HAUFE.
Ausgabe 01/2026

ACCESS
IM UNTERNEHMEN

ORDNER UND DATEIEN VERWALTEN

Lesen Sie Ordner und Dateien in Access ein und zeigen Sie diese in einem TreeView-Steuerelement an (ab Seite 50).

In diesem Heft:

WARUM REFERENZIELLE INTEGRITÄT IN ACCESS?

Lesen Sie, wie Sie mit Beziehungen mit referenzieller Integrität Inkonsistenzen und Fehleingaben verhindern.

SEITE 2

SYNCHRONE CHECKBOXEN IN HAUPT- UND UNTERFORMULAR

Entdecken Sie einen spannenden Anwendungsfall für Checkboxen in Formularen.

SEITE 25

SQL-ABFRAGEN MIT EXECUTE STATT RUNSQL

Lernen Sie die beiden Methoden kennen und erfahren Sie, warum »Execute« mehr Möglichkeiten bietet.

SEITE 46

Mat-Nr. 01583-5153

Ordner und Dateien mit Access verwalten

Auf einer Festplatte häuft sich mit der Zeit Einiges an. Der Windows Explorer oder Alternativen bieten meist ausreichende Funktionen, um der Datenmenge Herr zu werden. Aber manchmal würde man gern zusätzliche Informationen wie Metadaten zu Büchern, MP3s, Videodateien et cetera hinterlegen und diese durchsuchen können. Das Dateisystem bietet jedoch keine ausreichenden Möglichkeiten dazu. In dieser Ausgabe schauen wir uns daher an, wie wir Ordner und Dateien in Access einlesen und diese in einem TreeView-Steuerelement anzeigen können.

Eines vorneweg: Die Lösungen eignen sich nicht, um die Festplatte vollständig einzulesen, sondern sind eher für Unterordner mit einigen Ordnern und Dateien geeignet, da wir nicht um das Einlesen der einzelnen Elemente herumkommen. Der große Nutzen entsteht dadurch, dass die Dateien eine eindeutige File-ID aufweisen, die wir ebenfalls speichern und durch die wir den Dateien über Tabellen Metadaten hinzufügen können. Zu Beginn steht das Einlesen der Elemente, was wir ausführlich im Beitrag **Ordner und Dateien in Access-Tabellen einlesen** ab Seite 50 beschreiben.

Die Ordner und Dateien möchten wir übersichtlich darstellen, was sich am besten im **TreeView**-Steuerelement erledigen lässt. Hier können wir die Elemente in der gleichen Hierarchie abbilden, wie es auch im Windows Explorer der Fall ist – und wir haben sogar zusätzliche Möglichkeiten, da wir alle Elemente gleichzeitig im Baum abbilden können. Im Beitrag **Dateien schnell im TreeView-Steuerelement anzeigen** stellen wir ab Seite 64 eine performance-optimierte Version zum Füllen des TreeViews mit den Ordnern und Dateien vor.

Ein grundlegendes Problem vieler Access-Anwendungen ist die mangelhafte Definition von Beziehungen mit referenzieller Integrität, was in der Folge zu Inkonsistenzen und weiteren Problemen führen kann. Warum Beziehungen so wichtig sind, beschreiben wir im Beitrag **Warum Beziehungen mit referenzieller Integrität?** ab Seite 2.

Ein weiteres Problem entsteht, wenn man Daten wie die Kategorie eines Produkts mit einer 1:n-Beziehung verknüpft, wodurch sich genau eine Kategorie je Produkt auswählen lässt. Stellt man später fest, dass doch mehrere Kategorien je Produkt ausgewählt werden sollen, muss man das Datenmodell umbauen. Wie das gelingt, beschreiben wir im Beitrag **Reflexive 1:n-Beziehung zu m:n-Beziehung** ab Seite 10.

Wenn zu einem Auftrag mehrere Teilaufträge gehören, möchte man die Erledigung für beide Elemente anzeigen können. Das sollte synchron geschehen, sodass beim Markieren des Auftrags als »Erledigt« automatisch die Teilaufträge markiert werden und umgekehrt. Wie das gelingt, beschreiben wir unter dem Titel **Erledigt-Status in Haupt- und Unterformular synchron** ab Seite 25.

Schließlich gehen wir in den beiden Beiträgen **Daten bearbeiten: Execute vs. Recordset in DAO** (ab Seite 34) und **SQL ausführen mit Execute statt DoCmd.RunSQL** (ab Seite 46) noch auf verschiedene Techniken rund um das VBA-gesteuerte Ausführen von Abfragen zum Einfügen, Bearbeiten oder Löschen von Daten ein.

Viel Spaß beim Lesen wünscht Ihnen

Ihr André Minhorst

Warum Beziehungen mit referenzieller Integrität?

In unseren Access-Audits mit unseren Kunden treffen wir immer wieder auf das folgende Problem: Es gibt Tabellen, die zwar über ein Feld Datensätze aus anderen Tabellen referenzieren, aber es wurde gar keine Beziehung für diese Zuordnung definiert. Und wenn eine Beziehung angelegt wurde, wurde für diese keine referenzielle Integrität festgelegt. Das birgt verschiedene Gefahren, die unter Umständen sogar Auswirkungen auf den Unternehmensumsatz haben. Welche das sind und wie Sie diese Probleme beheben, zeigen wir in diesem Beitrag. Die Definition von Beziehungen mit referenzieller Integrität ist essenziell und sollte, wenn diese noch nicht vorhanden sind, schnellstens nachgerüstet werden. Das funktioniert in vielen Fällen aber gar nicht so leicht, weil die Tabellen bereits inkonsistente Daten enthalten. Auch zur Identifizierung und Korrektur solcher Datensätze liefert dieser Beitrag die passenden Lösungen.

Stellen wir uns vor, wir hätten eine Tabelle zum Speichern von Bestellungen und eine für die entsprechenden Bestellpositionen. Die Tabelle der Bestellungen enthält ein Primärschlüsselfeld, welches die erste Voraussetzung für ein konsistentes Datenmodell ist – somit können alle Datensätze dieser Tabelle eindeutig identifiziert werden.

Auch die Tabelle der Bestellpositionen enthält ein solches Primärschlüsselfeld. Neben den übrigen, typischen Feldern für eine Bestellposition enthält diese auch ein Feld, mit dem wir die Bestellposition einer Bestellung zuordnen können – nennen wir es **BestellungID**.

Grundsätzlich ist die Zuordnung von Bestellpositionen zu einer Bestellung also gewährleistet. Aber welche Probleme können auftreten, wenn wir keine referenzielle Integrität definieren?

Probleme bei fehlender Beziehung oder ohne referenzielle Integrität

Schauen wir uns das Datenmodell aus Bild 1 an. Hier sehen wir die beiden Tabellen, anhand derer wir die Vorteile der referenziellen Integrität beschreiben wollen.

Der Tabelle **tblBestellungen** fügen wir die Datensätze aus Bild 2 hinzu.

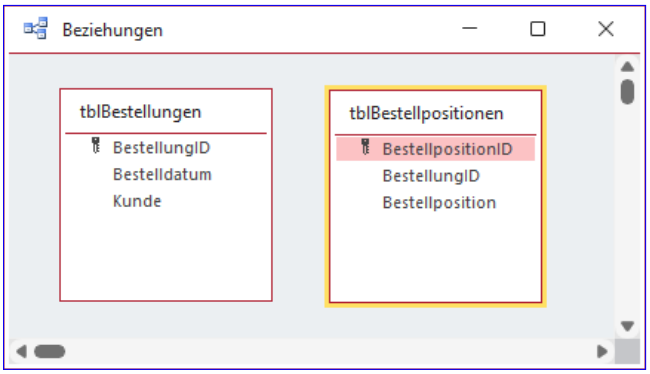

Bild 1: Tabellen der Beispieldatenbank

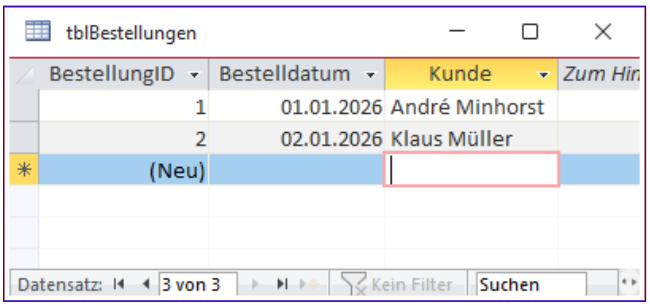

Bild 2: Beispieldaten in der Tabelle **tblBestellungen**

In der Tabelle **tblBestellpositionen** legen wir nun eine Bestellposition für die erste Bestellung an, indem wir das Feld **BestellungID** auf den Wert **1** einstellen, also auf die erste Bestellung der Tabelle **tblBestellungen**. Soweit ist das kein Problem – die Bestellposition ist einer Bestellung zugeordnet.

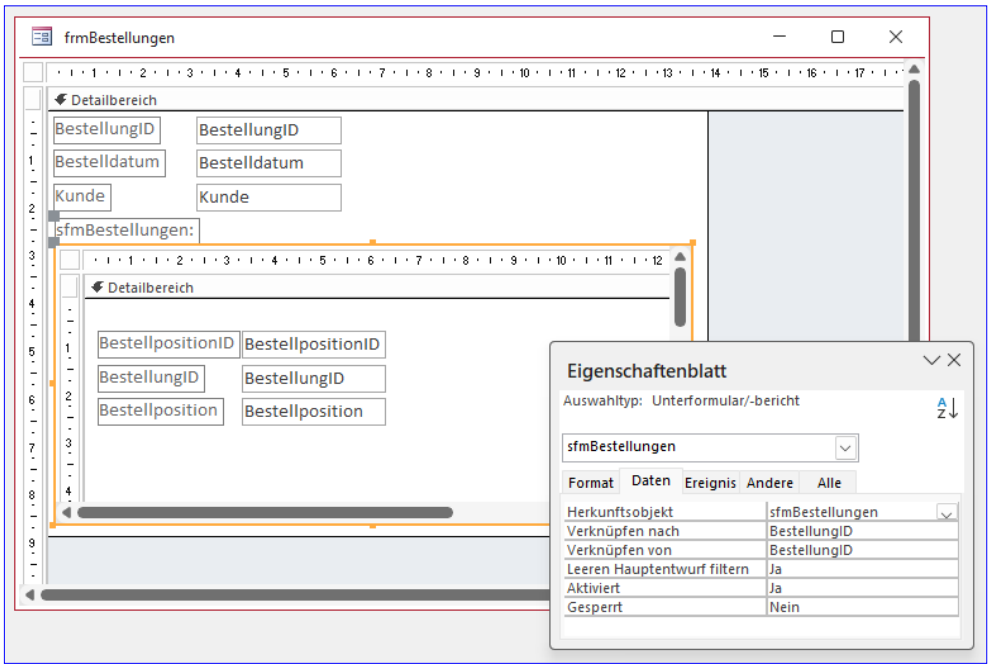

Bild 3: Bestellungen und Bestelldetails in Haupt- und Unterformular

Das decken wir normalerweise ab, indem wir jeweils eine Bestellung in einem Hauptformular abbilden und in einem Unterformular die Bestellpositionen. Wenn der Name des Fremdschlüsselfelds in der Tabelle des Unterformulars mit dem des Primärschlüsselfelds in der Datensatzquelle des Hauptformulars übereinstimmt, erkennt Access dies sogar beim Hinzufügen des Unterformulars und trägt dies korrekt in die Eigenschaften **Verknüpfen nach** und **Verknüpfen von** des Unterformular-Steuerelements ein (siehe Bild 3).

Damit ordnet das Formular neue Bestellpositionen, die wir in das Unterformular eintragen, automatisch dem Bestelldatensatz im Hauptformular zu, weil das Fremdschlüsselfeld **BestellungID** direkt mit dem entsprechenden Primärschlüsselwert aus dem Hauptformular gefüllt wird.

Bei 99 % der ausgewerteten Anwendungen funktionierte dies nicht!

Auch wenn dies eine scheinbare Sicherheit vorgaukelt: In unseren Untersuchungen von Kundendatenbanken haben wir bei fast allen Anwendungen herausgefunden, dass dies nicht zuverlässig funktioniert. Die Gründe dafür sind nicht genauer bekannt, weil die betroffenen Datensätze oft vor langer Zeit angelegt wurden. Aber: Wir haben nahezu überall Probleme aufgedeckt, die zeigen, dass Benutzer erfinderisch sind und Wege finden, um die Zuordnung von Bestellungen und Bestellpositionen zu unterminieren.

Dies zeigte sich in den folgenden Ergebnissen:

- Entweder haben wir Datensätze in der Tabelle **tblBestellpositionen** gefunden, die einen Fremdschlüsselwert aufweisen, der in der Tabelle **tblBestellungen** nicht mehr vorhanden war (oder nie gewesen ist). Das heißt, dass entweder Bestellungen gelöscht wurden, ohne dass die entsprechenden Einträge in **tblBestellpositionen** auch entfernt wurden (wahrscheinlichere Variante), oder die Benutzer es geschafft haben, Fremdschlüsselwerte einzutragen, für die es kein Pendant im Primärschlüsselfeld der Tabelle **tblBestellungen** gab.

- Oder wir haben Einträge in der Tabelle **tblBestellpositionen** gefunden, die im Fremdschlüsselfeld **BestellungID** den Wert **NULL** enthielten, also leer waren.

Beides ist ungünstig, denn es wurden scheinbar einmal Bestellungen plus Bestellpositionen eingetragen, die dann nicht zur Ausführung kamen und somit potenziell den Umsatz vermindert haben.

Dies kann passieren, wenn entweder das Formular nicht sauber programmiert und mit entsprechenden Validierungen ausgestattet wurde oder wenn die Benutzer Wege gefunden haben, die Daten direkt in den zugrunde liegenden Tabellen zu manipulieren und entweder Bestelldatensätze gelöscht oder Bestellpositionen verändert haben.

Die Lösung ist also zum Beispiel, die Anwendung so sicher zu machen, dass derartige Manipulationen nicht mehr möglich sind. Dazu muss das Formular vor Fehleingaben geschützt werden und/oder man muss verhindern, dass die Benutzer direkten Zugriff auf die Tabellen erhalten und so die Daten direkt in den Tabellen manipulieren.

Dies war in den meisten Anwendungen nicht der Fall, auch wenn die Entwickler uns glaubhaft machen wollten, dass sie alle notwendigen Maßnahmen getroffen hätten.

Es gibt jedoch noch eine einfachere Möglichkeit, um sicherzustellen, dass es keine Datensätze in einer Tabelle wie **tblBestellpositionen** gibt, die keiner Bestellung zugeordnet sind. Dabei handelt es sich um das Anlegen einer Beziehung zwischen den Tabellen (falls bisher nicht vorhanden) und die Festlegung von referenzieller Integrität für diese Beziehung.

Funktion der referenziellen Integrität

Wenn wir referenzielle Integrität definieren, aktivieren wir zwei wichtige Funktionen für die Beziehung. Nehmen wir als Beispiel wieder die Beziehung zwischen der Tabelle **tblBestellungen** und **tblBestellpositionen**. Wir fügen mit der referenziellen Integrität eine Restriktion hinzu, welche die Werte im Fremdschlüsselfeld **BestellungID** in der Tabelle **tblBestellpositionen** auf die folgenden Werte einschränkt:

- Es sind alle Werte erlaubt, die im Primärschlüsselfeld der verknüpften Tabelle vorhanden sind.
- Und zusätzlich kann, wenn wir dies nicht anderweitig unterbinden, der Wert **NULL** vorliegen.

Letzteres können wir verhindern, indem wir die Eigenschaft **Eingabe erforderlich** für das Fremdschlüsselfeld auf **Ja** einstellen. Das ist sinnvoll, um auch diese Lücke zu schließen. Es gibt jedoch Fälle abseits von Bestellungen und Bestellpositionen, wo man vielleicht erst die Datensätze in der Tabelle mit dem Fremdschlüsselfeld anlegt und diese erst später zuordnen möchte – dann kann man **NULL**-Werte zulassen.

Hier sollte man jedoch regelmäßig prüfen, ob sich keine nicht zugeordneten Datensätze in dieser Tabelle befinden.

Wenn wir keine referenzielle Integrität festlegen, können wir eine Bestellposition zur Tabelle **tblBestellpositionen** hinzufügen, die einen Wert im Feld **BestellungID** enthält, der nicht in der Tabelle **tblBestellungen** enthalten ist (siehe Bild 4).

