HAUFE.
Ausgabe 06/2013

ACCESS

IM UNTERNEHMEN

DATENMODELLE VERGLEICHEN

Sie haben mehrere Versionen einer Datenbank und wissen nicht mehr, welche Änderungen Sie am Datenmodell durchgeführt haben? Kein Problem: Mit unserer Lösung finden Sie die Unterschiede per Mausklick (S. 61).

In diesem Heft:

PARAMETERABFRAGEN UNTER DER LUPE

Erfahren Sie alles Wissenswerte rund um den Einsatz von Parameterabfragen in Access-Datenbanken.

SEITE 10

UNDO IN HAUPT- UND UNTERFORMULAR

Änderungen an verknüpften Daten en bloc rückgängig machen? Das erledigen Sie natürlich mit Klasse.

SEITE 31

PERFORMANCE OPTIMIEREN

Wie Sie durch die Umstellung von Recordsets und Do While auf Aktionsabfragen richtig Zeit sparen.

SEITE 20

DOWNLOAD

Die Downloads zu dieser Ausgabe finden Sie wie gewohnt unter:

http://www.access-im-unternehmen.de/download

Benutzername: **nur für die Komplettausgabe**

Kennwort: **nur für die Komplettausgabe**

Die Direktlinks zu den Downloads befinden sich am unteren Rand des jeweiligen Beitrags.

Impressum

ISBN 978-3-8092-1496-0 ISSN: 1616-5535 Mat.-Nr. 01583-5080

Access im Unternehmen

Kommanditgesellschaft, Sitz Freiburg
Registergericht Freiburg, HRA 4408
Komplementäre: Haufe-Lexware Verwaltungs GmbH, Sitz Freiburg,
Registergericht Freiburg, HRB 5557; Martin Laqua
Geschäftsführung: Isabel Blank, Markus Dränert, Jörg Frey, Birte Hackenjos, Randolf Jessl, Markus Reithwiesner, Joachim Rotzinger, Dr. Carsten Thies

Beiratsvorsitzende: Andrea Haufe
Steuernummer: 06392/11008
Umsatzsteuer-Identifikationsnummer: DE 812398835

Redaktion: Dipl.-Inform. (FH) Michael Forster, (Chefredakteur, V.i.S.d.P.), Dipl.-Ing. André Minhorst (Chefredaktion, extern),
Frieda Flechler (Redaktionsassistentin), Rita Klingenstein (sprachl. Lektorat), Sascha Trowitzsch (Fachlektorat)
Autoren: André Minhorst, Sascha Trowitzsch

Anschrift der Redaktion: Postfach 100121, 79120 Freiburg, Tel. 0761/898-0, Fax: 0761/898-3919,
E-Mail: computer@haufe.de, Internet: http://computer.haufe.de

Druck: J.P. Himmer GmbH & Co KG, Postfach 10 18 05, 86008 Augsburg

Auslieferung und Vertretung für die Schweiz: H.R. Balmer AG, Neugasse 12, CH-6301 Zug, Tel. 041/711 4735, Fax: 041/711 0917

Schritt halten

Wer mit Access programmiert, hat im Vergleich zu anderen Entwicklungsumgebungen ein entspanntes Leben: Es gibt für die Desktop-Entwicklung kaum Neuerungen, und der VBA-Editor wurde bereits seit Äonen nicht mehr weiterentwickelt. Dennoch ist Access allen aktuellen Anforderungen gewachsen – es gibt kaum etwas, was sich nicht mit Bordmitteln und der einen oder anderen externen Bibliothek erledigen lässt.

In diesem Sinne wollen wir selbst dafür sorgen, dass sich Access dennoch weiterentwickelt und immer nützlicher und einfacher zu bedienen wird – dies vor allem durch den Einsatz geeigneter Lösungen und Add-Ins.

In dieser Ausgabe von **Access im Unternehmen** finden Sie so eine Lösung zum Vergleichen von Datenmodellen zweier Datenbanken. Damit können Sie beispielsweise die Unterschiede zweier verschiedener Versionen des Datenmodells einer Datenbank vergleichen.

Die Lösung zeigt Ihnen, welche Tabellen oder Felder hinzugekommen oder weggefallen sind und wie sich die Eigenschaften von Tabellen und Feldern verändert haben (s. **Datenmodelle vergleichen** ab S. 61).

Wer viel mit Access programmiert, sollte außerdem alle wichtigen Informationen rund um die Anwendungsdatei MSAccess.exe kennen. Wie öffne ich es in verschiedenen Modi, beispielsweise im Kontext eines anderen Benutzers oder als Administrator, wie starte ich Access im Runtime-Modus? Wie ermittle ich wichtige Systemverzeichnisse wie das Add-In-Verzeichnis oder das Benutzerverzeichnis? Unter **MSAccess.exe und Co.** ab S. 2 erfahren Sie außerdem, wie Sie die eingebauten Assistenten manipulieren können.

Wer Abfragen mit dynamischen Kriterien nicht immer als SQL-Ausdruck per VBA zusammenstellen möchte, verwendet Parameterabfragen. Unter **Parameterabfragen unter der Lupe** erfahren Sie ab S. 10 alles rund um das Thema Parameterabfragen. Dort lernen Sie weiterhin zwei praktische Funktionen kennen, mit denen Sie Recordsets auf Basis von Parameterabfragen ermitteln oder Aktionsabfragen mit Abfrageparametern ausführen.

Der Beitrag **Aktionsabfragen statt Schleifen** widmet sich ab S. 20 der Optimierung von Vorgängen, die Änderungen an mehreren Datensätzen bewirken sollen. Wer des SQL nicht mächtig ist, bemüht für solche Aufgaben gern die Methoden der DAO-Bibliothek und führt die gewünschten Änderungen in einer **Do While**-Schleife über die betroffenen Datensätze aus. Der Beitrag zeigt eindrucksvoll, dass es sich lohnt, hier ein wenig Hirnschmalz zu investieren: In unserem Beispiel konnten wir die Performance einer Änderung von rund 1.500 Datensätzen um 98% verbessern.

Verknüpfte Daten präsentieren Sie dem Benutzer in der Regel in Haupt- und Unterformular. An dieser Stelle kann dieser nicht wissen, dass Änderungen an den Daten im Unterformular schon gespeichert werden, obwohl er das Speichern noch gar nicht bewusst angestoßen hat.

Wir haben eine Klasse entwickelt, mit der Sie die Änderungen am aktuellen Datensatz im Hauptformular und den dazugehörigen Daten im Unterformular komplett rückgängig machen können. Und das Beste ist: Die Klasse lässt sich ganz einfach in Ihre Formulare integrieren (s. **Undo in Haupt- und Unterformular mit Klasse** ab S. 31).

Die Verweise eines VBA-Projekts spielen eine große Rolle, denn darüber referenzieren Sie Bibliotheken und greifen auf ihre Objekte, Methoden, Eigenschaften und Ereignisse zu. Dumm nur, dass der eingebaute Verweise-Dialog beispielsweise die Pfade zu den meisten Bibliotheken nicht komplett anzeigt. Im Beitrag **Bibliotheken und Verweise untersuchen** erfahren Sie ab S. 44, wie Sie diesen Umstand ändern können.

Für den Profi sind VBA-Funktionen das tägliche Brot, für den Einsteiger oder Poweruser gibt es aber immer neue Erkenntnisse rund um diese Prozedur-Art. Der Beitrag **VBA-Funktionen von A bis Z** liefert die wichtigsten Informationen rund um dieses Thema (ab S. 53).

Viel Spaß mit der neuen Ausgabe!

Ihr Michael Forster

MSAccess.exe und Co.

Access besteht nicht nur aus einer einzigen Anwendungsdatei, sondern auch noch aus einer Reihe weiterer wichtiger Dateien wie etwa den Bibliotheken VBA, DAO oder ADODB oder Add-In-Datenbanken, die Funktionen zu Access hinzufügen. In diesem Beitrag sehen wir uns diese Dateien an und schauen, welche Informationen sich aus diesen gewinnen lassen. Außerdem gibt es eine Reihe VBA-Funktionen, mit denen Sie beispielsweise die Speicherorte dieser Dateien ermitteln können.

MSAccess.exe

Die Datei **MSAccess.exe** wird aufgerufen, wenn der Benutzer Access startet – sei es über einen Eintrag im Startmenü, über einen zur Taskleiste hinzugefügten Eintrag oder auch durch das Starten einer Datei mit einer entsprechenden Dateiendung wie beispielsweise **.mdb**, **.accdb** et cetera.

Sie liegt in der Version Access 2010, 32bit unter Windows 7 im Verzeichnis **C:\Program Files (x86)\Microsoft Office\Office14**, in anderen Versionen von Access unterscheidet sich der Name des letzten Verzeichnisses (unter Access 2013 etwa Office15).

Wenn Sie die 64bit-Variante installieren, was nicht empfehlenswert ist (hier werden beispielsweise keine der ActiveX-Steuerelemente der Bibliothek **MSCOMCTL.ocx** unterstützt), landet diese im Verzeichnis **C:\Program Files\Microsoft Office\Office14**.

MSAccess.exe suchen

Wenn Sie die Datei **MSAccess.exe** nicht finden, können Sie diese mit einem VBA-Befehl ermitteln, den Sie beispielsweise im Direktbereich des VBA-Editors absetzen:

```
Debug.Print SysCmd(acSysCmdAccessDir)
C:\Program Files (x86)\Microsoft Office\Office14\
```

Access-Version ermitteln

Wenn Sie schon die Funktion **SysCmd** nutzen, können Sie auch gleich die Version der geöffneten Access-Anwendung ausgeben lassen:

```
Debug.Print SysCmd(acSysCmdAccessVer)
14.0
```

Benutzer und Gruppen

In früheren Access-Versionen (bis Access 2003) konnten Sie noch eine Access-interne Benutzer- und Gruppenverwaltung verwenden. Daher nur der Vollständigkeit halber hier die Funktion zum Ermitteln des Standortes der aktuell verwendeten Arbeitsgruppeninformationsdatei, die solche Informationen speicherte:

```
Debug.Print SysCmd(7
                acSysCmdGetWorkgroupFile)
C:\Users\Andre\AppData\Roaming\Microsoft\Access\System.mdw
```

Wie man sieht, gibt es diese Datei immer noch – Sie kann allerdings nicht mehr zur Sicherung der Access-Objekte und Daten genutzt werden (sicher war diese ohnehin schon längst nicht mehr). Wer seine Daten wirklich sichern will, verwendet ein System wie den Microsoft SQL Server oder MySQL.

Runtime oder Vollversion?

Für eine Anwendung kann es wichtig sein, neben der Version einer Access-Anwendung auch noch zu erkennen, ob es sich um die Vollversion oder um die Runtime-Version handelt. Wenn Sie beispielsweise zur Laufzeit dynamisch Formulare oder Berichte anpassen, benötigen Sie unbedingt die Vollversion – unter der Runtime-Version können Sie keine Änderungen am Entwurf der Datenbankobjekte vornehmen. Daher kommt uns die folgende Funktion sehr gelegen:

```
Debug.Print SysCmd(acSysCmdRuntime)
Falsch
```

Access als Runtime starten

Apropos Runtime: Sie müssen nicht die Runtime-Version von Access installieren, um den Großteil der Runtime-Einschränkungen zu testen. Dazu reicht je nach Access-Version auch eine der folgenden Aktionen aus:

- Bis Access 2003 rufen Sie Access per Befehlszeile mit dem Parameter **/runtime** auf.
- Ab Access 2007 ändern Sie die Dateiendung der Datenbankdatei von **.accdb** in **.accdr** um.

Für den Aufruf des Runtime-Modus über die Befehlszeile geben Sie erst den Pfad zur **MSAccess.exe**, dann den Pfad zu der zu öffnenden Datenbankdatei und schließlich den Befehlszeilenparameter an:

```
"C:\Program Files (x86)\Microsoft Office\Office14\MSAccess.exe" "c:\Daten\Fachmagazine\AccessImUnternehmen\2013\06\Accessfunktionen\Accessfunktionen.mdb" /Runtime
```

Dafür müssen Sie übrigens nicht umständlich eine Windows-Verknüpfung anlegen. Eine Datei mit der Dateiendung **.bat** (etwa **Start.bat**) reicht völlig aus.

Access erscheint dann in der für die Runtime üblichen spartanischen Ausstattung, die nach benutzerdefinierten Menüleisten/Ribbons oder einem Übersichtsformular zum Aufruf der Formulare und Funktionen verlangt (s. Bild 1).

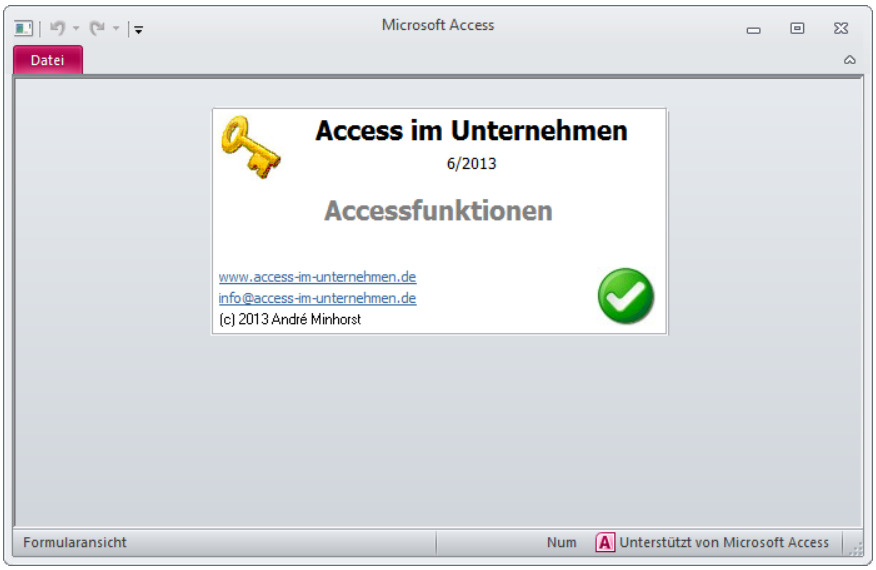

Bild 1: Start der Runtime-Version von Access

Access als Administrator starten

Wenn Sie Access mit Administratorrechten starten müssen, obwohl Sie aktuell nicht als Administrator angemeldet sind, können Sie dies über eine entsprechende Verknüpfung oder über die Datei **MSAccess.exe** selbst erledigen.

Klicken Sie einfach bei gedrückter Umschalttaste mit der rechten Maustaste auf die Datei und wählen Sie dann den Eintrag **Als Administrator ausführen** aus dem Kontextmenü aus. Access startet nun mit Administratorrechten (s. Bild 2).

Access als anderer Benutzer starten

Gelegentlich wollen Sie vielleicht testen, wie Ihre Anwendung arbeitet, wenn Sie sich unter einem anderen Windows-Konto anmelden. Dann heißt es, den aktuellen Benutzer abmelden, sich unter dem Konto des neuen Benutzers anmelden, die Tests ausführen, wieder abmelden und unter dem zunächst verwendeten Konto wieder anmelden.

Das machen Sie allerdings nur so, wenn Sie zu viel Zeit haben. Wenn Ihnen die Prozedur zu aufwendig ist, können Sie sich nämlich über einen weiteren Kontextmenü-Eintrag namens **Als anderer Benutzer ausführen** unter dem anderen Benutzerkonto anmelden und Access in diesem Kontext ausführen.

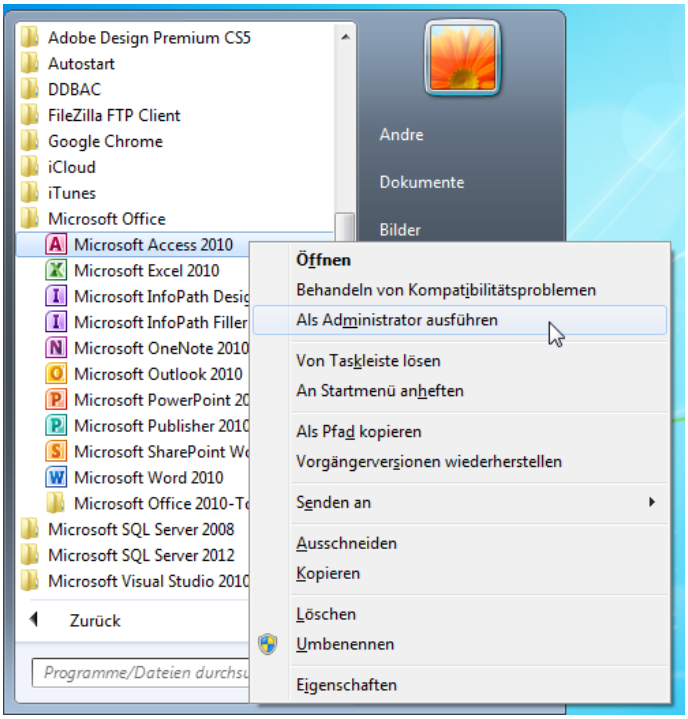

Bild 2: Access mit Administratorrechten starten

Aber wo steckt dieser Kontextmenü-Eintrag – im Screenshot des vorherigen Beispiels ist dieser doch nicht zu finden?

Nun: Dieser Eintrag erscheint nur, wenn Sie das Kontextmenü der Datei **MSAccess.exe** selbst bei gedrückter Umschalttaste aufrufen.

Das gelingt beispielsweise über das Startmenü, wenn Sie im Suchen-Feld den Begriff **MSAccess.exe** eingeben (s. Bild 3).

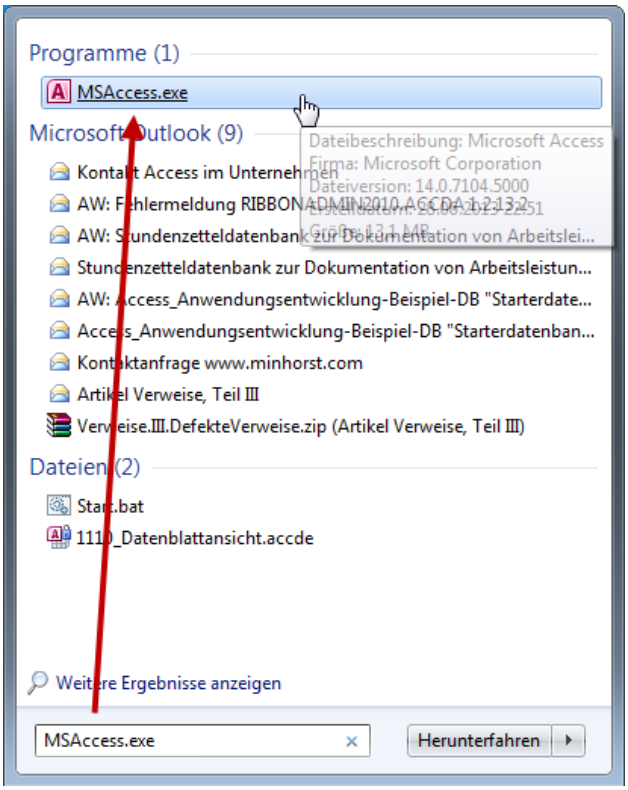

Bild 3: **MSAccess.exe** über die Suche im Startmenü auffinden

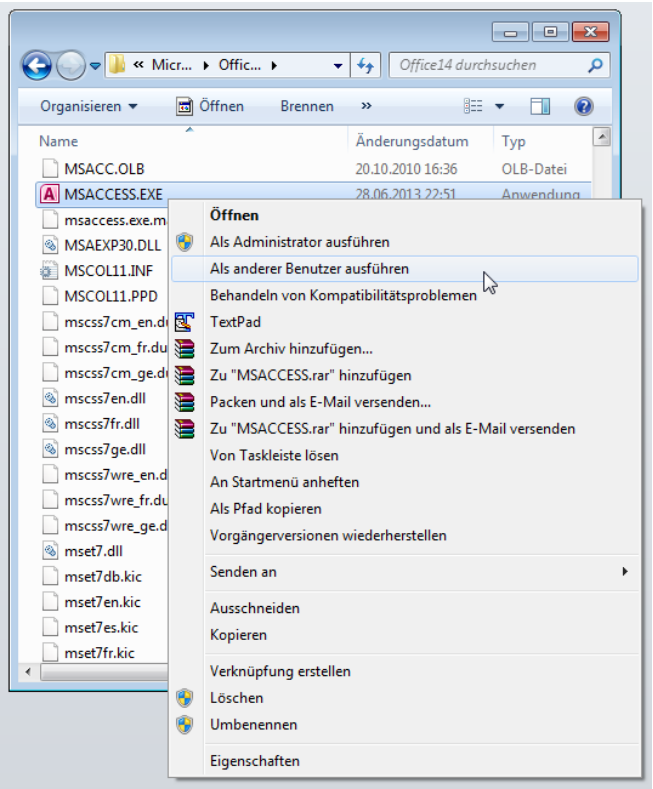

Bild 4: Starten von Access als anderer Benutzer über das Kontextmenü von **MSAccess.exe** im Windows Explorer

Alternativ navigieren Sie im Windows Explorer zu der entsprechenden Datei (s. Bild 4).

