HAUFE.
Ausgabe 04/2026

ACCESS
IM UNTERNEHMEN

ABHÄNGIGE KOMBINATIONSFELDER

Nutzen Sie abhängige Kombinationsfelder – sogar in Endlosformularen (ab Seite 14).

In diesem Heft:

FILTERFORMULAR MIT SPEICHERFUNKTION

Erweitern Sie das Filterformular aus 3/2026 um Kombinationsfeld-Filter und das Speichern von Filtern.

SEITE 27

ZOOMEN IN FORMULAREN

Lernen Sie die neuen Funktionen von Access zum Zoomen in Formularen in der Formular- und Endlos-Ansicht kennen.

SEITE 2

MODULE AUTOMATISCH SPEICHERN

Nutzen Sie unser COM-Add-In zum automatischen Sichern von Formularen, Berichten und Modulen.

SEITE 60

Impressum

ISBN 978-3-8092-1496-0 ISSN: 1616-5535 Mat.-Nr. 01583-5154
Access im Unternehmen

Kommanditgesellschaft, Sitz Freiburg
Registergericht Freiburg, HRA 4408
Komplementäre: Haufe-Lexware Verwaltungs GmbH, Sitz Freiburg,
Registergericht Freiburg, HRB 5557; Martin Laqua
Geschäftsführung der Komplementär-GmbH: Dr. Carsten Block, Iris Bode, Frank Enders, Matthias Schätzle, Carsten Schröder, Christian Steiger

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

Redaktion: Dipl.-Ing. André Minhorst (Chefredakteur, V.i.S.d.P.),
Sabine Wißler (Redaktionsassistentin), Carsten Gromberg (sprachl. Lektorat und Fachlektorat)
Herausgeber: Dipl.-Ing. André Minhorst
Autoren: Dipl.-Ing. Christoph Jüngling, Dipl.-Ing. André Minhorst

Anschrift der Redaktion: Postfach 100121, 79120 Freiburg, Tel. 0761/898-0, Fax: 0761/898-3990,
E-Mail: computer@haufe.de, Internet: https://www.haufe.de

Druck: Franz X. Stückle Druck und Verlag e.K., Ettenheim

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

Abhängige Kombinationsfelder in Endlosformularen – endlich gelöst

Mit »abhängigen Kombinationsfeldern« sorgen wir dafür, dass wir per 1:n-Beziehung verknüpfte Daten mit zwei Kombinationsfeldern auswählen können. Das erste wählt beispielsweise eine Kategorie, und im zweiten erscheinen anschließend nur noch die Produkt-Einträge, die zu der gewählten Kategorie gehören. Das ist für Einzelformulare bereits gelöst, aber in Endlosformularen gelingt dies nicht ohne Probleme bei der Darstellung. In dieser Ausgabe präsentieren wir einen Ansatz, der es nun ermöglich.

Zum Thema »Abhängige Kombinationsfelder« finden Sie in dieser Ausgabe gleich zwei Beiträge. Im ersten namens **Abhängige Kombinationsfelder** schauen wir uns an, wie wir grundsätzlich ein Kombinationsfeld zum Filtern der Einträge eines weiteren Kombinationsfeldes nutzen (ab Seite 14).

Im zweiten Beitrag zu diesem Thema mit dem Titel **Gebundene abhängige Kombinationsfelder** gehen wir ab Seite 18 ins Eingemachte: Wir stellen die Probleme bei der Darstellung von abhängigen Kombinationsfeldern in Endlosformularen dar und zeigen eine Lösung dafür auf.

Gute Neuigkeiten gibt es für alle, die sich immer wieder fragen, ob Access überhaupt noch eine Zukunft hat. Das beste Zeichen dafür, dass Access nicht von der Bildfläche verschwindet, ist die aktuelle Aktivität des Microsoft-Entwicklerteams. Es meldet in letzter Zeit immer wieder, dass an neuen Funktionen gearbeitet wird. Im Beitrag **Formulare zoomen in Access** (ab Seite 2) schauen wir uns eine der Funktionen an, die vor Kurzem veröffentlicht wurde: Wir können nun Formulare in der Einzel- und in der Endlosansicht zoomen! Damit können Benutzer die Größe von Texten, Steuerelementen et cetera an ihre Bedürfnisse anpassen.

Eine weitere Neuerung ist eine VBA-Erweiterung, mit der wir nun wichtige Informationen über die aktuelle Installation von Access abfragen können. Dazu gehören die Version und die Architektur (32/64-Bit). Beides konnte man bisher nur umständlich über die Benutzeroberfläche abfragen. Dies ist nun mit neuen Konstanten für die **SysCmd**-Funktion möglich. Welche das sind und wie Sie diese nutzen, zeigen wir im Beitrag **Access: Version und Architektur einfach ermitteln** ab Seite 7.

Unter **Access-Infos per COM-Add-In anzeigen** liefern wir ab Seite 49 ein Add-In, mit dem Sie die Version und die Architektur komfortabel im Datei-Menü von Access anzeigen können.

Das Filterformular, das wir in 3/2026 vorgestellt haben, entwickeln wir stetig weiter. Im Beitrag **Filterformular erweitern: Kombinationsfelder** zeigen wir ab Seite 27, wie wir nach den Inhalten von Kombinationsfeldern filtern können. Und unter dem Titel **Filterformular erweitern: Filter speichern** lesen Sie ab Seite 35, wie Sie die zusammengestellten Filterkriterien speichern und wieder aufrufen können. So gehen aufwendig erstellte Filter nicht verloren.

Schließlich haben wir noch ein weiteres Add-In, das Änderungen am Code während der Entwicklung automatisch speichert – damit verlieren Sie keine Arbeit, wenn Access einmal abstürzt (siehe **VBA-Module regelmäßig sichern per COM-Add-In** ab Seite 60).

Viel Spaß beim Lesen wünscht Ihnen

Ihr André Minhorst

Formulare zoomen in Access

Allen Unkenrufen zum Trotz: Microsoft Access wird immer noch weiterentwickelt. Vor kurzem wurde ein neues Feature veröffentlich, das nun schrittweise ausgerollt wird: Eine Zoom-Funktion für Datenblattansichten von Tabellen und Abfragen und von Formularen in der Formularansicht! In diesem Beitrag schauen wir uns die neue Funktion an. Wir zeigen, wie die Zoom-Funktion genutzt werden kann und welche Limitierungen sie noch hat.

Welche Versionen bieten die Zoom-Funktion an?

Zunächst einmal ein Blick auf die Versionen von Access, die ab jetzt die Zoom-Funktion anbieten.

Hier wird aktuell nur Access aus dem Office 365-Abonnement unterstützt, und zwar ab Version 2605.

Welche Elemente können wir zoomen?

Aktuell lassen sich folgende Elemente der Access-Benutzeroberfläche zoomen:

- Tabellen und Abfragen in der Datenblattansicht
- Formulare in der Formular- und in der Datenblattansicht

Außerdem berücksichtigt die Zoom-Funktion nur native Access-Steuerelemente.

ActiveX-Steuerelemente wie **TreeView** oder **ListView** und Steuerelemente von Drittanbietern werden nicht unterstützt.

Außerdem werden aktuell zum Beispiel die Entwurfsansicht und die Endlos-Ansicht von Formularen noch nicht von der Zoom-Funktion unterstützt. Dies soll jedoch noch nachgereicht werden.

Wie nutzt man die Zoom-Funktion?

Es gibt verschiedene Möglichkeiten, die Zoom-Funktion zu nutzen:

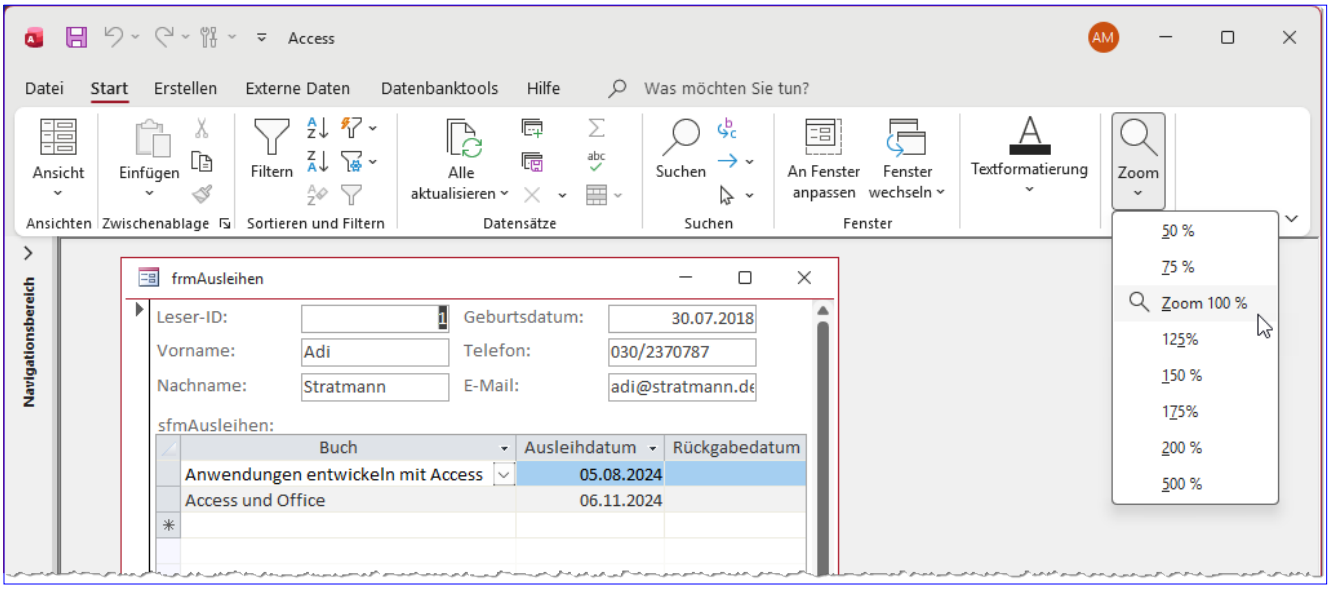

Bild 1: Zoomen per Ribbon-Eintrag

- Über das Ribbon
- Über die Statusleiste
- Mit der Tastenkombination **Strg + Alt + Plus** zum Vergrößern und **Strg + Alt + Minus** zum Verkleinern
- Mit den Tasten **Strg + Alt** und dem Scrollrad der Maus
- Mit VBA-Befehlen wie **DoCmd.RunCommand acCmdZoom150**
- Und wir können einen Standard-Zoom-Faktor über die Access-Optionen festlegen.

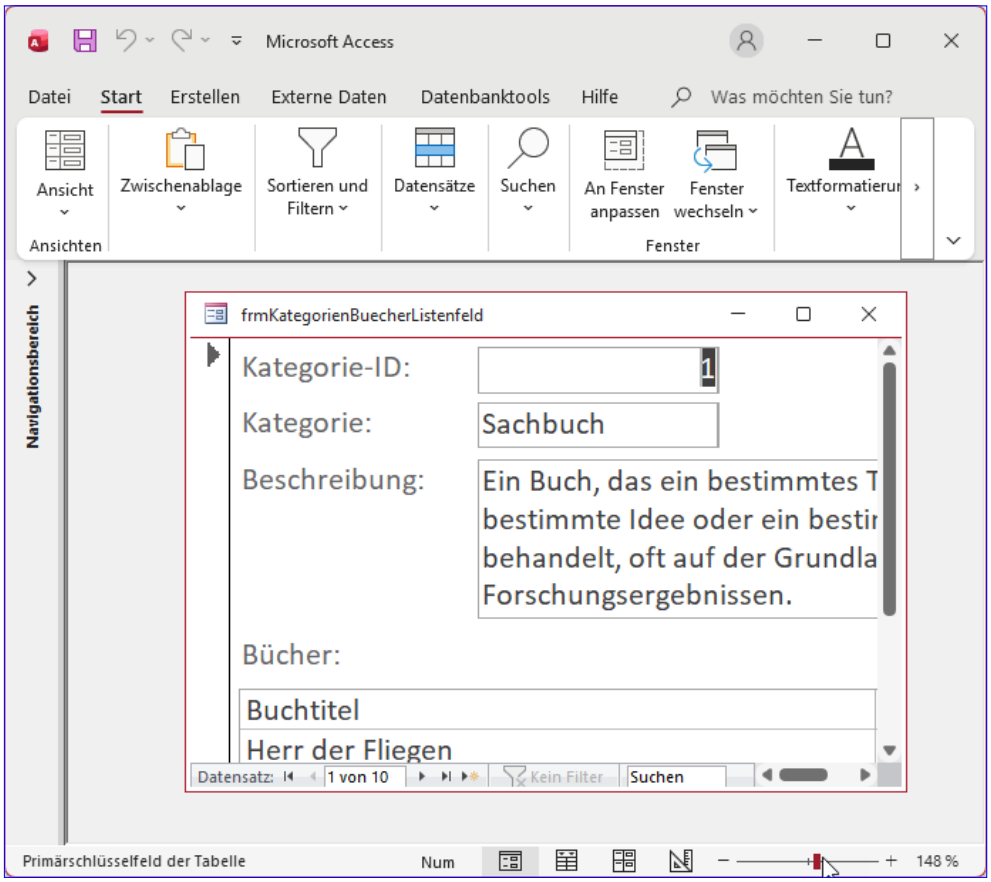

Bild 2: Zoomen per Statusleiste

Zoomen per Ribbon

Die offensichtlichste Möglichkeit zum Zoomen ist der neue Eintrag **Zoom** im Start-Ribbon von Access. Hier können wir die Zoom-Faktoren **50%**, **75%**, **100%**, **125%**, **150%**, **175%**, **200%** und **500%** auswählen (siehe Bild 1).

Zoomen über die Statusleiste

Die nächste Variante, den Zoom-Faktor über die Benutzeroberfläche einzustellen, finden wir unten rechts in der Statusleiste (siehe Bild 2). Hier können wir stufenlos zoomen – im Beispiel auf 148%.

Zoomen per Tastenkombination

Spannender, weil viel schneller auszuführen, ist das Zoomen über Tastenkombinationen. Mit der Tastenkombination **Strg + Alt + Plus** können wir den Zoom-Faktor vergrößern. Mit **Strg + Alt + Minus** verringern wir den Zoom-Faktor. Mit beiden Varianten ändern wir den Zoom-Faktor jeweils um 10 Prozent.

Mit der Tastenkombination **Strg + Alt + 0** stellen wir den Zoom-Faktor wieder auf 100 Prozent ein.

Zoomen mit dem Mausrad

Wenn die rechte Hand ohnehin gerade die Maus bedient, können wir auch diese zum Zoomen nutzen. Dazu halten wir die Tastenkombination **Strg + Alt** gedrückt und scrollen mit dem Mausrad, um den Zoom-Faktor um jeweils 10 Prozent nach oben oder unten zu korrigieren.

Achtung: Hier muss sich der Mauszeiger im Hauptformular befinden. Wenn das Formular ein Unterformular enthält und die Maus sich über diesem befindet, funktioniert das Zoomen nicht.

Zoomen per VBA

Außerdem können wir den Zoom-Faktor per VBA einstellen. Dazu gibt es eine Reihe neuer RunCommand-Konstanten, die wir beispielsweise im Objektkatalog einsehen können (siehe Bild 3).

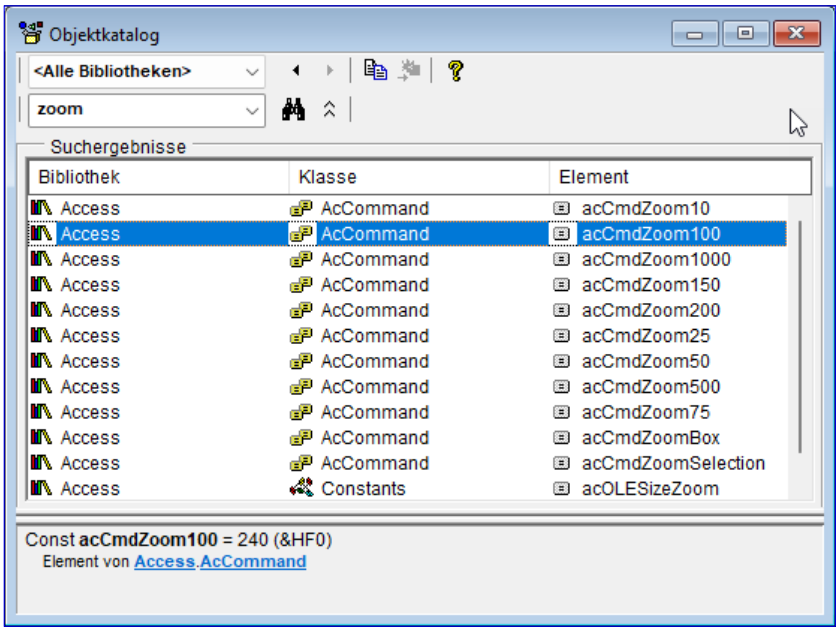

Bild 3: Zoom-Konstanten für den **RunCommand**-Befehl

Hier finden wir die folgenden Konstanten:

- **acCmdZoom10**
- **acCmdZoom25**
- **acCmdZoom50**
- **acCmdZoom75**
- **acCmdZoom100**
- **acCmdZoom150**
- **acCmdZoom200**
- **acCmdZoom500**

Es funktionieren aber nur die Werte von 50 Prozent bis 500 Prozent für Formulare, die anderen sind nur für Berichte nutzbar. Die Werte **acCmdZoom10** und **acCmdZoom25** stellen den Zoom-Faktor auf 50 Prozent ein.

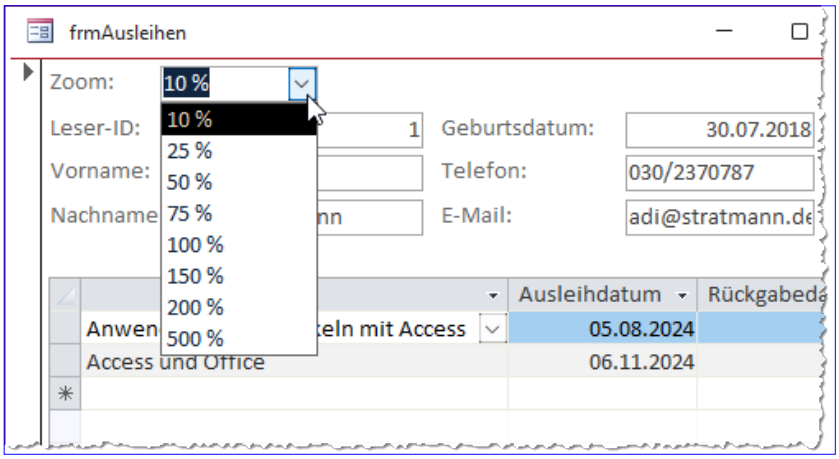

Bild 4: Kombinationsfeld zur Auswahl des Zoom-Faktors

Wir haben dem Beispielformular ein Kombinationsfeld hinzugefügt, mit dem sich der Zoom-Faktor auswählen lässt (siehe Bild 4).

Dazu haben wir die Eigenschaft **Herkunftsart** auf **Wertliste** und die Eigenschaft **Datensatzherkunft** auf folgenden Ausdruck eingestellt:

```
10;"10 %";25;"25 %";50;"50 %";75;"75 %";100;"100 %";150;"150 %";200;"200 %";500;"500 %"
```

Wir haben auch die Werte für 10 Prozent und 25 Prozent übernommen und dann herausgefunden, dass diese zu einem Zoom-Faktor von 50 Prozent führen.

Damit nur die Prozent-Werte angezeigt werden, haben wir außerdem **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** festgelegt.

Für die Ereigniseigenschaft **Nach Aktualisierung** wurde außerdem die folgende Ereignisprozedur hinterlegt:

```
Private Sub cboZoom_AfterUpdate()
    Select Case Me.cboZoom
        Case 10
            Application.RunCommand acCmdZoom10
        Case 25
            Application.RunCommand acCmdZoom25
        Case 50
            Application.RunCommand acCmdZoom50
        Case 75
            Application.RunCommand acCmdZoom75
        Case 100
            Application.RunCommand acCmdZoom100
        Case 150
            Application.RunCommand acCmdZoom150
        Case 200
            Application.RunCommand acCmdZoom200
        Case 500
            Application.RunCommand acCmdZoom500
    End Select
End Sub
```

Dies liest jeweils den Wert der ersten Spalte aus und legt mit **Application.RunCommand accmdZoomXXX** den jeweiligen Zoom-Faktor fest.

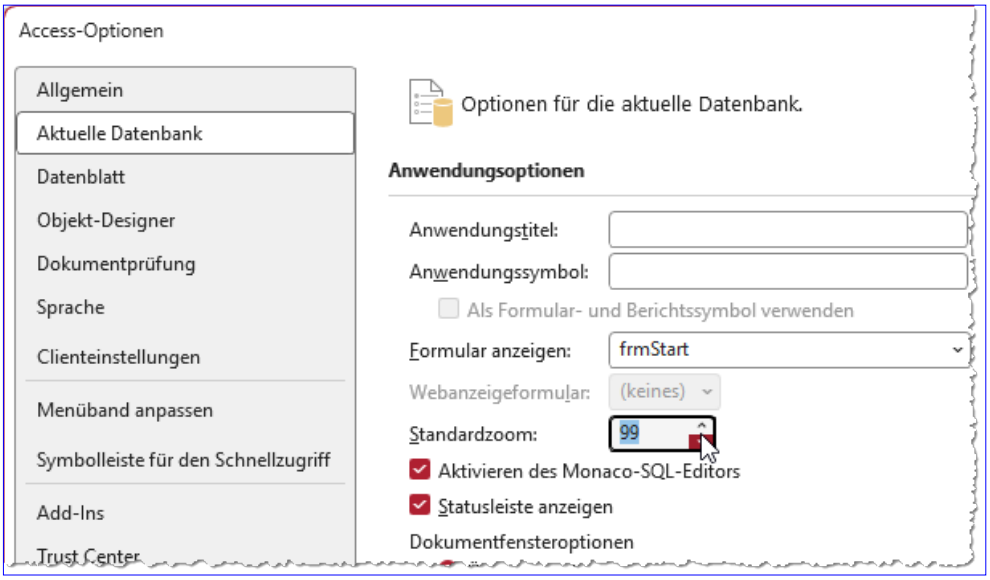

Bild 5: Zoom-Faktor über die Access-Optionen einstellen

Hier zeigen sich, zumindest auf unserem System, erste Schwächen – hat man den Zoom-Faktor einmal geändert, kann man die Werte des Kombinationsfeldes nicht immer korrekt auswählen.

Leider gibt es derzeit auch noch keine Möglichkeit, den aktuellen Zoom-Faktor auszulesen, sodass sich Änderungen zum Beispiel über die Tastenkombinationen nicht im aktuellen Wert des Kombinationsfeldes niederschlagen.

Festlegen des Standard-Zoomfaktors

In den Access-Optionen können wir den Standard-Zoom-Faktor für die Anwendung auswählen. Hier können wir wiederum auf ein Prozent genau den gewünschten Zoom-Faktor einstellen (siehe Bild 5).

Datenblätter und Formulare werden nun beim Öffnen direkt mit dem gewählten Zoom-Faktor angezeigt.

Offene Baustellen

Die von Microsoft präsentierte neue Funktion lässt noch einige Wünsche offen. Wir haben noch einige unschöne Effekte feststellen können. Zum Beispiel gibt es in der Datenblattansicht einige unschöne Artefakte, die sich bei einigen Zoom-Faktoren zum Beispiel in Unterformularen in der Datenblattansicht zeigen (siehe Bild 6).

Außerdem fehlt noch die Umsetzung beispielsweise für die Entwurfsansicht und für Endlosformulare.

Schließlich wäre es sehr hilfreich, wenn es ein Ereignis gäbe, das beim Ändern des Zoom-Faktors ausgelöst wird. So könnte man auf Änderungen reagieren

und beispielsweise Steuerelemente wie **TreeView** und **ListView** per VBA-Code anpassen – etwa um die Schriftgröße entsprechend einzustellen.

Zusammenfassung und Ausblick

Microsoft hat hier einen lang gehegten Wunsch der Access-Community umgesetzt – leider vorerst nur teilweise und noch nicht konsequent. Allerdings können wir der Roadmap entnehmen, dass weiter an diesem Feature gearbeitet wird und wir noch nicht die finale Ausbaustufe sehen. Die Roadmap finden Sie unter diesem Link:

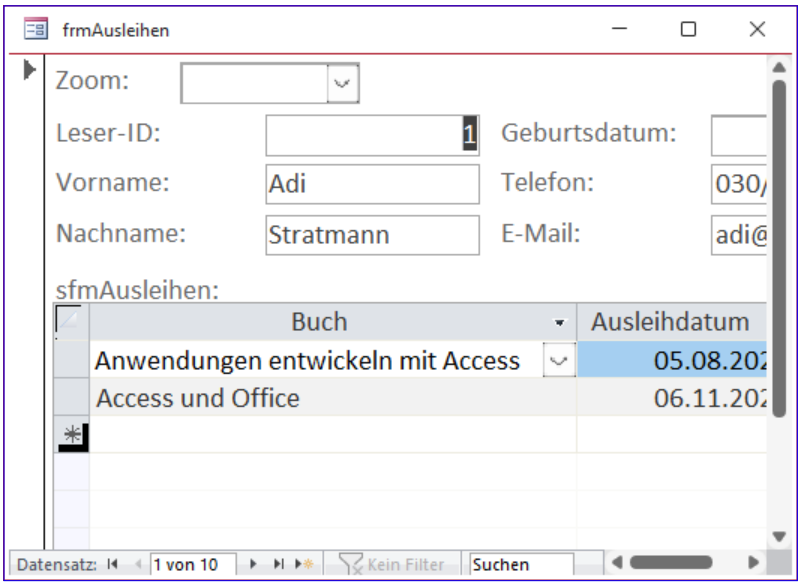

Bild 6: Unschöne Artefakte in der Datenblattansicht von Unterformularen

`https://www.microsoft.com/en-us/microsoft-365/roadmap?msockid=0223bd328fb962831b3babca-8e8b6337&filters=%5B%22Access%22%5D#Roadmap`

Hier finden wir auch noch weitere, für die nähere Zukunft geplante Features für Microsoft Access (siehe Bild 7).

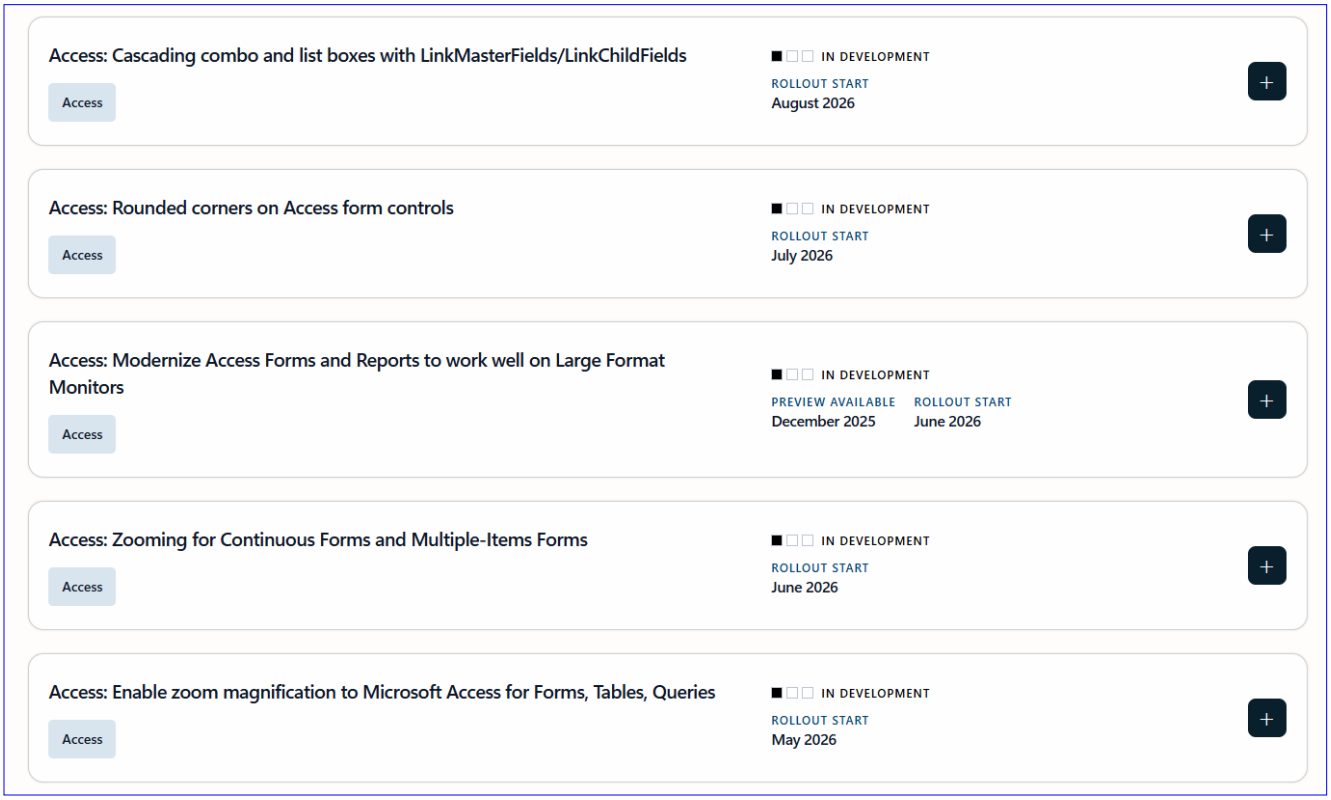

Bild 7: Roadmap für Microsoft Access

Access: Version und Architektur einfach ermitteln

Wenn man für einen Kunden eine neue Anwendung programmieren oder diesem ein Tool wie etwa ein Access-Add-In, ein COM-Add-In oder auch eine Bibliotheksdatenbank zur Verfügung stellen möchte, benötigt man einige grundlegende Informationen zu der verwendeten Access-Version und der Architektur – also ob die 32- oder die 64-Bit-Version im Einsatz ist. Bislang konnte man diese Informationen nur umständlich über mehrere Dialoge ermitteln. Nun bietet Microsoft einige neue Elemente für VBA an, mit denen wir diese Eigenschaften per Einzeiler auslesen können. In diesem Beitrag stellen wir den bisherigen Weg vor und zeigen auch die neuen VBA-Elemente. Außerdem erstellen wir ein kleines COM-Add-In, mit dem sich diese Informationen ganz einfach über den Datei-Bereich von Access abrufen lassen.

Version und Architektur – wofür benötigt man diese Informationen?

Microsoft Access kommt in verschiedenen Versionen und Architekturen. Gängige Versionen sind zum Beispiel Access 2016, 2019, 2022 und 2024 oder Access aus dem Office 365-Paket. Außerdem gibt es die 32- und die 64-Bit-Architektur.

Wenn man einfach mit der vorliegenden Version programmiert, muss man der Version und der Architektur normalerweise nur wenig Aufmerksamkeit schenken – außer, man programmiert die Windows-API mit den entsprechenden VBA-Funktionen. Wichtiger wird die Kenntnis der Version und der Architektur, wenn man Datenbankanwendungen oder Tools für Kunden entwickelt.

Bezüglich der Access-Version gilt hier, dass man immer die Version des Kunden oder eine frühere Version nutzen sollte. Verwendet man eine neuere Version, baut man eventuell Funktionen ein, die in der älteren Version des Kunden nicht vorhanden sind.

Version und Architektur bei kompilierten Datenbankdateien

Sowohl die Version als auch die Architektur werden wichtig, wenn man eine Anwendung für den Kunden programmiert und ihm diese als kompilierte Version ausliefern möchte. Unter kompilierter Version verstehen wir die als **.accde**- beziehungsweise **.mde**-Datei gespeicherte Version, die nur noch ausführbaren Code enthält und keinen Einblick in den VBA-Code mehr erlaubt. Auch die Änderung des Entwurfs der Datenbankobjekte ist hier nicht mehr möglich.

Hier ist wichtig, dass man die Datenbank mit einer Access-Version kompiliert, deren Versionsnummer kleiner oder gleich der Version ist, die der Kunde verwendet. Nutzt der Kunde also Access 2016, müssen wir die Datenbank mit Access 2016 oder älter kompilieren.

Bild 1: Meldung beim Öffnen einer **.accde**-Datei mit der falschen Access-Architektur

Hier kommt auch die Architektur ins Spiel. Wir müssen die Datenbank mit einer Access-Version kompilieren, deren Architektur mit der Access-Version auf dem Zielrechner übereinstimmt.

Verwendet der Kunde Access beispielsweise in der 32-Bit-Version, müssen wir mit Access in der 32-Bit-Version kompilieren.

Wenn wir etwa eine für 64-Bit kompilierte **.accde**-Datei öffnen, erhalten wir die Fehlermeldung aus Bild 1.

Verwendung von Bibliotheksdatenbanken

Das Gleiche gilt, wenn Sie eine Bibliotheksdatenbank erstellen, die im einfachsten Fall beispielsweise VBA-Funktionen enthält, aber auch Formulare oder Berichte bereitstellen kann.

Eine solche Bibliotheksdatenbank können wir in das VBA-Projekt einer anderen Access-Datenbank per Verweis einbinden und dann die enthaltenen Funktionen nutzen. Das ist zum Beispiel sinnvoll, wenn man oft verwendete Funktionen oder Objekte nur an einem Ort vorhalten möchte.

Auch dies funktioniert nur, wenn wir eine Access-Version zum Kompilieren verwenden, die genauso alt oder älter wie die Zielversion ist. Auch hier muss die Architektur mit der Architektur der Zielversion übereinstimmen.

Version und Architektur der aktuellen Access-Installation ermitteln

Um die korrekte Version und Architektur für das Erstellen einer Anwendung für den Kunden zu ermitteln, können wir einfach den Kunden fragen. In vielen Fällen weiß er, welche Access-Version installiert ist und welche Architektur vorliegt.