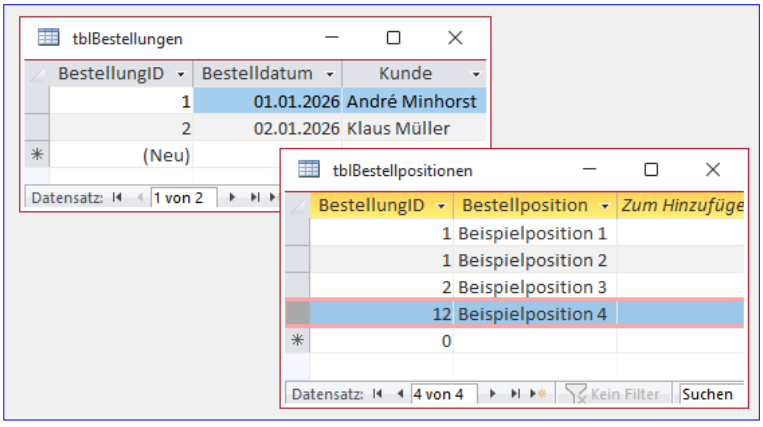

Bild 4: Bestellposition ohne passende Bestellung

Wenn keine referenzielle Integrität festgelegt werden kann

Wir versuchen nun, eine Beziehung mit referenzieller Integrität für die Tabellen anzulegen. Dazu ziehen wir im Beziehungen-Fenster das Feld **BestellungID** der Tabelle **tblBestellpositionen** auf die Tabelle **tblBestellungen**. Es erscheint der Dialog **Beziehungen bearbeiten**, wo wir die Option **Mit referenzieller Integrität** aktivieren (siehe Bild 5).

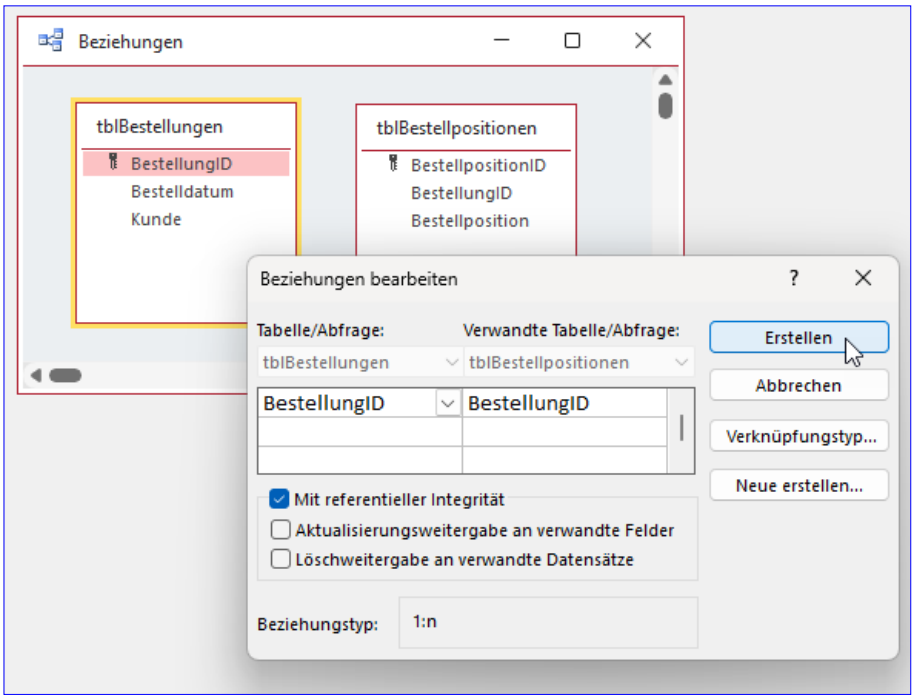

Bild 5: Versuch, referenzielle Integrität zu definieren

Wenn wir jetzt auf **Erstellen** klicken, erhalten wir die Fehlermeldung aus Bild 6. Der Grund ist offensichtlich: Wir haben in der Tabelle **tblBestellpositionen** Werte im Feld **BestellungID**, die in der Tabelle **tblBestellungen** nicht vorhanden sind.

Dies ist der einfachste Test, um zu prüfen, ob alle Datensätze aus **tblBestellpositionen** korrekt der Tabelle **tblBestellungen** zugeordnet sind.

Etwas schwieriger wird es, im Anschluss herauszufinden, welche der Datensätze der Tabelle **tblBestellpositionen** das Definieren referenzieller Integrität verhindern – dazu weiter unten mehr.

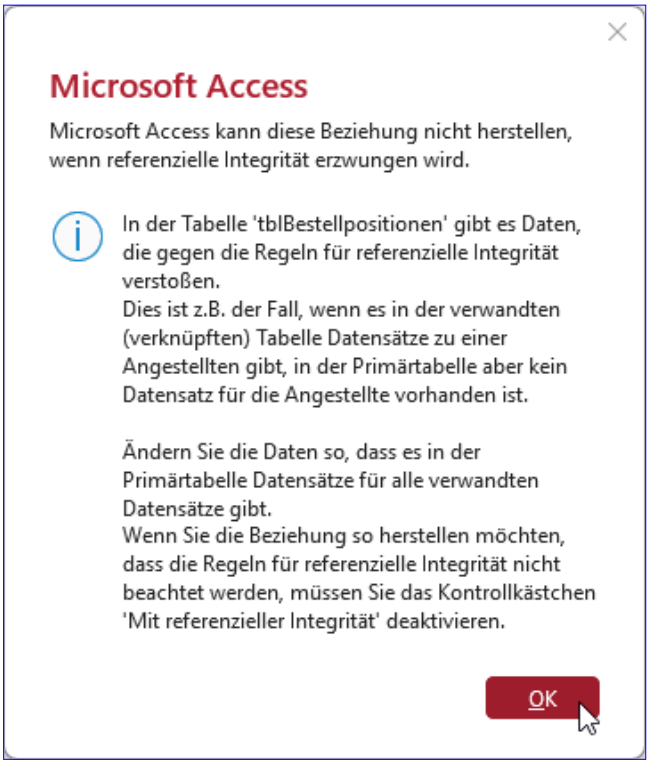

Bild 6: Fehlermeldung beim Versuch, referenzielle Integrität zu definieren

Erfolgreiches Festlegen referenzieller Integrität

In diesem Fall löschen wir einfach den Datensatz, der auf die nicht vorhandene Bestellung mit dem Wert **12** im Feld **BestellungID** verweist. Danach können wir die referenzielle Integrität für diese Tabelle aktivieren.

Das Ergebnis sehen wir in Bild 7. Dass es sich um eine Beziehung mit referenzieller Integrität handelt, erkennen wir am Unendlich-Symbol auf der einen und der Pfeilspitze auf der anderen Seite.

Reflexive 1:n-Beziehung zu m:n-Beziehung

Manchmal legt man eine 1:n-Beziehung zwischen zwei Tabellen an, um später festzustellen, dass eine m:n-Beziehung doch die funktionalere Variante ist. Ein Beispiel sind reflexive Beziehungen, mit denen man etwa Vater-Kind-Beziehungen abbildet oder Partner-Beziehungen. Andere Beispiele sind solche wie zwischen Produkten und Kategorien – man dachte zunächst, dass es reicht, wenn man jedes Produkt nur einer Kategorie zuordnen kann, aber man dann erkennt, dass es für verschiedene Anwendungsfälle doch günstiger wäre, wenn man ein Produkt mehr als einer Kategorie zuordnen kann. Ähnliche Fälle sind Mitarbeiter und Funktionen oder Abteilungen. In diesem Beitrag schauen wir uns das Beispiel eines Kunden an, der in seinem Sportverein partnerschaftliche Beziehungen über ein Fremdschlüsselfeld der Tabelle tblMitglieder auf diese selbst abgebildet hat. Hier gab es mehrere Gründe, um diese Beziehung in eine m:n-Beziehung umzuwandeln. Welche das sind und wie wir die Umwandlung durchgeführt haben, lesen Sie in diesem Beitrag.

Das Abbilden von reflexiven Beziehungen über eine 1:n-Beziehung ist an sich nicht falsch. Im Beispiel geht es um einen Sportverein, dessen Mitglieder mit einer Access-Datenbank verwaltet werden. In der Tabelle **tblMitglieder** gibt es zwei Felder, um die partnerschaftliche Beziehung zwischen zwei Mitgliedern abzubilden. Das erste heißt **PartnerID** und dient dazu, die ID eines anderen Datensatzes dieser Tabelle zu referenzieren und damit anzugeben, welches andere Mitglied mit diesem Mitglied verbunden ist. Zusätzlich gibt es ein **Ja/Nein**-Feld namens **Ehepartner**, das angibt, ob es sich um eine amtlich dokumentierte Beziehung handelt, sprich um eine eheliche Beziehung (**Ja**) oder um eine Beziehung ohne Trauschein (**Nein**). Der Entwurf der Tabelle sieht in abgespeckter Form wie in Bild 1 aus.

Das erste Problem, das sich daraus ergibt, ist die wechselseitige Abhängigkeit. Wenn wir für Mitglied A das Mitglied B als Partner festgelegt haben, müssen wir für Mitglied B auch Mitglied A als Partner hinterlegen – und für beide muss das Feld **Ehepartner** den gleichen Wert enthalten. Anderenfalls erhalten wir einen Fall von inkonsistenten Daten.

Diese Inkonsistenz kann nicht nur den Fall enthalten, dass die Beziehung nur in einer Richtung dokumentiert ist, etwa von Mitglied A zu Mitglied B. Es kann auch passieren, dass nicht nur Mitglied B als Partner von Mit-

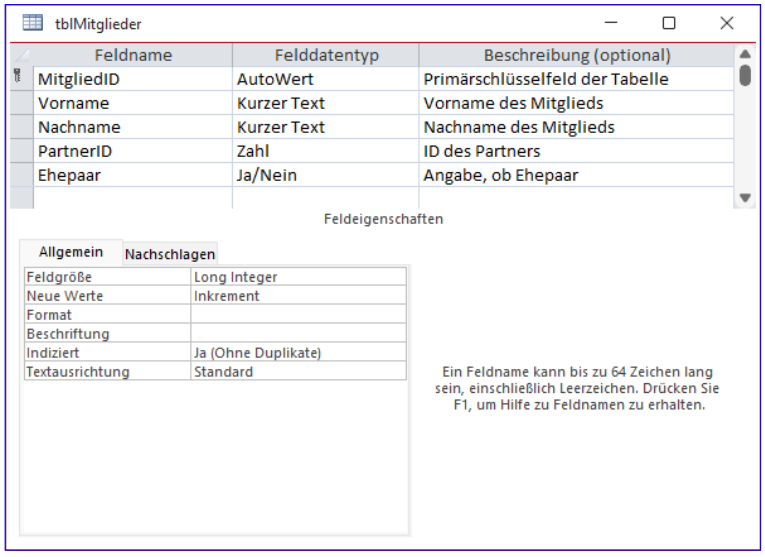

Bild 1: Entwurf der Tabelle **tblMitglieder**

glied A angegeben wird, sondern auch noch Mitglied C als Partner von Mitglied B. Es konnte auch vorkommen, dass ein Mitglied als Ehepartner von mehreren anderen Mitgliedern angegeben wird.

Das zweite Problem bei der Tabelle **tblMitglieder** bei diesem Kunden war, dass bereits sehr viele andere Tabellen auf die Tabelle **tblMitglieder** verwiesen hatten und somit die Grenze von maximal 32 Beziehungen erreicht war.

Diese Grenze gilt für Beziehungen, die über ein Fremdschlüsselfeld in dieser Tabelle hergestellt werden, wobei wir hier sehr viele Lookup-Tabellen vorgefunden haben.

Die Beziehung war wie in Bild 2 aufgebaut, wobei das Fremdschlüsselfeld **PartnerID** auf das Feld **MitgliedID** der gleichen Tabelle verweist.

Mehrere Partner für das gleiche Mitglied verhindern

Die Möglichkeit, dass ein Mitglied für mehr als ein anderes Mitglied als Partner ausgewählt wird, konnten wir durch das Setzen eines eindeutigen Schlüssels für das Fremdschlüsselfeld **PartnerID** erreichen.

Dadurch, dass nun jede **MitgliedID** nur einmal in das Feld **PartnerID** eingetragen werden konnte, stellen wir sicher, dass es nur monogame Beziehungen gibt. Hier lassen wir außen vor, dass es Kulturen gibt, in denen dies möglich ist.

Auswahl einer Beziehung mit sich selbst verhindern

Eine weitere mögliche Fehleingabe ist die Auswahl des Mitglieds selbst als Partner.

Dies können wir verhindern, indem wir im Formular, das wir gleich beschreiben, nur die anderen Datensätze der Tabelle **tblMitglieder** anzeigen.

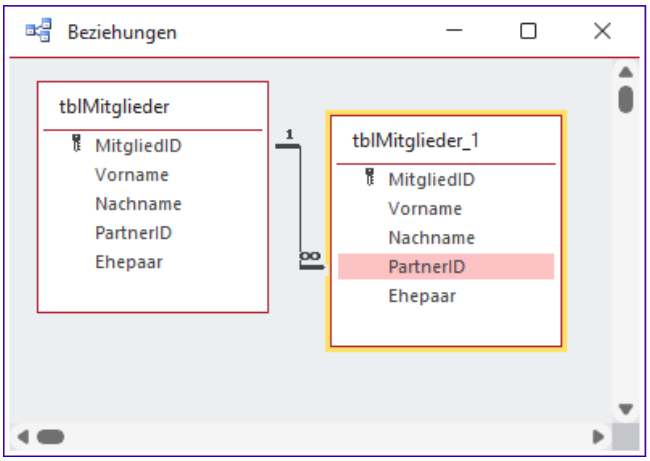

Bild 2: Reflexive Beziehung der Mitglieder

Nur eindeutige Partnerschaften erlauben

Wenn wir sicherstellen wollen, dass die in einer Partnerschaft befindlichen Mitglieder immer wechselseitig miteinander verbunden werden, müssen wir das über die Anwendungslogik realisieren – allein über den Tabellenentwurf können wir das mit einem Fremdschlüsselfeld, das die gleiche Tabelle referenziert, nicht erreichen.

Wir müssen also im Formular zum Verwalten der Mitglieder eine VBA-Funktion einbauen, die beim Auswählen des Partners eines Mitglieds automatisch auch das Feld **PartnerID** des Partners einstellt – und die auch den gleichen Wert im **Ja/Nein**-Feld **Ehepaar** für die verknüpften Datensätze festlegt.

Im Entwurf der Tabelle **tblMitglieder** haben wir für das Fremdschlüsselfeld **PartnerID** ein Nachschlagefeld

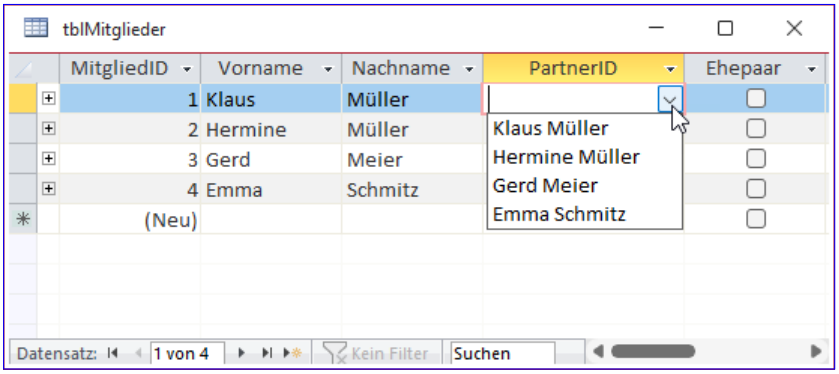

Bild 3: Auswahlfeld für den Partner

definiert, damit man den Partner einfach auswählen kann (siehe Bild 3).

Formular zum Verwalten der Mitglieder und Partnerschaften

Um die Mitglieder zu verwalten und die Partner zuordnen zu können, haben wir ein Formular wie in Bild 4 vorgefunden.