Anschließend erscheint dann ein Anmeldedialog, mit dem Sie Benutzername und Kennwort des betroffenen Benutzers eingeben (s. Bild 5). Anschließend wird Access im Kontext dieses Benutzers gestartet.

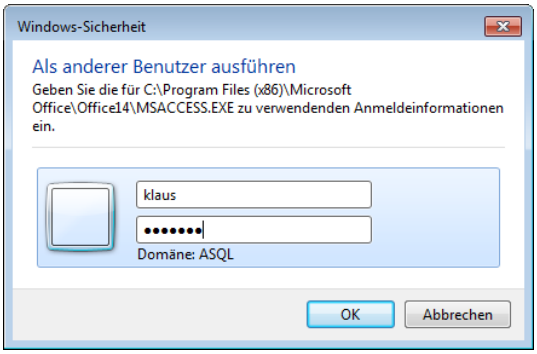

Bild 5: Benutzer und Kennwort zum Starten von Access eingeben

Aktuelle Datenbankdatei

Auch zur aktuellen Datenbankdatei können Sie per VBA eine ganze Reihe Informationen herauskitzeln.

Die Access-Bibliothek bietet einige Informationen, aber auch die DAO-Bibliothek bietet Möglichkeiten für den Zugriff etwa auf den Pfad zur aktuell geöffneten Access-Datenbank.

Bis mit Access 2002 das **CurrentProject**-Objekt eingeführt wurde, lieferte allein das **CurrentDb**-Objekt Informationen über die Herkunft der Datenbankdatei. Die Eigenschaft **Name** lieferte nämlich den kompletten Pfad der mit **CurrentDb** referenzierten, also der aktuell geöffneten Datenbankdatei:

```
Debug.Print CurrentDb.Name
C:\Daten\Fachmagazine\AccessImUnternehmen\2013\06\Accessfunktionen\Accessfunktionen.mdb
```

Wer bis Access 2002 nur den Namen oder den Pfad ermitteln wollte, musste sich mit einer Hilfsfunktion begnügen.

Den Namen einer Datei auf Basis des kompletten Pfades liefert bekanntlich die **Dir**-Funktion:

```
? Dir(CurrentDb.Name)
Accessfunktionen.mdb
```

Wenn man von der mit **CurrentDb.Name** ermittelten Zeichenkette hinten eine Zeichenkette abschneidet, deren Länge der Länge des Dateinamens entspricht, erhält man das Verzeichnis inklusive abschließendem Backslash:

Parameterabfragen unter der Lupe

Abfragen lassen sich auf unterschiedlichste Weise mit den Vergleichswerten für ihre Kriterien bestücken – beispielsweise, indem man einfach den gewünschten SQL-Ausdruck per VBA zusammensetzt. Es gelingt jedoch unter Einsatz von Abfrageparametern: Sie fügen in eckige Klammern eingefasste Bezeichnungen in den Abfrageentwurf ein und füllen diese dann entweder per eingebauter Inputbox oder per VBA. Dieser Beitrag stellt alle wichtigen Techniken rund um den Einsatz von Parameterabfragen vor.

Warum Parameterabfragen?

In vielen Fällen benötigen Sie keine Parameterabfragen, weil die Abfrage entweder bereits die notwendigen Kriterien enthält oder diese nachträglich per Filter appliziert werden. Dies kann beispielsweise der Fall sein, wenn Sie einem Formular eine komplette Tabelle als Datenherkunft zuweisen, dann aber etwa mit der **OpenForm**-Methode des **DoCmd**-Objekts einen entsprechenden Filterausdruck übergeben. Dieser wird dann beim Öffnen des Formulars angewendet.

In manchen Fällen kommen Sie damit aber nicht weiter: Wenn Sie etwa eine Aktualisierungsabfrage erstellen, die ein oder mehrere Kriterien enthält, können Sie die Vergleichswerte nur auf zwei Arten anwenden: Entweder, indem Sie den SQL-Ausdruck der Abfrage in eine VBA-Variable laden und dieser die entsprechenden Kriterien hinzufügen oder eben mit Abfrageparametern.

Bestimmte Anwendungsfälle erfordern es auch, dass man mehrere aufeinander aufbauende Abfragen nutzt. Das kann etwa sein, wenn Sie zunächst die Daten einer Abfrage mit **OUTER JOIN**-Verknüpfung ermitteln und das Ergebnis dann in einer weiteren Abfrage mit den Daten aus weiteren Tabellen verknüpfen. In diesem Fall kommen Sie kaum um den Einsatz von Abfrageparametern herum. Sie könnten zwar noch den Abfragetext der Ausgangsabfrage per VBA anpassen und erst dann die andere Abfrage darauf zugreifen lassen, aber das wäre doch etwas umständlich.

Darüber hinaus haben Parameterabfragen den Vorteil, dass Access für diese nach der ersten Ausführung einen Ausführungsplan erstellt. Das bedeutet, dass ein Plan aufgestellt, mit dem das Ergebnis optimal ermittelt wird. Wenn Sie die gleiche Abfrage anschließend nochmals aufrufen, kann diese auf den bereits ermittelten Ausführungsplan zurückgreifen. Wenn Sie eine Abfrage bei jedem Aufruf erneut per VBA zusammenstellen, muss auch der Ausführungsplan mit jedem Aufruf erneut erstellt werden.

Parameterabfrage erstellen

Eine Parameterabfrage unterscheidet sich nicht wesentlich von einer herkömmlichen Abfrage. Der Unterschied ist, dass beispielsweise ein Kriteriumsausdruck nicht direkt beim Erstellen mit dem Vergleichswert für diese Abfrage ausgestattet wird, sondern stattdessen einen in eckigen Klammern eingefassten Parameter aufnimmt.

Dies sieht beispielsweise so wie in Bild 1 aus. Wenn Sie die Abfrage nun ausführen, zeigt diese alle Datensätze der Tabelle **tblArtikel** an, deren Feld **KategorieID** den Wert **1** enthält. Dies wollen wir nun flexibler gestalten: Der Benutzer soll diesen Wert beim Aufruf der Abfrage

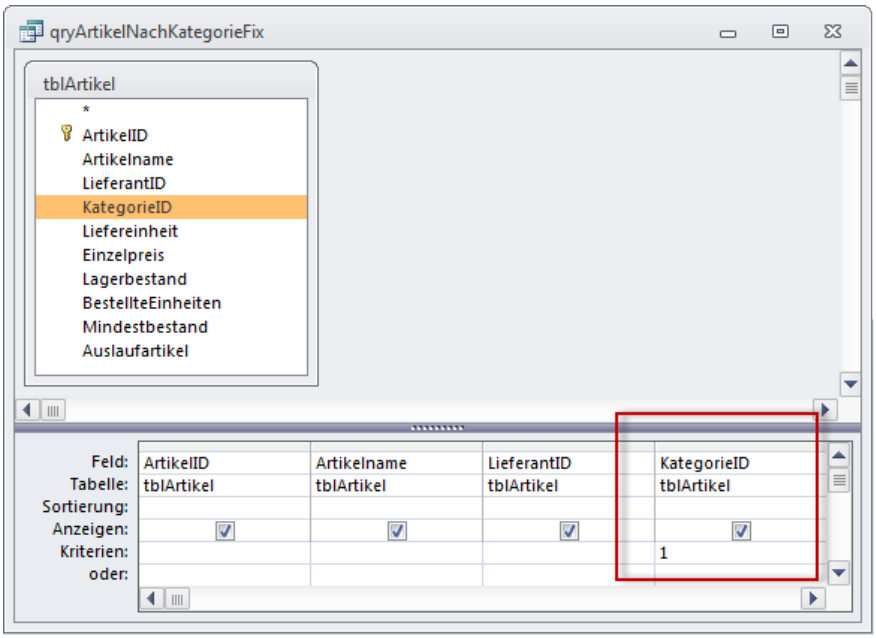

Bild 1: Abfrage mit festem Vergleichswert

selbst eingeben können. Dazu ersetzen Sie den Wert **1** einfach durch den Ausdruck **[VergleichskategorieID]**.

Wenn Sie die Abfrage dann ausführen, zeigt Access einen Dialog zur Eingabe des Vergleichswertes an (s. Bild 2). Dieser erhält als Titel den Text **Parameterwert eingeben** und als Meldungstext den in eckigen Klammern angegebenen Ausdruck. Hier ist zu beachten, dass der Text in eckigen Klammern, also der Name des Parameters, nicht den Namen eines der Felder der Tabellen der Datenherkunft enthalten darf. In diesem Fall wird der Parameter mit dem jeweiligen Wert der Datenherkunft gefüllt.

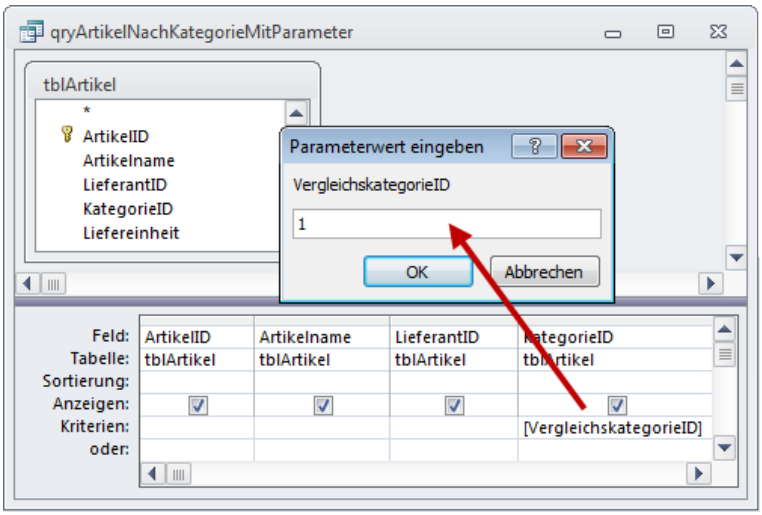

Bild 2: Abfrage mit Parameter als Vergleichswert

Bei der Benennung der Parameter gibt es sonst kaum Einschränkungen – Sie dürfen allerdings keinen Punkt und kein Ausrufezeichen verwenden. Dies führt direkt zu einer entsprechenden Meldung (s. Bild 3).

Wenn Sie den Parameter tatsächlich über den dafür vorgesehenen Dialog vom Benutzer abfragen möchten, können Sie allerdings auch komplette Sätze unterbringen – nur ohne Punkt. Aber das ist kein Problem – Sie können einen solchen Satz ja auch mit einem Doppelpunkt abschließen:

```
[Geben Sie die KategorieID ein:]
```

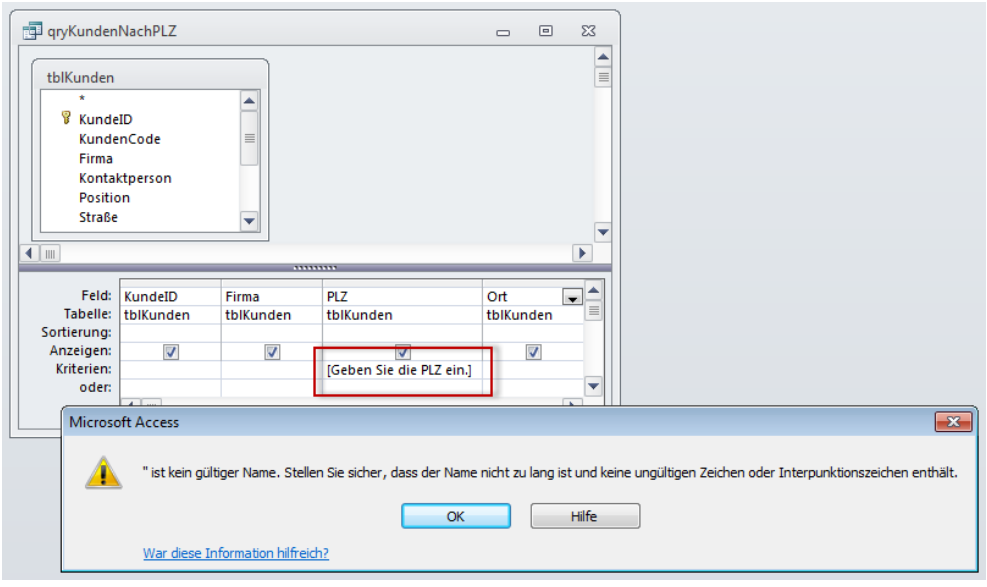

Bild 3: Abfrageparameter mit ungültigem Zeichen

Zeichenketten und Parameter

Wenn Sie eine Zeichenkette mit einem Parameter vergleichen wollen, gehen Sie genauso vor wie mit einem Zahlenwert wie im obigen Beispiel. Allerdings möchten Sie den Parameter ja vielleicht als Teil des Vergleichsausdrucks einsetzen.

Auch das ist kein Problem: Sie müssen den Abfrageparameter lediglich per Und-Zeichen mit den übrigen Elementen verbinden. Wenn der Benutzer etwa Kundendatensätze anhand der Postleitzahl eingrenzen soll, können Sie ihm folgendes Kriterium anbieten:

```
Wie [Geben Sie die PLZ ein:] & "*"
```

Dieses fragt wie in Bild 4 eine Zeichenkette ab und hängt an diese noch das Sternchen (*) als Platzhalter an. Die Abfrage liefert dann alle Datensätze der Tabelle **tblKunden**, deren Feld **PLZ** mit dem als Parameter übergebenen Wert übereinstimmt.

Datentypen für Parameter festlegen

Aktuell kann der Benutzer für die Parameter alle denkbaren Werte eingeben. So landet dann auch schon mal eine Zeichenkette in einem Parameter, der eigentlich ein Datumsfeld betrifft. Dies führt natürlich zu einer

entsprechenden Fehlermeldung, die – wie in vielen Fällen – für Otto Normalverbraucher wenig Aussagekraft besitzt. Wenn Sie sicherstellen wollen, dass der Benutzer einen Wert als Parameter übergibt, der einem bestimmten Datentyp entspricht, müssen Sie den Parameter in einem separaten Dialog deklarieren. Diesen Dialog rufen Sie auf, indem Sie das Kontextmenü des Abfrageentwurfs anzeigen und dort den Eintrag **Parameter** auswählen (s. Bild 5).

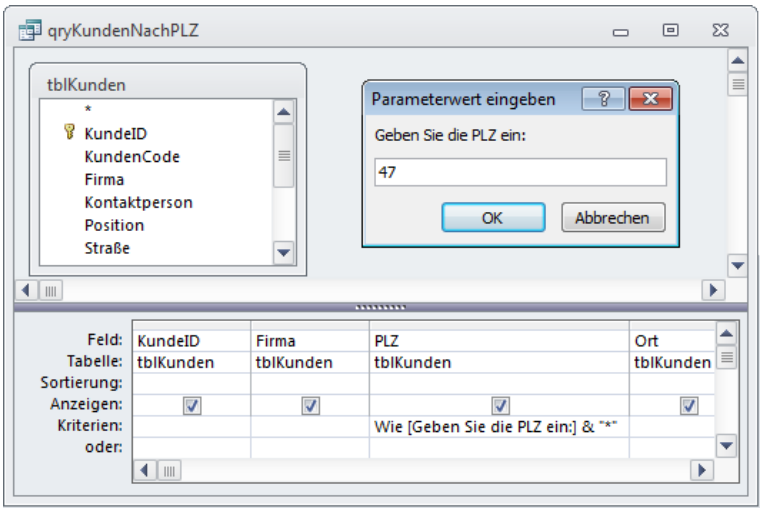

Bild 4: Abfrageparameter als Teil des Vergleichswertes

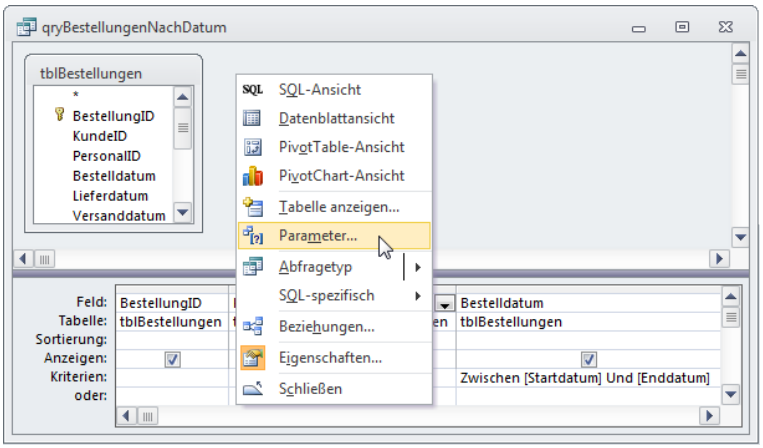

Bild 5: Anzeigen des Dialogs zum Deklarieren von Parametern

In diesem Fall wollen wir dafür sorgen, dass der Benutzer für die beiden Parameter **Startdatum** und **Enddatum** nur Datumswerte eintragen darf. Der Dialog, den Sie mit dem Kontextmenüeintrag Parameter anzeigen, enthält leider noch nicht automatisch die bereits im Kriterium angegebenen Abfrageparameter. Sie müssen diese also manuell dort eintragen und dann den Datentyp festlegen. Dabei stellen wir fest, dass die Datentypen nicht exakt mit denen im Tabellenentwurf übereinstimmen. Die Zahlendatentypen etwa werden hier direkt mit der entsprechenden Feldgröße angegeben, also **Byte**, **Long** et cetera (s. Bild 6). Wir tragen hier die beiden Parameter **Startdatum** und **Enddatum** ein und legen als Felddatentyp jeweils **Datum/Uhrzeit** fest.

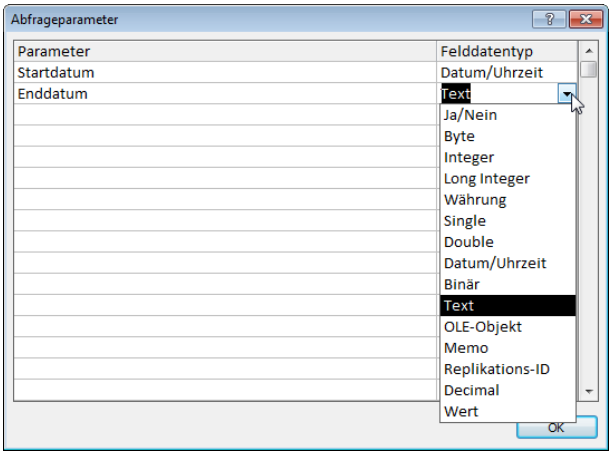

Bild 6: Der Dialog zum Anzeigen von Parametern

Wenn der Benutzer dann einen falschen Wert einträgt, erscheint die Meldung **Sie haben einen Wert eingegeben, der für dieses Feld nicht gültig ist** – also die gleiche Meldung, die auch bei der Eingabe falscher Daten in Tabellenfelder erscheint.

Davon abgesehen arbeiten Abfrageparameter, die auch als solche deklariert wurden, genau wie nicht deklarierte Abfrageparameter.

Nun wird es Zeit, einen Blick auf den SQL-Code für eine Abfrage mit deklarierten Parametern zu werfen. Die Parameter werden hier in einer eigenen Zeile vor Beginn der eigentlichen SQL-Anweisung mit dem Schlüsselwort **PARAMETERS** unter Angabe des entsprechenden Datentyps deklariert:

```
PARAMETERS Startdatum DateTime, Enddatum
DateTime;
SELECT BestellungID, KundeID, Persona-
lID, Bestelldatum
FROM tblBestellungen
WHERE Bestelldatum Between [Startdatum]
And [Enddatum];
```

Sollten die Namen der im Kriterium untergebrachten Parameter nicht mit denen im **Parameter**-Dialog übereinstimmen, fragt Access alle verschiedenen Versionen ab.