Oft ist dies jedoch nicht der Fall. Dann können wir dem Kunden zeigen, wo er die benötigten Informationen findet. In einer geöffneten Access-Datenbank finden wir diese in aktuelleren Access-Versionen wie folgt:

- Klick auf den Ribbon-Bereich **Datei**

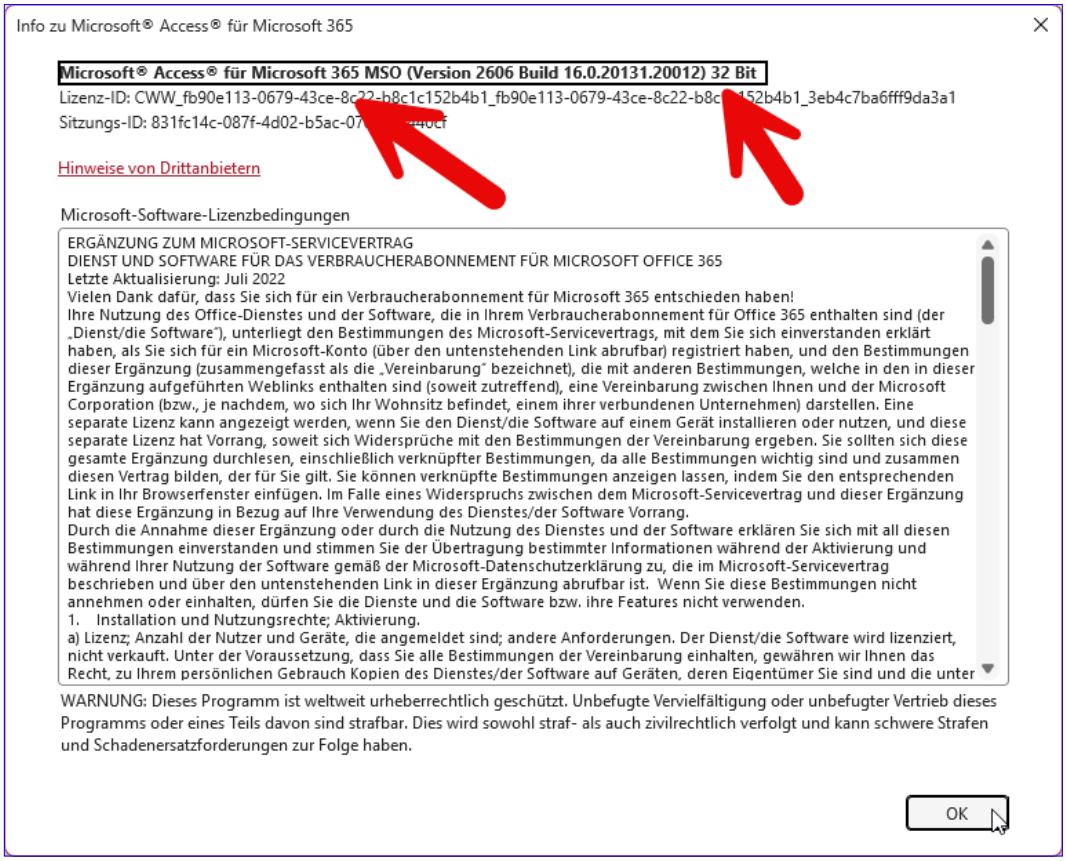

Bild 2: Auslesen von Version und Architektur

- Wechseln zum Bereich **Konto**
- Betätigen der Schaltfläche **Info zu Access**

Nun erscheint der Info-Dialog aus Bild 2, wo wir sowohl die Access-Version als auch die Architektur auslesen können. Außerdem finden wir hier weitere Informationen wie beispielsweise die eigentliche Version (hier **2606**).

Auslesen von Access-Informationen per VBA

Diese Informationen können wir nun in neueren Access-Versionen auch einfach per VBA ermitteln.

Laut Microsoft sind die Konstanten, die wir nachfolgend vorstellen, ab der Version 2604 von Microsoft 365 enthalten, aber nicht in den Versionen LTSC 2021 oder LTSC 2024, weil diese bereits vor dem Feature veröffentlicht wurden.

Auf der Seite von Colin Riddington finden wir außerdem den Hinweis, dass wir diese Konstanten auch in Access 2016 bis 2024 finden (nur in den Click-To-Run-Editionen):

`https://www.isladogs.co.uk/sys-cmd-actions/index.html`

Neue Konstanten für SysCmd

Die **SysCmd**-Funktion gibt es schon lange. Sie bietet verschiedene Konstanten an, mit denen wir verschiedene Einstellungen auslesen und auch Elemente der Benutzeroberfläche steuern können, zum Beispiel die Anzeige des Fortschritts in der Statusleiste von Access.

Der Aufruf der Funktion erfolgt einfach über **SysCmd** mit einem oder mehreren Parameterwerten. Für die Abfrage der Version und der Architektur benötigen wir jedoch immer nur einen Parameter.

Die Parameter finden wir im Objektkatalog, wenn wir unter **Klassen** den Eintrag **AcSysCmdAction** auswählen (siehe Bild 3). Hier die Liste der Parameter mit dem entsprechenden Zahlenwert und der Bedeutung des zurückgelieferten Wertes:

- **acSysCmdGetMsoBuildNumber** (**715**): **MSO.dll**-Hauptbuildnummer (**Long**)
- **acSysCmdGetFullVersion** (**720**): komplette Versionsinfo (**String**)
- **acSysCmdGetVersion** (**721**): YYMM-Version, zum Beispiel **2604** (**Long**)
- **acSysCmdGetFullBuildNumber** (**722**): voller Build-String, zum Beispiel **16.0.19922.20000**

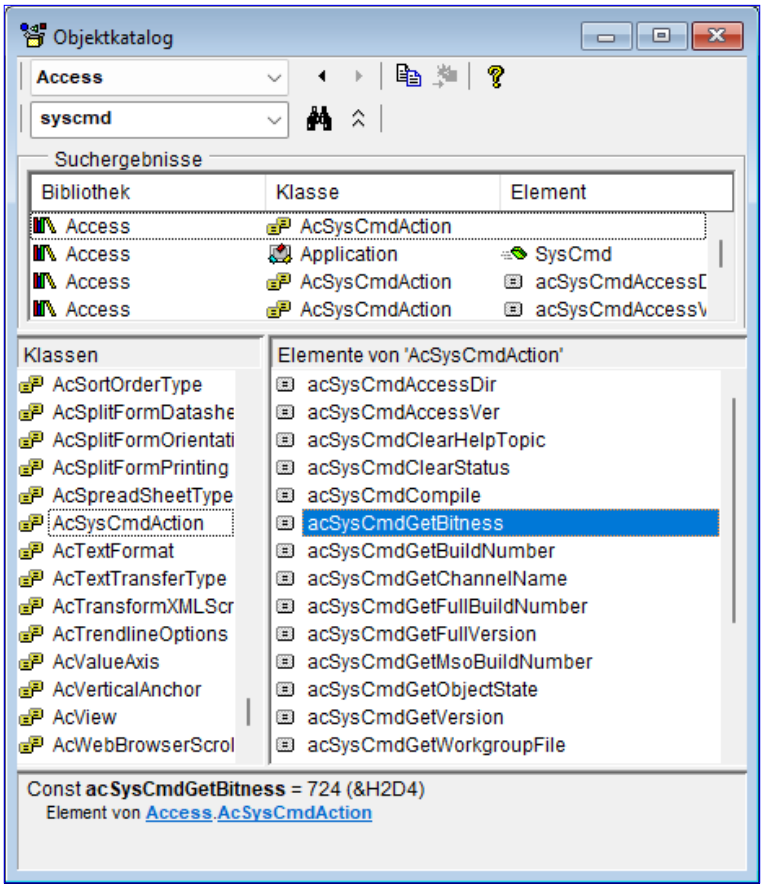

Bild 3: Die neuen Elemente im Objektkatalog

- **acSysCmdGetChannelName** (**723**): Update-Kanal (**String**)
- **acSysCmdGetBitness** (**724**): 32-bit/64-bit (String)
- **acSysCmdGetBuildNumber** (**725**): Office-Buildnummer (**Long**)

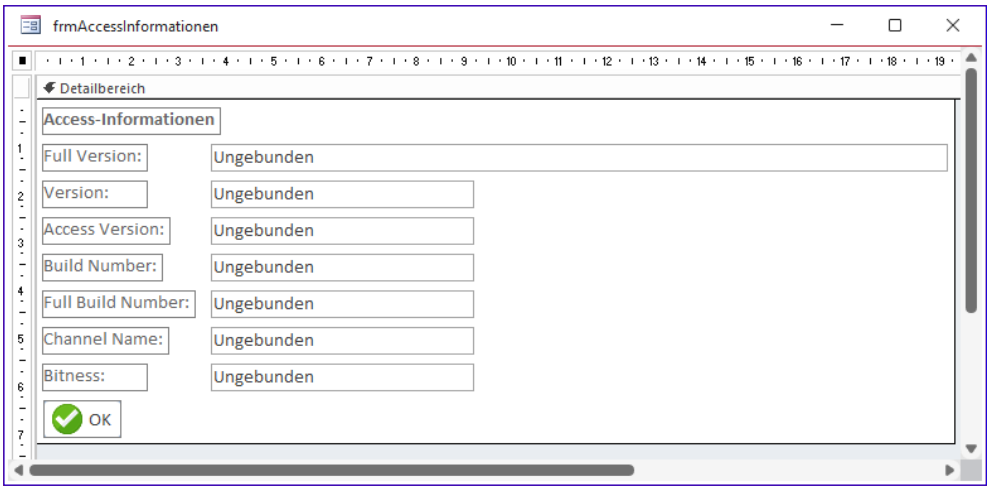

Bild 4: Entwurf eines Formulars zur Anzeige der Access-Informationen

Wenn wir nun eine der Informationen auslesen wollen, gelingt dies am einfachsten über den Direktbereich des VBA-Editors. Hier geben wir zum Beispiel zum Ermitteln der Architektur den folgenden Befehl ein:

```
Debug.Print SysCmd(acSysCmdGetBitness)
32-bit
```

Mit den übrigen Konstanten holen wir weitere Informationen:

```
Debug.Print SysCmd(acSysCmdGetMsoBuildNumber)
 20131

Debug.Print SysCmd(acSysCmdGetFullVersion)
Microsoft Access (Version 2606) Build 16.0.20131.20024
Aktueller Kanal (Vorschau) 32-bit

Debug.Print SysCmd(acSysCmdGetVersion)
2606

Debug.Print SysCmd(acSysCmdGetFullBuildNumber)
16.0.20131.20024

Debug.Print SysCmd(acSysCmdGetChannelName)
Aktueller Kanal (Vorschau)

Debug.Print SysCmd(acSysCmdGetBuildNumber)
 20131
```

Um nur die Access-Version zu holen, verwenden wir die bereits bekannte Konstante **acSysCmdAccessVer**:

```
Debug.Print SysCmd(acSysCmdAccessVer)
16.0
```

Formular zur Anzeige der Access-Informationen

Um das Einlesen und Anzeigen der Versions- und Architektur-Informationen zu demonstrieren, erstellen wir ein Formular namens **frmAccessInformationen**.

Diesem fügen wir wie in Bild 4 einige Textfelder hinzu.

Damit diese automatisch gefüllt werden, legen wir für das Ereignis **Beim Laden** des Formulars die folgende Prozedur an:

```
Private Sub Form_Load()
    Me.txtFullVersion = SysCmd(acSysCmdGetFullVersion)
    Me.txtVersion = SysCmd(acSysCmdGetVersion)
    Me.txtAccessVersion = SysCmd(acSysCmdAccessVer)
    Me.txtBuildNumber = SysCmd(acSysCmdGetBuildNumber)
    Me.txtFullBuildNumber = _
        SysCmd(acSysCmdGetFullBuildNumber)
    Me.txtChannelName = SysCmd(acSysCmdGetChannelName)
    Me.txtBitness = SysCmd(acSysCmdGetBitness)
End Sub
```

Wechseln wir zur Formularansicht, erhalten wir die gewünschten Werte in den jeweiligen Textfeldern (siehe Bild 5).

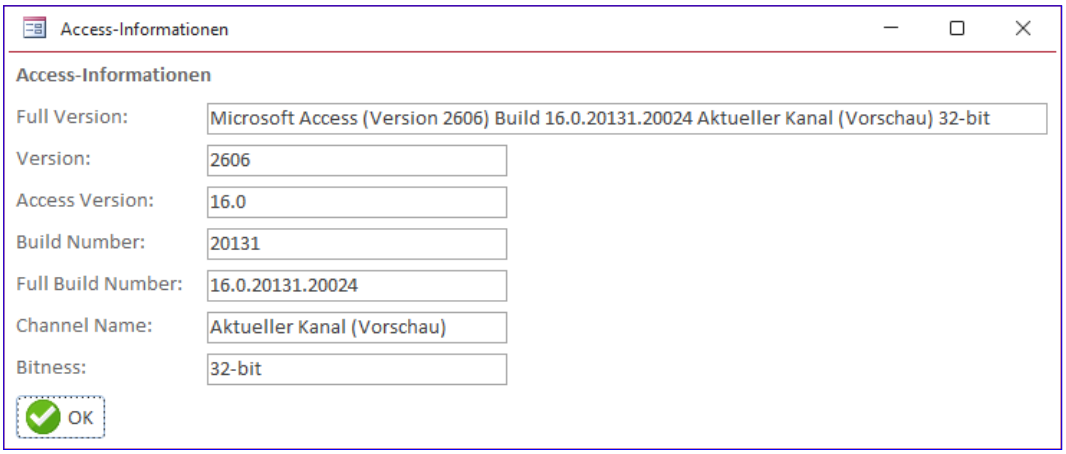

Bild 5: Formular zur Anzeige der Access-Informationen

Fehler in Access-Versionen, welche die Funktionen nicht unterstützen

Wenn wir diese Datenbank allerdings mit einer Access-Version öffnen, welche die neuen Befehle nicht unterstützt, erhalten wir die Fehlermeldung aus Bild 6.

Diesen Kompilierungsfehler können wir auch mit **On Error Resume Next** nicht umgehen.

Um zumindest das zu ermöglichen, ersetzen wir die Konstanten durch die jeweiligen Zahlenwerte.

Außerdem ergänzen wir eine rudimentäre Fehlerbehandlung, die die auftretenden Laufzeitfehler durch die ungültigen Zahlenwerte abfängt:

```
Private Sub Form_Load()
    On Error Resume Next
    Me.txtFullVersion = SysCmd(720) 'acSysCmdGetFullVersion
    Me.txtVersion = SysCmd(721) 'acSysCmdGetVersion
    Me.txtAccessVersion = SysCmd(7) 'acSysCmdAccessVer
    Me.txtBuildNumber = SysCmd(725)
        'acSysCmdGetBuildNumber
    Me.txtFullBuildNumber = SysCmd(722)
        'acSysCmdGetFullBuildNumber
    Me.txtChannelName = SysCmd(723)
        'acSysCmdGetChannelName
    Me.txtBitness = SysCmd(724) 'acSysCmdGetBitness
    Debug.Print Err.Number, Err.Description
End Sub
```

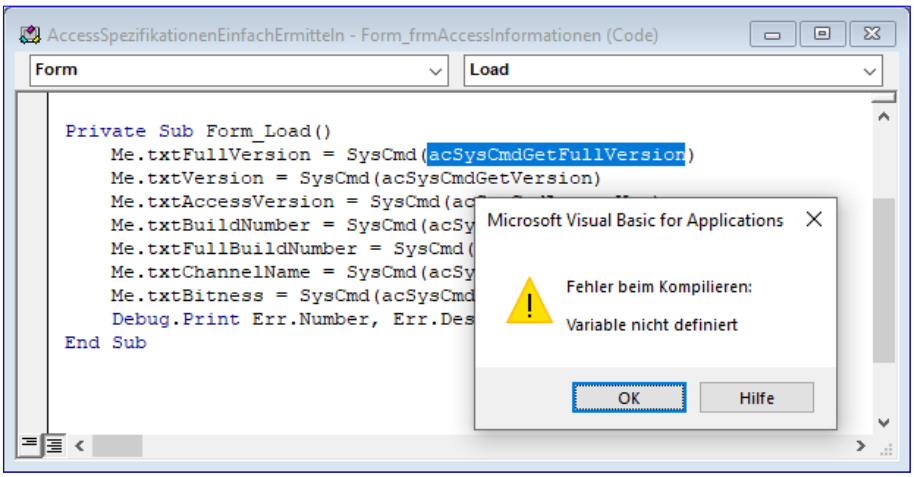

Bild 6: Fehlermeldung in nicht unterstützten Access-Versionen

Wir erhalten in Access-Versionen, welche die neuen Funktionen nicht unterstützen, den Fehler **7952** mit der Meldung **Sie haben einen ungültigen Funktionsaufruf ausgegeben**.

In diesem Fall wird im Formular nur das Feld **txtAccessVersion** gefüllt, denn nur die **SysCmd**-Konstante **acSysCmdAccessVer** wird auch in älteren Versionen unterstützt.

Alternative Funktion zum Ermitteln der Architektur von Access

Für das Weitergeben von **.accde**-Datenbanken und COM-Add-Ins beziehungsweise COM-DLLs sind die Access-Version und die Bitness die wichtigste Information. So kann man zumindest mit der korrekten Architektur kompilieren und im Falle einer älteren Ziel-Version die entsprechende Version zum Erstellen von **.accde**-Datenbanken verwenden.

Die Access-Version holen wir mit der auch in älteren Versionen unterstützen Anweisung **SysCmd(acSysCmd–AccessVer)**.

Die Bitness können wir mit der folgenden einfachen VBA-Funktion auslesen:

```
Public Function GetBitness() As String
#If Win64 Then
    GetBitness = "64-Bit"
#Else
    GetBitness = "32-Bit"
#End If
End Function
```

Sie verwendet die Kompilierungs-Konstante **Win64**, um zu prüfen, ob Access mit 64-Bit arbeitet und gibt den Wert 64-Bit zurück, wenn **Win64** den Wert **True** liefert. Anderenfalls lautet der Rückgabewert **32-Bit**.

Access-Informationen in älteren Versionen

Damit das Formular zumindest die wichtigsten Informationen in älteren Access-Versionen anzeigt, passen wir die **Form_Load**-Prozedur wie in Listing 1 an.

```
Private Sub Form_Load()
    On Error Resume Next
    Me.txtFullVersion = SysCmd(720) 'acSysCmdGetFullVersion
    Me.txtFullVersion = Nz(Me.txtFullVersion, "Nicht verfügbar.")

    Me.txtVersion = SysCmd(721) 'acSysCmdGetVersion
    Me.txtVersion = Nz(Me.txtVersion, "Nicht verfügbar.")

    Me.txtAccessVersion = SysCmd(7) 'acSysCmdAccessVer

    Me.txtBuildNumber = SysCmd(725) 'acSysCmdGetBuildNumber
    Me.txtBuildNumber = Nz(Me.txtBuildNumber, "Nicht verfügbar.")

    Me.txtFullBuildNumber = SysCmd(722) 'acSysCmdGetFullBuildNumber
    Me.txtFullBuildNumber = Nz(Me.txtFullBuildNumber, "Nicht verfügbar.")

    Me.txtChannelName = SysCmd(723) 'acSysCmdGetChannelName
    Me.txtChannelName = Nz(Me.txtChannelName, "Nicht verfügbar.")

    Me.txtBitness = SysCmd(724) 'acSysCmdGetBitness
    Me.txtBitness = Nz(Me.txtBitness, GetBitness)

End Sub
```

Listing 1: Anpassung der Prozedur für älteren Access-Versionen

Damit sorgen wir dafür, dass bei Feldern, die leer bleiben, weil die entsprechende **SysCmd**-Funktion nicht verfügbar ist, der Text **Nicht verfügbar** erscheint. Es werden lediglich die Access-Version (zum Beispiel **16.0**) und die Bitness eingetragen, die wir in diesem Fall mit der Funktion **GetBitness** eintragen.

Anwendungsfälle für das Abfragen von Version und Architektur

Der Sinn der **SysCmd**-Variante liegt nicht im Nachschauen, sondern darin, dass ein Programm die gelieferten Informationne automatisch verarbeiten kann, ohne dass ein Mensch hinschaut. Das macht in mehreren Situationen einen echten Unterschied.

Der wichtigste Fall ist die Fehlerdiagnose bei ausgelieferten

Anwendungen. Wenn ein Kunde ein Problem meldet, will man nicht erklären müssen, wie er sich durch den Konto-Dialog klickt und einem Version, Build, Kanal und Bitness vorliest. Stattdessen schreibt die Fehlerbehandlung diese Werte automatisch ins Log oder in die Fehlermail.

Dazu kommt ein praktischer Punkt, den man leicht übersieht: Die Angaben im Info-Dialog lassen sich nicht ohne Weiteres kopieren – per **SysCmd** bekommt man dagegen fertige Strings, die man speichern, vergleichen und versenden kann.

Der zweite Fall ist bedingte Logik zur Laufzeit – also Code, der sich je nach Umgebung anders verhält. Bei der Bitness ist hier allerdings eine Einschränkung wichtig, sonst verkauft man die Funktion zu gut: Für den eigenen laufenden Code kennen wir die Architektur nämlich bereits zur Kompilierzeit über **#If Win64**, da ist **acSysCmdGetBitness** überflüssig.

Der dritte Fall ist Feature-Gating über Build und Kanal. Wenn eine Funktion erst ab einem bestimmten Build existiert – die neuen **SysCmd**-Konstanten selbst sind ja das beste Beispiel –, kann der Code vorab prüfen und entweder die Funktion freischalten oder einen sauberen Hinweis ausgeben, statt Fehler **7952** zu werfen.

Genauso lässt sich der Kanal abfragen, etwa um eine Funktion nur im Beta-Kanal zu aktivieren oder umgekehrt im produktiven Current Channel zurückzuhalten.

Und schließlich die Bestandsaufnahme über viele Rechner: Ein Entwickler oder Administrator, der wissen will, welche Versionen, Kanäle und Architekturen in seiner Anwenderschaft tatsächlich im Einsatz sind, kann diese Werte per Code einsammeln und zentral auswerten. Von Hand bei jedem Rechner in den Dialog zu schauen, skaliert nicht.

Man würde hier also zum Beispiel bei jedem Start einer Anwendung, beispielsweise eines CRM-Systems, einen Eintrag in einer Tabelle im Backend generieren oder aktualisieren, der Informationen über die Access-Version und -Architektur enthält.

Über die Abfrage dieser Tabelle im Backend kann man sich dann einen Überblick über alle in Betrieb befindlichen Access-Instanzen und ihre Version und Architektur verschaffen.

Zusammenfassung und Ausblick

Die neuen **SysCmd**-Konstanten erlauben in neueren Access-Versionen das Auslesen der System-Informationen, die man sonst nur umständlich über die Benutzeroberfläche ermitteln konnte.

Der Beitrag präsentiert ein Formular, mit dem in neueren Access-Versionen alle relevanten Informationen aus den entsprechenden **SysCmd**-Konstanten ausgelesen werden können. Für ältere Versionen liefert das Formular immerhin die Access-Version und die Bitness.

Weitere Informationen könnte man zwar auch per VBA ermitteln, aber dazu sind aufwendigere Techniken wie das Auslesen der Registry erforderlich.

Da die neuen **SysCmd**-Konstanten noch nicht in allen Access-Versionen verfügbar sind und manche Versionen diese auch nicht erhalten werden, muss man dies berücksichtigen und die wichtigsten Informationen auf anderen Wegen ermitteln.

In einem weiteren Beitrag namens **Access: Version und Architektur einfach ermitteln** (**www.access-im-unternehmen.de/1606**) zeigen wir ein COM-Add-In, mit dem sich die gewünschten Informationen schnell im Datei-Bereich von Access anzeigen lassen.

Abhängige Kombinationsfelder

Ein Kombinationsfeld mit hunderten Einträgen ist mehr Last als Hilfe. Wählt der Benutzer aber zuerst eine Kategorie, lässt sich die Liste im zweiten Feld auf eine überschaubare Handvoll eindampfen. Genau das leisten abhängige Kombinationsfelder: Die Auswahl im ersten Feld bestimmt, was im zweiten zur Verfügung steht. Wie das gelingt – vom einfachen Grundgerüst bis zu den Feinheiten, die im Alltag den Unterschied machen –, zeigt der vorliegende Beitrag.

Das Datenmodell

Als Beispiel dienen zwei Tabellen. In **tblKategorien** stehen die Kategorien mit den Feldern **KategorieID** und **Kategorie**, in **tblProdukte** die Produkte mit **ProduktID**, **Produkt** und dem Fremdschlüssel **KategorieID** (siehe Bild 1). Jedes Produkt gehört damit zu genau einer Kategorie – die klassische 1:n-Beziehung.

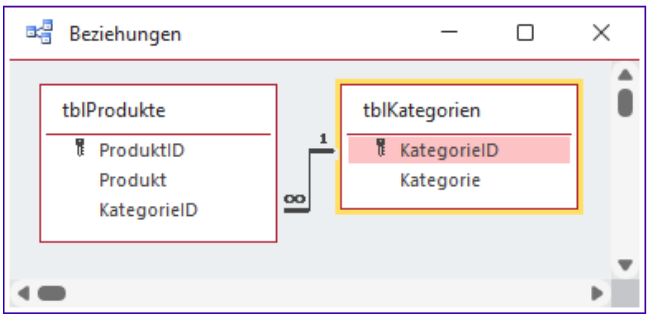

Bild 1: Das Datenmodell aus Kategorien und Produkten, verknüpft über die **KategorieID**

Die beiden Kombinationsfelder

Auf einem Formular platzieren wir zwei Kombinationsfelder, wie Bild 2 zeigt. Das erste, **cboKategorieID**, führt die Kategorien; das zweite, **cboProduktID**, später die dazu passenden Produkte. Beide arbeiten mit zwei Spalten, wobei wir die erste – die ID – über die Spaltenbreite **0** ausblenden. So sieht der Benutzer nur den Namen, während im Hintergrund der Schlüsselwert gebunden ist.

Als Datensatzherkunft bekommt **cboKategorieID** eine Abfrage über alle Kategorien:

```
SELECT tblKategorien.KategorieID, tblKategorien.Kategorie
FROM tblKategorien
ORDER BY tblKategorien.Kategorie;
```

Und **cboProduktID**? Zunächst schlicht alle Produkte – diese Liste schränken wir gleich per VBA ein:

```
SELECT tblProdukte.ProduktID, tblProdukte.Produkt
FROM tblProdukte
ORDER BY tblProdukte.Produkt;
```

Der Grundmechanismus

Der Kern steckt in einem einzigen Ereignis. Sobald der Benutzer eine Kategorie gewählt hat, bauen wir die Datensatzherkunft des zweiten Feldes neu auf – gefiltert auf eben diese Kategorie. Das übernimmt die Prozedur **cboKategorieID_AfterUpdate** aus dem folgenden Listing, das wir für die Ereignisprozedur **Nach Aktualisierung** des Kombinationsfeldes **cboKategorieID** hinterlegen:

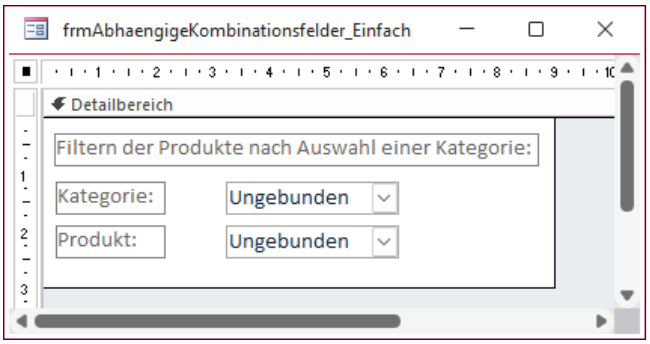

Bild 2: Die beiden Kombinationsfelder im Entwurf – oben die Kategorie, darunter das Produkt

```
Private Sub cboKategorieID_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT ProduktID, Produkt " _
        & "FROM tblProdukte WHERE KategorieID = " _
        & Nz(Me.cboKategorieID,0) & " ORDER BY ProduktID"
    Me.cboProduktID.RowSource = strSQL
End Sub
```

Wir setzen eine **SELECT**-Anweisung zusammen, die nur die Produkte der gewählten Kategorie liefert – das Kriterium bildet der gebundene Wert von **cboKategorieID**, also die **KategorieID**.

Diese Zeichenkette weisen wir der Eigenschaft **RowSource** von **cboProduktID** zu, und schon enthält das zweite Feld ausschließlich die passenden Produkte (siehe Bild 3).

Die Liste gleich aufklappen

Ein kleiner Handgriff macht die Bedienung flüssiger, wie wir im Formular **frmAbhaengigeKombinationsfelder_Dropdown** zeigen.

Statt den Benutzer nach der Kategorieauswahl selbst das Produktfeld öffnen zu lassen, geben wir ihm den Fokus und öffnen die Liste automatisch, wie folgendes Listing zeigt:

```
Private Sub cboKategorieID_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT ProduktID, Produkt " _
        & "FROM tblProdukte WHERE KategorieID = " _
        & Me.cboKategorieID & " ORDER BY ProduktID"
    Me.cboProduktID.RowSource = strSQL
    Me.cboProduktID.SetFocus
    Me.cboProduktID.Dropdown
End Sub
```

Neu sind die beiden letzten Zeilen: **SetFocus** setzt den Cursor ins Produktfeld, **Dropdown** klappt dessen Liste auf.

Der Benutzer sieht seine Auswahl sofort und muss nur noch zugreifen.

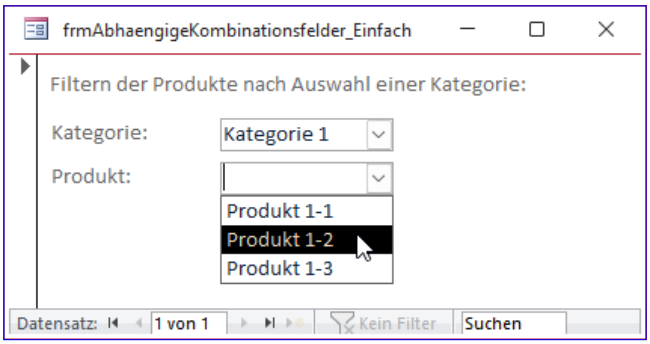

Bild 3: Nach der Wahl einer Kategorie führt das Produktfeld nur noch die dazugehörigen Einträge

Ein tückischer Nebeneffekt

Bis hierher funktioniert alles – solange der Benutzer der Reihe nach vorgeht. Doch was geschieht, wenn er nach getroffener Produktwahl noch einmal die Kategorie wechselt? Dann behält **cboProduktID** seinen alten Wert, obwohl dieser in der neuen Liste gar nicht mehr vorkommt.

Das Feld zeigt daraufhin – scheinbar grundlos – nichts mehr an, denn ein Kombinationsfeld kann einen Wert nur darstellen, wenn er auch in seiner Liste steht.

Den im Kombinationsfeld **cboProduktID** in der Eigenschaft Value gespeicherten Wert können wir am einfachsten per VBA sichtbar machen.

Dazu geben wir im Direktbereich des VBA-Editors den folgenden Befehl ein (in einer Zeile), und betätigen die Eingabetaste:

```
Debug.Print Forms!frmAbhaengigeKombinationsfelder_Dropdown!cboProduktID
1
```

Hier erhalten wir beispielsweise den Wert **1**. Da wir durch die erneute Auswahl im ersten Kombinationsfeld dem zweiten Kombinationsfeld eine neuen Datensatzquelle zugewiesen haben, welche den Eintrag mit dem Wert **1** im Feld **ProduktID** nicht mehr enthält, zeigt **cboProduktID** keinen Wert an, obwohl dieser enthalten ist.

Die Abhilfe ist eine einzige zusätzliche Zeile, die wir im Formular **frmAbhaengigeKombinationsfelder_Zuruecksetzen** eingebaut haben. Bevor wir die neue Liste zuweisen, setzen wir **cboProduktID** mit **Value = Null** zurück. Damit ist die veraltete Auswahl verschwunden, und das Feld startet sauber in die neue Kategorie.

```
Private Sub cboKategorieID_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT ProduktID, Produkt " _
        & "FROM tblProdukte WHERE KategorieID = " _
        & Me.cboKategorieID & " ORDER BY ProduktID"
    Me.cboProduktID.Value = Null
    Me.cboProduktID.RowSource = strSQL
    Me.cboProduktID.SetFocus
    Me.cboProduktID.Dropdown
End Sub
```

cboProduktID zeigt nun nach wie vor keinen Eintrag an, aber nun ist auch der in der Eigenschaft **Value** gespeicherte Wert gelöscht.

Wozu ist das wichtig? Es könnte sein, dass wir das Formular als Hilfsmittel zur Auswahl eines Produkts verwenden und dabei die Vorselektion nach der Kategorie anbieten. Wenn ein aufrufendes Formular nun den Wert aus **cboProduktID** ausliest und zurückgibt, enthält es eine Produkt-ID, obwohl der Benutzer kein Produkt sichtbar ausgewählt hat.

Ein Eintrag für alle Fälle

Manchmal möchte der Benutzer die Einschränkung bewusst aufheben und wieder sämtliche Produkte sehen. Dafür spendieren wir dem ersten Kombinationsfeld einen zusätzlichen Eintrag namens **Alle anzeigen**. Wie das gelingt, zeigen wir im Formular **frmAbhaengigeKombinationsfelder_AlleAnzeigen**.

Der Kniff steckt in einer **UNION**-Abfrage, die der echten Kategorienliste eine künstliche Zeile voranstellt:

```
SELECT 0 AS KategorieID, 'Alle anzeigen' AS Kategorie
FROM tblKategorien
UNION
SELECT KategorieID, Kategorie
FROM tblKategorien
ORDER BY Kategorie;
```

Der erste Teil der Abfrage erzeugt eine Zeile mit der **KategorieID 0** und dem Text **Alle anzeigen**; der zweite liefert die tatsächlichen Kategorien.

Da echte AutoWert-Schlüssel bei **1** beginnen, ist die **0** garantiert frei und dient uns als Kennung für „keine Einschränkung". Diese Abfrage tragen wir als Datensatzherkunft von **cboKategorieID** ein. Das Ergebnis dieser Abfrage in der Datenblattansicht sehen wir in Bild 4.

Der zugehörige Code muss die **0** nun noch auswerten – das erledigt folgende Ereignisprozedur:

```
Private Sub cboKategorieID_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT ProduktID, Produkt " _
        & "FROM tblProdukte WHERE KategorieID = " _
        & Me.cboKategorieID & " OR " & Me.cboKategorieID _
        & " = 0 ORDER BY ProduktID"
    Me.cboProduktID.Value = Null
    Me.cboProduktID.RowSource = strSQL
    Me.cboProduktID.SetFocus
    Me.cboProduktID.Dropdown
End Sub
```

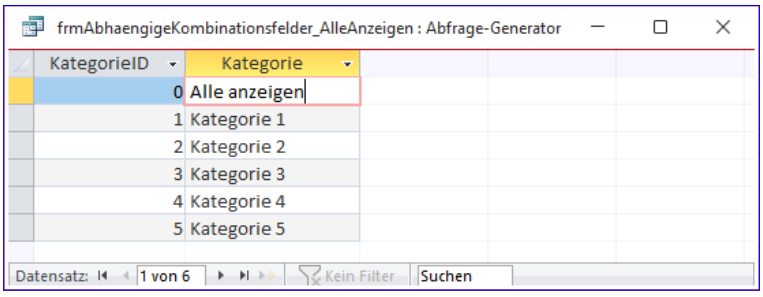

Bild 4: Ergebnis der **UNION**-Abfrage

```
Private Sub Form_Load()
    Me.cboKategorieID = 0
End Sub
```

Die **WHERE**-Klausel erhält eine zweite Bedingung: **OR ... = 0**. Wählt der Benutzer eine echte Kategorie, greift der erste Teil; wählt er **Alle anzeigen** – also die **0** –, ist die Bedingung **0 = 0** stets wahr, und die Abfrage liefert alle Produkte.

