HAUFE.
Ausgabe 04/2018

ACCESS
IM UNTERNEHMEN

IMPORT UND EXPORT

Nutzen Sie gespeicherte Importe und Exporte, um Daten blitzschnell auszutauschen (ab S. 32)

In diesem Heft:

ACTIVEX UND COM OHNE REGISTRIERUNG

Nutzen Sie externe Bibliotheken, ohne diese registrieren zu müssen!

SEITE 59

EVERYTHING FERNSTEUERN

Suchen Sie mithilfe des Tools Everything sehr schnell nach Dateien auf Ihrer Festplatte.

SEITE 45

TREEVIEW IN 64BIT-ACCESS NUTZEN

Die 64bit-Version des TreeView-Steuerelements ist da! Erfahren Sie, wie Sie es einsetzen.

SEITE 26

Import und Export

Still und heimlich hat Microsoft die veralteten Import- und Exportspezifikationen, die Sie im jeweiligen Assistenten speichern und abrufen konnten, durch eine neue Möglichkeit zum Speichern der notwendigen Konfiguration ersetzt. Diese hat den großen Vorteil, dass Sie sie nicht nur über die Benutzeroberfläche einfach nutzen, sondern auch per VBA fernsteuern können.

Daher widmen wir diesem Thema in der aktuellen Ausgabe gleich drei Beiträge. Der erste heißt **Duplikate aus Textdateien entfernen** (ab Seite 20). Hier nehmen wir uns zum Aufwärmen ein Beispiel aus der Praxis vor: Zwei Textdateien mit einer Liste von E-Mail-Adressen sollen zusammengeführt werden, sodass jede Adresse, die entweder auf der ersten oder der zweiten Liste vorkommt, in der zu erstellenden Liste erscheint. Dazu nutzen wir zunächst den Import, um die Textdateien in zwei Tabellen zu überführen und selektieren dann per Abfrage die gewünschten E-Mail-Adressen. Schließlich speichern wir das Ergebnis per Export wieder in einer identisch aufgebauten Textdatei.

Der Beitrag **Gespeicherte Importe und Exporte per VBA** liefert ab Seite 32 das Know-how, mit dem Sie per VBA sowohl Importe und Exporte ausführen als auch die Einstellungen für die gespeicherten Konfigurationen anpassen können. Damit liefert diese Funktion viel mehr Möglichkeiten als die bisher verwendeten Export- und Importspezifikationen.

Einen Schritt weiter gehen wir im Beitrag **Tabellenimport per VBA** ab Seite 37. Hier nutzen wir die gespeicherten Importspezifikationen, um die Tabellen einer Quelldatenbank wiederholt in die Zieldatenbank importieren zu können. Dabei werden die gegebenenfalls bereits vorhandenen Tabellen allerdings nicht überschrieben, sondern die Tabellen werden unter einem neuen Namen importiert. Dies hebeln wir durch ein paar zusätzliche VBA-Anweisungen aus. Schließlich zeigen wir auch noch, wie Sie einzelne Tabellen ersetzen können und sogar die vorhandenen Beziehungen löschen und diese nach dem Import wieder herstellen – wobei der Aufwand allerdings auch entsprechend steigt.

Sascha Trowitzsch liefert diesmal gleich drei tolle Beiträge. Im ersten greift er ein Thema aus der vorherigen Ausgabe auf und beschreibt, wie Sie mit SVG unter Access malen und zeichnen können. Diesen Beitrag finden Sie unter dem Titel **Malen und Zeichnen mit SVG** ab Seite 8. Ein weiterer Beitrag aus seiner Feder zeigt, wie Sie mit dem Tool **Everything** einfach in Dateien suchen können – und zwar erstens VBA-gesteuert und zweitens schneller als über den Windows Explorer. Mehr lesen Sie unter **Dateien schnell suchen mit Everything** ab Seite 45. Im Beitrag **ActiveX und COM ohne Registrierung verwenden** zeigt Sascha Trowitzsch ab Seite 59, wie Sie auf DLLs zugreifen können, ohne diese registrieren zu müssen! In diesem Beitrag steckt einiges an Know-how, das auch für absolute Profis interessant sein dürfte – lassen Sie sich überraschen!

Schließlich zeigt der Beitrag **Löschen von in Beziehung stehenden Daten** (ab Seite 2) noch, wie Sie verknüpfte Daten löschen, ohne Fehler zu erzeugen, und unter dem Titel **TreeView 64bit ist da – Anwendungen umrüsten** liefern wir ab Seite 26 Know-how, wie Sie Ihre Anwendung jetzt, da es die Bibliothek **MSCOMCTL.OCX** in einer 64bit-Version gibt, von 32bit auf 64bit umrüsten können.

Und nun: Viel Spaß beim Lesen!

Ihr André Minhorst

Löschen von in Beziehung stehenden Daten

Wenn Sie die Beziehung zwischen zwei Tabellen festlegen, haben Sie die Möglichkeit, referenzielle Integrität für die Daten zu definieren. Ist dies geschehen, legen Sie mit der Option Löschweitergabe auch noch fest, ob verknüpfte Daten beim Löschen von Datensätzen auf der einen Seite der Beziehung ebenfalls gelöscht werden sollen. Wenn Sie solche Daten in Formularen löschen, erhalten Sie unter Umständen Systemmeldungen, die für den Endnutzer der Anwendung nur wenig Aussagekraft haben. Dieser Beitrag zeigt, wie Sie mit dem Löschen von in Beziehung stehenden Daten umgehen können.

Unsere Südsturm-Beispieldatenbank liefert ausreichend Tabellen und Beziehungen, um die verschiedenen Konstellationen des Einsatzes der Löschweitergabe oder eben des Weglassens der Löschweitergabe zu beurteilen.

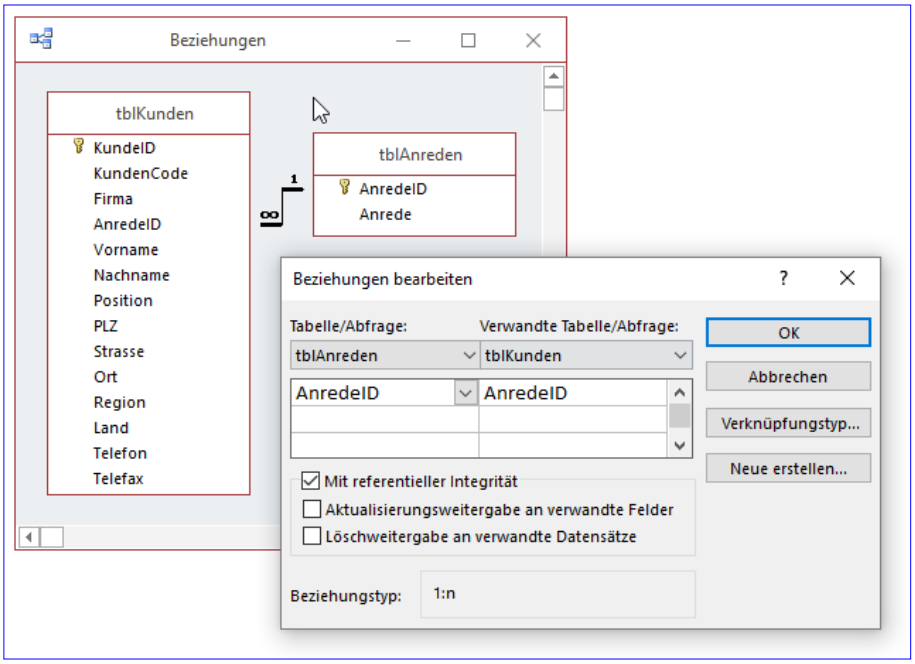

Bild 1: Anlegen einer Beziehung mit referenzieller Integrität, aber ohne Löschweitergabe

Löschweitergabe oder nicht?

Wie und wo wirkt die Löschweitergabe eigentlich? In der Beziehung zwischen Kunden und Anreden sind diese nicht definiert (siehe Bild 1).

Dies sorgt dafür, dass Sie in einer 1:n-Beziehung zwischen einer Kunden- und einer Anreden-Tabelle zwar Kunden löschen können, ohne dass die Datensätze der Anreden-Tabelle davon berührt werden.

Sie können aber keine Anreden löschen, die bereits mindestens einem Kunden zugeordnet wurden. Wenn Sie dies dennoch versuchen, erhalten Sie etwa in der Datenblattansicht die Meldung aus Bild 2.

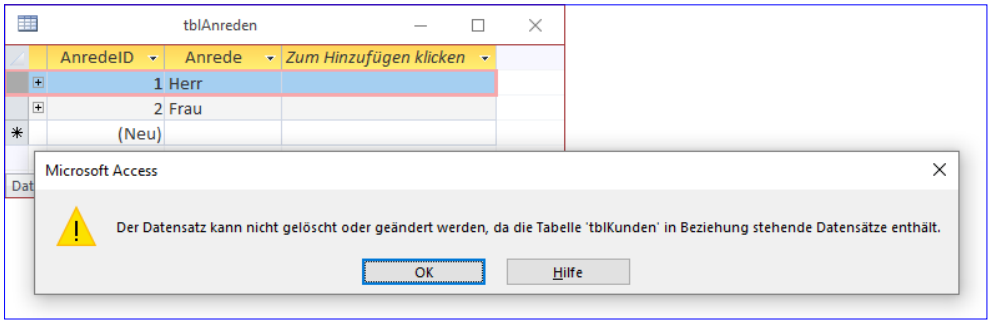

Bild 2: Ohne Löschweitergabe werden verknüpfte Datensätze auf der n-Seite der Beziehung geschützt.

Wenn hier nun die Option **Löschweitergabe** aktiviert wäre, würde das Löschen einer Anrede dazu führen, dass auch

alle damit verknüpften Kunden-Datensätze gelöscht würden – und das wäre einigermaßen fatal. Deshalb sollten Sie die Löschweitergabe nicht einfach so festlegen.