Für das Steuerelement **cboPartnerID** haben wir zunächst die folgende Datensatzherkunft eingestellt, um aus allen Mitgliedern auswählen zu können:

```
SELECT MitgliedID, Vorname & ' ' & Nachname
FROM tblMitglieder
```

Damit beim Anzeigen eines Datensatzes nicht das aktuelle Mitglied selbst erscheint und auch nicht solche Mitglieder, die bereits einer Partnerschaft zugeordnet sind, haben wir für das Ereignis **Beim Anzeigen** die folgende Prozedur hinterlegt:

```
Private Sub Form_Current()
    Me.cboPartnerID.RowSource = "SELECT MitgliedID, " _
        & "Vorname & ' ' & Nachname FROM tblMitglieder " _
        & "WHERE NOT (MitgliedID = " & Me.MitgliedID & ") " _
        & " AND (MitgliedID NOT IN (" _
        & "    SELECT PartnerID FROM tblMitglieder " _
        & "    WHERE PartnerID IS NOT NULL))"
End Sub
```

Für das Mitglied mit dem Wert **1** im Feld **MitgliedID** erhalten wir so die folgende Abfrage:

```
SELECT MitgliedID, Vorname & ' ' & Nachname
FROM tblMitglieder
WHERE NOT (MitgliedID = 1)
AND (MitgliedID NOT IN (
    SELECT PartnerID FROM tblMitglieder
    WHERE PartnerID IS NOT NULL)
)
```

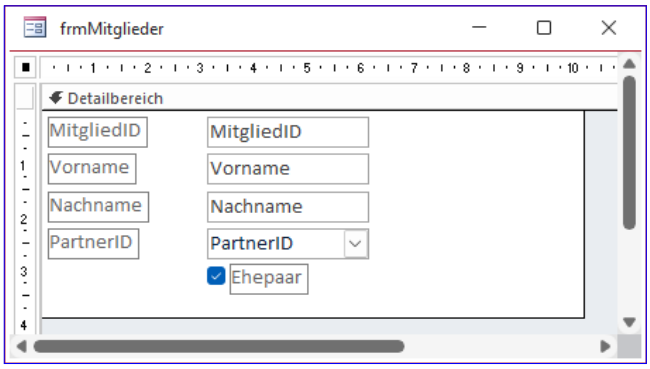

Bild 4: Formular zur Mitgliederverwaltung

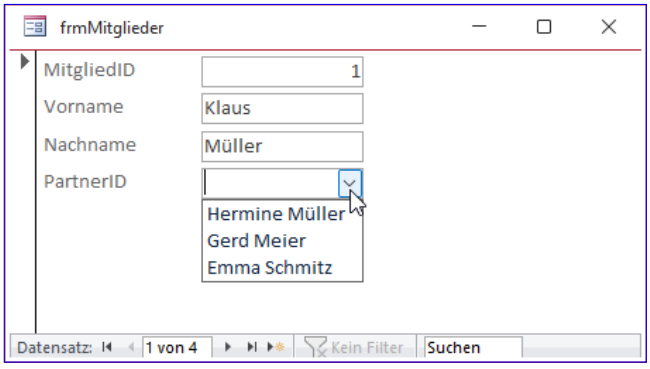

Bild 5: Formular zur Mitgliederverwaltung in der Formularansicht

Damit erscheinen, wenn wir noch keinem Mitglied einen Partner zugewiesen haben, für jedes Mitglied nur die aktuell verfügbaren Partner im Feld **cboPartnerID** – siehe Bild 5.

Partnerschaft für den Partner einstellen

Wenn wir nun für das Mitglied mit dem Wert **1** im Feld **MitgliedID** einen Partner auswählen, etwa den mit dem Wert **2**, soll für das Mitglied **2** das Mitglied **1** als Partner eingestellt werden.

Dazu hinterlegen wir für das Ereignis **Nach Aktualisierung** des Kombinationsfeldes **cboPartnerID** die folgende Prozedur:

```
Private Sub cboPartnerID_AfterUpdate()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "UPDATE tblMitglieder SET PartnerID = " _
```

```
        & Me.MitgliedID & " WHERE MitgliedID = " _
        & Me.cboPartnerID, dbFailOnError
End Sub
```

Die so ausgeführte Aktionsabfrage lautet beispielsweise:

```
UPDATE tblMitglieder SET PartnerID = 1 WHERE MitgliedID = 2
```

Damit erhalten wir das gewünschte Ergebnis in der zugrunde liegenden Tabelle, aber wenn wir zum Datensatz des gewählten Partners wechseln und dann zurück, zeigt das Kombinationsfeld nun für beide gar keinen Wert mehr an. Das liegt daran, dass wir die bereits vergebenen Partner vollständig ausschließen und diese somit auch für den ausgewählten Eintrag nicht mehr angezeigt werden. Das Kombinationsfeld enthält zwar den korrekten Eintrag, was wir im Direktbereich mit folgender Anweisung ermitteln können:

```
? Screen.ActiveControl.Value
 2
```

Aber da der Eintrag nicht in der Datensatzherkunft des Kombinationsfeldes enthalten ist, erscheint der Anzeigewert nicht.

Wir müssen die Datensatzherkunft also nochmals anpassen, indem wir mit der **OR**-Klausel am Ende den gewählten Partner wieder mit in die Auswahl hineinnehmen – hier der besseren Lesbarkeit halber direkt mit den Werten **1** für **MitgliedID** und **2** für **PartnerID**:

```
SELECT MitgliedID, Vorname & ' ' & Nachname
FROM tblMitglieder
WHERE NOT (MitgliedID = 1) AND (MitgliedID NOT IN (
    SELECT PartnerID FROM tblMitglieder
    WHERE PartnerID IS NOT NULL)
OR MitgliedID = 2)
```

Damit sehen wir nun den ausgewählten Partner sowohl für den ersten als auch für den zweiten Datensatz. In der **OR**-Bedingung sorgen wir außerdem mit der **Nz**-Funktion dafür, dass im Falle des Wertes **NULL** im Feld **PartnerID** der Wert **0** verwendet wird – sonst erhalten wir beim Wechsel von einem Datensatz ohne **PartnerID** zu einem anderen Datensatz einen Fehler.

Schließlich müssen wir auch noch dafür sorgen, dass der Wert für das Feld **Ehepaar** für das als Partner angegebene Mitglied so einstellen wie für das Mitglied selbst. Dazu fügen wir eine Prozedur für das Ereignis **Nach Aktualisierung** des Kontrollkästchens **chkEhepaar** hinzu:

```
Private Sub chkEhepaar_AfterUpdate()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "UPDATE tblMitglieder SET Ehepaar = " _
        & IIf(Me.chkEhepaar = True, "True", "False") _
        & " WHERE MitgliedID = " & Me.cboPartnerID, _
        dbFailOnError
End Sub
```

Warum auf m:n-Beziehung umstellen?

Eingangs haben wir erwähnt, dass sich viele der hier beschriebenen Probleme beheben lassen, wenn wir die reflexive Beziehung über die 1:n-Verknüpfung aufheben, die Felder **PartnerID** und **Ehepaar** entfernen und beides in eine m:n-Beziehung auslagern.

Wir schauen uns erst einmal an, wie die Verknüpfungstabelle aussieht und wie wir diese mit der Tabelle **tblMitglieder** verknüpfen. Damit das Beispiel mit der reflexiven 1:n-Beziehung erhalten bleibt, erstellen wir in der Beispieldatenbank eine Kopie der Tabelle **tblMitglieder** (siehe Bild 6) unter dem Namen **tblMitgliederMN**. Außerdem fügen wir die Tabelle **tblPartnerschaften** hinzu, die folgende Felder enthält:

- **PartnerschaftID**: Primärschlüsselfeld der Tabelle
- **PartnerA**: Erster Partner, Fremdschlüsselfeld zur Tabelle **tblMitglieder**, eindeutiger Index, Eingabe erforderlich

- **PartnerB**: Zweiter Partner, Fremdschlüsselfeld zur Tabelle **tblMitglieder**, eindeutiger Index, Eingabe erforderlich

- **Ehepaar**: **Ja/Nein**-Feld, das angibt, ob es sich um ein Ehepaar handelt

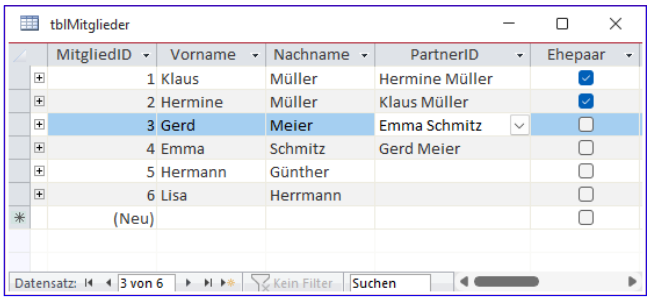

Bild 6: Aktueller Zustand der Daten in der Tabelle **tblMitglieder**

Dadurch, dass wir einen eindeutigen Index auf die Felder **PartnerA** und **PartnerB** legen, stellen wir sicher, dass jedes Mitglied nur einmal als PartnerA oder PartnerB angegeben werden kann.

Wir haben allerdings noch eine kleine Lücke: Wir können die Kombination aus Mitglied **1** und Mitglied **2** immer noch zweimal angeben – einmal mit dem Wert **1** im Feld **PartnerA** und **2** in **PartnerB** und umgekehrt. Außerdem können wir theoretisch auch Mitglied **1** sowohl in das Feld **PartnerA** als auch in **PartnerB** eingeben.

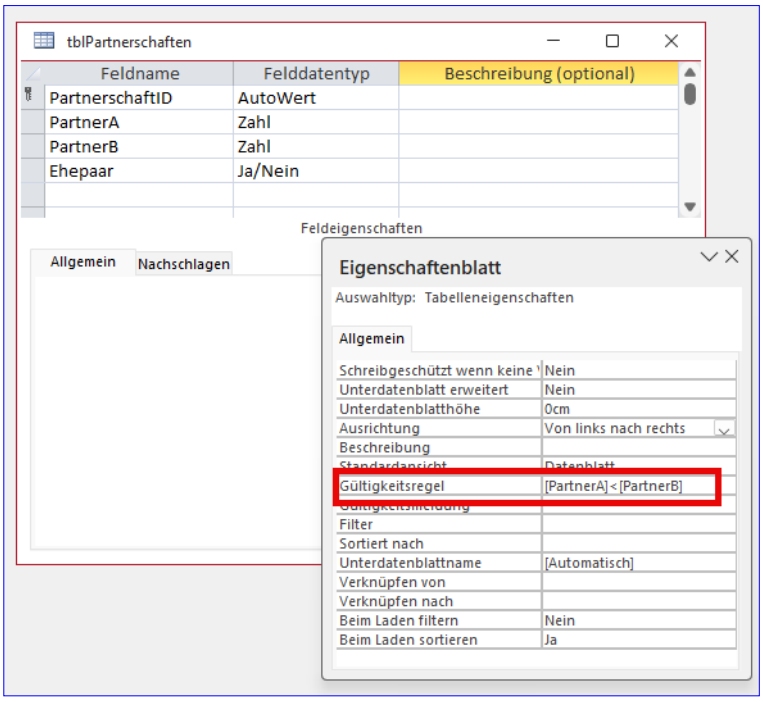

Bild 7: Tabelle **tblPartnerschaften**

Dies verhindern wir, indem wir für die Tabelle eine Gültigkeitsregel anlegen, die wir wie in Bild 7 definieren.

Wenn wir nun eine Partnerschaft eingeben, bei welcher der Wert von **PartnerA** kleiner oder gleich der von **PartnerB** ist, erhalten wir die Meldung aus Bild 8.

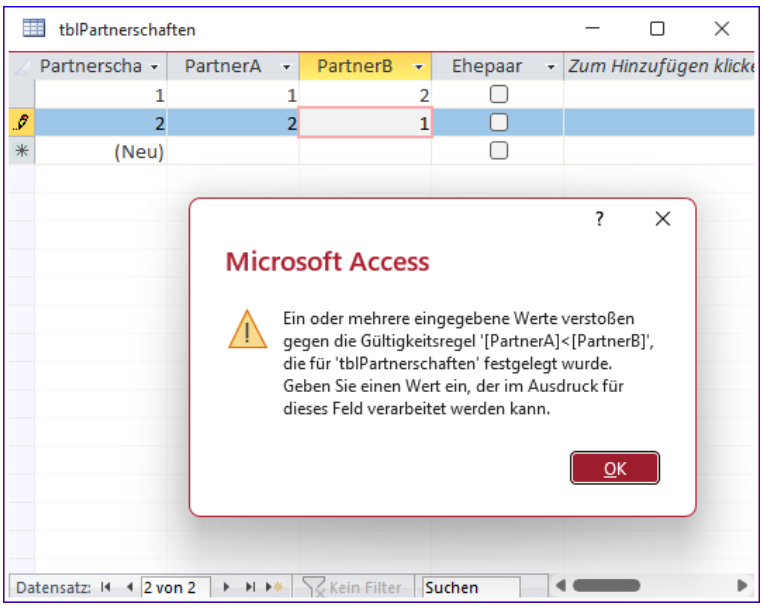

Bild 8: Meldung beim Eingeben ungültiger Daten

Diese Regel ist nur für den Fall vorgesehen, dass jemand versucht, Daten direkt über die Tabelle einzugeben. Das eigentliche Anlegen erledigen wir ohnehin über das Formular.

Beziehung zwischen Mitgliedern und Partnerschaften

Damit nur Werte aus dem Primärschlüsselfeld der Tabelle **tblMitglieder** eingegeben werden können, fügen wir dem Datenmodell die beiden Beziehungen aus Bild 9 hinzu.

Wichtig ist hier die Richtung, in der wir die Beziehungspfeile ziehen. Da die beiden Felder **PartnerA** und **PartnerB** mit einem eindeutigen Index versehen wurden, müssen wir den Pfeil von **tblMitglieder** auf **tblPartnerschaften** ziehen, sodass die Mitgliedertabelle unter **Tabelle/Abfrage** und die Partnerschaftstabelle unter **Verwandte Tabelle/Abfrage** angegeben wird.

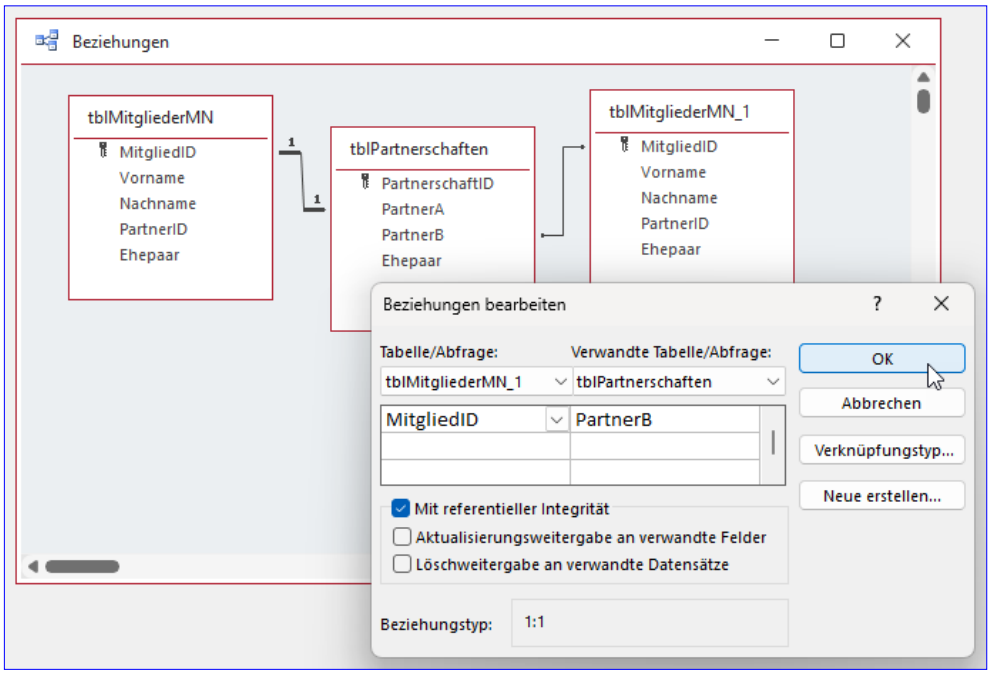

Bild 9: Beziehungen für die Verwaltung von Partnerschaften per m:n-Beziehung

Entsprechend könnten wir keine referenzielle Integrität für die Beziehungen festlegen, da dann das Feld **MitgliedID** der Tabelle **tblMitglieder** als Fremdschlüsselfeld betrachtet würde und es darin Werte gibt, die zurzeit noch nicht im Feld **PartnerA** oder **PartnerB** der Tabelle **tblPartnerschaften** enthalten sind.

Daten migrieren

Es gibt bereits eingetragene Partnerschaften in der Tabelle **tblMitarbeiterMN**, die wir nun über die Verknüpfungstabelle **tblPartnerschaften** abbilden wollen.

Das ist anspruchsvoll, weil wir die Partnerschaften so eintragen müssen, dass der Wert im Feld **PartnerA** immer kleiner als der in **PartnerB** ist. Wir könnten starten, indem wir die Partnerschaften aufsteigend nach der ID des Mitglieds sortieren und zusätzlich die **PartnerID** und den **Ehepaar**-Status mit folgender Abfrage ausgeben lassen:

```
SELECT
    tblMitgliederMN.MitgliedID,
    tblMitgliederMN.PartnerID,
    tblMitgliederMN.Ehepaar
FROM
    tblMitgliederMN
WHERE
    (((tblMitgliederMN.PartnerID) IS NOT NULL))
ORDER BY
    tblMitgliederMN.MitgliedID;
```

Dies liefert jede Partnerschaft in doppelter Ausführung, jeweils einmal mit jeder Mitglieds-ID im Feld **MitgliedID** und einmal im Feld **PartnerID** (siehe Bild 10).