Damit beim Öffnen des Formulars sofort das vollständige Sortiment bereitsteht, setzen wir das Kategoriefeld in der Prozedur **Form_Load** von vornherein auf **0**, wie in Bild 5 zu sehen ist.

Übertragbar auf jede 1:n-Beziehung

Das Muster ist nicht an Kategorien und Produkte gebunden.

Überall, wo ein Wert die Auswahl eines zweiten einschränkt – Land und Region, Kunde und Projekt, Hersteller und Modell –, genügt dieselbe Handvoll Zeilen; wir tauschen lediglich die Tabellen- und Feldnamen in der **SELECT**-Anweisung aus.

Mehrere abhängige Kombinationsfelder

Wir können dieses Prinzip auch noch auf weitere Kombinationsfelder anwenden. Wenn wir beispielsweise nach der Auswahl eines Produkts noch Produkteigenschaften in einem weiteren Kombinationsfeld auswählen wollen, legen wir eine ähnliche Prozedur wie für **cboKategorieID** auch noch für **cboProduktID** an.

Ein Beispiel dafür zeigen wir in der folgenden Ausgabe im Beitrag **Navigieren in hierarchischen Daten** (**www.access-im-unternehmen.de/1613**).

Zusammenfassung und Ausblick

In diesem Beitrag haben wir die grundlegenden Techniken für die Auswahl von Daten aus abhängigen Kombinationsfeldern vorgestellt.

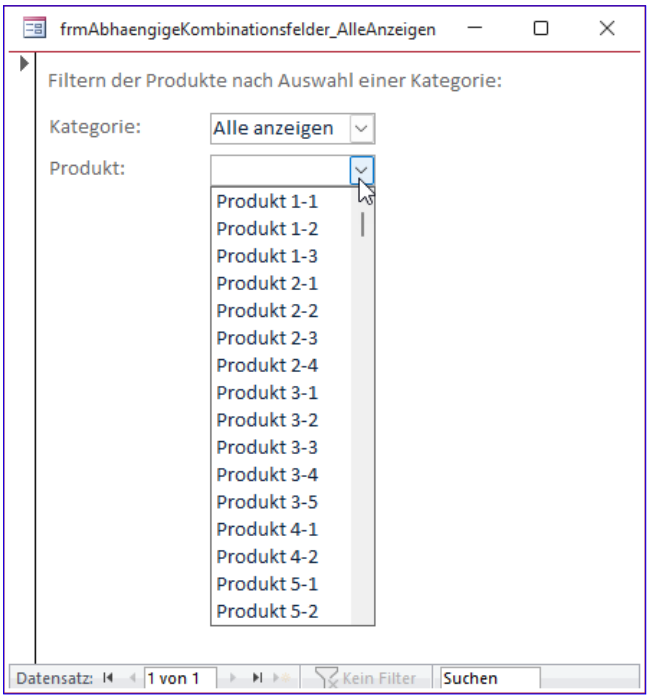

Bild 5: Das erste Kombinationsfeld mit dem zusätzlichen Eintrag **Alle anzeigen**

Hierbei ist zu beachten, dass die Beispielformulare selbst nicht an eine Datensatzquelle gebunden sind. Sie dienen allein der Auswahl einer Kategorie und nachfolgend eines Produkts dieser Kategorie.

Dieses Formular können wir, wie oben bereits angerissen, beispielsweise als Hilfsformular öffnen, wenn wir in einem aufrufenden Formular einen Artikel auswählen wollen – beispielsweise in den Bestelldetails eines Bestellformulars.

Hier müssten wir nun noch eine Schaltfläche hinzufügen, mit welcher der Benutzer die Auswahl des Produkts abschließen oder verwerfen kann.

In einem weiteren Beitrag namens **Gebundene abhängige Kombinationfelder** (**www.access-im-unternehmen.de/1609**) schauen wir uns noch an, wie wir mit Tabellen umgehen, die selbst zwei Felder enthalten, von denen der Wert des zweiten Feldes von der Auswahl im ersten Feld abhängig sein soll.

Gebundene abhängige Kombinationsfelder

Im Beitrag »Abhängige Kombinationsfelder« (www.access-im-unternehmen.de/1608) haben wir beschrieben, wie wir per 1:n-Beziehung verknüpfte Daten in zwei aufeinanderfolgenden Kombinationsfeldern erst filtern und dann auswählen können. Im vorliegenden Beitrag gehen wir einen Schritt weiter und schauen uns an, wie wir abhängige Kombinationsfelder in gebundenen Formularen darstellen. In Formularen in der Einzelansicht gelingt dies ohne Probleme. In der Endlosansicht und der Datenblattansicht stoßen wir jedoch auf Einschränkungen, die wir mehr oder weniger gut umgehen können. Wie das gelingt, lesen Sie ebenfalls in diesem Beitrag.

Datenmodell der Beispieldatenbank

Die Beispieldatenbank erweitert das Modell aus dem ersten Beitrag um eine Ebene.

In **tblKategorien** stehen die Kategorien, in **tblUnterkategorien** die Unterkategorien – jede Unterkategorie gehört über den Fremdschlüssel **KategorieID** zu genau einer Kategorie.

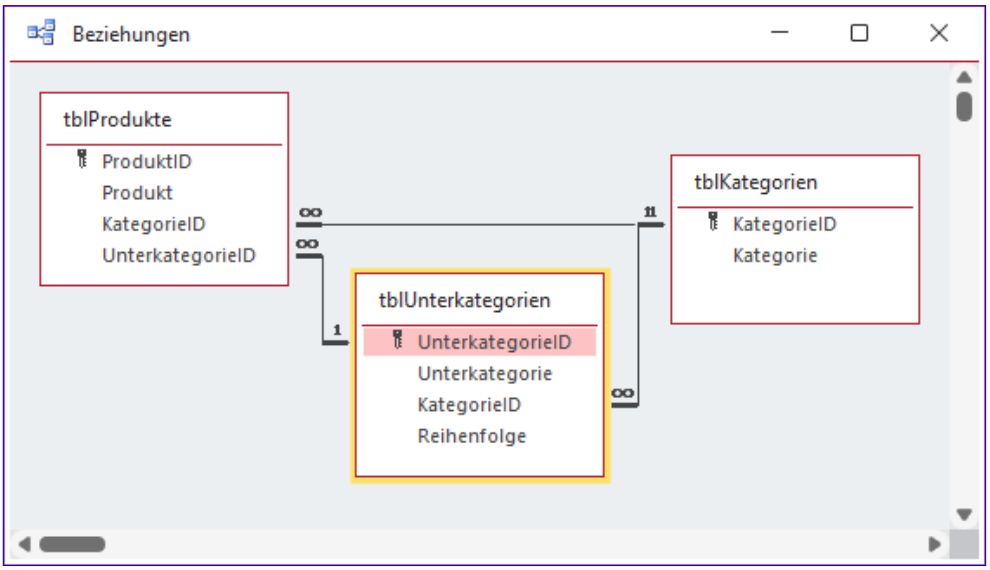

Bild 1: Das Datenmodell mit **tblKategorien**, **tblUnterkategorien** und **tblProdukte**

Die Produkte in **tblProdukte** verweisen mit den beiden Fremdschlüsseln **KategorieID** und **UnterkategorieID** zugleich auf ihre Kategorie und ihre Unterkategorie (siehe Bild 1).

So entstehen zwei gestufte 1:n-Beziehungen, an denen sich das Zusammenspiel gebundener abhängiger Kombinationsfelder gut zeigen lässt.

Außerdem haben wir noch eine 1:n-Beziehung zwischen den Tabellen **tblKategorien** und **tblUnterkategorien** hinzugefügt, damit festgelegt ist, welche Unterkategorien zu welcher Kategorie gehören.

Abhängige Kombinationsfelder in gebundenen Formularen

In der Einzelansicht ist die Sache unkompliziert. Als Beispiel dient das Formular **frmProdukteDetail**, das an die Tabelle **tblProdukte** gebunden ist und ihre Felder in vier Steuerelementen zeigt (siehe Bild 2): die Textfelder **ProduktID** und **Produkt** sowie die beiden Kombinationsfelder **cboKategorieID** und **cboUnterkategorieID**, die an die Fremdschlüssel **KategorieID** und **UnterkategorieID** gebunden sind. Zum Erstellen dieses Formulars haben wir als Datensatzquelle die Tabelle **tblProdukte** eingestellt und alle Einträge der Feldliste in den Detailbereich gezogen.

Als Datensatzherkunft erhält **cboKategorieID** eine Abfrage über alle Kategorien:

```
SELECT [tblKategorien].[KategorieID],
[tblKategorien].[Kategorie]
FROM tblKategorien
ORDER BY [Kategorie];
```

Das abhängige Feld **cboUnterkategorieID** bekommt zunächst eine Abfrage über sämtliche Unterkategorien als Datensatzherkunft – diese Liste schränken wir gleich per VBA auf die gewählte Kategorie ein.

```
SELECT [tblUnterkategorien].[UnterkategorieID],
[tblUnterkategorien].[Unterkategorie]
FROM tblUnterkategorien
ORDER BY [Unterkategorie];
```

Den eigentlichen Mechanismus übernimmt die Ereignisprozedur **cboKategorieID_AfterUpdate**, die wir für das Ereignis **Nach Aktualisierung** des Kombinationsfeldes **cboKategorieID** hinterlegen (siehe Listing 1).

Wir stellen eine **SELECT**-Anweisung zusammen, die nur die Unterkategorien der gewählten Kategorie liefert; das Kriterium bildet der gebundene Wert von **cboKategorieID**, also die **KategorieID**. Innerhalb eines **With**-Blocks setzen wir **cboUnterkategorieID** zunächst mit **Value = Null** zurück – so verschwindet eine womöglich noch gespeicherte, nicht mehr passende Unterkategorie –, weisen anschließend die neue Datensatzherkunft zu und klappen die Liste über **SetFocus** und **Dropdown** gleich auf.

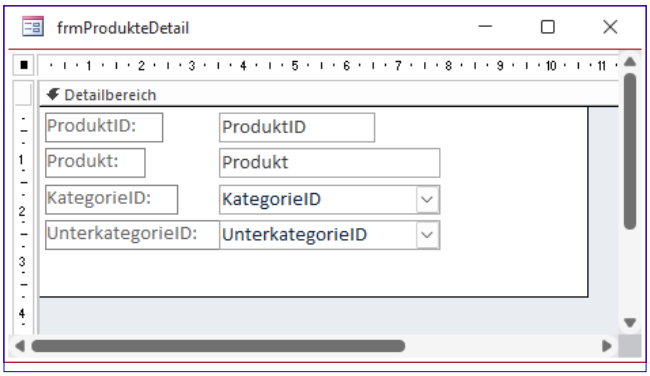

Bild 2: Der Entwurf des Formulars **frmProdukteDetail**

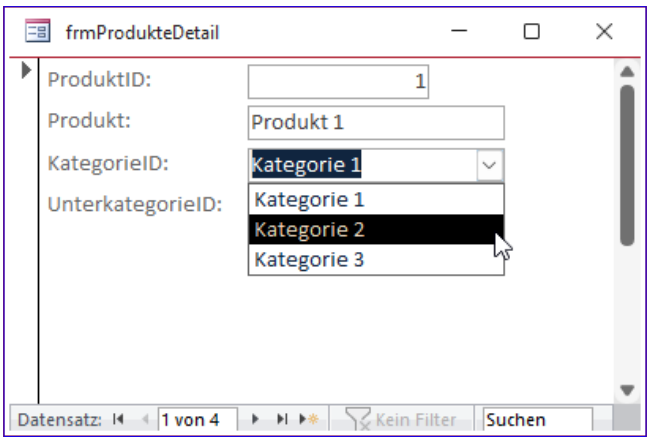

Bild 3: Auswahl einer Kategorie im Feld **cboKategorieID**

```
Private Sub cboKategorieID_AfterUpdate()
    Dim strSQL As String

    strSQL = "SELECT UnterkategorieID, Unterkategorie FROM tblUnterkategorien WHERE KategorieID = " _
        & Me.cboKategorieID & " ORDER BY Unterkategorie"
    With Me.cboUnterkategorieID
        .Value = Null
        .RowSource = strSQL
        .SetFocus
        .Dropdown
    End With
End Sub
```

Listing 1: Die Unterkategorien an die gewählte Kategorie anpassen

Das Ergebnis zeigen die beiden Abbildungen: Bild 3 hält die Auswahl einer Kategorie im Feld **cboKategorieID** fest.

Unmittelbar nach der Auswahl eines dieser Einträge bietet **cboUnterkategorieID** nur noch die dazugehörigen Einträge an (siehe Bild 4).

In der Einzelansicht war das – wie versprochen – ohne Hürden. Interessant, und stellenweise widerspenstig, wird es in der Endlos- und der Datenblattansicht; diesen Ansichten wenden wir uns im nächsten Abschnitt zu.

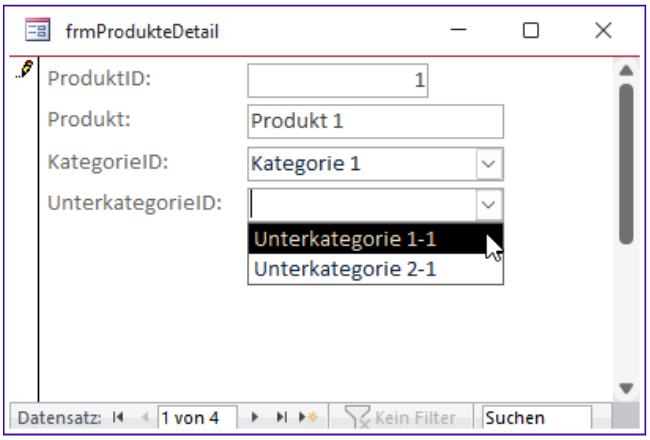

Bild 4: Das gefilterte Kombinationsfeld **cboUnterkategorieID** nach der Kategorieauswahl

Abhängige Kombinationsfelder in gebundenen Endlosformularen

Um die abhängigen Kombinationsfelder in einem Endlosformular zu nutzen, bauen wir das Formular zunächst wie zuvor auf und speichern dieses unter dem Namen **frmProdukteEndlos**. Hier nutzen wir nun die Anordnen-Funktion von Access.

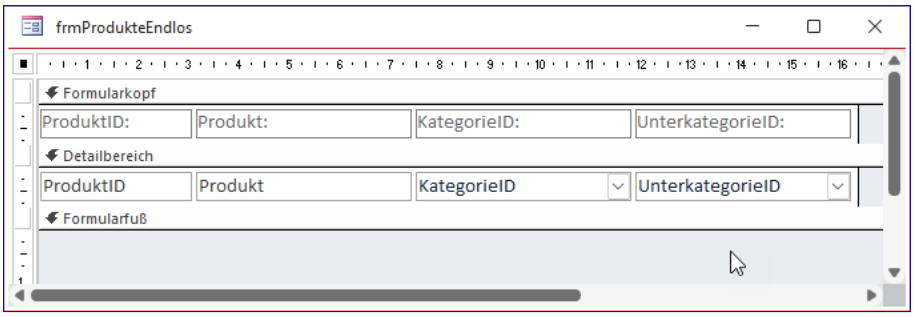

Bild 5: Entwurf des Formulars in der Endlosansicht

Wir markieren zunächst alle Steuerelemente des Formulars. Dann wählen wir im Ribbon unter Anordnen den Befehl **Tabellel Tabelle** aus. Dies verschiebt die Bezeichnungsfelder in den Formularkopf und belässt die gebundenen Steuerelemente im Detailbereich. Außerdem werden die Steuerelemente tabellarisch wie in Bild 5 angeordnet.

Anschließend markieren wir die Anordnung und entfernen diese durch einen Klick auf **Tabellel Layout entfernen**.

Außerdem stellen wir für die Eigenschaft **Standardansicht** den Wert **Endlosformular** ein.

Für das Ereignis **Nach Aktualisierung** des Steuerelements **cboKategorieID** hinterlegen wir wieder die folgende Ereignisprozedur:

```
Private Sub cboKategorieID_AfterUpdate()
    Dim strSQL As String
    strSQL = "SELECT UnterkategorieID, Unterkategorie " _
        & "FROM tblUnterkategorien WHERE KategorieID = " _
        & Me.cboKategorieID & " ORDER BY Unterkategorie"
    With Me.cboUnterkategorieID
        .Value = Null
        .RowSource = strSQL
        .SetFocus
        .Dropdown
    End With
End Sub
```

Verhalten der abhängigen Kombinationsfelder im Endlosformular

Danach wechseln wir in die Endlosformular-Ansicht und legen einen neuen Datensatz an, für den wir den Produktnamen festlegen. Wir können nun im Feld **cboUnterkate-**

gorieID noch alle verfügbaren Kategorien auswählen (siehe Bild 6). Für bereits vorhandene Datensätze werden die Unterkategorien noch korrekt angezeigt.

Hier treffen wir keine Auswahl, sondern selektieren zunächst eine Kategorie. Anschließend klappen wir erneut das Kombinationsfeld **cboUnterkategorieID** auf.

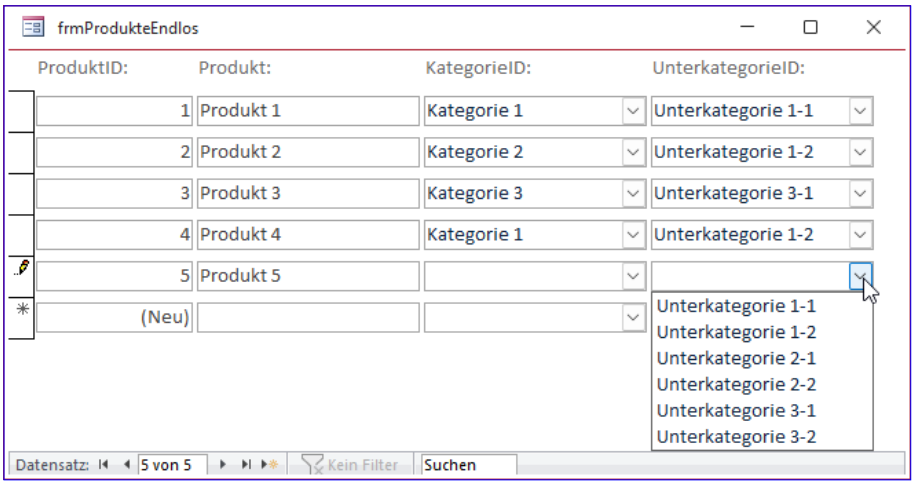

Bild 6: Auswahl der Unterkategorie ohne vorherige Auswahl einer Kategorie

Wie Bild 7 zeigt, hat die Prozedur **cboUnterkategorieID_AfterUpdate** korrekt funktioniert: Das Kombinationsfeld **cboUnterkategorieID** zeigt nur noch die zu der gewählten Kategorie gehörenden Unterkategorien an.

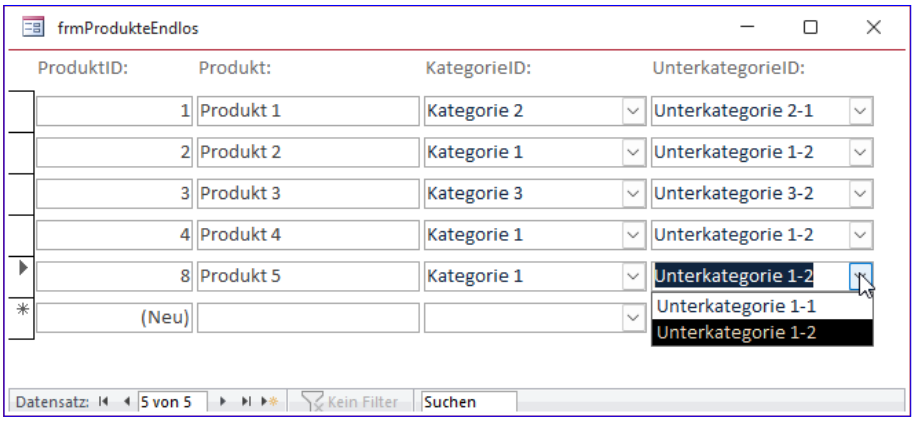

Bild 7: Auswahl der Unterkategorie nach vorheriger Auswahl einer Kategorie

Wenn wir eine Unterkategorie ausgewählt haben und den Datensatz verlassen, geschieht jedoch etwas, mit dem wir zunächst nicht gerechnet haben: Das Feld **UnterkategorieID** wird für alle Datensätze, deren Kategorie nicht der zuletzt gewählten Kategorie entspricht, geleert (siehe Bild 8).

Das Verhalten ändert sich, wenn wir die Kategorie und die Unterkategorie für einen vorhanden Datensatz neu auswählen.

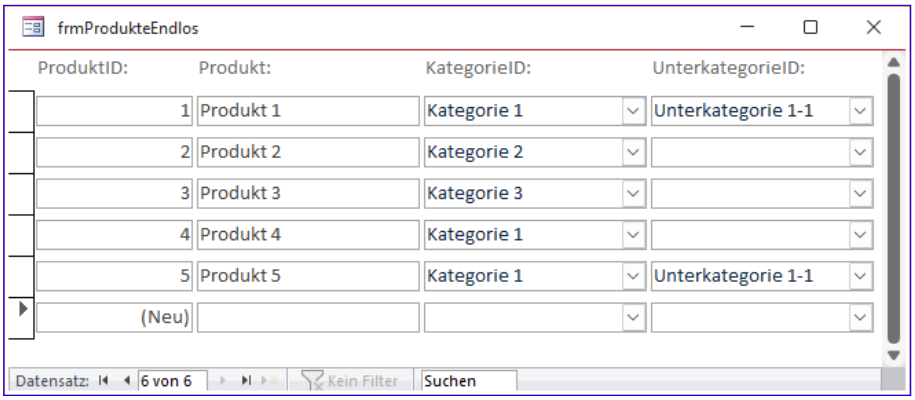

Bild 8: Verschiedene Unterkategorien werden scheinbar willkürlich geleert.

Hier bleiben zunächst alle Einträge im Steuerelement **cboUnterkategorieID** erhalten. Erst wenn wir einen Datensatz markieren, dessen Kategorie nicht der zuletzt gewählten Kategorie entspricht, wird dessen Feld **cboUnterkategorieID** geleert.

Der Grund ist in beiden Fällen schnell erläutert: Wir filtern die Datensatzherkunft von cboUnterkategorieID, sodass die anzuzeigenden Werte für die Datensätze mit einer anderen übergeordneten Kategorie nicht mehr verfüg-

bar sind. Dadurch können diese schlicht nicht mehr angezeigt werden.

Tatsächlich sind die gespeicherten Werte für alle Datensätze der Tabelle noch vorhanden, wovon wir uns durch das Schließen und erneute Öffnen des Formulars **frmProdukteEndlos** überzeugen können (siehe Bild 9).

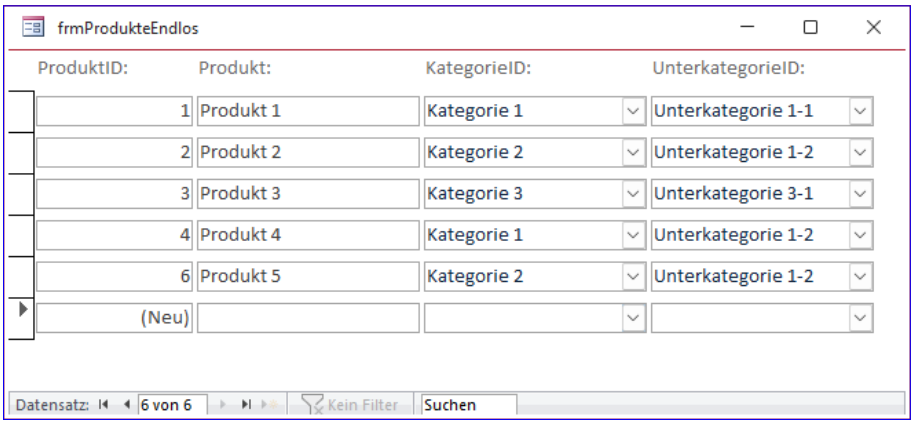

Bild 9: Nach dem erneuten Öffnen des Formulars sind alle Unterkategorien wieder sichtbar.

Dennoch ist dieses Verhalten äußerst ungünstig, denn wir wollen den Benutzer nicht unnötig verwirren.

Einfache Lösung zum Anzeigen aller Unterkategorien

Unsere Idee zu diesem Problem ist, die Datensatzherkunft des Kombinationsfeldes **cboUnterkategorieID** nach dem Verlassen des zuletzt bearbeiteten Datensatzes wieder auf die Abfrage einzustellen, die alle Unterkategorien liefert.

Dazu können wir am einfachsten das Ereignis **Nach Aktualisierung** von **cboUnterkategorieID** verwenden. Für dieses hinterlegen wir die Ereignisprozedur aus Listing 2.

Wenn wir einen Datensatz speichern, dessen Kategorie wir geändert und damit die Datensatzherkunft gefiltert haben, erhält das Kombinationsfeld **cboUnterkategorieID** wieder alle Unterkategorien zur Auswahl. Somit zeigt dieses Kombinationsfeld auch nach dem Speichern des Datensatzes wieder alle Unterkategorien für die übrigen Datensätze an.

Nachteil dieser Lösung

Der einzige offensichtliche Nachteil dieser Vorgehensweise ist, dass nun beim Auswählen der Unterkategorie für jeden Datensatz alle Einträge angezeigt werden – unabhängig davon, welche Kategorie für diesen Datensatz ausgewählt wurde (siehe Bild 10).

Auswahl der Unterkategorie einschränken

Wir wünschen uns, dass wir das Kombinationsfeld **cboUnterkategorieID** öffnen können und dass nur die zur Kategorie passenden Unterkategorien angezeigt werden – ohne, dass die Unterkategorien der übrigen Datensätze ausgeblendet werden.

Das realisieren wir über einige Umwege, die wir nachfolgend genau erläutern.

```
Private Sub cboUnterkategorieID_AfterUpdate()
    Dim strSQL As String

    strSQL = "SELECT UnterkategorieID, Unterkategorie FROM tblUnterkategorien ORDER BY Unterkategorie"
    With Me.cboUnterkategorieID
        .RowSource = strSQL
    End With
End Sub
```

Listing 2: Zurücksetzen der Datensatzherkunft von **cboUnterkategorieID**

Die Grundidee

Der Ausweg ist verblüffend einfach, wenn man ihn einmal gesehen hat: Wir drehen die Sache um.

Statt die Liste dauerhaft einzuschränken, lassen wir sie die ganze Zeit vollständig – mit allen Unterkategorien. Dann findet jede Zeile ihren gespeicherten Wert in der Liste und zeigt ihn an.

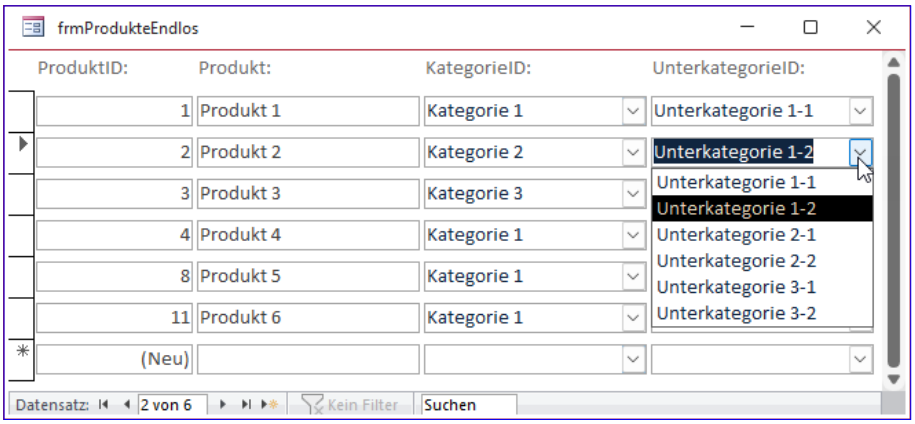

Bild 10: Nun können alle Unterkategorien für einen Datensatz ausgewählt werden.

Nur für den kurzen Moment, in dem der Benutzer in einer Zeile tatsächlich eine neue Unterkategorie auswählt, blenden wir die passende, kurze Liste ein. Sobald er fertig ist, schalten wir wieder auf die vollständige Liste zurück.

Man kann sich das wie eine Speisekarte vorstellen: Würde man alle Gerichte bis auf eine Kategorie durchstreichen, stünden alle Tische, die schon etwas anderes bestellt haben, plötzlich ohne Angabe da. Also lassen wir die volle Karte hängen und reichen nur dem Tisch, der gerade neu bestellt, für einen Augenblick die kurze Auswahl.

Wenn sich die Kategorie ändert

Beginnen wir mit dem ersten Feld. Wählt der Benutzer in der aktuellen Zeile eine Kategorie, bauen wir die Liste des Unterkategorie-Feldes passend dazu neu auf, wie Listing 3 zeigt.

Wir setzen **cboUnterkategorieID** zunächst mit **Value = Null** zurück, weisen dann die auf die Kategorie gefilterte Liste zu und klappen sie über **SetFocus** und **Dropdown** gleich auf. So weit ist das der bekannte Ablauf aus dem Einzelformular.

Der Normalzustand: die vollständige Liste

Damit alle Zeilen ihren Wert anzeigen, braucht das Unterkategorie-Feld im Ruhezustand die vollständige Liste. Diese kleine Hilfsprozedur stellt sie her (Listing 4).

Sie schreibt schlicht eine Datensatzherkunft ohne Filter in **cboUnterkategorieID** – also alle Unterkategorien.

```
Private Sub cboKategorieID_AfterUpdate()
    Dim strSQL As String

    strSQL = "SELECT UnterkategorieID, Unterkategorie FROM tblUnterkategorien WHERE KategorieID = " _
        & Me.cboKategorieID & " ORDER BY Unterkategorie"
    With Me.cboUnterkategorieID
        .Value = Null
        .RowSource = strSQL
        .SetFocus
        .Dropdown
    End With
End Sub
```

Listing 3: Nach der Kategorieauswahl die passenden Unterkategorien zeigen

```
Private Sub AlleUnterkategorienAnzeigen()
    Dim strSQL As String

    strSQL = "SELECT UnterkategorieID, Unterkategorie FROM tblUnterkategorien ORDER BY Unterkategorie"
    With Me.cboUnterkategorieID
        .RowSource = strSQL
    End With
End Sub
```

Listing 4: Die Liste auf alle Unterkategorien setzen

Aufgerufen wird sie immer dann, wenn wir zu einem Datensatz wechseln. Dafür sorgt das Ereignis **Beim Anzeigen**, das im Endlosformular für die jeweils aktuelle Zeile ausgelöst wird:

```
Private Sub Form_Current()
    Call AlleUnterkategorienAnzeigen
End Sub
```

Beim Bearbeiten kurz die kurze Liste

Jetzt der eigentliche Kniff. Betritt der Benutzer das Unterkategorie-Feld, um eine Auswahl zu treffen, möchten wir ihm nur die passenden Einträge zeigen und die Liste sofort aufklappen.

Beides lässt sich aber nicht im selben Augenblick erledigen – Access braucht einen winzigen Moment, bis der Wechsel in das Feld abgeschlossen ist. Deshalb arbeiten wir mit einem kurzen Timer, wie folgendes Listing zeigt:

```
Private Sub cboUnterkategorieID_GotFocus()
    Call AlleUnterkategorienAnzeigen
    Me.TimerInterval = 50
End Sub
```

Beim Fokuserhalt stellen wir zunächst wieder die volle Liste her – so bleibt der gespeicherte Wert in allen Datensätzen sichtbar – und setzen dann **TimerInterval** auf **50**. Damit bitten wir Access, in 50 Millisekunden das Timer-Ereignis auszulösen. Dieses erledigt anschließend die eigentliche Arbeit (siehe Listing 5).

Zuerst schalten wir den Timer mit **TimerInterval = 0** wieder aus, damit er nur ein einziges Mal feuert. Dann bauen wir die auf die aktuelle Kategorie gefilterte Liste, setzen den Fokus und klappen sie auf. Für den Benutzer sieht es aus, als öffne sich sofort die richtige, kurze Liste. Er sieht maximal ganz kurz das Aufklappen der vollständigen Liste.

```
Private Sub Form_Timer()
    Dim strSQL As String

    Me.TimerInterval = 0
    strSQL = "SELECT UnterkategorieID, Unterkategorie FROM tblUnterkategorien WHERE KategorieID = " _
        & Me.cboKategorieID & " ORDER BY Unterkategorie"
    With Me.cboUnterkategorieID
        .RowSource = strSQL
        .SetFocus
        .Dropdown
    End With
End Sub
```

Listing 5: Kurz darauf die gefilterte Liste zeigen und aufklappen

```
Private Sub cboUnterkategorieID_MouseDown(Button As Integer, Shift As Integer, X As Single, Y As Single)
    If fIsComboOpen = True Then
        Exit Sub
    End If

    If Not Y > Me.cboUnterkategorieID.Top + Me.cboUnterkategorieID.Height Then
        Call cboUnterkategorieID_GotFocus
    End If
End Sub
```

Listing 6: Beim Anklicken dieselbe Auswahl-Logik starten

Auch der Mausklick soll es auslösen

Nicht jeder betritt ein Feld mit der Tastatur – viele klicken es einfach an. Diesen Fall fangen wir zusätzlich ab, indem wir das Ereignis **MouseDown** von **cboUnterkategorieID** implementieren (siehe Listing 6).

Zuerst fragen wir mit der Hilfsfunktion **fIsComboOpen** nach, ob bereits eine Liste offen ist – wenn ja, tun wir nichts weiter. Diese Funktion stammt aus einem eigenen Modul namens **mdlAPI** und meldet über die Windows-API, ob ein Kombinationsfeld gerade aufgeklappt ist.

Andernfalls prüfen wir, ob der Klick oben auf das Feld selbst ging und nicht in die bereits geöffnete Liste darunter, und rufen dann dieselbe Prozedur wie beim Fokuserhalt auf.

Warum diese Hilfsfunktion? Wenn wir zum Beispiel über die Auswahl einer Kategorie bereits das Kombinationsfeld **cboUnterkategorieID** geöffnet haben oder auch durch einen Mausklick auf das Steuerelement selbst, und klicken dann erneut darauf, soll sich das Kombinationsfeld wie üblich einfach wieder schließen.

Da Access keine eingebaute Möglichkeit bietet, den »Ausgeklappt«-Zustand eines Kombinationsfeldes zu prüfen, müssen wir hier die Windows-API hinzuziehen.

Stellt diese fest, dass das Kombinationsfeld bereits geöffnet ist, wird die Prozedur beendet und der Klick bewirkt das Schließen des Kombinationsfeldes.