Eine Stelle, an der die Löschweitergabe sinnvoll ist, ist etwa die Beziehung von Kunden und Bestellungen. Wenn ein Kunde Sie auffordert, alle Kundendaten zu löschen, sind davon natürlich auch die Bestellungsdaten betroffen – Sie können ja keinen Datensatz in der Tabelle **tblBestellungen** behalten, ohne dass dieser einem Kunden zugeordnet ist. Wir gehen an dieser Stelle davon aus, dass Sie die Kunden- und Bestelldaten tatsächlich nicht mehr benötigen und diese löschen können. In diesem Fall legen Sie für die Beziehung zwischen den Tabellen **tblKunden** und **tblBestellungen** die Eigenschaft **Löschweitergabe an verwandte Datensätze** fest (siehe Bild 3).

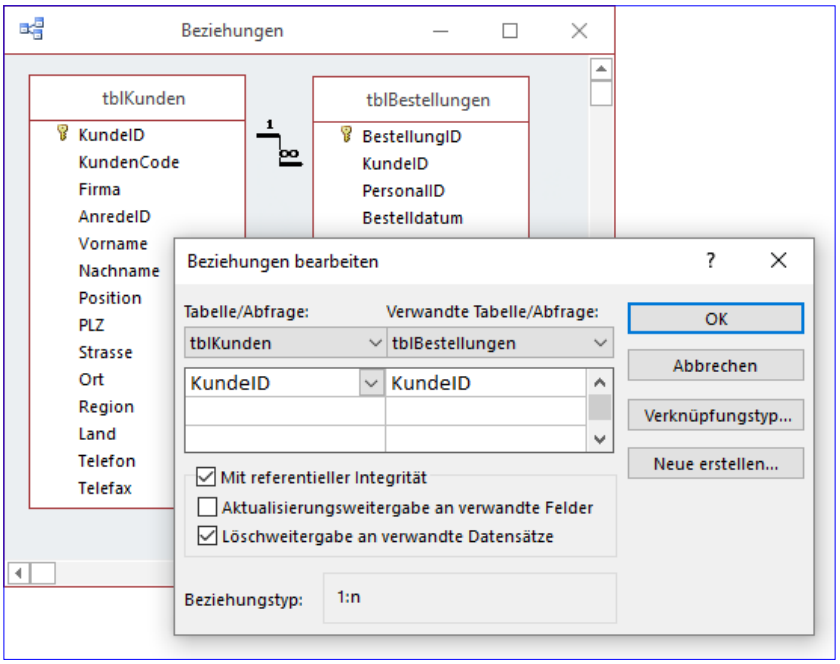

Bild 3: Löschweitergabe zwischen Kunden und Bestellungen

Ist dies der Fall und die Option **Clienteinstellungen|Bearbeiten|Bestätigen|Datensatzänderungen** in den Access-Optionen ist aktiviert, erhalten Sie vor dem Löschen eines Kunden-Datensatzes, dem bereits Bestelldatensätze zugeordnet sind, die Meldung aus Bild 4.

Wie Sie solche Meldungen durch eigene Meldungen ersetzen, können Sie im Beitrag **Löschen in Formularen: Ereignisse** nachlesen (**www.access-im-unternehmen.de/1128**). Wir schauen uns nun an, wie Sie die Meldungen, die beim Fehlschlagen des Löschvorgangs von

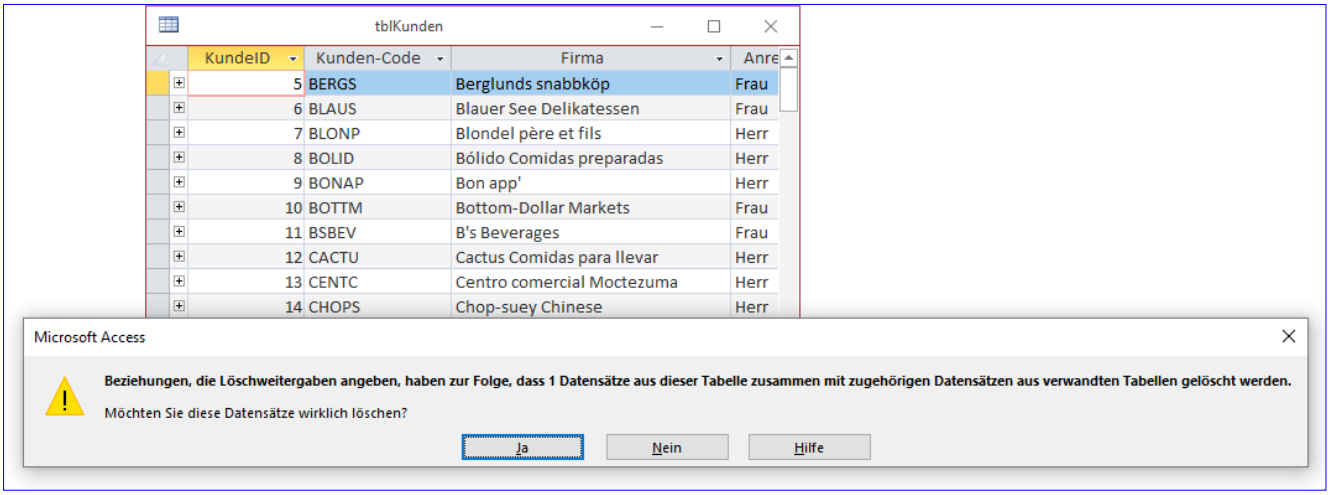

Bild 4: Bei aktiver Löschweitergabe erscheint mit Standardoptionen diese Meldung beim Versuch, einen Datensatz mit verknüpften Daten zu löschen.

Datensätzen ohne Löschweitergabe erscheinen, durch eine selbst definierte Meldung ersetzen können.

Eigene Meldung bei gescheitertem Löschen

Die obige Meldung **Der Datensatz kann nicht gelöscht oder geändert werden, da die Tabelle "tblKunden" in Beziehung stehende Datensätze enthält** können Sie nicht so einfach deaktivieren – auch nicht, wenn Sie die Access-Option **Clienteinstellungen|Bearbeiten|Bestätigen|Datensatzänderungen** deaktivieren.

Es gibt auch scheinbar keine Möglichkeit, per VBA in den Aufruf dieser Meldung einzugreifen – was auch logisch scheint, dann die Meldung kann ja auch komplett über die Benutzeroberfläche ausgelöst werden.

Die einzige Möglichkeit, hier einzugreifen, wird klar, wenn Sie diese Hinweismeldung anders interpretieren – nämlich als Fehlermeldung! Dennoch bleibt die Frage: Wo können wir eine Fehlermeldung unterbinden, die gar nicht per VBA ausgelöst wird? Die Lösung ist nicht leicht zu erraten, aber letztlich doch einfach.

Erstens gelingt dies ausschließlich in Formularen und nicht in der Datenblattansicht von Tabellen oder Abfragen. Zweitens erledigen wir das mit der Ereignisprozedur, die durch das Ereignis **Bei Fehler** des Formulars ausgelöst wird.

Für das Beispiel entfernen wir nun zunächst den Haken bei der Option **Löschweitergabe an verwandte Datensätze** der Beziehung zwischen den Tabellen **tblKunden** und **tblBestellungen**. Damit sorgen wir also dafür, dass die oben erwähnte Meldung wieder erscheint statt der Rückfrage, ob die verknüpften Datensätze mitgelöscht werden sollen (siehe Bild 5).

In einem Beispielformular, das einfach die Tabelle **tblKunden** mit allen Feldern im Detailbereich und dem Wert **Datenblatt** für die Eigenschaft **Standardansicht** verwendet (siehe Bild 6), wollen wir den Einsatz der Ereignisprozedur **Bei Fehler** demonstrieren.

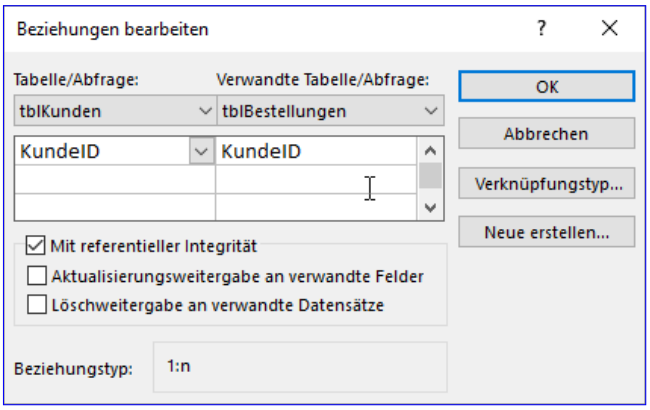

Bild 5: Entfernen der Löschweitergabe

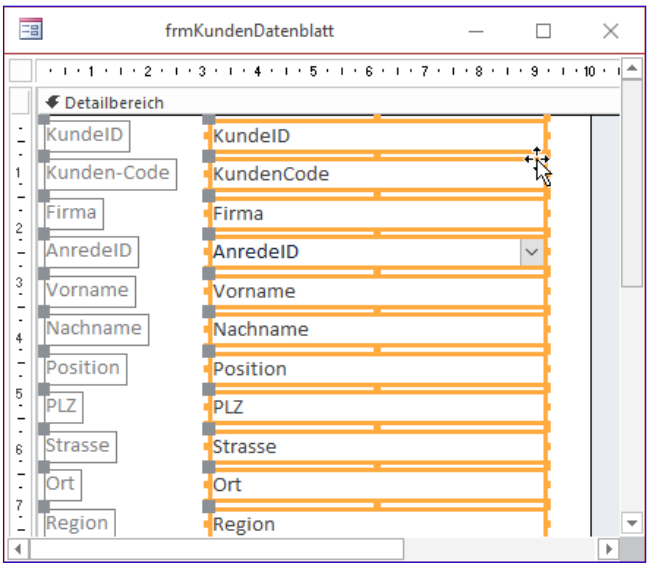

Bild 6: Entwurf des Beispielformulars

Dazu stellen wir den Wert der Eigenschaft **Bei Fehler** des Formulars auf **[Ereignisprozedur]** ein und klicken auf die Schaltfläche mit den drei Punkten. Die so im VBA-Editor automatisch angelegte Ereignisprozedur ergänzen wir mit folgender **MsgBox**-Anweisung:

```
Private Sub Form_Error(DataErr As Integer, _
                       Response As Integer)
    MsgBox DataErr
End Sub
```

Wenn wir nun einen Datensatz aus diesem Formular löschen, erscheint die Fehlermeldung aus Bild 7. Damit

Malen und Zeichnen mit SVG

Obwohl es sich bei SVG um Vektorgrafiken handelt, können Sie diese ebenfalls interaktiv mit der Maus in ein Webbrowser-Steuerelement zeichnen. Das Verfahren hat gegenüber den Pixel-Grafiken von HTML5 sogar noch einige Vorzüge, die die hier vorgestellte Lösung anbietet und erläutert. Mit keiner anderen Methode lassen sich auf so einfache Weise Grafiken erzeugen und in Tabellen abspeichern.

