach.

Was ist der Unterschied? Theoretisch könnte man auch einfach immer die **get...**-Attribute nutzen und die Werte bei jedem neuen Einblenden des Ribbons erneut abrufen. Allerdings kostet auch dies Performance, und warum sollten wir, wenn ein Element feste Attribute wie etwa Bilder hat, Zeit vergeuden – auch wenn es nur sehr wenig Zeit ist?

Wenn Sie sich die **Kunde löschen**-Schaltfläche ansehen und entscheiden wollen, ob diese abhängig vom jeweiligen Benutzer entweder per Definition oder dynamisch aktiviert oder deaktiviert werden soll, gibt es beispielsweise nur einen praktischen Ansatz: Sie sollten diese vor der Definition mit **btnKundeLoeschen.Enabled** entsprechend des aktuellen Benutzers einstellen. Wenn Sie sich für die dynamische Variante entschieden, kann die **Enabled**-Eigenschaft später jederzeit per Code geändert werden – Sie müssen dann dort immer nochmals prüfen, ob der aktuelle Benutzer diese Schaltfläche überhaupt verwenden darf.

Schaltflächen mit Leben füllen

Damit kommen wir zu den im ersten Teil dieser Beitragsreihe versprochenen Funktionen, denn wir haben dort ja zunächst einige Schaltflächen im Ribbon zum Formular **frmKundeneubersicht** angelegt. Mit diesen wollen wir die folgenden Aktionen durchführen:

- einen neuen Kunden anlegen,

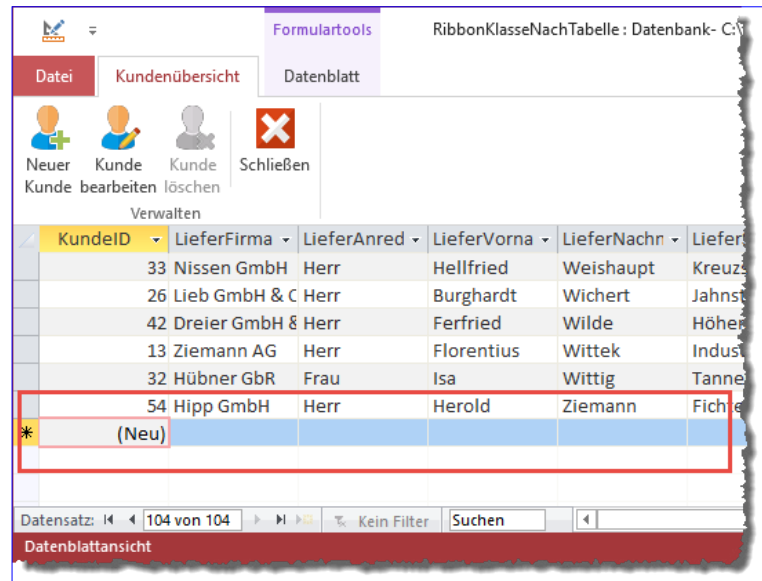


Bild 2: Aktuell können noch neue Einträge zum Formular **frmKundeneubersicht** hinzugefügt werden.

- den aktuell markierten Kunden bearbeiten und
- den aktuell markierten Kunden löschen.

Die erste Aufgabe erfordert keine Vorarbeiten, denn wir können den Dialog zum Anlegen eines neuen Kunden direkt öffnen und dem Benutzer die Möglichkeit geben, den neuen Kunden zu bearbeiten.

Nach dem Klick auf die **OK**-Schaltfläche soll das Formular geschlossen und das Formular **frmKundeneubersicht** aktualisiert werden.

Die zweite und die dritte Aufgabe erfordern es, dass der Benutzer einen der Kunden im Formular **frmKundeneubersicht** markiert hat. Aber ist das überhaupt nötig? Momentan können wir mindestens noch neue Datensätze über die letzte Zeile der Datenblattansicht hinzufügen (s. Bild 2). Da wir aber ein eigenes Formular für das Anlegen und Bearbeiten von Kunden einsetzen wollen, das auch Aufgaben wie die Validierung der eingegebenen Daten ausführen könnte, sollten wir die Möglichkeit der Eingabe neuer Datensätze unterbinden.