Wenn Sie also einen Parameter namens **Datum eingeben:** im Abfrageentwurf unterbringen und im **Parameter**-Dialog einen Parameter namens **Datumeingabe:** definieren, erscheinen Abfragedialoge für beide Varianten. Allerdings wirkt sich nur derjenige, der auch im Kriterium eingesetzt wird, auf das Abfrageergebnis aus.

Parameterabfragen per VBA

Für einen Schnellschuss kann man den Benutzer mal mit den von Access erzeugten Dialogen zum Abfragen von Parametern konfrontieren, aber in der Regel sollte man die Werte per Formular abfragen. Dort kann man diese gegebenenfalls vorbelegen, eine eigene Fehlerbehandlung für den Fall der Eingabe ungültiger Werte unterbringen et cetera. Also schauen wir uns an, wie man unter VBA mit Parameterabfragen umgeht.

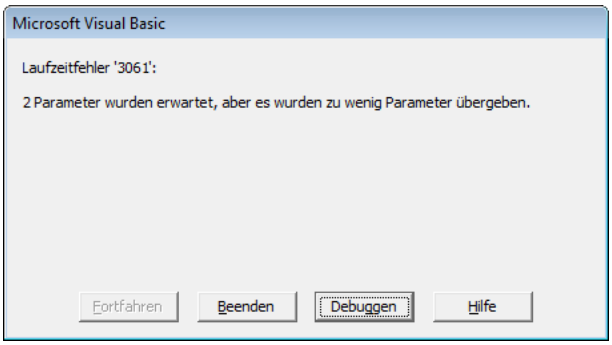

Bild 7: Fehlermeldung beim Versuch, eine Abfrage mit Parametern per **OpenRecordset** zu öffnen

```
Public Sub ParameterabfrageMitFehler()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set rst = db.OpenRecordset("qryBestellungenNachDatum")
    Do While Not rst.EOF
        rst.MoveNext
    Loop
End Sub
```

Listing 1: Dieser Zugriff auf die Parameterabfrage löst einen Fehler aus.

```
Public Sub Parameterabfrage()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim prmStart As DAO.Parameter
    Dim prmEnde As DAO.Parameter
    Dim rst As DAO.Recordset
    Set db = CurrentDb
    Set qdf = db.QueryDefs("qryBestellungenNachDatum")
    Set prmStart = qdf.Parameters("Startdatum")
    prmStart.Value = "1.1.2012"
    Set prmEnde = qdf.Parameters("Enddatum")
    prmEnde.Value = "31.1.2012"
    Set rst = qdf.OpenRecordset
    Do While Not rst.EOF
        Debug.Print rst!BestellungID
        rst.MoveNext
    Loop
End Sub
```

Listing 2: Parameterabfrage mit per VBA zugewiesenen Parametern

Wenn Sie die Abfrage etwa als Recordset öffnen und durchlaufen möchten und annehmen, dass Access wie beim manuellen Öffnen den Dialog zum Eingeben der Parameter anzeigt, probieren Sie einmal die Prozedur aus Listing 1 aus.

Das Ergebnis ist die Fehlermeldung aus Bild 7. Access zeigt also keine Eingabedialoge an, wenn Sie eine Parameterabfrage als Recordset öffnen. Wie also bringen Sie die Parameterwerte in der Abfrage unter?

VBA und QueryDef

Dazu benötigen Sie zwei weitere DAO-Objekte, nämlich das **QueryDef**-Objekt, mit dem Sie auf Abfragen zugreifen und ihre Eigenschaften verändern können, sowie das **Parameter**-Objekt, das den Zugriff auf die Parameter einer Abfrage erlaubt. Für das Beispiel der Abfrage **qryBestellungenNachDatum** benötigen wir eigentlich sogar zwei Parameter-Objekte, die wir als **prmStart** und **prmEnde** deklarieren. Theoretisch würde zwar eine Variable reichen (**prm**), aber wir wollen für eine bessere Übersicht für jeden Parameter eine eigene Variable

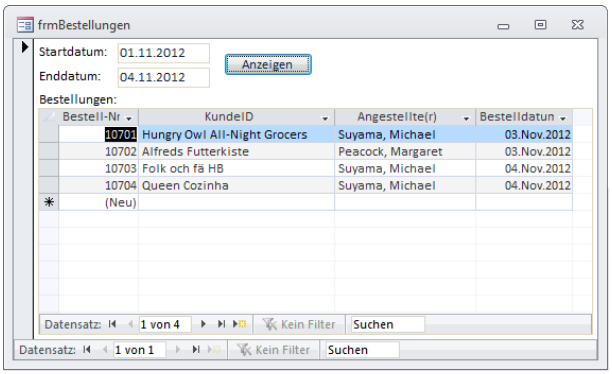

Bild 8: Eingabe der Parameter per Textfeld

einsetzen. Der Einsatz von Parametern in VBA sieht wie in Listing 2 aus.

Neben den beiden **Parameter**-Variablen deklarieren wir dort eine **QueryDef**-Variable namens **qdf** sowie ein **Database**-Objekt und ein **Recordset**-Objekt. Das **Database**-Objekt **db** füllen wir mit einem Verweis auf die aktuelle Datenbank. Dann referenzieren wir die Abfrage **qryBestellungenNachDatum** mit dem **QueryDef**-Objekt **qdf**, das wir über den Namen der Abfrage aus der **QueryDefs**-Auflistung des **Database**-Objekts beziehen.

Und natürlich müssen wir noch die beiden **Parameter**-Variablen füllen. Dies erledigen wir über die **Parameters**-Auflistung, deren Elemente wir beispielsweise direkt über den Namen ansprechen können – also **Startdatum** und **Enddatum**.

Das **Parameter**-Objekt bietet eine Eigenschaft namens **Value**, mit der wir den Parameterwert übergeben können. Schließlich erstellen wir ein **Recordset**-Objekt auf Basis des mit den entsprechenden Parametern versehenen **QueryDef**-Objekts **qdf** und durchlaufen zum Test dessen Datensätze.

```
Private Sub cmdAnzeigen_Click()
    Set Me!sfmBestellungen.Form.Recordset = _
        BestellungenNachDatum(Me!txtStartdatum, Me!txtEnddatum)
End Sub
```

Listing 3: Füllen des Unterformulars per Recordset aus einer benutzerdefinierten Funktion

```
Public Function BestellungenNachDatum(datStart As Date, datEnde As Date) _
        As DAO.Recordset
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim prmStart As DAO.Parameter
    Dim prmEnde As DAO.Parameter
    Set db = CurrentDb
    Set qdf = db.QueryDefs("qryBestellungenNachDatum")
    Set prmStart = qdf.Parameters("Startdatum")
    prmStart.Value = datStart
    Set prmEnde = qdf.Parameters("Enddatum")
    prmEnde.Value = datEnde
    Set BestellungenNachDatum = qdf.OpenRecordset
End Function
```

Listing 4: Zusammenstellung des Recordsets mit den angegebenen Kriterien

Parameter aufnehmen

Im nächsten Beispiel bieten wir dem Benutzer eine Möglichkeit, um die beiden Parameter einzugeben und gleich die gesuchten Datensätze anzuzeigen. Dazu erstellen Sie ein Hauptformular namens **frmBestellungen**, das die beiden Textfelder **txtStartdatum** und **txtEnddatum** enthält, sowie eine Schaltfläche namens **cmdAnzeigen** (s. Bild 8). Das Unterformular enthält zunächst die Daten der Tabelle **tblBestellungen**. Beim Laden belegt das Formular die beiden Textfelder vor:

```
Private Sub Form_Load()
    Me!txtEnddatum = Date
    Me!txtStartdatum = DateAdd("yyyy", 7
                               -1, Date)
End Sub
```

Ein Klick auf die Schaltfläche **cmdAnzeigen** sorgt schließlich für das Einlesen eines Recordsets auf Basis der Abfrage **qryBestellungenNachDatum** und der beiden angegebenen Datumsangaben (s. Listing 3).

Die Funktion **BestellungenNachDatum** erwartet schließlich die beiden Datumsangaben als Parameter (s. Listing 4). Sie arbeitet prinzipiell wie die Prozedur Parameterabfrage aus dem vorherigen Beispiel, gibt das **Recordset**-Objekt jedoch als Funktionswert zurück. Dieses wird dann direkt der **Recordset**-Eigenschaft des Unterformulars zugewiesen. Achtung: Dies führte in der Beispieldatenbank zu nicht reproduzierbaren Abstürzen, wenn man das Recordset mehrfach nacheinander aktualisiert hat.

Parameter auslesen

Die Auflistung **Parameters** des **QueryDefs**-Objekts haben Sie ja bereits genutzt, um über den Namen auf die

Aktionsabfragen statt Schleifen

Änderungen an den Daten in den Tabellen einer Datenbank kann man auf verschiedenste Arten durchführen – zum Beispiel über die Benutzeroberfläche. Gelegentlich werden Sie jedoch auch Daten per VBA ändern wollen. In diesem Falle gibt es eine ganze Reihe von Varianten, die normalerweise mit den Methoden der DAO-Bibliotheken abgebildet werden: Das Öffnen eines Recordsets und Ändern der enthaltenen Daten mit AddNew/Edit und Update, das Zusammenstellen einer SQL-Aktionsabfrage, die dann mit der Execute-Methode des Database-Objekts ausgeführt wird, oder der Einsatz gespeicherter Aktionsabfragen, die man mit der Execute-Methode des QueryDef-Objekts startet. Wir schauen uns die verschiedenen Varianten an und optimieren diese.

Performance und Stil

Der Grund für diesen Beitrag ist offensichtlich: Die verschiedenen Methoden zum Ändern von Daten unterscheiden sich stark in der Performance. Wenn Sie eine Schleife über eine Datensatzgruppe durchlaufen und daraus entnommene Daten mit den DAO-Methoden **AddNew** und **Update** zu einem weiteren Recordset hinzufügen, dauert das schlicht und einfach länger als wenn Sie die gesamte Prozedur in einer einzigen Aktionsabfrage unterbringen.

Und das ist nicht nur das Bild, das man bei der Unterstützung anderer Entwickler vorfindet, sondern man selbst neigt ebenso dazu, erstmal den scheinbar einfacheren Weg über DAO zu wählen, bevor man auf den ersten Blick kompliziert erscheinende Aktionsabfragen zusammenstellt.

Und damit landen wir gleich beim anderen Aspekt: In vielen Fällen kostet das Entwickeln einer eleganten, performanten Lösung schlicht und einfach mehr Zeit als das stumpfe Herunterprogrammieren bekannter Codestrukturen – zumindest wenn man nicht über ausreichend Programmiererfahrung verfügt.

Dann hat man nämlich oft genug erst Datensatz für Datensatz mit DAO durchpflügt, bevor man den ersten Entwurf dann in einem oder mehreren Schritten zu einer zufriedenstellenden Lösung weiterentwickelt hat, die sowohl den Anspruch an die Performance als auch an die Eleganz befriedigt.

Eine solche Entwicklung können Sie in der Lösung zum Beitrag **Datenmodelle vergleichen** (**www.access-im-unternehmen.de/916**) nachvollziehen. Im vorliegenden Beitrag schauen wir uns ein etwas einfacheres Beispiel an und führen die Schritte vom ersten Ansatz bis zur perfekten Lösung durch.

Beispiel: Kundendaten archivieren

Im Beispiel dieses Beitrags wollen wir einfach bestimmte Kundendaten in eine andere Tabelle kopieren – beispielsweise, um diese zu archivieren. Die Kundendaten sollen nach bestimmten Kriterien ausgewählt werden – beispielsweise nach dem Zeitpunkt der letzten Bestellung. Liegt diese mehr als ein Jahr zurück, soll der Kunde in eine entsprechende Tabelle kopiert werden.

Außerdem soll die neue Tabelle noch ein weiteres Feld mit dem Datum der Archivierung enthalten (s. Bild 1).

Daten mit bestimmten Kriterien aus einer Tabelle in eine weitere Tabelle kopieren, die überdies noch um weitere Daten ergänzt werden soll – das hört sich sehr kompliziert an. Also gehen wir das Ganze mal schön langsam Schritt für Schritt an – so, dass man zu jeder Zeit kontrollieren kann, was dort geschieht.

Start der Archivierung

Die Archivierung soll über Schaltflächen eines Formulars namens **frmKundenArchivieren** gestartet werden (s. Bild 2).

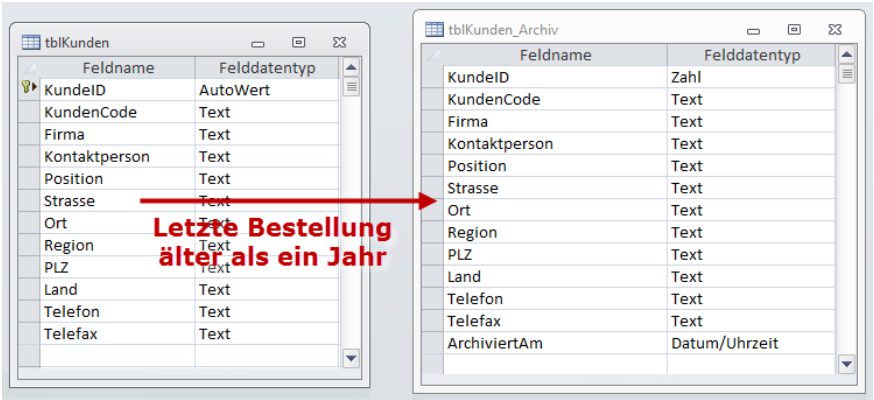

Bild 1: Quell- und Zieltabelle

Dieses Formular ruft die verschiedenen Varianten der Archivierung auf und übergibt dabei jeweils das im Textfeld **txtStichtag** gespeicherte Datum an die **Funktionen**, für die erste Schaltfläche etwa so:

```
Private Sub cmd1_Click()
    Dim intAnzahl As Integer
    intAnzahl = Datenkopieren_7
          DAOMitAbfrage(Me!txtStichtag)
    MsgBox intAnzahl _
        & " Datensätze archiviert"
End Sub
```

Die „sichere" Variante

Wer noch nicht erfahren im Umgang mit Access ist, hat vielleicht noch nicht mit SQL gearbeitet und wird dementsprechend zunächst eine Abfrage wie oben beschrieben erstellen.

Dann wird er eine VBA-Funktion bauen, die wie die in Listing 1 aussieht. Sie sucht nach Kunden, die seit dem per Parameter übergebenen Stichtag keine Bestellung mehr durchgeführt haben und archiviert werden sollen, und liefert die Anzahl der betroffenen Datensätze zurück.

Die Funktion erstellt ein **Database**-Objekt namens **db** sowie zwei Datensatzgruppen namens **rstQuelle** und **rstZiel**. Das **Recordset**-Objekt **rstQuelle** wird mit einem Verweis auf alle Datensätze der Tabelle **tblKunden** gefüllt, das **Recordset**-Objekt **rstZiel** mit einer Datenherkunft, die auf der Zieltabelle **tblKunden_Archiv** basiert, aber wegen des Kriteriums **1=2** keine Datensätze enthält.

Die Prozedur durchläuft nun alle Datensätze des Recordsets **tblKunden**. Dabei ermittelt sie zunächst mit dem Aufruf von **DLookup** das aktuellste Datum einer Bestellung aus der Tabelle **tblBestellungen** für die Datensätze, deren Feld **KundeID** mit dem Kunden des aktuellen Datensatzes aus **rstQuelle** übereinstimmt.

Für den Fall, dass für den Kunden noch gar keine Bestellung vorliegt, soll **datLetzteBestellung** den Wert **0** erhalten, was die **Nz**-Funktion mit entsprechendem zweiten Parameter bewerkstelligt.

Die folgende **If...Then**-Bedingung prüft, ob das in **datLetzteBestellung** gespeicherte Datum kleiner als **datStichtag** ist. Nur in diesem Fall geht es weiter, und zwar mit einer weiteren Prüfung: Wenn der Kunde bereits in der Archivtabelle gespeichert ist, soll dieser nicht erneut gespeichert werden.

Erst danach wird der neue Datensatz angelegt, und zwar mit der **AddNew**-Methode des **Recordset**-Objekts **rstZiel**. Die folgenden Anweisungen tragen jeweils den Wert eines Feldes der Quelltabelle in das entsprechende Feld der Zieltabelle ein.

Danach fügt die Funktion noch dem Feld **ArchiviertAm** das aktuelle Datum und die aktuelle Uhrzeit hinzu. Die **Update**-Methode sorgt für das Speichern des neu angelegten Datensatzes in der Tabelle **tblKunden_Archiv**.

Schließlich erhöht die Prozedur den Wert der Variablen **intAnzahl**, welche die Anzahl der übertragenen Datensätze zählen soll, um **1**. Die **MoveNext**-Methode bewegt den Datensatzzeiger zum nächsten Datensatz, die **Loop**-Anweisung lässt die Funktion mit einem weiteren Durchlauf der **Do While**-Schleife beginnen.

Schließlich liefert die Funktion die in **intAnzahl** gespeicherte Anzahl betroffener Datensätze zurück.

Wenn Sie in Betracht ziehen, dass eine solche Vorgehensweise auf große Datenmengen angewendet wird, sind eine Menge Codezeilen abzuarbeiten, bis das gewünschte Resultat erreicht wird.

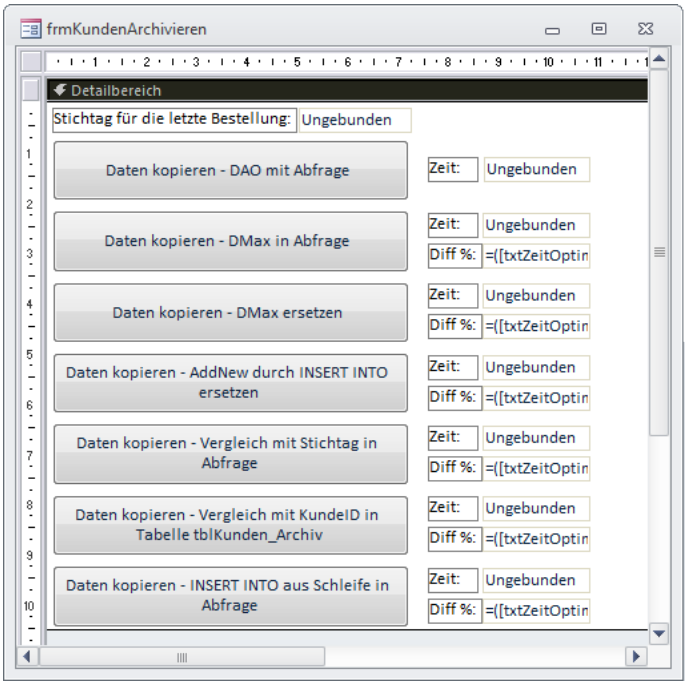

Bild 2: Formular mit Schaltflächen zum Starten der verschiedenen Archivierungen

```
Public Function Datenkopieren_DAOMitAbfrage(datStichtag As Date) As Integer
    Dim db As DAO.Database
    Dim rstQuelle As DAO.Recordset
    Dim rstZiel As DAO.Recordset
    Dim datLetzteBestellung As Date
    Dim intAnzahl As Integer
    Set db = CurrentDb
    Set rstQuelle = db.OpenRecordset("SELECT * FROM tblKunden", dbOpenDynaset)
    Set rstZiel = db.OpenRecordset("SELECT * FROM tblKunden_Archiv WHERE 1 = 2", dbOpenDynaset)
    Do While Not rstQuelle.EOF
        datLetzteBestellung = Nz(DMax("Bestelldatum", "tblBestellungen", "KundeID = " & rstQuelle!KundeID), 0)
        If datLetzteBestellung < datStichtag Then
            If IsNull(DLookup("KundeID", "tblKunden_Archiv", "KundeID = " & rstQuelle!KundeID)) Then
                With rstZiel
                    .AddNew
                    !KundeID = rstQuelle!KundeID
                    !KundenCode = rstQuelle!KundenCode
                    !Firma = rstQuelle!Firma
                    !Kontaktperson = rstQuelle!Kontaktperson
                    !Strasse = rstQuelle!Strasse
                    !Ort = rstQuelle!Ort
                    !Region = rstQuelle!Region
                    !PLZ = rstQuelle!PLZ
                    !Land = rstQuelle!Land
                    !Telefon = rstQuelle!Telefon
                    !Telefax = rstQuelle!Telefax
                    !ArchiviertAm = Now
                    .Update
                    intAnzahl = intAnzahl + 1
                End With
            End If
        End If
        rstQuelle.MoveNext
    Loop
    Datenkopieren_DAOMitAbfrage = intAnzahl
End Function
```

Listing 1: Die „sichere" Variante zum Kopieren von Daten

Also kümmern wir uns darum, dies zu optimieren. Gleichwohl ist zu erwähnen, dass die hier eingesetzte Vorgehensweise nicht völlig abwegig ist. Erstens kommt sie fast komplett ohne den Einsatz von SQL-Anweisungen aus, was für viele Einsteiger erst in späteren Schritten folgt. Somit kommt man mit den grundlegenden DAO-Anweisungen sowie Domänenfunktionen über die Runden.

