AUSGABE 06/2025

Access, SQL und Cloud AUTOMATION

MAGAZIN FÜR DIE PROGRAMMIERUNG VON MICROSOFT ACCESS, SQL SERVER UND CLOUD-AUTOMATIONEN MIT VBA UND CO.

IN DIESEM HEFT:

André Minhorst Verlag

Aus VB-Entwickler wird Access, SQL und Cloud Automation

Um der Entwicklung in der Welt rund um VBA,VB, Access und den übrigen Office-Anwendungen gerecht zu werden, haben wir den Titel dieses Magazins geändert – und damit wird sich auch der Inhalt auf neue Schwerpunkte konzentrieren.

In Access, SQL und Cloud Automation werden wir uns um alle Themen kümmern, die über das reine Programmieren von Access-Anwendungen hinausgehen. Dabei behandeln wir die folgenden Schwerpunkte:

- Access: Hier zeigen wir fortgeschrittene Techniken rund um die Entwicklung von Access-Anwendungen. VBA-Programmierung, Fehlerbehandlung, API-Programmierung – alles, was über das reine Erstellen von Tabellen, Abfragen, Formularen und Berichten hinausgeht.

- SQL Server: Wir zeigen, wie man zum SQL Server migriert, wie Access optimal mit SQL Server zusammenarbeitet und wie wir von Access aus performant auf SQL Server-Datenbanken zugreifen können.

- Cloud: Wer langfristig mit Access arbeiten möchte, muss Kompatibilität zu den vielen verschiedenen Cloud-Diensten und Saas-Lösungen herstellen und in der Lage sein, diese von Access aus zu steuern und Daten mit diesen auszutauschen. Genau darum kümmern wir uns, zum Beispiel durch Verwendung von VBA und Rest APIs.

Und natürlich automatisieren wir alles rund um Microsoft Access. Egal, ob mit reinem VBA innerhalb der Anwendung oder durch COM-DLLs, mit denen wir Funktionalitäten in Form von Bibliotheken bereitstellen oder durch COM-Add-Ins, mit denen wir die Funktionen der Benutzeroberfläche von Access oder dem VBA-Editor erweitern.

Alle aktuellen Artikel in unserer neuen Lernplattform

Außerdem findest Du ab jetzt alle Artikel in unserer neuen, modernen Lernplattform unter der folgenden Adresse:

`https://minhorst.learningsuite.io`

Dafür musst Du Dich dort allerdings erst einmal registrieren. Dein unschlagbarer Vorteil: Du brauchst Dich nur einmal zu registrieren und Deine Zugangsdaten gelten von da an für ein gesamtes Jahr.

Die Registrierung führst Du hier durch:

`https://andreminhorst.de/anmeldung-an-learningsuite`

Bitte gib dort neben Deinem Vornamen, Deinem Nachnamen und Deiner E-Mail-Adresse den Benutzernamen und das Kennwort an, dass Du auf Seite 2 dieses PDF-Dokuments findest. Danach bekommst Du eine Mail mit einem Link, über den Du die Registrierung abschließen kannst.

Du findest in der Lernplattform übrigens auch ein Forum, in dem Du Dich mit mir und anderen Lesern direkt austauschen kannst!

Achtung: vbentwickler.de bleibt erhalten!

Aus technischen Gründen findest Du die Artikel weiterhin auf der Webseite **https://www.vbentwickler.de**.

Viel Spaß beim Erkunden der neuen Lernplattform!

Dein André Minhorst

Access: Ordner und Dateien im Formular anzeigen

In Access-Anwendungen kann es interessant sein, Ordner und Dateien zu einem Datensatz verfügbar zu machen. Das bietet sich an, wann immer Dateien im Kontext eines Datensatzes in einem bestimmten Bereich im Dateisystem gespeichert sind – etwa zu Kunden, Projekten, Produkten und anderen Tabellen. In diesem Artikel zeigen wir, wie man die Verzeichnisse und Dateien eines Verzeichnisses über ein ListView-Steuerelement einfach in einem Formular anzeigen kann. Die Standardfunktionen zu diesen Elementen sollen direkt über die Einträge dieses ListView-Steuerelements verfügbar sein – zum Beispiel das Öffnen in der jeweiligen Zielanwendung, das Löschen einer Datei oder auch das Navigieren in unter- oder übergeordneten Verzeichnissen.

Wenn wir Ordner und Dateien in einem **ListView**-Steuerelement anzeigen wollen, haben wir verschiedene Möglichkeiten, diese Elemente einzulesen. Wir können diese in einer Tabellenstruktur mit Tabellen für Verzeichnisse und Dateien speichern und diese Daten in das **ListView**-Steuerelement einlesen oder einfach die Daten direkt aus dem Verzeichnis holen.

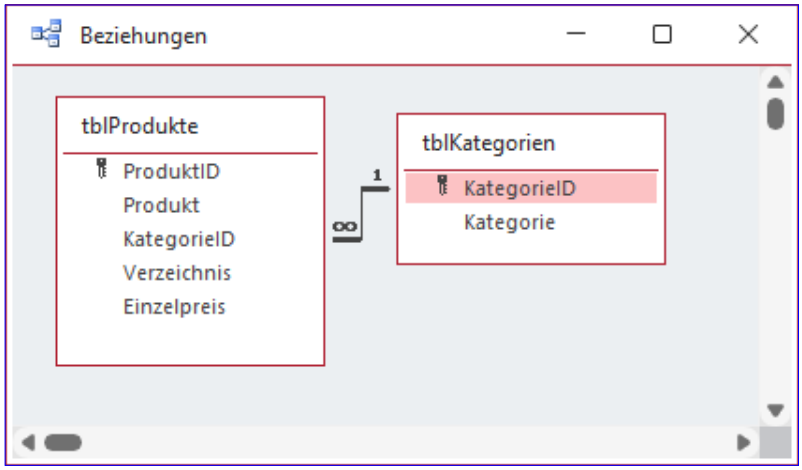

Bild 1: Tabellen der Beispielanwendung

Da wir immer nur die Dateien aus einem Verzeichnis anzeigen wollen und davon ausgehen, dass es sich dabei nicht um Hunderte Elementen handelt, gehen wir den letzteren Weg und lesen die Elemente einfach direkt aus dem Dateisystem ein.

Wir benötigen also keine Tabellen, um die Elemente zwischenzuspeichern, sondern holen diese immer beim Anzeigen eines Datensatzes. So ist auch sichergestellt, dass wir den aktuellen Dateibestand anzeigen.

Datenmodell der Anwendung

Daher fügen wir der Beispielanwendung nur die beiden Tabellen aus Bild 1 hinzu. Die Tabelle **tblProdukte** enthält als wichtigste Information das Verzeichnis, in dem sich die Dateien zum jeweiligen Produkt befinden.

Formular der Anwendung

Anschließend legen wir ein neues, leeres Formular namens **frmProdukte** in der Entwurfsansicht an.

Diesem weisen wir über die Eigenschaft Datensatzquelle die Tabelle **tblProdukte** hinzu und ziehen alle Felder dieser Tabelle aus der Feldliste in den Formularentwurf.

Die Felder ordnen wir anschließend wie in Bild 2 an. Das Textfeld **Verzeichnis** versehen wir mit dem Namen **txtVerzeichnis**, außerdem fügen wir neben diesem eine Schaltfläche zum Auswählen des Verzeichnisses hinzu.

Für das Textfeld **txtVerzeichnis** stellen wir die Eigenschaft **Horizontaler Anker** auf **Beide** ein, für die Schaltfläche **cmdOrdnerauswahl** auf **Rechts** und für das Bezeichnungsfeld des Textfeldes auf **Links**. Dadurch vergrößert sich das Textfeld, wenn wir die Breite des Formulars vergrößern.

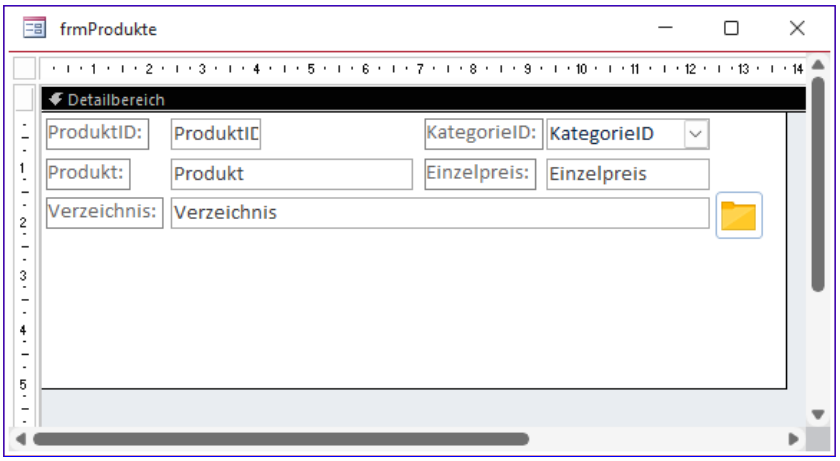

Bild 2: Das Formular **frmProdukte**

Ordner auswählen

Für die Schaltfläche hinterlegen wir die Prozedur aus Listing 1.

```
Private Sub cmdOrdnerauswahl_Click()
    Dim strInitialFolder As String
    If Not Len(Nz(Me.txtVerzeichnis, 0)) = 0 Then
        strInitialFolder = CurrentProject.Path
    Else
        strInitialFolder = Me.txtVerzeichnis
    End If
    Me.txtVerzeichnis = ChooseFolder(strInitialFolder)
End Sub
```

Listing 1: Prozedur zum Auswählen des Ordners für das aktuelle Produkt

Diese Prozedur prüft, ob bereits ein Verzeichnis im Textfeld gespeichert ist. Falls ja, wird es der Variablen **strInitialFolder** zugewiesen, anderenfalls erhält **strInitialFolder** den Pfad zur aktuellen Datenbank (**CurrentProject.Path**).

Danach ruft die Prozedur die Funktion **ChooseFolder** auf und übergibt dieser den Wert der Variablen **strInitialFolder** als Parameter.

Diese verwendet den eingebauten Office-Dialog zum Auswählen von Verzeichnissen. Dazu benötigen wir einen Verweis auf die Bibliothek **Microsoft Office 16.0 Object Library**, den wir über den **Verweise**-Dialog hinzufügen (siehe Bild 3).

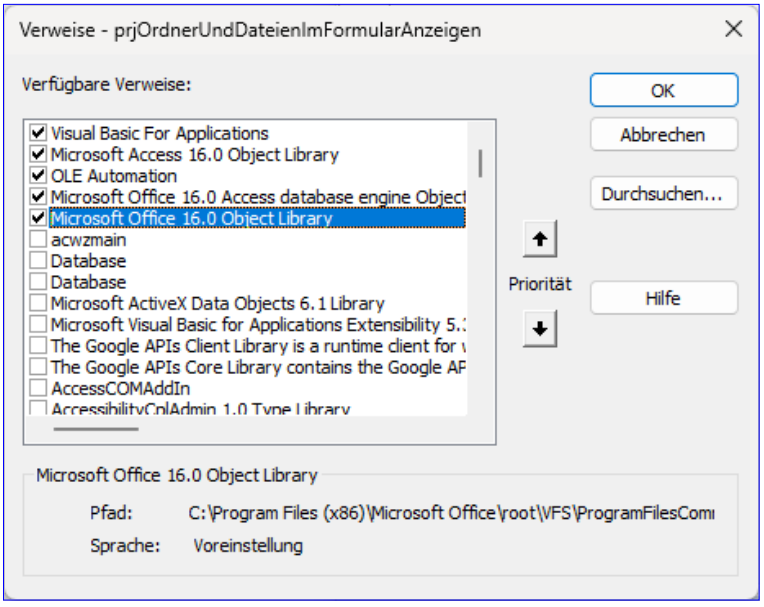

Bild 3: Verweis auf die Office-Bibliothek

Die Funktion **ChooseFolder** deklariert eine Objektvariable auf Basis der **FileDialog**-Klasse und eine Variable zum Zwischenspeichern des gewählten Verzeichnisses (siehe Listing 2).

Dann weisen wir der Variablen eine Instanz der Klasse **FileDialog** zu und übergeben da-

bei den Parameter **msoFileDialogFolderPicker**. Dann tragen wir die Werte für die Eigenschaften **Title**, **ButtonName** und **InitialFilename** ein und rufen die Methode **Show** auf, um den Dialog anzuzeigen.

```
Public Function ChooseFolder(strInitialFolder As String)
    Dim objFileDialog As Office.FileDialog
    Dim strTemp As String
    Set objFileDialog = Application.FileDialog(msoFileDialogFolderPicker)
    With objFileDialog
        .Title = "Verzeichnis auswählen"
        .ButtonName = "Auswählen"
        .InitialFilename = strInitialFolder
        If .Show = True Then
            strTemp = .SelectedItems(1)
        End If
    End With
    ChooseFolder = strTemp
End Function
```

Listing 2: Funktion zum Auswählen eines Ordners

Der Code stoppt an dieser Stelle, bis der Benutzer ein Verzeichnis ausgewählt hat, und liest dann das gewählte Verzeichnis aus. Dieses liefert die Funktion schließlich als Ergebnis zurück, sodass das gewählte Verzeichnis von der aufrufenden Prozedur in das Textfeld **txtVerzeichnis** eingetragen werden kann.