Referenz

Die Ausführungen dieses Beitrags setzen jene aus dem Artikel **SVG als Grafikkomponente** in dieser Ausgabe fort. Dort sind bereits die Grundlagen zum Einsatz des **Webbrowser-Steuerelements** im Verein mit **SVG** beschrieben.

SVG-Zeichnen im Unterschied zur Alternative HTML5

Im Beitrag **Malen und Zeichnen mit HTML5** veröffentlichten wir eine Beispielanwendung, über die Sie mit der Maus oder mit einem Grafiktablett in eine Zeichenfläche malen können. Die dabei entstehende Pixel-Grafik können Sie dabei in eine Bilddatei oder direkt binär in eine Tabelle abspeichern.

Hier bauen wir diese Anwendung nach und verwenden dabei **SVG** im **Webbrowser-Steuerelement**, wobei der Komfort des Malformulars noch etwas erhöht wird. Die Vorteile der **SVG**-Lösung lassen sich schnell aufzählen:

- **HTML5** erzeugt Pixel-Grafiken, die sich binär nur in einem der Standardformate abspeichern lassen. Verlustfreie Speicherung ermöglichen dabei nur die Formate **BMP**, **PNG**, **TIFF** und **GIF**. Der Speicherbedarf fällt dabei leider ziemlich hoch aus. Anders bei **SVG**, wo die Zeichnungen nur aus einer Liste von **XML**-Anweisungen bestehen, die erheblich weniger Bytes benötigen. Zudem lassen sich die **SVG**-Bilder beliebig hochskalieren.

- Die Elemente der Zeichnung lassen sich in einer **HTML5**-Grafik später nicht mehr identifizieren. In einer **SVG**-Grafik ist das anders. Einzelne Elemente können Sie entfernen oder nachbearbeiten. Die vorliegende Malanwendung erlaubt so etwa ein schrittweises **Undo**.

- **SVG** bringt von Haus aus Filtereffekte mit, die **HTML5** nicht kennt. Zwar gibt es auch dort einen **Shadow**-Effekt, der einen Schattenwurf der Zeichenelemente zulässt, doch von den komplexen Filterkombinationen von **SVG** ist das alles weit entfernt.

Der einzige Nachteil besteht darin, dass **SVG**-Elemente zur Laufzeit vom Browser gerendert werden. Das beeinträchtigt natürlich die Performance. Kritzeln Sie mit einem Grafiktablett etwa schnell einige Linien, so hinkt die Darstellung den Bewegungen deutlich nach. Aber Sinn und Zweck dieser Lösung ist ja auch nicht, unter Access ein vollwertiges Vektorgrafikprogramm zu realisieren – die Einsatzmöglichkeiten erwähnten wir im analogen **HTML5**-Beitrag.

SVG-Zeichenformular

Beschreiben wir zunächst die Bedienung des Zeichenformulars **frmSVGDraw** anhand von Bild 1. Im Zentrum befindet sich die Zeichenfläche in Gestalt eines Webbrowser-Steuerelements, das verankert ist und sich damit bei Größenänderungen des Formulars entsprechend ausdehnt. Die anderen Elemente richten sich dagegen am rechten oder unteren Rand aus.

Das Formular ist an eine Tabelle **tblBilderDraw** gebunden, so dass Sie pro Datensatz jeweils eine Zeichnung anfertigen oder darstellen können. Das Laden der Grafiken geschieht automatisch beim Wechsel eines Daten-

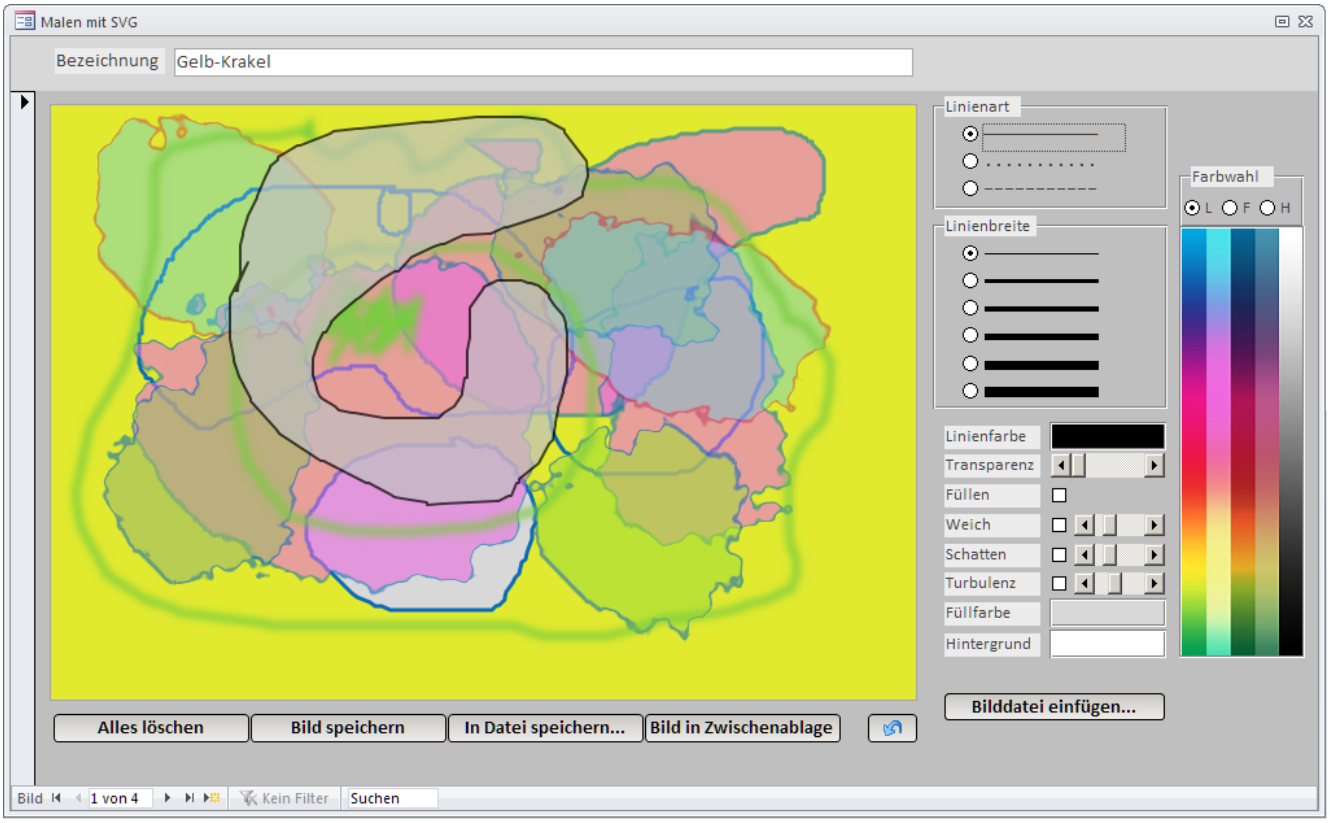

Bild 1: Zur Laufzeit entsteht im Formular beim Malen mit verschiedenen Werkzeugeinstellungen additiv eine speicherbare **SVG**-Grafik.

satzes. Jedem Bild können Sie oben eine **Bezeichnung** vergeben. Mit dem Button **Alles löschen** versetzen Sie die Zeichenfläche in den Ursprungszustand und erhalten eine weiße Zeichenfläche. **Bild speichern** erst transferiert das Gemälde in die Tabelle, welches ansonsten beim Datensatzwechsel verloren ginge. Sie können es zudem wahlweise über **In Datei speichern...** als **JPG**-Datei oder mit **Bild in Zwischenablage** in selbige exportieren. Der **Undo**-Button mit dem blauen Pfeil macht den letzten Zeichenvorgang oder auch frühere rückgängig. Sobald Sie die linke Maustaste loslassen, wird intern ein **SVG-path**-Element angelegt, das den gemalten Polygonzug repräsentiert. Das **Undo** löscht dieses Element aus dem **SVG-XML** wieder und setzt einen Zähler herab. Ein erneutes **Undo** löscht dann das Element mit diesem Zähler.

Rechts können Sie die **Linienart** bestimmen, bevor Sie loszeichnen. Möglich sind durchgezogene Linienzüge, gepunktete oder gestrichelte. Die Breite der Linien kann aus der darunterliegenden Optionsgruppe gewählt werden. Welche Malfarbe dabei verwendet wird, kann mit Klick auf die Fläche **Linienfarbe** zugewiesen werden. Das öffnet den normalen Windows-Farbauswahldialog. Alternativ haben Sie aber auch die Möglichkeit, die Farbe aus dem Spektrum rechts auszuwählen. Über diesem Spektrum befindet sich eine Optionsgruppe, die das Ziel der Farbauswahl angibt. Das **L** bezieht sich auf die **Linienfarbe**, das **F** auf die **Füllfarbe** und das **H** auf die Hintergrundfarbe.

Neben diesen Elementen können Sie mehrere Checkboxen bedienen. Ist **Füllen** aktiviert, so wird nicht nur ein Linienzug gezeichnet, sondern dieser automatisch geschlossen und mit einer einfarbigen Füllung versehen. Das geschieht bereits während des Malens! Die Formen in der Abbildung machen den Vorgang deutlich. Hier wurde der Schieberegler für die **Transparenz** eingesetzt. Befindet der sich ganz

links, so werden sowohl Linien als auch wie Füllungen, opak, während das Verschieben nach rechts beides immer durchsichtiger macht. Auf diese Weise überlagern Sie verschiedene Formen sequenziell.