Andernfalls prüfen wir, ob der Benutzer in das Kombinationsfeld geklickt hat oder ob er einen der Einträge ausgewählt hat – das Mouse Down-Ereignis wird nämlich in beiden Fällen ausgelöst.

Hier untersuchen wir die vertikale Position des Mauszeigers zum Zeitpunkt des Mausklicks. Diese erhalten wir über den Y-Parameter der entsprechenden Ereignisprozedur und prüfen, ob dieser Wert kleiner ist als die Summe der Eigenschaften **Top** und **Height** des Kombinationsfeldes.

Nur in diesem Fall wollen wir mit dem Aufruf von **cboUnterkategorie_GotFocus** dafür sorgen, dass das Kombinationsfeld mit der korrekten Datensatzherkunft geöffnet wird.

Nach der Auswahl zurück zur vollen Liste

Hat der Benutzer seine Unterkategorie gewählt, ist der Ausnahmezustand vorbei.

Wir schalten sofort wieder auf die vollständige Liste zurück, damit alle Zeilen weiterhin ihre Werte anzeigen können:

```
Private Sub cboUnterkategorieID_AfterUpdate()
    Call AlleUnterkategorienAnzeigen
End Sub
```

Die Prozedur ruft nur unsere Hilfsprozedur **AlleUnterkategorienAnzeigen** auf – mehr ist nicht nötig.

```
Private Sub cboUnterkategorieID_BeforeUpdate(Cancel As Integer)
    Dim lngKategorieID As Long
    Dim lngUnterkategorieID As Long

    lngKategorieID = Nz(Me.cboKategorieID, 0)
    lngUnterkategorieID = Nz(Me.cboUnterkategorieID, 0)
    If Nz(DLookup("UnterkategorieID", "tblUnterkategorien", "UnterkategorieID = " & lngUnterkategorieID _
            & " AND KategorieID = " & lngKategorieID), 0) = 0 Then
        MsgBox "Diese Unterkategorie kann nicht für diese Kategorie ausgewählt werden.", vbOKOnly + vbExclamation, _
            "Ungültige Unterkategorie"
        Cancel = True
        Me.cboUnterkategorieID.Dropdown
    End If
End Sub
```

Listing 7: Prüfen, ob Unterkategorie und Kategorie zusammenpassen

Falsche Kombinationen verhindern

Sollte doch einmal die vollständige Liste angezeigt werden und nicht die für die aktuelle Kategorie gefilterte, beugen wir Fehleingaben durch eine entsprechende Prüfung vor.

Dabei prüfen wir im Ereignis **Vor Aktualisierung**, ob der gewählte Wert in **cboUnterkategorieID** der Kategorie aus **cboKategorieID** entspricht. Das erledigt die Ereignisprozedur aus Listing 7.

Mit **DLookup** schlagen wir nach, ob es in **tblUnterkategorien** einen Eintrag gibt, dessen **UnterkategorieID** und **KategorieID** beide zur aktuellen Auswahl passen.

Kommt dabei nichts heraus – der **Nz**-Ausdruck liefert dann **0** –, passt die Kombination nicht: Wir zeigen eine kurze Meldung, setzen **Cancel** auf **True**, wodurch die Eingabe verworfen wird, und klappen die Liste erneut auf, damit der Benutzer es gleich richtig wählen kann.

Zusammenfassung und Ausblick

Damit arbeiten abhängige Kombinationsfelder auch im Endlosformular sauber: Alle Zeilen zeigen ihre Werte, und beim Bearbeiten erscheint trotzdem die passende, kurze Auswahl.

Der Kern ist der Wechsel zwischen zwei Zuständen – vollständige Liste zum Anzeigen, gefilterte Liste zum Auswählen –, gesteuert über die Ereignisse des Formulars und des Kombinationsfeldes.

Leider funktioniert dies nicht vollständig in der Datenblattansicht, da hier die im **Timer**-Ereignis angegebene Filterung nicht greift.

Für Endlosformulare haben wir damit jedoch eine gute Lösung, deren einziger Nachteil das kurze Aufflackern der gesamten Liste vor der eigentlichen Filterung ist.

Wichtig: Die vorgestellten Ereignisprozeduren müssen wie gezeigt übernommen werden – vor allem aber muss der Teil, der in die Timer-Prozedur ausgelagert wurde, dort verbleiben, andernfalls wird nach dem Anzeigen aller Einträge in **cboUnterkategorieID** das erneute Filtern auf die der Kategorie entsprechenden Einträge nicht durchgeführt.

Filterformular erweitern: Kombinationsfelder

In den ersten beiden Teilen dieser Beitragsreihe haben wir ein universell einsetzbares Filterformular vorgestellt und seine Programmierung beleuchtet. Bisher bietet das Formular für Vergleichswerte schlicht ein Textfeld an. Bei einem Feld wie »AnredeID« oder »KategorieID«, das im aufrufenden Formular über ein Kombinationsfeld dargestellt wird, muss der Benutzer aber die nackte Zahl kennen, nach der er filtern will – wenig komfortabel. In diesem Teil erweitern wir das Filterformular so, dass es solche Lookup-Felder erkennt und im Vergleichsfeld dieselbe Auswahlliste anbietet wie das Originalformular.

Die Ausgangslage

Viele Tabellen arbeiten mit Fremdschlüsseln. In der Kundentabelle aus den ersten Teilen könnte etwa ein Feld **AnredeID** stehen, das auf eine Tabelle **tblAnreden** verweist. Im Datenformular zeigt ein gebundenes Kombinationsfeld dann nicht die ID, sondern den sprechenden Namen an – also »Herr« statt »1«. Das Filterformular kennt diese Zuordnung jedoch nicht: Es liest nur das Recordset aus und stellt für ein Zahlenfeld ein Texteingabefeld bereit.

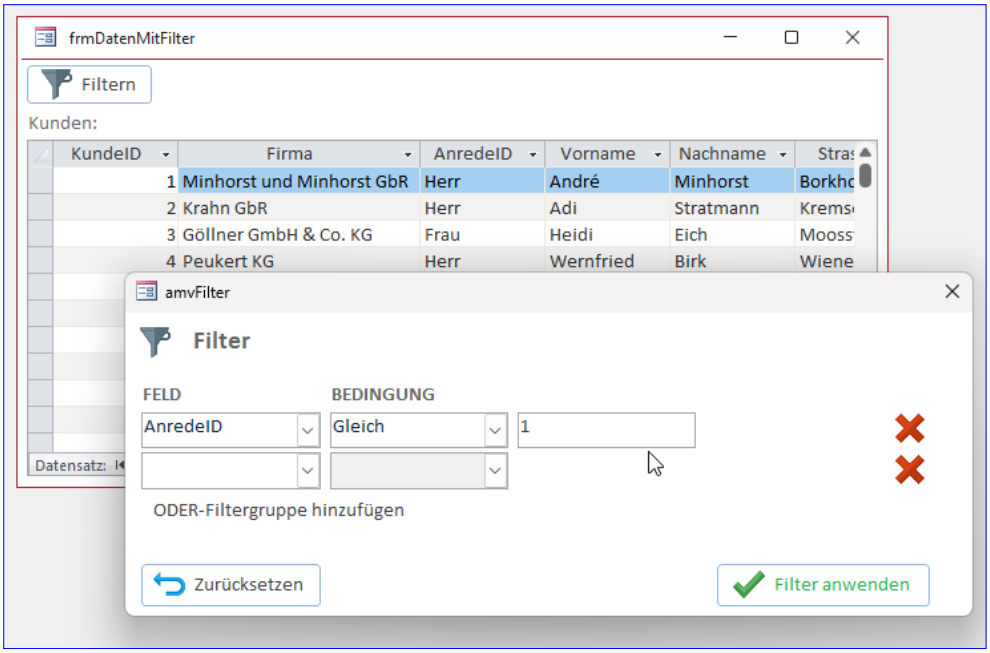

Bild 1: Bisher muss der Benutzer die interne ID des Kategorie-Feldes kennen

Wie sich das auswirkt, ist in Bild 1 zu sehen: Der Benutzer möchte nach der Kategorie filtern, bekommt aber nur ein leeres Eingabefeld und müsste die interne ID erraten. Genau hier setzen wir an. Wir wollen die Filterkriterien für Kombinationsfelder so gestalten, dass wir erstens die Bedingungen anpassen und zweitens die vorhandenen Werte

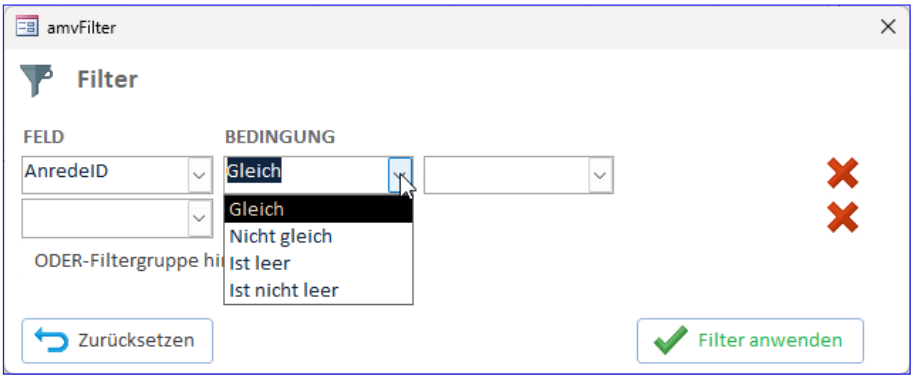

Bild 2: Der Zwischenschritt: Wir bekommen für Nachschlagefelder neue Bedingungen.

als Vergleichswerte auswählen können.

Bild 2 zeigt die neuen Bedingungen für Nachschlagefelder an:

- **Gleich**: Liefert alle Datensätze, für die der gleiche Wert ausgewählt ist.
- **Nicht gleich**: Liefert alle Datensätze, die nicht den ausgewählten Wert enthalten.
- **Ist leer**: Liefert alle Datensätze, für die das Nachschlagefeld leer ist.
- **Ist nicht leer**: Liefert alle Datensätze, für die das Nachschlagefeld einen Wert enthält.

Bild 3 zeigt, wie wir die vorhandenen Werte als Vergleichswerte auswählen können.

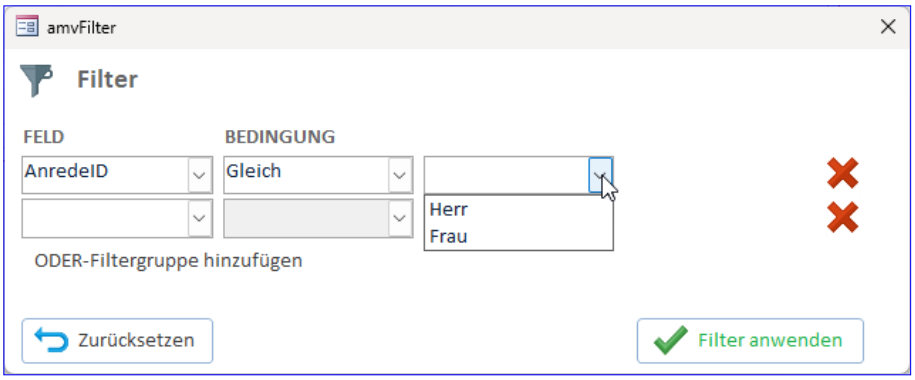

Bild 3: Das Ziel – im Vergleichsfeld erscheint dieselbe Auswahlliste wie im Datenformular.

Das Konzept

Die Lösung besteht aus drei Bausteinen. Erstens muss das Filterformular beim Einlesen der Felder erkennen, ob ein Feld im aufrufenden Formular über ein Kombinationsfeld dargestellt wird, und sich dessen Datensatzherkunft merken.

Zweitens braucht jede Filterzeile ein zusätzliches Kombinationsfeld als alternatives Wertfeld. Drittens muss beim Aufbau des SQL-Ausdrucks der gebundene Wert – also die ID – verwendet werden, nicht der angezeigte Text.

Der angenehme Nebeneffekt: An der eigentlichen SQL-Logik in **GetFilterausdruck** ändert sich nichts.

Da das neue Kombinationsfeld seinen gebundenen Wert liefert und der zugrunde liegende Feldtyp (Zahl oder Text) erhalten bleibt, quotiert die Funktion den Wert wie gewohnt – Zahlen ohne, Texte mit Anführungszeichen.

Vorbereitung: Übergabe des zu filternden Recordsets durch Formular ersetzen

Bisher haben wir beim Aufrufen des Filterformulars einen Verweis auf das aufrufende Formular sowie auf das zu filternde Recordset übergeben. Um für ein zu filterndes Nachschlagefeld jedoch die notwendigen Informationen zu holen, benötigen wir einen Verweis auf das jeweilige Nachschlagefeld. Deshalb haben wir den Aufruf des Filterformulars so angepasst, dass wir keinen Verweis auf das zu filternde Recordset übergeben, sondern einen auf das **Form**-Objekt, welches das Recordset anzeigt.

Der Aufruf ändert sich also wie folgt:

```
Private Sub cmdFilter_Click()
    DoCmd.OpenForm "frmFilter"
    Set Forms!frmFilter.CallingForm = Me
    Set Forms!frmFilter.RecordsetForm = _
        Me.sfmDatenMitFilter.Form
End Sub
```

In **Form_frmFilter** haben wir dazu eine neue Variable hinterlegt:

```
Private m_frmDaten As Form
```

Sie kann über diese öffentliche **Property Set**-Prozedur gefüllt werden:

```
Public Property Set CallingForm(frm As Form)
```

```
    Set m_frmAufrufend = frm
End Property
```

Schritt 1: Lookup-Felder erkennen

Beim Öffnen liest das Filterformular in **LadeFelder** alle Felder des Recordsets aus. An dieser Stelle ergänzen wir die Erkennung. Für jedes Feld prüfen wir, ob im aufrufenden Formular ein an dieses Feld gebundenes Kombinationsfeld existiert. Die dazu nötigen Metadaten halten wir in einem benutzerdefinierten Typ fest, den wir wie folgt definieren:

```
Private Type TKombiInfo
    IstKombi As Boolean
    RowSource As String
    RowSourceType As String
    ColumnCount As Integer
    ColumnWidths As String
    BoundColumn As Integer
End Type
```

Das Array **m_Kombi** speichert für jedes Recordset-Feld die komplette Konfiguration eines eventuell vorhandenen Kombinationsfelds:

```
Private m_Kombi() As TKombiInfo
```

Die Matrix **m_bolAktKombi** merkt sich pro Filterzeile, ob das dort gerade gewählte Feld ein Lookup-Feld ist – analog zur bereits vorhandenen Feldtyp-Matrix:

```
Private m_bolAktKombi(1 To 5, 1 To 5) As Boolean
```

Die eigentliche Erkennung erledigt **ErmittleKombiInfo**. Die Prozedur sucht im datentragenden Formular nach einem Steuerelement, dessen Datensatzherkunft (die Eigenschaft **ControlSource**) dem Feldnamen entspricht, und kopiert dessen Eigenschaften in die Struktur (siehe Listing 1).

Hier ist **m_frmDaten** der entscheidende Punkt. Das zu filternde Recordset gehört immer zu genau einem Formular – entweder dem Hauptformular selbst oder dem konkret angegebenen Unterformular.

Und genau in diesem Formular sitzen auch die gebundenen Kombinationsfelder, deren Liste wir brauchen. Die Funktion **SucheKombiSteuerelement** durchsucht deshalb Steuerelemente dieses einen Formulars, wie Listing 2 zeigt.

Damit dieses datentragende Formular überhaupt zur Verfügung steht, übergeben wir es, wie oben bereits beschrieben, statt wie bisher nur das Recordset.

```
Private Sub ErmittleKombiInfo(strFeld As String, ByRef tKombi As TKombiInfo)
    Dim cbo As ComboBox
    tKombi.IstKombi = False
    If m_frmAufrufend Is Nothing Then
        Exit Sub
    End If
    Set cbo = SucheKombiSteuerelement(m_frmDaten, strFeld)
    If cbo Is Nothing Then
        Exit Sub
    End If
    If Len(Nz(cbo.RowSource, "")) = 0 Then
        Exit Sub
    End If
    tKombi.IstKombi = True
    tKombi.RowSource = cbo.RowSource
    tKombi.RowSourceType = cbo.RowSourceType
    tKombi.ColumnCount = cbo.ColumnCount
    tKombi.ColumnWidths = Nz(cbo.ColumnWidths, "")
    tKombi.BoundColumn = cbo.BoundColumn
End Sub
```

Listing 1: **ErmittleKombiInfo** merkt sich die Liste eines gebundenen Kombinationsfelds

Schritt 2: Passende Bedingungen anbieten

Für ein Lookup-Feld ergeben nur diskrete Vergleiche Sinn. Operatoren wie **Größer als** oder **Enthält** wären hier unpassend, weil der Benutzer ja einen konkreten Listeneintrag auswählt. Gleichzeitig wollen wir für die Nachschlagefelder keine Eingabe von Vergleichsausdrücken mit Platzhaltern erlauben. Deshalb spendieren wir dem Modul **mdlFilter** eine eigene, schlanke Bedingungsliste, die das folgende Listing wiedergibt:

```
Private Function SucheKombiSteuerelement(frm As Form, strFeld As String) As ComboBox
    Dim ctl As Control
    For Each ctl In frm.Controls
        If ctl.ControlType = acComboBox Then
            If StrComp(Nz(ctl.ControlSource, ""), strFeld, vbTextCompare) = 0 Then
                Set SucheKombiSteuerelement = ctl
                Exit Function
            End If
        End If
    Next ctl
End Function
```

Listing 2: SucheKombiSteuerelement durchsucht das datentragende Formular

```
Public Function GetBedingungRowsourceKombi() As String
    GetBedingungRowsourceKombi = _
        "Gleich;Nicht gleich;Ist leer;Ist nicht leer"
End Function
```

```
Private Sub FeldGewaehlt(g As Integer, z As Integer)
    Dim strFeld As String
    Dim lngFeldtyp As Long
    Dim i As Integer
    Dim intIdx As Integer
    strFeld = Nz(Me("cboFeld_" & g & "_" & z).Value, "")
    If Len(strFeld) = 0 Then Exit Sub
    intIdx = -1
    For i = 0 To m_intFeldanzahl - 1
        If m_strFeldnamen(i) = strFeld Then
            lngFeldtyp = m_lngFeldtypen(i)
            intIdx = i
            Exit For
        End If
    Next i
    m_lngAktFeldtyp(g, z) = lngFeldtyp
    m_bolAktKombi(g, z) = False
    If intIdx >= 0 Then
        If m_Kombi(intIdx).IstKombi Then m_bolAktKombi(g, z) = True
    End If
    If m_bolAktKombi(g, z) Then
        Me("cboBedingung_" & g & "_" & z).RowSource = GetBedingungRowsourceKombi()
    Else
        Me("cboBedingung_" & g & "_" & z).RowSource = GetBedingungRowsource(lngFeldtyp)
    End If
    ...
```

Listing 3: Die Prozedur **FeldGewaehlt** (Teil 1)

```
    ...
    Me("cboBedingung_" & g & "_" & z).Enabled = True
    Me("cboBedingung_" & g & "_" & z).Value = Null
    Me("txtWert_" & g & "_" & z).Visible = False
    Me("txtWert_" & g & "_" & z).Left = 4440
    Me("txtDatum_" & g & "_" & z).Visible = False
    Me("txtDatum_" & g & "_" & z).Left = 6560
    Me("txtDatumA_" & g & "_" & z).Visible = False
    Me("txtWert2_" & g & "_" & z).Visible = False
    Me("cboDatumstyp_" & g & "_" & z).Visible = False
    Me("cboJaNein_" & g & "_" & z).Visible = False
    Me("cboWert_" & g & "_" & z).Visible = False
    If m_bolAktKombi(g, z) Then
        With Me("cboWert_" & g & "_" & z)
            .RowSourceType = m_Kombi(intIdx).RowSourceType
            .RowSource = m_Kombi(intIdx).RowSource
            .ColumnCount = m_Kombi(intIdx).ColumnCount
            .ColumnWidths = m_Kombi(intIdx).ColumnWidths
            .BoundColumn = m_Kombi(intIdx).BoundColumn
            .Left = 4440
            .Value = Null
        End With
    End If
    If z = m_intAktiveZeilen(g) And z < cMaxZeilen Then
        m_intAktiveZeilen(g) = z + 1
        ZeileEinblenden g, z + 1
        AktualisiereFormularHoehe
    End If
    AktualisiereSchaltflaechen
End Sub
```

Listing 4: Die Prozedur **FeldGewaehlt** (Teil 2)

Die Prozedur **FeldGewaehlt** ist die zentrale Reaktion auf die Feldauswahl (erster Teil siehe Listing 3). Sie wird von allen 25 **After_Update**-Ereignissen der **cboFeld**-Kombinationsfelder über die Parameter **g** (Gruppe) und **z** (Zeile) aufgerufen und richtet die betreffende Filterzeile vollständig ein. Zuerst liest sie den gewählten Feldnamen aus, wobei das umschließende **Nz** den noch leeren Zustand abfängt. Ist kein Feld gewählt – etwa weil der Benutzer die Auswahl wieder geleert hat –, bricht die Prozedur sofort ab, denn ohne Feld gibt es nichts einzurichten.

Den Feldtyp liest die Prozedur nicht erneut aus dem Recordset, sondern aus dem beim Öffnen gefüllten Array **m_strFeldnamen**. Bei einem Treffer übernimmt sie den zugehörigen Typ aus **m_lngFeldtypen** und merkt sich zusätzlich den Array-Index in **intIdx**. Dieser Index ist gleich doppelt nützlich: Er identifiziert das Feld eindeutig und dient kurz darauf als Schlüssel in den Kombi-Metadaten.

Der ermittelte Typ wandert in die Matrix **m_lngAktFeldtyp**, damit spätere Ereignisse derselben Zeile – die Tastaturvalidierung oder der SQL-Aufbau – ihn ohne erneute Suche zur Hand haben.

Dass **intIdx** mit dem Wert **-1** startet, ist Absicht: So bleibt ein nicht gefundenes Feld erkennbar.

Im nächsten Schritt entscheidet sich, ob es sich um ein Lookup-Feld handelt. Das Flag **m_bolAktKombi** für diese Zeile wird zunächst auf **False** gesetzt und nur dann auf **True** eingestellt, wenn ein gültiger Index vorliegt und die zugehörige Kombi-Information **lstKombi** meldet – also beim Laden ein an dieses Feld gebundenes Kombinationsfeld im datentragenden Formular gefunden wurde. In der geschachtelten **If**-Konstruktion wird erst der Index geprüft, bevor mit **m_Kombi(intIdx)** auf das Array zugegriffen wird, damit ein ungültiger Index keinen Laufzeitfehler auslöst.

Vom Kombi-Flag hängt nun ab, welche Operatorenliste das Bedingungsfeld erhält. Für ein Lookup-Feld liefert **GetBedingungRowsourceKombi** die schlanke Liste mit nur den vier sinnvollen Vergleichen, für alle anderen Felder kommt wie gewohnt **GetBedingungRowsource** mit der feldtypabhängigen Liste zum Einsatz.

Anschließend wird das Bedingungsfeld aktiviert – beim Anlegen der Zeile war es deaktiviert – und sein Wert auf **Null** gesetzt (siehe Listing 4). Dieses Leeren ist wichtig, denn beim Wechsel des Feldes könnte eine zuvor gewählte Bedingung zum neuen Feldtyp gar nicht mehr passen.

Danach blendet die Prozedur sämtliche Wertfelder der Zeile aus und setzt deren Positionen zurück. Welches Wertfeld tatsächlich gebraucht wird, steht zu diesem Zeitpunkt noch nicht fest – eine Bedingung ist ja noch nicht gewählt –, also wird vorsorglich alles ausgeblendet. Das Zurücksetzen der **Left**-Eigenschaften von **txtWert** und **txtDatum** auf ihre Ausgangswerte ist dabei mehr als Kosmetik: Beide Felder werden bei bestimmten Bedingungen wie Datum „Ist zwischen“ oder „Letzte X Tage“ per Code an andere Positionen verschoben. Ohne das Zurücksetzen bliebe ein Feld nach einem Feldwechsel an der falschen Stelle stehen.

Handelt es sich um ein **Lookup**-Feld, wird das Kombinationsfeld **cboWert** der Zeile mit den gemerkten Eigenschaften des Originalfelds bestückt: Datensatzherkunft, Spaltenanzahl, Spaltenbreiten und gebundene Spalte. Gerade die letzten drei sind entscheidend. Sie sorgen dafür, dass das Feld dieselbe – meist mehrspaltige – Darstellung mit ausgeblendeter ID-Spalte zeigt wie das Original und über **BoundColumn** später den richtigen gebundenen Wert, also die ID, liefert. Das tatsächliche Einblenden übernimmt allerdings nicht **FeldGewaehlt**, sondern erst **BedingungGewaehlt**, sobald eine Bedingung feststeht.

Den Abschluss bilden zwei Aufgaben rund um die Oberfläche. War die gerade befüllte Zeile die letzte aktive der Gruppe und das Maximum von fünf Zeilen noch nicht erreicht, blendet die Prozedur eine weitere leere Eingabezeile ein, erhöht den Zähler **m_intAktiveZeilen** und passt die Formularhöhe an – so steht nach jeder Feldauswahl automatisch eine frische Zeile bereit.

Der abschließende Aufruf von **AktualisiereSchaltflaechen** aktiviert schließlich die Schaltflächen zum Anwenden und Zurücksetzen und steuert die Sichtbarkeit der Schaltfläche zum Hinzufügen einer **ODER**-Gruppe.

Schritt 3: Übrige Prozeduren anpassen

Damit auch das mehrfache Löschen und Wiederherstellen von Zeilen sauber funktioniert, berücksichtigen wir das neue Steuerelement zudem an allen Stellen, an denen die übrigen Wertfelder bereits verwaltet werden – also in **ZeileSetzenSichtbar**, **PositioniereZeilen**, **ZeileLeeren** und **ZeileWiederherstellen**. Das Muster ist dort jeweils dasselbe wie für **txtWert** – ein- und ausblenden, Position setzen, Wert leeren. Die Änderungen wollen wir hier aus Platzgründen nicht vollständig abbilden.

Schritt 4: Den richtigen Wert für SQL

Jetzt kommt der entscheidende Punkt. Ein Kombinationsfeld zeigt zwar einen Text an, liefert über seine Value-Eigenschaft aber den Wert der gebundenen Spalte – bei einem Fremdschlüssel also die ID. Beim Auslesen in **cmdAnwenden_Click** ziehen wir den Kombi-Fall deshalb vor und greifen auf **cboWert** statt auf das Textfeld zu. Der entsprechende Ausschnitt sieht wie folgt aus:

```
If m_bolAktKombi(g, z) Then
    strWert = Nz(Me("cboWert_" & g & "_" & z).Value, "")
Else
    strWert = Nz(Me("txtWert_" & g & "_" & z).Value, "")
    If Len(strWert) = 0 Then
        strWert = Nz(Me("txtDatum_" & g & "_" & z).Value, "")
    End If
End If
```

Den so ermittelten Wert übergeben wir wie bisher an **GetFilterausdruck** – zusammen mit dem unveränderten Feldtyp aus **m_lngAktFeldtyp**. Dadurch setzt die Funktion eventuelle Anführungszeichen korrekt: Bei einem Zahlen-Fremdschlüssel entsteht ein Ausdruck wie **(KategorieID = 3)**, bei einer Textbindung etwa **(Land = 'DE')**. Die SQL-Erzeugung selbst mussten wir also kein einziges Mal anfassen.

Die 25 Kombinationsfelder per Code anlegen

Bleibt die Frage, wie die 25 neuen Steuerelemente in das Formular kommen. Sie von Hand anzulegen, wäre mühsam und fehleranfällig – schließlich müssen alle exakt dem Namensschema **cboWert_g_z** folgen. Wenn Sie also nicht ohnehin die neue Version des Filterformulars in Ihre Anwendung übernehmen wollen, weil Sie dieses gegebenenfalls bereits angepasst haben, können Sie die Kombinationsfelder mit der Prozedur aus Listing 5 einfügen.

Die Prozedur öffnet **frmFilter** versteckt im Entwurf und legt mit **CreateControl** die fehlenden Kombinationsfelder an. Bereits vorhandene Felder überspringt es – der Aufruf lässt sich also gefahrlos wiederholen. Datensatzherkunft, Spalten und Position bleiben hier bewusst leer beziehungsweise auf Standardwerten: Das Formularmodul

```
Public Sub ErstelleCboWert()
    Const cFormName As String = "frmFilter"
    Dim frm As Form
    Dim ctlNeu As Control
    Dim g As Integer, z As Integer
    Dim strName As String
    DoCmd.OpenForm cFormName, acDesign, , , , acHidden
    Set frm = Forms(cFormName)
    For g = 1 To 5
        For z = 1 To 5
            strName = "cboWert_" & g & "_" & z
            If SteuerelementVorhanden(frm, strName) Then GoTo Weiter
            Set ctlNeu = CreateControl(cFormName, acComboBox, acDetail, "", "", 4440, 0, 1985, 300)
            ctlNeu.Name = strName
            ctlNeu.Visible = False
            ctlNeu.LimitToList = True
            ctlNeu.RowSourceType = "Value List"
            ctlNeu.ColumnCount = 1
            ctlNeu.BoundColumn = 1
Weiter:
            Set ctlNeu = Nothing
        Next z
    Next g
    DoCmd.Save acForm, cFormName
    DoCmd.Close acForm, cFormName, acSaveYes
End Sub
```

Listing 5: Nach dem Aufruf von **ErstelleCboWert** sind alle 25 Kombinationsfelder vorhanden

setzt sie ja zur Laufzeit, sobald der Benutzer ein Feld wählt. Wichtig ist allein der korrekte Name und die feste Startposition bei **Left = 4440**.

Den Aufruf erledigt man einmalig im Direktfenster mit **ErstelleCboWert**. Nach kurzer Zeit meldet die Prozedur, wie viele Felder angelegt wurden (siehe Bild 4).

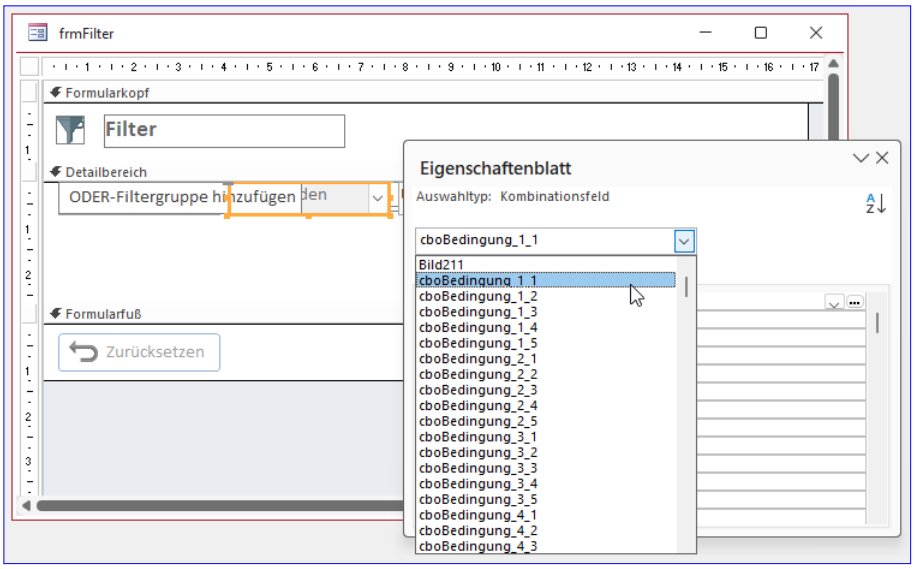

Bild 4: Die neuen Kombinationsfelder im Filter-Formular

Das Ergebnis in der Praxis

Damit ist die Erweiterung fertig. Filtert der Benutzer nun nach einem Lookup-Feld, wählt er den Wert bequem aus derselben Liste, die er aus dem Datenformular kennt – ganz ohne interne IDs. Bild 5 zeigt einen kombinierten Filter, bei dem eine Bedingung das Kategorie-Kombinationsfeld nutzt und eine zweite wie gewohnt ein Textfeld.

Für alle Feldtypen, die kein gebundenes Kombinationsfeld im Originalformular haben, bleibt das Verhalten unverändert: Text-, Zahlen-, Datums- und Ja/Nein-Felder zeigen weiterhin ihre gewohnten Eingabefelder. Die Erweiterung greift ausschließlich dort, wo sie einen echten Mehrwert bietet.

Zusammenfassung und Ausblick

Mit überschaubarem Aufwand haben wir das Filterformular um eine komfortable Funktion erweitert. Wir können nun nicht nur nach Zahlenwerten, Datumsangaben, Ja/Nein und Texten suchen, sondern auch gezielt die Werte aus Nachschlagefeldern als Suchkriterien angeben.

Dabei haben wir nicht nur die Vergleichsoperatoren entsprechend angepasst, sondern auch noch Kombinationsfelder hinzugefügt, mit denen sich die Vergleichswerte auf Basis der tatsächlich enthaltenen Werte einfach auswählen lassen.

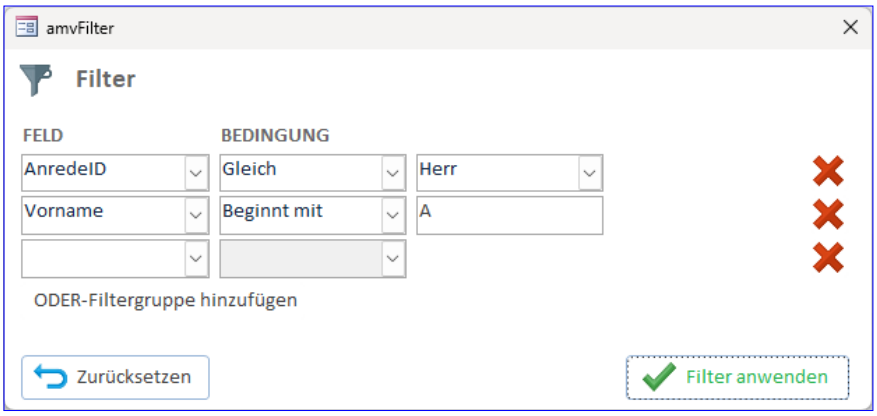

Bild 5: Lookup-Feld und klassisches Textfeld lassen sich frei kombinieren.

In einer weiteren Folge dieser Beitragsreihe fügen wir noch eine Funktion hinzu, mit der wir die einmal eingegebenen Filterkriterien unter dem gewünschten Namen speichern und wiederherstellen können.