ListView zur Anzeige der Ordner und Dateien hinzufügen

Nun fügen wir unterhalb der bisher angelegten Steuerelemente ein **ListView**-Steuerelement ein. Dazu wählen wir im Ribbon den Befehl **Formularentwurf|Steuerelemente|ActiveX-Steuerelemente** aus. Im folgenden Dialog selektieren wir den Eintrag **Microsoft ListView Control, version 6.0** (siehe Bild 4), klicken auf **OK** und passen anschließend die Größe des Steuerlements so an, dass es die vollständige Formularbreite einnimmt.

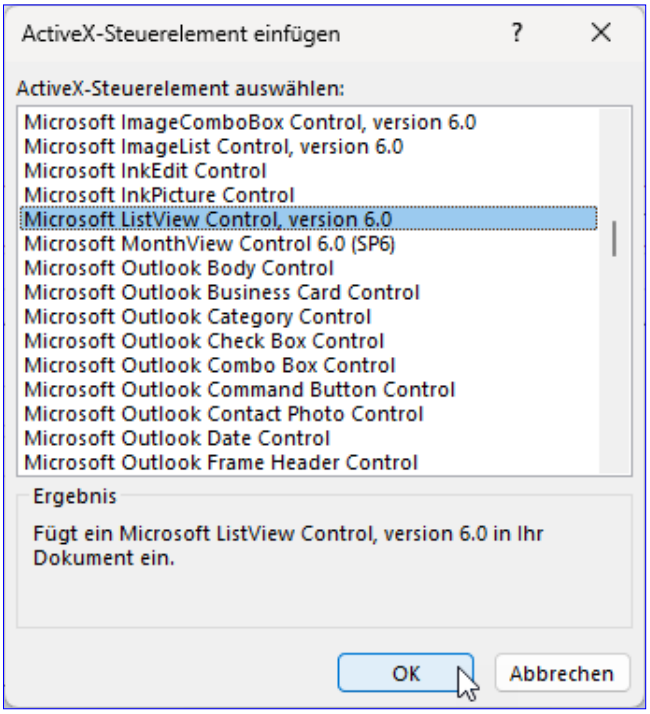

Bild 4: **ListView**-Steuerelement hinzufügen

Für das **ListView**-Steuerelement legen wir den Namen **ctlListView** fest.

ImageList zum Speichern von Icons hinzufügen

Außerdem fügen wir auf dem gleichen Weg ein **ImageList**-Steuerelement zum Formular hinzu und nennen es **ctlImageList**.

Icons einlesen

Wir wollen im **ListView**-Steuerelement zunächst zwei Spalten anzeigen. Die erste soll das Icon der Anwendung enthalten, mit der die jeweilige Datei standardmäßig geöffnet wird, die zweite den Dateinamen.

Diese Bilder müssen wir allerdings erst einmal ermitteln. Am besten wäre es, wenn diese in der Tabelle **MSysResources** landen, wo wir sie mit wenigen Codezeilen auslesen und dem **ImageList**-Steuerelement

zuweisen können. Die größere Herausforderung ist jedoch, die Bilder für die verschiedenen Dateitypen zu erhalten.

Wie das gelingt, haben wir in einem eigenen Artikel namens **VBA: Datei-Icons einlesen und speichern** (**www.vbentwickler.de/492**) beschrieben.

Das Ergebnis der dort beschriebenen Techniken ist eine Funktion namens **SaveFileIconToMSysResources**, der wir den Namen der Datei, deren Icon wir erhalten wollen, und einige weitere Informationen übergeben. Als Ergebnis landen die gewünschten Dateien wie in Bild 5 in der Tabelle **MSysResources**.

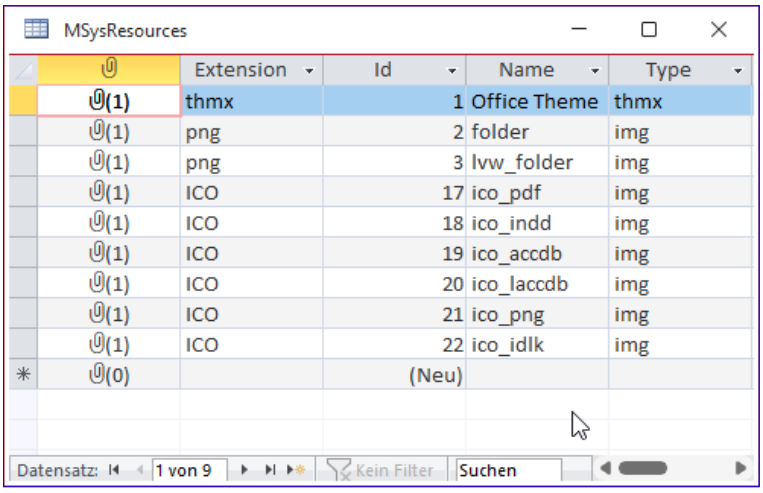

Bild 5: Die Tabelle mit den Datei-Icons

Dateien in das ListView-Steuerelement einlesen

Die Dateien sollen immer beim Anzeigen eines Datensatzes im Formular **frmProdukte** in das **ListView**-Steuerelement geladen werden.

Deshalb füllen wir die Prozedur, die durch das Ereignis **Beim Anzeigen** ausgelöst wird, mit dem Aufruf der Funktion **FillListView**, die wir im Anschluss beschreiben. Dieser Funktion übergeben wir den Pfad aus dem Feld **txtVerzeichnis**:

```
Private Sub Form_Current()
    Call FillListView(Nz(Me.txtVerzeichnis, ""))
End Sub
```

Den ersten Teil der Funktion **FillListView** finden wir in Listing 3. Nach dem Deklarationsteil referenzieren wir das **ListView**- und das **ImageList**-Steuerelement mit den Variablen **objListView** und **objImageList**. Dabei greifen wir jeweils über die **Object**-Eigenschaft auf das eigentliche ActiveX-Objekt zu, um mit allen Eigenschaften und Methoden der **MSComctl**-Steuerelemente arbeiten zu können.

Dann schalten wir das Neuzeichnen des Formulars aus, bis das **ListView**-Steuerelement vollständig gefüllt ist. Für das **ListView**-Steuerelement nehmen wir anschließend einige grundlegende Einstellungen vor.

Diese könnte man teilweise auch direkt im Eigenschaftenblatt des Steuerelements setzen. Da wir jedoch in vielen Formularen identische Einstellungen benötigen, haben wir uns angewöhnt, diese Konfiguration per VBA vorzunehmen und bei Bedarf in andere Formularmodule zu übernehmen.

Auf diese Weise behalten wir die vollständige Kontrolle über das Verhalten des **ListViews** im Code.

Bevor wir das **ListView** konfigurieren, führen wir eine wichtige Initialisierung durch: Zunächst lösen wir eine eventuell bestehende Zuordnung der **ImageList** zum **ListView**, indem wir die Eigenschaft **SmallIcons** auf **Nothing** setzen.

Anschließend leeren wir sowohl die vorhandenen Einträge im **ListView** als auch die **ListImages**-Auflistung der **ImageList**. Dieser Schritt ist entscheidend, da Änderungen an einer **ImageList** zur Laufzeit nur dann stabil funktionieren, wenn sie nicht gleichzeitig von einem **ListView**-Steuerelement verwendet wird.

VBA: Datei-Icons einlesen und speichern

Symbole und Icons spielen in Microsoft-Access-Anwendungen eine oft unterschätzte Rolle. Dabei können sie die Bedienbarkeit und Verständlichkeit einer Datenbank erheblich verbessern – insbesondere dann, wenn Dateien, Ordner oder Dokumenttypen visuell unterschieden werden sollen. Während Access für viele Steuerelemente wie TreeView, ListView, Symbolleisten oder Ribbon-Schaltflächen grundsätzlich Icon-Unterstützung bietet, stellt sich in der Praxis häufig die Frage: Wie lassen sich die echten System-Icons von Dateien – so wie sie auch im Windows-Explorer angezeigt werden – in einer Access-Anwendung verwenden? Genau das zeigen wir in diesem Artikel und liefern auch noch das Know-how, um die eingelesenen Icons direkt in der Tabelle MSysResources zu speichern. Von dort können wir sie beispielsweise in ein ImageList-Steuerelement schreiben – um sie dann in TreeView- und ListView-Steuerelementen anzuzeigen.

Mit Bordmitteln von Access ist das Ermitteln der Icon-Dateien nicht möglich. Stattdessen ist ein gezielter Zugriff auf die Windows-Shell erforderlich, um Datei-Icons dynamisch zu ermitteln und für die eigene Benutzeroberfläche nutzbar zu machen.

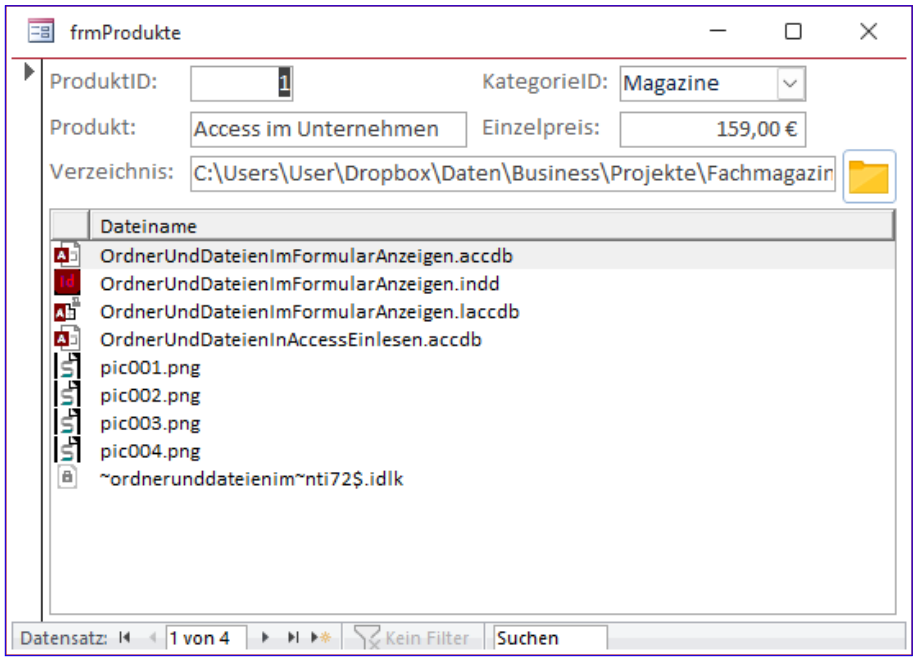

Bild 1: Dateien mit Icons in einem **ListView**-Steuerelement

Genau hier setzt die in diesem Artikel vorgestellte Technik an: Mithilfe der Windows-API lassen sich die systemweit registrierten Icons für beliebige Dateitypen auslesen, in ein **ImageList**-Steuerelement übernehmen und anschließend in verschiedenen Access-Steuerelementen einsetzen.

Der große Vorteil dieser Vorgehensweise liegt darin, dass die Icons automatisch dem jeweils installierten Programm entsprechen. PDF-Dateien, Word-Dokumente, ZIP-Archive oder ausführbare Dateien werden genauso dargestellt, wie der Benutzer sie aus dem Windows-Explorer kennt – also ohne zusätzliche Grafikdateien.

Der Artikel zeigt Schritt für Schritt, wie diese Technik umgesetzt wird, wie sich Icons performant zwischenspeichern lassen und an welchen Stellen einer Access-Datenbank sie sinnvoll eingesetzt werden können.