Ist die Checkbox **Weich** aktiv, so werden Linien und Ecken weichgezeichnet. Die Checkbox **Schatten** aktiviert selbigen, wobei mit dem Schieberegler dessen Ausdehnung geregelt werden kann. Schließlich können Sie noch die Checkbox **Turbulenz** aktivieren und mit dem Schieberegler daneben den Grad derselben einstellen. Linien malen Sie damit nicht mehr als Linien, sondern als, sondern als turbulent verzerrte Polygonzüge, wie einige Formen in der Abbildung beweisen (siehe auch Bild 2).

Außerdem ist es über den Button **Bilddatei einfügen...** noch möglich, über einen Dateiauswahldialog eine **JPG**-Datei in die Zeichnung zu laden. Der Cursor über der Zeichenfläche ändert dann seine Gestalt in ein Kreuz, und erst, nachdem Sie nun auf eine Koordinate im Bild klicken, wird das Bild an dieser Stelle eingefügt. Ist ein Transparenzwert aktiviert, so wird auch diese Bilddatei transparent eingefügt.

Eine ganze Menge Features, die dieses Formular bereithält! Da sollte man wohl meinen, dass die Programmierung einen hohen Aufwand erfordert. Tatsächlich kommt

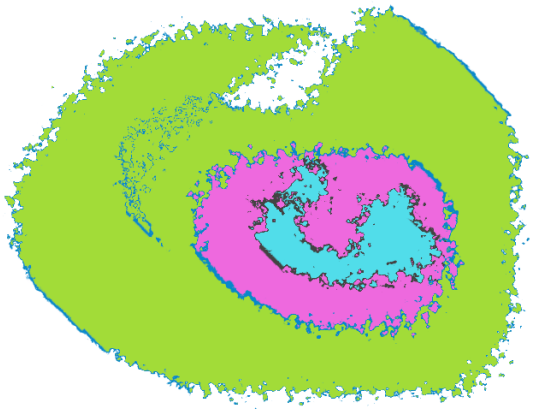

Bild 2: Linienzüge können bei aktivierter Turbulenz verzerrt werden.

die Formularklasse aber mit etwa **400** Zeilen VBA aus, ein Wert, der deutlich macht, wieviel Arbeit der Browser im Verein mit **SVG** uns da abnimmt. Wollten Sie dasselbe etwa mit dem Windows-**GDI-API** erreichen, so kämen Sie sicher auf einige Tausend Zeilen.

Tabelle zum Speichern

Die Tabelle **tblBilderDraw**, an die das Formular gebunden ist, hat den sehr einfachen Aufbau aus Bild 3. Genau genommen ist nur das Feld **Bildname** an das Textfeld **Bezeichnung** des Formulars gebunden, während der Inhalt der Grafik im Feld **BildOLE** per VBA abgespeichert wird. Der Name des Felds ist etwas irreführend, denn die Tabelle wurde schlicht aus der **HTML5**-Version der Malanwendung importiert. Dort kamen die Pixel binär tatsächlich in ein **OLE-Feld**. Hier aber haben wir es ja mit **XML-Text** zu tun, für den ein **Memo**-Feld besser passt.

Die mit Beispielen gefüllte Tabelle in der Datenblattansicht zeigt Bild 4. Sie erkennen sofort, dass es sich beim Inhalt von **BildOLE** um jeweils komplette **HTML**-Dokumente handelt, die sogar **JavaScript** enthalten. Die **SVG**-Anweisungen folgen erst am unteren Rand der Zeilen und leiten sich mit dem **Tag <svg...>** ein. Wenn Sie sich den Inhalt einer solchen Datenblattzelle einmal in einen Texteditor kopieren, so wird folgender Aufbau deutlich: Das **HTML**-Dokument liefert zunächst Basisdefinitionen, an die sich ein Block **JavaScript** anschließt, welcher Steuerungsfunktionen für die **SVG**-Filter enthält. Dann folgt der **SVG**-Teil. Hier gibt es einen Block, der die **SVG**-Filter und -Effekte definiert. Erst dann kommen die eigentlichen Zeichenelemente, bei denen es sich um eine Reihe von **polylines** handelt.

Die Größe des in **Bild 1** gezeigten Gemäldes beträgt im Feld **BildOLE** ungefähr **40 kB**. Davon entfallen auf den immer gleichen **HTML**-Rumpf **3 kB**. Also nehmen die **polyline**-Elemente den weitaus größten Teil ein. Da es sich hier um Text mit sich häufig wiederholenden Zahlen handelt, ließe sich das Feld im Bedarfsfall auch gut einer Textkompression unterziehen. Im Beitrag zur **RTLCom-**

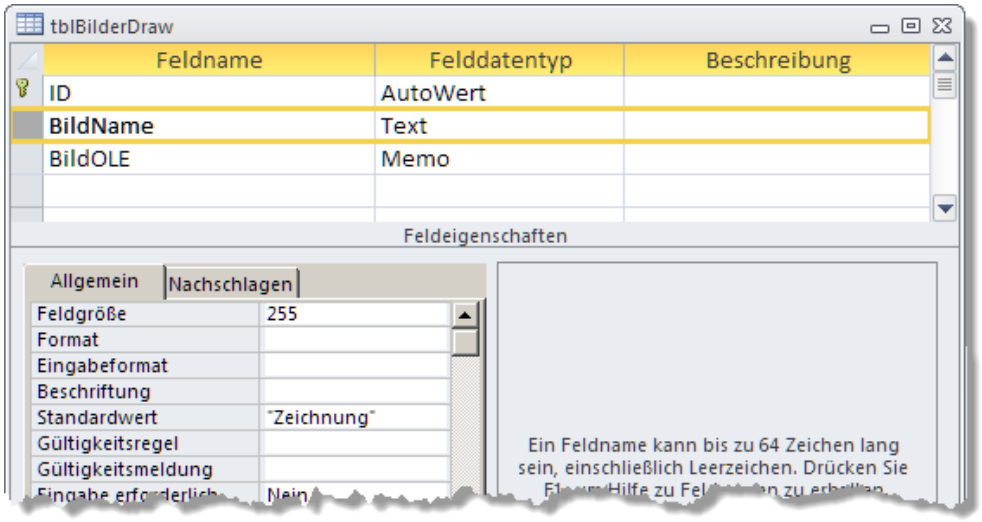

Bild 3: Die Tabelle **tblBilderDraw** speichert nur die gemalten Grafiken und ihre Namen nebst **ID**.

pression, die auch in der **HTML5**-Version zum Einsatz kommt, lernten Sie eine solche universelle Kompressionsroutine kennen.

Formularentwurf

Ans sich hält der Entwurf des Formulars **frmSVGDraw** wenig Überraschungen bereit. Das ungebundene Webbrowser-Steuerelement **ctlIE** hat den Steuerelementinhalt **="about:blank"** damit es sich mit leerer Seite öffnet und ist auf die **Verankerung Quer und nach unten dehnen** eingestellt. Die Verankerung der Button-Reihe steht dagegen auf **Unten links**, die der restlichen Steuerelemente für die Malwerkzeuge auf **Oben rechts**. Bild 5 gibt die Anordnung wieder.

Zwei eingesetzte Steuerelemente bedürfen wohl der Erläuterung. Die horizontalen Schieberegler etwa finden Sie nicht im Umfang von Access selbst. Es handelt sich hier um **ActiveX**-Elemente, horizontale Scrollbars, aus der Sammlung der **MSForms**-Steuerelemente, die grundsätzlich mit jeder Office-Version installiert werden. Nach dem Einsetzen des **ActiveX**-Steuerelements begeben Sie sich zum Eigenschaftenblatt und stellen unter dem Reiter **Andere** den Eintrag **Orientation** auf den Wert

Bild 4: In der Datenblattansicht der Tabelle **tblBilderDraw** wird deutlich, dass es sich bei den **SVG**-Grafiken um **HTML**-Anweisungen handelt.

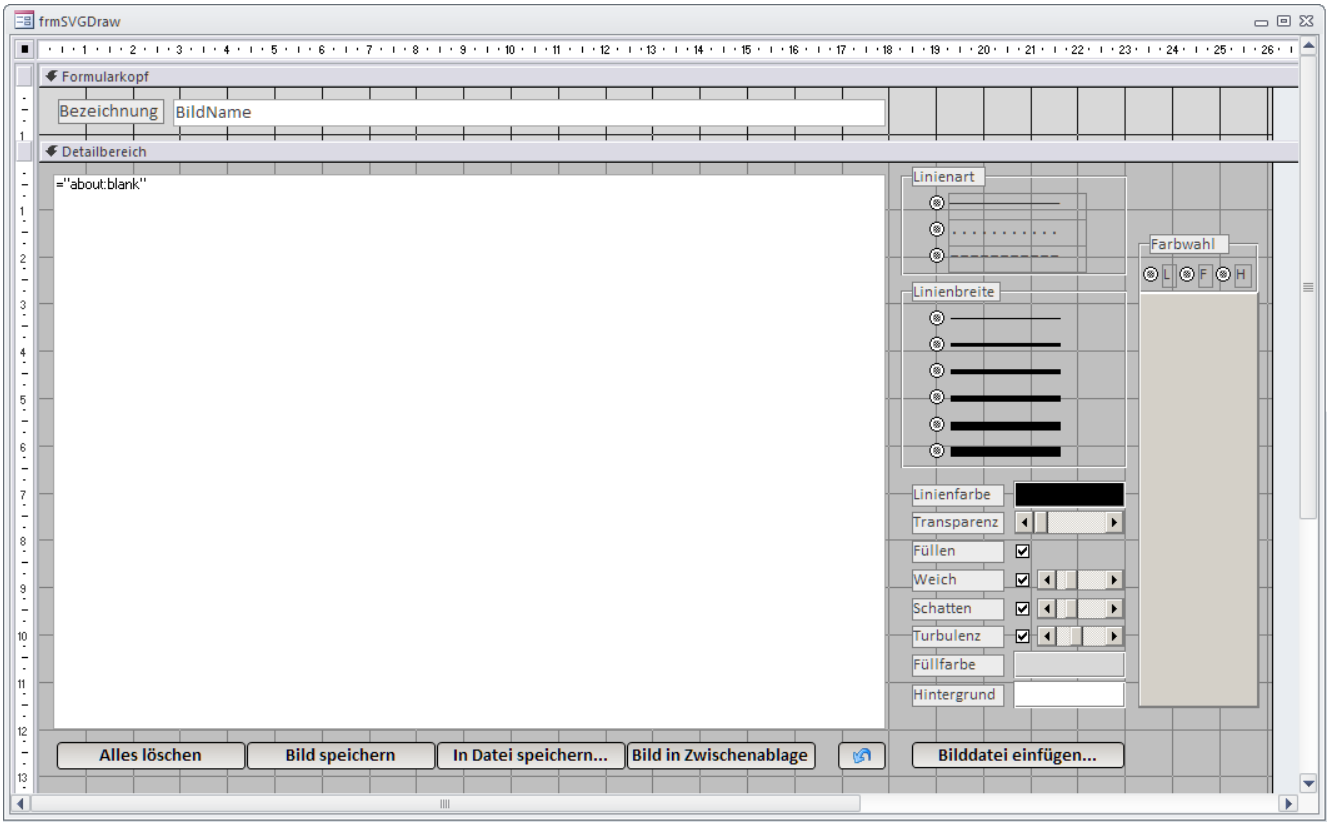

Bild 5: Die Entwurfsansicht der Malanwendung **frmSVGDraw** zeigt zentral das Webbrowsersteuerelement und rechts die Zeichenwerkzeuge.