Wenn wir hier allerdings den Ausdruck, den wir als Gültigkeitsregel für die Tabelle **tblPartnerschaften** verwendet haben, als Kriterium einsetzen, erhalten wir jede Partnerschaft nur noch einmal:

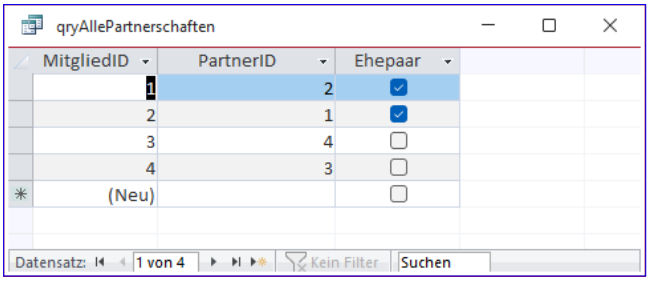

Bild 10: Ausgabe aller Partnerschaften laut **tblMitgliederMN**

```
SELECT
    tblMitgliederMN.MitgliedID,
    tblMitgliederMN.PartnerID,
    tblMitgliederMN.Ehepaar
FROM
    tblMitgliederMN
WHERE
    (
        ((tblMitgliederMN.MitgliedID) < [PartnerID])
        AND ((tblMitgliederMN.PartnerID) IS NOT NULL)
    )
ORDER BY
    tblMitgliederMN.MitgliedID;
```

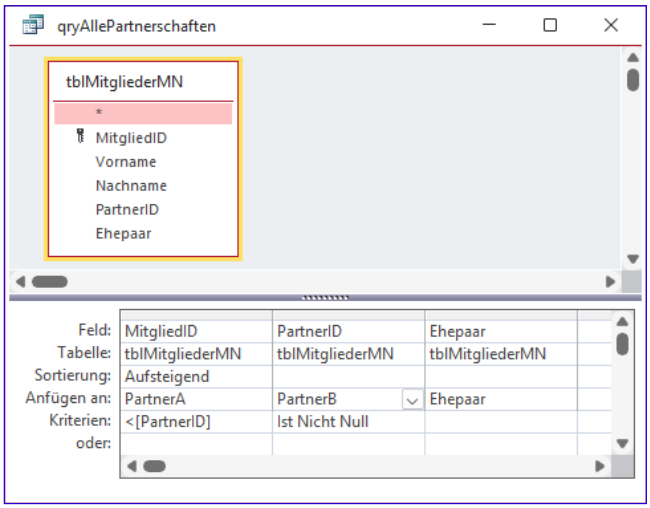

Bild 11: Anfügeabfrage für die Partnerschaftstabelle

Für diese Abfrage aktivieren wir nun den Abfragetyp **Anfügen** und geben als Zieltabelle die Tabelle **tblPartnerschaften** an. Im Entwurf weisen wir das Feld **MitgliedID** dem Feld **PartnerA** und das Feld **PartnerID** dem Feld **PartnerB** hinzu. Das Feld **Ehepaar** weisen wir dem gleichnamigen Feld der Tabelle **tblPartnerschaften** zu (siehe Bild 11).

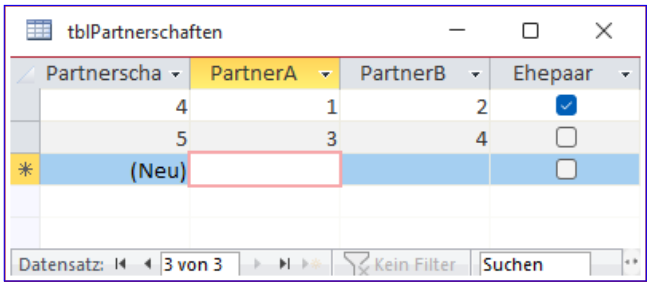

Bild 12: Ergebnis der Anfügeabfrage

Führen wir diese Abfrage aus, erhalten wir das Ergebnis aus Bild 12. Durch das Kriterium landen auch nur gültige Datensätze in dieser Abfrage – sollte die Tabelle **tblMitgliederMN** zuvor Datensätze enthalten haben, bei denen **PartnerA** gleich **PartnerB** wäre, oder es gäbe inkonsistente Daten, würden diese nicht angelegt werden.

Es ist allerdings nicht auszuschließen, dass für Mitglied **1** eine Partnerschaft mit Mitglied **2** angelegt war und für Mitglied **2** eine Partnerschaft mit Mitglied **3**. Dies können wir über das Datenmodell in der aktuellen Form nicht ausschließen und müssten es letztlich über die Anwendungslogik steuern. Allerdings bietet Access auch noch die Datenmakros an. Wie wir damit auch die letzte mögliche Fehleingabe verhindern, zeigen wir in den folgenden Abschnitten.

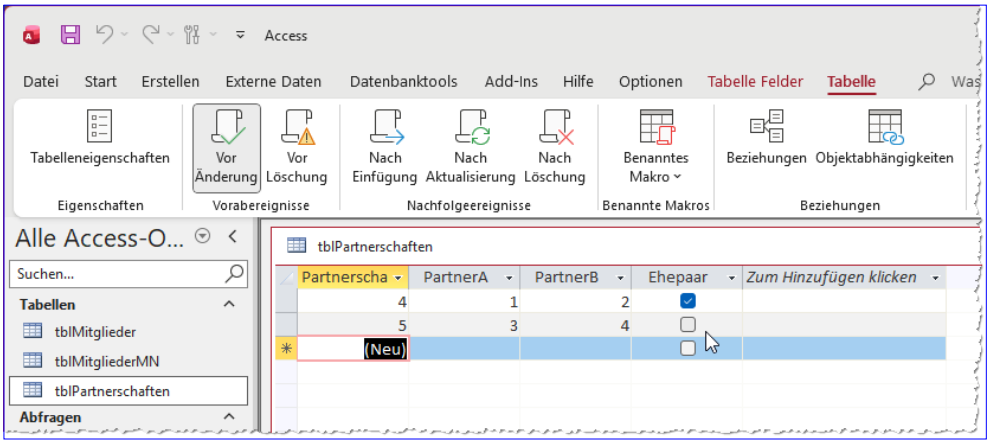

Bild 13: Datenmakro für das Ereignis **Vor Änderung** anlegen

Erledigt-Status in Haupt- und Unterformular synchron

Ein Kunde hatte neulich die Anforderung, dass er Produktionsaufträge mit den zu produzierenden Teilen im Haupt- und Unterformular abbilden wollte. An sich kein Problem, wenn man Haupt- und Unterformular entsprechend verknüpft. Er wünschte sich jedoch sowohl in der Tabelle der Produktionsaufträge als auch in der für die Teile jeweils ein Kontrollkästchen, das den Status abbildet. Wenn der vollständige Auftrag erledigt ist, soll dieser samt Teileliste einen Haken erhalten. Ist der Auftrag noch offen, sind alle Kontrollkästchen leer. Aber wenn nicht alle Teile fertig produziert sind, sollte dies im Produktionsauftrag auf eine spezielle Art gekennzeichnet werden. Er hat dabei den Dreifachstatus des Kontrollkästchens entdeckt und wünschte sich, dass das Kontrollkästchen in diesem Fall für den Produktionsauftrag den dritten Status anzeigt – in aktuellen Access-Versionen ein gefülltes Kontrollkästchen mit einem Minus-Zeichen. Wie das gelingt und wie wir die Zustände von Produktionsauftrag und Teilen synchron halten, zeigen wir in diesem Beitrag.

Datenmodell

Zunächst stellen wir die beiden Tabellen zusammen, mit denen wir die Produktionsaufträge und die enthaltenen, zu fertigenden Teile verwalten.

Die erste Tabelle heißt **tblProduktionsauftraege** und soll die Felder aus Bild 1 enthalten. Neben dem Primärschlüsselfeld und dem Feld für die Bezeichnung des Produktionsauftrags finden wir das **Ja/Nein**-Feld **Erledigt**.

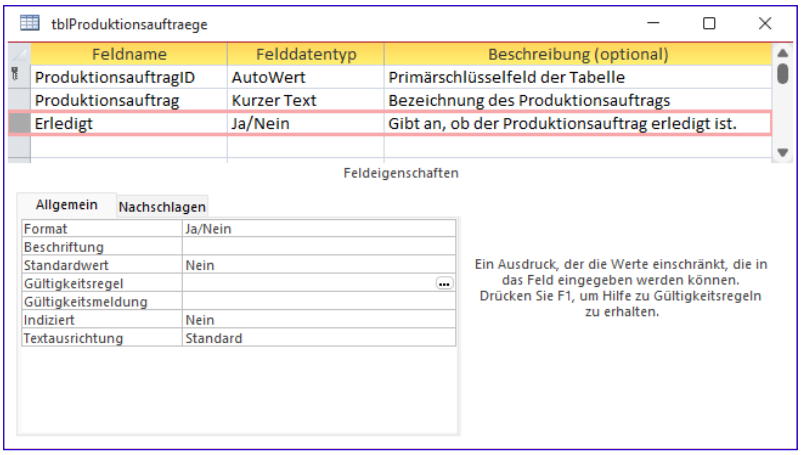

Bild 1: Tabelle für die Produktionsaufträge

Die zweite Tabelle namens **tblProduktionsteile** enthält ähnliche Felder, also auch das Feld **Erledigt** mit dem Datentyp **Ja/Nein**. Außerdem finden wir hier noch das Feld **ProduktionsauftragID**, mit dem wir das zu produzierende Teil dem jeweiligen Produktionsauftrag zuweisen können (siehe Bild 2).

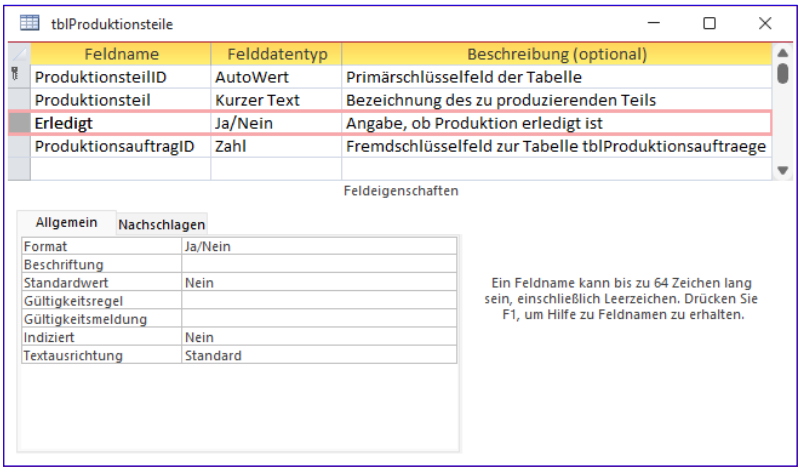

Bild 2: Tabelle für die Produktionsteile

Zwischen den beiden Tabellen stellen wir über das Fremdschlüsselfeld **Produktions-**

auftragID eine Beziehung her und definieren referenzielle Integrität zwischen den beiden Tabellen (siehe Bild 3).

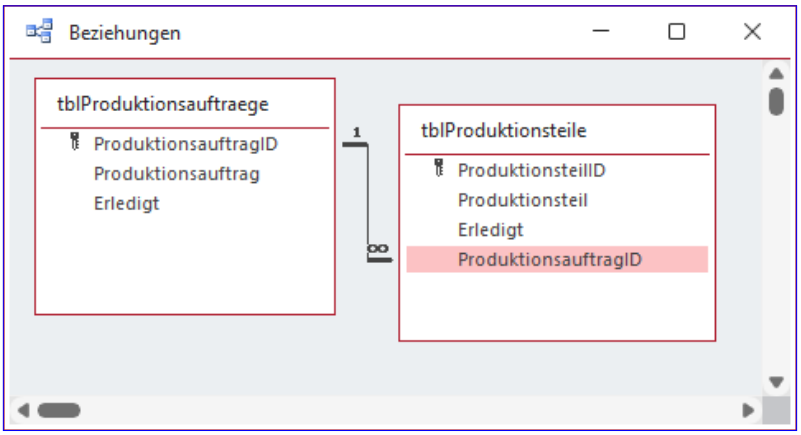

Bild 3: Beziehung zwischen den beiden Tabellen

Formulare für das Beispiel

Nun erstellen wir zunächst das Unterformular, das die Produktionsteile zu einem Produktionsauftrag anzeigen soll. Dazu fügen wir einem leeren Formular die Tabelle **tblProduktionsteile** als Datensatzquelle hinzu und ziehen alle Felder außer **ProduktionsauftragID** zur Detailansicht hinzu.

Die Eigenschaft **Standardansicht** legen wir auf den Wert **Datenblatt** fest. Anschließend speichern wir das Formular unter dem Namen **sfmProduktionsauftraege** und schließen es (siehe Bild 4).

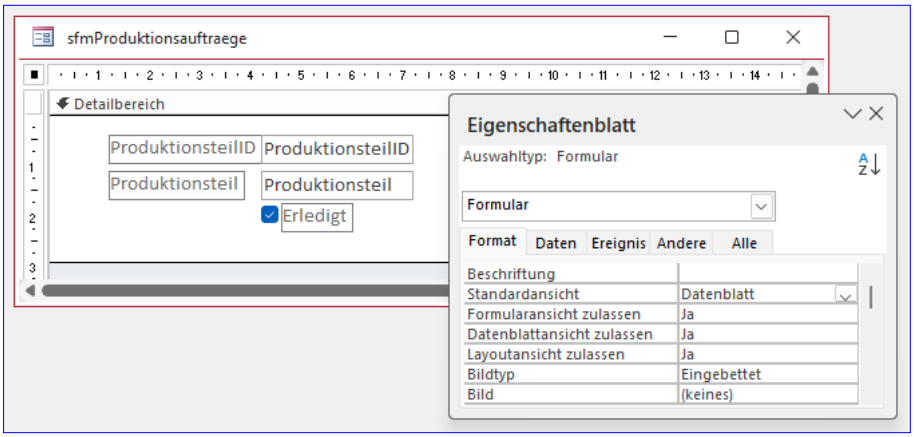

Bild 4: Entwurf des Unterformulars

Das Hauptformular erstellen wir auf ähnliche Weise, allerdings verwenden wir hier die Tabelle **tblProduktionsauftraege** als Datensatzquelle.

Zudem ziehen wir hier noch das soeben erstellte Unterformular **sfmProduktionsauftraege** in den Detailbereich, sodass das Ergebnis wie in Bild 5 aussieht.

Hier prüfen wir noch, ob die Eigenschaften **Verknüpfen von** und **Verknüpfen nach** für das Unterformular-Steuerelement korrekt eingestellt wurden.

Anschließend können wir in die Formularansicht wechseln und

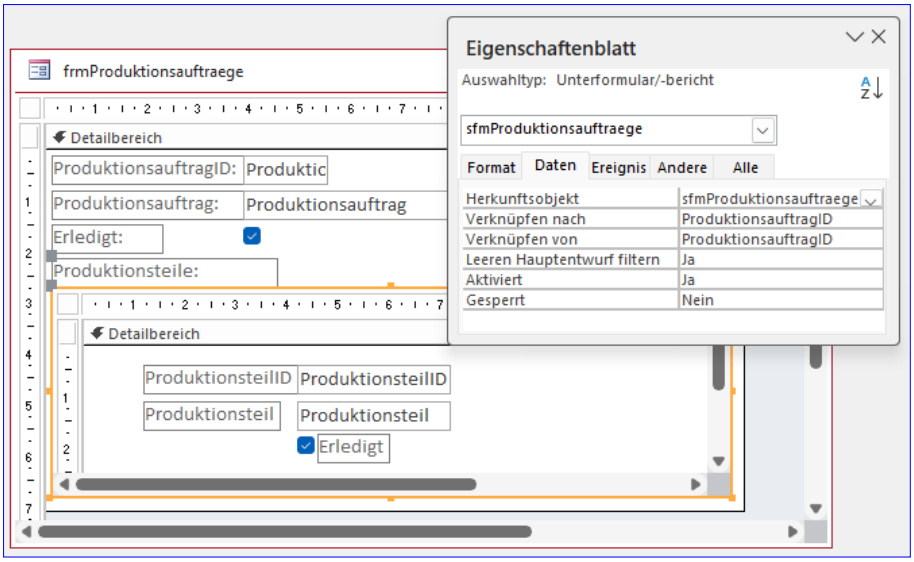

Bild 5: Entwurf des Hauptformulars

direkt einige Beispieldaten in Haupt- und Unterformular eingeben (siehe Bild 6).

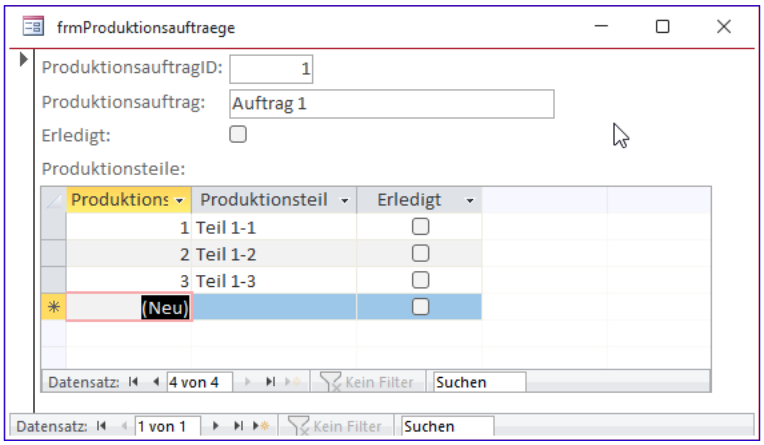

Bild 6: Eingabe von Beispieldaten

Erledigt-Status synchron halten

Damit kommen wir zur eigentlichen Aufgabe:

- Wenn der Benutzer nun das Feld **Erledigt** für den Produktionsauftrag im Hauptformular markiert, sollen auch alle Einträge im Unterformular markiert werden.