Die Icons können wir etwa für die Anzeige von Dateien in einem **ListView**-Steuerelement in einem Formular nutzen (siehe Bild 1), wie wir es im Artikel

```
#If VBA7 Then
    Private Declare PtrSafe Function SHGetFileInfo Lib "shell32.dll" Alias "SHGetFileInfoW" ( _
        ByVal pszPath As LongPtr, ByVal dwFileAttributes As Long, _
        psfi As SHFILEINFO, ByVal cbFileInfo As Long, _
        ByVal uFlags As Long) As LongPtr
#Else
    Private Declare Function SHGetFileInfo Lib "shell32.dll" Alias "SHGetFileInfoW" ( _
        ByVal pszPath As Long, _
        ByVal dwFileAttributes As Long, _
        psfi As SHFILEINFO, _
        ByVal cbFileInfo As Long, _
        ByVal uFlags As Long) As Long
#End If

Private Declare PtrSafe Function OleCreatePictureIndirect Lib "oleaut32.dll" ( _
    ByRef picDesc As PICTDESC, _
    ByRef RefIID As GUID, _
    ByVal fPictureOwnsHandle As Long, _
    ByRef IPic As Object) As Long

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

Private Type PICTDESC
    cbSizeofStruct As Long
    picType As Long
    hImage As LongPtr
    xExt As Long
    yExt As Long
End Type

Private Type SHFILEINFO
    hIcon As LongPtr
    iIcon As Long
    dwAttributes As Long
    szDisplayName As String * 260
    szTypeName As String * 80
End Type

Private Const PICTYPE_ICON = 3
Private Const SHGFI_ICON = &H100
Private Const SHGFI_SMALLICON = &H1
Private Const SHGFI_LARGEICON = &H0
Private Const SHGFI_USEFILEATTRIBUTES = &H10
Private Const FILE_ATTRIBUTE_NORMAL = &H80
```

Listing 1: API-Deklarationen

Access: Ordner und Dateien im Formular anzeigen (**www.vbentwickler.de/493**) vorstellen.

Vorbereitung: API-Funktionen und -Deklarationen

Für die nachfolgend vorgestellten Techniken benötigen wir als Erstes eine Reihe von API-Funktionen, Konstanten und Typen.

Diese haben wir in Listing 1 zusammengestellt.

Speichern von Icon-Dateien in der Tabelle MSysResources

Die Tabelle **MSysResources** speichert die für eine Access-Anwendung notwendigen Ressourcen wie etwa die Bilddateien, die in Formularen, Berichten und Steuerelementen wie der Schaltfläche oder dem Bild-Steuerelement angezeigt werden.

Dies ist der Platz, an dem wir auch die Icon-Dateien speichern können. Diese können wir dann beispielsweise von dort in ein **ImageList**-Steuerelement übertragen, um sie von dort aus in einem **TreeView**- oder **ListView**-Steuerelement anzuzeigen.

Funktion zum Speichern von Icon-Dateien

Die Hauptfunktion heißt **SaveFileIconToMSysResources** und erwartet drei Parameter:

- **strFilePath**: Pfad der Datei, dessen Icon wir speichern wollen
- **strResourceName**: Name, unter dem die Datei in der Tabelle **MSysResources** gespeichert werden soll
- **bolSmallIcon**: Gibt an, ob ein kleines Icon gespeichert werden soll (16 x 16 Pixel)

```
Public Sub SaveFileIconToMSysResources(ByVal strFilePath As String, ByVal strResourceName As String, _
        Optional ByVal bolSmallIcon As Boolean = True)

    Dim hIcon As LongPtr
    Dim pic As Object
    Dim strTempFile As String

    hIcon = GetFileIconHandle(strFilePath, bolSmallIcon)
    If hIcon = 0 Then Exit Sub

    Set pic = IconHandleToPicture(hIcon)
    If pic Is Nothing Then Exit Sub

    strTempFile = CurrentProject.Path & "\icon.ico"
    SavePicture pic, strTempFile

    SaveIconToMSysResources strResourceName, "ICO", strTempFile

    On Error Resume Next
    Kill strTempFile
    On Error GoTo 0
End Sub
```

Listing 2: Die Funktion **SaveFileIconToMSysResources**

VBA: Early Binding und Late Binding

Wenn wir Objektvariablen deklarieren und instanzieren wollen, gibt es zwei Varianten: Early Binding und Late Binding. Beide haben ihre Daseinsberechtigung. Bei der ersten können wir IntelliSense nutzen, benötigen allerdings einen Verweis auf die jeweilige Bibliothek. Durch das Vorhandensein des Verweises ist die Performance außerdem ein wenig besser. Beim Late Binding deklarieren wir die Variable mit dem Typ Object und weisen diese anders zu. Hier benötigen wir keinen Verweis, was wiederum Vorteile mit sich bringt. Ferner können wir kein IntelliSense nutzen. In diesem Artikel zeigen wir zuerst die Unterschiede und die Vor- und Nachteile von Early Binding und Late Binding. Zudem stellen wir eine Möglichkeit vor, beide Varianten gleichzeitig zu definieren und zur Laufzeit zwischen den Methoden zu wechseln.

In den meisten Fällen kommt man beim Programmieren mit den Elementen der standardmäßig verfügbaren Bibliotheken aus. Welche das sind, sehen wir im **Verweise**-Dialog des VBA-Projekts einer frisch angelegten Access-Datenbank (siehe Bild 1).

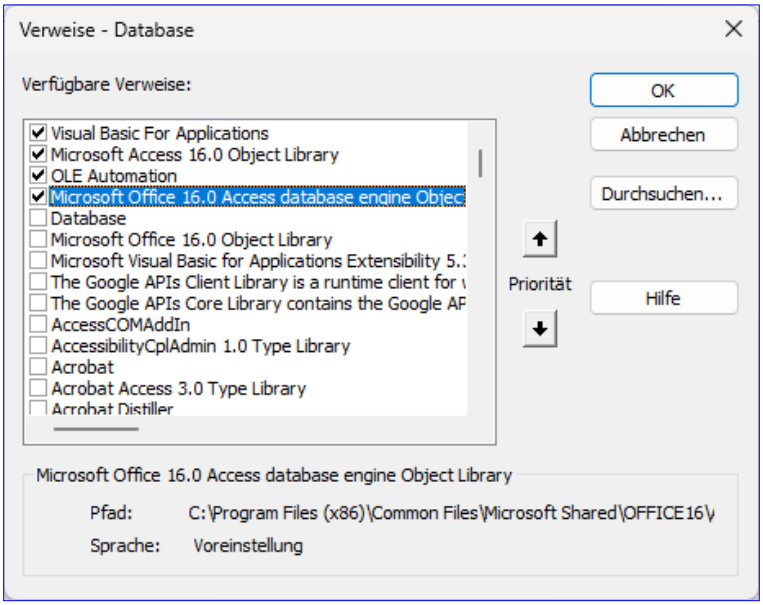

Bild 1: Standardmäßig aktivierte Verweise

Wenn wir Elemente aus weiteren Bibliotheken benötigen, fügen wir diese Bibliotheken am einfachsten zunächst über den Verweise-Dialog hinzu. Wenn wir etwa mit ADODB auf Daten zugreifen wollen, benötigen wir die Bibliothek **Microsoft ActiveX Data Objects 6.1 Library**.

Danach können wir IntelliSense nutzen, um nach Eingabe von **ADODB** und dem Punkt die enthaltenen Elemente auszuwählen (siehe Bild 2).

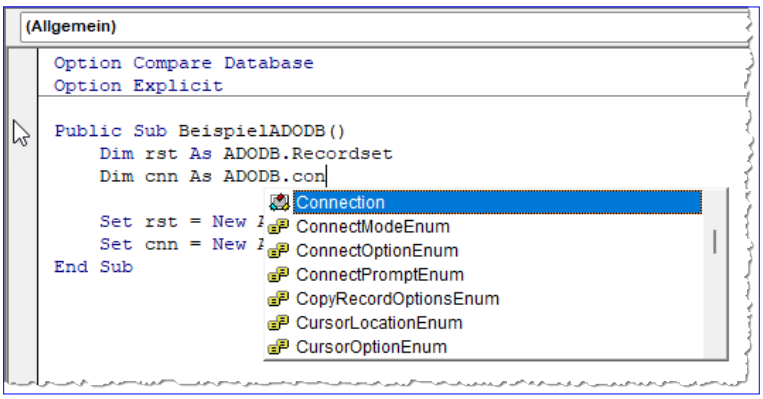

Bild 2: Deklaration per IntelliSense

Hierbei handelt es sich um das sogenannte **Early Binding**.

Wir können auch ohne einen Verweis auf die Bibliothek arbeiten. Dazu entfernen wir zunächst den Verweis. Wenn wir dann mit dem Menüeintrag **Debuggen|Kompilieren**

das Projekt kompilieren, erhalten wir einige fehlerhafte Stellen, da die deklarierten Typen nicht mehr gefunden werden können (siehe Bild 3).

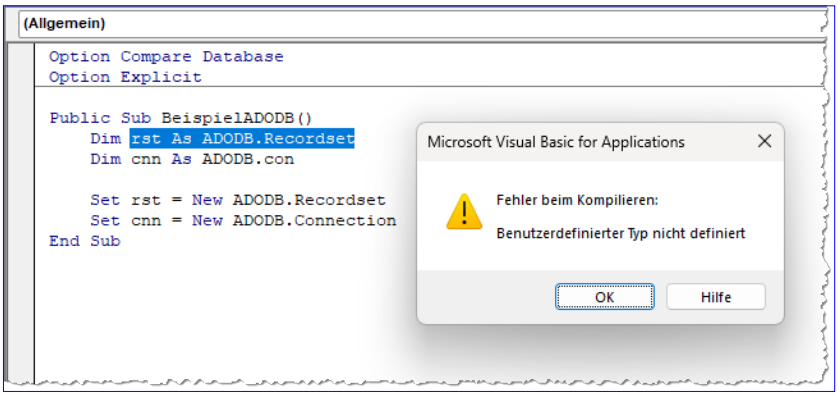

Bild 3: Fehler bei nicht auffindbaren Typen

Diese müssen wir nun zunächst durch den Typ **Object** ersetzen:

```
Dim rst As Object
Dim cnn As Object
```

Beim erneuten Kompilieren werden auch die Zeilen zur Initialisierung der Variablen mit **New** als fehlerhaft markiert.

Diese ersetzen wir durch den Aufruf der **CreateObject**-Anweisung:

```
Set rst = CreateObject("ADODB.Recordset")
Set cnn = CreateObject("ADODB.Connection")
```

Damit erhalten wir das sogenannte **Late Binding** und der Code kann nun ebenfalls kompiliert werden.

Der Nachteil hierbei ist, dass wir kein Intellisense mehr zum Programmieren mit diesen Elementen nutzen können.

Vorteil: Wir können auf nicht vorhandene Bibliotheken reagieren

Der Vorteil tritt erst zutage, wenn wir die Anwendung auf einem Rechner ausführen, auf dem die verwendete Bibliothek nicht vorhanden ist. Wenn wir eine Anwendung mit einem Verweis auf eine nicht vorhandene Bibliothek auf einem solchen Rechner öffnen, erhalten wir eine Meldung wie die aus Bild 4. In der Folge erhalten wir weitere Meldungen, mit denen der Benutzer normalerweise nicht viel anfangen kann – er wird sich dann beim Entwickler melden und damit zusätzlichen Aufwand verursachen.

Wenn wir hier mit Late Binding arbeiten, erscheint erst einmal keine solche Meldung – auch beim Kompilieren/Debuggen wird keine Fehlermeldung auftreten.

Wir können aus einer solchen Datenbank also sogar eine **.accde**-Datei erstellen.

Und es wird noch besser: Statt der nicht behandelbaren Fehlermeldung, die bei fehlerhaften Verweisen bei Early Binding auftaucht, können wir das Vorhandensein der notwendigen Bibliotheken explizit testen und

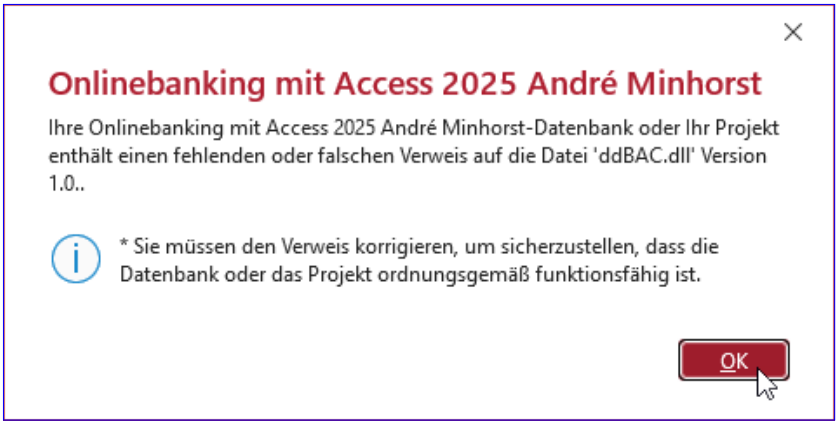

Bild 4: Meldung bei nicht vorhandener Bibliothek

den Benutzer darauf aufmerksam machen, dass diese gegebenenfalls noch installiert werden müssen.