Horizontal und die **Min/Max**-Werte auf **0** bis **100**. Das Bewegen eines Schiebers löst das Ereignis **Change** aus, auf das im Formularmodul in der zugehörigen Ereignisprozedur reagiert wird:

```
Private Sub ctlTransparent_Change()
    m_Transparency = 100 - ctlTransparent.Value
End Sub
```

Die Modulvariable **m_Transparency** erhält nun den negierten Wert, den das Steuerelement **ctlTransparent** zurückgibt. Für das Farbspektrum rechts verwenden wir ein **MSForms-Image**-Steuerelement. Das wird beim Laden des Formulars schlicht mit einer vorgefertigten Bilddatei **colors.jpg** versehen. Diese werden Sie im Projektordner indessen vergeblich suchen, denn sie ist tatsächlich in der Systemtabelle **MSysResources** abgespeichert, die für Grafiken auf Steuerelementen vorgesehen ist, aber ebenso manuell mit Dateien bestückt werden kann. Ein **MSForms-Image** kann sein Bild allerdings nicht direkt aus dieser Tabelle beziehen. Hier ist etwas Zusatzprogrammierung angesagt. Beim Laden des Formulars wird unter anderen diese Zeile durchlaufen:

```
Set Me!ctlImgColors.Object.Picture = _
                ArrayToPicture(GetColorTableJPG)
```

Das **Picture**-Objekt für das Steuerelement errechnet sich hier über zwei Hilfsfunktionen. **GetColorTableJPG** entnimmt der Systemtabelle die Binärdaten des Bilds und gibt sie als **Byte-Array** zurück:

```
Function GetColorTableJPG() As Byte()
    GetColorTableJPG = CurrentDb.OpenRecordset( _
        "SELECT BLOB FROM MSysResources WHERE Id=4", _
        dbOpenDynaset)(0).Value
End Function
```

Aus dem **Byte-Array** nun ein **StdPicture**-Objekt zu machen, ist natürlich eine wesentlich schwierigere Angelegenheit. Hier kommt mit der Funktion **ArrayToPicture** wieder einmal eine **GDIPlus-API**-Routine zum Einsatz, welche sich im Modul **mdlGDIPSpecial** versteckt.

Auf die Funktionsweise des Moduls können wir an dieser Stelle nicht eingehen. Der Klick auf einen Punkt im Spektrum-Image löst nun das Ereignis **MouseDown** aus. In dieser Ereignisprozedur muss die Farbe unter dem Mauszeiger bestimmt werden. Das lässt sich mit einer einzigen API-Funktion leichter erledigen, als gedacht:

```
Private Sub ctlImgColors_MouseDown( _
    ByVal Button As Integer, ByVal Shift As Integer, _
    ByVal x As Single, ByVal y As Single)
    Dim lColor As Long
    lColor = GetCursorPixel(x, y)
...
End Sub
```

GetCursorPixel nimmt die vom Ereignis zurückgegebenen **x**- und **y**-Koordinaten des Bildschirms entgegen und ermittelt für sie die Pixel-Farbe als **RGB-Long-Wert**.

Nebenbei sein noch erwähnt, dass Sie sich beim Webbrowser als Grafik-Engine nicht um die Skalierung von Bildern kümmern müssen. Sind geladenen oder erzeugte Bilder größer, als das Browser-Fenster, so blenden sich automatisch Scroll-Balken ein, mit denen Sie sich in die gewünschten Bildteile hineinbewegen können, wie Bild 6 demonstriert. Das gilt für geladenen Pixel-Grafiken in gleicher Weise, wie für **SVG**-Vektorgrafiken. Beim Abspeichern der Bilder wird dann aber selbstverständlich die teilweise nicht sichtbare Gesamtfläche verwendet.

Programmierung des Anwendungsformulars

Beginnen wir chronologisch mit den Ereignisprozeduren des Formulars, von denen die **Form_Open**-Prozedur die erste ist. Sie belegt lediglich einige modulweite **Member**-Variablen mit Vorgabewerten etwa zur Linienbreite

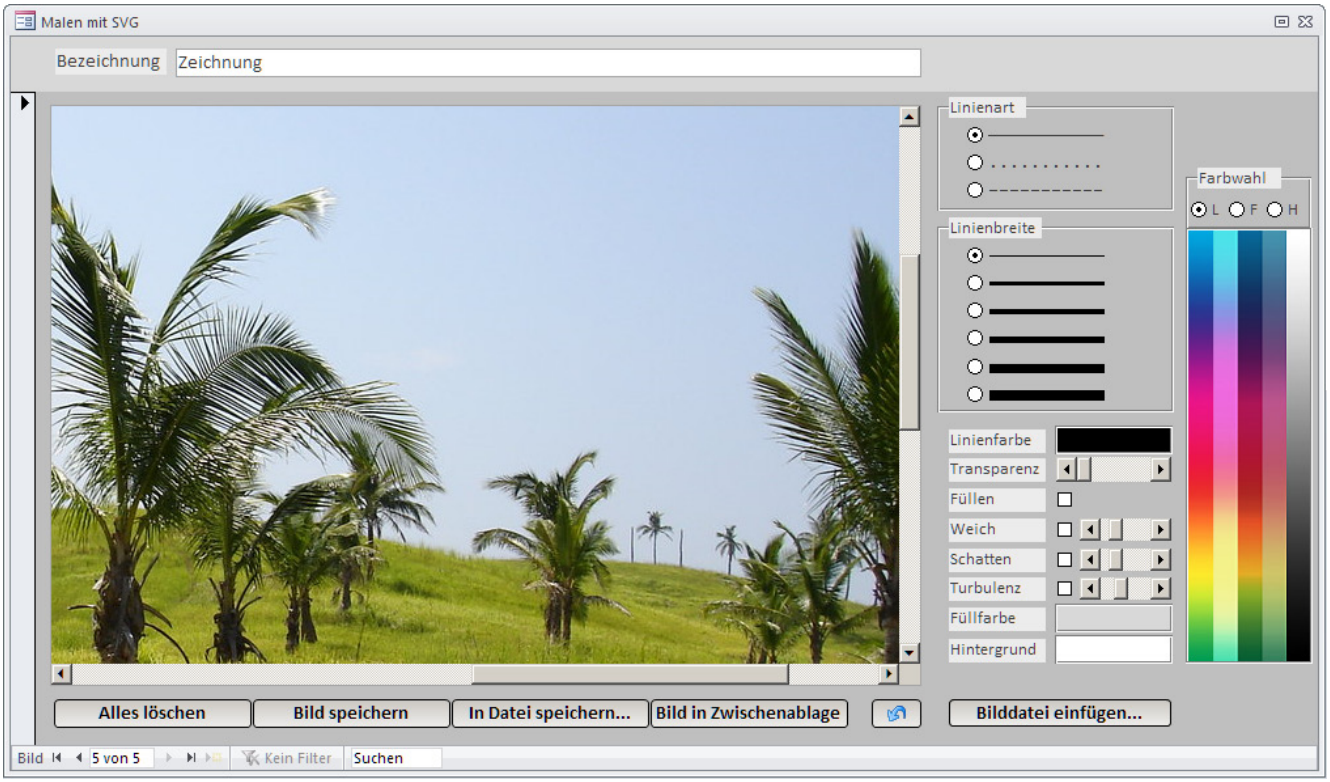

Bild 6: Um die Einblendung von Scroll-Balken brauchen Sie sich beim Webbrowser-Steuerelement mit **SVG**-Grafik nicht zu kümmern.

Duplikate aus Textdateien entfernen

Nehmen wir an, Sie erhalten zwei Textdateien mit E-Mail-Adressen, die teilweise gleich sind. Manche E-Mail-Adressen kommen aber nur in der ersten Textdatei vor und andere nur in der zweiten. Wir benötigen aber eine Liste, die alle E-Mail-Adressen enthält, die in mindestens einer der beiden Listen vorkommen. Bei einer kurzen Liste würde man die beiden wahrscheinlich nebeneinanderlegen und abgleichen, aber ab einer gewissen Anzahl wird es unübersichtlich und somit fehleranfällig. Also bemühen wir einfach unsere Lieblingsanwendung – Microsoft Access!

Listen nach Access

Die beiden Textdateien enthalten in unserem Beispielfall jeweils eine Liste von E-Mail-Adressen, Anreden und einer Statusinformation – alles für die spätere Nutzung in einem Newsletter (siehe Bild 1). Um aus den beiden Listen eine Liste mit allen vorkommenden E-Mail-Adressen zu machen, müssen wir diese zunächst in Access verfügbar machen. Dann bauen wir eine Abfrage, welche alle E-Mail-Adressen der beiden Listen so zusammenführt, dass jede nur einmal auftaucht. Diese können wir dann als neue Textdatei exportieren.