- Wenn der Benutzer die Markierung dieses Feldes aufhebt, sollen auch alle Einträge im Unterformular abgewählt werden.

- Setzt der Benutzer die Markierung für einen Eintrag im Unterformular oder hebt diese auf, soll geprüft werden, ob aktuell alle Einträge markiert sind oder auch nur einige oder keiner. Sind alle Einträge markiert, soll auch das Feld **Erledigt** im Hauptformular einen Haken erhalten. Wenn kein Haken markiert ist, soll **Erledigt** im Hauptformular auch abgewählt werden. Und schließlich fehlt noch der Fall, dass nur einige Einträge markiert sind: Dann soll das Feld **Erledigt** im Hauptformular den dritten Status eines Kontrollkästchens erhalten.

Wie das gelingt, beschreiben wir in den nächsten Abschnitten.

Kontrollkästchen umbenennen

Zuvor versehen wir die beiden Kontrollkästchen im Haupt- und Unterformular jedoch noch mit einem entsprechenden Präfix, hier **chk** für **Checkbox** (Kontrollkästchen). Beide heißen nun **chkErledigt**.

Produktionsauftrag als vollständig erledigt markieren

Wenn der Benutzer den Produktionsauftrag als erledigt kennzeichnet, sollen alle Produktionsteile zu diesem Auftrag auch als erledigt markiert werden. Um auf die Änderung des Zustandes des Kontrollkästchens **chkErledigt** im Hauptformular zu reagieren, hinterlegen wir eine Ereignisprozedur für das Ereignis **Nach Aktualisierung** (siehe Bild 7).

Dazu wählen wir hier den Eintrag **[Ereignisprozedur]** aus und klicken auf die Schaltfläche mit den drei Punkten.

Die jetzt im VBA-Editor erscheinende Ereignisprozedur füllen wir wie in Listing 1. Hier deklarieren wir ein **Database**-Objekt, das wir mit einem Verweis auf die aktuelle Datenbank füllen, sowie eine Variable namens **bolErle-**

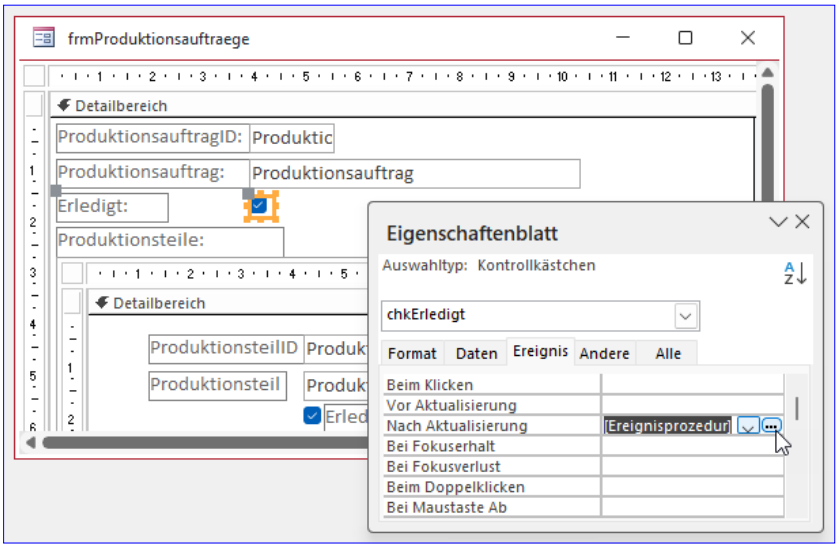

Bild 7: Anlegen einer Ereignisprozedur für das Kontrollkästchen

```
Private Sub chkErledigt_AfterUpdate()
    Dim db As DAO.Database
    Dim bolErledigt As Boolean

    Set db = CurrentDb

    bolErledigt = Me.chkErledigt
    db.Execute "UPDATE tblProduktionsteile SET Erledigt = " & CInt(bolErledigt) & " WHERE ProduktionsauftragID = " _
        & Me.ProduktionsauftragID, dbFailOnError
    Me.sfmProduktionsauftraege.Form.Requery
End Sub
```

Listing 1: Übertragen der Markierung aus dem Hauptformular in das Unterformular

digt, in die wir den aktuellen Zustand des Kontrollkästchens **chkErledigt** aus dem Hauptformular einlesen.

Dann führen wir eine Aktualisierungsabfrage aus, die für alle Datensätze der Tabelle **tblProduktionsteile**, die zum aktuellen Produktionsauftrag gehören, den Wert aus der Variablen **bolErledigt** einträgt.

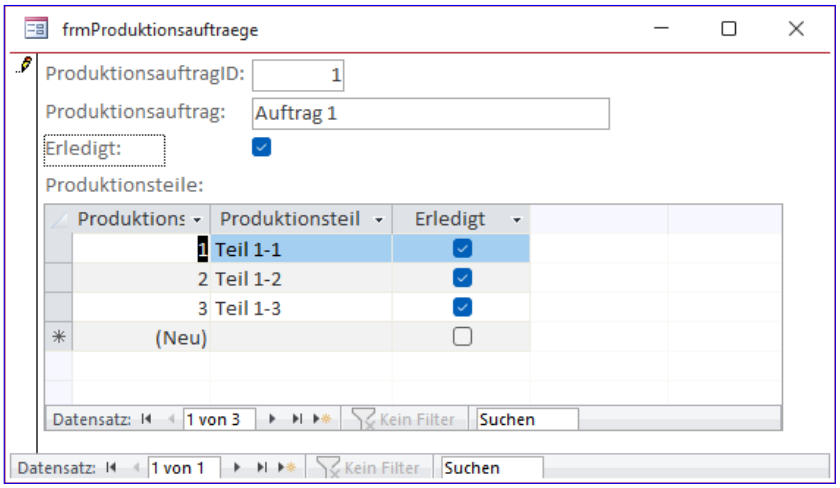

Bild 8: Synchronisierte Daten in Haupt- und Unterformular

Dabei konvertieren wir den Wert des **Boolean**-Feldes mit der **CInt**-Funktion noch in den Datentyp **Integer**. Das ist nötig, da **Boolean**-Werte bei der Ausgabe als **Wahr** oder **Falsch** ausgegeben werden und die SQL-Anweisung damit nichts anfangen kann. Also transformieren wir den Wert zuvor noch in **-1** oder **0**.

Danach aktualisieren wir noch die Daten im Unterformular und erhalten das Ergebnis aus Bild 8.

Hier stellt sich noch die Frage, ob wir dies ohne vorherige Rückfrage durchführen wollen, wenn bereits einige Einträge im Unterformular abgehakt wurden.

Zur Sicherheit bauen wir also noch eine Meldung ein, die wir aber nicht erst im Ereignis **Nach Aktualisierung** aufrufen, sondern bereits im Ereignis **Vor Aktualisierung**

```
Private Sub chkErledigt_BeforeUpdate(Cancel As Integer)
    If MsgBox("Dies setzt den Status Erledigt für alle Teile auf '" & CBool(Me.chkErledigt) & "'. Fortsetzen?", _
            vbYesNo, "Status ändern") = vbNo Then
        Cancel = True
    End If
End Sub
```

Listing 2: Rückfrage, ob die Änderung übertragen werden soll

Daten bearbeiten: Execute vs. Recordset in DAO

Es kommt regelmäßig vor, dass wir Daten in den Tabellen unserer Datenbank bearbeiten müssen. Normalerweise geschieht das über die Benutzeroberfläche. Aber es gibt auch Konstellationen, in denen wir automatisiert Daten zu einer Tabelle hinzufügen oder diese ändern wollen. Manchmal legen wir vollständige Hierarchien inklusive der Daten in verknüpften Tabellen. Oder wir ändern auch nur den Wert eines einzelnen Feldes in einem Datensatz. Dazu können wir verschiedene Techniken nutzen, die wir in diesem Beitrag einmal vorstellen und vergleichen wollen. Dabei konzentrieren wir uns auf das Hinzufügen oder Bearbeiten von einzelnen Datensätzen und schauen uns zwei verschiedene Ansätze an: Das Anlegen oder Aktualisieren von Daten mit INSERT INTO oder UPDATE-Abfragen, die wir per VBA zusammenstellen und dann mit der Execute-Methode ausführen oder das Anlegen mit der Recordset-Methode AddNew/Update und das Bearbeiten mit der Edit-Methode.

DAO oder ADODB?

Mit den eingangs erwähnten beiden Möglichkeiten der Anlage und Änderung von Daten mit **Execute** beziehungsweise **AddNew**/**Edit** und **Update** decken wir die Optionen ab, die uns die DAO-Bibliothek bietet. Wir könnten dies auch noch mit den Methoden der ADODB-Bibliothek erledigen.

Wie wir diese Aufgaben mit ADODB erledigen, beschreiben wir in einem weiteren Beitrag namens **Daten bearbeiten: Execute vs. Recordset in ADODB** (**www.access-im-unternehmen.de/1582**).

Unterschied Execute vs. AddNew/Update

Mit der **Execute**-Anweisung, der wir eine **INSERT INTO**-SQL-Anweisung übergeben, und der Kombination aus **AddNew** und **Update** eines **Recordset**-Objekts erreichen wir grundsätzlich das Gleiche: Wir fügen einer Tabelle einen Datensatz hinzu, der die gewünschten Werte enthält.

Das gilt auch für das Ändern von Datensätzen. Wir können dies mit einer **UPDATE**-SQL-Abfrage erledigen, die wir über die **Execute**-Methode absetzen, oder wir verwenden die **Edit**-Methode, führen dann die Änderungen an den gewünschten Feldern durch und speichern diese mit der **Update**-Methode in der Tabelle.

Mit beiden Methoden können wir bei der Neuanlage eines Datensatzes anschließend die ID des Primärschlüsselwertes auslesen, sofern für dieses Feld die **Autowert**-Funktion aktiviert ist.

Die weiteren Unterschiede, die wir in den folgenden Abschnitten besprechen werden, beziehen sich auf den Komfort, der sich beim Zusammenstellen der jeweiligen Codezeilen ergibt.

Wenn wir die **Execute**-Methode verwenden wollen, müssen wir uns grundlegend mit der Schreibweise von SQL-Anweisungen auskennen, zumindest für die SQL-Abfragen **INSERT INTO** und **UPDATE**.

Außerdem sind hier im Gegensatz zur Verwendung von **AddNew**/**Edit** und **Update** noch einige Besonderheiten bei der Angabe der einzufügenden oder zu ändernden Feldwerte relevant: Wenn wir beispielsweise Textfelder

füllen wollen, müssen wir diese in Hochkommata einfassen, bei Datumsfeldern müssen wir ein SQL-kompatibles Datumsformat verwenden und bei Zahlen mit Dezimaltrennzeichen müssen wir sicherstellen, dass das vom SQL Server verwendete Dezimaltrennzeichen verwendet wird.

Außerdem müssen wir auch die Werte für **Ja/Nein**-Felder entsprechend formatieren.

Die Methoden **AddNew/Edit** plus **Update** sind hier wesentlich einfacher in der Handhabung. Wir können alle Werte einfach übergeben, so wie wir auch in VBA damit arbeiten.

Beispieltabelle

Als Beispiel verwenden wir die Tabelle **tblKunden** aus Bild 1.

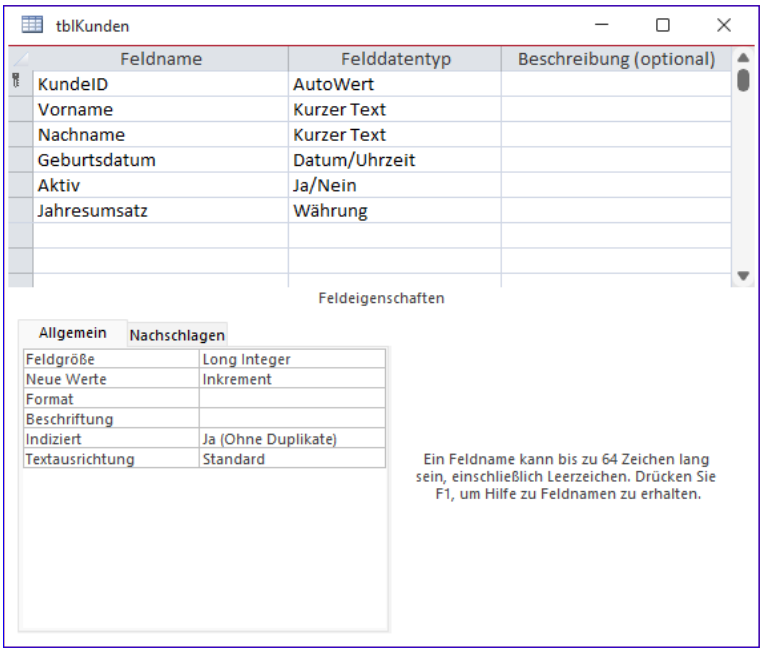

Bild 1: Beispieltabelle **tblKunden**

Diese enthält alle relevanten Datentypen, die wir für die unterschiedlichen Schreibweisen in **INSERT INTO**- und **UPDATE**-Abfragen benötigen: **Kurzer Text**, **Datum**, **Ja/Nein** und **Währung** (stellvertretend für alle Felddatentypen mit Nachkommastellen).

Einfügen von Datensätzen per AddNew/Update

Wir schauen uns zuerst das Einfügen eines Datensatzes mit der **AddNew**- und der **Update**-Methode eines Recordsets an.

Hier deklarieren wir als Erstes zwei Variablen. Mit **db** referenzieren wir das mit der **CurrentDb**-Funktion ermittelte **Database**-Objekt der aktuellen Datenbank. Mit **rst** holen wir uns ein **Recordset**-Objekt auf Basis der Tabelle **tblKunden**.

Dazu nutzen wir die **OpenRecordset**-Methode des **Database**-Objekts:

```
Public Sub Einfuegen_AddNew()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("tblKunden", dbOpenDynaset)
```

Danach können wir direkt mit dem Einfügen eines Datensatzes beginnen. Dazu versetzen wir das Recordset mit der **AddNew**-Methode in den Einfügemodus für einen neuen Datensatz:

```
    rst.AddNew
```

Dann weisen wir den einzelnen Feldern der Tabelle, die wir über das Ausrufezeichen angeben, die gewünschten Werte zu:

```
    rst!Vorname = "André"
    rst!Nachname = "Minhorst"
    rst!Geburtsdatum = "23.01.1971"
    rst!Aktiv = True
    rst!Jahresumsatz = 9999.99
```

Danach schließen wir das Anlegen des neuen Datensatzes ab, indem wir die **Update**-Methode aufrufen und damit

den Datensatz in der dem Recordset zugrunde liegenden Tabelle speichern:

```
    rst.Update
End Sub
```

Damit legen wir den Datensatz aus Bild 2 in der Tabelle an.

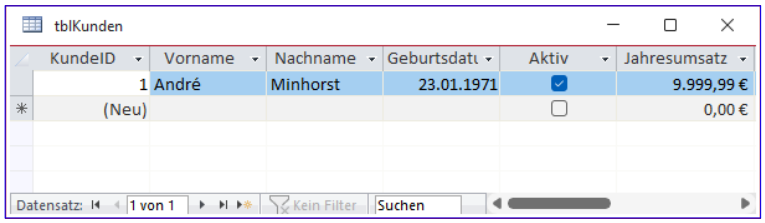

Bild 2: Neuer Datensatz in der Tabelle **tblKunden**

Wo ist das Feld KundeID?

Das Feld **KundeID** haben wir in der Tabelle als **Autowert**-Feld definiert. Das heißt, dass wir es nicht zu füllen brauchen – es wird automatisch mit dem durch die **Autowert**-Funktion ermittelten Wert gefüllt.

Diese entspricht immer dem zuletzt hinzugefügten Wert für dieses Feld plus **1**.

Wir können aber auch das Feld **KundeID** übergeben, wenn wir einmal einen anderen Wert als den durch die Autowert-Funktion vorgegebenen Wert angeben wollen:

```
rst!KundeID = 111
```

Dabei sind folgende Dinge zu beachten:

- Der Wert darf noch nicht vergeben sein, sonst tritt der Fehler **3022** auf, weil das Primärschlüsselfeld jeden Wert nur einmal enthalten darf.
- Der **Autowert** zählt anschließend an dem Wert weiter, den wir manuell zugewiesen haben. Das kann zu Problemen führen, wenn die Tabelle vorher beispielsweise Datensätze mit den ID-Werten **1** und **3** enthalten hat und wir nun einen Datensatz mit dem ID-Wert **2** anlegen. Die **Autowert**-Funktion wird nun als nächsten Wert **3** nutzen, was wiederum zum Fehler **3022** führt.

Die manuelle Vorgabe eines Wertes für ein **Autowert**-Feld sollte also mit Bedacht durchgeführt werden.