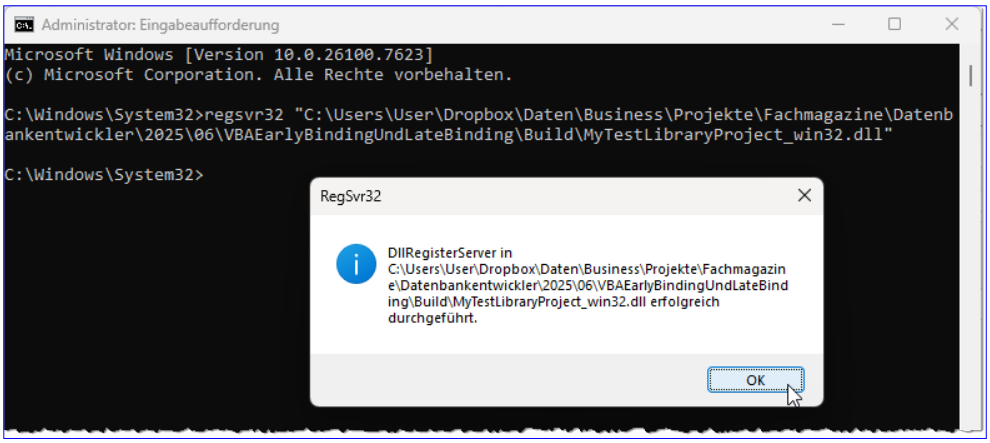

Bild 6: Registrieren der Beispiel-DLL

Angenommen, wir wollen eine selbst erstellte DLL in einem VBA-Projekt nutzen, zum Beispiel eine DLL namens **MyTestLibraryProject**, die eine Klasse namens **MyTestLibrary** zur Verfügung stellt.

Die DLL findest Du im Download zu diesem Artikel im Ordner **Build** unter den folgenden Namen:

- Für 32-Bit: **MyTestLibraryProject_win32.dll**
- Für 64-Bit: **MyTestLibraryProject_win64.dll**

Um diese zu registrieren, verwendest Du in der Eingabeaufforderung von Windows (als Administrator gestartet) den folgenden Befehl:

```
regsvr32.exe "C:\...\Build\MyTestLibraryProject_win32.dll"
```

In der Eingabeaufforderung sieht das wie in Bild 6 aus.

Diese binden wir über den **Verweise**-Dialog wie in Bild 5 in das VBA-Projekt einer Datenbank ein. Danach können wir die einzige Funktion dieser DLL wie folgt nutzen, wobei wir hier zunächst Early Binding nutzen:

```
Public Sub TestLibrary()
    Dim obj As MyTestLibraryProject.MyTestLibrary
    Set obj = New MyTestLibraryProject.MyTestLibrary
    Debug.Print obj.MultiplyByTen(10)
End Sub
```

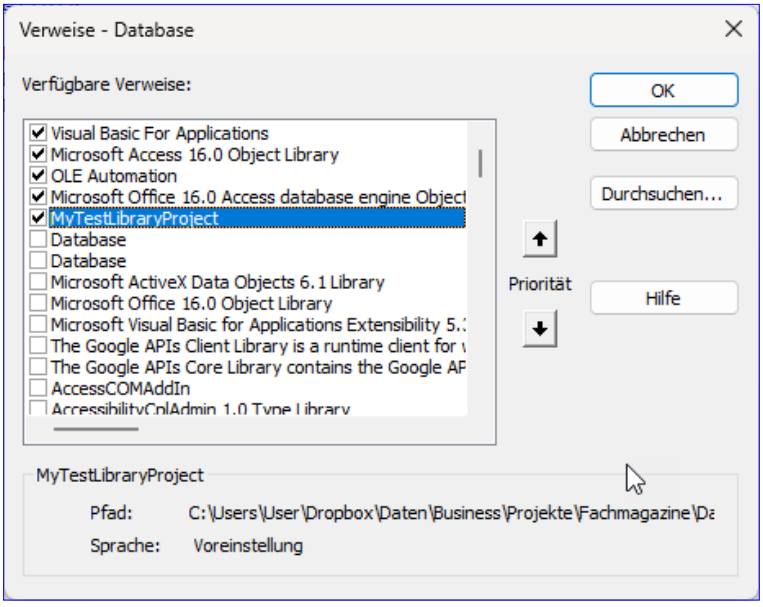

Bild 5: Einbinden einer Beispiel-DLL

Der Aufruf der Prozedur liefert das gewünschte Ergebnis, in diesem Fall **100**.

Registrierung der DLL aufheben

Nun schauen wir uns den Fall an, dass die DLL nicht wie erwartet registriert ist. Dazu heben wir die Regist-

VBA: Bedingte Kompilierung nutzen

In VBA-Projekten kann es vorkommen, dass Anweisungen nur in bestimmten Situationen kompiliert werden sollen. Das bekannteste Beispiel sind die Deklarationen von API-Funktionen, die je nach VBA-Version mal in der 32-Bit- und mal in der 64-Bit-Variante bereitgestellt werden sollen. Da die 64-Bit-Version bei Verwendung von 32-Bit-Access unter Umständen Datentypen mit sich bringt, die es in der 32-Bit-Version nicht gibt, würde dies beim Kompilieren zu Fehlern führen. Daher gibt es die sogenannte bedingte Kompilierung, bei der man mit speziellen If...Then-Bedingungen dafür sorgen kann, dass nur die für die aktuelle Version relevanten Codezeilen kompiliert werden können. In diesem Artikel zeigen wir, wie die bedingte Kompilierung funktioniert. Außerdem stellen wir ein weiteres Beispiel vor, in dem wir entweder die Early Binding- oder die Late Binding-Verwendung von Variablen nutzen wollen – abhängig von einer zur Laufzeit gesetzten Bedingung.

Bedingte Kompilierung

Die bedingte Kompilierung arbeitet mit **If...Then**-Bedingungen, die mit einem vorangestellten Raute-Zeichen angelegt werden:

```
#If VBA7 Then
'API-Deklarationen für VBA7
#Else
'API-Deklarationen für ältere VBA-Version
#End If
```

Hier haben wir bereits die erste von einigen wenigen eingebauten Kompilierungskonstanten verwendet, nämlich **VBA7**. Diese hat den Wert **True**, wenn VBA in der Version 7 verwendet wird. Den Wert dieser Konstanten können wir nur in einer mit dem Raute-Zeichen beginnenden Zeile auslesen, sie kann nicht einfach mit Debug.Print ermittelt werden. Die folgende Anweisung liefert kein Ergebnis:

```
Debug.Print VBA7
```

Wir können aber eine Prozedur schreiben, in der wir per **#If...#Then**-Bedingung prüfen, ob **VBA7** den Wert **True** oder **False** hat:

```
Public Sub IsVBA7()
#If VBA7 Then
    Debug.Print "VBA7"
#Else
    Debug.Print "Älteres VBA"
#End If
End Sub
```

Dies liefert für aktuelle Office-Versionen (ab Version 2010) den Wert **VBA7**.

32-Bit oder 64-Bit?

Auf die gleiche Weise können wir herausfinden, ob die aktuelle Office-Version in der 32-Bit- oder in der 64-Bit-Version vorliegt.

Hier verwenden wir die Kompilierungskonstante **Win64**:

```
Public Sub Is32Or64Bit()
#If Win64 Then
    Debug.Print "64-Bit"
#Else
    Debug.Print "32-Bit"
#End If
End Sub
```

Wir können auch explizit auf die 32-Bit-Version prüfen:

```
#If Win32 Then
    Debug.Print "32-Bit"
#Else
    Debug.Print "64-Bit"
#End If
```

Bedingte Kompilierung mit benutzerdefinierten Konstanten

Wir können auch eigene Kompilierungskonstanten definieren und diese per **#If...#Then**-Bedingung abfragen.

Diese Konstanten müssen ohne Datentyp angegeben werden und werden ebenfalls mit führenden Raute-Zeichen definiert, zum Beispiel:

```
#Const cEarlyBinding = -1
```

Die Konstanten dürfen außerdem nur **Long**-Werte enthalten.

Diese fragen wir dann wie folgt ab:

```
Public Sub EigeneKonstante()
#If cEarlyBinding = -1 Then
    Debug.Print "cEarlyBinding ist True"
#Else
    Debug.Print "cEarlyBinding ist False"
#End If
End Sub
```

Bedingte Kompilierung für Early Binding und Late Binding

Wenn wir auf dem Entwicklungsrechner mit Early Binding arbeiten wollen, um IntelliSense nutzen zu können, aber auf dem Produktivrechner sicherstellen wollen, dass das Projekt auch ohne Vorhandensein der jeweiligen Bibliothek zumindest ohne Kompilierfehler verwendet werden kann, können wir hier die Anweisungen zum Deklarieren und Initialisieren von Objektvariablen einfügen:

```
Public Sub EarlyBinding()
#If cEarlyBinding = -1 Then
    Dim rst As adodb.Recordset
    Set rst = New adodb.Recordset
#Else
    Dim rst As Object
    Set rst = CreateObject("ADODB.Recordset")
#End If
End Sub
```

Wenn wir **#cEarlyBinding** auf **-1** einstellen, werden die Anweisungen im **#If**-Teil der Bedingung kompiliert und ausgeführt, andernfalls die aus dem **#Else**-Teil. Das können wir leicht prüfen, indem wir die Anweisungen schrittweise durchlaufen.

Wenn wir die Konstante **#cEarlyBinding** auf den Wert **-1** einstellen und die Bibliothek **Microsoft ActiveX Data Objects x.y** nicht per Verweis eingebunden ist, erhalten wir außerdem einen Kompilierfehler (siehe Bild 1). Hier müssen wir also, solange wir auf dem

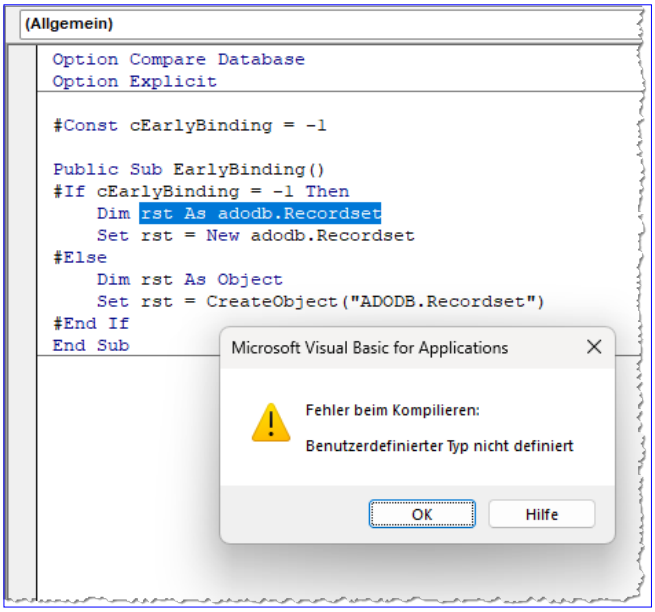

Bild 1: Kompilierfehler, weil die ADODB-Bibliothek fehlt

Per VBA von Early Binding zu Late Binding wechseln

Im Artikel »VBA: Early Binding und Late Binding« (www.vbentwickler.de/494) haben wir die beiden Methoden Early Binding und Late Binding vorgestellt und ihre Vor- und Nachteile beschrieben. Im vorliegenden Artikel zeigen wir nun eine automatische Lösung, um schnell einige oder alle per Early Binding definierten Elemente nach Late Binding zu migrieren. Dazu nutzen wir Code, der zunächst alle Early Binding-Elemente ermittelt, diese in einem Formular anzeigt und es dann ermöglicht, diese in Late Binding-Elemente umzuwandeln.

Die Lösung aus diesem Artikel soll es ermöglichen, alle Module der aktuellen Datenbank aus einem Listenfeld auszuwählen und alle dort in Deklarationszeilen vorhandenen Klassen, Typen und Enumerationen einzulesen.

Wir benötigen zwar die Typen und Enumerationen nicht, da wir diese nicht nach Late Binding migrieren können, aber aus technischen Gründen können wir diese nicht ohne erheblichen Aufwand aus der Ermittlung ausschließen.

Das Formular zur Steuerung dieses Vorgangs sehen wir in Bild 1. Hier haben wir das Modul **mdlTest** ausgewählt und anschließend auf die Schaltfläche **Typen einlesen** geklickt, um alle dort enthaltenen Typen zu ermitteln und in einem weiteren Listenfeld anzuzeigen.

Wenn wir nun einen oder mehrere dieser Einträge markieren und auf die Schaltfläche Early Binding ersetzen klicken, werden die als Early Binding deklarierten Elemente samt der zur Initialisierung verwendeten Anweisungen in Late Binding umgewandelt – und zwar in allen Modulen, die im oberen Listenfeld markiert sind.