Das Entscheidende für den Entwickler dieser Zeilen mag aber sein, dass sich zu jeder Zeit im Debug-Modus beobachten lässt, was dort geschieht. Dies ist beim Einsatz etwa von SQL-Aktionsabfragen nur eingeschränkt möglich.

Optimierung 1: DMax auflösen

Die obige Funktion muss wirklich viel Arbeit leisten, weil sie jeden einzelnen Datensatz der Herkunftstabelle **tblKunden** durchläuft und diese dahingehend prüft, wie lange die letzte Bestellung des Kunden zurückliegt und ob dieser Datensatz bereits in der Tabelle **tblKunden_Archiv** gespeichert ist. Dabei ist für

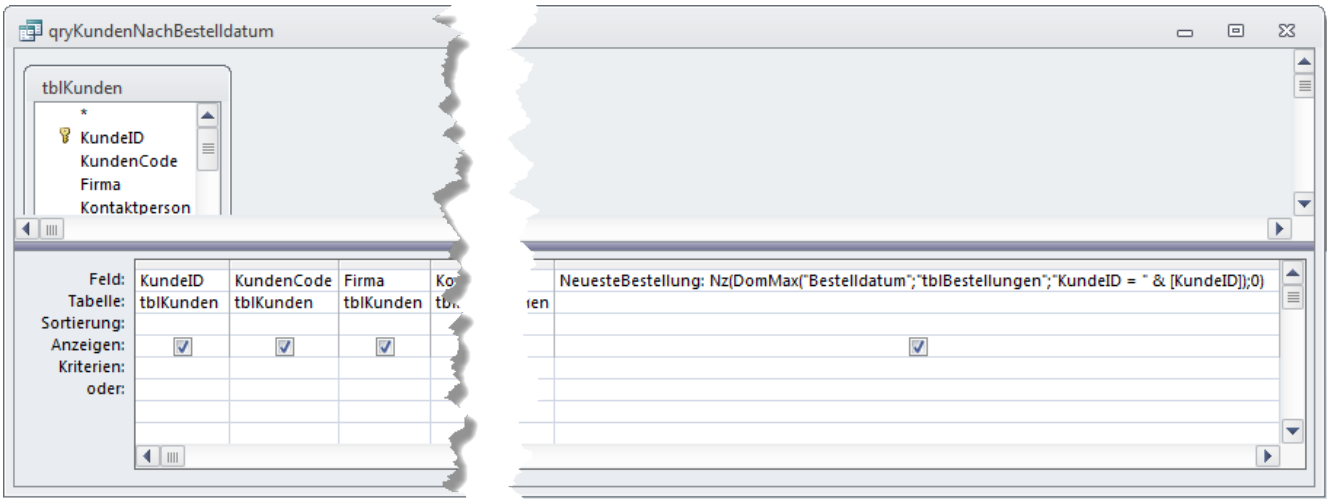

Bild 3: Abfrage basierend auf der Tabelle **tblKunden** und der neuesten Bestellung für diesen Kunden

jeden Datensatz der Aufruf der **DMax**-Funktion fällig.

Wir wollen in kleinen Schritten optimieren und versuchen zunächst, die **DMax**-Funktion aus der Schleife herauszunehmen. Die Datenherkunft besteht bisher aus der Tabelle **tblKunden**. Deren Feld **KundeID** verwenden wir, um per **DMax** das Datum der letzten Bestellung zu ermitteln.

Wie bekommen wir nun die **DMax**-Anweisung aus der **Do While**-Schleife heraus? Der erste Schritt dazu ist einfach: Wir erstellen einfach eine Abfrage auf Basis der Tabelle **tblKunden** und fügen ein weiteres Feld hinzu, welches das Datum der neuesten Bestellung ermittelt – dies wiederum durch den Einsatz der **DMax**-Funktion. Der Ausdruck für dieses Feld sieht so aus, die komplette Abfrage **qryKundenNachBestelldatum** finden Sie in Bild 3:

```
NeuesteBestellung: Nz(DomMax("Bestelldatum";"tblBestellungen";"KundeID = " & [KundeID]);0)
```

Der Clou hierbei ist, dass die **DMax**-Funktion die ID des Kunden, für den die neueste Bestellung ermittelt werden soll, aus dem Feld **KundeID** der Datenherkunft bezieht.

Die Abfrage **qryKundenNachBestelldatum** nutzen wir nun als Datenherkunft für die zu übertragenen Daten. Das Feld **NeuesteBestellung** nutzen wir nun direkt als Kriterium der **If...Then**-Bedingung, anstatt diesen Wert erst noch in der Prozedur per **DMax** ermitteln zu müssen (s. Listing 2).

Zeit messen

Nachdem wir nun eine erste vermeintliche Verbesserung angewendet haben, wollen wir natürlich auch wissen, ob sich dies auf die Performance auswirkt.

Die Zeit zwischen zwei Ereignissen messen Sie unter Windows am genauesten mit den beiden API-Funktionen **QueryPerformanceFrequency** und **QueryPerformanceCounter**. Die Erste liefert die Frequenz, die Zweite den Zählerstand des Prozessors. Wenn Sie diesen einmal

```
Public Function Datenkopieren_II(datStichtag As Date) As Integer
    ...
    Set rstQuelle = db.OpenRecordset("SELECT * FROM qryKundenNachBestelldatum", dbOpenDynaset)
    Set rstZiel = db.OpenRecordset("SELECT * FROM tblKunden_Archiv WHERE 1 = 2", dbOpenDynaset)
    Do While Not rstQuelle.EOF
        If rstQuelle!NeuesteBestellung < datStichtag Then
            ...
    Datenkopieren_II = intAnzahl
End Function
```

Listing 2: Optimierung der Schleife durch Übertragen von **DMax** in die Datenherkunft

```
Private Declare Function QueryPerformanceCounter Lib "Kernel32" (X As Currency) As Long
Private Declare Function QueryPerformanceFrequency Lib "Kernel32" (Y As Currency) As Long
Dim curFreq As Currency
Dim curStart As Currency
Dim curEnde As Currency

Private Sub cmd1_Click()
    Dim intAnzahl As Integer
    Dim db As DAO.Database
    Set db = CurrentDb
    db.Execute "DELETE FROM tblKunden_Archiv", dbFailOnError
    QueryPerformanceFrequency curFreq
    QueryPerformanceCounter curStart
    intAnzahl = Datenkopieren_I(Me!txtStichtag)
    QueryPerformanceCounter curEnde
    Me!txtZeitI = (curEnde - curStart) / curFreq
    MsgBox intAnzahl & " Datensätze archiviert"
End Sub
```

Listing 3: Einrichtung zum Messen der Zeit

beim Start der Messung einlesen und einmal zum Ende, brauchen Sie nur noch die Differenz durch die Frequenz zu teilen und erhalten die Zeit in Sekunden.

In Listing 3 werden zunächst die beiden API-Funktionen deklariert, außerdem drei **Currency**-Variablen zum Speichern der Frequenz, des Zählerstands beim Start und des Zählerstands nach Beenden der zu messenden Vorgänge. Die Prozedur, die beim Anklicken der Schaltflächen zum Starten der verschiedenen Varianten unseres Vorgangs ausgelöst wird, erledigt nun folgende Aufgaben:

- Leeren der Archivtabelle **tblKunden_Archiv**, damit immer gleich viele Datensätze übertragen werden
- Einlesen der Taktfrequenz des Prozessors
- Speichern des Zählers vor Beginn des zu messenden Vorgangs
- Start des Vorgangs
- Speichern des Zählers nach dem Vorgang
- Ausgabe der verstrichenen Zeit in Sekunden im Textfeld neben der Schaltfläche

Bild 4: Ergebnis der ersten beiden Varianten

Diese Zeilen fügen wir jeder Schaltfläche hinzu, nur der Aufruf der Funktion sowie das Textfeld für die Ausgabe der Zeit ändert sich. Wie Bild 4 zeigt, hat sich die erste Maßnahme durchaus gelohnt: Wir sparen rund 10% der Zeit gegenüber der ersten Variante ein.

Optimierung II:

Nun führen wir allerdings immer noch für jeden Datensatz einen Aufruf der **DMax**-Funktion aus. Wie können wir dies verhindern? Eine Möglichkeit wäre, statt der **DMax**-Funktion mit einer Unterabfrage zu arbeiten. Diese formulieren wir in einer neuen Abfrage namens

Undo in Haupt- und Unterformular mit Klasse

Das Problem beim Einsatz von Haupt- und Unterformularen mit Daten aus verknüpften Tabellen ist, dass der Benutzer diese als Einheit ansieht. Enthält das Hauptformular eine Abbrechen-Schaltfläche, geht er davon aus, dass er die Änderungen an Daten im Haupt- oder Unterformular damit komplett rückgängig machen kann. Leider ist das nicht so – die Änderungen im Unterformular bleiben gespeichert, und auch die Werte im Hauptformular lassen sich nach dem Speichern etwa durch einen Mausklick auf den Datensatzmarkierer nicht mehr rückgängig machen. Grund genug, unsere bereits einmal beschriebene Technik nochmal unter die Lupe zu nehmen und in eine flexibel einsetzbare Klasse zu exportieren.

Beispieltabellen und -formulare

Im Rahmen dieses Beitrag verwenden wir die Tabellen der Südsturm-Datenbank, genau genommen die Tabellen **tblBestellungen**, **tblKunden**, **tblBestelldetails** und **tblArtikel**. Außerdem erstellen wir zunächst ein Hauptformular namens **frmBestellungen** und ein Unterformular namens **sfmBestellungen**, um die herkömmliche Situation zu betrachten.

Später statten wir ähnliche Formulare mit einem Klassenmodul und dessen Methoden und Eigenschaften aus, um das Undo sowohl im Haupt- als auch im Unterformular zu ermöglichen.

Die an der Beispiellösung beteiligten Tabellen finden Sie in der Übersicht aus Bild 1. Direkt in die Formulare eingebunden werden dabei nur die beiden Tabellen **tblBestellungen** und **tblBestelldetails**, die Daten der Tabelle **tblKunden** stehen im Hauptformular als Daten eines Kombinationsfeldes zur Verfügung, die Daten der Tabelle **tblArtikel** in einem Kombinationsfeld im Unterformular.

Ausgangssituation

Wir wollen ein Element der Benutzeroberfläche einer Anwendung optimieren, das im Hauptformular jeweils einen Datensatz der Tabelle **tblBestellungen** anzeigt und im Unterformular die damit verknüpften Datensätze der Tabelle **tblBestelldetails**. Dazu weisen Sie einfach der Eigenschaft Datenherkunft von Haupt- und Unterformular die beiden Tabellen **tblBestellungen** und **tblBestelldetails** zu.

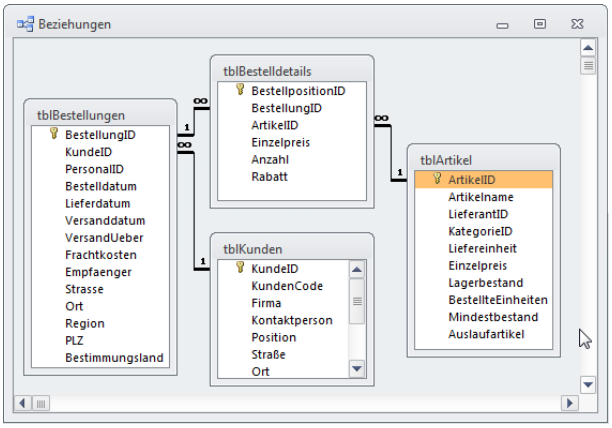

Bild 1: Tabellen der Beispieldatenbank

Das Hauptformular soll in diesem Fall die Felder **BestellungID**, **KundeID**, **Bestelldatum**, **Lieferdatum** und **Versanddatum** der Tabelle **tblBestellungen** enthalten, das Unterformular die Felder **ArtikelID**, **Einzelpreis**, **Anzahl**, **Rabatt** und **BestellungID** der Tabelle **tblBestelldetails**.

In der Entwurfsansicht sieht der Aufbau der beiden Formulare nun wie in Bild 2 aus. Das Unterformularsteuerelement haben wir **sfm** genannt, damit wir es später im Code vom eingebetteten Unterformular **sfmBestellungen** unterscheiden können.

Die beiden Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformularsteuerelements **sfm** enthalten beide den Namen des Feldes **BestellungID**. Auf diese Weise zeigt das Unterformular jeweils die zum Datensatz des Hauptformulars passenden Daten an. Das Hauptformular enthält noch zwei Schaltflächen. **cmdOK** löst diesen Code aus:

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

Damit schließt sie das Formular und speichert den aktuellen Zustand der Daten. Die zweite Schaltfläche enthält die Beschriftung **Abbrechen** oder, in diesem Fall, **Verwerfen**. Sie führt lediglich die **Undo**-Methode des Hauptformular aus und verwirft somit die Änderungen seit dem letzten Speichern im Hauptformular:

```
Private Sub cmdVerwerfen_Click()
    Me.Undo
End Sub
```

Dies ist für den Benutzer mitunter irreführend, da er den Eindruck erhalten kann, dass sich mit einer solchen Schaltfläche alle seit dem letzten Öffnen getätigten Änderungen verwerfen lassen. Dies ist mitnichten so: Änderungen im Unterformular werden auf keinen Fall verworfen, denn diese werden spätestens dann in der zugrunde liegenden Tabelle gespeichert, wenn das Unterformular den Fokus verliert. Und das ist zwangsläufig der Fall, wenn der Benutzer ein Steuerelement des Hauptformulars betätigt, in diesem Fall die **Verwerfen**-Schaltfläche.

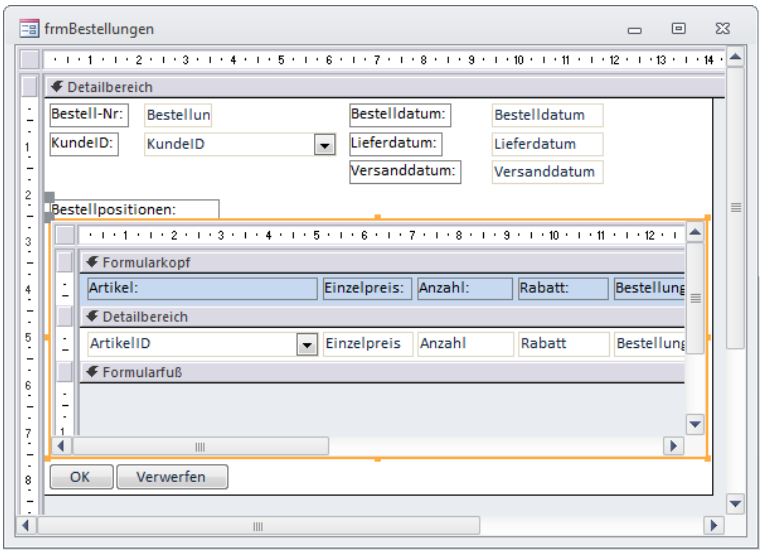

Bild 2: Haupt- und Unterformular in der Entwurfsansicht

Andersherum wird der Datensatz im Hauptformular sofort gespeichert, wenn der Benutzer den Fokus in das Unterformular verschiebt. Beim klassischen Fall, also dem Anlegen der grundlegenden Bestelldaten wie Kunde, Bestelldatum et cetera und dem anschließenden Hinzufügen von Bestellpositionen bewirkt das Betätigen der **Verwerfen**-Schaltfläche schlicht und einfach nichts. Zu diesem Zeitpunkt sind alle Daten bereits gespeichert. Da die **Undo**-Methode genau die gleiche Wirkung hat wie das Betätigen der Escape-Taste, macht auch diese keine der getätigten und gespeicherten Eingaben mehr rückgängig.

Dies wollen wir nun ändern – und zwar mit einer Klasse, die alle dazu nötigen Funktionen enthält. Das Ganze ist nicht gerade trivial, denn wir müssen einige Fälle beachten. Bei der Programmierung einer solchen Klasse wird man kaum auf einen Schlag alle denkbaren Konstellationen erschlagen. Man beginnt also damit, einfache Vorfälle abzudecken, und programmiert deren Behandlung. Wenn dies funktioniert, schaut man sich den nächsten Vorfall an und programmiert weiter.

Beim Testen des Programmierfortschritts sollte man jedoch auch immer wieder die vorher behandelten Vorfälle testen. Damit stellen Sie sicher, dass Sie durch das Hinzufügen oder Ändern des bestehenden Codes die bereits vorhandene Funktionalität beeinflussen.

Grundsätzlich soll die Klasse dafür sorgen, dass alle seit dem Öffnen eines Datensatzes getätigten Änderungen mit einem Klick auf die **Verwerfen**-Schaltfläche wieder rückgängig gemacht werden können.

Es gibt eine Ausnahme: Das Löschen einer kompletten Bestellung, also eines Datensatzes der Tabelle **tblBestellungen** und damit auch aller damit verknüpften Datensätze der Tabelle **tblBestellpositionen**. In dem Moment, in dem der Benutzer den Datensatz im Hauptformular löscht, spielt es keine Rolle mehr, ob die bisherigen Änderungen der Transaktion durchgeführt werden oder nicht.

Tests festlegen

Um zu prüfen, ob alles jederzeit wie gewünscht funktioniert, legen wir einige Tests zurecht – diese sollten alle paar neuen Codezeilen geprüft werden, um vorzeitig ein Programmieren in die falsche Richtung zu verhindern:

- Eine Bestellung (Bestelldatum **1.1.2013**) ohne Bestellpositionen wird erstellt und gespeichert. Ist die Bestellung noch vorhanden?
- Eine Bestellung (Bestelldatum **2.1.2013**) mit einer Bestellposition wird erstellt und gespeichert. Sind die Bestellung und die Bestellpositionen noch vorhanden?
- Eine Bestellung (Bestelldatum **3.1.2013**) wird erstellt und gespeichert (zum Beispiel durch einen Klick auf den Datensatzmarkierer) und mit der Schaltfläche **Verwerfen** verworfen. Ist die Bestellung verschwunden?
- Eine Bestellung (Bestelldatum **4.1.2013**) mit einer Bestellposition wird erstellt und endgültig gespeichert (durch Wechseln zu einem anderen Datensatz oder Schließen

und erneutes Öffnen des Formulars). Dann wird zu dieser Bestellung eine Bestellposition hinzugefügt und gespeichert. Entfernt **Verwerfen** diese Position wieder, während die Bestellung selbst erhalten bleibt?

- Eine Bestellung (**5.1.2013**) erstellen, eine Bestellposition hinzufügen und Formular schließen. Bestellung wieder anzeigen, Bestellposition löschen und mit **Verwerfen** wiederherstellen.

- Eine Bestellung (**6.1.2013**) erstellen, eine Bestellposition hinzufügen und Formular schließen. Bestellung wieder anzeigen, Bestellposition löschen, mit **Verwerfen** wiederherstellen, wieder löschen und wieder herstellen.

Am schönsten wäre es natürlich, wenn man solche und ähnliche Tests der Benutzeroberfläche automatisch ablaufen lassen könnte. Dies ist jedoch sehr aufwendig zu realisieren. Gegebenenfalls kümmern wir uns zu einem späteren Zeitpunkt um ein entsprechendes Tool.

Transaktionen

Der Wechsel zum Unterformular speichert bereits die Daten im Hauptformular und der Wechsel von einem Datensatz zum nächsten im Unterformular das Gleiche mit den Datensätzen im Unterformular. Wie wollen wir dann dafür sorgen, dass die vollständigen Änderungen am aktuellen Datensatz im Hauptformular samt den verknüpften Daten im Unterformular rückgängig gemacht werden können?