Schreibweisen für das Datum

Wir haben hier das Datum einfach als Zeichenkette übergeben (**"23.01.1971"**). Damit haben wir Potenzial für einen Laufzeitfehler geschaffen, denn das Datum muss unbedingt ein gültiges Datum sein. Wir könnten auch die folgende Schreibweise verwenden:

```
rst!Geburtsdatum = #1971-01-23#
```

Diese wird auf Systemen mit deutschen Lokaleinstellungen jedoch direkt in die folgende Zeile umgewandelt:

```
rst!Geburtsdatum = #1/23/1971#
```

Wie können aber auch die Zeichenkette **"23.01.1971"** vorsichtshalber mit der **CDate**-Funktion in ein gültiges Datum umwandeln oder vorab mit **IsDate** prüfen, ob es sich um ein gültiges Datum handelt.

AddNew mit Variablen

Dies können wir auch erledigen, indem wir die Werte für die einzelnen Felder zuvor in Variablen speichern und diese dann den Feldern zuweisen. Wir starten wie zuvor:

```
Public Sub Einfuegen_AddNew_Variablen()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
```

Dann deklarieren wir die Variablen, die wir mit den einzufügenden Werten füllen wollen, und versehen diese gleich mit den entsprechenden Datentypen:

```
    Dim strVorname As String
```

```
    Dim strNachname As String
    Dim datGeburtsdatum As Date
    Dim bolAktiv As Boolean
    Dim curJahresumsatz As Currency
```

Das **Database**-Objekt und das **Recordset**-Objekt füllen wir wie zuvor:

```
    Set db = CurrentDb
    Set rst = db.OpenRecordset("tblKunden", dbOpenDynaset)
```

Dann weisen wir die Werte den Variablen zu, die wir gleich zum Einfügen nutzen wollen:

```
    strVorname = "Klaus"
    strNachname = "Müller"
    datGeburtsdatum = "01.01.2000"
    bolAktiv = False
    curJahresumsatz = 8888.88
```

Schließlich rufen wir **AddNew** auf, weisen die Werte aus den Variablen den einzelnen Feldern zu und speichern den Datensatz mit der **Update**-Methode:

```
    rst.AddNew
    rst!Vorname = strVorname
    rst!Nachname = strNachname
    rst!Geburtsdatum = datGeburtsdatum
    rst!Aktiv = bolAktiv
    rst!Jahresumsatz = curJahresumsatz
    rst.Update
End Sub
```

Dies ist erst einmal wesentlich mehr Schreibarbeit, aber wir bereiten damit etwas vor, was in der Praxis wesentlich häufiger vorkommen wird als das Eintragen von fest im Code angegebenen Werten, nämlich das Übergeben der anzulegenden Informationen per Parameter. Damit können wir mit einem einzigen Aufruf – unter Angabe der für den neuen Datensatz einzufügenden Werte – einen neuen Datensatz in der gewünschten Tabelle anlegen.

AddNew mit Parametern

Dazu holen wir die Variablen einfach in die Parameterliste der Prozedur:

```
Public Sub Einfuegen_AddNew_Parameter( _
        strVorname As String, _
        strNachname As String, _
        datGeburtsdatum As Date, _
        bolAktiv As Boolean, _
        curJahresumsatz As Currency)
```

Die folgenden Schritte sind identisch mit denen der vorherigen Prozedur:

```
    Dim db As DAO.Database
    Dim rst As DAO.Recordset

    Set db = CurrentDb
    Set rst = db.OpenRecordset("tblKunden", dbOpenDynaset)

    rst.AddNew
    rst!Vorname = strVorname
    rst!Nachname = strNachname
    rst!Geburtsdatum = datGeburtsdatum
    rst!Aktiv = bolAktiv
    rst!Jahresumsatz = curJahresumsatz
    rst.Update
End Sub
```

Diese Funktion können wir nun von beliebiger Stelle innerhalb des VBA-Projekts wie folgt aufrufen und haben damit eine Wrapper-Funktion zum Anlegen eines neuen Datensatzes in die Tabelle **tblKunden** geschaffen:

```
Call Einfuegen_AddNew_Parameter("Theo", "Meier",
"31.12.1999", True, 7777.77)
```

ID des neuen Datensatzes bei AddNew auslesen

Wenn wir wie zuvor beschrieben erst einen Kunden anlegen und dann eine Bestellung für diesen hinzufügen wollen, benötigen wir den Wert des Feldes **KundeID** für

den neu hinzugefügten Kunden, um die neue Bestellung mit diesem verknüpfen zu können. Bei Verwendung von **AddNew/Update** ist das Ermitteln allerdings recht einfach.

Wir müssen lediglich den Wert des Feldes **KundeID** abfragen, bevor wir den Datensatz mit der **Update**-Methode speichern.

Warum vorher? Weil durch die **Update**-Methode der Datensatzzeiger nicht mehr auf dem angelegten Datensatz steht. Den Wert des Feldes **KundeID** lesen wir also wie folgt aus:

```
...
rst!Jahresumsatz = 9999.99
Debug.Print "Neuer Kunde: " & rst!KundeID
rst.Update
...
```

Es gibt jedoch noch eine weitere Möglichkeit, die gerade bei Verwendung von SQL Server als Backend notwendig ist. Dabei setzen wir mit **LastModified** ein Bookmark auf den Datensatz, der zuletzt geändert wurde – in diesem Fall den zuletzt angelegten Datensatz.

Anschließend können wir damit wieder den Wert des Feldes **KundeID** für den neuen Datensatz auslesen:

```
...
rst.Update
rst.Bookmark = rst.LastModified
Debug.Print "Neuer Kunde: " & rst!KundeID
...
```

Einfügen von Datensätzen mit Execute/INSERT INTO

Wenn wir die **Execute**-Methode des **Database**-Objekts nutzen wollen, um beispielsweise einen neuen Datensatz mit **INSERT INTO** einzufügen, benötigen wir im Unterschied zu **AddNew/Update** kein **Recordset**-Objekt.

Dafür müssen wir die auszuführende Abfrage aber direkt vollständig zusammenstellen, statt bequem die einzelnen Werte den Feldern zuzuweisen. Das sieht auf den ersten Blick unübersichtlicher aus, aber letztlich sind die gleichen Elemente enthalten.

Wie beginnen mit dem Definieren von Variablen für das **Database**-Objekt und für die zu verwendende SQL-Anweisung:

```
Public Sub Einfuegen_INSERTINTO()
    Dim db As DAO.Database
    Dim strSQL As String
```

Die Variable **strSQL** benötigt man nicht zwangsläufig, aber es kann hilfreich sein, wenn man zu Testzwecken die verwendete SQL-Anweisung im Direktbereich ausgeben möchte. Außerdem wird die Lesbarkeit so verbessert.

Wir füllen wieder die Variable **db** mit dem Wert aus **CurrentDb**:

```
    Set db = CurrentDb
```

Dann stellen wir die SQL-Anweisung in **strSQL** zusammen (in einer Zeile eingeben):

```
    strSQL = "INSERT INTO tblKunden(Vorname, Nachname,
Geburtsdatum, Aktiv, Jahresumsatz) VALUES('André', 'Min-
horst', #1971/01/23#, -1, 9999.99)"
```

INSERT INTO erwartet zunächst den Namen der Zieltabelle und dahinter in Klammern die Liste der Felder, die wir füllen möchten.

Dann folgt das **VALUES**-Schlüsselwort mit den in Klammern eingefassten Werten.

Hier sehen wir direkt die Unterschiede, die das Verwenden von **Execute/INSERT INTO** ein wenig komplizierter machen:

SQL ausführen mit Execute statt DoCmd.RunSQL

In unseren Audits mit unseren Kunden und Lesern untersuchen wir auch regelmäßig den VBA-Code in deren Access-Anwendungen. Dabei fallen uns immer wieder Programmiergewohnheiten auf, die irgendwann einmal eingeführt und seitdem nie wieder geändert wurden. Eine davon ist, SQL-Anweisungen wie INSERT INTO, UPDATE oder DELETE mit der Methode RunSQL der DoCmd-Klasse auszuführen. Das ist grundsätzlich nicht falsch, solange dies zum Ziel führt. Es gibt jedoch noch mindestens eine Alternative, insbesondere den Aufruf mit der Execute-Methode der Database-Klasse. Diese führt zwar auch nur die übergebene Aktionsabfrage aus, bietet aber dennoch Vorteile gegenüber DoCmd.RunSQL. Welche Vorteile das sind und wie wir überhaupt die DoCmd.RunSQL-Methode durch die Execute-Methode ersetzen können, zeigen wir in diesem Beitrag.

RunSQL und Execute einsetzen

Grundsätzlich sind die beiden Methoden ähnlich und dienen dem Aufruf von Aktionsabfragen zum Löschen, Anlegen oder Bearbeiten von Datensätzen einer Tabelle. Als Beispiel verwenden wir eine Tabelle namens **tblKategorien** mit den beiden Feldern **KategorieID** (Primärschlüsselfeld) und **Kategorie** (Textfeld mit eindeutigem Index).

Wenn wir einen Eintrag zu einer Tabelle hinzufügen wollen, erledigen wir das mit **RunSQL** wie folgt (in einer Zeile im Direktbereich eingeben):

```
DoCmd.RunSQL "INSERT INTO tblKategorien(Kategorie) ⤸
                                    VALUES('Kategorie 1')"
```

Bei der **Execute**-Methode können wir direkt mit **CurrentDb** arbeiten und übergeben die gleiche Abfrage:

```
CurrentDb.Execute "INSERT INTO tblKategorien(Kategorie) ⤸
                                    VALUES('Kategorie 1')"
```

Es bietet sich jedoch an, direkt eine Variable für das **Database**-Objekt zu deklarieren. Das ist auch Voraussetzung für das Nutzen der weiteren Vorteile der **Execute**-Methode:

```
Public Sub Beispiel_Execute()
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "INSERT INTO tblKategorien(Kategorie) ⤸
                                    VALUES('Kategorie 2')"
End Sub
```

Warum wird RunSQL überhaupt verwendet?

Einer der Gründe, warum sich die **RunSQL**-Methode der **DoCmd**-Klasse so großer Beliebtheit erfreut, ist vermutlich in der technischen Nähe der **DoCmd**-Methoden zu den Aktionen in den Access-Makros zu finden.

Access-Makros waren einer der Gründe, warum auch Nicht-Programmierer mit Access schnell Ergebnisse erzielen können: Man braucht nicht VBA zu beherrschen, sondern kann schnell im Makro-Editor ein paar Befehle zusammenstellen, die beispielsweise durch den Klick auf eine Schaltfläche ausgeführt werden.

Die Befehle des Makro-Editors finden wir zum größten Teil in den Methoden der **DoCmd**-Klasse.

Wer also in seiner Anfangszeit im Makro-Editor die Methode **AusführenSQL** genutzt hat, und dann zur Nutzung

von VBA übergegangen ist, wird logischerweise zu der entsprechenden **DoCmd**-Methode **RunSQL** gegriffen haben, um das gleiche Ergebnis zu erzielen.

Die Makro-Aktion **AusführenSQL** ist übrigens mindestens seit Access 2010 nicht mehr verfügbar – wir mussten ein altes Access 97-Buch heranziehen, um sicherzugehen, dass es diese Makro-Aktion einmal gab.

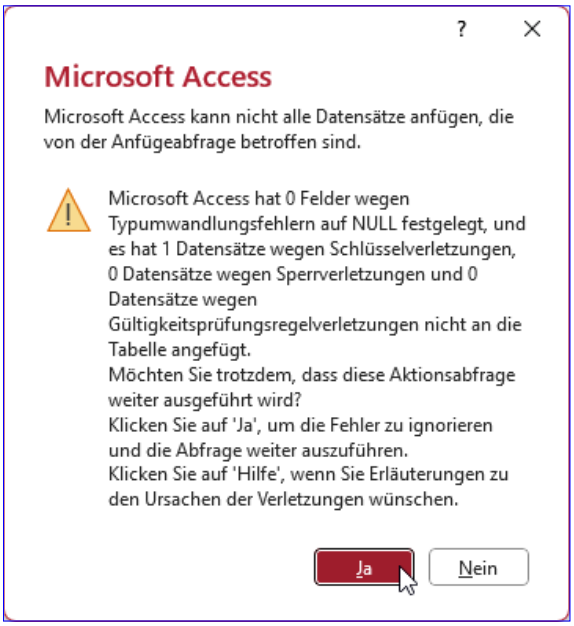

Bild 1: Datenfehler beim **DoCmd.RunSQL**

Und da die **RunSQL**-Methode nach wie vor funktioniert, gab es für viele Entwickler keinen Grund, sich nach einer Alternative umzusehen.

Diese stellen wir in diesem Beitrag mit der **Execute**-Methode der **Database**-Klasse vor und zeigen auch, warum dies die bessere Variante ist. Dafür sprechen die folgenden Gründe:

- Wir können Fehler bei Verwendung von **Execute** über eine benutzerdefinierte Fehlerbehandlung abfangen. Bei **DoCmd.RunSQL** gelingt dies nicht.
- Wir können nach dem Ausführen der **Execute**-Methode direkt ermitteln, wie viele Datensätze von der Aktionsabfrage betroffen sind.
- Und wir können beim Hinzufügen eines Datensatzes mit **INSERT INTO** direkt die ID des Autowertfeldes des hinzugefügten Datensatzes ermitteln.
- Wenn wir mehrere Aktionsabfragen in einer Transaktion ausführen wollen, ist dies nur mit der **Execute**-Methode möglich.

Fehlerbehandlung beim RunSQL vs. Execute

Wenn wir eine SQL-Anweisung mit **RunSQL** ausführen, können wir bestimmte Fehler nicht mit einer benutzerdefinierten Fehlerbehandlung erkennen.

Grundsätzlich werden bei Verwendung von **RunSQL** ohne weitere Maßnahmen alle Fehler über die Benutzeroberfläche gemeldet, zum Beispiel, wenn wir einen Datensatz anfügen wollen und damit einen bereits vorhandenen Wert in einem eindeutigen Feld hinzufügen würden:

```
Public Sub Beispiel_RunSQL_Fehler()
    DoCmd.RunSQL "INSERT INTO tblKategorien(Kategorie) ↵
                                VALUES('Kategorie 1')"
End Sub
```

Dieser Fehler würde uns nur über die Benutzeroberfläche gemeldet werden (siehe Bild 1).

Wir können diesen Fehler nicht über eine Fehlerbehandlung etwa mit **On Error Resume Next** abfangen und auch die Fehlernummer anschließend nicht mit **Debug.Print Err.Number** auswerten.

Wir können lediglich die Anzeige der Fehlermeldung unterbinden, indem wir zuvor die Anweisung **DoCmd.SetWarnings False** einstellen und diese anschließend mit **DoCmd.SetWarnings True** wieder aktivieren. In diesem Fall würden wir den Fehler jedoch gar nicht bemerken.

Andere Fehler, wie Tippfehler in Tabellen- oder Feldnamen, können wir hingegen mit einer benutzerdefinierten Fehlerbehandlung abfangen:

Ordner und Dateien in Access-Tabellen einlesen

Es gibt verschiedene Gründe, warum man Ordner und Dateien aus dem Filesystem in eine entsprechende Datenstruktur einlesen sollte. Der Erste ist offensichtlich: Weil man die Laufwerke, Ordner und Dateien oder auch nur Teile davon innerhalb der Datenbank anzeigen möchte, beispielsweise um zu sehen, welche Dateien zu einem bestimmten Projekt oder Kunden gehören. Der erste Schritt auf dem Weg zu einer solchen Anzeige ist das Einlesen der gewünschten Struktur – unabhängig davon, ob der komplette Inhalt einer Festplatte oder nur der Inhalt eines Unterverzeichnisses abgebildet werden soll. Zum Einlesen von Laufwerken, Ordnern und Dateien gibt es verschiedene Möglichkeiten auf beiden Seiten. Auf der Seite des Dateisystems können wir mit der Dir-Funktion oder alternativ mit dem FileSystemObject arbeiten, und beim Schreiben in die Tabellen der Datenbank bietet sich unter DAO das Schreiben mit AddNew/Update oder mit der Execute-Methode an. In diesem Artikel stellen wir die schnellsten Versionen vor, damit das Einlesen umfangreicher Verzeichnis- und Dateistrukturen nicht unnötig lange dauert.

Alles oder nur einen Teil einlesen?

Technisch haben wir alle Möglichkeiten. Wir können mit den Elementen und Methoden der **FileSystemObject**-Klasse auf alle Laufwerke zugreifen und uns von dort auch durch die einzelnen Verzeichnisse arbeiten und schließlich die darin enthaltenen Dateien ermitteln.

Das ist jedoch nur bedingt sinnvoll, da die Datenmengen schnell riesig werden und wir den in unserer Datenbank gespeicherten Bestand möglichst synchron mit der Festplatte halten wollen. Das erfordert regelmäßige Aktualisierungen, was jeweils Minuten oder sogar Stunden dauern kann.

Also entscheiden wir uns bereits an dieser Stelle, immer nur einen Teil des Dateisystems einzulesen – in diesem Fall beginnend mit der Angabe des Verzeichnisses, dessen Inhalte wir erfassen wollen.

Den Ausgangspunkt für den zu entwickelnden Algorithmus bildet also die Auswahl des Verzeichnisses, dessen Unterelemente wir in unser Datenmodell überführen wollen.