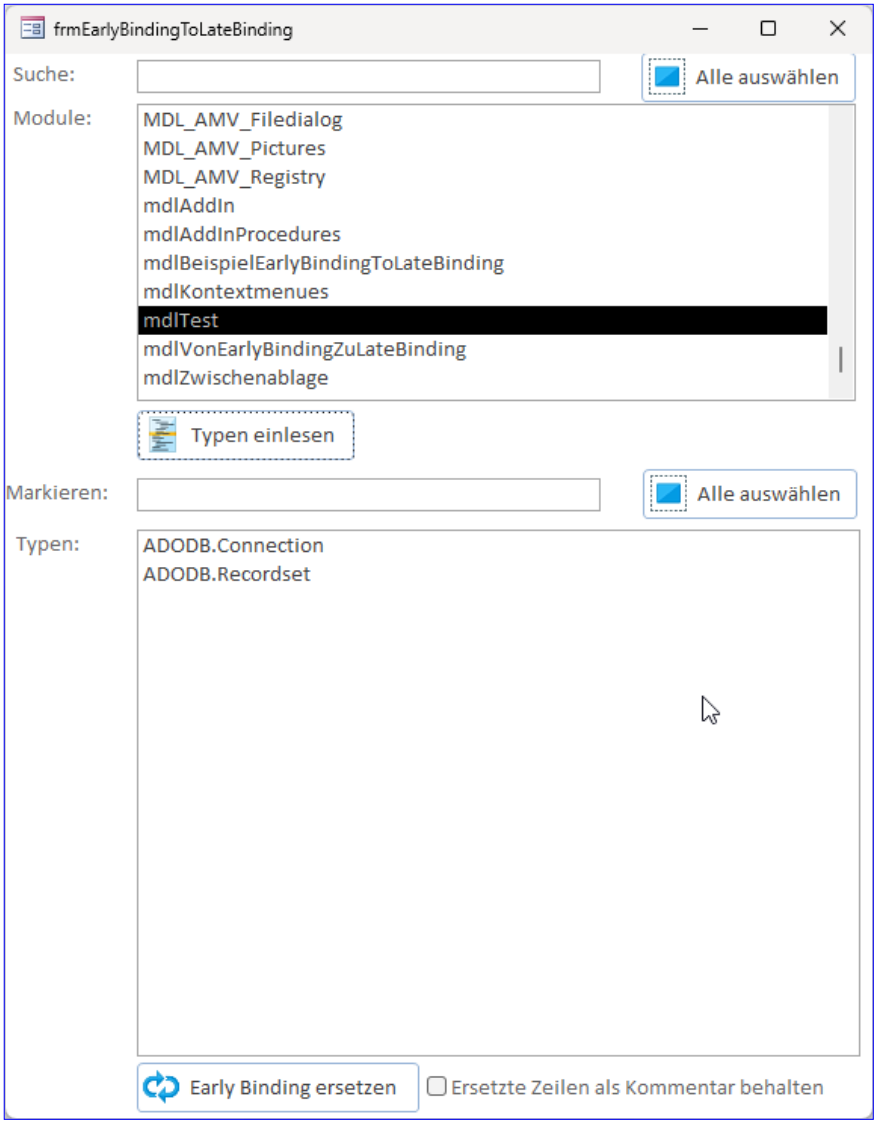

Bild 1: Formular zum Steuern der Migration nach Late Binding

Zusätzlich finden wir dort noch ein Kontrollkästchen namens **Ersetzte Zeilen als Kommentar behalten**. Damit können wir festlegen, dass die Early Binding-Anweisungen nicht gelöscht, sondern lediglich auskommentiert werden.

Im Beispielmodul **mdlTest** haben wir einige mit Early Binding versehene Anweisungen untergebracht – und zwar in den unterschiedlichsten Ausprägungen:

- als einfache Deklarationszeilen,
- als Parameter von Prozeduren,
- als Deklaration in Prozeduren,
- als Rückgabewert von Prozeduren und
- mit und ohne Zeilenumbrüche (siehe Bild 2).

```
(Allgemein)
Option Compare Database
Option Explicit

Public rst1 As ADODB.Recordset
Dim rst2 As ADODB.Recordset

Public Sub TestParameter(rst As ADODB.Recordset)

End Sub

Public Sub TestParameterOptional(Optional rst As ADODB.Recordset)

End Sub

Public Sub TestParameterByVal(ByVal rst As ADODB.Recordset)

End Sub

Public Sub TestParameterByRef(ByRef rst As ADODB.Recordset)

End Sub

Public Sub TestParammeterOptionalDefault(Optional rst As ADODB.Recordset = Nothing)

End Sub
```

Bild 2: Beispielanweisungen mit Early Binding

```
(Allgemein)
Option Compare Database
Option Explicit

'Public rst1 As ADODB.Recordset
Public rst1 As Object
'Dim rst2 As ADODB.Recordset
Dim rst2 As Object

'Public Sub TestParameter(rst As ADODB.Recordset)
Public Sub TestParameter(rst As Object)

End Sub

'Public Sub TestParameterOptional(Optional rst As ADODB.Recordset)
Public Sub TestParameterOptional(Optional rst As Object)

End Sub

'Public Sub TestParameterByVal(ByVal rst As ADODB.Recordset)
Public Sub TestParameterByVal(ByVal rst As Object)

End Sub

'Public Sub TestParameterByRef(ByRef rst As ADODB.Recordset)
Public Sub TestParameterByRef(ByRef rst As Object)

End Sub

'Public Sub TestParammeterOptionalDefault(Optional rst As ADODB.Recordset = Nothing)
Public Sub TestParammeterOptionalDefault(Optional rst As Object = Nothing)

End Sub
```

Bild 3: Beispielanweisungen mit Late Binding und auskommentierter Early Binding-Version

Nachdem wir die beiden Typen **ADODB.Connection** und **ADODB.Recordset** mit unserem Formular umgestellt haben, sieht der abgebildete Ausschnitt des Moduls wie in Bild 3 aus.

Beschreibung des Formulars

Im Formular finden wir in der Entwurfsansicht die folgenden Steuerelemente (siehe Bild 4):

- **txtSucheModule**: Erlaubt das schnelle Filtern der im Listenfeld **lstModule** angezeigten Module.

- Schaltfläche **cmdAlleAuswahlen**: Markiert alle Einträge im Listenfeld **lstModules**.

- Listenfeld **lstModules**: Zeigt die gefundenen Module an, die den jeweiligen Typ enthalten.

- Schaltfläche **cmdTypenEinlesen**: Liest alle Typen der markierten Module ein und zeigt diese im Listenfeld **lstTypes** an.

- Textfeld **txtSucheTypen**: Filtert das Listenfeld **lstTypes** nach dem eingegebenen Suchbegriff.

- Schaltfläche **cmdAlleTypenAuswaehlen**: Markiert alle Einträge des Listenfeldes **lstTypes**.

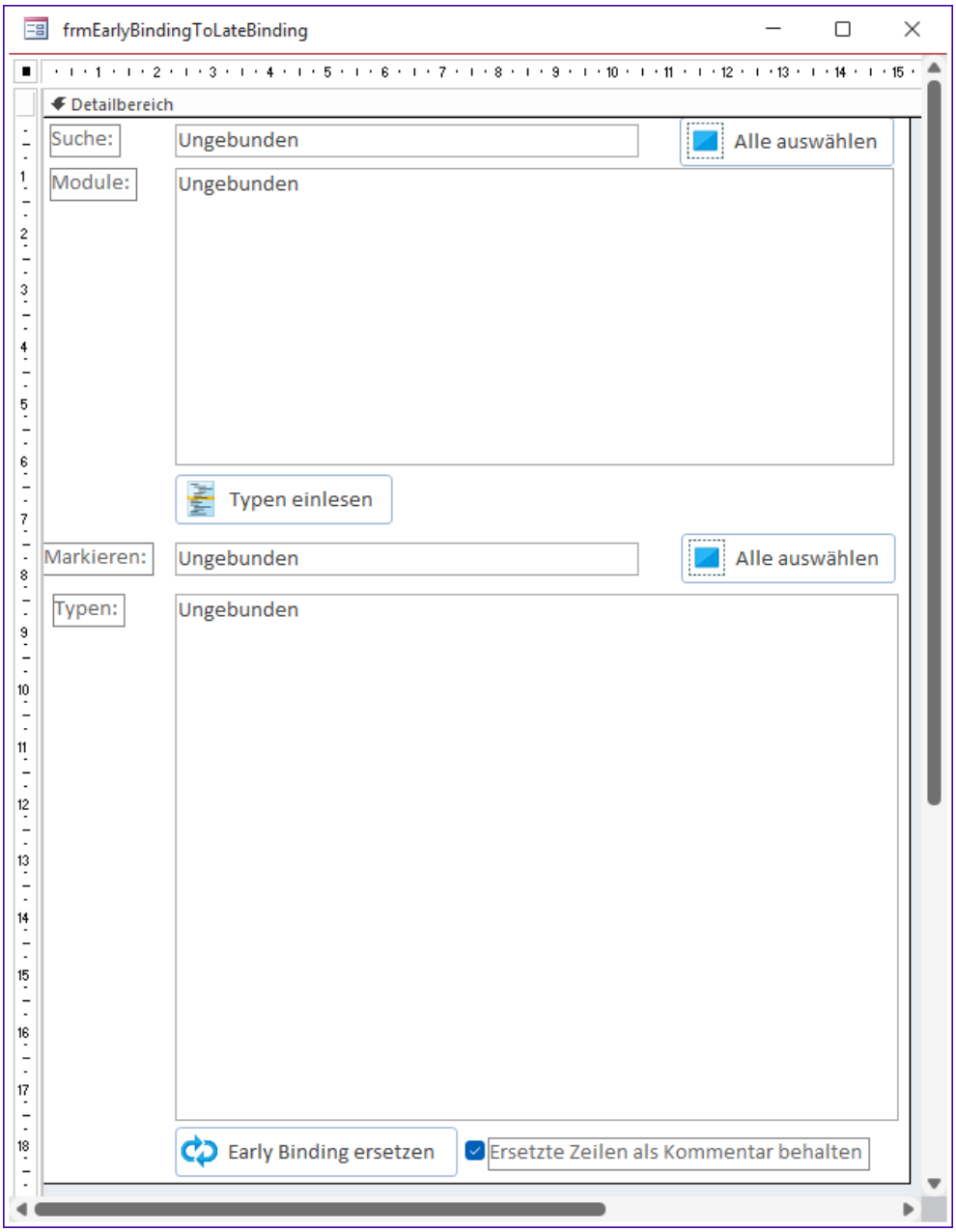

Bild 4: Das Formular **frmEarlyBindingToLateBinding** in der Entwurfsansicht

- Listenfeld **lstTypes**: Zeigt alle gefundenen Typen in den markierten Modulen an.

- Schaltfläche **cmdEarlyBindingErsetzen**: Ersetzt für alle markierten Typen in den markierten Modulen Early Binding durch Late Binding.

- Kontrollkästchen **chkErsetzeZeilenAlsKommentarBehalten**: Gibt an, ob die ersetzten Zeilen auskommentiert oder einfach ersetzt werden sollen.

Ereignis beim Laden des Formulars

Beim Laden des Formulars wird das Ereignis aus Listing 1 ausgelöst. Es referenziert die aktuelle Datenbank

```
Private Sub Form_Load()
    Dim db As DAO.Database
    Dim objVBProject As VBIDE.VBProject
    Dim objVBComponent As VBIDE.VBComponent

    Set db = CodeDb
    db.Execute "DELETE * FROM tblModules", dbFailOnError
    db.Execute "DELETE * FROM tblTypes", dbFailOnError

    Set objVBProject = CurrentVBProject
    For Each objVBComponent In objVBProject.VBComponents
        db.Execute "INSERT INTO tblModules(Modul) VALUES('" & objVBComponent.name & "')", dbFailOnError
    Next objVBComponent

    Me.lstModules.Requery
    Me.lstTypes.Requery

    If Not IsNull(Me.OpenArgs) Then
        Dim i As Integer
        Dim strVBComponent As String
        strVBComponent = Me.OpenArgs
        For i = 0 To Me.lstModules.ListCount - 1
            If Me.lstModules.Column(1, i) = strVBComponent Then
                Me.lstModules.Selected(i) = True
                Call cmdTypenEinlesen_Click
            End If
        Next i
    End If
End Sub
```

Listing 1: Ereignisprozedur, die beim Laden des Formulars ausgelöst wird

mit der **CodeDb**-Funktion (dies ist eine Vorbereitung, um die Lösung als Add-In zu nutzen). Dann löscht es die beiden Tabellen **tblModules** und **tblTypes**, in denen wir die ermittelten Daten speichern, um sie in den Listenfeldern anzuzeigen. **tblModules** enthält das Primärschlüsselfeld **ModulID** und das Textfeld **Modul**. Die Tabelle **tblTypes** enthält die beiden Felder **TypeID** und **Type** sowie **Line** und **LineNumber**, um jeweils eine Zeile zu speichern, in der dieser Typ auftritt (diese wurden eher zu Testzwecken während der Programmierung der Lösung genutzt).