Dafür gibt es verschiedene Ansätze. Der erste wäre, den aktuellen Datensatz im Hauptformular vor der Bearbeitung im Hauptformular in einer temporären Tabelle zu speichern – ebenso wie die Daten des Unterformulars. Beim Mausklick auf die Schaltfläche **OK** werden dann alle Daten in die Originaltabellen übertragen, beim Anklicken von **Verwerfen** löschen wir einfach die Daten der temporären Tabellen.

Der zweite Ansatz verwendet Transaktionen. Das bedeutet, dass wir bei der ersten Änderung an dem im Hauptformular angezeigten Datensatz oder an einem der Datensätze im Unterformular eine Transaktion starten müssen. Das gilt natürlich auch dafür, wenn wir im Hauptformular einen neuen Datensatz anlegen oder Datensätze zum Unterformular hinzufügen.

Der Einsatz von Transaktionen ist recht einfach: Sie definieren eine Workspace-Variable, die den Workspace der aktuellen Datenbank referenziert (mit **DBEngine.Workspaces(0)**. Dieses Workspace-Objekt stellt dann die folgenden drei Methoden zur Verfügung:

- **BeginTrans**: Startet eine Transaktion.

- **CommitTrans**: Speichert die seit dem Start der Transaktion durchgeführten Änderungen.

- **Rollback**: Verwirft alle Änderungen seit Beginn der Transaktion.

Nun muss man allerdings wissen, welche Änderungen vom Start der Transaktion an protokolliert werden und entsprechend rückgängig gemacht werden können. Dabei handelt es lediglich um solche Transaktionen, die über das **Database**-Objekt der aktuellen Datenbank durchgeführt wurden. Wenn Sie also etwa nach dem Aufruf von **BeginTrans** mit der **db.Execute**-Methode eine Aktionsabfrage durchführen oder die Daten eines DAO-Recordsets mit **Edit**/**AddNew** und **Update** ändern, können Sie diese Änderungen durch **CommitTrans** speichern oder durch **Rollback** verwerfen.

Dummerweise laufen die Änderungen, die Sie an den Daten eines über die Eigenschaft **Datenherkunft** an eine Datenquelle gebundenen Formulars durchführen, nicht im Kontext einer Transaktion. Und hier wird es interessant: Sie können ein Formular nämlich auch über die **Recordset**-Eigenschaft mit den Daten aus einer Tabelle oder Abfrage füllen. Und wenn Sie dieses Recordset im Kontext einer Transaktion mit **OpenRecordset** erstellen und der Datenherkunft des Formulars beziehungsweise des Unterformulars zuweisen, können Sie auch die Änderungen am Formular durch ein **Rollback** verwerfen oder durch ein **CommitTrans** speichern.

Das hat allerdings auch kleinere Nachteile: Normalerweise weisen Sie dem Hauptformular die eine Tabelle der 1:n-Beziehung als Datenherkunft zu und dem Unterformular die andere Tabelle. Dabei stellen Sie die beiden Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformular-Steuerelements auf das Fremdschlüsselfeld im Unterformular und das Primärschlüsselfeld im Hauptformular ein, damit das Unterformular jeweils die passenden Daten zum Hauptformular anzeigt.

Wenn Sie Haupt- und Unterformular jedoch etwa im Ereignis **Beim Laden** des Hauptformulars mit den zuvor erstellten **Recordset**-Objekten versehen, sind die beiden Eigenschaften **Verknüpfen von**

und **Verknüpfen nach** wirkungslos. Deshalb müssen Sie dem Unterformular beim Wechsel des Datensatzes im Hauptformular jeweils ein neues Recordset zuweisen, dass die zum Datensatz im Hauptformular passenden Datensätze enthält.

So schlimm ist das aber auch nicht – wir müssen diesen Vorgang ja auch nur einmal programmieren.

Formular anpassen

Nachdem wir wissen, dass wir das Formular und das Unterformular mit einer Klasse ausstatten wollen, die dafür sorgt, dass Änderungen an den kompletten angezeigten Daten des aktuellen Datensatzes des Hauptformulars durch einen Mausklick auf eine Schaltfläche rückgängig gemacht werden, bereiten wir zunächst das Formular vor.

Formular und Unterformular erstellen

Grundsätzlich sollten Haupt- und Unterformular zunächst mit den Tabellen oder Abfragen als Datenherkunft ausgestattet werden, die sie im normalen Betrieb verwenden würden.

Dies geschieht aus reiner Bequemlichkeit: Wir können so nämlich die Steuerelemente der jeweiligen Datenherkunft aus der Feldliste in die gewünschten Bereiche des Formulars ziehen. Danach leeren wir einfach die Eigenschaft **Datenherkunft** des Formulars.

Auf die gleiche Weise gehen wir beim Unterformular vor. Hier ist darauf zu achten, dass Sie auch die beiden Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformular-Steuerelements leeren (s. Bild 3).

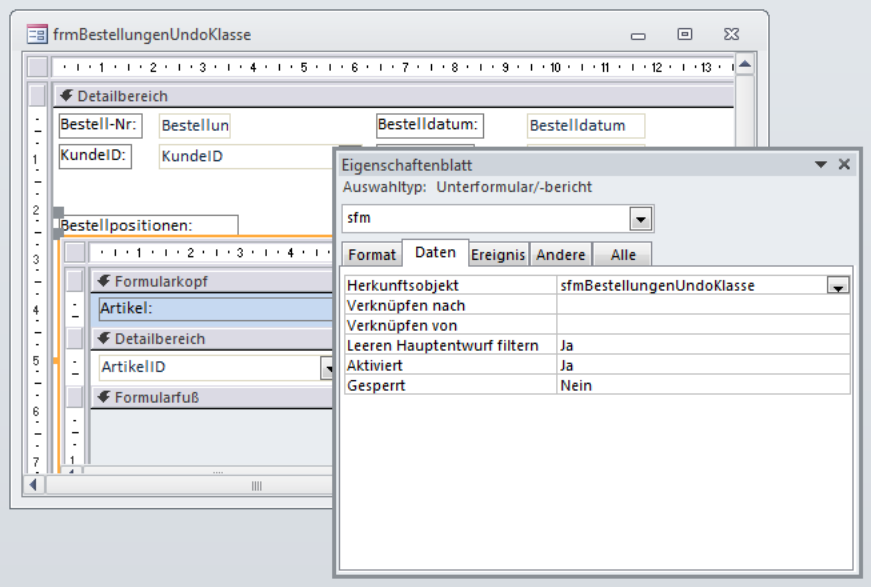

Bild 3: Die Verknüpfungseigenschaften zwischen Haupt- und Unterformular werden geleert.

Klasse erstellen

Anschließend erstellen wir schon das Klassenmodul (VBA-Editor, **EinfügenlKlassenmodul**). Dieses nennen wir schlicht und einfach **clsUndo**.

Normalerweise hätten wir die Ereignisse, die dazu führen, dass eine Transaktion gestartet, beendet oder verworfen wird, direkt in den Klassenmodulen des Haupt- und des Unterformulars untergebracht. Dann müssten Sie diesen Code jedoch, wenn Sie diesen in einem anderen Formular weiterverwenden wollten, jeweils in das entsprechende Klassenmodul des Formulars/Unterformulars übertragen.

Wenn Sie eine solche Funktionalität einmal programmiert haben und diese dann auf einen anderen Anwendungsfall übertragen möchten, sind meist individuelle Anpassungen nötig – was zu Fehlern führen kann, insbesondere dadurch bedingt, dass man nicht mehr genau weiß, an welchen Schrauben man wie drehen muss. Wenn Sie hingegen die komplette Funktion in einem eigenen Klassenmodul kapseln und dieses von dem damit auszustattenden Formular einfach nur instanzieren und mit einigen Eigenschaften ausstatten müssen, haben Sie leichtes Spiel.

Referenzen an das Klassenmodul übergeben

Die Ereignisse, Variablen und Prozeduren, die Sie normalerweise im Klassenmodul des betroffenen Formulars angelegt hätten, deklarieren Sie nun komplett im Klassenmodul **clsUndo**. Dabei ist allerdings zu beachten, dass dieses ja nicht weiß, auf welches Formular es sich beziehen soll. Genauso weiß es nicht, welches Unterformular betroffen ist (wenn mehrere vorhanden sind) und welche Schaltflächen für das Übernehmen oder Verwerfen der Änderungen verantwortlich sind.

Damit wir der Klasse beim Laden des Formulars die benötigten Informationen übergeben können, legen wir in der Klasse zunächst einige Variablen fest,

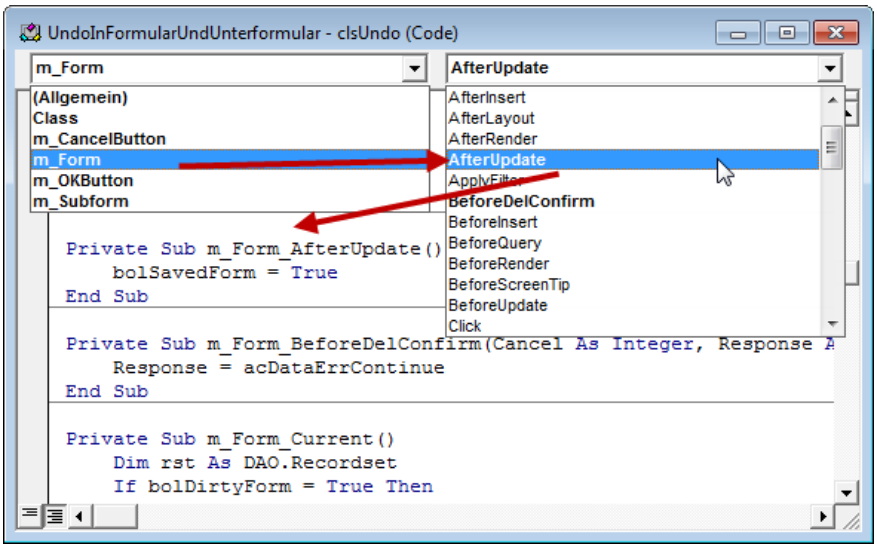

Bild 4: Hinzufügen einer Ereignisprozedur für eine mit **WithEvents** deklarierte Objektvariable

welche die Verweise auf diese Objekte speichern sollen. Dies sieht wie folgt aus:

Wir benötigen also zwei Variablen des Typs **Form** und zwei des Typs **CommandButton**. Für alle vier Objekte wollen wir innerhalb des Klassenmoduls **clsUndo** eigene Ereignisprozeduren implementieren. Das geschieht prinzipiell genauso wie im Klassenmodul eines Formulars, allerdings steht die einfache Möglichkeit zum Erstellen solcher Ereignisprozeduren durch Eintragen des Wertes **[Ereignisprozedur]** in die jeweilige Ereigniseigenschaft und anschließendes Klicken auf die Schaltfläche mit den drei Punkten (...) rechts neben der Eigenschaft nicht zur Verfügung.

Bliebe noch die Möglichkeit, die entsprechenden Objektvariablen im linken Kombinationsfeld des VBA-Editors auszuwählen und das passende Ereignis aus dem rechten Kombinationsfeld zu ergänzen. Das gelingt aber auch nur unter einer Bedingung: Wenn die passende Objektvariable mit dem **WithEvents**-Schlüsselwort deklariert wurde – und genau dies erledigen wir wie folgt:

```
Private WithEvents m_Form As Form
Private WithEvents m_Subform As Form
Private WithEvents m_OKButton As ⤶
                                CommandButton
Private WithEvents m_CancelButton As ⤶
                                CommandButton
```

Anschließend fügen Sie über die beiden Kombinationsfelder im Codefenster die benötigten Ereignisprozeduren hinzu (s. Bild 4).

Wie gelangen aber nun die Verweise auf die entsprechenden Elemente des Formulars in diese Variablen – und wie sorgen wir dafür, dass das Formular weiß, dass es im Klassenmodul **clsUndo** noch Prozeduren gibt, die beim Eintreten verschiedener Ereignisse des Formulars, des Unterformulars oder der beiden Schaltflächen ausgelöst werden sollen? Dies geschieht in drei Schritten:

- Wir legen in der Klasse **clsUndo** für jedes betroffene Objekt eine **Property Set**-Prozedur an, welche die übergebenen Objekte der Variablen **m_Form** et cetera zuweist.
- Diese **Property Set**-Prozeduren statten wir gleichzeitig mit Anweisungen aus, welche die Ereigniseigenschaften wie etwa **Beim Anzeigen** (**On-Current**) des Unterformulars mit dem Wert **[Event Procedure]** füllen. Dies entspricht dem Setzen des Wertes **[Ereignisprozedur]** für die Ereigniseigenschaften im Eigenschaftsfenster des Formulars.
- Der Ereignisprozedur **Beim Laden** des Formulars fügen wir dann Code hinzu, der die Klasse instanziert und dieser die Verweise auf das Formular, das Unterformular und die beiden Schaltflächen übergibt.

Prozeduren als Eigenschaften

Legen wir zunächst die vier **Property Set**-Prozeduren in der Klasse **clsUndo** an. Diese erscheinen dann später im Klassenmodul des Formulars, das die Klasse **clsUndo** instanziert, als Eigenschaften des Klassenobjekts.

Wir beginnen mit den beiden Schaltflächen zum Speichern und zum Verwerfen der Änderungen.

Die **OK**-Schaltfläche soll über die folgende **Property Set**-Prozedur an die Klasse **clsUndo** übergeben werden:

```
Public Property Set OKButton(cmd As ⤶
                                CommandButton)
    Set m_OKButton = cmd
    With m_OKButton
        .OnClick = "[Event Procedure]"
    End With
End Property
```

Bibliotheken und Verweise untersuchen

Wenn Sie Access-Anwendungen programmieren, nutzen Sie verschiedene Bibliotheken wie etwa die VBA-Bibliothek, die Access-Bibliothek, die DAO-Bibliothek et cetera. In diesem Beitrag schauen wir uns an, wie Sie an wichtige Informationen über diese Bibliotheken gelangen und was dies beispielsweise im Hinblick auf das Ermitteln der neuen Features einer neuen Access-Version bedeutet.

Wo stecken die Bibliotheken?

Als Access-Entwickler wissen Sie natürlich, wie Sie den Speicherort der im aktuellen VBA-Projekt eingebundenen Bibliotheken finden – nämlich im **Verweise**-Dialog, den Sie im VBA-Editor mit dem Menüpunkt **Extras|Verweise** öffnen.

Allerdings hilft dieser Dialog in vielen Fällen nicht wirklich weiter – er ist schlicht nicht breit genug, um den Pfad des Verweises anzuzeigen (s. Bild 1).

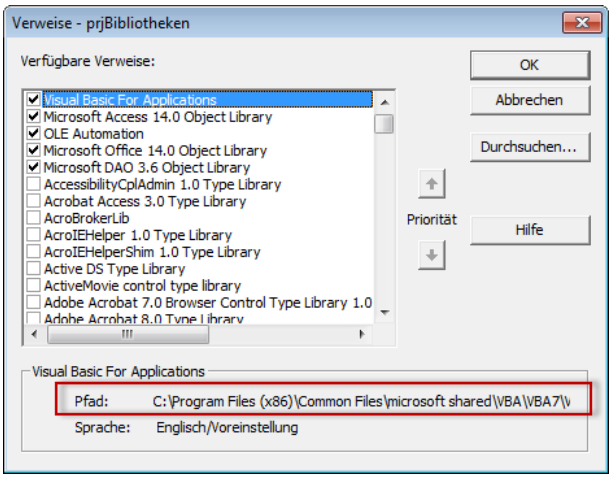

Bild 1: Verweise auf die aktuell eingebundenen Bibliotheken

Pfade und weitere Informationen einlesen

Wer des VBA einigermaßen mächtig ist, hat wohl schon davon gehört, dass man auch programmatisch auf die Verweise zugreifen kann – und zwar über die **References**-Auflistung des **Application**-Objekts. Um dieses zu nutzen, erstellen wir schnell eine Tabelle namens **tblBibliotheken**, welche die Informationen über die referenzierten Bibliotheken aufnehmen soll. Diese Tabelle sieht im Entwurf wie in Bild 2 aus.

Zum Anzeigen der eingelesenen Daten erstellen wir zwei Formulare – ein Hauptformular namens **frmBibliotheken** und ein Unterformular namens **sfmBibliotheken**. Beide sollen an die Tabelle **tblBibliotheken** gebunden sein, wobei das Hauptformular allerdings die Details und das Unterformular die vollständige Liste der Verweise liefern soll.

Deshalb fügen Sie dem Detailbereich des Unterformulars **sfmBibliotheken** zunächst nur das Feld **Bezeichnung** der Datenherkunft hinzu. Das Hauptformular nimmt zwei Schaltflächen auf, deren Funktion wir im Anschluss erläutern. Außerdem fügen Sie diesem das Unterformular hinzu und ziehen alle Felder der Datenherkunft mit Ausnahme des Primärschlüsselfeldes **BibliothekID** in einen eingerahmten Bereich. Das Ergebnis sieht schließlich wie in Bild 3 aus.

Bibliotheken-Tabelle füllen

Bevor das Formular Daten anzeigen kann, müssen Sie die Tabelle **tblBib-**

tblBibliotheken

Feldname	Felddatentyp
BibliothekID	AutoWert
Bezeichnung	Text
Eingebaut	Ja/Nein
Pfad	Text
BibliothekGUID	Text
Art	Zahl
Major	Text
Minor	Text
Beschreibung	Text

Bild 2: Die Tabelle **tblBibliotheken** nimmt Informationen zu den Verweisen der aktuellen Datenbank auf.

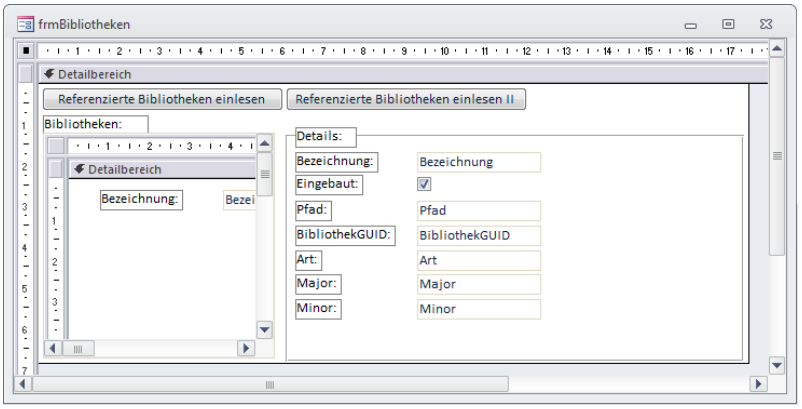

Bild 3: Formular und Unterformular zur Anzeige der Bibliotheken in der Entwurfsansicht

liotheken mit den Informationen über die referenzierten Bibliotheken füllen. Dies erledigt die Prozedur, die durch das Ereignis **Beim Klicken** der linken Schaltfläche namens **cmdEinlesen** ausgelöst wird. Diese Prozedur sieht wie in Listing 1 aus.

Die Prozedur deklariert einige Variablen, darunter die Variable **ref** des Typs **Reference**. Darüber greifen Sie auf die Verweise zu. Zu Beginn leert die Prozedur die Tabelle **tblBibliotheken** mit dem Aufruf einer entsprechenden **DELETE**-Anweisung.