Schritt für Schritt

Ob wir die Listen einmalig nach Access importieren oder diese als verknüpfte Tabellen verfügbar machen, hängt jeweils vom Anwendungsfall ab. Wenn es öfter vorkommt, dass Sie solche Listen zusammenführen müssen, können Sie eine Verknüpfung herstellen und dann jeweils die Dateien austauschen. Wir wollen an dieser Stelle allerdings aus Performancegründen einen Import vornehmen, denn wenn die Daten sich in einer Access-Tabelle innerhalb der Datenbank befinden, können wir besser damit arbeiten, als wenn wir über eine Verknüpfungstabelle auf eine externe Textdatei zugreifen.

Also rufen wir zunächst den Ribbon-Befehl **Externe Daten|Importieren und Verknüpfen|Neue Datenquelle|Aus Datei|Textdatei** auf (siehe Bild 2). Im ersten Schritt des nun erscheinenden Assistenten wählen Sie die zu importierende Datei aus und behalten die Option **Importieren Sie die Quelldaten in eine neue Tabelle in der aktuellen Datenbank** bei. Danach folgt

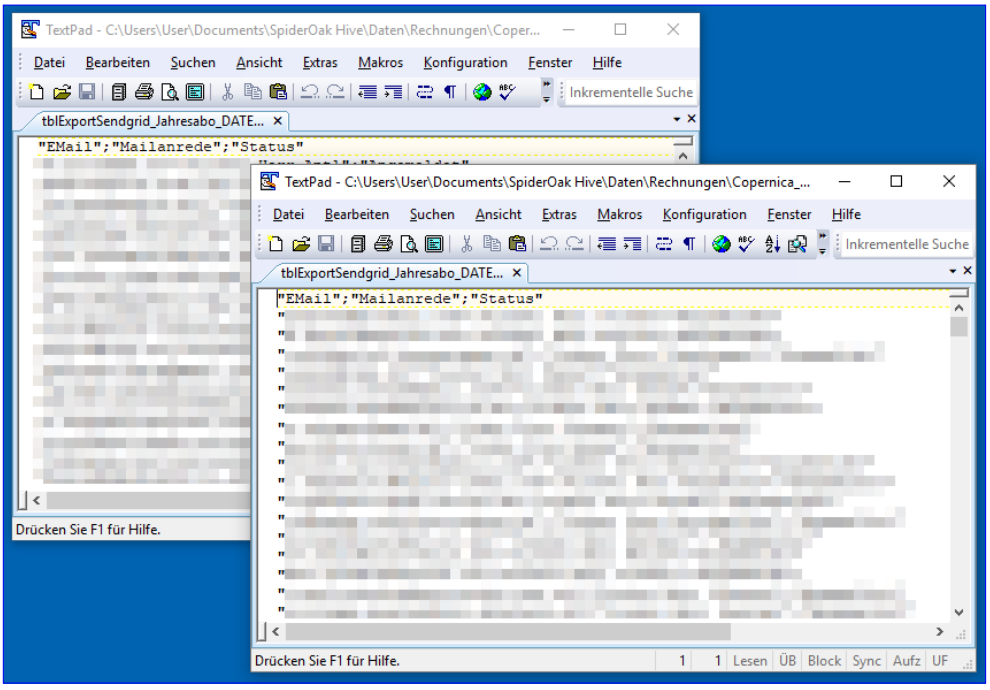

Bild 1: Zwei zusammenführende Textdateien

der Textimport-Assistent. Hier gehen Sie gleich weiter zur zweiten Seite des Assistenten und aktivieren dort die Option **Erste Zeile enthält Feldnamen** (siehe Bild 3). Auf der dritten Seite gibt es nichts zu tun, auf der vierten behalten Sie die Option **Primärschlüssel soll von Access hinzugefügt werden** bei. Auf der letzten Seite legen Sie noch den Namen der Tabelle fest, unter der die importierten Daten gespeichert werden sollen. Wir verwenden den Namen **tblEMail1**.

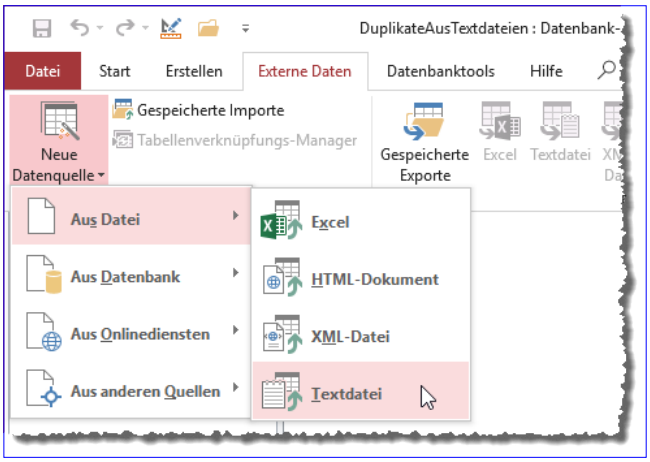

Bild 2: Start des Imports einer Textdatei

Nach dem Import fragt der Assistent im letzten Schritt, ob die Importschritte gespeichert werden sollen. Dies wollen wir in diesem Fall einmal tun, da wir ja gegebenenfalls noch öfter Daten auf die gleiche Art importieren wollen (siehe Bild 4). Auf die gleiche Art und Weise importieren wir nun auch die zweite Textdatei, diesmal allerdings in die Zieltabelle **tblEMails2**. Auch diesen Import können wir speichern.

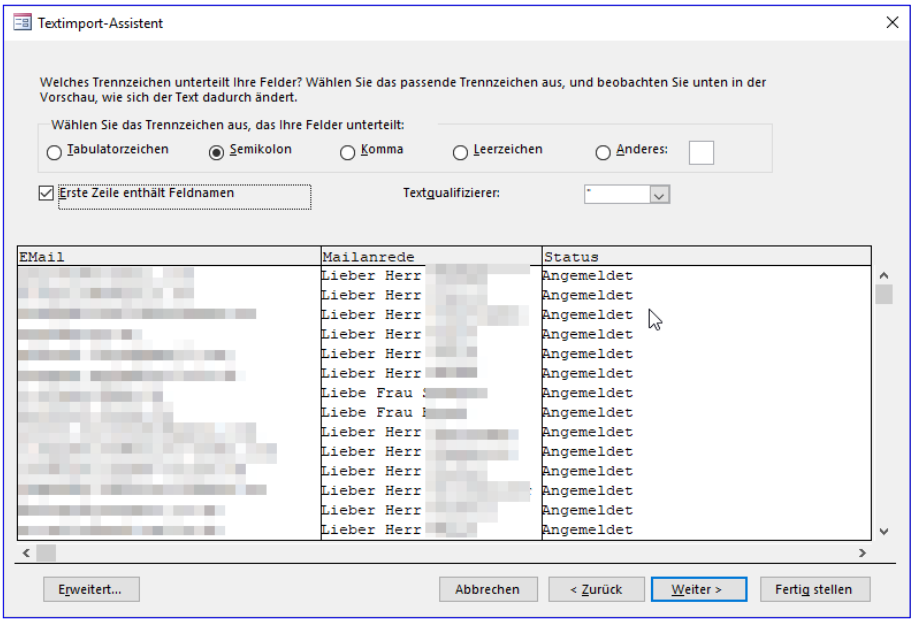

Bild 3: Einstellung der Option **Erste Zeile enthält Feldnamen**

Abfrage zum Zusammenführen der Datensätze

Nun kümmern wir uns um die Abfrage, welche die Datensätze zusammenführen soll, ohne Duplikate zu liefern. Der neuen Abfrage namens **qryEMailAdressenZusammenfuehren** fügen wir zunächst die beiden Tabellen **tblEMails1** und **tblEMails2** als Datenherkunft hinzu – allerdings nicht auf herkömmliche Weise über die Entwurfsansicht, sondern über die SQL-Ansicht. Der Grund ist, dass wir eine **UNION**-Abfrage nutzen wollen, um die Datensätze der beiden Tabellen zusammenzuführen.

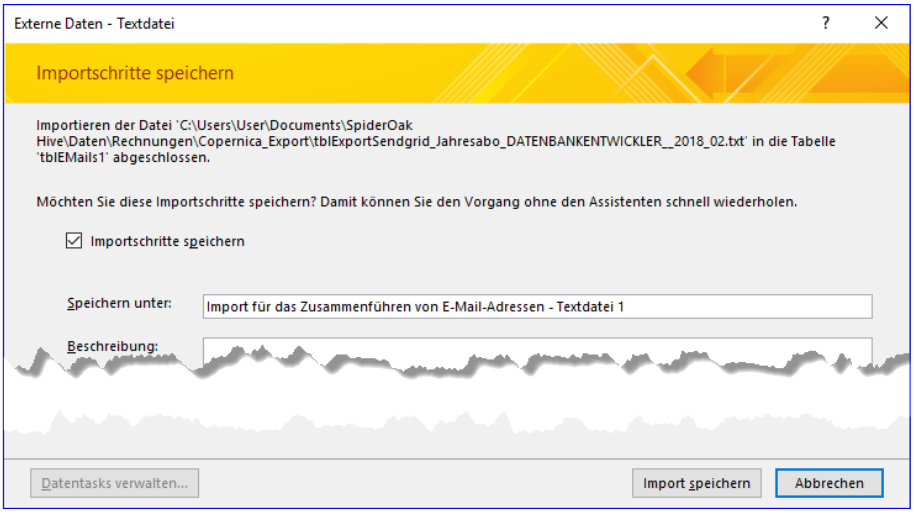

Bild 4: Speichern der Importschritte

Dazu geben Sie dann den folgenden Text ein:

```
SELECT EMail, MailAnrede, Status FROM tblEMails1
UNION
SELECT EMail, MailAnrede, Status FROM tblEMails2
```

Die erste Beispieltabelle hat 674 Datensätze, die zweite hat 672. Die erste Tabelle enthält ein paar Datensätze, die nicht in der zweiten Tabelle vorkommen und umgekehrt. Das Ergebnis der **UNION**-Abfrage liefert 675 Datensätze. Das scheinen zu wenige Datensätze zu sein: Es würde ja bedeuten, dass die zweite Tabelle nur einen Datensatz enthält, der nicht in der ersten Tabelle vorkommt, und das ist nicht der Fall – davon haben wir uns zuvor durch Überfliegen der beiden Tabellen überzeugt.