Filterformular erweitern: Filter speichern

Bis jetzt kann unser Filterformular schon eine ganze Menge: Man klickt sich seine Suchbedingungen zusammen und sieht sofort nur noch die passenden Datensätze. Doch sobald man das Formular schließt, ist der mühsam eingestellte Filter wieder weg. Wer denselben Filter morgen erneut braucht, muss alles noch einmal eintippen. Das ändern wir jetzt. In diesem Teil bringen wir dem Filterformular bei, einen fertigen Filter unter einem Namen zu speichern und ihn später mit einem Klick wieder hervorzuholen.

Die Idee

Stellen Sie sich vor, Sie malen ein schönes Bild und legen es in eine Schublade. Wollen Sie es später wieder ansehen, holen Sie es einfach heraus. Genau das machen wir mit unseren Filtern: Wir schreiben die eingestellten Bedingungen in eine Tabelle – das ist unsere Schublade – und können sie jederzeit wieder herausholen. Damit das Ganze aufgeräumt bleibt, bekommt jeder gespeicherte Filter einen Namen, den man sich selbst aussucht, zum Beispiel »Große Kunden« oder »Geburtstage diesen Monat«.

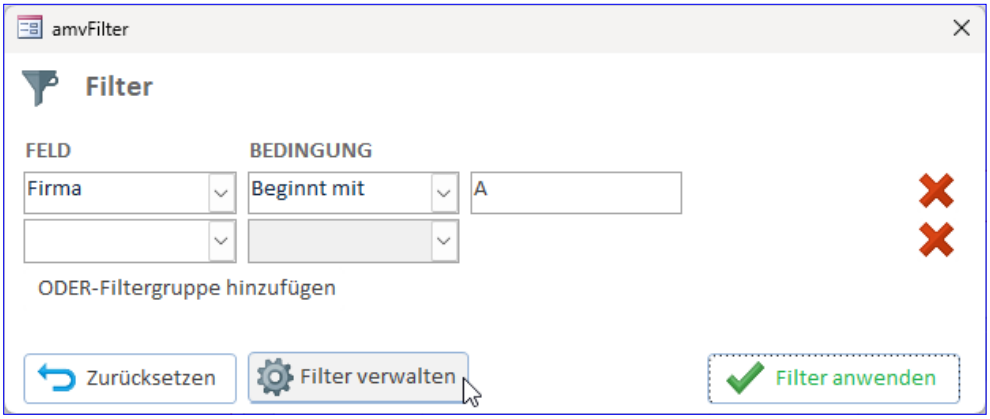

Bild 1: Der neue Knopf »Filter verwalten« öffnet das Verwaltungsfenster

Wichtig ist uns dabei eine Sache: Ein gespeicherter Filter soll nicht nur wieder angewendet werden können, sondern auch wieder vollständig im Formular erscheinen – mit allen Feldern, Bedingungen und Werten. So kann man einen alten Filter laden und ihn bei Bedarf noch ein bisschen anpassen, statt ganz von vorne anzufangen.

Ein Knopf, der alles kann

Man könnte nun zwei Knöpfe einbauen: einen zum Speichern und einen zum Laden. Doch das würde schnell unübersichtlich. Wo wählt man beim Laden aus mehreren Filtern den richtigen aus? Wie löscht man einen, den man nicht mehr braucht? Deshalb gehen wir einen anderen Weg: Ein einziger Knopf mit der Aufschrift **Filter verwalten** im Fuß des Formulars öffnet ein kleines Fenster, in dem man alles erledigen kann – speichern, laden, umbenennen und löschen. So bleibt der Formularfuß aufgeräumt.

Wie dieser neue Knopf aussieht, zeigt Bild 1. Er sitzt zwischen den bereits bekannten Schaltflächen zum Anwenden und Zurücksetzen.

Ein Klick darauf öffnet das Verwaltungsfenster **frmFilterVerwaltung** (siehe Bild 2). Links sieht man die Liste der bereits gespeicherten Filter, rechts die Knöpfe für die einzelnen Aktionen. Praktisch: Es werden immer nur die Filter angezeigt, die zu genau diesem Formular gehören – so vermischt sich nichts, wenn man das Filterformular an mehreren Stellen einsetzt.

Die Prozedur hinter der Schaltfläche **frmVerwalten** lautet wie folgt:

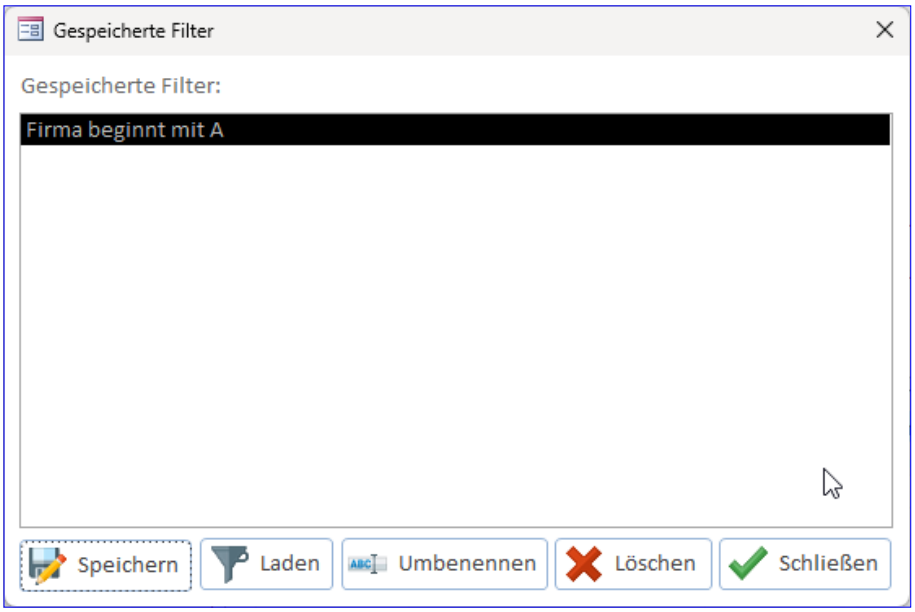

Bild 2: Das Formular zum Verwalten der Filter

```
Private Sub cmdVerwalten_Click()
    DoCmd.OpenForm "frmFilterVerwaltung", , , , , _
        acDialog, DatentragendesFormularName()
End Sub
```

Wohin die Filter wandern

Zum Aufbewahren brauchen wir zwei Tabellen. Die erste, **tblFilter**, merkt sich zu jedem Filter einen Steckbrief: zu welchem Formular er gehört, wie er heißt und wann er angelegt und zuletzt geändert wurde.

Für diese Tabelle haben wir außerdem einen eindeutigen Index über die Felder **Formularname** und **Filtername** definiert, damit keine doppelten Filternamen je Formular vergeben werden können (siehe Bild 3).

Die zweite, **tblFilterZeilen**, speichert die einzelnen Filterzeilen – also welches Feld, welche Bedingung und welcher Wert in jeder Zeile stehen. Weil ein Filter aus mehreren Zeilen bestehen kann, gehören zu einem Steckbrief in der ersten Tabelle ein oder mehrere Einträge in der zweiten.

Den Entwurf dieser Tabelle sehen wir in Bild 4.

Damit beim Löschen eines Filters auch seine Zeilen mitverschwinden, verknüpfen wir die beiden Tabellen mit einer Beziehung, die eine sogenannte Löschweitergabe hat. Löscht man den Steckbrief, räumt Access die zugehörigen Zeilen automatisch mit weg (siehe Bild 5).

Ein kleiner, aber wichtiger Trick steckt bei den Nachschlagefeldern aus dem vorherigen Teil. Dort wählt man

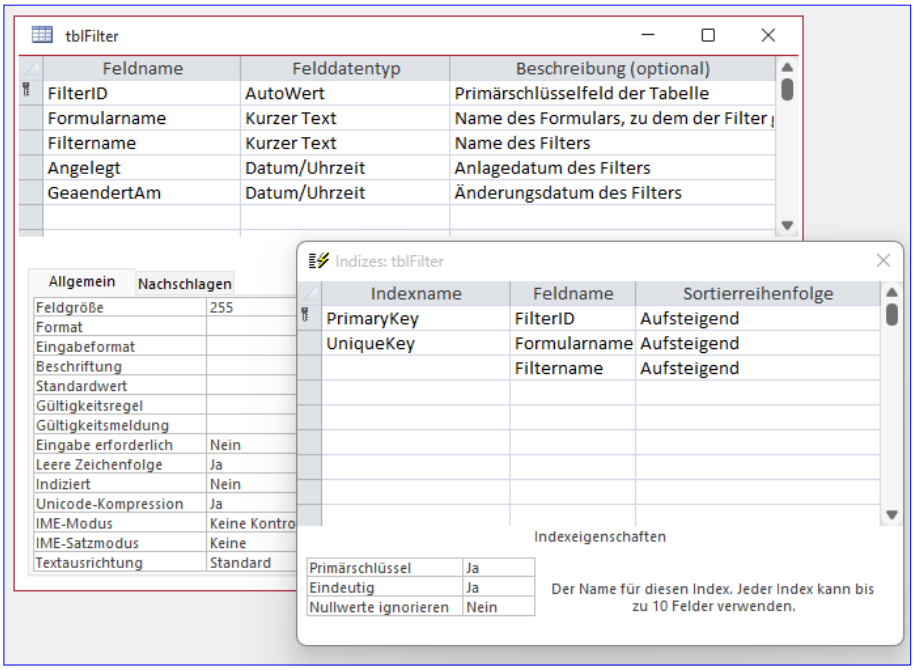

Bild 3: Tabelle zum Speichern der Filter

einen Wert bequem aus einer Liste aus – angezeigt wird zum Beispiel »Stammkunde«, gespeichert wird aber die dahinterliegende Nummer. Genau diese Nummer schreiben wir auch in unsere Schublade.

Beim Hervorholen setzt das Formular die Nummer wieder ein, und die Liste zeigt von selbst wieder den richtigen Text an. So bleibt alles stimmig, selbst wenn sich der angezeigte Text einmal ändert.

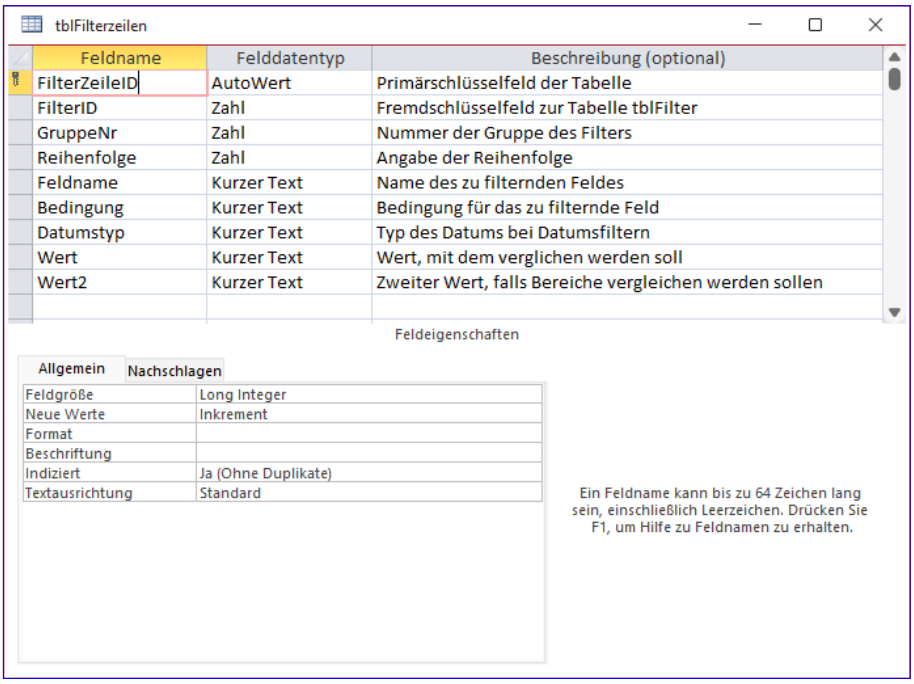

Bild 4: Tabelle zum Speichern der Filterzeilen eines Filters

Einen Filter in die Schublade legen

Das Fenster zum Verwalten der Filter sehen wir in der Entwurfsansicht in Bild 6.

Klickt man hier auf **Speichern**, startet der Speicher-Vorgang für den aktuell definierten Filter.

Die dadurch ausgelöste Prozedur **cmdSpeichern_Click** (siehe Listing 1) besorgt sich zunächst über **FilterFormular** eine Referenz auf das noch geöffnete Filterformular – schließlich liegen die einzustellenden Bedingungen ja dort und nicht im aktuellen Dialog.

Lässt sich das Formular nicht finden, bricht die Prozedur still ab, denn ohne Filterformular gibt es nichts zu speichern.

Danach wird der Name erfragt. Ist in der Liste bereits ein Filter markiert, verwendet die Prozedur dessen Namen als Vorgabe – so lässt sich ein bestehender Eintrag bequem aktualisieren, ohne den Namen erneut eintippen zu müssen. Die

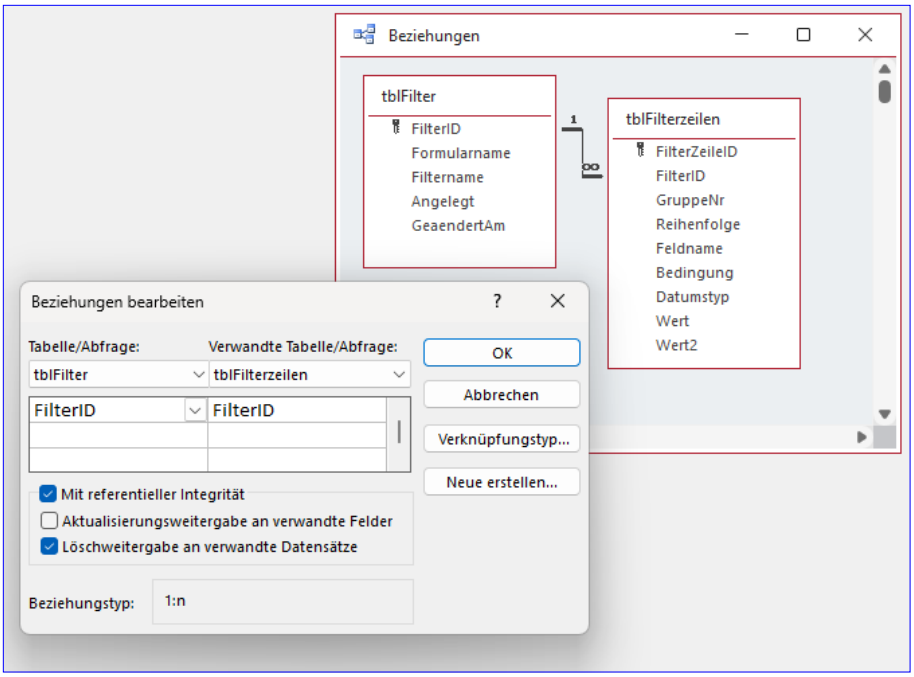

Bild 5: Tabellen der Lösung und ihre Beziehung

InputBox zeigt diesen Vorschlag an, und das umschließende **Trim** entfernt überflüssige Leerzeichen am Rand. Gibt der Benutzer keinen Namen ein oder bricht er ab, endet die Prozedur an dieser Stelle.

Bevor gespeichert wird, klärt **FilterExistiert**, ob unter diesem Namen schon ein Filter abgelegt ist. Falls ja, klärt eine Rückfrage, ob er überschrieben werden soll. Verneint der Benutzer, geschieht nichts weiter; bestätigt er, merkt sich die Prozedur das im Kennzeichen **bolUeberschreiben**. Handelt es sich dagegen um einen neuen

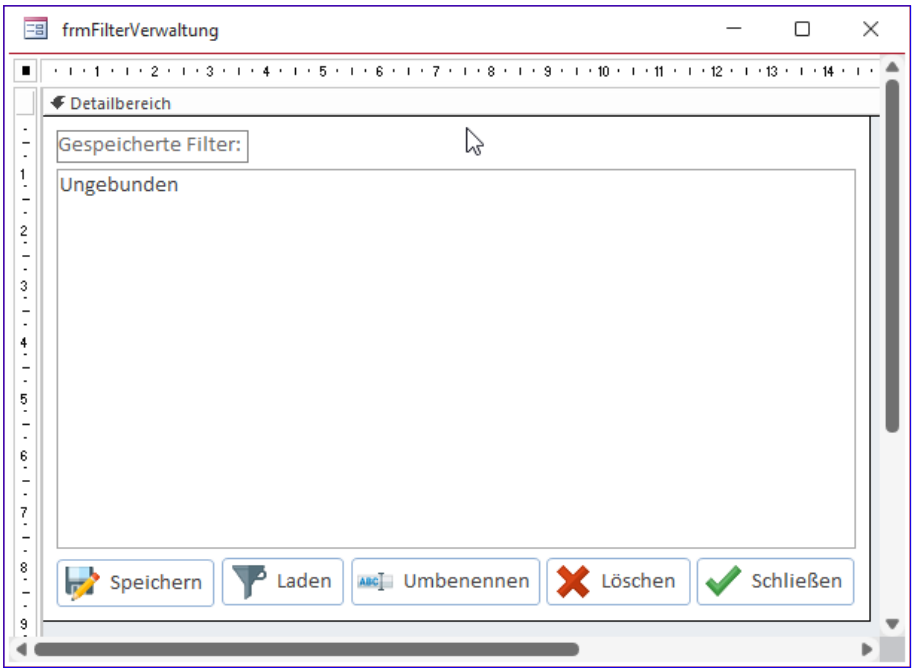

Bild 6: Entwurf des Formulars zum Verwalten der Filter

```
Private Sub cmdSpeichern_Click()
    Dim frm As Form, strName As String, bolUeberschreiben As Boolean, varVorgabe As Variant

    Set frm = FilterFormular()
    If frm Is Nothing Then Exit Sub

    varVorgabe = ""
    If Not IsNull(Me.lstFilter.Value) Then
        varVorgabe = Nz(Me.lstFilter.Column(1), "") 'Filtername
    End If
    strName = Trim(Nz(InputBox("Name fuer den Filter:", "Filter speichern", varVorgabe), ""))
    If Len(strName) = 0 Then Exit Sub

    If FilterExistiert(strName) Then
        If MsgBox("Ein Filter '" & strName & "' existiert bereits." & vbCrLf & "Ueberschreiben?", _
                vbQuestion + vbYesNo, "Ueberschreiben") = vbNo Then
            Exit Sub
        End If
        bolUeberschreiben = True
    End If

    If frm.FilterSpeichern(strName, bolUeberschreiben) Then
        ListeAktualisieren
        EintragMarkieren strName
    End If
End Sub
```

Listing 1: Aufruf des Speichervorgangs für einen Filter

Namen, bleibt dieses Kennzeichen auf **False** – es wird schlicht ein neuer Filter angelegt.

Die eigentliche Arbeit übergibt die Prozedur nun an das Filterformular, indem sie dessen Funktion **FilterSpeichern** mit Namen und Überschreiben-Kennzeichen aufruft.

Meldet diese Erfolg, baut **ListeAktualisieren** die Liste im Dialog neu auf, und **EintragMarkieren** hebt den soeben gespeicherten Filter darin hervor.

Auf diese Weise sieht der Benutzer sofort, dass sein Filter angekommen ist, und das Fenster bleibt für weitere Aktionen geöffnet.

Die Funktion FilterSpeichern im Detail

Das Speichern eines Filters übernimmt die öffentliche Funktion **FilterSpeichern**. Sie erhält zwei Parameter: den vom Benutzer vergebenen Namen sowie ein Kennzeichen, ob ein bereits vorhandener Filter gleichen Namens überschrieben werden darf.

Als Ergebnis liefert sie **True** zurück, wenn das Speichern geklappt hat – so kann der aufrufende Verwaltungsdialog anschließend die Liste aktualisieren. Den ersten Teil des Codes zeigt Listing 2.

Vorbereitung und Namensprüfung

Nach den Variablendeklarationen aktiviert **On Error GoTo Fehler** eine Fehlerbehandlung, die am Ende der Funktion steht. Sollte beim Zugriff auf die Tabellen etwas schiefgehen, landet die Ausführung dort und zeigt eine verständliche Meldung, statt mit einem rohen Laufzeitfehler abzustürzen.

Zunächst ermittelt die Funktion über **DatentragendesFormularName** den Namen des Formulars, dessen Datensätze gefiltert werden. Dieser Name dient später als Zuordnung, damit im Verwaltungsdialog nur die Filter des jeweils passenden Formulars erscheinen. Ist gar kein Name für den Filter angegeben, bricht die Funktion mit einem Hinweis ab – ohne Namen lässt sich ein Filter später schließlich nicht wiederfinden.

Neu anlegen oder überschreiben

Der nächste Block klärt, ob unter diesem Namen bereits ein Filter existiert. Die Hilfsfunktion **FilterIDErmitteln** liefert entweder die Kennung eines vorhandenen Filters oder den Wert 0. Findet sie einen Treffer und wurde das Überschreiben nicht ausdrücklich erlaubt, meldet die Funktion, dass der Name schon vergeben ist, und beendet sich. So kann ein bestehender Filter niemals versehentlich überschrieben werden.

Darf hingegen überschrieben werden, räumt die Funktion zunächst auf: Sie löscht alle bisherigen Zeilen dieses Filters aus **tblFilterZeilen** und vermerkt im Steckbrief über **GeaendertAm** den Zeitpunkt der Änderung. Anschließend werden die Zeilen neu geschrieben – so bleibt kein Rest einer früheren Fassung zurück. Gibt es den Filter dagegen noch nicht, legt die Funktion einen frischen Steckbrief an. Bemerkenswert ist hier die Zeile mit **rstKopf.Bookmark = rstKopf.LastModified**: Sie springt zum eben angelegten Datensatz, um dessen automatisch vergebene Kennung **FilterID** auszulesen. Diese Kennung brauchen wir gleich, um die einzelnen Filterzeilen dem richtigen Steckbrief zuzuordnen.

Die einzelnen Filterzeilen schreiben

Nun folgt das Herzstück (zweiter Teil der Prozedur **FilterSpeichern**, siehe Listing 3). Zwei ineinander verschachtelte Schleifen durchlaufen alle aktiven Gruppen und deren Zeilen – genau in derselben Systematik, mit der auch das Anwenden eines Filters arbeitet. Für jede Zeile liest die Funktion das gewählte Feld und die Bedingung aus. Nur wenn beide gefüllt sind, gilt die Zeile als echte Filterzeile und wird gespeichert; leere Zeilen überspringt die Schleife stillschweigend.

Beim Auslesen des Vergleichswerts zeigt sich, wie eng das Speichern an die übrige Logik angelehnt ist. Handelt es sich um ein Nachschlagefeld, holt die Funktion den

```
Public Function FilterSpeichern(strName As String, bolUeberschreiben As Boolean) As Boolean
    Dim db As DAO.Database
    Dim rstKopf As DAO.Recordset
    Dim rstZeile As DAO.Recordset
    Dim lngFilterID As Long
    Dim strFormular As String
    Dim g As Integer, z As Integer
    Dim strFeld As String, strBedingung As String
    Dim strWert As String, strWert2 As String, strDatumstyp As String

    On Error GoTo Fehler

    strFormular = DatentragendesFormularName()
    If Len(strName) = 0 Then
        MsgBox "Bitte einen Namen fuer den Filter angeben.", vbExclamation, "Name fehlt"
        Exit Function
    End If

    Set db = CurrentDb

    'Pruefen ob schon ein Filter dieses Namens existiert
    lngFilterID = FilterIDErmitteln(strFormular, strName)

    If lngFilterID > 0 And Not bolUeberschreiben Then
        MsgBox "Ein Filter mit diesem Namen existiert bereits.", vbExclamation, "Bereits vorhanden"
        Exit Function
    End If

    If lngFilterID > 0 Then
        'Vorhandene Zeilen entfernen, Kopf aktualisieren
        db.Execute "DELETE FROM tblFilterZeilen WHERE FilterID = " & lngFilterID, dbFailOnError
        db.Execute "UPDATE tblFilter SET GeaendertAm = Now() " & "WHERE FilterID = " & lngFilterID, dbFailOnError
    Else
        'Neuen Kopf anlegen
        Set rstKopf = db.OpenRecordset("tblFilter", dbOpenDynaset)
        rstKopf.AddNew
        rstKopf!Formularname = strFormular
        rstKopf!Filtername = strName
        rstKopf!Angelegt = Now()
        rstKopf!GeaendertAm = Now()
        rstKopf.Update
        rstKopf.Bookmark = rstKopf.LastModified
        lngFilterID = rstKopf!FilterID
        rstKopf.Close
    End If
    ...
```

Listing 2: Prozedur zum Speichern des Filters, Teil 1

```
    ...
    'Zeilen schreiben
    Set rstZeile = db.OpenRecordset("tblFilterZeilen", dbOpenDynaset)
    For g = 1 To m_intAktiveGruppen
        For z = 1 To m_intAktiveZeilen(g)
            strFeld = Nz(Me("cboFeld_" & g & "_" & z).Value, "")
            strBedingung = Nz(Me("cboBedingung_" & g & "_" & z).Value, "")
            If Len(strFeld) > 0 And Len(strBedingung) > 0 Then
                'Wert wie beim Anwenden ermitteln
                If m_bolAktKombi(g, z) Then
                    strWert = Nz(Me("cboWert_" & g & "_" & z).Value, "")
                Else
                    strWert = Nz(Me("txtWert_" & g & "_" & z).Value, "")
                    If Len(strWert) = 0 Then
                        strWert = Nz(Me("txtDatum_" & g & "_" & z).Value, "")
                    End If
                End If
                strWert2 = Nz(Me("txtWert2_" & g & "_" & z).Value, "")
                If Len(strWert2) = 0 Then
                    strWert2 = Nz(Me("txtDatumA_" & g & "_" & z).Value, "")
                End If
                strDatumstyp = Nz(Me("cboDatumstyp_" & g & "_" & z).Value, "")

                rstZeile.AddNew
                rstZeile!FilterID = lngFilterID
                rstZeile!GruppeNr = g
                rstZeile!Reihenfolge = z
                rstZeile!Feldname = strFeld
                rstZeile!Bedingung = strBedingung
                rstZeile!Datumstyp = strDatumstyp
                rstZeile!Wert = strWert
                rstZeile!Wert2 = strWert2
                rstZeile.Update
            End If
        Next z
    Next g
    rstZeile.Close

    FilterSpeichern = True
    Exit Function

Fehler:
    MsgBox "Beim Speichern ist ein Fehler aufgetreten:" & vbCrLf & _
        Err.Description, vbCritical, "Fehler"
End Function
```

Listing 3: Prozedur zum Speichern des Filters, Teil 2

Wert aus dem Auswahl-Kombinationsfeld **cboWert** – und damit die dahinterliegende Nummer, nicht den angezeigten Text.

Andernfalls stammt der Wert aus dem Textfeld **txtWert**; ist dieses leer, wird ersatzweise das Datumsfeld **txtDatum** herangezogen. Nach demselben Muster ermittelt die Funktion einen etwaigen zweiten Wert – nötig bei der Bedingung **Ist zwischen** – aus **txtWert2** beziehungsweise **txtDatumA**. Zusätzlich sichert sie den Datumstyp, damit vordefinierte Zeiträume wie »Diesen Monat« später korrekt wiederhergestellt werden.

Sind alle Daten ermittelt, wird der Datensatz mit **AddNew** angelegt, mit den Werten gefüllt und mit **Update** gespeichert. Die Felder **GruppeNr** und **Reihenfolge** halten dabei fest, an welcher Stelle die Zeile stand – diese Angaben sind später beim Laden entscheidend, damit die Zeilen wieder in der richtigen Anordnung erscheinen. Sind alle Zeilen geschrieben, schließt die Funktion das Recordset, meldet mit **True** den Erfolg und ist fertig.

Einen Filter wieder hervorholen

Das Hervorholen eines gespeicherten Filters ist der spannendste Teil, denn hier muss das Formular seinen früheren Zustand exakt nachbauen.

Der Trick dabei ist einfach und clever zugleich: Statt sich etwas Neues auszudenken, tut das Formular beim Laden genau das, was auch ein Mensch täte. Es wählt ein Feld aus, dann eine Bedingung, dann trägt es den Wert ein – Zeile für Zeile. Dadurch ist garantiert, dass sich ein geladener Filter genauso verhält wie ein von Hand eingegebener.

Damit das klappt, ruft die Prozedur **FilterLaden** für jede gespeicherte Zeile dieselben Bausteine auf, die auch bei der normalen Bedienung laufen: erst das Feld setzen und **FeldGewaehlt** aufrufen, dann die Bedingung setzen und **BedingungGewaehlt** aufrufen, und zum Schluss den Wert eintragen.

Die Prozedur FilterLaden im Detail

Die öffentliche Prozedur **FilterLaden** bekommt die Kennung des gewünschten Filters übergeben und baut den früher gespeicherten Zustand vollständig im Formular wieder auf (erster Teil des Codes siehe Listing 4).

Ein zweiter, optionaler Parameter steuert, ob der Filter nach dem Aufbau gleich angewendet werden soll – voreingestellt ist ja, denn in aller Regel möchte man das geladene Ergebnis sofort sehen.

Erst aufräumen, dann laden

Bevor irgendetwas eingelesen wird, räumt **FilterLaden** das Formular auf. Dafür ruft die Prozedur **FilterZuruecksetzenIntern** auf. Das ist bewusst nicht dieselbe Routine wie hinter dem Zurücksetzen-Knopf: Jene würde nämlich zusätzlich einen leeren Filter an das aufrufende Formular melden und damit kurz alle Datensätze wieder einblenden.

Genau dieses Flackern wollen wir beim Laden vermeiden, deshalb der ausgelagerte Kern, der nur die Anzeige leert, ohne etwas nach außen zu senden.

```
Private Sub FilterZuruecksetzenIntern()
    Dim i As Integer
    Dim j As Integer
    'Alle Felder leeren
    For i = 1 To cMaxGruppen
        For j = 1 To cMaxZeilen
            Me("cboFeld_" & i & "_" & j).Value = Null
            Me("cboBedingung_" & i & "_" & j).Value = Null
            Me("cboBedingung_" & i & "_" & j).Enabled = False
            Me("cboDatumstyp_" & i & "_" & j).Value = Null
            Me("txtWert_" & i & "_" & j).Value = Null
            Me("txtWert_" & i & "_" & j).Left = 4440
            Me("txtDatum_" & i & "_" & j).Value = Null
            Me("txtDatum_" & i & "_" & j).Left = 6560
            Me("txtDatumA_" & i & "_" & j).Value = Null
            Me("txtWert2_" & i & "_" & j).Value = Null
            Me("cboJaNein_" & i & "_" & j).Value = Null
            Me("cboWert_" & i & "_" & j).Value = Null
```

```
Public Sub FilterLaden(lngFilterID As Long, Optional bolAnwenden As Boolean = True)
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim g As Integer, z As Integer
    Dim strDatumstyp As String

    On Error GoTo Fehler

    'Formular zuerst leeren - ohne leeren Filter zu senden
    FilterZuruecksetzenIntern

    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT * FROM tblFilterZeilen WHERE FilterID = " & lngFilterID & " " & _
        "ORDER BY GruppeNr, Reihenfolge", dbOpenSnapshot)

    If rst.EOF Then
        rst.Close
        Exit Sub
    End If

    'Lademodus aktiv: FeldGewaehlt blendet keine Folgezeile ein
    m_bolLadeModus = True

    Do While Not rst.EOF
        g = rst!GruppeNr
        z = rst!Reihenfolge

        'Bei Bedarf Gruppe aktivieren (analog cmdODERHinzufuegen)
        Do While g > m_intAktiveGruppen
            m_intAktiveGruppen = m_intAktiveGruppen + 1
            m_intAktiveZeilen(m_intAktiveGruppen) = 1
            ZeileEinblenden m_intAktiveGruppen, 1
        Loop

        'Bei Bedarf Zeile dieser Gruppe einblenden
        Do While z > m_intAktiveZeilen(g)
            m_intAktiveZeilen(g) = m_intAktiveZeilen(g) + 1
            ZeileEinblenden g, m_intAktiveZeilen(g)
        Loop
        ...
```

Listing 4: Prozedur zum Laden eines Filters, Teil 1

```
            m_lngAktFeldtyp(i, j) = 0
            m_bolAktKombi(i, j) = False
        Next j
        m_intAktiveZeilen(i) = 1
    Next i
    AlleZeilenAusblenden
    m_intAktiveGruppen = 1
    m_intAktiveZeilen(1) = 1
```

```
    ZeileEinblenden 1, 1
    cmdODERHinzufuegen.Visible = False
    cmdZuruecksetzen.Enabled = False
    cmdAnwenden.Enabled = False
    AktualisiereFormularHoehe
End Sub
```

Anschließend öffnet die Prozedur ein Recordset auf Basis der Tabelle **tblFilterZeilen**, beschränkt auf die gewünschte Filterkennung und sortiert nach **GruppeNr** und **Reihenfolge**. Diese Sortierung ist wichtig: Nur so werden die Zeilen später in genau der Anordnung aufgebaut, in der sie einmal gespeichert wurden.

Enthält der Filter überhaupt keine Zeilen, bricht die Prozedur an dieser Stelle bereits ab.

Der Lademodus verhindert Chaos

Jetzt kommt der bereits angesprochene Schalter ins Spiel. Unmittelbar vor der Schleife setzt die Prozedur **m_bolLadeModus** auf **True**. Dadurch unterlässt **FeldGewaehlt** bei jeder Feldauswahl das automatische Einblenden einer leeren Folgezeile.

Würde man darauf verzichten, entstünden nach jeder wiederhergestellten Zeile überflüssige Leerzeilen, und die Zähler für aktive Zeilen gerieten durcheinander. Im Lademodus baut die Prozedur die Struktur stattdessen selbst gezielt auf.

Zeile für Zeile wiederherstellen

Die Hauptschleife durchläuft alle gespeicherten Zeilen. Zu Beginn liest sie aus jedem Datensatz die Gruppen- und die Zeilennummer. Falls die betreffende Gruppe oder Zeile noch nicht sichtbar ist, blendet die Prozedur sie zunächst ein – und zwar so, wie es auch der Knopf zum Hinzufügen einer ODER-Gruppe täte.

Zwei kleine Schleifen sorgen dafür, dass bei Bedarf mehrere Gruppen oder Zeilen nacheinander aktiviert werden, bis die richtige Position erreicht ist.

Der eigentliche Aufbau folgt danach demselben Weg wie eine Eingabe von Hand, und genau darin liegt der Kniff (siehe Listing 5).

Zuerst setzt die Prozedur das Feld und ruft **FeldGewaehlt** auf – damit wird die passende Bedingungsliste erzeugt, ein etwaiges Nachschlagefeld erkannt und dessen Auswahlliste vorbereitet.

Danach setzt sie die Bedingung und ruft **BedingungGewaehlt** auf, wodurch das jeweils richtige Wertfeld eingeblendet wird. Ist ein Datumstyp gespeichert, wird auch dieser gesetzt und mit **DatumstypGewaehlt** verarbeitet. Weil die Prozedur genau die Bausteine der normalen Bedienung nutzt, verhält sich ein geladener Filter am Ende exakt wie ein von Hand eingegebener.