Datenmodell für die Erfassung von Verzeichnissen und Dateien

Um die Struktur des Dateisystems bezüglich des von uns gewählten Ordners in einer Datenbank zu speichern, haben wir ebenfalls mehrere Möglichkeiten.

Wir können einfach eine Tabelle erstellen, in die wir immer den vollständigen Pfad der Verzeichnisse und Dateien schreiben. Das macht es aber aufwendiger, etwa ein **TreeView** mit diesen Daten zu füllen.

Wir müssten uns dann mit vielen Zeichenkettenoperationen durch die einzelnen Verzeichnisebenen eines Pfades arbeiten, was sehr viel Zeit kostet. Außerdem ist es nicht unbedingt sehr platzsparend, wenn wir immer wieder die gleichen übergeordneten Verzeichnisse in einem Datensatz ablegen.

Also wählen wir die Alternative, die aus einem Satz von drei Tabellen besteht. Hier benötigen wir zunächst eine Tabelle, um die Verzeichnisse zu speichern, beginnend mit den Verzeichnissen der ersten Ebene. Die dazu be-

nötigten Felder lauten beispielsweise **FolderID** und **Foldername**. Damit sind wir allerdings darauf beschränkt, nur Ordnernamen speichern zu können – wir müssen also noch einen Weg finden, die Zuordnung der einzelnen Verzeichnisse zum jeweils übergeordneten Verzeichnis zu markieren.

Also fügen wir der Tabelle noch ein Feld namens **ParentID** hinzu, mit dem wir für einen Ordner jeweils den Datensatz mit dem übergeordneten Ordner angeben können. Wir speichern also in einer Tabelle sowohl die Ordnernamen als auch die Information über die Hierarchie dieser Ordner.

Über das Feld **ParentID** erzeugen wir eine reflexive Beziehung der Datensätze der Tabelle auf sich selbst. Schließlich fügen wir der Tabelle, die wir **tblFolder** nennen und deren Entwurf wie in Bild 1 aussieht, noch ein Feld namens **UID** hinzu.

In NTFS-Dateisystemen (New Technology File System), die bereits mit Windows 3.1 eingeführt wurden, können wir mit **API**-Funktionen eine eindeutige ID für Ordner und Dateien ermitteln. Wozu wir diese benötigen und wie wir diese auslesen, erläutern wir später.

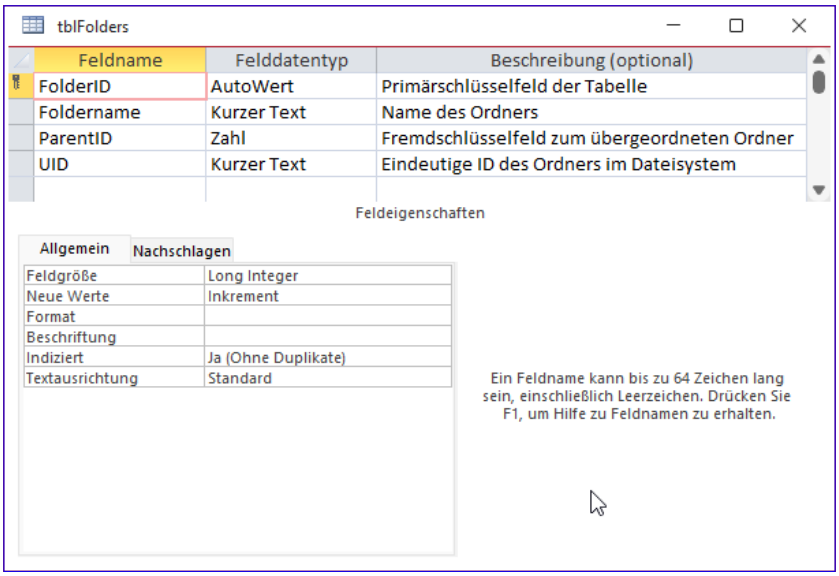

Bild 1: Tabelle zum Speichern der Ordner

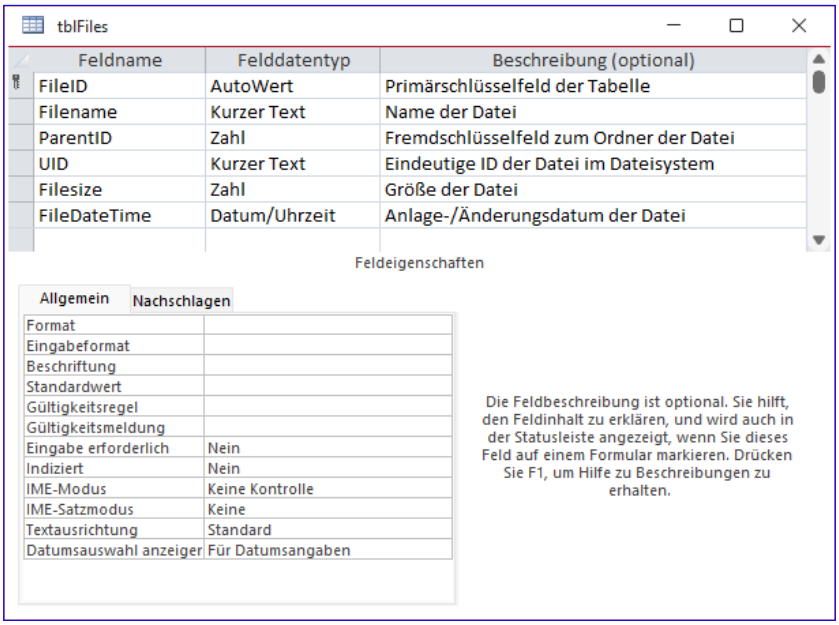

Bild 2: Tabelle zum Speichern der Dateien

Zunächst kümmern wir uns aber um die Tabelle zum Speichern der Dateiinformationen. Diese enthält wiederum ein Primärschlüsselfeld (**FileID**), ein Feld zum Speichern des Dateinamens (**Filename**) sowie ein Feld, mit dem wir die Beziehung zu dem Ordner herstellen, in dem sich die Datei befindet, und die wir wiederum **ParentID** nennen. Außerdem fügen wir auch hier ein Feld namens **UID** für den eindeutigen Identifizierer für die Datei sowie zwei Felder zum Speichern der Dateigröße und des Anlage- beziehungsweise letzten Änderungsdatums hinzu (siehe Bild 2).

Schließlich ergänzen wir im **Beziehungen**-Fenster noch die notwendigen Beziehungen (siehe Bild 3). Hier ziehen wir zunächst die Tabelle **tblFolder** zwei Mal hinein und erstellen eine Beziehung des Feldes **ParentID** des im **Be-**

ziehungen-Fenster mit **tblFolders_1** benannten zweiten Exemplars der Tabelle **tblFolders** zu dem mit **tblFolder** benannten Exemplar. Damit realisieren wir die Beziehung von Unterordnern zum übergeordneten Ordner. Außerdem ziehen wir noch einen Beziehungspfeil vom Feld **ParentID** der Tabelle **tblFiles** zum Feld **FolderID** der Tabelle **tblFolders**.

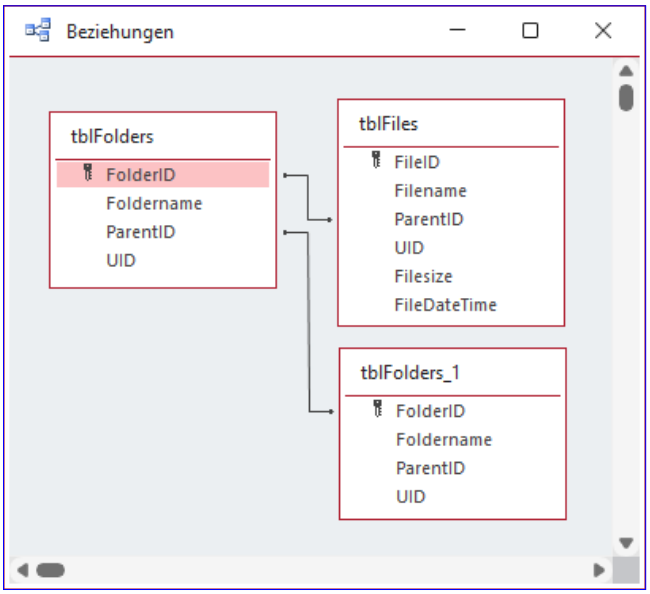

Bild 3: Beziehungen zwischen den Tabellen

Einlesen der Ordner und Dateien

Die intuitive Vorgehensweise zum Einlesen der Ordner und Dateien des gewünschten Ordners würde sich nach dem Aufbau des Dateisystems und unserer Tabellenstruktur richten.

Wir würden also etwa die Klassen und Methoden der **FileSystemObject**-Klasse nutzen, um ausgehend vom Basisordner zunächst die darin enthaltenen Ordner einzulesen und in die Tabelle **tblOrdner** zu schreiben. Beim Durchlaufen dieser Ordner würden wir in einer rekursiven Prozedur die untergeordneten Ordner und die Dateien dieses Ordners einlesen und so weiter.

Diese Vorgehensweise ist jedoch nicht schnell genug. Beim Einlesen umfangreicher Ordnerstrukturen wollen wir schließlich nicht ewig warten. Deshalb wählen wir hier einen alternativen Ansatz, der allerdings etwas komplexer ist und wie in Listing 1 beginnt.

Was macht die Prozedur **OrdnerUndDateienEinlesen** überhaupt?

- Wir haben einen Startordner (zum Beispiel **C:\Buecher**).
- Darin sind Unterordner und Dateien.
- In den Unterordnern sind wieder Unterordner und Dateien. Das Ganze als Baum.
- Die Prozedur läuft durch den ganzen Baum, schreibt alle Ordner in **tblFolders**, schreibt alle Dateien in **tblFiles**, merkt sich zu jeder Datei und jedem Ordner, wo sie liegen (**ParentID**), und speichert außerdem eine UID (damit wir sie später wiedererkennen) und Größe und Datum (für Dateien).

Danach können wir mit diesen Tabellen bequem arbeiten, zum Beispiel zum Füllen eines **TreeView**-Steuerelements.

Die Prozedur OrdnerUndDateienEinlesen Schritt für Schritt erklärt

Die Prozedur bekommt mit dem Parameter **strRoot** einen Startpfad. Als Erstes deklarieren wir die Variablen:

- **wrk** und **db**: Verweise auf die aktuelle Datenbank und das **Workspace**-Objekt
- **rstFolders** und **rstFiles**: Recordsets für **tblFolders** und **tblFiles**
- **colTodo**: Eine Collection als To-do-Liste mit Ordnern, die noch abgearbeitet werden müssen
- **strPfad**, **strEintrag** und **strVollPfad**: String-Variablen für aktuelle Pfade/Namen

```
Public Sub OrdnerUndDateienEinlesen(ByVal strRoot As String)
    Dim wrk As DAO.Workspace
    Dim db As DAO.Database
    Dim rstFolders As DAO.Recordset
    Dim rstFiles As DAO.Recordset
    Dim colTodo As Collection
    Dim strPfad As String
    Dim strEintrag As String
    Dim strVollPfad As String
    Dim lngAttr As Long
    Dim lngCounter As Long
    Dim strUID As String
    Dim lngParentID As Long
    Dim lngCurrentFolderID As Long
    Dim lngTimer As Long
    Dim bolIsRoot As Boolean

    lngTimer = Timer

    If Right$(strRoot, 1) = "\" Then
        strRoot = Left$(strRoot, Len(strRoot) - 1)
    End If

    Set wrk = DBEngine(0)
    Set db = wrk.Databases(0)

    Call TabellenZuruecksetzen(db)

    Set rstFolders = db.OpenRecordset("tblFolders", dbOpenDynaset)
    Set rstFiles = db.OpenRecordset("tblFiles", dbOpenDynaset)
    Set colTodo = New Collection

    Call TodoAdd(colTodo, strRoot, 0)
    bolIsRoot = True
    DoCmd.Echo False
    DoCmd.Hourglass True
    wrk.BeginTrans
    On Error GoTo Fehler
    ...
```

Listing 1: Die Prozedur **OrdnerstrukturEinlesen** (Teil 1)

- **lngAttr**: Dateiattribut (ist es ein Ordner oder eine Datei?)
- **lngCounter**: Wie viele Dateien haben wir schon gefunden?
- **strUID**: Datei-/Ordner-ID, die wir mit der Funktion **GetFileID** holen
- **lngParentID** und **lngCurrentFolderID**: Verweise auf übergeordnete Ordner

- **lngTimer**: Erfassung der Laufzeit

- **bolIsRoot**: Gibt an, ob wir noch im Root-Ordner sind

Zu Beginn speichern wir den aktuellen Timer-Wert in **lngTimer**, um später die Gesamtzeit für den Vorgang ausgeben zu können. Außerdem schneiden wir vom **Root**-Ordner in **strRoot** noch ein eventuell am Ende befindliches Backslash-Zeichen ab, falls dieses noch vorhanden ist.

Workspace und Transaktion für schnelleres Schreiben

Danach initialisieren wir die **Workspace**-Variable **wrk** und die **Database**-Variable **db**. Das **Workspace**-Objekt benötigen wir, weil wir damit die vielen Anlegevorgänge in einer Transaktion bündeln können, was wesentlich schneller funktioniert, als wenn wir jeden Vorgang einzeln durchführen.

Tabellen zurücksetzen und leeren

Danach rufen wir die Prozedur **TabellenZuruecksetzen** auf. Diese löscht nicht nur einfach die Daten, sondern fügt zuvor einen neuen Datensatz in die beiden Tabellen **tblFolders** und **tblFiles** ein, der im Primärschlüsselfeld den Wert **0** enthält. Damit setzen wir den Autowert der beiden Tabellen zurück, sodass beim Neuanlegen von Datensätzen nachfolgend wieder mit dem Wert **1** gestartet wird. Anschließend löschen wir alle Datensätze aus diesen beiden Tabellen:

```
Public Sub TabellenZuruecksetzen(db As DAO.Database)
    db.Execute _
        "INSERT INTO tblFiles(FileID, Filename) " _
        & "VALUES(0, '')", dbFailOnError
    db.Execute _
        "INSERT INTO tblFolders(FolderID, Foldername) " _
        & "VALUES(0, '')", dbFailOnError

    db.Execute "DELETE FROM tblFiles", dbFailOnError
    db.Execute "DELETE FROM tblFolders", dbFailOnError
End Sub
```

Weitere Initialisierungen

Danach öffnen wir zwei Recordsets: **rstFolders** für die Ordner und **rstFiles** für die Dateien. Außerdem legen wir ein **Collection**-Objekt namens **colToDo** an, mit dem wir noch zu bearbeitende Ordner speichern.

Hier legen wir als Erstes den Startordner aus dem Parameter **strRoot** mit dem Wert **0** ab. Das geschieht in einer weiteren Hilfsprozedur namens **TodoAdd**.

Dieser übergeben wir das **Collection**-Objekt, den Pfad und die ID des übergeordneten Ordners als Parameter.

Wir fügen der Collection dann einen Eintrag hinzu, der aus der ID des übergeordneten Ordners, dem Pipe-Zeichen (**|**) und dem Pfad besteht. Im ersten Aufruf tragen wir also den Wert **0|[Pfad]** ein:

```
Private Sub TodoAdd(ByRef col As Collection, _
        ByVal strPfad As String, ByVal lngParentID As Long)
    col.Add CStr(lngParentID) & "|" & strPfad
End Sub
```

Der Wert **0** bedeutet in diesem Fall, dass es keinen übergeordneten Ordner gibt.

Da dieser erste Ordner eine Spezialbehandlung erfahren soll, stellen wir außerdem die Variable **bolIsRoot** auf **True** ein.

Schließlich deaktivieren wir die Bildschirmaktualisierung mit **DoCmd.Echo False** und aktivieren die Sanduhr mit **DoCmd.Hourglass True**.

Starten der Transaktion und der Do While-Schleife

Nun starten wir die Transaktion und integrieren die Fehlerbehandlung (siehe Listing 2).