Danach holen wir mit der Funktion **CurrentVBProject** einen Verweis auf das VBA-Projekt der aktuellen Datenbank. Dies ist notwendig, da beim Vorhandensein von Access-Add-Ins oder eingebundenen Bibliotheksdatenbanken sonst gegebenenfalls das falsche VBA-Projekt verwendet wird. Die Funktion **CurrentVBProject** sieht wie folgt aus:

```
Public Function CurrentVBProject() As VBIDE.VBProject
    Dim objVBProject As VBIDE.VBProject
    For Each objVBProject In VBE.VBProjects
        If objVBProject.FileName = CurrentDb.Name Then
            Set CurrentVBProject = objVBProject
            Exit Function
        End If
    Next objVBProject
```

```
End Function
```

Sie durchläuft alle vorhandenen VB-Projekte und prüft, ob der Pfad dem Pfad der aktuell geöffneten Access-Datenbank entspricht. Ist das der Fall, wird der Verweis auf dieses VB-Projekt zurückgegeben.

Für die Verwendung dieser und anderer nachfolgend genutzter Elemente, die auf den VBA-Editor und seine Module zugreifen, fügen wir dem VBA-Projekt einen Verweis auf die Bibliothek **Microsoft Visual Basic for Applications Extensibility 5.3 Object Library** hinzu.

Die Prozedur **Form_Load** durchläuft nun alle Elemente der Auflistung **VBComponents**, was den Modulen entspricht. Für jedes Modul wird ein Eintrag in der Tabelle **tblModules** angelegt. Danach werden die beiden Listenfelder **lstModules** und **lstTypes** aktualisiert, damit sie den aktuellen Inhalt der beiden Tabellen **tblModules** und **tblTypes** anzeigen.

Schließlich haben wir noch den Fall vorbereitet, dass das Formular direkt für ein bestimmtes Modul aufgerufen wird. Dann würden wir das Argument OpenArgs mit dem Namen des gewünschten Moduls füllen. Dies wird dann direkt im Listenfeld lstModules markiert.

Der Aufruf des Formulars für diesen Fall sieht wie folgt aus:

```
DoCmd.OpenForm "frmEarlyBindingToLateBinding", _
    OpenArgs:="mdlTest"
```

Auswählen aller Module

Um alle Module auszuwählen, klicken wir auf die Schaltfläche **cmdAlleTypenAuswaehlen**. Diese durchläuft alle Elemente des Listenfeldes und stellt die Eigenschaft **Selected** für den jeweiligen Index auf **True** ein:

```
Private Sub cmdAlleTypenAuswaehlen_Click()
    Dim lngItem As Long
    For lngItem = 0 To Me.lstTypes.ListCount - 1
        Me.lstTypes.Selected(lngItem) = True
    Next lngItem
End Sub
```

Einlesen der Typen der markierten Module

Ein Klick auf die Schaltfläche **cmdTypenEinlesen** soll alle Typen der markierten Module ermitteln und in die Tabelle **tblTypes** schreiben.

Dazu referenziert sie wieder die Datenbank mit dem Formular und leert die Tabelle **tblTypes**. Dann prüft sie, ob überhaupt Einträge im Listenfeld **lstModules** markiert sind, und weist darauf hin, falls das nicht der Fall ist.

Danach durchläuft sie alle markierten Einträge des Listenfeldes **lstModules** über die **ItemsSelected**-Auflistung und ermittelt mit **Column(1, var)** den Namen des jeweiligen Moduls. Innerhalb der Schleife ruft sie für jedes dieser Elemente die Prozedur **TypesToTable** auf und übergibt dieser den Namen des Moduls. Schließlich aktualisiert sie die Liste der Typen.

```
Private Sub cmdTypenEinlesen_Click()
    Dim var As Variant
    Dim strModule As String
    Dim db As DAO.Database

    Set db = CodeDb
    DoCmd.Hourglass True
    db.Execute "DELETE * FROM tblTypes", dbFailOnError

    If Me.lstModules.ItemsSelected.Count = 0 Then
        MsgBox "Markiere die zu untersuchenden Module.", _
            vbOKOnly + vbExclamation, "Kein Modul markiert"
        Exit Sub
    End If

    For Each var In Me.lstModules.ItemsSelected
```

```
        strModule = Me.lstModules.Column(1, var)

        Call TypesToTable(strModule)
    Next var
    Me.lstTypes.Requery
    DoCmd.Hourglass False
End Sub
```

Auslesen der Typen eines Moduls

Die Prozedur **TypesToTable** referenziert wieder die Code-Datenbank und das aktuelle VBA-Projekt (siehe Listing 2).

Danach füllt sie die Variable **objVBComponent** mit einem Verweis auf das übergebene Modul und holt einen weiteren Verweis auf das enthaltene **CodeModule**-Objekt, das in **objCodeModule** landet.

Dann durchläuft sie in einer **For...Next**-Schleife alle Codezeilen, wobei die letzte Zeile mit der Eigenschaft **CountOfLines** ermittelt wird.

Hier speichert sie die Originalzeile in **strLineOriginal** und in **strLine** die um führende und folgende Leerzeichen bereinigte Version der Zeile.

Für diese ruft sie nun die Funktion **KommentarAbschneiden** auf, die wir ebenfalls im Modul finden und die alle am Ende der Zeile befindlichen Kommentare aus **strLine** entfernt.

Eine weitere Funktion namens **TextAusLiteralenEntfernen** leert eventuell in Anführungszeichen vorhandene Texte. Aus der Zeile **strText = "Beispieltext"** wird dann zum Beispiel **strText = ""**.

```
Public Sub TypesToTable(strModule As String)
    Dim db As DAO.Database
    Dim objVBProject As VBIDE.VBProject
    Dim objVBComponent As VBIDE.VBComponent
    Dim objCodeModule As VBIDE.CodeModule
    Dim strLine As String
    Dim strProc As String
    Dim lngProcType As Long
    Dim lngProcBodyLine As Long
    Dim strLineOriginal As String
    Dim lngLine As Long
    Dim lngProcline As Long

    Set db = CodeDb

    Set objVBProject = CurrentVBProject
    Set objVBComponent = objVBProject.VBComponents(strModule)
    Set objCodeModule = objVBComponent.CodeModule
    For lngLine = 1 To objCodeModule.CountOfLines
        strLineOriginal = objCodeModule.Lines(lngLine, 1)
        strLine = Trim(strLineOriginal)
        strLine = KommentarAbschneiden(strLine)
        strLine = TextAusLiteralenEntfernen(strLine)
        ...
```

Listing 2: Einlesen der Typen (Teil 1)

SQL Server-Datenbanken vergleichen mit VS.Code

Es gibt verschiedene Gründe, zwei Datenbanken miteinander zu vergleichen. Wenn uns zwei Versionen einer Datenbank vorliegen, unterscheiden diese sich gegebenenfalls und wir möchten herausfinden, welche die aktuellere ist. Bei der Gelegenheit kann man auch gleich noch prüfen, welche Unterschiede zwischen den Datenbanken es überhaupt gibt. Damit eröffnen sich praktische Möglichkeiten: So können wir etwa ein Skript erstellen lassen, welches die Unterschiede zwischen zwei Versionen einer Datenbank aufzeigt. Damit erkennen wir nicht nur die Unterschiede selbst, sondern können das Skript sogar nutzen, um die ältere der beiden Datenbanken auf den aktuellen Stand bringen. Das ist hilfreich, wenn wir eine beim Kunden befindliche Datenbank aktualisieren wollen. Wir erstellen dann einfach ein Skript mit den Unterschieden und führen es beim Kunden aus, damit er die aktuellste Version der Datenbank erhält. Für das Ermitteln der Unterschiede gibt es verschiedene Werkzeuge, etwa die SQL Server-Tools für Visual Studio Code, die leichtgewichtige Entwicklungsumgebung von Microsoft. In diesem Artikel zeigen wir, wie dieses installiert wird und wie wir die SQL Server-Tools aktivieren und nutzen, um die Unterschiede zwischen zwei Datenbanken in einem Skript zusammenzustellen.

Visual Studio Code herunterladen und installieren

Als Erstes benötigen wir **Visual Studio Code** auf unserem Rechner. Dieses laden wir von der folgenden Webseite herunter:

```
https://code.visualstudio.com/
```

Nach dem Download können wir **Visual Studio Code** direkt installieren. Hier gibt es kaum Optionen – wir können lediglich noch die Einträge zum Öffnen von

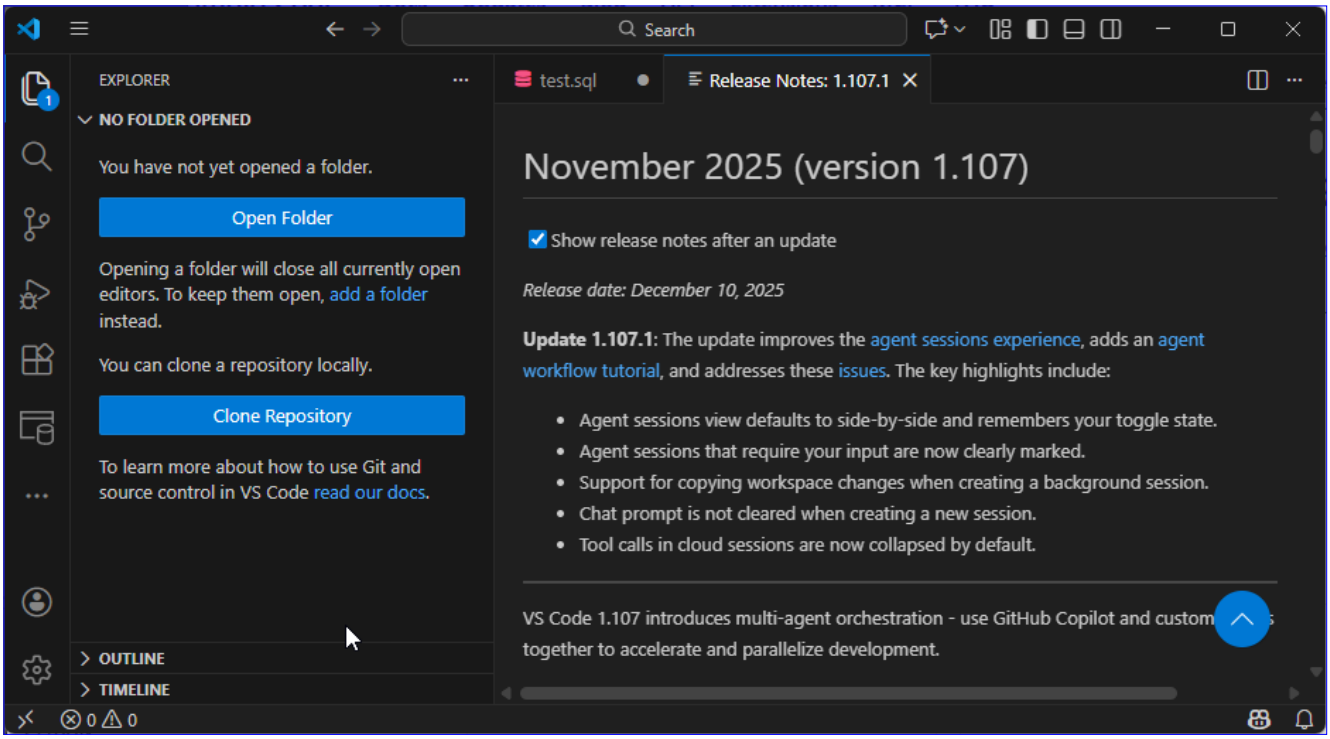

Bild 1: Visual Studio Code direkt nach dem Start

Dateien mit **Visual Studio Code** für die verschiedenen Kontextmenüs von Windows aktivieren.

Nach der Installation und dem Start präsentiert sich **Visual Studio Code** wie in Bild 1.