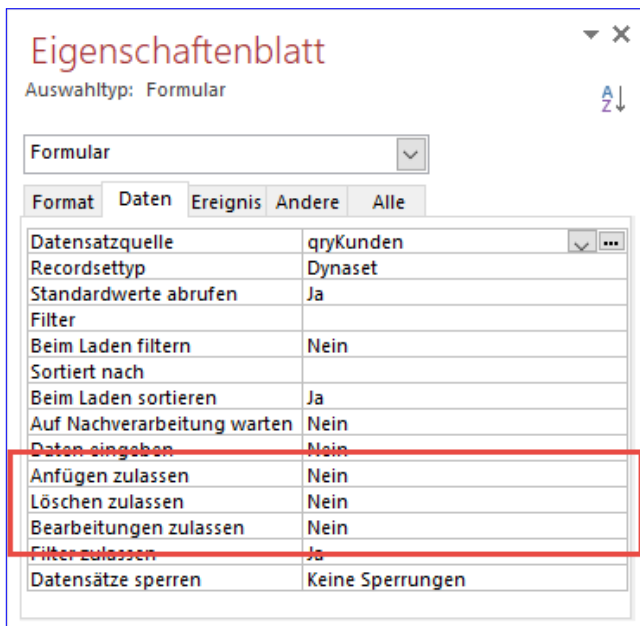


Bild 3: Löschen, Bearbeiten und Anlegen von Daten im Formular `frmKundenebersicht` verhindern

Dazu stellen Sie in den Eigenschaften des Formulars die Eigenschaft **Anfügen zulassen** auf den Wert **Nein** ein. Da wir auch zum Bearbeiten ein entsprechendes Detailformular öffnen wollen, können Sie auch die Eigenschaft **Bearbeitungen zulassen** auf **Nein** einstellen. Und schließlich soll auch das Löschen nur über die entsprechende Ribbon-Schaltfläche geschehen, sodass auch **Löschen zulassen** den Wert **Nein** erhält (s. Bild 3).

Müssen wir also noch prüfen, ob ein Artikel markiert ist, bevor der Benutzer auf eine der Schaltflächen **Kunde bearbeiten** oder **Kunde löschen** klickt? Ja, denn es gibt noch eine einzige Möglichkeit, bei der kein Datensatz

markiert ist – dann nämlich, wenn die zugrunde liegende Tabelle **tblKunden** überhaupt keinen Datensatz enthält.

Schaltflächen abhängig von Datensätzen aktivieren und deaktivieren

Wir sollten also dafür sorgen, dass eine entsprechende Meldung erscheint, wenn der Benutzer eine der beiden Schaltflächen **Kunde bearbeiten** oder **Kunde löschen** klickt, während kein Kunden-Datensatz markiert ist. Wobei dieser Ansatz nicht ganz ergonomisch ist, denn warum sollte der Benutzer überhaupt einen der Befehle anklicken können, obwohl wir die beiden Schaltflächen auch einfach deaktivieren könnten? Immerhin haben wir mit unseren Ribbon-Klassen ja für eine einfache Steuerung dieser Elemente per VBA gesorgt.

Welches Ereignis aber nutzen wir, um zu prüfen, ob sich aktuell ein Datensatz in der Tabelle befindet oder nicht? Der erste Ansatz wäre die Eigenschaft **Beim Anzeigen**, die bei jedem Wechsel auf einen anderen Datensatz ausgelöst wird. Wir legen dazu zunächst eine Ereignisprozedur für die Schaltfläche **btnKundeLoeschen** des Ribbons an, welche den aktuell markierten Datensatz im Formular `frmKundenebersicht` löschen soll. Diese soll im ersten Ansatz wie in Listing 2 aussehen. Nachdem die Schaltfläche **Kunde löschen** wie im Artikel **btn-KundeLoeschen** beschrieben wie folgt angelegt wurde, können Sie ihre Ereignisprozeduren implementieren:

```
Dim WithEvents btnKundeLoeschen As clsButton
```

```
Private Sub btnKundeLoeschen_OnAction(control As Office.IRibbonControl)
    Dim db As DAO.Database
    If MsgBox("Kunde wirklich löschen?", vbYesNo) = vbYes Then
        Set db = CurrentDb
        db.Execute "DELETE FROM tblKunden WHERE KundeID = " & Me!KundeID, dbFailOnError
        Me.Requery
    End If
End Sub
```

Listing 2: Löschen eines Kunden per Ribbon-Schaltfläche

Dazu wählen Sie im Codefenster des Klassenmoduls des Formulars **frmKundenuebersicht** erst im linken Kombinationsfeld den Eintrag **btnKundeLoeschen** aus und dann im rechten den Eintrag **OnAction** (s. Bild 4). Dies erstellt eine neue, leere Ereignisprozedur namens **btnKundeLoeschen_OnAction**, die Sie dann entsprechend füllen.