Anschließend starten wir eine **Do While**-Schleife, in der wir alle Elemente der Collection **colToDo** durchlaufen. Die

```
...
Do While colTodo.Count > 0
    Call TodoPop(colTodo, strPfad, lngParentID)
    If Right$(strPfad, 1) <> "\" Then
        strPfad = strPfad & "\"
    End If
    If Not bolIsRoot Then
        rstFolders.AddNew
        rstFolders!FolderName = GetFolderNameFromPath(strPfad)
        If lngParentID > 0 Then
            rstFolders!ParentID = lngParentID
        Else
            rstFolders!ParentID = Null
        End If
        rstFolders!UID = GetFileID(strPfad)
        lngCurrentFolderID = rstFolders!FolderID

        rstFolders.Update
    Else
        lngCurrentFolderID = 0
        bolIsRoot = False
    End If
    strEintrag = Dir$(strPfad & "*", vbDirectory)
    Do While strEintrag <> ""
        If strEintrag <> "." And strEintrag <> ".." Then
            strVollPfad = strPfad & strEintrag
            strUID = GetFileID(strVollPfad)
            If Len(strUID) > 0 Then
                lngAttr = GetAttr(strVollPfad)
                If (lngAttr And vbDirectory) = vbDirectory Then
                    Call TodoAdd(colTodo, strVollPfad, lngCurrentFolderID)
                Else
                    rstFiles.AddNew
                    rstFiles!FileName = strEintrag
                    rstFiles!ParentID = lngCurrentFolderID
                    rstFiles!UID = strUID
                    rstFiles!Filesize = FileLen(strVollPfad)
                    rstFiles!FileDateTime = FileDateTime(strVollPfad)
                    rstFiles.Update
                    lngCounter = lngCounter + 1
                End If
            End If
        End If
        strEintrag = Dir$()
    Loop
...
```

Listing 2: Die Prozedur **OrdnerstrukturEinlesen** (Teil 2)

Dateien schnell im TreeView-Steuerelement anzeigen

Im Artikel »Ordner und Dateien in Access-Tabellen einlesen« (www.access-im-unternehmen.de/1583) haben wir gezeigt, wie wir den Inhalt kompletter Ordner samt Unterordnern und Dateien in Tabellen speichern. Doch was helfen die dort liegenden Daten, wenn wir sie nicht in einem Access-Formular anzeigen können? Wie das gelingt, zeigen wir im vorliegenden Artikel. Als Steuerelement für die Anzeige hierarchischer Daten ist das TreeView-Steuerelement prädestiniert. Wir möchten alle Elemente der Tabellen aus dem oben genannten Artikel in einem solchen Steuerelement anzeigen und weitere Funktionen hinzufügen: die Anzeige des jeweiligen Ordners direkt im Windows Explorer, das Öffnen der aktuell markierten Datei oder auch das Ausschneiden, Kopieren und Einfügen, das wir nicht nur auf die Elemente des TreeView-Steuerelements anwenden, sondern auch auf die Originaldateien. Auch das Umbenennen von Ordnern und Dateien soll möglich sein – und schließlich wollen wir auch noch deren Speicherort durch Drag and Drop anpassen können. In diesem Artikel erfahren Sie, wie Sie das TreeView schnell mit Ordnern und Dateien füllen können.

Formular mit TreeView-Steuerelement erstellen

Zunächst legen wir ein neues Formular namens **frmDateienImTreeview** in der Entwurfsansicht an.

Diesem fügen wir gleich das **TreeView**-Steuerelement hinzu, mit dem wir die Dateien anzeigen wollen, und nennen es **ctlTreeView**.

Das **TreeView**-Steuerelement soll für seine Einträge Icons anzeigen, daher fügen wir noch ein **ImageList**-Steuerelement namens **ctlImageList** hinzu. Oben fügen wir weitere Steuerelemente ein:

- ein Textfeld namens **txtBasispfad** zur Anzeige des aktuellen Hauptverzeichnisses,

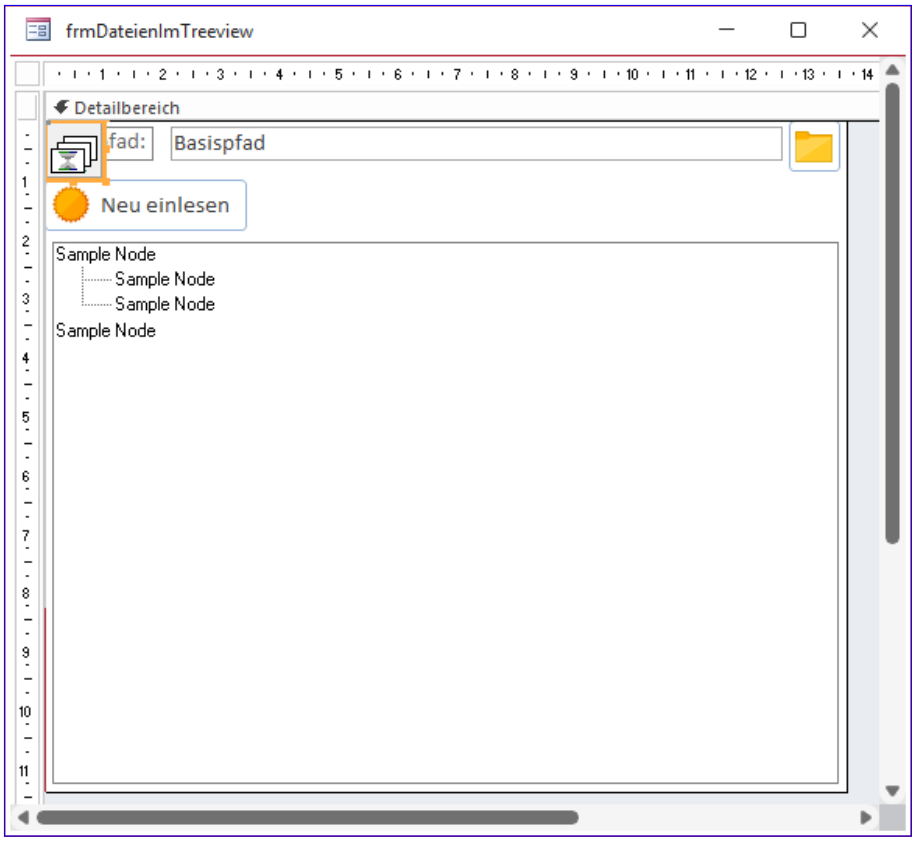

Bild 1: Grundaufbau des Formulars

- eine Schaltfläche namens **cmdOrdnerAuswaehlen** zum Auswählen des Hauptverzeichnisses, dessen Unterordner und Dateien angezeigt werden sollen,
- ein Kontrollkästchen namens **chkDateienImTreeViewAnzeigen**, um festzulegen, ob die Dateien im **TreeView**-Steuerelement ein- oder ausgeblendet werden sollen, und
- eine weitere Schaltfläche namens **cmdNeuEinlesen**, mit der wir das **TreeView**-Steuerelement erneut füllen können.

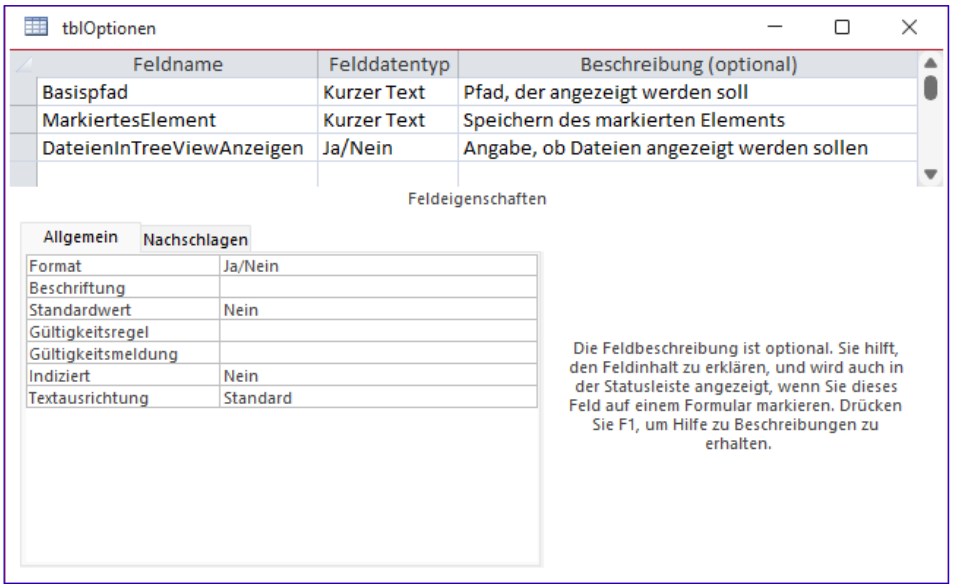

Bild 2: Entwurf der Optionen-Tabelle

Der Entwurf sieht nun zunächst wie in Bild 1 aus.

Optionentabelle anlegen

Die Einstellungen der beiden Steuerelemente **txtBasispfad** und **chkDateienImTreeViewAnzeigen** wollen wir speichern und jeweils beim nächsten Öffnen des Formulars wiederherstellen. Dazu benötigen wir eine Tabelle namens **tblOptionen**, deren Entwurf wie in Bild 2 aussieht.

Das Formular binden wir über die Eigenschaft **Datensatzquelle** an die Tabelle **tblOptionen**, die beiden Steuerelemente **txtBasispfad** und **chkDateienImTreeViewAnzeigen** über die Eigenschaft **Steuerelementinhalt** an die jeweiligen Felder der Optionentabelle.

Basispfad auswählen

Damit wir den Ordner auswählen können, dessen Unterordner und Dateien im **TreeView**-Steuerelement angezeigt werden sollen, hinterlegen wir für die Schaltfläche **cmdOrdnerAuswaehlen** die folgende Ereignisprozedur:

```
Private Sub cmdOrdnerAuswaehlen_Click()
    Me.txtBasispfad = ChooseFolder
    Me.Dirty = False
End Sub
```

Diese zeigt über die Funktion **ChooseFolder** einen Ordnerauswahl-Dialog an:

```
Public Function ChooseFolder()
    Dim objFileDialog As Office.FileDialog
    Dim strTemp As String

    Set objFileDialog = _
        Application.FileDialog(msoFileDialogFolderPicker)

    With objFileDialog
        .Title = "Datei auswählen"
        .ButtonName = "Auswählen"
        .InitialFilename = CurrentProject.Path & "\"
        If .Show = True Then
            strTemp = .SelectedItems(1)
        End If
    End With
    ChooseFolder = strTemp
End Function
```

```
Private Sub Form_Load()

    DoCmd.Hourglass True

    Set objImageList = Me.ctlImageList.Object

    objImageList.ImageHeight = 16
    objImageList.ImageWidth = 16

    Call ImageListFuellen

    Set objTreeview = Me.ctlTreeView.Object
    With objTreeview
        Set .ImageList = objImageList
        .Nodes.Clear
        .Appearance = ccFlat
        .BorderStyle = ccNone
        .HideSelection = False
        .LineStyle = tvwRootLines
        .Indentation = 250
        .Font.name = "Calibri"
        .Font.Size = 10
        .OLEDragMode = ccOLEDragAutomatic
        .OLEDropMode = ccOLEDropManual
    End With

    Call FillTreeView(Me.chkDateienInTreeviewAnzeigen)
    Call SelectCurrentNode

    DoCmd.Hourglass False
End Sub
```

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

Um die hier verwendete Klasse **FileDialog** zu nutzen, müssen wir noch einen Verweis auf die Bibliothek **Microsoft Office 16.0 Object Library** hinzufügen.

Nach der Auswahl werden die Daten des Formulars mit **Me.Dirty = False** gespeichert.

Füllen des TreeView-Steuerelements

Bevor wir uns die Prozeduren ansehen, mit denen wir die Ordner und Dateien aus den Tabellen **tblFolders** und **tblFiles** in das **TreeView**-Steuerelement laden, fügen wir im Kopf des Klassenmoduls des Formulars **frmDateienImTreeView** die folgenden Deklarationsanweisungen hinzu:

```
Dim objTreeview As MSComctlLib.TreeView
Dim objImageList As MSComctlLib.ImageList
Dim dicFolders As Scripting.Dictionary
Dim dicFiles As Scripting.Dictionary
```

Die ersten beiden benötigen wir, um die Steuerelemente **ctlTreeView** und **ctlImageList** zu referenzieren, die beiden übrigen als temporären Speicher für die anzuzeigenden Ordner und Dateien. Um diese **Dictionary**-Elemente nutzen

zu können, benötigen wir einen weiteren Verweis, dieses Mal auf die Bibliothek **Microsoft Scripting Runtime**.

Die Ereignisprozedur Beim Laden

In der Prozedur, die durch das Ereignis **Beim Laden** des Formulars ausgelöst wird (siehe Listing 1), aktivieren wir zunächst die Sanduhr (die wir am Ende wieder deaktivieren) und weisen der Variablen **objImageList** die Eigenschaft **Object** des Steuerelements **ctlImageList** zu. Damit können wir per IntelliSense auf die spezifischen Eigenschaften dieses Steuerelements zugreifen.

Dann stellen wir seine Eigenschaften **ImageHeight** und **ImageWidth** jeweils auf 16 Pixel ein, um die Größe der anzuzeigenden Icons festzulegen. Schließlich rufen wir die Prozedur **ImageListFuellen** auf (siehe Listing 2). Diese geht davon aus, dass wir die Icons, die wir im **TreeView**-Steuerelement anzeigen wollen, in der Systemtabelle **MSysResources** gespeichert haben.

Das haben wir bereits erledigt (siehe Bild 3). Die Prozedur **ImageListFuellen** öffnet ein Recordset basierend auf der Tabelle **MSysResources**, gefiltert nach den Elementen des Typs **png**.

Dann referenziert sie das **ImageList**-Steuerelement mit der bereits im Modulkopf deklarierten Variablen **objImageList**. Anschließend wird die Eigenschaft **ImageList**

```
Private Sub ImageListFuellen()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim objImageList As MSComctlLib.ImageList

    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM MSysResources WHERE Extension = 'PNG'", dbOpenSnapshot)

    Set objImageList = Me.ctlImageList.Object

    Set ctlTreeView.Object.ImageList = Nothing

    objImageList.ListImages.Clear

    Do While Not rst.EOF
        Call amvAddIconToImageListFromResourcesByName(objImageList, rst!name)
        rst.MoveNext
    Loop

    Dim objListImage As MSComctlLib.ListImage
    For Each objListImage In objImageList.ListImages
        Debug.Print objListImage.Index, objListImage.Key
    Next objListImage

    rst.Close
    Set rst = Nothing
    Set db = Nothing
End Sub
```

Listing 2: Füllen des **ImageList**-Steuerelements mit den Bildern aus **MSysResources**

des Steuerelements **ctlTreeView** geleert – dieses wird später erneut zugewiesen.

Dann leert die Prozedur das **ImageList**-Steuerelement mit der **Clear**-Methode der **ListImages**-Auflistung. Dann ruft die Prozedur eine weitere Prozedur namens **amvAddIconToImageListFromResourcesByName** auf.

Diese wollen wir hier nicht im Detail beschreiben – sie lädt das Element mit dem Namen aus **rst!Name** für den aktuellen Datensatz des Recordsets und fügt es zum **ImageList**-Steuerelement hinzu (siehe Modul **MDL_AMV_Pictures**). Auf diese Weise landen alle **.png**-Dateien aus der Tabelle **MSysResources** im **ImageList**-Steuerelement und können so im **TreeView**-Steuerelement verwendet werden.

msysresources

	Extension	Id	Name	Type
(1)	thmx	1	Office Theme	thmx
(1)	png	3	book	img
(1)	png	4	folder	img
(1)	png	5	new	img
(1)	png	6	delete	img
(1)	png	7	refresh	img
(0)		(Neu)		

Datensatz: 1 von 6 | Kein Filter | Suchen

Bild 3: Icons für das **TreeView**-Steuerelement in der Tabelle **MSysResources**

Um dies nicht im **ImageList**-Steuerelement prüfen zu müssen, gibt die Prozedur den Index und die Namen aller Elemente einmal im Direktbereich des VBA-Editors aus – diesen Bereich können wir im produktiven Einsatz später entfernen. Das Ergebnis sieht in diesem Fall wie folgt aus:

```
1           book
2           folder
3           new
4           delete
5           refresh
```

Danach schließt und leert die Prozedur alle Objektvariablen.

Zurück in der Prozedur **Form_Load** referenzieren wir nun das **TreeView**-Steuerelement aus **Me.ctlTreeView.Object** mit der Variablen **objTreeView** und stellen einige Eigenschaften ein. Als Erstes weisen wir diesem für die Eigenschaft **ImageList** den Inhalt von **objImageList** zu. So weiß das **TreeView**-Steuerelement, woher es seine Icons beziehen soll, die wir nachher den Elementen zuweisen.

Dann leert die Prozedur alle gegebenenfalls noch vorhandenen Elemente. Schließlich folgt die Einstellung weiterer Eigenschaften, die für das Aussehen des **TreeView**-Steuerelements verantwortlich sind:

- Appearance erhält den Wert **ccFlat**, damit es nicht mit 3d-Effekt angezeigt wird,
- mit **BorderStyle** (Wert: **ccNone**) blenden wir den Rahmen des Steuerelements aus (wir fügen diesen über die Eigenschaften des Steuerelements selbst über das Eigenschaftenblatt wieder hinzu),
- mit **HideSelection** (**False**) legen wir fest, dass das aktivierte Element auch beim Fokusverlust zumindest grau hinterlegt wird,
- mit **LineStyle** (**tvwRootLines**) legen wir die Art der Verbindungslinien zwischen den Elementen fest,
- **Indentation** stellen wir auf **250** ein, damit die Elemente nicht so weit eingerückt werden wie im Standard,
- mit **Font.Name** und **Font.Size** stellen wir Schriftart und -größe fest und
- mit **OLEDragMode** (**ccOLEDragAutomatic**) und **OLEDropMode** (**ccOLEDropManual**) legen wir fest, dass