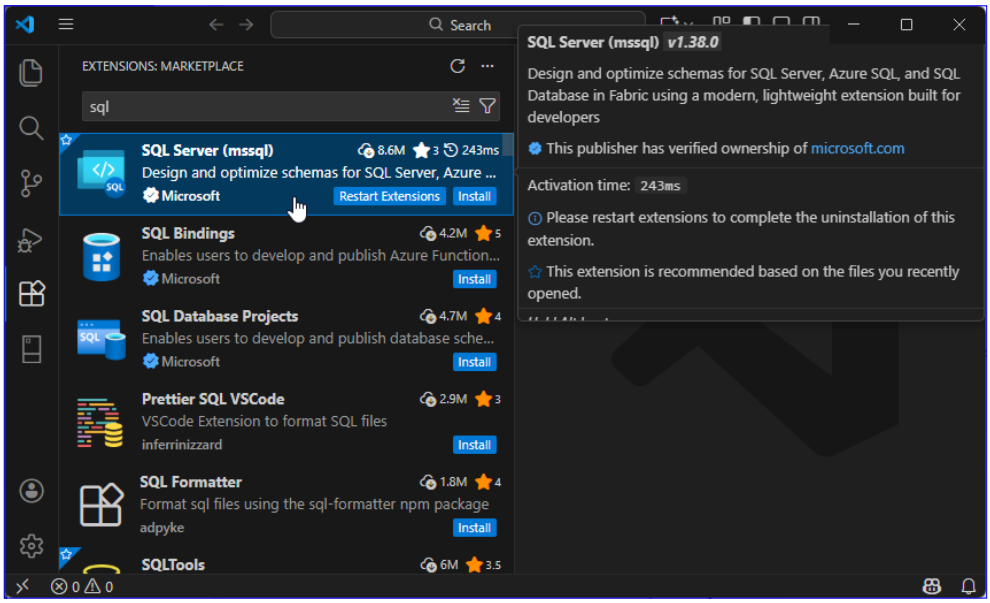

Bild 2: Installieren der SQL Server-Erweiterung

MSSQL Extension installieren

Als Nächstes benötigen wir die Erweiterung für die Arbeit mit SQL Server-Datenbanken. Dazu aktivieren wir mit **Strg + Umschalt + X** die Anzeige der Extensions. Hier geben wir SQL als Suchbegriff ein und erhalten unter anderem den Eintrag **SQL Server (mssql)** – siehe Bild 2.

Ein Klick auf die Schaltfläche **Install** fügt diese Extension zu **Visual Studio Code** hinzu.

Verbindung zu den zu vergleichenden Datenbanken herstellen

Nun benötigen wir zwei Verbindungen, jeweils eine für die beiden zu vergleichenden Datenbanken. Dazu betätigen wir die Tastenkombination **Strg + Umschalt + P** und wählen den Eintrag **MS SQL: Add Connection** aus (siehe Bild 3).

Es erscheint ein Dialog, in dem wir die Verbindungsdaten eingeben – den Namen des Servers, die Angabe, ob wir dem Server-Zertifikat vertrauen wollen, die Authentifizierungsmethode und bei SQL Server-Authentifizierung die Benutzerdaten sowie den Namen der Datenbank. Diesen lassen wir allerdings weg, denn wir wollen nicht nur eine Verbindung zu einer einzelnen

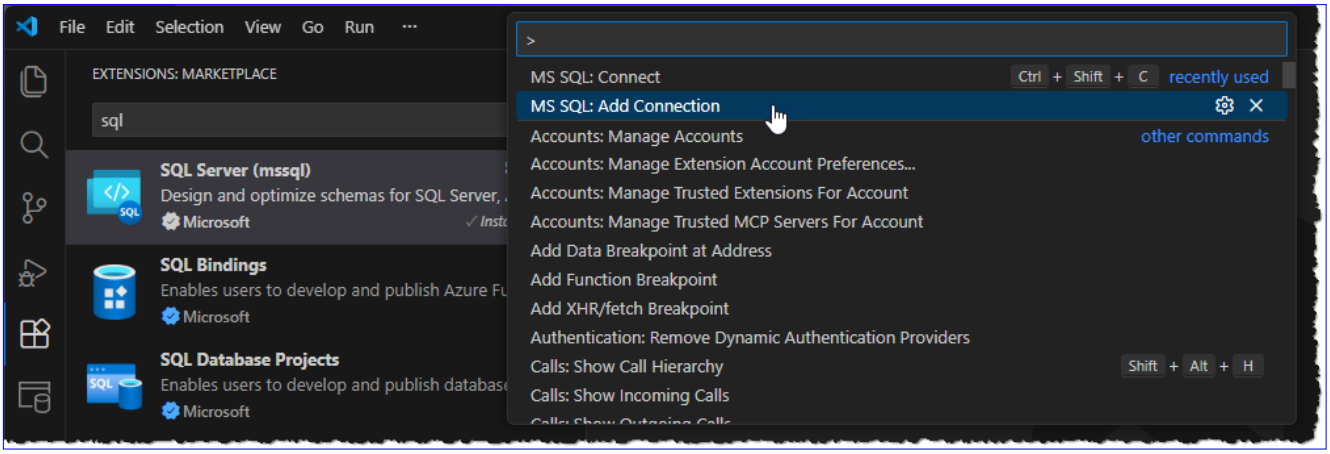

Bild 3: Hinzufügen einer Verbindung

SQL Server-Datenbank von Access aus updaten

Wenn wir eine Access-Anwendung mit SQL Server-Frontend an einen Kunden ausgeliefert haben, müssen wir sicherstellen, dass Updates problemlos funktionieren. Updates haben hier zwei Seiten: Einerseits kann das Access-Frontend um neue Funktionen erweitert werden, andererseits können diese Funktionen eine Anpassung der SQL Server-Datenbank erfordern. Das Aktualisieren der Access-Datenbank erfolgt im einfachsten Fall durch einfaches Ersetzen der .accdb-Datei. Beim Backend wird es ein wenig aufwendiger: Wir können es nicht einfach ersetzen, da die enthaltenen Daten im laufenden Betrieb bearbeitet wurden. Hier gibt es nun zwei Wege: Wir lassen uns ein Backup des Backends zukommen, aktualisieren es und spielen es anschließend wieder ein. Wenn die Anwendung bei mehreren Kunden verwendet wird, ist dies jedoch zu aufwendig. In diesem Fall können wir das Backend aber auch automatisch durch entsprechenden Code in der neuen Version des Frontends aktualisieren lassen. Wie das gelingt, zeigen wir in diesem Artikel.

Access-Entwickler wissen: Eine Access-Datenbank ist niemals fertig. Kunden haben immer neue Anforderungen, die umgesetzt werden müssen. Wenn die Anforderungen auch die Tabellen der Datenbank betreffen, müssen wir bei einer Aktualisierung des Frontends auch das Backend entsprechend erneuern, zum Beispiel indem wir Tabellen hinzufügen oder vorhandene Tabellen um Felder erweitern.

Bei einer Kombination aus Access-Frontend und -Backend ist es damit bereits getan. Wenn die Backend-Datenbank jedoch eine SQL Server-Datenbank ist, können noch weitere Änderungen hinzukommen: Neue gespeicherte Views, gespeicherte Prozeduren, Trigger oder Funktionen.

In den folgenden Abschnitten stellen wir die Voraussetzungen vor und zeigen auch, wie die Aktualisierung beim Start der neuen Version des Frontends automatisch durchgeführt werden kann, sodass die Anwendung direkt danach wieder in Betrieb genommen werden kann.

Aktualisierung per SQL

Die Aktualisierung der Elemente einer Backend-Datenbank geschieht beispielsweise beim Hinzufügen oder Ändern des Tabellenentwurfs durch entsprechende SQL-Skripte wie **CREATE TABLE**, **ALTER TABLE** oder, wenn Tabellen gelöscht werden sollen, auch durch **DROP TABLE**. Indizes und andere Elemente erstellen wir ebenfalls mit SQL-Anweisungen.

Bei einer reinen Access-Lösung mit einem Access-Backend kommt hier der zusätzliche Aufwand auf uns zu, diese Aufrufe manuell zusammenstellen zu müssen.

Es gibt keinen eingebauten Mechanismus, mit dem wir beispielsweise die Unterschiede zwischen zwei Access-Tabellen erfassen und in ein SQL-Skript gießen können.

Beim SQL Server haben wir es zumindest beim Erstellen vollständig neuer Elemente etwas leichter, denn die entsprechenden Skripte können wir uns im SQL Server Management Studio generieren lassen.

Schwieriger wird es, wenn wir nur ein Feld oder einen Index zu einer Tabelle hinzufügen wollen – hier müssen wir grundsätzlich erst einmal selbst das benötigte Skript schreiben.

Allerdings gibt es auch Tools, mit denen man die Unterschiede zwischen zwei Datenbankversionen ermitteln kann.

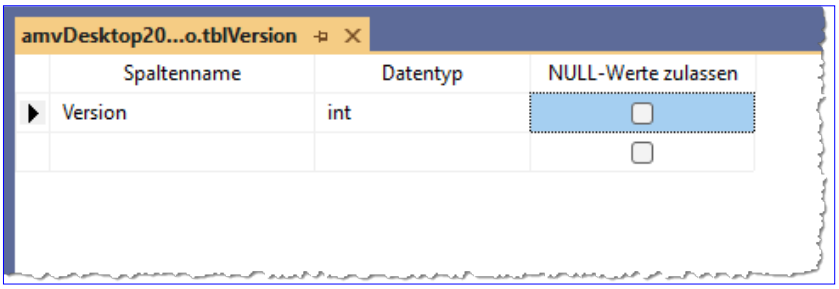

Bild 1: Versionstabelle im SQL Server

Eines davon stellen wir im Artikel **SQL Server-Datenbanken vergleichen mit VS.Code** (**www.vbentwickler.de/472**) vorgestellt – hier können wir zumindest die Unterschiede zwischen zwei Versionen ermitteln und daraus die notwendigen Anweisungen ableiten, zum Beispiel zum Ergänzen eines Feldes in einer Tabelle.

Voraussetzungen für den Abgleich

Wenn wir eine neue Version des Frontends an die Kunden verteilen und damit auch das Backend im SQL Server aktualisieren wollen, benötigen wir einige grundlegende Elemente.

Das erste ist eine Tabelle im Backend, in der wir die aktuelle Version des Backends festhalten. Diese enthält lediglich das Feld **Version** mit dem Datentyp **integer**, in dem wir die aktuelle Versionsnummer speichern (siehe Bild 1).

Außerdem benötigen wir noch zwei Tabellen im Access-Frontend, in denen wir die Informationen zum Aktualisieren des Backends speichern.

Die erste heißt **tblVersionen** und sieht im Entwurf wie in Bild 2 aus.

Hier speichern wir grundlegende Informationen zur jeweiligen Version, zum Beispiel die Versionsnummer, das Datum, an dem die Version erstellt wurde, Bemerkungen, das Ausführungsdatum des Updates und den Status des Updates.

Die zweite Tabelle heißt **tblVersionsdetails** (siehe Bild 3). Sie speichert die einzelnen Schritte, die zum Aktualisieren auf die jeweilige Version notwendig sind.

Hier finden wir zunächst ein Fremdschlüsselfeld namens **VersionID**, mit der die Zuordnung zu der Version aus der Tabelle **tblVersionen** hergestellt wird. Das Feld **SQL** enthält die auszuführende Anweisung, zum Beispiel zum Anlegen oder Löschen einer Tabelle, zum Hinzufügen von Feldern oder Indizes oder auch zum

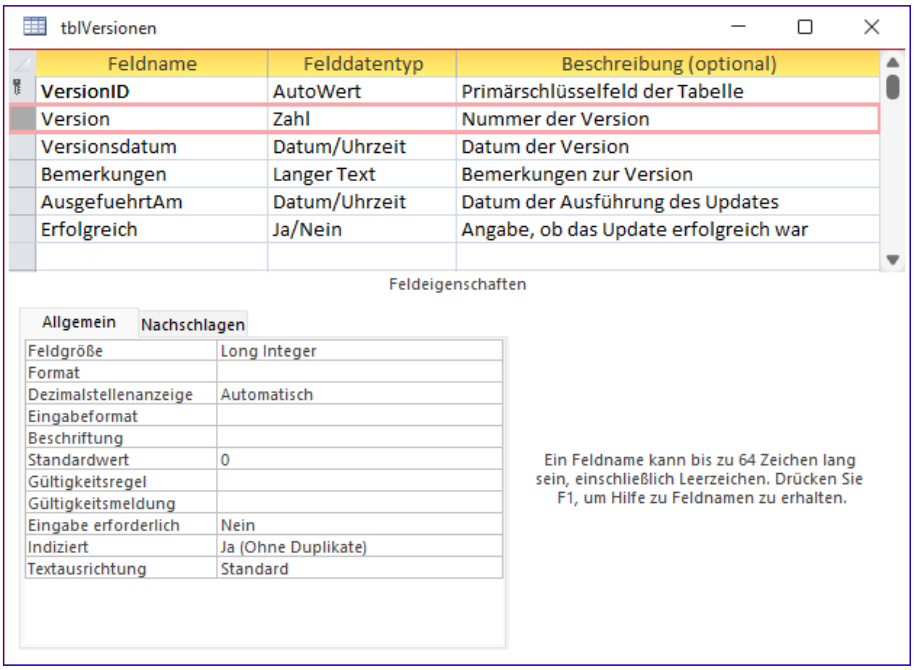

Bild 2: Versionstabelle im Access-Frontend

Anlegen von Views, gespeicherten Prozeduren oder Funktionen. Im Feld **ReihenfolgeID** legen wir fest, in welcher Reihenfolge diese Schritte ausgeführt werden sollen.