Also prüfen wir genau, wie viele Datensätze in beiden Tabellen vorkommen – und verschaffen uns so eine kleine VBA-Fingerübung. Hier gehen wir von der Anzahl der Datensätze in der ersten Tabelle aus, die wir mit der **DCount**-Funktion ermitteln:

```
Public Sub DatensaetzeZaehlen()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim lngAnzahl As Long
    Set db = CurrentDb
    lngAnzahl = DCount("*", "tblEMails1")
```

Dann erstellen wir ein Recordset auf Basis der Tabelle **tblEMails2** und durchlaufen alle Datensätze dieses Recordsets. Dabei prüfen wir für jeden Datensatz, ob dieser auch in der Tabelle **tblEMails1** vorkomme. Falls nicht, erhöhen wir **lngAnzahl** jeweils um **1**:

```
    Set rst = db.OpenRecordset("tblEMails2", _
        dbOpenDynaset)
    Do While Not rst.EOF
        If Nz(DLookup("EMail", "tblEMails1", "EMail = '" _
                & rst!EMail & "'"), "") = "" Then
            lngAnzahl = lngAnzahl + 1
        End If
        rst.MoveNext
    Loop
```

Das Ergebnis geben wir dann im Direktfenster aus:

```
    Debug.Print lngAnzahl
End Sub
```

Diese Prozedur liefert als Ergebnis für die gemeinsamen Datensätze den Wert **678**, was realistisch erscheint. Wie können wir dieses per Abfrage nachbilden?

In tblEMails1, aber nicht in tblEMails2

Um herauszufinden, wie viele E-Mails in der Tabelle **tblEMails1** vorhanden sind, die wir nicht in **tblEMails2** finden, erstellen wir die Abfrage aus Bild 5.

Dies liefert sechs Datensätze, was das Ergebnis der VBA-Prozedur bestätigt (siehe Bild 6). Wenn wir die 672 Datensätze aus **tblEMails2** zugrunde legen und die 6 aus der Abfrage hinzuaddieren, kommen wir auch auf **678**.

Andersherum können wir das auch noch machen, um das Ergebnis abzusichern. Hier kommt dann **4** heraus, was in

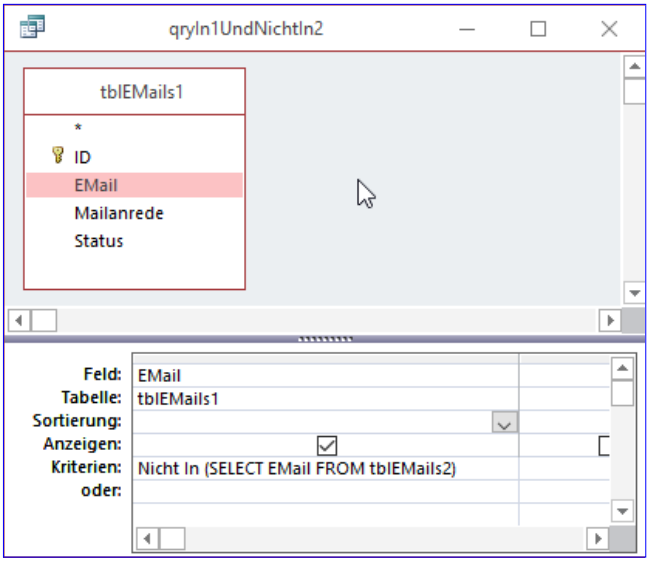

Bild 5: Datensätze in **tblEMails1**, aber nicht in **tblEMails2**

TreeView 64bit ist da – Anwendungen umrüsten

Es ist geschehen: Jahre nach Einführung der ersten 64bit-Version von Microsoft Access (wer erinnert sich noch?) hat Microsoft die 64bit-Version auch mit einer passenden Version der Bibliothek MSCOMCTL.ocx versehen. Das Fehlen von TreeView, ListView und Co. war bisher einer der Hauptgründe, nicht auf 64bit zu wechseln. Derweil gehen dennoch viele Unternehmen diesen Schritt – meist vermutlich einfach, um eine »modernere« Version von Access zu nutzen. Hier und da vernimmt man allerdings auch Stimmen, dass die 64bit-Version an verschiedenen Stellen Probleme vermeidet, die unter der 32bit-Version aufgetreten sind. In diesem Beitrag schauen wir uns an, was beim Einsatz von Steuerelementen der MSCOMCTL.ocx-Bibliothek zu beachten ist und wie Sie 32bit-Anwendungen fit machen für die 64bit-Version.

Als ich von einem Entwicklerkollegen hörte, dass es nunmehr endlich eine 64bit-Version der **MSCOMCTL.ocx** gibt, habe ich alles andere liegen lassen und eine virtuelle Maschine mit Office 2016 in der 64bit-Version ausgestattet. Anmerkung an dieser Stelle: Dies geschah mit einer Office-Version eines Office 365-Abonnements.

Experimente

Unter der 64bit-Version von Access habe ich also eine neue Datenbank erstellt, dieser ein Formular hinzugefügt und dann tatsächlich in der Liste der verfügbaren ActiveX-Steuerelemente das TreeView-Steuerelement gefunden (siehe Bild 1).

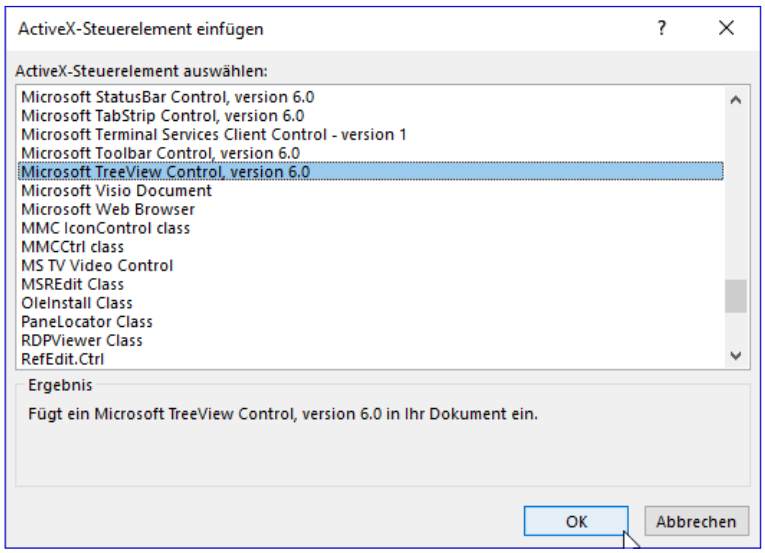

Bild 1: Einfügen des TreeView-Steuerelements unter 64bit

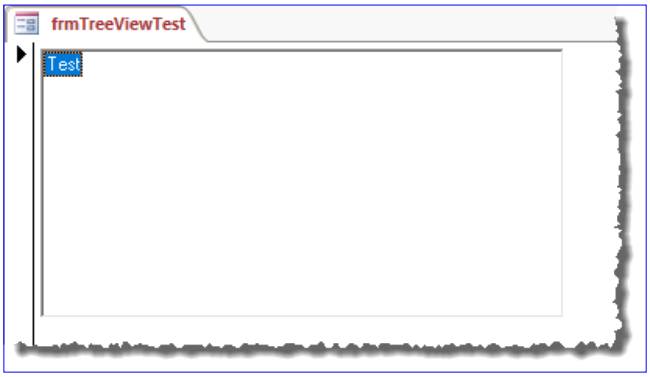

Bild 2: Das TreeView-Steuerelement funktioniert!

Das so hinzugefügte Steuerelement hat den Namen **ctlTreeView** bekommen. Um es zu testen, habe ich folgenden Code zum Klassenmodul des Formulars hinzugefügt:

```
Private Sub Form_Load()
    Dim objTreeView As MSComctlLib.TreeView
    Set objTreeView = Me!ctlTreeView.Object
    objTreeView.Nodes.Add , , , "Test"
End Sub
```

Das Ergebnis sehen Sie in Bild 2: Das TreeView wird angezeigt und funktioniert wie gewünscht.

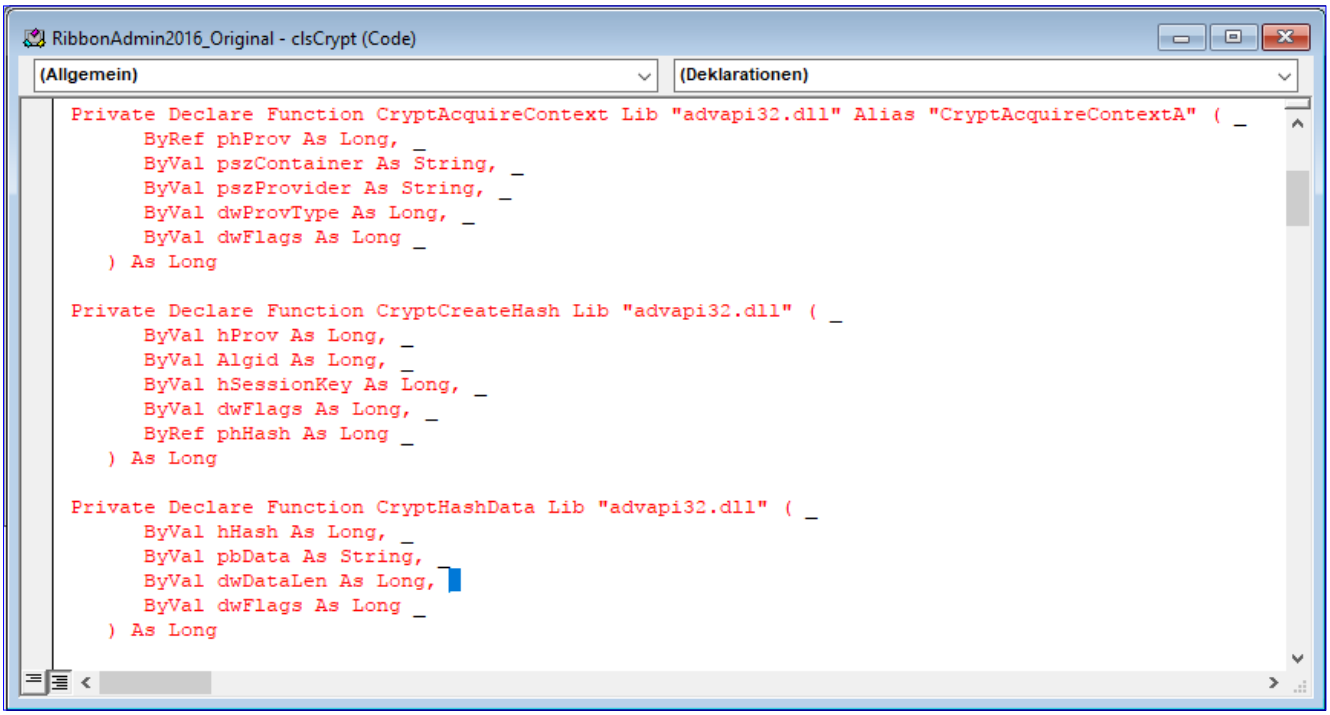

Bild 3: Fehlerhaft markierte API-Funktionen

TreeView: Von 32bit zu 64bit und zurück

Wir haben dann die unter der 64bit-Version von Access erstellte Datenbank auf ein System mit Access in der 32bit-Version von Access kopiert und gestartet. Das TreeView-Steuerelement dieser mit der 64bit-Version erstellten Datei funktioniert auch unter der 32bit-Version.