Den Abschluss jeder Zeile bildet das Eintragen der Werte. Diese Aufgabe übernimmt die Hilfsprozedur **WertWiederherstellen**, die den gespeicherten Wert – und bei **Ist zwischen** auch den zweiten Wert – in das jeweils passende Feld schreibt. Bei einem Nachschlagefeld setzt sie die gespeicherte Nummer in das Auswahl-Kombinationsfeld, worauf dieses von selbst wieder den zugehörigen Text anzeigt. Danach rückt die Schleife zum nächsten Datensatz vor.

Den letzten Schliff geben

Ist die Schleife komplett durchlaufen, sind alle gespeicherten Zeilen wiederhergestellt – es fehlt aber noch die leere Eingabezeile, die man von der normalen Bedienung gewohnt ist. Deshalb hängt die Prozedur pro Gruppe noch eine leere Folgezeile an, sofern dort überhaupt noch Platz ist.

So lässt sich ein geladener Filter anschließend genauso bequem erweitern, als hätte man ihn gerade selbst eingegeben.

Zum Schluss schaltet **FilterLaden** den Lademodus wieder aus und bringt die Anzeige in einem Rutsch auf den neuesten Stand: **AktualisiereFormularHoehe** passt die

```
        ...
        'Denselben Weg gehen wie bei manueller Eingabe: Feld -> FeldGewaehlt -> Bedingung -> BedingungGewaehlt
        Me("cboFeld_" & g & "_" & z).Value = Nz(rst!Feldname, "")
        FeldGewaehlt g, z

        Me("cboBedingung_" & g & "_" & z).Value = Nz(rst!Bedingung, "")
        BedingungGewaehlt g, z

        'Datumstyp setzen falls vorhanden
        strDatumstyp = Nz(rst!Datumstyp, "")
        If Len(strDatumstyp) > 0 Then
            Me("cboDatumstyp_" & g & "_" & z).Value = strDatumstyp
            DatumstypGewaehlt g, z
        End If

        'Werte eintragen
        WertWiederherstellen g, z, Nz(rst!Wert, ""), Nz(rst!Wert2, "")

        rst.MoveNext
    Loop
    rst.Close

    'Pro Gruppe eine leere Folge-Eingabezeile anhaengen, damit sich der geladene Filter wie ein manuell eingegebener
    'weiter bearbeiten laesst (sofern noch Platz in der Gruppe).
    For g = 1 To m_intAktiveGruppen
        If m_intAktiveZeilen(g) < cMaxZeilen Then
            m_intAktiveZeilen(g) = m_intAktiveZeilen(g) + 1
            ZeileEinblenden g, m_intAktiveZeilen(g)
        End If
    Next g

    'Lademodus beenden, Zustand gebuendelt aktualisieren
    m_bolLadeModus = False
    AktualisiereFormularHoehe
    AktualisiereSchaltflaechen

    'Optional sofort anwenden
    If bolAnwenden Then
        cmdAnwenden_Click
    End If
    Exit Sub

Fehler:
    m_bolLadeModus = False
    MsgBox "Beim Laden ist ein Fehler aufgetreten:" & vbCrLf & Err.Description, vbCritical, "Fehler"
End Sub
```

Listing 5: Prozedur zum Laden des Filters, Teil 2

Höhe des Formulars an die nun sichtbaren Zeilen an, **AktualisiereSchaltflaechen** schaltet die Knöpfe richtig. Die Prozedur löst nun abschließend denselben Vorgang aus wie ein Klick auf **Filter anwenden**, sodass im Hintergrund sofort die gefilterten Datensätze erscheinen.

Wenn etwas schiefgeht

Über allem liegt eine Fehlerbehandlung. Sollte beim Zugriff auf die Tabelle oder beim Aufbau einer Zeile ein Problem auftreten, springt die Ausführung in den Fehlerzweig.

Dort wird zuerst **m_bolLadeModus** wieder auf False gesetzt – das ist wichtig, damit das Formular nicht versehentlich im Lademodus steckenbleibt und danach keine Folgezeilen mehr einblendet – und anschließend eine verständliche Meldung ausgegeben.

So bleibt das Filterformular auch im Fehlerfall bedienbar.

Einen Filter umbenennen

Manchmal passt ein einmal vergebener Name nicht mehr recht, und man möchte ihn ändern, ohne den Filter neu anlegen zu müssen. Genau dafür sorgt **cmdUmbenennen_Click** (siehe Listing 6).

Zunächst holt sich die Prozedur über **MarkierteFilterID** die Kennung des in der Liste markierten Filters. Ist gar keiner ausgewählt, liefert diese Funktion 0, und ein kurzer Hinweis bittet den Benutzer, zunächst einen Eintrag zu wählen – danach ist Schluss.

Liegt eine gültige Auswahl vor, merkt sich die Prozedur den bisherigen Namen und fragt über eine **InputBox** nach dem neuen. Praktischerweise steht der alte Name dabei schon als Vorschlag im Eingabefeld, sodass man ihn nur anpassen muss. Das umschließende **Trim** entfernt versehentliche Leerzeichen am Rand. Gibt der Benutzer

```
Private Sub cmdUmbenennen_Click()
    Dim lngID As Long
    Dim strAlt As String
    Dim strNeu As String
    lngID = MarkierteFilterID()
    If lngID = 0 Then
        MsgBox "Bitte einen Filter aus der Liste waehlen.", vbExclamation, "Kein Filter gewaehlt"
        Exit Sub
    End If
    strAlt = Me.lstFilter.Column(1)
    strNeu = Trim(Nz(InputBox("Neuer Name:", "Filter umbenennen", strAlt), ""))
    If Len(strNeu) = 0 Or strNeu = strAlt Then Exit Sub

    If FilterExistiert(strNeu) Then
        MsgBox "Ein Filter '" & strNeu & "' existiert bereits.", vbExclamation, "Bereits vorhanden"
        Exit Sub
    End If

    CurrentDb.Execute "UPDATE tblFilter SET Filtername = '" & Replace(strNeu, "'", "''") & "', GeaendertAm = Now() " & _
        "WHERE FilterID = " & lngID, dbFailOnError
    ListeAktualisieren
    EintragMarkieren strNeu
End Sub
```

Listing 6: Umbenennen eines Filters

nichts ein oder lässt er den Namen unverändert, bricht die Prozedur ab, denn dann gibt es nichts zu tun.

Bevor der neue Name übernommen wird, prüft **FilterExistiert**, ob er nicht schon von einem anderen Filter belegt ist. Wäre das der Fall, entstünden zwei Filter gleichen Namens – ein Zustand, den die eindeutige Zuordnung in der Tabelle ohnehin verbietet.

In diesem Fall erscheint ein Hinweis, und die Umbenennung unterbleibt. Erst wenn der Name frei ist, schreibt ein **UPDATE**-Befehl den neuen Namen in **tblFilter** und vermerkt zugleich über **GeaendertAm** den Zeitpunkt der Änderung.

Zum Abschluss wird die Liste neu aufgebaut und der umbenannte Eintrag wieder markiert, damit der Benutzer sofort sieht, dass die Änderung angekommen ist.

Einen Filter löschen

Filter, die man nicht mehr benötigt, lassen sich über **cmdLoeschen_Click** wieder entfernen (siehe Listing 7).

Der Einstieg gleicht dem Umbenennen: Auch hier wird zuerst die markierte Filterkennung ermittelt, und ohne Auswahl weist ein Hinweis freundlich darauf hin, dass erst ein Eintrag zu wählen ist.

Weil Löschen nicht rückgängig zu machen ist, fragt die Prozedur mit dem Namen des betroffenen Filters noch einmal ausdrücklich nach.

Erst wenn der Benutzer bestätigt, geht es weiter – ein Klick auf Nein lässt alles unverändert. Diese kleine Rückfrage bewahrt vor ärgerlichen Versehen.

Das eigentliche Löschen ist dann erfreulich kurz: Ein einziger **DELETE**-Befehl entfernt den Steckbrief aus **tblFilter**. Um die zugehörigen Filterzeilen muss sich die Prozedur nicht kümmern – dank der beim Anlegen der Tabellen eingerichteten Löschweitergabe verschwinden sie automatisch mit.

Danach wird die Liste aktualisiert, und der gelöschte Filter ist verschwunden.

Von Version 3 auf Version 4 umbauen

Wer die dritte Fassung des Filterformulars bereits im Einsatz hat, muss nicht von vorn beginnen. Mit wenigen Handgriffen lässt sich die Speichern-Funktion nachrüsten.

```
Private Sub cmdLoeschen_Click()
    Dim lngID As Long
    Dim strName As String
    lngID = MarkierteFilterID()
    If lngID = 0 Then
        MsgBox "Bitte einen Filter aus der Liste waehlen.", vbExclamation, "Kein Filter gewaehlt"
        Exit Sub
    End If
    strName = Me.lstFilter.Column(1)
    If MsgBox("Filter '" & strName & "' wirklich loeschen?", vbQuestion + vbYesNo, "Loeschen") = vbNo Then Exit Sub

    'Dank Loeschweitergabe werden die Zeilen automatisch entfernt
    CurrentDb.Execute "DELETE FROM tblFilter WHERE FilterID = " & lngID, dbFailOnError
    ListeAktualisieren
End Sub
```

Listing 7: Löschen eines Filters

Die folgenden Schritte führen der Reihe nach durch den Umbau – am Ende steht ein Filterformular, das seine Filter dauerhaft merken kann.

Schritt 1: Tabellen anlegen. Importieren Sie das Modul **mdlFilterSpeichern** aus der Beispieldatenbank in Ihre Anwendung und rufen Sie im Direktfenster einmalig **ErstelleFilterTabellen** auf. Damit entstehen die beiden Tabellen **tblFilter** und **tblFilterZeilen** samt der Beziehung mit Löschweitergabe.

Schritt 2: Verwaltungsformular übernehmen. Holen Sie das Formular **frmFilterVerwaltung** aus der Beispieldatenbank in Ihre Datenbank. Es bringt sein Klassenmodul und alle nötigen Steuerelemente – Listenfeld und Schaltflächen – bereits mit.

Schritt 3: Neue Steuerelemente im Filterformular. Öffnen Sie **frmFilter** im Entwurf und legen Sie im Fuß eine weitere Schaltfläche mit dem Namen **cmdVerwalten** an; als Beschriftung bietet sich **Filter verwalten** an. Die 25 Wert-Kombinationsfelder aus der dritten Fassung bleiben unverändert bestehen.

Schritt 4: Modulvariable ergänzen. Fügen Sie im Klassenmodul von **frmFilter** bei den übrigen Deklarationen die neue Variable **m_bolLadeModus** hinzu. Sie steuert später, dass beim Laden keine überflüssigen Leerzeilen entstehen.

Schritt 5: FeldGewaehlt anpassen. In der Prozedur **FeldGewaehlt** wird der Block, der automatisch die nächste Zeile einblendet, in eine Abfrage auf **m_bolLadeModus** gefasst. So unterbleibt dieses Einblenden, solange ein gespeicherter Filter wiederhergestellt wird. Den genauen Code zeigt der Beitrag im Abschnitt zum Lademodus.

Schritt 6: Neue Prozeduren einfügen. Ergänzen Sie im Klassenmodul von **frmFilter** die Ereignisprozedur **cmdVerwalten_Click** sowie die öffentlichen Routinen **FilterSpeichern** und **FilterLaden** mit ihren Hilfsprozeduren **DatentragendesFormularName**, **FilterIDErmitteln** und **WertWiederherstellen**. Diese Bausteine erledigen das eigentliche Speichern und Wiederherstellen.

Schritt 7: Zurücksetzen aufteilen. Der gemeinsame Kern des Zurücksetzens wandert in die neue Prozedur **FilterZuruecksetzenIntern**, die nur die Anzeige leert.

Die bestehende Prozedur **cmdZuruecksetzen_Click** ruft künftig nur noch diesen Kern auf und sendet anschließend wie gehabt den leeren Filter an das aufrufende Formular.

Auch **FilterLaden** greift auf diesen Kern zurück – allerdings ohne den leeren Filter zu senden, damit beim Laden nichts flackert.

Nach diesen sieben Schritten ist der Umbau abgeschlossen. Ein Klick auf die neue Schaltfläche öffnet das Verwaltungsfenster, und Filter lassen sich fortan speichern, wieder hervorholen, umbenennen und löschen. Da keine bestehende Logik entfernt, sondern nur ergänzt wurde, arbeiten alle Funktionen der dritten Fassung unverändert weiter.

Zusammenfassung

Unser Filterformular hat jetzt ein Gedächtnis bekommen. Einmal eingestellte Filter lassen sich unter einem Namen speichern und später mit einem Doppelklick wieder hervorholen – vollständig aufgebaut und sofort angewendet.

Drei Dinge sind dabei besonders erwähnenswert. Erstens speichern wir die Filter nicht als fertigen Suchbefehl, sondern Zeile für Zeile – nur so lassen sie sich später wieder vollständig im Formular herstellen und bearbeiten. Zweitens baut das Formular einen geladenen Filter auf, indem es dieselben Bausteine aufruft wie bei der Handeingabe, sodass sich beide gar nicht unterscheiden. Und drittens sorgt ein kleiner Schalter dafür, dass beim Hervorholen keine leeren Zwischenzeilen entstehen.

Access-Infos per COM-Add-In anzeigen

Im Beitrag »Access: Version und Architektur einfach ermitteln« (www.access-im-unternehmen.de/1606) haben wir die neuen SysCmd-Konstanten zum Abfragen von Access-Version und Access-Architektur vorgestellt. Dort haben wir auch gezeigt, wie wir diese Informationen übersichtlich in einem Formular darstellen können. Im vorliegenden Beitrag erstellen wir nun ein COM-Add-In, das den Dateibereich von Access erweitert und dort die verschiedenen Versionsinformationen, die Architektur und den aktuellen Update-Kanal von Access anzeigt. Außerdem liefern wir dort auch gleich noch den aktuellen Datenbankpfad und eine Möglichkeit, diesen in die Zwischenablage zu kopieren.

Die neuen **SysCmd**-Konstanten wie **acSysCmdGetMsoBuildNumber** oder **acSysCmdGetBitness** bieten uns die Möglichkeit, Versionsinformationen oder die Architektur der aktuellen Installation von Microsoft Access per VBA zu ermitteln.

Damit erhalten wir eine Alternative dazu, die Informationen umständlich über das Datei-Menü von Access zu ermitteln. Wenn wir allerdings die Version und die Bitness der Anwendung eines Kunden abfragen wollen, ist es wohl ebenso umständlich, diesem die entsprechende **SysCmd**-Anweisung zur Eingabe in den Direktbereich des VBA-Editors zu diktieren und uns das Ergebnis mitteilen zu lassen.

Da wir in früheren Beiträgen bereits einige COM-Add-Ins mit dem zumindest für die 32-Bit-Version kostenlosen twinBASIC programmiert haben (das Kompilieren von 64-Bit-COM-Add-Ins ist auch möglich, allerdings wird dort bei jedem Start ein Splash-Screen eingeblendet),

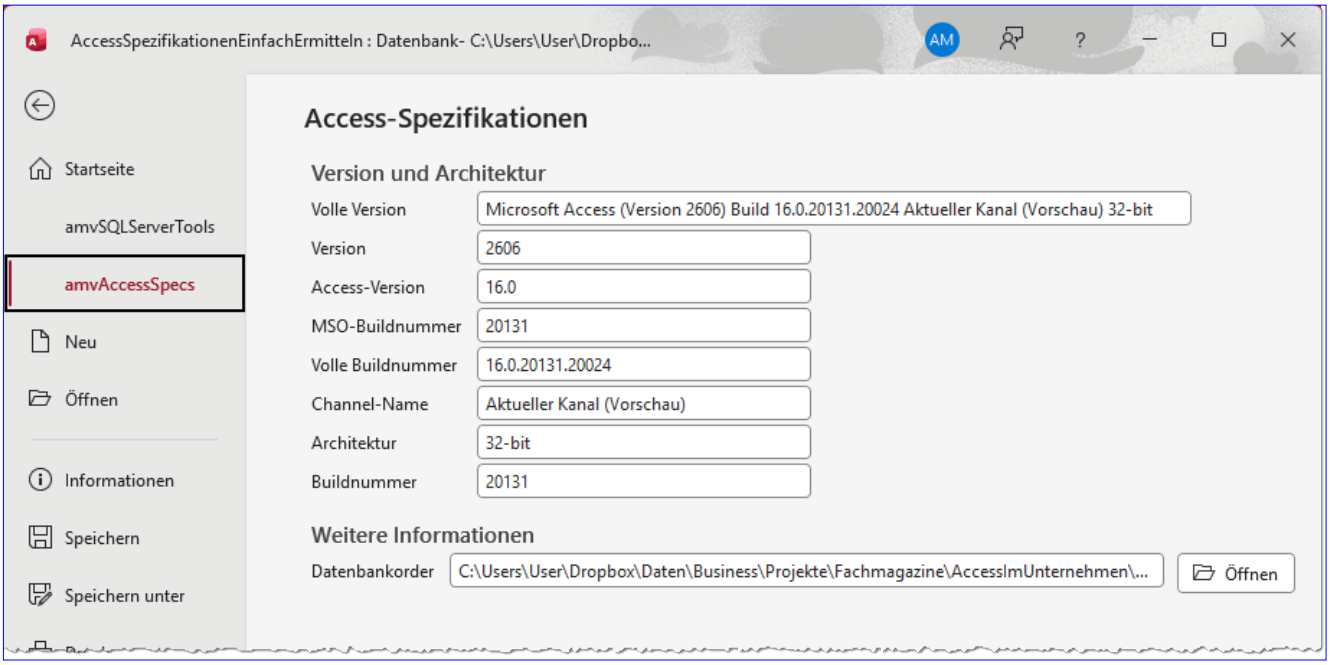

Bild 1: Neuer Bereich im **Datei**-Abschnitt des Ribbons von Microsoft Access

wollen wir hier ein weiteres Beispiel dafür liefern, wie man die Benutzerumgebung von Access selbst um praktische Funktionen erweitern kann.

Das Ergebnis sehen wir vorab in Bild 1. Wenn der Benutzer im Ribbon auf Datei klickt, findet er im Datei-Menü direkt einen Eintrag namens **amvAccessSpecs**. Dahinter zeigt sich die Ansicht mit den Access-Spezifikationen, die wir überwiegend mit den neuen **SysCmd**-Konstanten ermittelt haben.

Außerdem haben wir noch den aktuellen Datenbankpfad hinzugefügt, den wir entweder markieren und kopieren oder auch direkt im Windows Explorer öffnen können.

Funktionsweise des COM-Add-Ins

An dieser Stelle wollen wir uns die Beschreibung sparen, wie das vollständige COM-Add-In auf Basis der Vorlage **Sample 5 - MyCOMAddIn** erstellt wird – letztlich müssen wir dort nur den anschließend vorgestellten Code ersetzen.

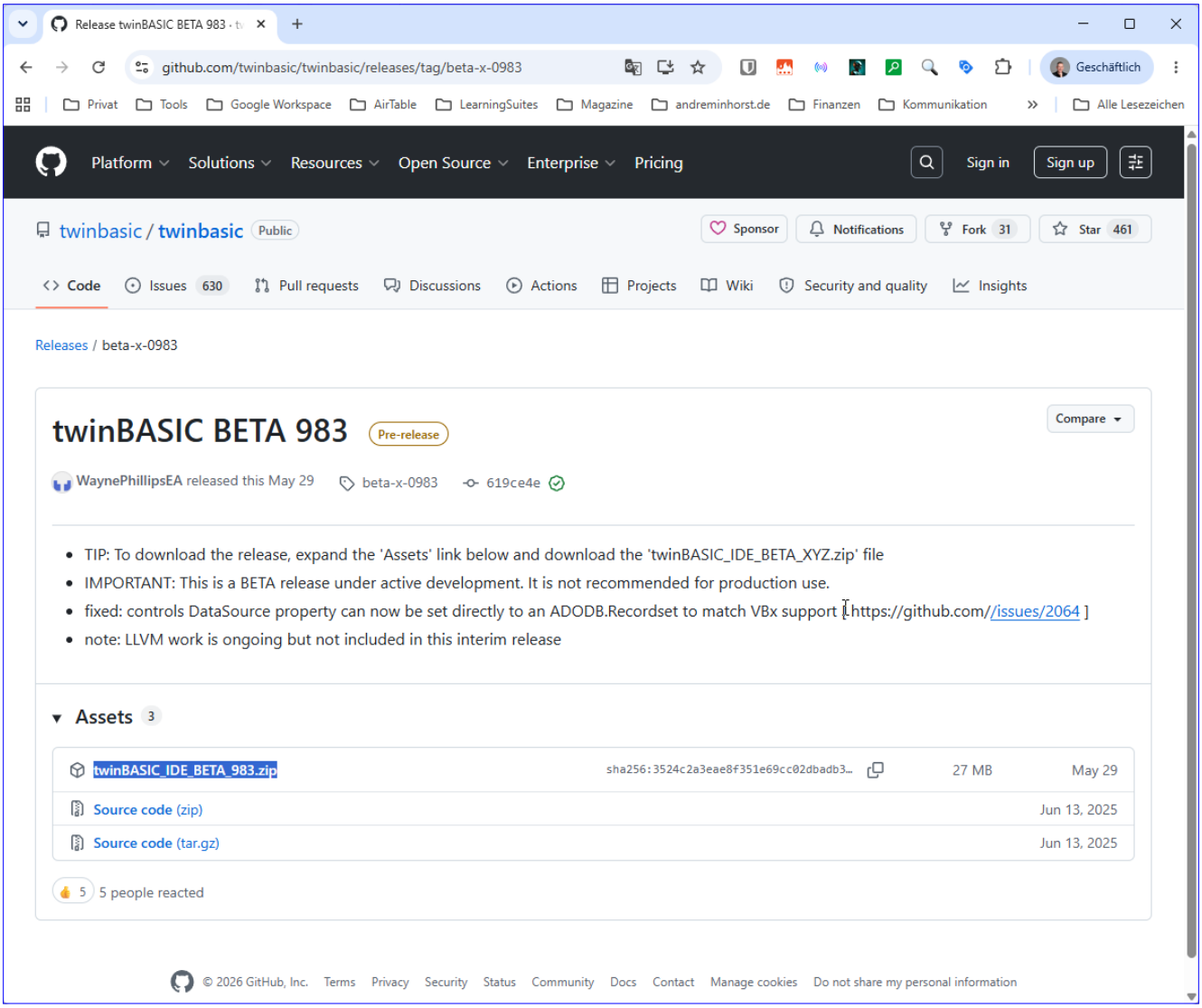

Bild 2: Herunterladen der aktuellen twinBASIC-Version

Also empfehlen wir, zum Studium und für eigene Anpassungen einfach das mit dem Download gelieferte twinBASIC-Projekt namens **amvAccessSpecs.twinproj** zu nutzen.

Außerdem benötigen Sie twinBASIC – die Auflistung der Versionen von twinBASIC finden Sie unter dem folgenden Link:

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

Hier klicken wir auf die aktuellste Version und laden diese wie in Bild 2 herunter – hier die Version **twinBASIC BETA 983**.

Und keine Sorge wegen des Beta-Status – die damit erzeugten COM-Add-Ins funktionieren einwandfrei.

Projekteigenschaften

In den Projekteigenschaften haben wir **Projekt Name**, **Project Description** und **Application Title** jeweils auf **amvAccessSpecs** eingestellt.

Außerdem benötigen wir Verweise auf die Bibliotheken **Microsoft Office 16.0 Object Library** und **Microsoft Access 16.0 Object Library**.

Code des COM-Add-Ins

Der Code, der das COM-Add-In beim Starten von Access lädt und für die Anzeige des neuen Bereichs im Backstage sorgt, befindet sich im Modul **amvAccessSpecs.twin**.

Diesen öffnen wir durch einen Doppelklick auf den entsprechenden Eintrag im Projektexplorer (siehe Bild 3).

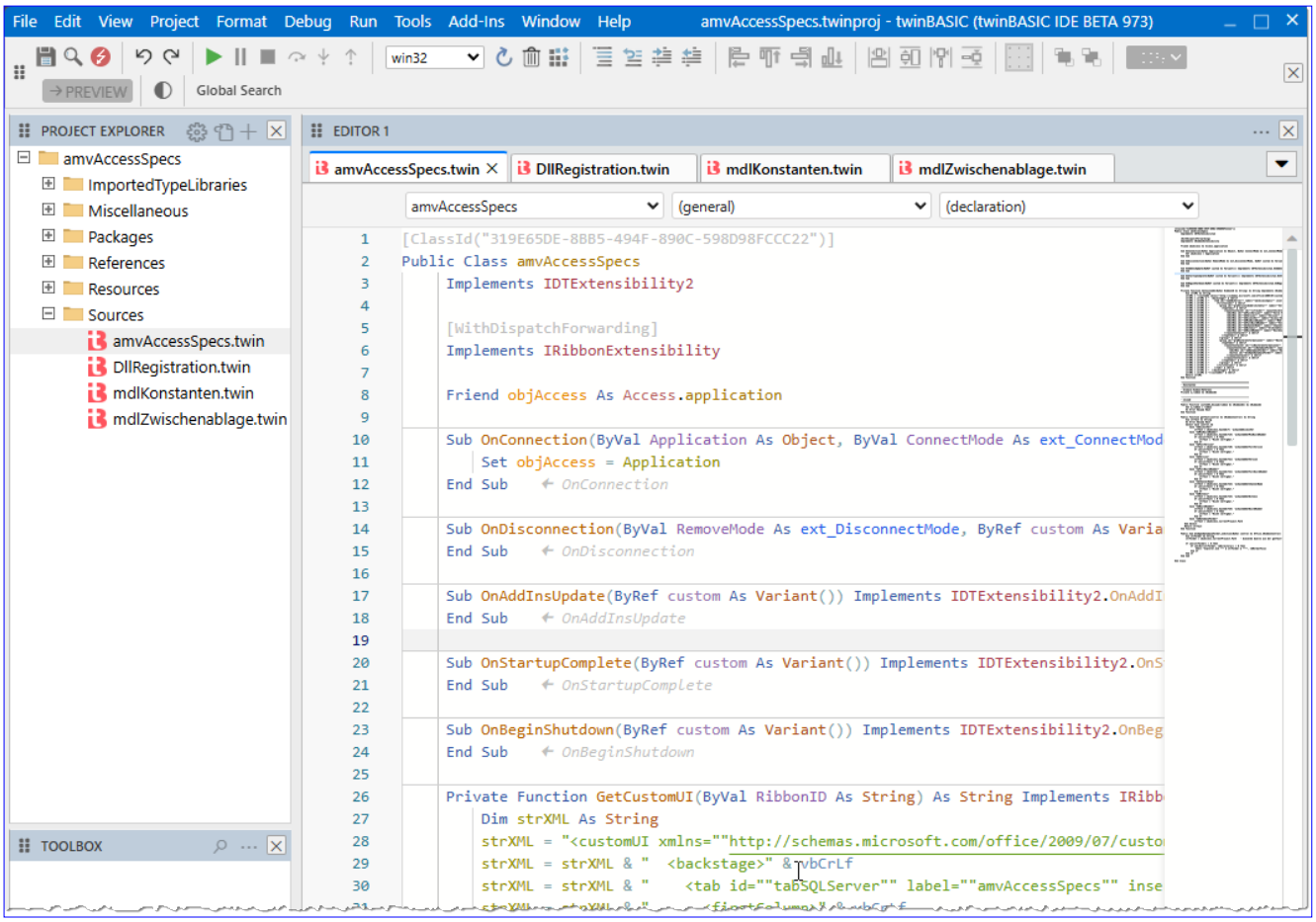

Bild 3: Überblick des Projekts und das Modul **amvAccessSpecs.twin**

Wir schauen uns den Code nun Schritt für Schritt an.

Die Klasse und ihre Schnittstellen

Im oberen Bereich des Moduls finden wir die Deklaration der Klasse **amvAccessSpecs** samt den beiden Schnittstellen, die sie umsetzt.

Ganz oben sehen wir die **ClassId**:

```
[ClassId("319E65DE-8BB5-494F-890C-598D98FCCC22")]
```

Die **ClassId** ist die eindeutige Identität der COM-Klasse – ein weltweit einmaliger Schlüssel (eine GUID), unter dem Windows das COM-Add-In in der Registry findet und startet.

Der Hintergrund: COM-Objekte werden nicht über ihren Namen angesprochen, sondern über diese GUID (CLSID genannt). Wenn Access das Add-In lädt, schaut es in der Registry unter genau dieser Nummer nach, wo die zugehörige Datei liegt und welche Klasse zu instanziieren ist. Der Klassenname **amvAccessSpecs** ist nur für uns als Entwickler da; Windows arbeitet intern mit der GUID.

Deshalb ist zweierlei wichtig: Die Nummer muss wirklich einmalig sein (twinBASIC erzeugt sie zufällig, die Wahrscheinlichkeit einer Kollision ist praktisch null), und sie sollte stabil bleiben. Ändern wir sie nachträglich, gilt das Add-In für Windows als ein völlig anderes Objekt – die alte Registrierung zeigt dann ins Leere, und wir müssten neu registrieren. Beim Weitergeben oder Aktualisieren des Add-Ins lassen wir die **ClassId** also unverändert.

Kurz gesagt: Sie ist der Personalausweis der Klasse, über den COM sie zweifelsfrei wiederfindet.

Anschließend folgt die Definition der Klasse **amvAccessSpecs** selbst mit den beiden Schnittstellen, die sie implementiert:

```
Public Class amvAccessSpecs
    Implements IDTExtensibility2
    [WithDispatchForwarding]
    Implements IRibbonExtensibility
    Friend objAccess As Access.application
```

Eine Schnittstelle ist so etwas wie ein Versprechen: Wer sie umsetzt, sagt zu, bestimmte Prozeduren bereitzustellen. **IDTExtensibility2** sorgt dafür, dass Office unser Add-In starten und beenden kann. **IRibbonExtensibility** erlaubt es uns, dem Menüband – genauer dem Datei-Menü – etwas hinzuzufügen.

Die Zeile **Friend objAccess As Access.application** legt außerdem eine Variable an, in der wir gleich einen Verweis auf Access selbst ablegen. Über diese stellen wir Access später unsere Fragen nach Version und Architektur.

Die Angaben in eckigen Klammern sind technische Markierungen, die twinBASIC für ein COM-Add-In benötigt; um sie müssen wir uns nicht kümmern.

Wenn Access das Add-In lädt

Sobald Access startet und unser Add-In verbindet, ruft es die Prozedur **OnConnection** auf. Genau hier merken wir uns den Verweis auf Access (siehe Listing 1).

Set objAccess = Application speichert einen Verweis auf die übergebene Anwendung in unserer Variablen. Die Schnittstelle **IDTExtensibility2** verlangt noch vier weitere Prozeduren – **OnDisconnection**, **OnAddInsUpdate**, **On-**

```
Sub OnConnection(ByVal Application As Object, ByVal ConnectMode As ext_ConnectMode, ByVal AddInInst As Object, _
        ByRef custom As Variant()) Implements IDTExtensibility2.OnConnection
    Set objAccess = Application
End Sub
```

Listing 1: Beim Verbinden den Verweis auf Access merken

StartupComplete und **OnBeginShutdown** –, die wir hier aber leer lassen, weil wir sie nicht benötigen. Sie müssen aber angelegt sein.

Die Oberfläche im Datei-Menü

Wie unser Bereich im Datei-Menü aussieht, beschreiben wir als XML-Definition. Die Funktion **GetCustomUI** baut diesen Text Zeile für Zeile zusammen und gibt ihn an Access zurück (siehe Listing 2).

Das hat einen Vorteil: Wir bestimmen damit ganz genau, welche Felder und Schaltflächen erscheinen und wie sie angeordnet sind.

Ganz außen steht das Element **customUI**. Es ist die Klammer um alles Weitere und nennt mit **onLoad** die Prozedur, die Access einmal aufrufen soll, sobald die Oberfläche geladen ist.

Darin folgt **backstage** – und das ist genau der Bereich, der angezeigt wird, wenn der Benutzer links oben auf **Datei** klickt – nicht eine der gewöhnlichen Registerkarten des Menübands.

Im **backstage** legen wir mit **tab** unsere eigene Seite an. Sie trägt die Beschriftung **amvAccessSpecs**, und mit der Angabe **insertAfterMso** samt dem Wert **TabInfo** schieben wir sie genau hinter den bereits vorhandenen Bereich **Informationen**. So fügt sich unser Eintrag nahtlos in das gewohnte Datei-Menü ein.

Innerhalb dieser Seite ordnen wir den Inhalt in zwei Gruppen: **Version und Architektur** und **Weitere Informationen**. Eine Gruppe ist dabei nichts weiter als ein zusammengehöriger Block mit einer Überschrift.

Die eigentliche Anordnung der Felder übernimmt in jeder Gruppe ein **layoutContainer**.

Über dessen Angabe legen wir fest, ob die enthaltenen Elemente untereinander (**vertical**) oder nebeneinander (**horizontal**) stehen. Die vielen Versionsfelder stapeln wir untereinander; das Pfadfeld und die Öffnen-Schaltfläche dagegen setzen wir nebeneinander.

Die einzelnen Anzeigefelder sind Elemente vom Typ **editBox**. Jedes bekommt eine eigene Kennung (**id**), eine Beschriftung (**label**) und über **sizeString** eine feste Breite – dafür geben wir schlicht eine Kette von **W**-Buchstaben an, die so breit ist, wie das Feld später sein soll.

Und jedes Feld referenziert mit **getText** die Funktion, die seinen Inhalt liefert.

In der zweiten Gruppe kommt neben dem Pfadfeld noch eine Schaltfläche hinzu, das Element **button**. Sie zeigt über **imageMso** ein vorhandenes Office-Symbol – hier **FileOpen**, das Symbol zum Öffnen – und ruft über **onAction** beim Anklicken unsere Prozedur **btnOpenDatabaseFolder_onAction** auf.

Der eigentliche Kniff an dieser Definitionsweise: Sie besagt nur, welche Felder es gibt und wie sie aussehen – nicht, was darin steht. Den Inhalt holt sich jedes Feld selbst, indem es über seine **id** die Funktion **getText** befragt; ebenso meldet sich die Schaltfläche über **onAction**, sobald sie geklickt wird. So bleiben Aussehen und Inhalt sauber getrennt: Die XML-Definition kümmert sich um die Optik, der VBA-Code um die Werte.