In einer **For Each**-Schleife über die **References**-Auflistung des **Application**-Objekts durchläuft die Prozedur dann alle enthaltenen Elemente. Innerhalb der Schleife setzt die Prozedur in der Variablen **strSQL** eine SQL-Anweisung zusammen, welche die folgenden Eigenschaften des **Reference**-Objekts in die Tabelle **tblBibliotheken** schreibt:

- **Name**: Name der Bibliothek

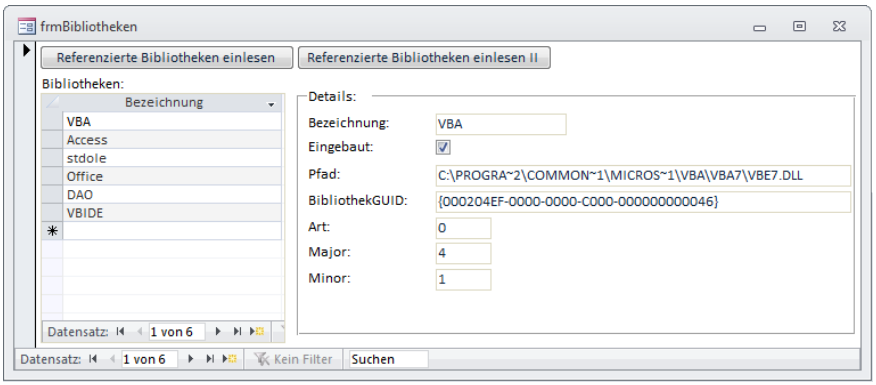

Bild 4: Liste der Verweise und Details des aktuell angezeigten Verweises nach dem Einlesen

- **BuiltIn**: Boolean-Wert, der angibt, ob der Verweis auf die Bibliothek eingebaut ist – und dementsprechend nicht entfernt werden kann
- **FullPath**: Pfad zur Bibliotheksdatei
- **Guid**: Eindeutiger Kennzeichner der Bibliothek
- **Kind**: Art der Bibliothek. Es gibt zwei mögliche Werte: **vbext_rk_TypeLib** (**0**) entspricht einer Typbibliothek, **vbext_rk_Project** (**1**) einem eingebundenen VBA-Projekt.
- **Major**: Hauptversion der Bibliothek
- **Minor**: Nebenversion der Bibliothek

Nach dem Durchlaufen aller Verweise aktualisiert die Prozedur noch das Unterformular. Das Ergebnis sieht beispielsweise wie in Bild 4 aus.

Haupt- und Unterformular synchronisieren

Fehlt noch eine Kleinigkeit: Wenn Sie auf einen der Einträge des Unterformulars klicken, soll das Hauptformular die Details zu diesem Datensatz anzeigen.

```
Private Sub cmdEinlesen_Click()
    Dim db As DAO.Database
    Dim ref As Reference
    Dim strSQL As String
    Set db = CurrentDb
    db.Execute "DELETE FROM tblBibliotheken", dbFailOnError
    For Each ref In Application.References
        With ref
            strSQL = "INSERT INTO tblBibliotheken(Bezeichnung, Eingebaut, Pfad, BibliothekGUID, Art, Major, Minor) VALUES('" _
                & .Name & "', " & CLng(.BuiltIn) & ", '" & .FullPath & "', '" & .Guid & "', " & .Kind & ", '" & .Major _
                & "', '" & .Minor & "')"
            db.Execute strSQL, dbFailOnError
        End With
    Next ref
    Me!sfmBibliotheken.Form.Requery
End Sub
```

Listing 1: Einlesen der referenzierten Bibliotheken

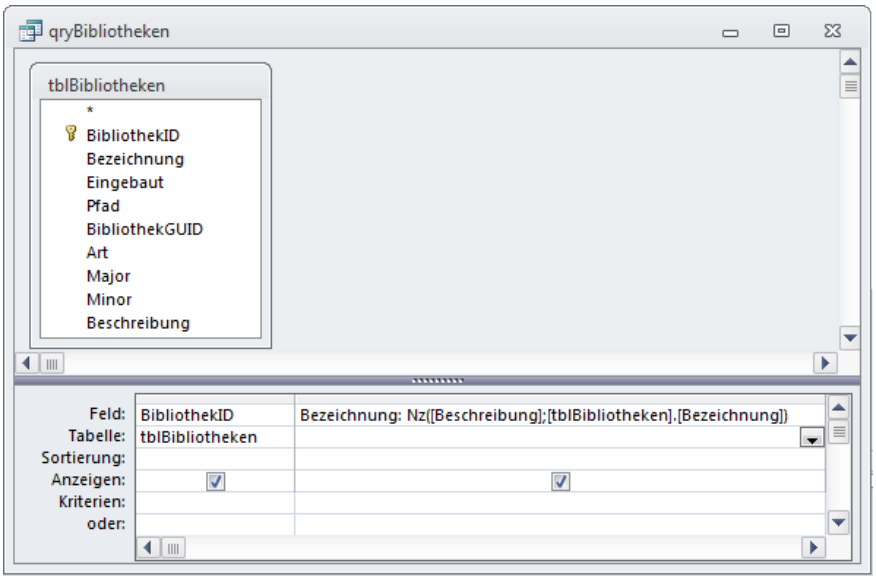

Bild 5: Abfrage, die nach Bedarf die Bezeichnung oder die Beschreibung liefert

Dazu fügen Sie zur Prozedur **Form_Load**, die beim Laden des Formulars ausgelöst wird, die folgende Zeile hinzu:

```
Private Sub Form_Load()
    Set Me.Recordset = _
        Me!sfmBibliotheken.Form.Recordset
End Sub
```

Dies synchronisiert den jeweils aktuellen Datensatz in beide Richtungen. Wenn Sie einen Datensatz im Unterformular auswählen, wird dieser im Hauptformular angezeigt und umgekehrt.

Wo ist die korrekte Bezeichnung?

Wenn Sie sich den **Verweise**-Dialog ins Gedächtnis rufen, stellen Sie fest, dass dieser wesentlich aussagekräftigere Bezeichnungen liefert als die Eigenschaft Name – mit **VBA** und **Access** lässt sich ja noch etwas anfangen, aber **stdole** hört sich kryptisch an.

Irgendwo müssen aber doch auch die entsprechenden Bezeichnungen aus dem **Verweise**-Dialog wie **Visual Basic for Applications**, **Microsoft Access 14.0 Object Library** oder **OLE Automatisch** abrufbar sein?

Und das ist auch der Fall: Es gibt nämlich noch eine weitere **References**-Auflistung mit entsprechenden **Reference**-Objekten. Diese stellt die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3** bereit, auf die wir nun einen Verweis hinzufügen.

Wir wollen die lange Version der **Bezeichnung** unter einem neuen Feld namens Beschreibung in der Tabelle **tblBibliotheken** speichern. Damit das Formular entweder die Beschreibung oder, falls diese nicht verfügbar ist, die Bezeichnung anzeigt, erstellen wir eine neue Abfrage als Datenherkunft. Diese sieht wie in Bild 5 aus und enthält ein Feld mit folgendem Ausdruck:

```
Bezeichnung: Nz([Beschreibung];[tblBibliotheken].[Bezeichnung])
```

Dieses neue Feld **Bezeichnung** prüft den Inhalt des Feldes **Beschreibung**. Ist dieses nicht leer, wird der Inhalt dieses Feldes zurückgegeben, sonst der Inhalt des Feldes **Bezeichnung**. Damit kein Zirkelbezug ausgelöst wird, verweist die Funktion auf **[tblBibliotheken].[Bezeichnung]** statt nur auf **[Bezeichnung]**.

Ersetzen Sie nun im Haupt- und im Unterformular die Datenherkunft **tblBibliotheken** durch **qryBibliotheken**, damit jederzeit die korrekten Daten angezeigt werden.

Lange Bezeichnung einlesen

Nun fehlt allerdings noch eine Prozedur, mit der Sie die lange Bezeichnung der Bibliothek einlesen können. Dazu rollen wir die zuvor erstellte Prozedur neu auf und legen diese für das Ereignis **Beim Klicken** der zweiten Schaltfläche **cmdEinlesenII** an (s. Listing 2).

Die Prozedur deklariert eine zusätzliche Variable namens **objProject** mit dem Typ **VBIDE.VBProject**. Dieses soll zunächst mit einem Verweis auf das VBA-Projekt der aktuellen Datenbank gefüllt werden. Normalerweise sollte man meinen, dass es ausreicht, das Projekt etwa mit folgender Anweisung zu referenzieren:

```
Set objProject = VBE.ActiveVBProject
```

Wenn Sie allerdings zufällig zuvor ein Access-Add-In genutzt haben, zeigt der VBA-Editor auch dessen VBA-Projekt an – und dann liefert **ActiveVBProject** nicht mehr zuverlässig das Projekt der aktuell geöffneten Access-Datenbank.

Diesen ermittelt die Prozedur in einer **For Each**-Schleife über alle aktuell geladenen VBA-Projekte. Wenn die Eigenschaft **FileName** des VBA-Projekts mit der Eigenschaft **Name** der mit **Current-**

```
Private Sub cmdEinlesenII_Click()
    Dim db As DAO.Database
    Dim ref As VBIDE.Reference
    Dim strSQL As String
    Dim objProject As VBIDE.VBProject
    Set db = CurrentDb
    db.Execute "DELETE FROM tblBibliotheken", dbFailOnError
    For Each objProject In VBE.VBProjects
        If objProject.FileName = CurrentDb.Name Then
            Exit For
        End If
    Next objProject
    For Each ref In objProject.References
        With ref
            strSQL = "INSERT INTO tblBibliotheken(Bezeichnung, Eingebaut, Pfad, BibliothekGUID, Art, Major, " _
                & "Minor, Beschreibung) VALUES('" & .Name & "', " & CLng(.BuiltIn) & ", '" & .FullPath & "', '" & .Guid _
                & "', " & .Type & ", '" & .Major & "', '" & .Minor & "', '" & .Description & "')"
            db.Execute strSQL, dbFailOnError
        End With
    Next ref
    Me!sfmBibliotheken.Form.Requery
End Sub
```

Listing 2: Einlesen der referenzierten Bibliotheken über die Elemente der VBE-Bibliothek

Db referenzierten (und gerade geöffneten) Datenbank übereinstimmt, haben Sie das richtige VBA-Projekt erwischt und die Schleife kann mit der Referenz auf dieses Projekt verlassen werden.

Die Variable **ref** ist diesmal als **VBIDE.Reference** deklariert und wir durchlaufen damit die **References**-Auflistung des soeben ermittelten und mit **objProject** referenzierten VBA-Projekts.

In den Eigenschaften des **Reference**-Objekts der **VBIDE**-Klasse findet sich dann auch noch eine weitere Eigenschaft namens **Description**, welche die volle Bezeichnung enthält, wie sie auch im **Verweise**-Dialog erscheint.

Außerdem gibt es einen weiteren kleinen Unterschied: Die Art des Verweises liefert hier die Eigenschaft **Type**, nicht **Kind**. All dies berücksichtigen wir in der Schleife über alle Verweise, in der diese Informationen wie bereits zuvor beschrieben in der Tabelle **tblBibliotheken** landen.

Anschließend zeigt das Formular auch die vollen Bezeichnungen an (s. Bild 6).

Verweise-Dialog anpassen

Vermutlich gab es bereits tausende Anfragen bei Microsoft mit der Bitte, den **Verweise**-Dialog so anzupassen, dass dieser die Pfade zu den Verweisen vollständig anzeigt. Leider hat sich in dieser Beziehung nichts getan.

Dabei wäre es wirklich sehr einfach, das Steuerelement, das derzeit den Pfad im **Verweise**-Dialog anzeigt, zumindest in ein Textfeld umzuwandeln, damit der Benutzer mit der Maus nach rechts scrollen und den vollständigen Pfad anzeigen und gegebenenfalls kopieren kann.

Sie können eine solche Änderung selbstständig durchführen – wir weisen jedoch ausdrücklich darauf hin, dass Microsoft dies in den Lizenzbedingungen untersagt. Die experimentelle Anwendung der nachfolgenden Schritte erfolgt außerdem auf eigene Gefahr.

Um die notwendige Änderung vorzunehmen, benötigen Sie zunächst ein Tool, mit dem Sie auf die Innereien des VBA-Editors zugreifen können. Dabei handelt es sich um sogenannte Resource Hacker. Der nachfolgende verwendete ist unter dem Link **http://www.angusj.com/resourcehacker/** zu finden. Laden Sie das Tool herunter und installieren Sie es. Wenn Sie es starten, verwenden Sie die Kontextmenü-Option **Als Admi-**

VBA-Funktionen von A bis Z

VBA-Funktionen kann doch jeder programmieren. Funktionsname, Parameter, Rückgabewert – fertig! Es gibt jedoch eine ganze Menge Feinheiten, die man als Programmierer kennen sollte. Dazu gehört beispielsweise das Zurückgeben mehr als eines Ergebniswertes, das Übergeben beliebig vieler Parameter oder auch optionale Parameter. Dieser Beitrag liefert alles, was Sie für den Umgang mit Funktionen wissen müssen.

Zur Programmierung von VBA-Funktionen muss man keine Raketentechnik studiert haben. Aber wie bereits in der Einleitung erwähnt, gibt es eine Menge verschiedener Techniken, die alle ihre Berechtigung haben. In diesem Beitrag wollen wir einmal einen Überblick über die Möglichkeiten von VBA-Funktionen liefern.

Funktion oder Prozedur?

Zunächst müssen wir klären, was genau der Unterschied zwischen einer **Sub**- und einer **Function**-Prozedur ist. Dieser liegt genau darin, dass eine **Function** ein Ergebnis zurückliefert, das über die Zuweisung des Funktionsaufrufs etwa an eine Variable, ein Steuerelement et cetera direkt weiterverarbeitet werden kann. Eine Besonderheit ergibt sich bei der Zuweisung dieses Wertes innerhalb der Funktion: Dieser wird nämlich einer Variablen zugewiesen, die den Namen der Funktion hat. Ein Beispiel sieht so aus:

```
Public Function NameDesAutors() As String
    NameDesAutors = "André Minhorst"
End Function
```

Rückgabewert verarbeiten

Diese Funktion weist ihrem Rückgabewert ein **String**-Literal zu, das beim Aufruf gleich verarbeitet werden kann – zum Beispiel durch die Ausgabe im Direktfenster:

```
Debug.Print NameDesAutors
André Minhorst
```

Auf die gleiche Weise können Sie den Rückgabewert der Funktion etwa in eine Variable schreiben und dann im Direktbereich ausgeben:

```
Public Sub Test_NameDesAutors()
    Dim strNameDesAutors As String
    strNameDesAutors = NameDesAutors
    Debug.Print strNameDesAutors
End Sub
```

Einfache Parameter

Nun liefern Funktionen, denen Sie keinen Parameter übergeben, nicht unbedingt immer den gleichen Wert zurück – Sie können ja von einer solchen Funktion auch etwa das aktuelle Datum oder die Uhrzeit ermitteln lassen oder Informationen wie den Datenbankpfad.

Allerdings erwarten die meisten Funktionen einen oder mehrere Parameter, auf deren Basis sie das Ergebnis der Funktion ermitteln. Die Parameter einer Funktion werden zunächst im Kopf der Funktion deklariert.

Dabei geben Sie standardmäßig den Datentyp an, da sonst der Datentyp **Variant** verwendet wird. Die folgende Funktion verkettet die beiden als Parameter übergebenen Zeichenketten:

```
Public Function Zeichenverkettung(_
        str1 As String, str2 As String)_
                            As String
    Zeichenverkettung = str1 & str2
End Function
```

Der Aufruf und das Ergebnis der Funktion sehen etwa wie folgt aus:

```
? Zeichenverkettung("Mon","tag")
Montag
```

Optionale Parameter

Sie können die Parameter einer Abfrage als optionale Parameter definieren. Dazu verwenden Sie das Schlüsselwort **Optional** vor dem eigentlichen Namen des Parameters. Dabei gibt es einen wichtigen Punkt zu beachten: Hinter einem optionalen Parameter dürfen nur noch weitere optionale Parameter folgen. Anderenfalls löst bereits die Definition der Kopfzeile der Funktion einen Fehler aus (s. Bild 1).

Standardwerte

Wenn Sie sicherstellen wollen, dass der Parameter auch dann einen Wert liefert, wenn die aufrufende Routine keinen Wert für den Parameter übergeben hat, können Sie wie in Listing 1 einen Standardwert angeben. Diesen tragen Sie

```
Public Function EingabeGleichAusgabe(Optional str1 As String = "Kein Wert angegeben")
    EingabeGleichAusgabe = str1
End Function
```

Listing 1: Verwendung von Standardwerten für optionale Parameter

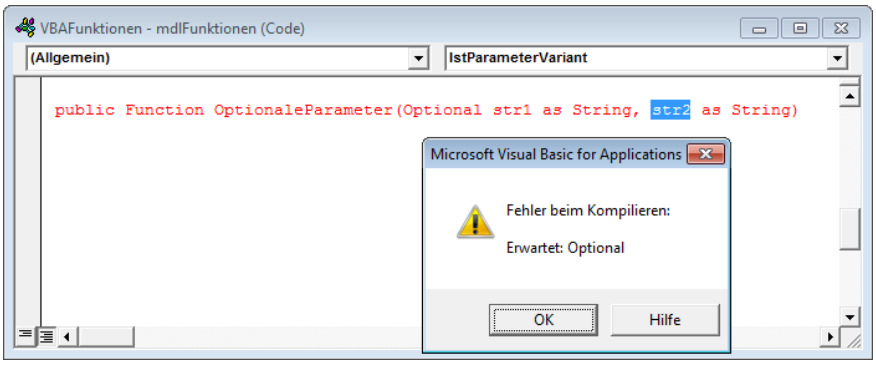

Bild 1: Start der Runtime-Version von Access

hinter dem Datentyp, sonst direkt hinter dem Namen des Parameters ein. Wenn Sie dem Parameter der Funktion beim Aufruf einen Wert übergeben, liefert sie das folgende Ergebnis:

```
? EingabeGleichAusgabe("Wert angegeben")
Wert angegeben
```

Geben Sie keinen Wert für den Parameter an, verwendet die Funktion den Standardwert:

```
? EingabeGleichAusgabe
Kein Wert angegeben
```

Auf optionale Parameter prüfen

Wenn Sie optionale Parameter verwenden, können Sie von Haus aus nur für den Datentyp Variant prüfen, ob der Benutzer diese übergeben hat. Variablen vom Typ Variant können als einziges den Wert **Null** aufnehmen. Ob der Parameter übergeben wurde, prüfen Sie mit der **IsMissing**-Funktion. Diese liefert den Wert **True** zurück, wenn der Parameter leer ist.

Übrigens ist **IsMissing** nicht gleichbedeutend mit **IsNull**. Dies können Sie mit der Funktion aus Listing 2 prüfen. Wenn Sie die Prozedur komplett ohne Parameter aufrufen, erhalten Sie folgende Ausgabe im Direktfenster:

```
Fehlt str1?   Falsch      Falsch
Fehlt var1?   Wahr        Falsch
```

Das heißt, dass eine leere **Variant**-Variable zwar als fehlend eingestuft wird, aber nicht den Wert **Null** enthält. Dies können Sie nochmals testen, indem Sie den Wert **Null** für den Parameter **var1** übergeben:

```
Debug.Print OptionaleParameter("", Null)
```

Das Ergebnis sieht so aus:

```
Fehlt str1?   Falsch      Falsch
Fehlt var1?   Falsch      Wahr
```

Die **Variant**-Variable mit dem Wert **Null** fehlt also nicht und wird mit **IsNull** korrekt als **Null** erkannt.

```
Public Function OptionaleParameter(Optional str1 As String, Optional var1 As Variant)
    Debug.Print "Fehlt str1?", IsMissing(str1), IsNull(str1)
    Debug.Print "Fehlt var1?", IsMissing(var1), IsNull(var1)
End Function
```

Listing 2: Prüfung von optionalen Parametern mit **IsMissing**

Wenn Sie einen anderen Datentyp als optionalen Parameter verwenden und prüfen wollen, ob die aufrufende Routine einen Wert für diesen übergeben hat, müssen Sie einen kleinen Trick anwenden. Das Werkzeug dafür haben Sie schon kennengelernt – den Standardwert. Geben Sie einfach einen Ausdruck als Standardwert an, der normalerweise nicht übergeben wird.

Wenn die Funktion dann aufgerufen wird, prüfen Sie, ob der Parameter den als Standardwert angegebenen Wert enthält oder einen anderen. Im ersten Fall können Sie dann davon ausgehen, dass kein Parameter übergeben wurde, anderenfalls liegt ein Wert für den Parameter vor.