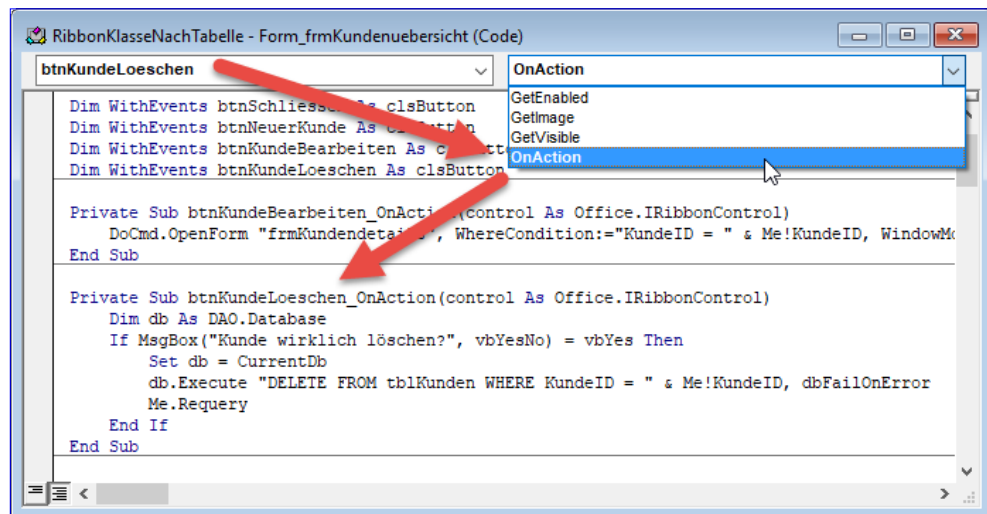


Bild 4: Anlegen einer Ereignisprozedur für eine Ribbon-Schaltfläche

Die Prozedur fragt den Benutzer, ob er den Datensatz wirklich löschen will und führt dann gegebenenfalls eine entsprechende SQL-Anweisung aus. Anschließend sorgt sie durch den Aufruf der Methode **Requery** des Formulars selbst (**Me**) für die Aktualisierung der angezeigten Datensätze. Und nun kommt das Problem: Wir legen zuvor die Ereignisprozedur an, die durch das Ereignis **Beim Anzeigen** ausgelöst wird, und fügen zwei Anweisungen hinzu, welche die beiden Schaltflächen **btnKundeBearbeiten** und **btnKundeLoeschen** des Ribbons aktivieren, wenn im Formular ein Datensatz markiert ist und deaktivieren, wenn kein Datensatz markiert ist:

```
Private Sub Form_Current()  
    btnKundeLoeschen.Enabled = Not IsNull(Me!KundeID)  
    btnKundeBearbeiten.Enabled = Not IsNull(Me!KundeID)  
End Sub
```

Allerdings wird die Prozedur **Form_Current** genau nur dann ausgelöst, wenn der Datensatzzeiger auch auf einen Datensatz springt – aber nicht, wenn kein Datensatz mehr vorhanden ist! Wenn Sie den letzten Datensatz gelöscht haben, feuert dieses Ereignis also nicht und die Schaltflächen **btnKundeBearbeiten** und **btnKundeLoeschen** werden nicht deaktiviert.

Warum nochmal verwenden wir eigentlich eine SQL-Anweisung, um einen Datensatz per Ribbon-Schaltfläche zu löschen, und rufen nicht einfach die Methode **RunCommand acCmdDeleteRecord** auf? Weil diese Methode deaktiviert ist, wenn wir die Eigenschaft **Löschen zulassen** des Formulars auf **Nein** eingestellt haben – und wir wollten ja erreichen, dass der Benutzer Bearbeitungen der Datensätze nur über die drei Ribbon-Schaltflächen initialisieren kann und nicht über die üblichen Elemente der Datenblattansicht.

Also gibt es nur eine einfache Möglichkeit: Wir fügen die beiden Anweisungen noch zur Prozedur **btnKundeLoeschen_OnAction** hinzu, und zwar verbunden mit der Bedingung, dass die mit der Eigenschaft **RecordCount** ermittelte Anzahl der Datenstze des Recordsets des Formulars den Wert **0** enthält. Diesmal stellen wir die Werte der Eigenschaft **Enabled** allerdings nicht auf einen noch zu ermittelnden Wert ein, sondern direkt auf **False** – wir haben ja soeben schon geprüft, dass kein Datensatz mehr vorhanden ist (s. Listing 3). Auf diese Weise stellen wir die **Enabled**-Eigenschaften auch jeweils nicht mehr als einmal ein.

Das Ergebnis nach dem Löschen des letzten vorhandenen Datensatzes in der Tabelle **tblKunden** sieht dann wie in