Das Ribbon meldet sich

Wenn Office unsere Oberfläche geladen hat, ruft es einmal die Funktion **customUI_OnLoad** auf. Darin speichern wir den Verweis auf das Menüband in der Variablen **m_ribbon**:

```
Private m_ribbon As IRibbonUI

Public Function customUI_OnLoad(ribbon As IRibbonUI) _
        As IRibbonUI
    Set m_ribbon = ribbon
    On Error Resume Next
End Function
```

```
Private Function GetCustomUI(ByVal RibbonID As String) As String Implements IRibbonExtensibility.GetCustomUI
    Dim strXML As String
    strXML = "<customUI xmlns=""http://schemas.microsoft.com/office/2009/07/customui"" onLoad=""customUI_OnLoad"">" & vbCrLf
    strXML = strXML & "  <backstage>" & vbCrLf
    strXML = strXML & "    <tab id=""tabSQLServer"" label=""amvAccessSpecs"" insertAfterMso=""TabInfo"" " _
        & "title=""Access-Spezifikationen"">" & vbCrLf
    strXML = strXML & "      <firstColumn>" & vbCrLf
    strXML = strXML & "        <group id=""grpVersionsUndArchitektur"" label=""Version und Architektur"">" & vbCrLf
    strXML = strXML & "          <topItems>" & vbCrLf
    strXML = strXML & "            <layoutContainer id=""lcTreiber"" layoutChildren=""vertical"">" & vbCrLf
    strXML = strXML & "              <editBox id=""ebFullVersion"" label=""Volle Version"" " _
        & "sizeString=""WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
    strXML = strXML & "              <editBox id=""ebVersion"" label=""Version"" " _
        & "sizeString=""WWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
    strXML = strXML & "              <editBox id=""ebAccessVer"" label=""Access-Version"" " _
        & "sizeString=""WWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
    strXML = strXML & "              <editBox id=""ebMsoBuildNumber"" label=""MSO-Buildnummer"" " _
        & "sizeString=""WWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
    strXML = strXML & "              <editBox id=""ebFullBuildNumber"" label=""Volle Buildnummer"" " _
        & "sizeString=""WWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
    strXML = strXML & "              <editBox id=""ebChannelName"" label=""Channel-Name"" " _
        & "sizeString=""WWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
    strXML = strXML & "              <editBox id=""ebBitness"" label=""Architektur"" " _
        & "sizeString=""WWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
    strXML = strXML & "              <editBox id=""ebBuildNumber"" label=""Buildnummer"" " _
        & "sizeString=""WWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
    strXML = strXML & "            </layoutContainer>" & vbCrLf
    strXML = strXML & "          </topItems>" & vbCrLf
    strXML = strXML & "        </group>" & vbCrLf
    strXML = strXML & "        <group id=""grpWeitereInformationen"" label=""Weitere Informationen"">" & vbCrLf
    strXML = strXML & "          <topItems>" & vbCrLf
    strXML = strXML & "            <layoutContainer id=""lcWeitereInformationen"" layoutChildren=""vertical"">" & vbCrLf
    strXML = strXML & "              <layoutContainer id=""lcDatabaseFolder"" layoutChildren=""horizontal"">" & vbCrLf
    strXML = strXML & "                <editBox id=""ebDatabaseFolder"" label=""Datenbankorder"" " _
        & "sizeString=""WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
    strXML = strXML & "                <button id=""btnOpenDatabaseFolder"" label=""Öffnen"" " _
        & "imageMso=""FileOpen"" onAction=""btnOpenDatabaseFolder_onAction""/>" & vbCrLf
    strXML = strXML & "              </layoutContainer>" & vbCrLf
    strXML = strXML & "            </layoutContainer>" & vbCrLf
    strXML = strXML & "          </topItems>" & vbCrLf
    strXML = strXML & "        </group>" & vbCrLf
    strXML = strXML & "      </firstColumn>" & vbCrLf
    strXML = strXML & "    </tab>" & vbCrLf
    strXML = strXML & "  </backstage>" & vbCrLf
    strXML = strXML & "</customUI>" & vbCrLf
    Return strXML
End Function
```

Listing 2: Die Funktion **GetCustomUI** stellt die Oberfläche als XML zusammen

Woher die Werte kommen

Nun zum Herzstück. Jedes Textfeld ruft beim Anzeigen die Funktion **getText** auf und gibt dabei seine eigene Kennung mit. Anhand dieser Kennung liefern wir den passenden Wert zurück, wie Listing 3 zeigt. Die eigentlichen Angaben holen wir uns über die neuen **SysCmd**-Konstanten. Jede Kennung steht für eine bestimmte Information:

- **ebAccessVer** – die Access-Version über **SysCmd(7)** (**acSysCmdAccessVer**)
- **ebFullVersion** – die volle Version über **SysCmd(720)**
- **ebVersion** – die Version über **SysCmd(721)**
- **ebMsoBuildNumber** – die MSO-Buildnummer über **SysCmd(715)**
- **ebFullBuildNumber** – die volle Buildnummer über **SysCmd(722)**
- **ebChannelName** – der Update-Kanal über **SysCmd(723)**
- **ebBitness** – die Architektur, also 32 oder 64 Bit, über **SysCmd(724)**
- **ebBuildNumber** – die Buildnummer über **SysCmd(725)**
- **ebDatabaseFolder** – der Pfad der aktuellen Datenbank über **CurrentProject.Path**

Liefert eine dieser Abfragen einen leeren Text, zeigen wir stattdessen **Nicht verfügbar** an – so bleibt kein Feld leer.

Den Ordner öffnen

Bleibt noch die Schaltfläche neben dem Datenbankpfad. Ein Klick darauf ruft die Prozedur **btnOpenDatabaseFolder_onAction** auf:

```
Public Sub btnOpenDatabaseFolder_onAction(ByVal control _
        As Office.IRibbonControl)
    Dim strFolder As String

    strFolder = objAccess.CurrentProject.Path
    If Len(strFolder) > 0 Then
        If Len(Dir(strFolder, vbDirectory)) > 0 Then
            Shell "explorer.exe """ & strFolder _
                & """, vbNormalFocus
        End If
    End If
End Sub
```

Wir holen uns den Pfad der aktuellen Datenbank, und öffnen ihn dann mit **Shell** und **explorer.exe** in einem neuen Fenster.

Das Add-In bekannt machen

Damit Access unser Add-In beim Start überhaupt findet, muss es in der Windows-Registry eingetragen sein. Genau darum kümmert sich das Modul **DllRegistration**.

Es enthält zwei Funktionen: eine zum Eintragen und eine zum Entfernen. Ihre Namen sind nicht frei gewählt, sondern fest vorgegeben – dazu am Ende mehr.

Die Konstanten

Am Anfang des Moduls legen wir vier Konstanten fest (siehe Listing 4).

AddinProjectName übernimmt über **VBA.Compilation.CurrentProjectName** schlicht den Namen des Projekts, also **amvAccessSpecs**. **AddinQualifiedClassName** ermittelt mit **GetDeclaredTypeProgId** die sogenannte ProgID unserer Klasse – den sprechenden Namen, unter dem COM sie kennt. Beide Werte stehen bereits zur Übersetzungszeit fest.

Aus der ProgID setzen wir die Registry-Pfade zusammen. **RootRegistryFolder_ACCESS** zeigt auf den Ort, an dem Access nach seinen COM-Add-Ins sucht: unterhalb von

```
Public Function getText(control As IRibbonControl) As String
    Dim strText As String
    On Error Resume Next
    Select Case control.Id
        Case "ebAccessVer"
            strText = objAccess.SysCmd(7) 'acSysCmdAccessVer
        Case "ebMsoBuildNumber"
            strText = objAccess.SysCmd(715) 'acSysCmdGetMsoBuildNumber
            If Len(strText) = 0 Then
                strText = "Nicht verfügbar."
            End If
        Case "ebFullVersion"
            strText = objAccess.SysCmd(720) 'acSysCmdGetFullVersion
            If Len(strText) = 0 Then
                strText = "Nicht verfügbar."
            End If
        Case "ebVersion"
            strText = objAccess.SysCmd(721) 'acSysCmdGetVersion
            If Len(strText) = 0 Then
                strText = "Nicht verfügbar."
            End If
        Case "ebFullBuildNumber"
            strText = objAccess.SysCmd(722) 'acSysCmdGetFullBuildNumber
            If Len(strText) = 0 Then
                strText = "Nicht verfügbar."
            End If
        Case "ebChannelName"
            strText = objAccess.SysCmd(723) 'acSysCmdGetChannelName
            If Len(strText) = 0 Then
                strText = "Nicht verfügbar."
            End If
        Case "ebBitness"
            strText = objAccess.SysCmd(724) 'acSysCmdGetBitness
            If Len(strText) = 0 Then
                strText = "Nicht verfügbar."
            End If
        Case "ebBuildNumber"
            strText = objAccess.SysCmd(725) 'acSysCmdGetBuildNumber
            If Len(strText) = 0 Then
                strText = "Nicht verfügbar."
            End If
        Case "ebDatabaseFolder"
            strText = objAccess.CurrentProject.Path
    End Select
    Return strText
End Function
```

Listing 3: Die Funktion **getText** füllt die Textfelder.

```
Const AddinProjectName As String = VBA.Compilation.CurrentProjectName
Const AddinQualifiedClassName As String = GetDeclaredTypeProgId(Of amvAccessSpecs.amvAccessSpecs)
Const RootRegistryFolder_ACCESS As String = "HKCU\SOFTWARE\Microsoft\Office\Access\Addins\" & AddinQualifiedClassName & "\"
Const RootRegistryFolder_EXCEL As String = "HKCU\SOFTWARE\Microsoft\Office\Excel\Addins\" & AddinQualifiedClassName & "\"
```

Listing 4: Die Konstanten des Registrierungsmoduls

HKCU – dem Zweig des aktuellen Benutzers – im Bereich der Office-Add-Ins.

Eintragen mit DllRegisterServer

Die erste Funktion trägt unser Add-In ein, wie Listing 5 zeigt.

Wir erzeugen ein **wscript.shell**-Objekt und schreiben damit drei Werte unter den Access-Schlüssel: **FriendlyName** und **Description** erhalten beide den Projektnamen, und **LoadBehavior** bekommt den Wert **3**.

Dieser Wert ist entscheidend – er bedeutet, dass Access das Add-In gleich beim Start laden soll.

Da wir in **HKCU** schreiben, also im Benutzerzweig, sind dafür keine Administratorrechte nötig. Geht beim Schreiben etwas schief, springt die Ausführung zur Marke **RegError**, zeigt eine Meldung mit Fehlerbeschreibung und Fehlernummer und liefert **False**; im Normalfall gibt die Funktion **True** zurück.

Entfernen mit DllUnregisterServer

Das Gegenstück löscht die Einträge wieder (siehe Listing 6).

Hier löschen wir mit **RegDelete** zuerst die drei Werte und anschließend den Schlüssel selbst. Die Fehlerbehandlung arbeitet genauso wie beim Eintragen.

Was beim Kompilieren aus twinBASIC passiert

Die Namen **DllRegisterServer** und **DllUnregisterServer** sind fest vorgegeben: COM erwartet genau diese beiden Funktionen, wenn sich eine DLL selbst registrieren soll. twinBASIC macht sich das zunutze.

Kompilieren wir das Projekt und lassen es registrieren, meldet twinBASIC zum einen die COM-Klasse an und ruft zum anderen unsere Funktion **DllRegisterServer** auf – diese schreibt dann die beschriebenen Einträge. Damit ist das Add-In sofort einsatzbereit, und Access lädt es beim nächsten Start. Beim Aufheben der Registrierung ruft twinBASIC entsprechend **DllUnregisterServer** auf.

```
Public Function DllRegisterServer() As Boolean
    On Error GoTo RegError
    Dim wscript As Object = CreateObject("wscript.shell")
    wscript.RegWrite RootRegistryFolder_ACCESS & "FriendlyName", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_ACCESS & "Description", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder_ACCESS & "LoadBehavior", 3, "REG_DWORD"
    Return True
RegError:
    MsgBox "DllRegisterServer -- An error occured trying to write to the system registry:" & vbCrLf & _
            Err.Description & " (" & Hex(Err.Number) & ")"
    Return False
End Function
```

Listing 5: Die Registry-Einträge anlegen

```
Public Function DllUnregisterServer() As Boolean
    On Error GoTo RegError
    Dim wscript As Object = CreateObject("wscript.shell")
    wscript.RegDelete RootRegistryFolder_ACCESS & "FriendlyName"
    wscript.RegDelete RootRegistryFolder_ACCESS & "Description"
    wscript.RegDelete RootRegistryFolder_ACCESS & "LoadBehavior"
    wscript.RegDelete RootRegistryFolder_ACCESS
    Return True
RegError:
    MsgBox "DllUnregisterServer -- An error occured trying to delete from the system registry:" & vbCrLf & _
            Err.Description & " (" & Hex(Err.Number) & ")"
    Return False
End Function
```

Listing 6: Die Registry-Einträge wieder entfernen

Was regsvr32 damit macht

Auf einem anderen Rechner – etwa beim Kunden – geben wir die fertige DLL weiter und registrieren sie mit dem Windows-Werkzeug **regsvr32**. Rufen wir **regsvr32 amvAccessSpecs.dll** auf, lädt **regsvr32** die DLL und ruft genau unsere Funktion **DllRegisterServer** auf – es läuft also derselbe Code wie beim Kompilieren.

Mit **regsvr32 /u amvAccessSpecs.dll** geht es andersherum: **regsvr32** ruft **DllUnregisterServer** auf und entfernt die Einträge wieder.

Ein Punkt ist dabei wichtig: **regsvr32** und die DLL müssen zur selben Architektur gehören. Eine 32-Bit-DLL registrieren wir mit dem 32-Bit-**regsvr32** aus dem Ordner **SysWOW64**, eine 64-Bit-DLL mit dem 64-Bit-**regsvr32**. Passt die Bitness nicht zusammen, verweigert **regsvr32** die Arbeit.

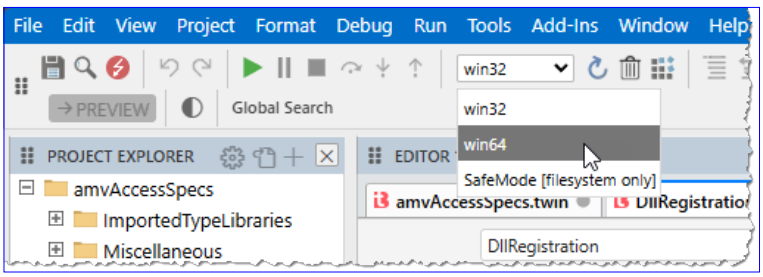

Bild 4: Kompilieren für 32-Bit- und 64-Bit-Access

So genügt derselbe kleine Code sowohl während der Entwicklung in twinBASIC als auch bei der späteren Verteilung per **regsvr32**.

Um die 64-Bit-Version zu kompilieren, stellen wir oben im twinBASIC von **win32** auf **win64** um (siehe Bild 4).

Das Kompilieren selbst erledigen wir mit dem Menübefehl **File|Build**.

Danach steht das COM-Add-In beim nächsten Start von Access bereits zur Verfügung. Wichtig ist, dass wir nicht neu kompilieren, während das COM-Add-In in Access geöffnet ist – dann erhalten wir eine Fehlermeldung.

Erweitern des COM-Add-Ins um weitere Informationen

Wenn Sie das COM-Add-In selbst um weitere Elemente erweitern wollen, sind Änderungen an zwei Stellen nötig.

Angenommen, wir wollen auch noch den Namen der Access-Datenbankdatei abbilden.

Dann fügen wir zuerst ein entsprechendes **editBox**-Element zur XML-Definition hinzu, zum Beispiel unter dem **layoutContainer**-Element

```
strXML = strXML & "              <layoutContainer id=""lcDatabaseName"" layoutChildren=""horizontal"">" & vbCrLf
strXML = strXML & "                <editBox id=""ebDatabaseName"" label=""Datenbankname"" " _
    & "sizeString=""WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW"" getText=""getText""/>" & vbCrLf
strXML = strXML & "              </layoutContainer>" & vbCrLf
```

Listing 7: Hinzufügen eines neuen Text-Elements

lcDatabaseFolder in einem neuen layoutContainer-Element (siehe Listing 7).

Außerdem erweitern wir die Funktion **getText** um den folgenden **Case**-Zweig – hier weisen wir mit **CurrentProject.Name** den Namen der Datenbankdatei zu:

```
Case "ebDatabaseName"
    strText = objAccess.CurrentProject.Name
```

Kompilieren wir das Projekt nun erneut und öffnen den entsprechenden Bereich in Access, sehen wir das neue Steuerelement mit dem entsprechenden Eintrag (siehe Bild 5).

Auf diese Weise können wir beliebige Informationen, die wir aus Access auslesen können, über unser COM-Add-In sichtbar machen.

Zusammenfassung und Ausblick

Dieser Beitrag zeigt, wie wir den Datei-Bereich von Access um eine eigene Sektion erweitern können, in der wir verschiedene Informationen und Steuerelemente unterbringen.

In diesem Fall haben wir die neuen **SysCmd**-Konstanten genutzt, um Informationen über die Version und die Architektur der aktuell installierten Access-Anwendung übersichtlich und mit wenigen Mausklicks darzustellen.

Außerdem haben wir gezeigt, wie dieses COM-Add-In einfach um eigene Steuerelemente erweitert werden kann.

Weitere Ideen zur Erweiterung sind ein Button zum Komprimieren und Reparieren einer Anwendung oder auch zum Anzeigen von Statistiken wie etwa die Anzahl von Tabellen, Abfragen, Formularen, Berichten et cetera.

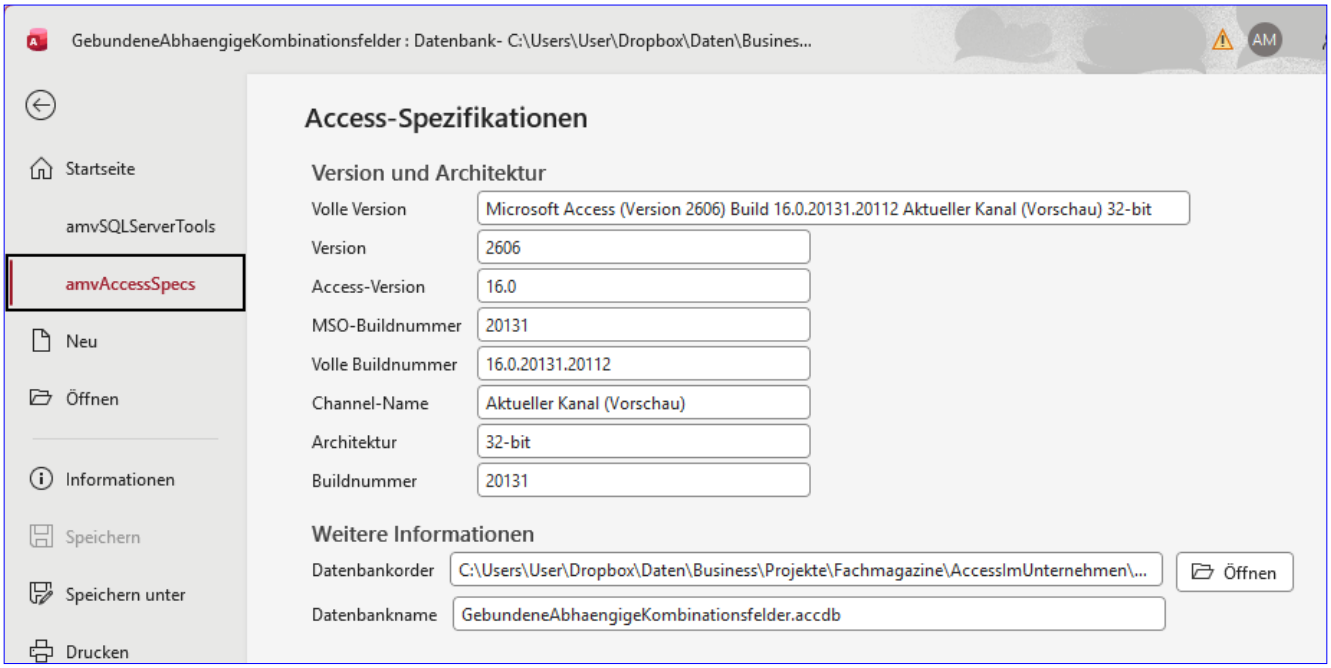

Bild 5: Neuer Eintrag in den Access-Spezifikationen

VBA-Module regelmäßig sichern per COM-Add-In

Wer viel in VBA programmiert, kennt den Schreck: Ein Absturz von Access, und die Arbeit der letzten halben Stunde ist weg. Natürlich kann man von Hand speichern – aber genau das vergisst man im Eifer des Gefechts. In diesem Beitrag bauen wir uns einen fleißigen Helfer, der uns diese Sorge abnimmt: ein COM-Add-In für den Access-Entwickler, das in einstellbaren Abständen alle geänderten Formulare, Berichte und Module automatisch in einzelne Textdateien sichert. So haben wir immer eine frische Kopie unseres Codes zur Hand, ganz ohne daran zu denken.

Was der Helfer leisten soll

Bevor wir in den Code schauen, halten wir kurz fest, was das Add-In eigentlich tun soll. In regelmäßigen Abständen – zum Beispiel alle fünf Minuten – sieht es nach, welche Module im aktuellen Projekt seit dem letzten Speichern geändert wurden. Jedes geänderte Modul schreibt es als eigene Textdatei in einen Unterordner namens **BAS**. Damit dabei nicht bei jedem Durchlauf sämtliche Module erneut gesichert werden, merkt sich das Add-In für jedes Modul einen kleinen Fingerabdruck des Inhalts und legt nur dann eine neue Datei an, wenn sich wirklich etwas geändert hat.

Bedienen lässt sich der Helfer bequem über einen eigenen Bereich im Datei-Menü von Access. Dort schalten wir die automatische Sicherung ein und aus, wählen das Zeitintervall und sehen auf einen Blick, welche Dateien zuletzt gesichert wurden. Wie dieser Bereich aussieht, zeigt Bild 1.

In Bild 2 sehen Sie, wie das Intervall zum Exportieren der geänderten VBA-Module ausgewählt wird.

Ein kleiner Wermutstropfen

Das Sichern bezieht sich nicht nur auf reine VBA-Module – auch wenn der Code in den Klassenmodulen von Formularen oder Berichten geändert wurden, werden diese gespeichert.

Wenn allerdings der Code eines Formulars oder Berichts geändert wurde, muss dieses vor dem Exportieren der neuen Version zunächst gespeichert werden – daher erscheint in diesem Fall jeweils eine Meldung wie die aus Bild 3.

Damit dies beim Programmieren nicht zu sehr stört, empfiehlt es sich, das Intervall entsprechend hochzusetzen.

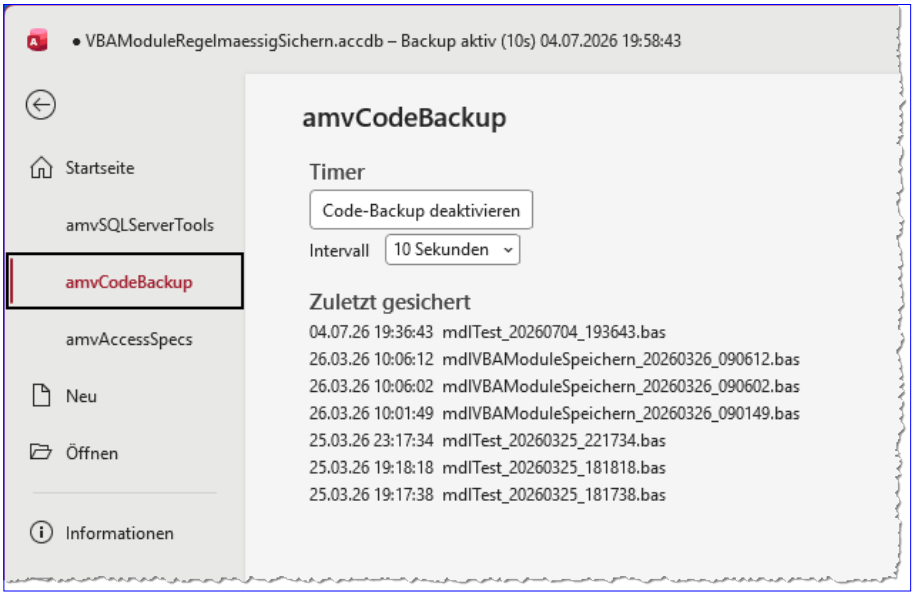

Bild 1: Das COM-Add-In **amvCodeBackup** im Dateimenü von Access

Welches Intervall für Sie optimal funktioniert, probieren Sie am besten selbst aus. Ein kleineres Intervall bringt diese Meldung öfter hervor, ein größeres ist mit dem Risiko behaftet, dass bei einem Absturz von Access eventuell mehr Änderungen verloren gehen.

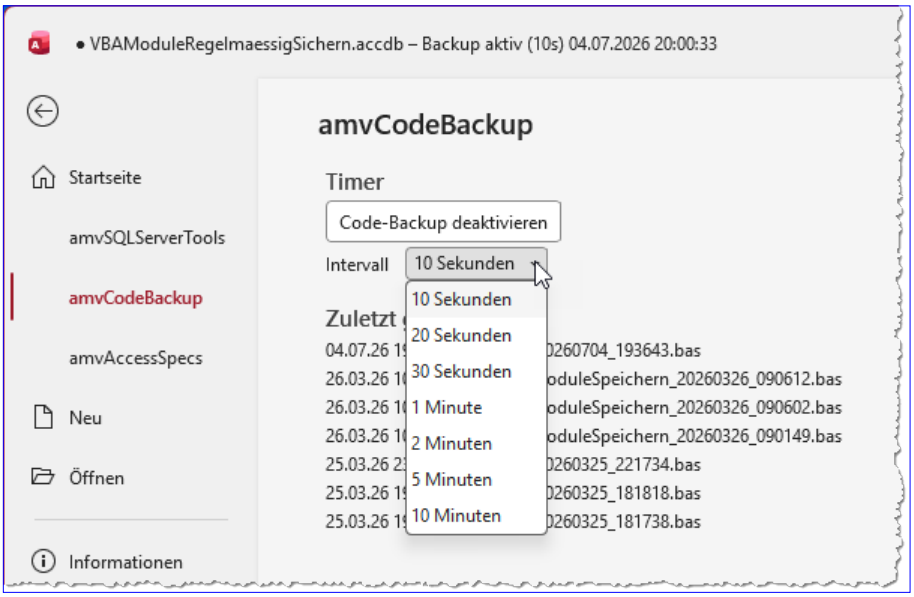

Bild 2: Auswählen des Intervalls zum Anlegen von Sicherungen

Das Projekt in twinBASIC

Wie in früheren Beiträgen entsteht das COM-Add-In mit twinBASIC, das für die 32-Bit-Version kostenlos ist. Wie man ein solches Projekt von Grund auf anlegt, haben wir bereits an anderer Stelle gezeigt; hier konzentrieren wir uns darauf, wie das fertige Projekt **amvCodeBackup** und der darin enthaltene Code funktionieren. Am bequemsten öffnen Sie die mitgelieferte Projektdatei und verfolgen die Bausteine parallel zum Beitrag.

Das Projekt gliedert sich in drei Teile. Die Klasse **amvCodeBackup** steuert den gesamten Ablauf. Das Modul **mdlTimer** verbindet den Windows-Wecker mit der Sicherungsroutine, und das Modul **DllRegistration** macht das Add-In gegenüber Access bekannt. Diese drei Teile gehen wir nun der Reihe nach durch.

Der Kopf der Klasse

Ganz oben in der Klasse **amvCodeBackup** steht die **ClassId**, eine weltweit einmalige Nummer, unter der Windows das Add-In in der Registry wiederfindet. Sie wird von twinBASIC einmalig vergeben und danach nicht mehr angetastet, denn eine geänderte Nummer gälte als ein gänzlich anderes Objekt. Direkt darunter meldet die Klasse die beiden Bausteine an, die sie umsetzt. Hier sehen wir den Kopf der Klasse:

```
[ClassId("DC8E7154-FAC8-43B7-88E5-A9352753999F")]
Class amvCodeBackup
    Implements IDTExtensibility2

    [WithDispatchForwarding]
    Implements IRibbonExtensibility
```

Eine Schnittstelle legt fest, welche Prozeduren eine Klasse bereitstellen muss. Über **IDTExtensibility2** kann Office das Add-In starten und wieder beenden, und **IRibbon-**

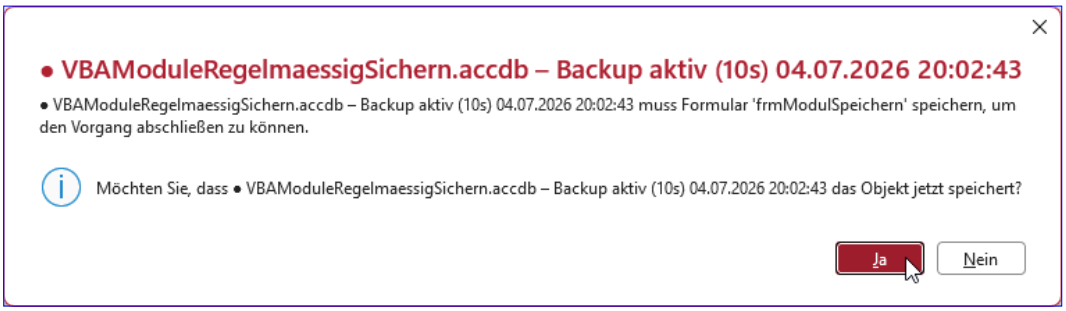

Bild 3: Meldung vor dem Speichern von Formularen oder Berichten

Extensibility öffnet uns den Weg, dem Menüband eigene Elemente hinzuzufügen. Die Angabe **WithDispatchForwarding** in eckigen Klammern ist eine technische Markierung, die twinBASIC für das Ribbon benötigt.

Was sich die Klasse merkt

Damit der Helfer arbeiten kann, hält er mehrere Verweise fest. Hier sehen wir diese im Überblick:

```
Private objAccess As Access.Application
Private objVBE As VBIDE.VBE
Private cbbCodeBackup As Object
Private objRibbon As IRibbonUI
Private WithEvents cbbCodeBackupEvents As VBIDE.Command-
BarEvents
Private cbbCodeBackupOptionen As Object
Private WithEvents cbbCodeBackupOptionenEvents As VBIDE.
CommandBarEvents
```

In **objAccess** liegt später Access selbst, in **objVBE** der VBA-Editor mit seinen Projekten und Modulen – an ihn wenden wir uns, um an den Quellcode zu gelangen. In **objRibbon** merken wir uns das Menüband, um seine Anzeige bei Bedarf auffrischen zu können. Die vier übrigen Variablen gehören zu zwei Einträgen, die wir zusätzlich in die Symbolleiste des Editors setzen: **cbbCodeBackup** zum Ein- und Ausschalten und **cbbCodeBackupOptionen** zum Einstellen des Intervalls. Das Schlüsselwort **WithEvents** sorgt jeweils dafür, dass wir auf Klicks reagieren können.

Ein Wecker aus Windows

Um in festen Abständen aktiv zu werden, borgt sich der Helfer eine Uhr von Windows. Die Funktion **SetTimer** stellt einen Wecker, der nach der eingestellten Zeit auslöst, **KillTimer** schaltet ihn wieder ab. Weil Access in einer 32- und einer 64-Bit-Version existiert und diese Funktionen dort geringfügig anders angesprochen werden, sind sie doppelt vereinbart. Welche Fassung gilt, entscheidet bereits beim Kompilieren die Abfrage **#If Win64 Then** (siehe Listing 1).

SetTimer liefert eine Kennnummer für den laufenden Wecker zurück, die wir in **lngTimerID** ablegen. Solange sie ungleich null ist, wissen wir, dass der Timer arbeitet – ein bequemer Weg, den Ein-/Aus-Zustand abzufragen.

Das Intervall dauerhaft merken

Das gewählte Intervall soll einen Neustart von Access überdauern, deshalb legen wir es in der Registry ab. Unter welchem Namen und in welchem Abschnitt der Wert liegt,

```
#If Win64 Then
    Private Declare PtrSafe Function SetTimer Lib "user32" (ByVal hWnd As LongPtr, ByVal nIDEvent As LongPtr, _
        ByVal uElapse As Long, ByVal lpTimerFunc As LongPtr) As LongPtr

    Private Declare PtrSafe Function KillTimer Lib "user32" (ByVal hWnd As LongPtr, ByVal nIDEvent As LongPtr) As Long

    Private lngTimerID As LongPtr
#Else
    Private Declare Function SetTimer Lib "user32" (ByVal hWnd As Long, ByVal nIDEvent As Long, _
        ByVal uElapse As Long, ByVal lpTimerFunc As Long) As Long

    Private Declare Function KillTimer Lib "user32" (ByVal hWnd As Long, ByVal nIDEvent As Long) As Long

    Private lngTimerID As Long
#End If
```

Listing 1: Die **Timer**-Funktionen werden je nach Architektur passend deklariert

halten ein paar Konstanten fest; fehlt noch ein gespeicherter Wert, greifen 300 Sekunden als Vorgabe. Die Konstanten dazu sehen wie folgt aus:

```
Private Const REG_APP As String = "amvCodeBackup"
Private Const REG_SECTION As String = "Optionen"
Private Const REG_INTERVALL As String = "Intervall"
Private Const REG_INTERVALL_DEFAULT As Long = 300
Private Const MAX_RECENT As Integer = 10
```

Das eigentliche Lesen und Schreiben übernehmen zwei winzige Prozeduren, die sich der fertigen VBA-Funktionen **GetSetting** und **SaveSetting** bedienen. Mehr als der gespeicherte Sekundenwert wandert hier nicht über die Leitung, wie folgende Listings belegen:

```
Private Function IntervallLesen() As Long
    IntervallLesen = CLng(GetSetting(REG_APP, REG_SECTION, _
        REG_INTERVALL, REG_INTERVALL_DEFAULT))
End Function

Private Sub IntervallSchreiben(lngSekunden As Long)
    SaveSetting REG_APP, REG_SECTION, REG_INTERVALL, _
        lngSekunden
End Sub
```

Die Konstante **MAX_RECENT** aus dem vorherigen Listing legt übrigens fest, wie viele der zuletzt gesicherten Dateien wir später in der Liste anzeigen – hier also zehn.

Die Oberfläche als XML beschreiben

Der Bereich im Datei-Menü wird nicht zusammengeklickt, sondern als XML beschrieben. Die Funktion **GetCustomUI** setzt diesen Text Zeile für Zeile zusammen und reicht ihn an Office weiter. Der Vorteil dieser Trennung: Das XML bestimmt allein das Aussehen und die Anordnung, während der eigentliche Inhalt später aus dem Code kommt. Die Definition zeigt Listing 2.