Ein Beispiel für einen solchen Test zeigt die Funktion **StringparameterVorhanden** aus Listing 3. Sie verwendet die Zeichenkette **** als Standardwert. Der folgende Aufruf zeigt, wie die Funktion reagiert:

```
? StringparameterVorhanden()
Falsch
```

Übergeben Sie einen Wert, liefert die Funktion den Wert **True** zurück:

```
? StringparameterVorhanden("bla")
Wahr
```

Benannte Parameter

Wenn eine Funktion ein oder mehrere optionale Parameter enthält, können Sie diese auf verschiedene Arten angeben:

- in der angegebenen Reihenfolge, durch Kommata getrennt und vollständig bis zum letzten übergebenen Parameter oder

Multi-Add-Ins

Wer sich einen nützlichen Satz von Access-Add-Ins erstellt oder installiert hat, findet in seinem Add-In-Verzeichnis einige Datenbanken vor. Das muss nicht sein: Sie können auch mehrere Add-Ins in einer einzigen Datenbank anlegen. Dazu müssen Sie lediglich mehrere Sätze von Registry-Informationen in der Tabelle USysRegInfo anlegen und die benötigten Funktionen und Formulare bereithalten.

Wenn Sie mehrere Add-Ins in einer Add-In-Datenbank einbauen, gewinnen Sie gleich mehrfach:

- Sie brauchen nur eine Add-In-Datenbank, die Sie gegebenenfalls in einem Rutsch auf einem weiteren Rechner installieren können.
- Sie haben mehrere Anwendungen in einer, wodurch oft verwendete Funktionen wiederverwendet werden können.

Um ein Multi-Add-In zu erstellen, müssen Sie gegenüber einem einfachen Add-In kaum mehr Aufwand betreiben. Dies sind die notwendigen Schritte:

- Sie tragen nicht nur einen Satz von Registrierungs-Informationen in die Tabelle **USysRegInfo** ein, sondern gleich mehrere – jeweils mit einem eigenen Schlüssel versehen, versteht sich.
- Sie legen eine Funktion etwa namens **Startup** fest, die einen Parameter erwartet, der die Nummer oder Bezeichnung des aufzurufenden Add-Ins auswertet und das entsprechende Formular anzeigt beziehungsweise Funktion aufruft (also **=Startup(1)**, **=Startup(2)** und so weiter). Diese Funktion geben Sie samt Parameterwert in der Tabelle **USysRegInfo** unter **Expression** an.

USysRegInfo

Subkey	Type	ValName	Value
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\MultiAddIn 1	0		
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\MultiAddIn 1	1	Expression	=MultiAddIn1()
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\MultiAddIn 1	1	Description	Das erste Multi-Add-In
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\MultiAddIn 1	1	Library	|ACCDIR\MultiAddIn.mda
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Multi-Add-In 2	0		
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Multi-Add-In 2	1	Expression	=MultiAddIn2()
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Multi-Add-In 2	1	Description	Das erste Multi-Add-In
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Multi-Add-In 2	1	Library	|ACCDIR\MultiAddIn.mda
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Multi-Add-In 3	0		
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Multi-Add-In 3	1	Expression	=MultiAddIn3()
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Multi-Add-In 3	1	Description	Das drittte Multi-Add-In
HKEY_CURRENT_ACCESS_PROFILE\Menu Add-Ins\Multi-Add-In 3	1	Library	|ACCDIR\MultiAddIn.mda

Datensatz: 1 von 12 | Kein Filter | Suchen

Bild 1: Die Tabelle **USysRegInfo** mit den Registrierungsinformationen für drei Add-Ins

- Oder Sie legen für jedes Add-In eine eigene Funktion fest, also etwa **=Startup1()**, **=Startup2()** et cetera – so müssen Sie keinen Parameter übergeben. Dafür legen Sie für jedes enthaltene Add-In eine entsprechende Funktion an.

MultiAddIn - mdlAddIn (Code)

(Allgemein) (Deklarationen)

```
Option Compare Database
Option Explicit

Public Function MultiAddIn1()
    MsgBox "MultiAddIn 1"
End Function

Public Function MultiAddIn2()
    MsgBox "MultiAddIn 2"
End Function

Public Function MultiAddIn3()
    MsgBox "MultiAddIn 3"
End Function
```

Bild 2: Die VBA-Funktionen, die durch die verschiedenen Add-In-Menü-Einträge aufgerufen werden

- Die Datenbankeigenschaften, die im Add-In-Manager angezeigt werden, müssen dann gleich alle enthaltenen Add-Ins beziehungsweise die Add-In-Sammlung beschreiben.

Beispiel-Multi-Add-In

Die Add-In-Datenbank **MultiAddIn.mda** enthält drei Sätze von Einträgen in der Tabelle **USysRegInfo** (s. Bild 1). Wichtig ist, dass Sie für ein Add-In jeweils identische Werte im **Feld Subkey** anlegen. Hier können Sie als Platzhalter für den Registry-Bereich auf dem aktuellen Rechner den Wert **HKEY_CURRENT_ACCESS_PROFILE** angeben – dieser wird beim Installieren durch den entsprechenden Bereich in der Registry ersetzt. Für **Expression** geben Sie den Namen der aufzurufenden Funktion mit führendem Gleichheitszeichen und nachfolgendem Klammernpaar an – je nach Bedarf mit Parameterwerten. Für die Datensätze mit dem Wert **Library** im Feld **ValName** geben Sie unter **Value** jeweils den Namen der aktuellen Datenbank an – gefolgt von einem weiteren Platzhalter namens

IACCDIR. Dieser wird bei der Installation durch den Add-In-Pfad für den aktuellen Benutzer ersetzt (etwa **C:\Users\Andre\AppData\Roaming\Microsoft\AddIns**).

VBA-Funktionen

Die Funktionen, die durch diese drei Add-In-Menüeinträge aufgerufen werden, finden Sie in Bild 2. Wir haben hier zu Testzwecken nur **MsgBox**-Anweisungen hinterlegt. Diese würden Sie in der Praxis durch die Aufrufe entsprechender Funktionen oder Formulare ersetzen.

Eigenschaften

Wenn Sie die Datenbank-Eigenschaften öffnen und dort zur Registerseite **Zusammenfassung** wechseln, finden Sie einige Eigenschaften vor (s. Bild 3). Diese werden später nach der Installation des Add-Ins im Add-In-Manager angezeigt. Das ist der einzige Nachteil von Multi-Add-Ins: Sie zeigen im Add-In-Manager alle die gleichen Informationen an, nämlich die der aktuellen Datenbankdatei. Dies können wir allerdings verkraften, denn wer kontrolliert schon die Eigenschaften von Add-Ins im Add-In-Manager?

Multi-Add-Ins per Add-In erstellen

Im Beitrag **Add-In-Creator** (**www.access-im-unternehmen.de/907**) haben wir ein Add-In vorgestellt, das die Erstellung von Add-Ins vereinfacht. Mit diesem können Sie die Eigenschaften verschiedener Add-Ins verwalten und diese auch nacheinander in die Tabelle **USysRegInfo** der auszustattenden Add-In-Datenbank schreiben (s. Bild 4).

Sie legen hier über die Schaltfläche **Neu anlegen** einfach nacheinander die gewünschten Add-Ins an. Dabei tragen Sie im oberen Bereich (**Basisdaten**) die Werte für die Tabelle **USysRegInfo** ein und fügen diese jeweils mit **Daten in USysRegInfo eintragen** ein. Die im unteren Bereich angegebenen Informationen sollten für alle Add-Ins gleich sein und brauchen nur einmal hinzugefügt zu werden.

Nach der Installation eines solchen Add-Ins zeigt das Add-In-Menü alle darin enthaltenen Add-Ins an (s. Bild 5).

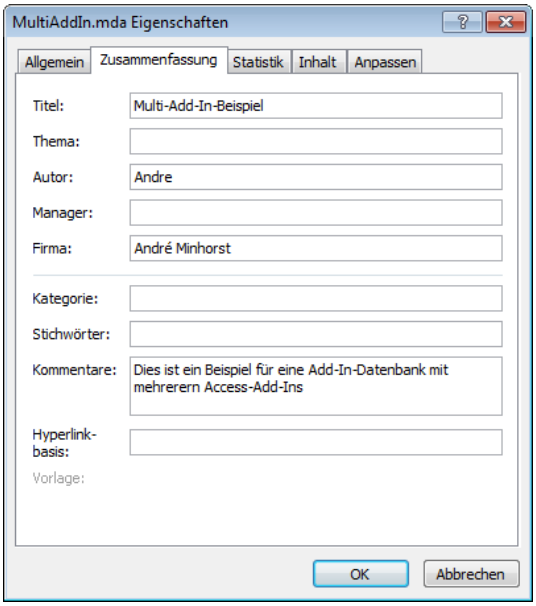

Bild 3: Eigenschaften, die im Add-In-Manager angezeigt werden

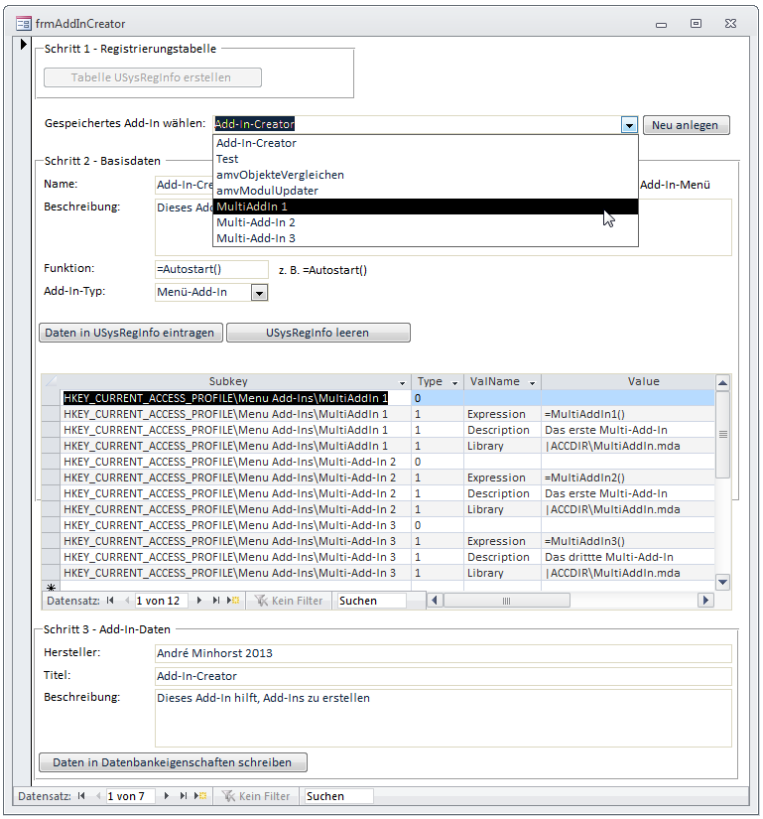

Bild 4: Erstellen von Multi-Add-Ins mit dem Add-In-Creator

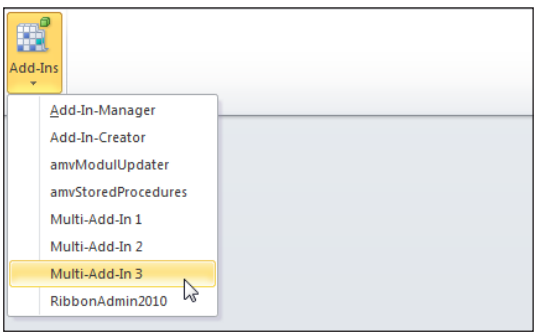

Bild 5: Menü-Einträge für die neuen Add-Ins

Datenmodelle vergleichen

Spätestens dann, wenn Sie eine Datenbank zum Kunden geben, gibt es zwei Versionen der gleichen Datenbank. Meist geschieht dies bereits viel früher – zum Beispiel, wenn Sie eine Sicherheitskopie eines bestimmten Entwicklungsstands der Datenbank anfertigen, um ohne Sorge Änderungen vornehmen zu können. Was aber, wenn Sie etwa nach Arbeiten am Datenmodell noch einmal prüfen möchten, wie die Unterschiede zum Datenmodell einer anderen Version der Datenbank aussehen? Genau: Dann hilft Ihnen die Lösung aus dem vorliegenden Beitrag weiter.

Datenmodell ändern

Es kann immer mal vorkommen, dass Sie unvorsichtigerweise Änderungen am Datenmodell vornehmen. Sie sollten dann anschließend allerdings noch nachvollziehen können, welche Änderungen dies waren. Aber wie das im Entwickleralltag so ist, gerät man in einem Programmierrausch und vergisst, die Änderungen zu dokumentieren. Schwamm drüber: Wenn sich die Möglichkeit bietet, Änderungen im Nachhinein durch einen Vergleich nachvollziehen zu können, kann man sich die Dokumentation doch sparen. Fehlt also nur noch ein geeignetes Tool, um die Datenmodelle zweier Datenbanken abzugleichen. Und dieses programmieren wir in diesem Beitrag.

Vorgehensweise

Es gibt verschiedene Vorgehensweisen für den Vergleich zweier Datenmodelle. Man könnte Tabelle für Tabelle, Feld für Feld und Eigenschaft für Eigenschaft durchlaufen und Änderungen herausschreiben. Alternativ schreibt man erstmal den Status Quo der aktuellen und der ursprünglichen Datenbank in Tabellen und vergleicht diese dann. In diesem Beitrag wollen wir die letztere Variante ausprobieren: Nach der Auswahl der Datenbanken sollen deren Informationen gleich in einem Satz von Tabellen landen. Nach dem Einlesen dieser Informationen für die betroffenen Datenbanken durchlaufen wir dann die angefallenen Daten.

Die Anwendung im Einsatz

Bild 1 zeigt das Formular der Lösung zu diesem Beitrag in der Formularansicht. Das Formular besteht aus den folgenden Elementen:

- Das Textfeld **txtDatei1** zeigt die erste der beiden zu vergleichenden Dateien an.
- Das Textfeld **txtDatei2** arbeitet analog zum Textfeld **txtDatei1**.
- Die Schaltflächen **cmdDatei1** und **cmdDatei2** rufen einen **Datei öffnen**-Dialog aus und tragen den Pfad zur ausgewählten Datei in die beiden Textfelder ein. Beide Schaltflächen stellen

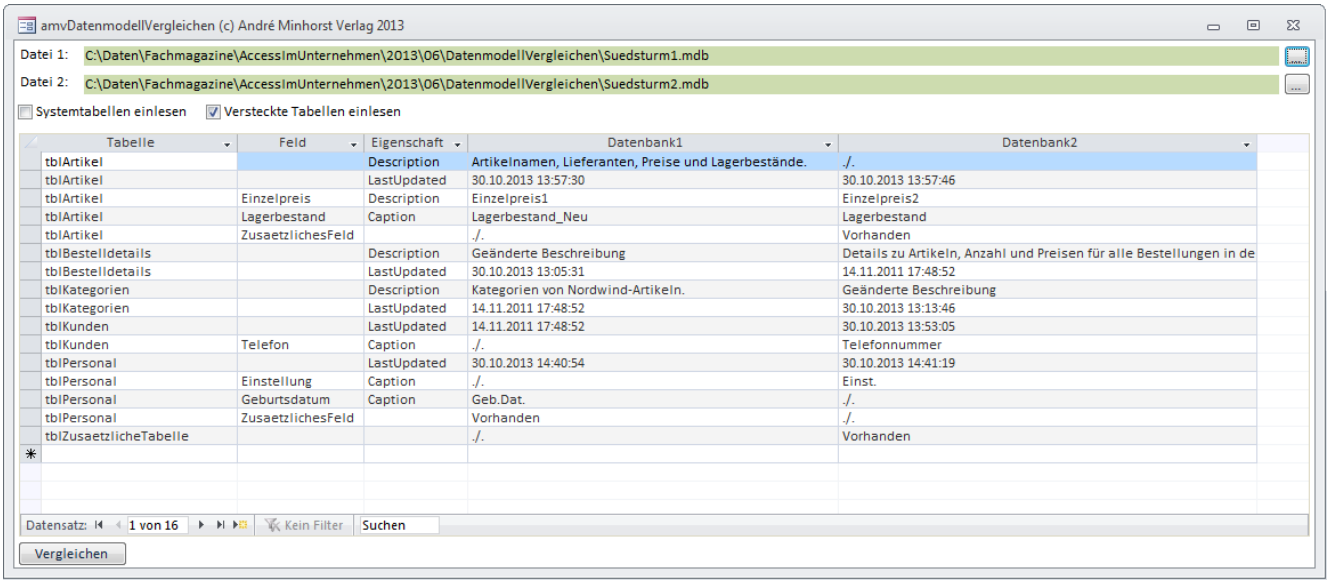

Tabelle	Feld	Eigenschaft	Datenbank1	Datenbank2
tblArtikel		Description	Artikelnamen, Lieferanten, Preise und Lagerbestände.	./.
tblArtikel		LastUpdated	30.10.2013 13:57:30	30.10.2013 13:57:46
tblArtikel	Einzelpreis	Description	Einzelpreis1	Einzelpreis2
tblArtikel	Lagerbestand	Caption	Lagerbestand_Neu	Lagerbestand
tblArtikel	ZusaetzlichesFeld		./.	Vorhanden
tblBestelldetails		Description	Geänderte Beschreibung	Details zu Artikeln, Anzahl und Preisen für alle Bestellungen in de
tblBestelldetails		LastUpdated	30.10.2013 13:05:31	14.11.2011 17:48:52
tblKategorien		Description	Kategorien von Nordwind-Artikeln.	Geänderte Beschreibung
tblKategorien		LastUpdated	14.11.2011 17:48:52	30.10.2013 13:13:46
tblKunden		LastUpdated	14.11.2011 17:48:52	30.10.2013 13:53:05
tblKunden	Telefon	Caption	./.	Telefonnummer
tblPersonal		LastUpdated	30.10.2013 14:40:54	30.10.2013 14:41:19
tblPersonal	Einstellung	Caption	./.	Einst.
tblPersonal	Geburtsdatum	Caption	Geb.Dat.	./.
tblPersonal	ZusaetzlichesFeld		Vorhanden	./.
tblZusaetzlicheTabelle			./.	Vorhanden

Bild 1: Die Lösung zum Vergleichen von Datenmodellen

nicht nur den Pfad in den Textfeldern ein, sondern lesen gleichzeitig auch bereits die Tabellen, Felder, Tabelleneigenschaften und Feldeigenschaften der ausgewählten Datenbank ein.

- Mit dem Kontrollkästchen **ctlSystemtabellen** legt der Benutzer fest, ob Systemtabellen beim Einlesen berücksichtigt werden sollen.

- Mit dem Kontrollkästchen **ctlVersteckteTabellen** legt der Benutzer fest, ob als versteckt markierte Tabellen eingelesen werden sollen.

- Das Unterformular zeigt die Unterschiede zwischen den beiden Datenmodellen an.

- Die Schaltfläche **cmdVergleich** startet den Vergleich der beiden Datenmodelle und zeigt das Ergebnis im Unterformular an.

Datenmodell der Lösung

Das Datenmodell ist hierarchisch aufgebaut. Die oberste Ebene nimmt die die Tabelle **tblDatenbanken** ein, die aus den beiden Feldern **DatenbankID** und **Datenbank** besteht, wobei **DatenbankID** den Part des Primärschlüsselfeldes übernimmt (s. Bild 2).

Die Tabelle **tblTabellen** nimmt alle Tabellen der beiden beteiligten Datenbanken auf. Dazu stellt sie die Felder **TabelleID** als Primärschlüsselfeld, **Tabelle** mit dem Tabellennamen sowie **Systemtabelle** und **VersteckteTabelle** als **Ja/Nein**-Felder, die anzeigen, ob die Tabelle eine Systemtabelle oder eine versteckte Tabelle ist, zur Verfügung. Außerdem referenziert sie über das Fremdschlüsselfeld **DatenbankID** die Tabelle **tblDatenbanken** und legt somit fest, zu welcher Datenbank die Tabelle gehört (s. Bild 3).

Die einzelnen Felder einer jeden Tabelle speichert die Anwendung in der Tabelle **tblFelder** (s. Bild 4). Neben dem Primärschlüsselfeld **FeldID** und dem Feld **Feld** zum Speichern des Feldnamens enthält auch diese Tabelle wieder ein Fremdschlüsselfeld zum Herstellen der übergeordneten Beziehung, in diesem Fall zur Tabelle **tblTabellen**.

Nun fehlen noch die Eigenschaften der Tabelle sowie der einzelnen Felder. Diese speichert die Anwendung in zwei weiteren Tabellen. Die Tabelle **tblTabelleneigenschaften** speichert alle Eigenschaften der mit dem Fremdschlüsselfeld **TabelleID** referenzierten Tabelle aus **tblTabellen**, die Tabelle **tblFeldeigenschaften** erledigt das Gleiche für die Felder. Dabei enthält die Tabelle **tblTabelleneigenschaften** neben dem Primär- und dem Fremdschlüsselfeld noch zwei Felder namens **Tabelleneigenschaft** und **Eigenschaftswert** (s. Bild 5).