Die übrigen Felder dienen der Aufzeichnung der Ergebnisse der Aktualisierung.

Sie nehmen den Zeitpunkt der Aktualisierung, den Status und eine eventuelle Fehlermeldung auf, damit diese beim Fehlschlagen einer Aktualisierung ausgewertet werden können.

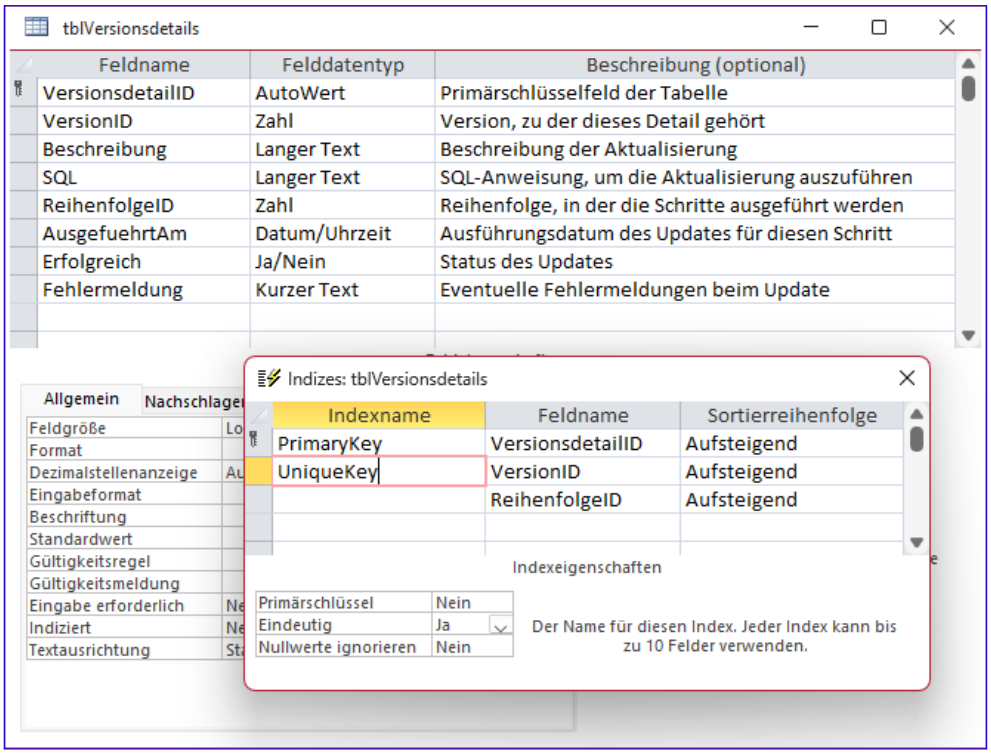

Bild 3: Tabelle der Versionsdetails

Für die beiden Felder **VersionID** und **ReihenfolgeID** haben wir einen zusammengesetzten, eindeutigen Index erstellt, damit jede **ReihenfolgeID** nur einmal je Version vorkommen kann.

Benutzeroberfläche zum Verwalten der Versionsupdates

Die Daten dieser Tabellen wollen wir in einem Formular samt Unterformular verwalten.

Der Entwurf des Hauptformulars samt Unterformular sieht wie in Bild 4 aus.

Das Hauptformular ist an die Tabelle **tblVersionen** gebunden und zeigt alle Felder dieser Tabelle an. Das Unterformular verwendet eine Abfrage basierend auf der Tabelle **tblVersionsdetails** als Datensatzquelle, welche die enthaltenen Daten nach dem Feld **ReihenfolgeID** filtert:

```
SELECT VersionsdetailID, VersionID, Beschreibung, SQL,
ReihenfolgeID, AusgefuehrtAm, Erfolgreich, Fehlermel-
dung FROM tblVersionsdetails ORDER BY tblVersionsdetails.
ReihenfolgeID;
```

Damit das Unterformular nur die Datensätze anzeigt, die zu dem im Hauptformular angezeigten Datensatz gehören, sind die Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformular-Steuerelements jeweils mit dem Wert **VersionID** gefüllt.

Das Unterformular (siehe Bild 5) ist als Endlosformular ausgelegt.

Es enthält neben den gebundenen Feldern noch zwei Schaltflächen, die das Ändern der Reihenfolge durch Verschieben nach oben oder nach unten ermöglichen.

Im Hauptformular haben wir für das Unterformular-Steuerelement das Ereignis **Beim Hingehen** definiert.

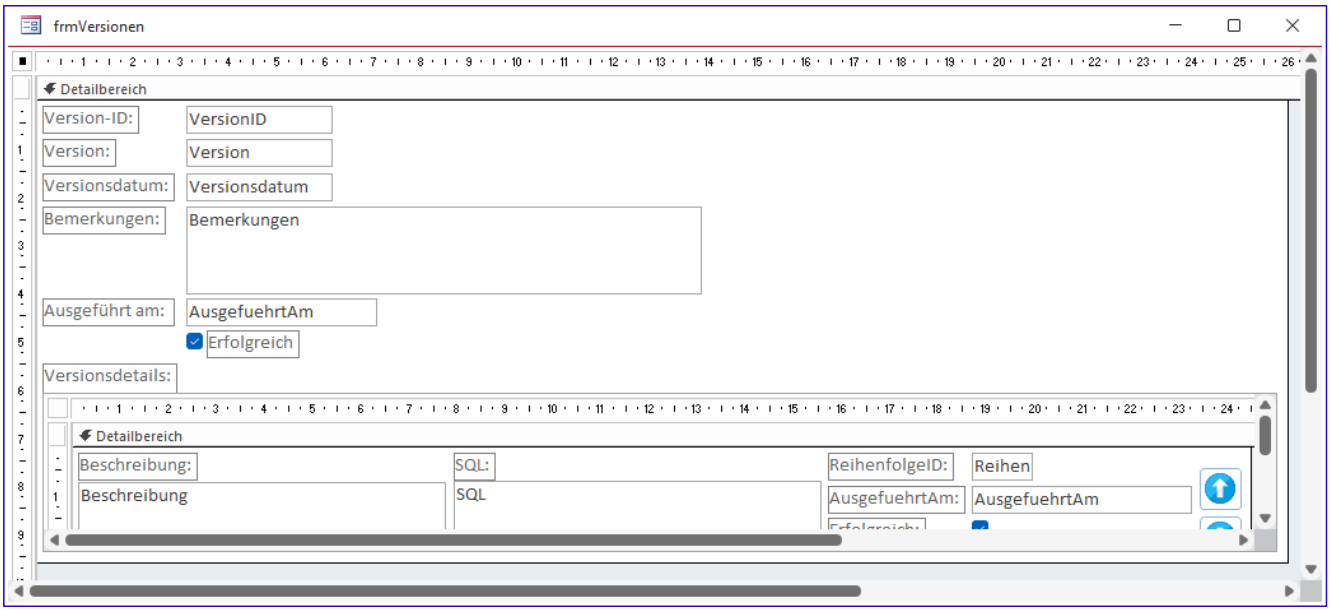

Bild 4: Haupt- und Unterformular zum Verwalten der Versionen und Versionsdetails

Hier prüfen wir, ob das Hauptformular einen vorhandenen oder einen neuen, leeren Datensatz anzeigt.

Falls es sich um einen neuen, leeren Datensatz handelt, soll eine Meldung angezeigt werden, damit zunächst ein Datensatz im Hauptformular angelegt wird:

```
Private Sub sfmVersionen_Enter()
    If Me.NewRecord Then
        MsgBox "Bitte lege zuerst eine Version an.", _
            vbOkOnly + vbExclamation, "Neue Version fehlt"
        Me.Version.SetFocus
    End If
End Sub
```

Ereignisse im Unterformular

Für das Ereignis **Beim Anzeigen** des Unterformulars haben wir die folgende Ereignisprozedur hinterlegt:

```
Private Sub Form_Current()
    Me.TimerInterval = 100
End Sub
```

Diese startet den Timer für 100 Millisekunden, dann wird die folgende Ereignisprozedur ausgelöst:

```
Private Sub Form_Timer()
    Me.TimerInterval = 0
    If Me.NewRecord Then
```

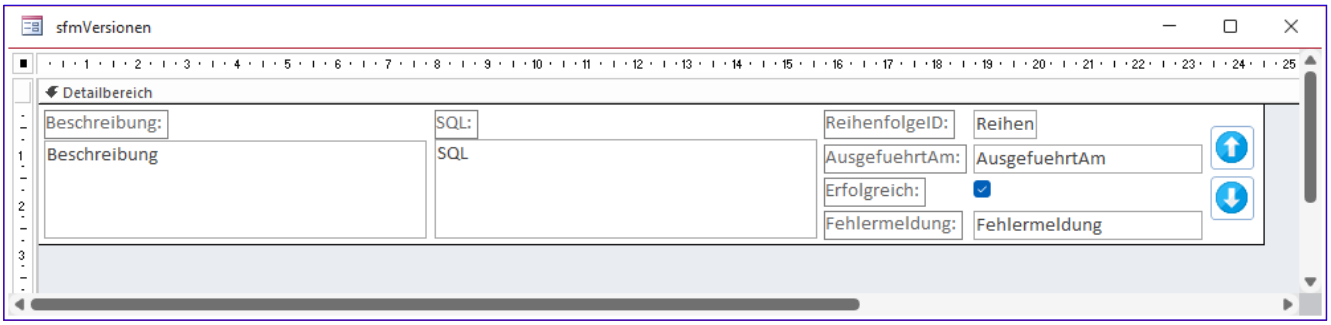

Bild 5: Unterformular zum Verwalten der Versionsdetails

```
        Me.ReihenfolgeID.DefaultValue = _
            Nz(DMax("ReihenfolgeID", "tblVersionsdetails", _
            "VersionID = " & Me.Parent.VersionID), 1)
    End If
End Sub
```

Diese setzt **TimerInterval** wieder auf **0** und prüft, ob der Benutzer gerade einen neuen, leeren Datensatz aktiviert hat.

In diesem Fall wird der Standardwert für das Feld **ReihenfolgeID** dieses Datensatzes auf den bisher höchsten vergebenen Reihenfolge-Wert der Datensätze aus **tblVersionsdetails** für die Version aus dem Hauptformular ermittelt und um eins erhöht.

Dies müssen wir verzögert machen, weil das Unterformular vor dem Hauptformular geladen wird und im Hauptformular noch kein Datensatz ist, für den wir die aktuell höchste vergebene **ReihenfolgeID** ermitteln können.

Nach 100 Millisekunden ist dies jedoch in der Regel der Fall.

```
Private Sub cmdNachOben_Click()
    Dim db As dao.Database
    Dim lngVersionID As Long
    Dim lngReihenfolgeZielID As Long
    Dim lngReihenfolgeAktuellID As Long
    Dim lngAktuellID As Long
    Dim lngZielID As Long

    Set db = CurrentDb

    lngVersionID = Me.Parent!VersionID

    Call ReihenfolgeErneuern(db, lngVersionID)

    lngReihenfolgeAktuellID = Me.ReihenfolgeID
    lngReihenfolgeZielID = Nz(DMax("ReihenfolgeID", "tblVersionsdetails", "VersionID = " & lngVersionID _
        & " AND ReihenfolgeID < " & Me!ReihenfolgeID), 0)
    If Not lngReihenfolgeZielID = 0 Then
        lngZielID = DLookup("VersionsdetailID", "tblVersionsdetails", "VersionID = " & lngVersionID _
            & " AND ReihenfolgeID = " & lngReihenfolgeZielID)
        lngAktuellID = Me!VersionsdetailID

        Call ReihenfolgeVertauschen(db, lngVersionID, lngAktuellID, lngZielID, lngReihenfolgeAktuellID, _
            lngReihenfolgeZielID)

        Me.Requery
    Else
        MsgBox "Kann nicht nach oben verschoben werden.", vbOKOnly + vbExclamation, "Kein Verschieben möglich"
    End If
End Sub
```

Listing 1: Verschieben des aktuellen Versionsdetails nach oben