Das äußerste Element **customUI** klammert alles ein und nennt über **onLoad** die Prozedur, die Office einmal aufruft, sobald die Oberfläche geladen ist. Darin folgt **backstage** – jener Bereich, der beim Klick auf Datei erscheint. Mit **tab** legen wir dort eine eigene Seite an und schieben sie über **insertAfterMso** hinter den vorhandenen Bereich. Die erste Gruppe **grpTimer** enthält die Schaltfläche **btnTimer** zum Umschalten und das Auswahlfeld **ddIntervall** mit den festen Zeitstufen. Jedes Element nennt dabei die Funktionen, die es mit Leben füllen – etwa **getLabel** für die Beschriftung und **onAction** für die Reaktion auf einen Klick.

Die zweite Gruppe listet die zuletzt gesicherten Dateien auf. Weil es davon zehn sind, wäre es mühsam, jede Zeile von Hand zu schreiben – eine Schleife erzeugt sie stattdessen.

Jede Zeile erhält eine eigene Kennung von **lblRecent1** bis **lblRecent10** und nennt über **getLabel** die Funktion, die ihren Text liefert. Danach schließt die Definition Gruppe, Spalte, Seite und Bereich wieder ordentlich, und die fertige Zeichenkette geht zurück an Office.

Das Menüband greifbar halten

Sobald Office unsere Oberfläche geladen hat, ruft es die in **onLoad** genannte Funktion auf. Sie tut nur eines, ist aber wichtig: Sie merkt sich das übergebene Menüband in **objRibbon**:

```
Public Sub customUI_OnLoad(ByVal ribbon As IRibbonUI)
    Set objRibbon = ribbon
End Sub
```

Über diesen gemerkten Verweis können wir später einzelne Steuerelemente auffrischen lassen – etwa die Beschriftung der Umschalt-Schaltfläche, sobald sich der Zustand ändert, oder die ganze Liste der letzten Sicherungen, wenn eine neue Datei hinzugekommen ist.

Woher die Liste der letzten Sicherungen kommt

Jede der zehn Listenzeilen fragt beim Anzeigen ihre Beschriftung ab. Damit dafür nicht zehn fast gleiche Funktionen nötig sind, reichen sie ihre Nummer an eine gemeinsame Helferin namens **GetRecentLabel** weiter. Diese liest

```
Private Function GetCustomUI(ByVal RibbonID As String) As String Implements IRibbonExtensibility.GetCustomUI
    Dim strXML As String
    Dim i As Integer
    strXML = "<customUI xmlns=""http://schemas.microsoft.com/office/2009/07/customui"" onLoad=""customUI_OnLoad"">" & vbCrLf
    strXML = strXML & "  <backstage>" & vbCrLf
    strXML = strXML & "    <tab id=""tabCodeBackup"" label=""amvCodeBackup"" insertAfterMso=""TabInfo"" " _
        & "title=""amvCodeBackup"">" & vbCrLf
    strXML = strXML & "      <firstColumn>" & vbCrLf
    strXML = strXML & "        <group id=""grpTimer"" label=""Timer"">" & vbCrLf
    strXML = strXML & "          <topItems>" & vbCrLf
    strXML = strXML & "            <button id=""btnTimer"" getLabel=""btnTimer_GetLabel"" " _
        & "onAction=""btnTimer_OnAction""/>" & vbCrLf
    strXML = strXML & "            <dropDown id=""ddIntervall"" label=""Intervall"" " _
        & "getSelectedItemIndex=""ddIntervall_GetSelectedItemIndex"" onAction=""ddIntervall_OnAction"">" & vbCrLf
    strXML = strXML & "              <item id=""i10"" label=""10 Sekunden""/>" & vbCrLf
    strXML = strXML & "              <item id=""i20"" label=""20 Sekunden""/>" & vbCrLf
    strXML = strXML & "              <item id=""i30"" label=""30 Sekunden""/>" & vbCrLf
    strXML = strXML & "              <item id=""i60"" label=""1 Minute""/>" & vbCrLf
    strXML = strXML & "              <item id=""i120"" label=""2 Minuten""/>" & vbCrLf
    strXML = strXML & "              <item id=""i300"" label=""5 Minuten""/>" & vbCrLf
    strXML = strXML & "              <item id=""i600"" label=""10 Minuten""/>" & vbCrLf
    strXML = strXML & "            </dropDown>" & vbCrLf
    strXML = strXML & "          </topItems>" & vbCrLf
    strXML = strXML & "        </group>" & vbCrLf
    strXML = strXML & "        <group id=""grpRecent"" label=""Zuletzt gesichert"">" & vbCrLf
    strXML = strXML & "          <topItems>" & vbCrLf
    For i = 1 To MAX_RECENT
        strXML = strXML & "            <labelControl id=""lblRecent" & i & """ getLabel=""lblRecent" & i _
            & "_GetLabel""/>" & vbCrLf
    Next i
    strXML = strXML & "          </topItems>" & vbCrLf
    strXML = strXML & "        </group>" & vbCrLf
    strXML = strXML & "      </firstColumn>" & vbCrLf
    strXML = strXML & "    </tab>" & vbCrLf
    strXML = strXML & "  </backstage>" & vbCrLf
    strXML = strXML & "</customUI>" & vbCrLf
    GetCustomUI = strXML
End Function
```

Listing 2: Die Funktion **GetCustomUI** liefert die Ribbon-Definition.

den **BAS**-Ordner nur einmal aus, ordnet die Dateien nach Datum und merkt sich die fertigen Beschriftungen. Den Kern dieser Sortierung hält Listing 3 fest.

Zum Einsatz kommt hier das Bubble-Sort-Verfahren: Es vergleicht immer zwei benachbarte Dateien und vertauscht sie, falls die ältere vor der jüngeren steht. Nach einigen Durchgängen wandert so die neueste Datei nach oben. Die zehn Zeilen holen sich dann nur noch ihren fertigen Eintrag mit Datum, Uhrzeit und Dateinamen – oder einen leeren Text, falls es noch nicht so viele Sicherungen gibt.

```
Private m_astrRecent() As String
Private m_intRecentCount As Integer
Private m_bolRecentReady As Boolean
Private m_sngRecentBuilt As Single

Private Function GetRecentLabel(intPos As Integer) As String
    If Not m_bolRecentReady Or Abs(Timer - m_sngRecentBuilt) > 0.5 Then
        BackupListeAufbauen
    End If
    If intPos > m_intRecentCount Then
        GetRecentLabel = ""
    Else
        GetRecentLabel = m_astrRecent(intPos - 1)
    End If
End Function

Private Sub BackupListeAufbauen()
    Dim strFolder As String, strFile As String, astrName() As String, adtDate() As Date, intCount As Integer
    Dim i As Integer, j As Integer, strTempN As String, dtTempD As Date, objVBProject As VBIDE.VBProject
    m_bolRecentReady = True
    m_sngRecentBuilt = Timer
    m_intRecentCount = 0
    Erase m_astrRecent
    Set objVBProject = GetVBProject()
    If objVBProject Is Nothing Then Exit Sub
    strFolder = Left(objVBProject.FileName, InStrRev(objVBProject.FileName, "\")) & "BAS"
    If Dir(strFolder, vbDirectory) = "" Then Exit Sub
    intCount = 0
    ReDim astrName(0): ReDim adtDate(0)
    strFile = Dir(strFolder & "\*.bas")
    Do While strFile <> ""
        ReDim Preserve astrName(intCount)
        ReDim Preserve adtDate(intCount)
        astrName(intCount) = strFile
        adtDate(intCount) = FileDateTime(strFolder & "\" & strFile)
        intCount = intCount + 1
        strFile = Dir()
    Loop
    If intCount = 0 Then Exit Sub
    For i = 0 To intCount - 2
        For j = 0 To intCount - 2 - i
            If adtDate(j) < adtDate(j + 1) Then
                dtTempD = adtDate(j): adtDate(j) = adtDate(j + 1): adtDate(j + 1) = dtTempD
                strTempN = astrName(j): astrName(j) = astrName(j + 1): astrName(j + 1) = strTempN
            End If
        Next j
    Next i
    ReDim m_astrRecent(intCount - 1)
    For i = 0 To intCount - 1
        m_astrRecent(i) = Format(adtDate(i), "DD.MM.YY HH:NN:SS") & "  " & astrName(i)
    Next i
    m_intRecentCount = intCount
End Sub
```

Listing 3: Die Sicherungsdateien werden nach Datum geordnet, die jüngste zuerst

Das Auswahlfeld für das Intervall

Damit das Auswahlfeld beim Öffnen die richtige Stufe hervorhebt, führt der Code die sieben möglichen Werte in einem kleinen Feld. Beim Anzeigen sucht er darin den gespeicherten Wert und meldet dessen Position zurück, sodass genau diese Stufe vorausgewählt erscheint:

```
Private intervallWerte(6) As Long

Private Sub IntervallWerteInitialisieren()
    intervallWerte(0) = 10 : intervallWerte(1) = 20
    intervallWerte(2) = 30 : intervallWerte(3) = 60
    intervallWerte(4) = 120 : intervallWerte(5) = 300
    intervallWerte(6) = 600
End Sub

Public Function ddIntervall_GetSelectedItemIndex( _
        ByVal control As IRibbonControl) As Integer
    IntervallWerteInitialisieren()
    Dim intAktuell As Integer, i As Integer
    intAktuell = IntervallLesen()
    For i = 0 To UBound(intervallWerte)
        If intervallWerte(i) = intAktuell Then
            ddIntervall_GetSelectedItemIndex = i
            Exit Function
        End If
    Next i
    ddIntervall_GetSelectedItemIndex = 5
End Function
```

Wählt der Benutzer eine andere Stufe, tritt **ddIntervall_OnAction** auf den Plan: Die Prozedur schreibt den neuen Wert in die Registry und startet einen laufenden Timer mit dem geänderten Intervall neu, damit die Umstellung sofort greift.

Ein- und ausschalten über die Schaltfläche

Die Umschalt-Schaltfläche kennt zwei Aufgaben. Zum einen liefert sie über **btnTimer_GetLabel** ihre eigene Beschriftung, die je nach Zustand »aktivieren« oder »deaktivieren« lautet. Zum anderen schaltet **btnTimer_OnAction** beim Klick zwischen an und aus um und frischt danach die Anzeige auf. Beide Routinen sehen wir hier:

```
Public Function btnTimer_GetLabel(control As _
        IRibbonControl) As String
    If TimerLaeuft() Then
        btnTimer_GetLabel = "Code-Backup deaktivieren"
    Else
        btnTimer_GetLabel = "Code-Backup aktivieren"
    End If
End Function

Public Sub btnTimer_OnAction(control As IRibbonControl)
    If TimerLaeuft() Then
        TimerStoppen()
    Else
        TimerStarten()
    End If
    objRibbon.Invalidate
End Sub
```

Der Aufruf **objRibbon.Invalidate** am Ende ist der Grund, warum wir uns das Menüband zuvor gemerkt haben: Er weist Office an, die Anzeige neu aufzubauen, sodass die Schaltfläche sofort ihre aktualisierte Beschriftung zeigt.

Wenn Access das Add-In lädt

Sobald Access startet und unser Add-In verbindet, ruft es **OnConnection** auf. Hier richten wir alles ein, was der Helfer später braucht (siehe Listing 4): Wir merken uns Access und den VBA-Editor, hinterlegen einen Verweis auf unsere eigene Instanz im Timer-Modul und legen über **CreateToolBar** die beiden Menüeinträge an.

Der Zeile **Set mdlTimer.addinInstanz = Me** gebührt besondere Aufmerksamkeit – ihre Bedeutung wird beim Timer gleich klar. Die Schnittstelle **IDTExtensibility2** verlangt noch vier weitere Prozeduren: **OnDisconnection**, **OnAddInsUpdate**, **OnStartupComplete** und **OnBeginShutdown**. Die meisten davon bleiben schlicht, doch **OnStartupComplete** hat eine Aufgabe: Es startet den Ti-

```
Sub OnConnection(ByVal Application As Object, _
        ByVal ConnectMode As ext_ConnectMode, _
        ByVal AddInInst As Object, ByRef custom As Variant()) _
        Implements IDTExtensibility2.OnConnection
    On Error GoTo fehler
    IntervallWerteInitialisieren()
    Set objAccess = Application
    Set objVBE = objAccess.VBE
    Set mdlTimer.addinInstanz = Me
    CreateToolBar()
    Exit Sub
Fehler:
    MsgBox(Err.Number & " " & Err.Description & " " & Erl, _
        vbOKOnly + vbCritical, "OnConnection")
End Sub
```

Listing 4: Beim Verbinden richtet sich das Add-In ein

mer, sobald Access vollständig geladen ist. Beim Beenden räumt **OnDisconnection** wieder auf.

Zwei Einträge in der Symbolleiste

Neben dem Bereich im Datei-Menü setzt das Add-In zwei Einträge in das Add-Ins-Menü des VBA-Editors. Das erledigt **CreateToolBar**, indem es zunächst das vorhandene Menü über seine feste Kennung aufspürt und dann zwei Schaltflächen hinzufügt. Den Ablauf zeigt Listing 5.

Der erste Eintrag dient zum Ein- und Ausschalten, der zweite zum Setzen des Intervalls. Wichtig ist jeweils die letzte Zeile: Erst indem wir das **CommandBarEvents**-Objekt in unseren mit **WithEvents** vereinbarten Variablen ablegen, kann die Klasse später überhaupt auf Klicks reagieren.

Die Reaktionen auf diese Klicks stehen in zwei Ereignisprozeduren. Die erste schaltet den Timer um, die zweite

```
Private Sub CreateToolBar()
    Const vbeAddinsMenuID As Long = 30038
    Dim vbeAddinsMenu As CommandBarControl = objVBE.CommandBars.FindControl(msoControlPopup, vbeAddinsMenuID)

    Set cbbCodeBackup = vbeAddinsMenu.Controls.Add(Temporary:=True)
    With cbbCodeBackup
        .Caption = "amvCodeBackup"
        Set cbbCodeBackupEvents = objVBE.Events.CommandBarEvents(cbbCodeBackup)
    End With

    Set cbbCodeBackupOptionen = vbeAddinsMenu.Controls.Add(Temporary:=True)
    With cbbCodeBackupOptionen
        .Caption = "amvCodeBackup - Intervall setzen"
        Set cbbCodeBackupOptionenEvents = objVBE.Events.CommandBarEvents(cbbCodeBackupOptionen)
    End With
End Sub
```

Listing 5: CreateToolBar hängt zwei Einträge in das Add-Ins-Menü

```
Private Sub cbbCodeBackupEvents_Click(ByVal CommandBarControl As Object, handled As Boolean, CancelDefault As Boolean)
    If TimerLaeuft() Then
        TimerStoppen()
    Else
        TimerStarten()
    End If
End Sub

Private Sub cbbCodeBackupOptionenEvents_Click(ByVal CommandBarControl As Object, _
        handled As Boolean, CancelDefault As Boolean)
    Dim strEingabe As String, lngNeu As Long
    strEingabe = InputBox("Intervall in Sekunden (aktuell: " & IntervallLesen() & "s):", _
        "amvCodeBackup - Intervall", IntervallLesen())
    If strEingabe = "" Then Exit Sub
    If IsNumeric(strEingabe) Then
        lngNeu = CLng(strEingabe)
        If lngNeu < 10 Then
            MsgBox "Mindestintervall: 10 Sekunden."
            Exit Sub
        End If
        IntervallSchreiben lngNeu
        If TimerLaeuft() Then
            TimerStoppen()
            TimerStarten()
        End If
    End If
End Sub
```

Listing 6: Die beiden Menüeinträge reagieren auf Klicks

fragt über eine Eingabebox nach einem neuen Intervall und weist Werte unter zehn Sekunden zurück (siehe Listing 6).

So lässt sich das Intervall auf zwei Wegen einstellen – bequem über die festen Stufen im Datei-Menü oder frei per Eingabebox über das Add-Ins-Menü.

Den Wecker stellen und abschalten

Das Stellen des Weckers besorgt eine eigene Prozedur. Sie ruft **SetTimer** mit dem gespeicherten Intervall auf – umgerechnet in Millisekunden, daher die Multiplikation mit 1000 – und merkt sich die zurückgelieferte Kennnummer.

Das folgende Listing zeigt den Ablauf:

```
Private Sub TimerStarten()
    On Error GoTo Fehler

    lngTimerID = SetTimer(0, 0, IntervallLesen() * 1000, _
        AddressOf mdlTimer.TimerCallback)
    If lngTimerID = 0 Then
        MsgBox "Timer konnte nicht gestartet werden!"
    Else
        StatusSetzen True
    End If
    Exit Sub
Fehler:
    MsgBox(Err.Number & " " & Err.Description & " " & Erl, _
        vbOKOnly + vbCritical, "TimerStarten")
End Sub
```

Der vierte Wert im Aufruf, **AddressOf mdlTimer.TimerCallback**, ist der springende Punkt: Er nennt Windows die Prozedur, die beim Auslösen des Weckers aufgerufen werden soll. Das Gegenstück **TimerStoppen** ruft **KillTimer** auf, setzt die Kennnummer zurück und schaltet die Anzeige auf inaktiv; die kurze Funktion **TimerLaeuft** prüft schlicht, ob die Kennnummer ungleich null ist.

Warum es das Modul mdlTimer braucht

Nun zu der angekündigten Feinheit. Windows kann beim Auslösen des Weckers nur eine gewöhnliche Prozedur aufrufen, keine Methode einer Klasse. Deshalb liegt der Empfänger des Weckrufs im Modul **mdlTimer** statt in der Klasse. Listing 7 macht das deutlich.

Und hier schließt sich der Kreis zu der Zeile aus **OnConnection**. Weil das Modul selbst nicht weiß, zu welcher Klasse es gehört, haben wir dort einen Verweis auf unsere Instanz in **addinInstanz** abgelegt. Löst der Wecker aus, ruft Windows die Modulprozedur auf, und diese reicht den Auftrag über den gemerkten Verweis an die Sicherungsroutine der Klasse weiter.

Sichtbar machen, ob der Helfer wacht

Ob die automatische Sicherung läuft, soll man auf einen Blick erkennen. Dafür ändert **StatusSetzen** gleich mehrere Dinge: Der Menüeintrag bekommt ein kleines Symbol vorangestellt – einen gefüllten Kreis für aktiv, einen leeren für inaktiv – samt passendem Tooltip. Zusätzlich wandert der Zustand sogar in die Titelleiste von Access, wie Listing 8 andeutet.

Das Ändern des Fensternamens übernimmt **AnwendungstitelAendern**. Die Prozedur setzt die Eigenschaft **AppTitle** der Datenbank – und legt sie zuvor an, falls es sie noch nicht gibt, denn nicht jede Datenbank bringt diese Eigenschaft von Haus aus mit. Ein abschließendes **RefreshTitleBar** sorgt dafür, dass die Änderung sofort sichtbar wird.

Das Herzstück: nur Änderungen sichern

Jetzt zum Kern des Add-Ins – jener Prozedur, die der Wecker regelmäßig anstoßt. Sie geht alle Bestandteile des Projekts durch und interessiert sich nur für die seit dem letzten Speichern geänderten; das verrät die Eigenschaft **Saved**. Für jedes solche Modul berechnet sie einen Fingerabdruck und vergleicht ihn mit dem zuletzt gemerkten. Nur bei einem Unterschied entsteht wirklich eine neue Datei, wie in Listing 9 zu sehen ist.

Der Dateiname setzt sich aus dem Modulnamen und dem aktuellen Zeitpunkt zusammen. Dadurch entsteht bei jeder Sicherung eine neue Datei, und ältere Fassungen bleiben erhalten – praktisch, wenn man einmal zu einem früheren Stand zurückkehren möchte. Je nach Art des Bestandteils wählt der Code das richtige Verfahren: Standard- und Klassenmodule schreibt er direkt weg, bei Formularen und Berichten löst er zuvor den eigentlichen Namen aus der Bezeichnung heraus. Zuletzt merkt er sich den neuen

```
Public addinInstanz As Object

#If Win64 Then
Public Sub TimerCallback(ByVal hWnd As LongPtr, ByVal uMsg As Long, ByVal nIDEvent As LongPtr, ByVal dwTime As Long)
#Else
Public Sub TimerCallback(ByVal hWnd As Long, ByVal uMsg As Long, ByVal nIDEvent As Long, ByVal dwTime As Long)
#End If
    If Not addinInstanz Is Nothing Then
        addinInstanz.BackupNotSavedModules
    End If
End Sub
```

Listing 7: Der Weckruf landet im Modul und wird an die Klasse weitergereicht

```
Private Sub StatusSetzen(bolAktiv As Boolean)
    Dim db As DAO.Database
    Dim strDB As String
    On Error GoTo Fehler
    If Not cbbCodeBackup Is Nothing Then
        Set db = objAccess.CurrentDb
        On Error Resume Next
        strDB = Mid(db.Name, InStrRev(db.Name, "\") + 1)
        If Len(strDB) = 0 Then
            Exit Sub
        End If
        If bolAktiv = True Then
            cbbCodeBackup.Caption = "• amvCodeBackup"
            cbbCodeBackup.TooltipText = "Aktiv - Intervall: " & IntervallLesen() & "s"
            Call AnwendungstitelAendern(db, "□ " & strDB & " - Backup aktiv (" & IntervallLesen() & "s) " & Now)
        Else
            cbbCodeBackup.Caption = "o amvCodeBackup"
            cbbCodeBackup.TooltipText = "Inaktiv"
            Call AnwendungstitelAendern(db, "□ " & strDB & " - Backup inaktiv")
        End If
        Set db = Nothing
    End If
    If Not objRibbon Is Nothing Then
        objRibbon.InvalidateControl "btnTimer"
    End If
    Exit Sub
Fehler:
    MsgBox Err.Number & " " & Err.Description & " " & Erl, vbOKOnly + vbCritical, "StatusSetzen"
End Sub
```

Listing 8: StatusSetzen aktualisiert Beschriftung, Tooltip und Fenstertitel

Fingerabdruck, damit dasselbe Modul nicht schon beim nächsten Durchgang erneut gesichert wird.

Der Fingerabdruck eines Moduls

Bleibt die Frage, wie dieser Fingerabdruck entsteht. **ModulHash** liest den gesamten Code eines Moduls und berechnet daraus eine kurze Prüfsumme nach dem MD5-Verfahren. Man kann sie sich als Zahlencode vorstellen, der schon bei der kleinsten Textänderung völlig anders ausfällt – ideal, um Änderungen zu erkennen. Den Kern zeigt Listing 10.

Der Umweg über ein **ADODB.Stream**-Objekt dient allein dazu, den Text sauber in eine Folge von Bytes zu verwandeln, denn genau die erwartet die MD5-Berechnung. Geht dabei einmal etwas schief, greift ein Ersatz, der aus der Länge des Codes und der Zeilenzahl eine einfachere Kennung bildet – weniger genau, aber für unseren Zweck völlig ausreichend.

Das Add-In bekannt machen

Damit Access den Helfer beim Start überhaupt findet, muss er in der Registry stehen. Darum kümmert sich das Modul **DllRegistration**. Am Anfang setzt es aus dem Projekt- und Klassennamen die Registry-Pfade zusammen:

```
Const AddinProjectName As String = _
    VBA.Compilation.CurrentProjectName
```

```
Public Sub BackupNotSavedModules()
    Dim objVBComponent As VBIDE.VBComponent, objVBProject As VBIDE.VBProject, strModulename As String
    Dim strModuleFolder As String, strModulePath As String, strAccessName As String, strHashAktuell As String
    Dim bolEtwasGesichert As Boolean
    If dicHashes Is Nothing Then
        Set dicHashes = CreateObject("Scripting.Dictionary")
    End If
    Set objVBProject = GetVBProject()
    If objVBProject Is Nothing Then Exit Sub
    strModuleFolder = Left(objVBProject.FileName, InStrRev(objVBProject.FileName, "\")) & "BAS"
    On Error Resume Next
    MkDir strModuleFolder
    On Error GoTo 0
    bolEtwasGesichert = False
    For Each objVBComponent In objVBProject.VBComponents
        If objVBComponent.Saved = False Then
            strHashAktuell = ModulHash(objVBComponent)
            If Not dicHashes.Exists(objVBComponent.Name) Or dicHashes(objVBComponent.Name) <> strHashAktuell Then

                strModulename = objVBComponent.Name
                strModulePath = strModuleFolder & "\" & strModulename & "_" & Format(Now, "YYYYMMDD_HHNNSS") & ".bas"

                Select Case objVBComponent.Type
                    Case vbext_ct_StdModule, vbext_ct_ClassModule
                        objAccess.SaveAsText acModule, strModulename, strModulePath
                    Case vbext_ct_Document
                        strAccessName = Mid(strModulename, InStr(strModulename, "_") + 1)
                        If Left(strModulename, 5) = "Form_" Then
                            objAccess.SaveAsText acForm, strAccessName, strModulePath
                        ElseIf Left(strModulename, 7) = "Report_" Then
                            objAccess.SaveAsText acReport, strAccessName, strModulePath
                        End If
                End Select
                dicHashes(objVBComponent.Name) = strHashAktuell
                bolEtwasGesichert = True
                If Not cbbCodeBackup Is Nothing Then
                    cbbCodeBackup.TooltipText = "Aktiv - letzter Backup: " & strModulename & " (" _
                        & Format(Now, "HH:NN:SS") & ")"
                End If
            End If
        End If
    Next objVBComponent

    If bolEtwasGesichert And Not objRibbon Is Nothing Then
        objRibbon.Invalidate
    End If
    Call StatusSetzen(True)
End Sub
```

Listing 9: Nur tatsächlich geänderte Module werden geschrieben

```
Private Function ModulHash(objComp As VBIDE.VBComponent) As String
    On Error GoTo Fallback
    Dim strCode As String
    Dim bytData() As Byte
    Dim objMD5 As Object
    Dim bytHash() As Byte
    Dim i As Integer
    Dim strHex As String
    Dim objStream As Object

    With objComp.CodeModule
        If .CountOfLines > 0 Then
            strCode = .Lines(1, .CountOfLines)
        End If
    End With

    Set objStream = CreateObject("ADODB.Stream")
    objStream.Open
    objStream.Charset = "utf-8"
    objStream.WriteText strCode
    objStream.Position = 0
    objStream.Type = 1
    bytData = objStream.Read
    objStream.Close

    Set objMD5 = CreateObject("System.Security.Cryptography.MD5CryptoServiceProvider")
    bytHash = objMD5.ComputeHash_2(bytData)

    For i = 0 To UBound(bytHash)
        strHex = strHex & Right("00" & Hex(bytHash(i)), 2)
    Next i

    ModulHash = strHex
    Exit Function

Fallback:
    ModulHash = CStr(Len(strCode)) & "_" & CStr(objComp.CodeModule.CountOfLines)
End Function
```

Listing 10: ModulHash bildet eine Prüfsumme über den Modulcode

```
Const AddinClassName As String = "amvCodeBackup"
Const AddinQualifiedClassName As String = _
    AddinProjectName & "." & AddinClassName
Const RootRegistryFolder As String = _
    "HKCU\SOFTWARE\Microsoft\Office\Access\Addins\" & _
    AddinQualifiedClassName & "\"
```

Der Pfad zeigt in den Zweig des aktuellen Benutzers – Abkürzung **HKCU** – in den Bereich der Access-Add-Ins. Weil wir dort und nicht im systemweiten Zweig schreiben, sind später keine Administratorrechte nötig. Das Eintragen selbst übernimmt **DllRegisterServer** (siehe Listing 11).

```
Public Function DllRegisterServer() As Boolean
    On Error GoTo RegError
    Dim wscript As Object = CreateObject("wscript.shell")
    wscript.RegWrite RootRegistryFolder & "FriendlyName", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder & "Description", AddinProjectName, "REG_SZ"
    wscript.RegWrite RootRegistryFolder & "LoadBehavior", 3, "REG_DWORD"
    Return True
RegError:
    MsgBox "DllRegisterServer -- An error occured trying to write to the system registry:" & vbCrLf & _
        Err.Description & " (" & Hex(Err.Number) & ")"
    Return False
End Function
```

Listing 11: DllRegisterServer legt die Registry-Einträge an

Die Funktion schreibt drei Werte. **FriendlyName** und **Description** erhalten den Projektnamen, und **LoadBehavior** bekommt die **3** – dieser Wert weist Access an, das Add-In gleich beim Start zu laden. Das Gegenstück **DllUnregisterServer** löscht dieselben Einträge spiegelbildlich wieder, wie Listing 12 zeigt.

Die Namen **DllRegisterServer** und **DllUnregisterServer** sind nicht frei wählbar, sondern von COM fest vorgegeben. twinBASIC ruft sie beim Registrieren und Aufheben automatisch auf; auf einem fremden Rechner übernimmt später das Windows-Werkzeug **regsvr32** dieselbe Aufgabe. Wichtig ist dabei nur, dass die Bitness von Werkzeug und DLL zusammenpasst.

Wiederherstellen gesicherter Elemente

Um Formulare oder Berichte wiederherzustellen, verwenden wir die **LoadFromText**-Methode beispielsweise wie folgt:

```
LoadFromText acform, "frmModulSpeichern", currentproject.
Path & "\BAS\Form_frmModulSpeichern_20260704_200253.bas"
```

Zum Wiederherstellen eines Moduls ziehen wir die gesicherte Datei einfach in den Objekt-Explorer des VBA-Editors.

Zusammenfassung und Ausblick

Aus überschaubaren Bausteinen ist ein nützlicher Helfer geworden. Das COM-Add-In sichert im Hintergrund in ein-

```
Public Function DllUnregisterServer() As Boolean
    On Error GoTo RegError
    Dim wscript As Object = CreateObject("wscript.shell")
    wscript.RegDelete RootRegistryFolder & "FriendlyName"
    wscript.RegDelete RootRegistryFolder & "Description"
    wscript.RegDelete RootRegistryFolder & "LoadBehavior"
    wscript.RegDelete RootRegistryFolder
    Return True
RegError:
    MsgBox "DllUnregisterServer -- An error occured trying to delete from the system registry:" & vbCrLf & _
        Err.Description & " (" & Hex(Err.Number) & ")"
    Return False
End Function
```

Listing 12: DllUnregisterServer entfernt die Einträge wieder

stellbaren Abständen alle geänderten Module in einzelne Textdateien – ohne Zutun des Entwicklers und dank des Fingerabdruck-Vergleichs nur dann, wenn sich wirklich etwas getan hat.

Mit der Einstellung des Intervalls zwischen den Sicherungen wählt man zwischen hoher Anzahl Backups und damit einer größeren Sicherheit bei Access-Abstürzen und weniger Meldungen, wenn man Änderungen an Formular- oder Berichtsmodulen durchgeführt hat.

Hier taucht, wie oben bereits beschrieben, jeweils eine Meldung auf, weil ungespeicherte Formulare nicht gesichert werden können.

Man könnte nun argumentieren, dass eine Sicherung von Formularen und Berichten nicht mehr nötig ist, wenn man diese ohnehin vor der Sicherung speichern muss.

Allerdings erhalten wir damit einen bisher noch nicht erwähnten Vorteil: Wir sichern so den Zustand aller geänderten Module in einzelnen Dateien. So können wir, wenn keine andere Versionsverwaltung im Einsatz ist, auch Zwischenstände wiederherstellen, wenn wir beim Entwickeln einmal in eine Sackgasse gelaufen sind. Bild 4 zeigt die im BAS-Ordner gesicherten Versionen verschiedener Module.

Zwei Kniffe bleiben besonders im Gedächtnis. Da ist der Umweg des Weckrufs über ein eigenes Modul, weil Windows beim Timer keine Klassenmethode direkt ansprechen kann. Und da ist der Fingerabdruck per MD5, der unnötige Sicherungen unveränderter Module verhindert. Wer möchte, baut den Helfer leicht aus – etwa um eine Obergrenze für die Zahl gespeicherter Fassungen je Modul oder um eine Schaltfläche, die den BAS-Ordner direkt öffnet.

Die gesicherten Module müssen momentan noch manuell hinzugefügt werden – man könnte noch eine Möglichkeit schaffen, diese übersichtlich darzustellen und sie zum Wiederherstellen per Mausklick einfach anzuklicken.

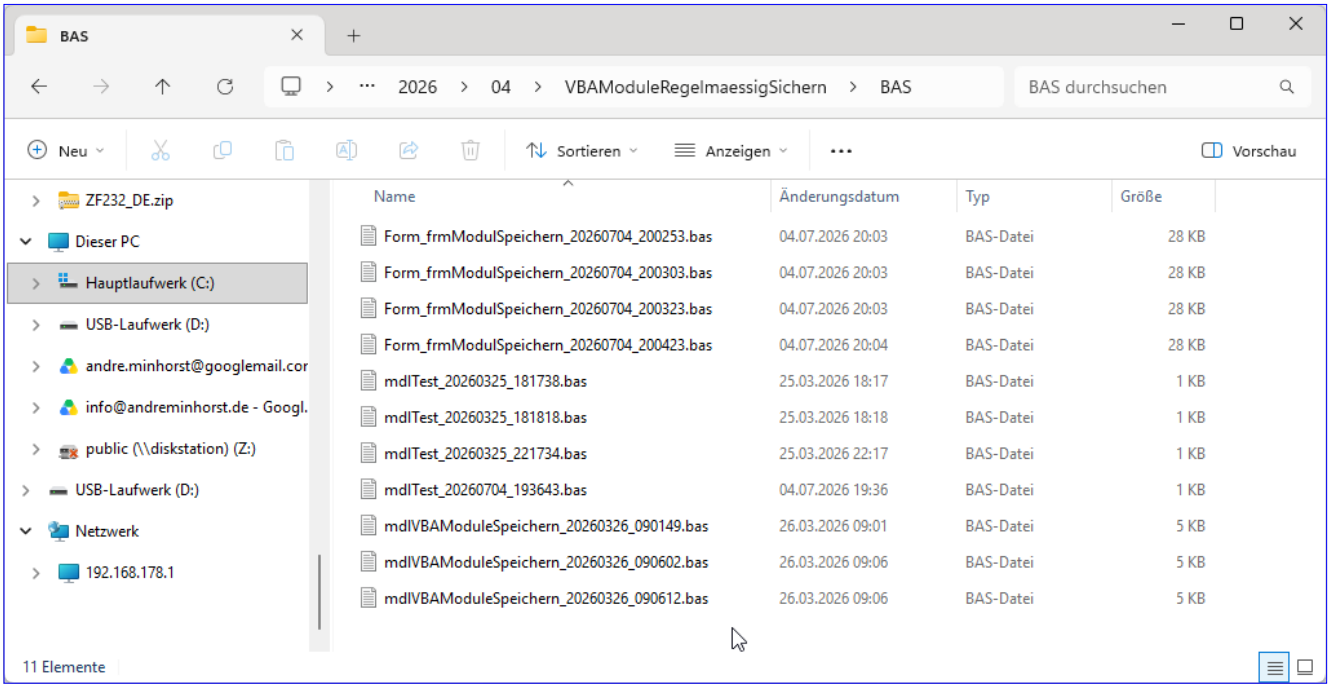

Bild 4: Gesicherte Formulare und Module im Windows Explorer