In der Tabelle **tblFeldeigenschaften** heißen die entsprechenden Felder **Feldeigenschaft** und **Eigenschaftswert** (s. Bild 6).

tblDatenbanken

Feldname	Felddatentyp	Beschreibung
DatenbankID	AutoWert	Primärschlüsselwert
Datenbank	Text	Pfad der Datenbank

Bild 2: Entwurf der Tabelle **tblDatenbanken**

tblTabellen

Feldname	Felddatentyp	Beschreibung
TabelleID	AutoWert	Primärschlüsselfeld
Tabelle	Text	Name der Tabelle
DatenbankID	Zahl	Fremdschlüsselwert zu tblDatenbanken
Systemtabelle	Ja/Nein	Ist Systemtabelle?
VersteckteTabelle	Ja/Nein	Ist versteckteTabelle?

Bild 3: Entwurf der Tabelle **tblTabellen**

tblFelder

Feldname	Felddatentyp	Beschreibung
FeldID	AutoWert	Primärschlüsselfeld
Feld	Text	Name des Feldes
TabelleID	Zahl	Fremdschlüsselfeld zu tblTabellen

Bild 4: Entwurf der Tabelle **tblFelder**

tblTabelleneigenschaften

Feldname	Felddatentyp	Beschreibung
TabelleneigenschaftID	AutoWert	Primärschlüsselfeld
Tabelleneigenschaft	Text	Name der Tabelleneigenschaft
Eigenschaftswert	Text	Wert der Eigenschaft
TabelleID	Zahl	Fremdschlüsselwert zu tblTabellen

Bild 5: Entwurf der Tabelle **tblTabelleneigenschaften**

tblFeldeigenschaften

Feldname	Felddatentyp	Beschreibung
FeldeigenschaftID	AutoWert	Primärschlüsselfeld
Feldeigenschaft	Text	Name der Feldeigenschaft
Eigenschaftswert	Text	Wert der Eigenschaft
FeldID	Zahl	Fremdschlüsselfeld zu tblFelder

Bild 6: Entwurf der Tabelle **tblFeldeigenschaften**

Referenzielle Integrität mit Löschweitergabe

Im Überblick sieht das Datenmodell wie in Bild 7 aus. Wichtig ist an dieser Stelle, dass wir für alle Beziehungen zwischen den Tabellen referenzielle Integrität mit Löschweitergabe definiert haben. Wenn die nachfolgenden Routinen dann einen Datensatz der Tabelle **tblDatenbanken** löschen, um die Daten zu dieser Datenbank erneut einzulesen, entfernt Access automatisch auch die Daten aller damit verknüpften Tabellen.

Optionen speichern

Das Formular ist an die Tabelle **tblOptionen** gebunden. Diese enthält Felder wie **Datei1**, **Datei2**, **SystemtabellenEinle-**

sen und **VersteckteTabellenEinlesen**. Die Textfelder **txtDatei1** und **txtDatei2** sowie die Kontrollkästchen **chkSystemtabellen** und **chkVersteckteTabellen** sind an diese Felder gebunden. Auf diese Weise merkt sich die Lösung die zuletzt verwendeten Einstellungen, was gerade beim Entwickeln einer solchen Lösung praktisch ist.

Datenmodell einlesen

Nachdem die Tabellen zum Erfassen der Elemente des Datenmodells, also Datenbanken, Tabellen, Felder, Tabelleneigenschaften und Feldeigenschaften, erstellt sind, können wir uns an die Füllung derselben begeben. Dies geht mit der Vorstellung einiger Details des Formulars **frmDatenmodellVergleichen** einher, das im Entwurf wie in Bild 8 aussieht.

Die Schaltfläche **cmdDatei1** ruft beim Anklicken die Prozedur aus Listing 1 auf. Diese Prozedur verwendet die Funktion **OpenFileName**, die Sie im Modul **mdlTools** der Datenbank finden, um einen **Datei öffnen**-Dialog anzuzeigen, mit dem der Benutzer die erste der beiden zu vergleichenden Datenbanken auswählt. Wenn der Benutzer im Dialog die **Abbrechen**-Taste betätigt, wird **Me!txtDatei** geleert. Die anschließende Auswertung des Inhalts von **txtDatei1** mit der **Dir**-Funktion und der Vergleich der Länge der ermittelten Zeichenkette liefert nur den Wert **True**, wenn es sich um eine gültige Datei handelt. Nur in diesem Fall ruft die Prozedur dann die Routine **DatenbankEinlesen** mit dem Pfad zur Datei als Parameter auf.

Die zweite Schaltfläche **cmdDatei2** arbeitet fast genauso – der einzige Unterschied ist, dass sie das Textfeld **txtDatei2** füllt.

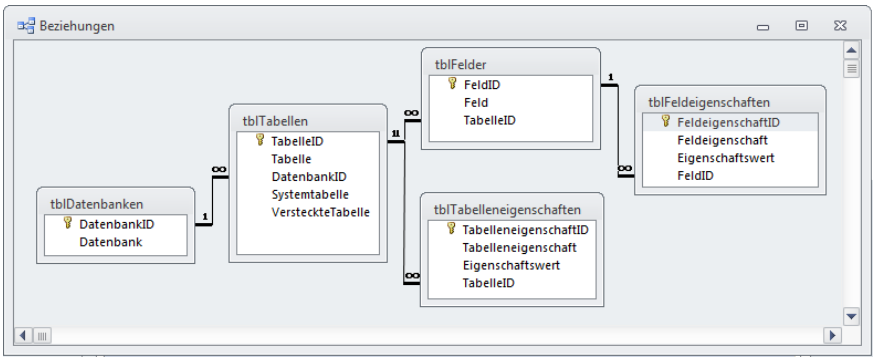

Bild 7: Datenmodell der Anwendung in der Übersicht

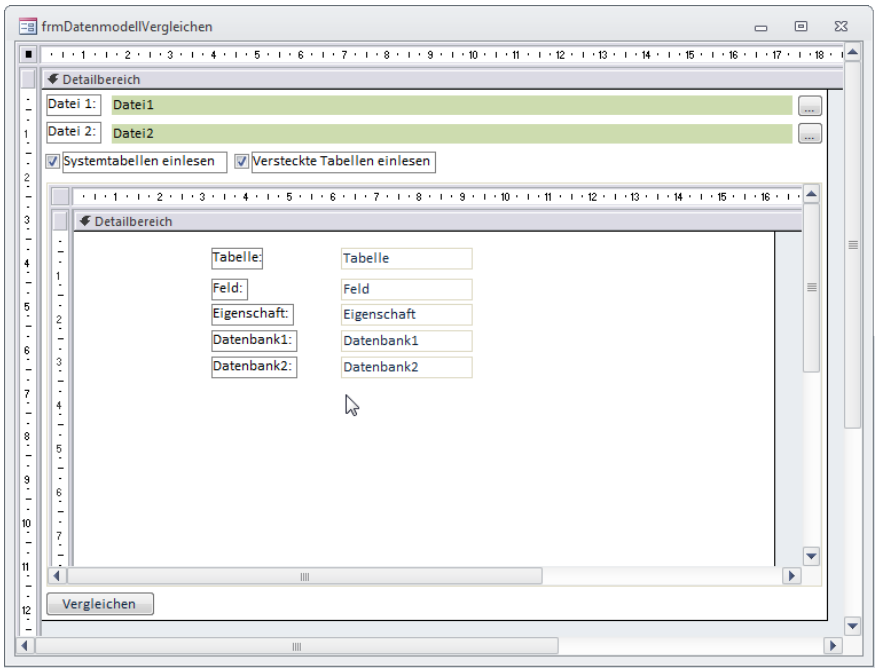

Bild 8: Das Formular **frmDatenmodellVergleichen** in der Entwurfsansicht

```
Private Sub cmdDatei1_Click()
    Me!txtDatei1 = OpenFileName(CurrentProject.Path, "Textdatei auswählen", _
        "Access-Datenbank (*.mdb;*.accdb;*.mda,*.accda)|Alle Dateien (*.*)")
    If Len(Dir(Me!txtDatei1)) > 0 Then
        DatenbankEinlesen Me!txtDatei1
    End If
End Sub
```

Listing 1: Auswählen der Datenbankdatei und Starten des Einlesevorgangs

Datenbank einlesen

Die Routine **DatenbankEinlesen**, die den Pfad zur einzulesenden Datenbank als Parameter erwartet, aktiviert zunächst die Sanduhr, da der Vorgang bereits für kleinere Datenmodelle ein paar Sekunden dauern kann (in aktuellen System erscheint natürlich längst keine Sanduhr mehr – aber Sie wissen ja, was gemeint ist). Die Prozedur

```
Private Sub DatenbankEinlesen(strDatenbank As String)
    Dim db As DAO.Database
    Dim dbQuelle As DAO.Database
    Dim lngDatenbankID As Long
    Set db = CurrentDb
    DoCmd.Hourglass True
    Set dbQuelle = OpenDatabase(strDatenbank, , True)
    db.Execute "DELETE FROM tblDatenbanken WHERE Datenbank = '" & dbQuelle.Name & "'", dbFailOnError
    db.Execute "INSERT INTO tblDatenbanken(Datenbank) VALUES('" & dbQuelle.Name & "')", dbFailOnError
    lngDatenbankID = db.OpenRecordset("SELECT @@IDENTITY").Fields(0)
    TabellenEinlesen db, dbQuelle, lngDatenbankID
    DoCmd.Hourglass False
End Sub
```

Listing 2: Starten des Einlesevorgangs der als Parameter übergebenen Datei

```
Private Sub TabellenEinlesen(db As DAO.Database, dbQuelle As DAO.Database, lngDatenbankID As Long)
    Dim tdf As DAO.TableDef
    Dim intSystemtabelle As Integer
    Dim intVersteckteTabelle As Integer
    Dim lngTabelleID As Long
    Dim bolEinlesen As Boolean
    For Each tdf In dbQuelle.TableDefs
        bolEinlesen = True
        SysCmd acSysCmdSetStatus, "Tabelle '" & tdf.Name & "'"
        intSystemtabelle = Not ((tdf.Attributes And dbSystemObject) = 0)
        intVersteckteTabelle = Not ((tdf.Attributes And dbHiddenObject) = 0)
        If Me!chkSystemtabellen = False Then
            If intSystemtabelle = True Then
                bolEinlesen = False
            End If
        End If
        If Me!chkVersteckteTabellen = False Then
            If intVersteckteTabelle = True Then
                bolEinlesen = False
            End If
        End If
        If bolEinlesen = True Then
            db.Execute "INSERT INTO tblTabellen(Tabelle, DatenbankID, Systemtabelle, VersteckteTabelle) VALUES('" _
                & tdf.Name & "', " & lngDatenbankID & ", " & intSystemtabelle & ", " & intVersteckteTabelle & ")"
            lngTabelleID = db.OpenRecordset("SELECT @@IDENTITY").Fields(0)
            FelderEinlesen db, tdf, lngTabelleID
            TabelleneigenschaftenEinlesen db, tdf, lngTabelleID
        End If
    Next tdf
    SysCmd acSysCmdClearStatus
End Sub
```

Listing 3: Einlesen der Tabellen

```
Private Sub FelderEinlesen(db As DAO.Database, tdf As DAO.TableDef, lngTabelleID As Long)
    Dim fld As DAO.Field
    Dim lngFeldID As Long
    For Each fld In tdf.Fields
        SysCmd acSysCmdSetStatus, "Tabelle '" & tdf.Name & "', Feld '" & fld.Name & "'"
        db.Execute "INSERT INTO tblFelder(Feld, TabelleID) VALUES('" & fld.Name & "', " & lngTabelleID & ")", dbFailOnError
        lngFeldID = db.OpenRecordset("SELECT @@IDENTITY").Fields(0)
        FeldeigenschaftenEinlesen db, tdf, fld, lngFeldID
    Next fld
End Sub
```

Listing 4: Einlesen der Felder der Tabelle mit dem Primärschlüsselwert **lngTabelleID**

deklariert zwei **Database**-Variablen: **db** referenziert die aktuelle Datenbank, um mit der **Execute**-Methode Datensätze zur Tabelle zum Speichern der Unterschiede hinzufügen zu können, **dbQuelle** referenziert die zu untersuchende Datenbank.

Diese Datenbank öffnet die Prozedur mit der **OpenDatabase**-Methode. Sie übergibt dieser Methode den Pfad zur Datenbank sowie den Wert **True** für den Parameter **ReadOnly**. Dann löscht sie alle Datensätze aus der Tabelle **tblDatenbanken**, deren Feld **Datenbank** den Pfad der hinzuzufügenden Datenbank enthält.

Anschließend legt sie gleich einen entsprechenden neuen Datensatz in der Tabelle **tblDatenbanken** an und ermittelt mit der Abfrage **SELECT @@IDENTITY** den Primärschlüsselwert des neu hinzugefügten Datensatzes. Dieser wird später zum Anlegen der Datensätze in der verknüpften Tabelle **tblTabellen** benötigt. Der Aufruf der Routine **TabellenEinlesen** startet das Einlesen der Tabellen der soeben geöffneten Datenbank. Ist dieser Vorgang, der alle weiteren Aktionen anstößt, abgeschlossen, lässt die Prozedur die Sanduhr wieder verschwinden.

Tabellen einlesen

Die Prozedur **TabellenEinlesen** erwartet Verweise auf die beiden betroffenen Datenbanken, also die aktuelle und die zu untersuchende Datenbank, sowie den Primärschlüsselwert des zuvor in der Tabelle **tblDatenbanken** angelegten Datensatzes (s. Listing 3). Die Prozedur durchläuft in einer Schleife alle Elemente der **TableDefs**-Auflistung des mit **dbQuelle** referenzierten **Database**-Objekts. Dabei stellt sie jeweils zunächst die Variable **bolEinlesen** auf den Wert **True** ein und gibt einen Status über den aktuellen Bearbeitungsstand in der Statusleiste aus.

Wenn Systemtabellen ignoriert werden sollen, die aktuell mit **tdf** referenzierte Tabelle aber eine Systemtabelle ist, wird **bolEinlesen** auf **False** eingestellt – das geschieht ähnlich für versteckte **Tabellen**. Ob es sich um eine Systemtabelle oder um eine versteckte Tabelle handelt, erfahren wir durch einen Vergleich des Wertes der Eigenschaft **Attributes** der Tabelle mit dem Wert der Konstanten **dbSystemobject** beziehungsweise **dbHiddenObject**. Hat **bolEinlesen** anschließend noch den Wert **True**, fügt die Prozedur der Tabelle **tblTabellen** einen entsprechenden Datensatz hinzu und speichert den Wert des Primärschlüsselfeldes des neuen Datensatzes in der Variablen **lngTabelleID**. Anschließend ruft sie die beiden Prozeduren **FelderEinlesen** und **TabelleneigenschaftenEinlesen** auf.

Felder einlesen

Die erste der beiden genannten Prozeduren erwartet wiederum zwei Objektvariablen. Dabei handelt es sich wiederum um den Verweis auf die aktuelle Datenbank.

Die Quelldatenbank ist diesmal nicht gefragt, denn wir benötigen ja Zugriff auf die Felder der übergeordneten Tabelle. Daher übergeben wir mit **tdf** einen Verweis auf das entsprechende **TableDef**-Objekt und außerdem den Primärschlüsselwert des zuvor angelegten Datensatzes der Tabelle **tblTabellen**.

Die Prozedur **FelderAnlegen** durchläuft schlicht über die Variable **fld** die **Fields**-Auflistung des **TableDef**-Objekts **tdf**. Dabei aktualisiert sie wiederum die Statuszeile von Access, trägt einen entsprechenden Datensatz in die Tabelle **tblFelder** ein, ermittelt den Primärschlüsselwert des neuen Datensatzes und ruft dann die Prozedur auf, die sich um das Speichern der Eigenschaften der Felder kümmert (s. Listing 4).

```
Private Sub FeldeigenschaftenEinlesen(db As DAO.Database, tdf As DAO.TableDef, fld As DAO.Field, lngFeldID As Long)
    Dim prp As DAO.Property
    Dim strEigenschaft As String
    Dim strWert As String
    For Each prp In fld.Properties
        SysCmd acSysCmdSetStatus, "Tabelle '" & tdf.Name & "', Feld '" & fld.Name & "', Eigenschaft '" & prp.Name & "'"
        Select Case prp.Name
            Case "FieldSize", "ForeignName", "OriginalValue", "ValidateOnSet", "Value", "VisibleValue"
            Case Else
                strEigenschaft = prp.Name
                strWert = prp.Value
                db.Execute "INSERT INTO tblFeldeigenschaften(Feldeigenschaft, Eigenschaftswert, FeldID) VALUES('" _
                    & strEigenschaft & "', '" & Replace(strWert, "'", "''") & "', " & lngFeldID & ")", dbFailOnError
        End Select
    Next prp
End Sub
```

Listing 5: Einlesen der Feldeigenschaften

Feldeigenschaften einlesen

Die Prozedur FeldeigenschaftenEinlesen wird nun für jedes Feld einmal aufgerufen.

Damit wir in der Statusmeldung einen Ausdruck der Form **Tabelle <Tabellenname>, Feld <Feldname>, Eigenschaft <Eigenschaftname>** ausgeben können, benötigt die Prozedur als Parameter zusätzlich zum Verweis auf die aktuelle Datenbank (**db**) Verweise auf das **TableDef**-Objekt und das **Field**-Objekt. Und natürlich darf der obligatorische Wert mit dem Primärschlüsselwert des übergeordneten Datensatzes aus der Tabelle **tblFelder** nicht fehlen (**lngFeldID**) – s. Listing 5.

Die Prozedur durchläuft alle Elemente der **Properties**-Auflistung des **Field**-Objekts. Diesmal sollen ja nicht nur Namen gespeichert werden – wie bei der Datenbank, den Tabellen und den Feldern, – sondern die Bezeichnungen der Eigenschaften sowie die entsprechenden Werte. Bei den Eigenschaften des **Field**-Objekts gibt es die folgende Besonderheit: Das **Field**-Objekt kommt gleich an mehreren Stellen im **DAO**-Objektmodell vor. Allerdings werden nicht alle Eigenschaften des **Field**-Objekts, die über die **Properties**-Auflistung verfügbar sind, in jedem Kontext genutzt.

So fallen etwa beim **Field**-Objekt der **Fields**-Auflistung einer Tabelle die Eigenschaften **FieldSize**, **ForeignName**, **OriginalValue**, **ValidateOnSet**, **Value** und **VisibleValue** unter den Tisch. Und nicht nur das: Wenn Sie versuchen, auf diese Eigenschaften über die **Properties**-Auflistung zuzugreifen, löst dies einen Fehler aus. Also prüfen wir vorab den Namen des aktuellen Elements der **Properties**-Auflistung in einer **Select Case**-Bedingung und ignorieren so die oben genannten Elemente.

Schließlich schreibt die Prozedur für jede Eigenschaft des Tabellenfeldes einen neuen Datensatz in die Tabelle **tblFeldeigenschaften**. Den neuen Primärschlüsselwert müssen wir uns nicht mehr merken, denn es gibt keine untergeordneten Elemente mehr.

Tabelleneigenschaften einlesen

Fehlt nur noch eine Art von Information: Die Eigenschaften der Tabellen selbst. Diese ermittelt die Prozedur **TabelleneigenschaftenEinlesen** aus Listing 6. Diese durchläuft wiederum alle **Property**-Elemente der **Properties**-Aufistung des **TableDef**-Objekts.

Hier gibt es eine Besonderheit: Es kann nämlich vorkommen, dass Eigenschaften wie **NameMap** den Datentyp **dbLongBinary** aufweisen, deren Inhalt sich nicht so einfach in ein Textfeld schreiben lässt. Deshalb ignoriert die Prozedur nach einer Prüfung auf **prp.Type = dbLongBinary** gleich komplett.

Die Namen und Werte der Eigenschaften landen dann per **INSERT INTO**-Aktionsabfrage wie gewohnt in der Zieltabelle, in diesem Fall **tblTabelleneigenschaften**.

Datenmodell vergleichen

Um die durch die nachfolgend beschriebenen Prozeduren ermittelten Unterschiede zwischen den Datenmodellen