Danach haben wir die gleiche Datei unter Access in der 32bit-Version erstellt und diese dann auf das mit der 64bit-Version ausgestattete System kopiert. Auch hier gelingt die Anzeige des TreeView-Steuerelements ohne Probleme.

Anwendung umrüsten

Damit sind die Voraussetzungen gegeben, einen genaueren Blick auf die Umrüstung einer Anwendung auf Basis von 32bit-Access auf die 64bit-Version zu werfen – oder gleich auf eine Umrüstung, die mit beiden Versionen kompatibel ist. Dazu nehmen wir uns als Beispiel die Ribbon-Admin-Anwendung vor, die eine Menge API-Code enthält – und genau dieser ist in der Regel nicht kompatibel mit 64bit-Systemen, wenn er unter 32bit-Access geschrieben wurde.

Probleme im Code

Die offensichtlichen Probleme tauchen schon auf, wenn Sie die Module mit API-Deklarationen einer 32bit-Datenbank öffnen – die API-Funktionen werden komplett rot markiert, was kein gutes Zeichen ist (siehe Bild 3).

Der Menübefehl **Debuggen|Kompilieren** liefert dann genauere Informationen in Form einer Meldung (siehe Bild

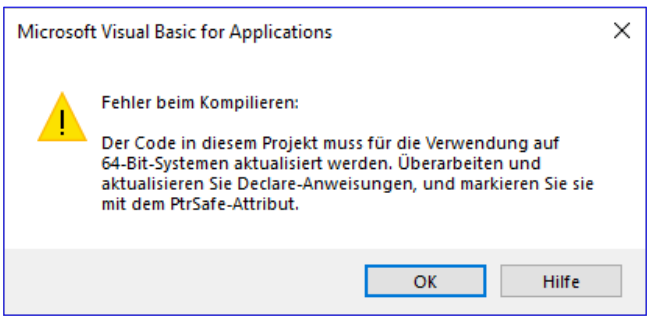

Bild 4: Meldung beim Versuch, das Projekt zu debuggen

4). Die Erläuterung ist eindeutig – wir sollen die Deklarationen mit dem Schlüsselwort **PtrSafe** erweitern. Nur wo? Das ist nach kurzem Experimentieren beantwortet – nämlich zwischen **Declare** und **Function/Sub**:

```
Private Declare PtrSafe Function SendMessage Lib "user32"
Alias "SendMessageA" (ByVal hwnd As Long, ByVal wMsg As
Long, ByVal wParam As Long, lParam As Any) As Long
```

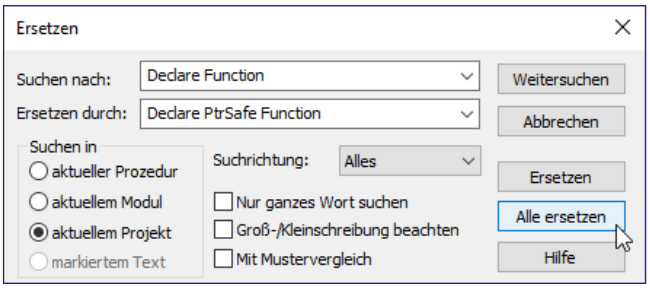

Bild 5: Suchen und Ersetzen

Dies manuell zu erledigen ist natürlich eine langweilige Aufgabe. Also ziehen wir den **Suchen/Ersetzen**-Dialog zur Unterstützung heran und geben im **Suchen nach**-Textfeld den Ausdruck **Declare Function** und im **Ersetzen durch**-Textfeld den Ausdruck **Declare PtrSafe Function** (siehe Bild 5).

Der folgende Versuch, die Anwendung zu debuggen, liefert die **Declare Sub**-Deklarationen als fehlerhafte Zeilen. Also führen wir den Suchen/Ersetzen-Vorgang nochmals durch, nur diesmal mit dem Suchbegriff **Declare Sub** und dem Ersetzungsausdruck **Declare PrtSafe Sub**.

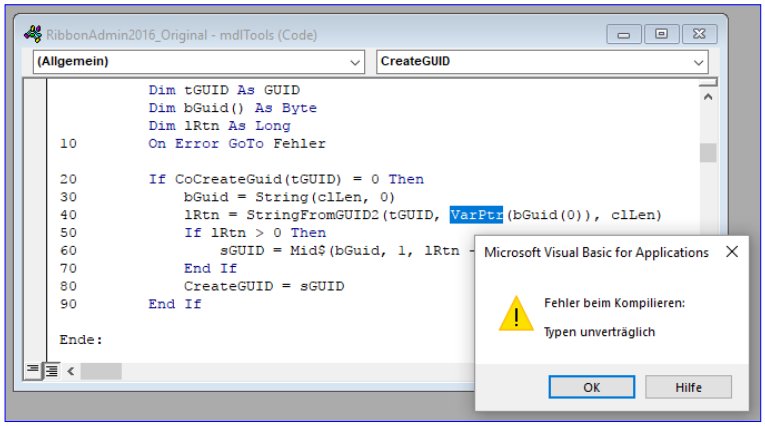

Bild 6: Unverträgliche Typen

Danach versuchen wir erneut, die Anwendung zu kompilieren. Nun tauchen Fehler wie der in Bild 6 auf. Was ist hier das Problem?

Um dies herauszufinden, schauen wir uns über den Kontextmenü-Eintrag **Definition** die Definition des markierten Elements an, hier der Funktion **VarPtr** (Bild 7). Diese erwartet als Parameter einen Wert des Typs **Any** und liefert einen Wert des Typs **LongPtr** zurück.

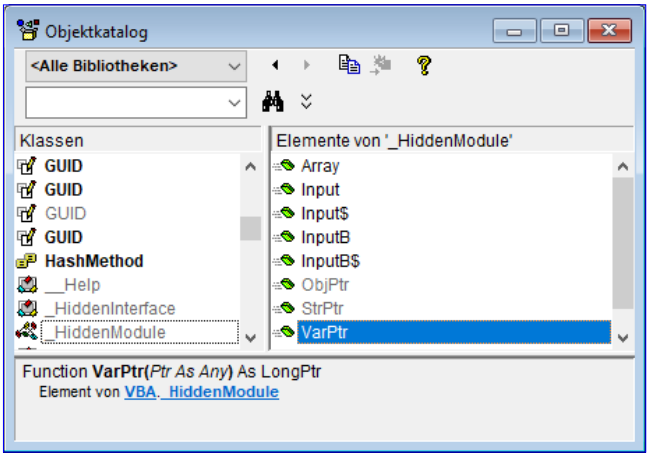

Bild 7: Prüfen der Datentypen von Parameter und Rückgabewert

Der Datentyp **LongPtr** ist 64bit-kompatibel und umfasst einen Wertebereich von 2 hoch 64, während der 32bit-kompatible Datentyp **Long** nur 2 hoch 32 Werte umfasst.

Warum aber macht die Funktion **VarPtr** nun Probleme? Dazu schauen wir uns die Deklaration der API-Funktion **StringFromGUID2** an, der wir das Ergebnis dieser Funktion als Parameter übergeben. Die Funktion wie folgt aus:

```
Private Declare PtrSafe Function StringFromGUID2 Lib
"ole32.dll" (rGUID As Any, ByVal lpstrClsId As Long, ByVal
cbMax As Long) As Long
```

Gespeicherte Importe und Exporte per VBA

Access bietet in aktuelleren Versionen die Möglichkeit, einen Import oder Export nach dem Abschluss zu speichern. So können Sie diese einfach wieder aufrufen, um den Import oder Export zu wiederholen. Noch schöner ist, dass man auf diese gespeicherten Informationen auch per VBA zugreifen kann. Wie das gelingt und welche Möglichkeiten sich daraus ergeben, schauen wir uns im vorliegenden Beitrag an.

Wenn Sie unter Access eine Datei in eine Tabelle importiert haben oder eine Verknüpfung auf Basis einer Datei angelegt haben oder Daten aus Tabellen oder Abfragen in eine Datei exportiert haben, finden Sie in aktuelleren Versionen von Access am Ende des Assistenten die Möglichkeit, die Import- beziehungsweise Exportschritte zu speichern. Anschließend können Sie diese, wenn Sie im Ribbon einen der Befehle **Externe Daten|Importieren und Verknüpfen|Gespeicherte Importe** oder **Externe Daten|Exportieren|Gespeicherte Exporte** aufrufen, den Dialog **Datentasks verwalten** öffnen (siehe Bild 1).

Hier finden Sie auf den beiden Registerkarten **Gespeicherte Importe** und **Gespeicherte Exporte** die bisher von dieser Datenbank aus gespeicherten Vorgänge. Klicken Sie doppelt auf einen dieser Vorgänge oder markieren Sie einen der Vorgänge und klicken dann auf die Schaltfläche Ausführen, wird der Vorgang mit den gespeicherten Parametern wiederholt.

In diesem Dialog können Sie die Bezeichnung des Vorgangs, den Namen der Quell- oder Zieldatei sowie die Beschreibung anpassen und vorhandene Vorgän-

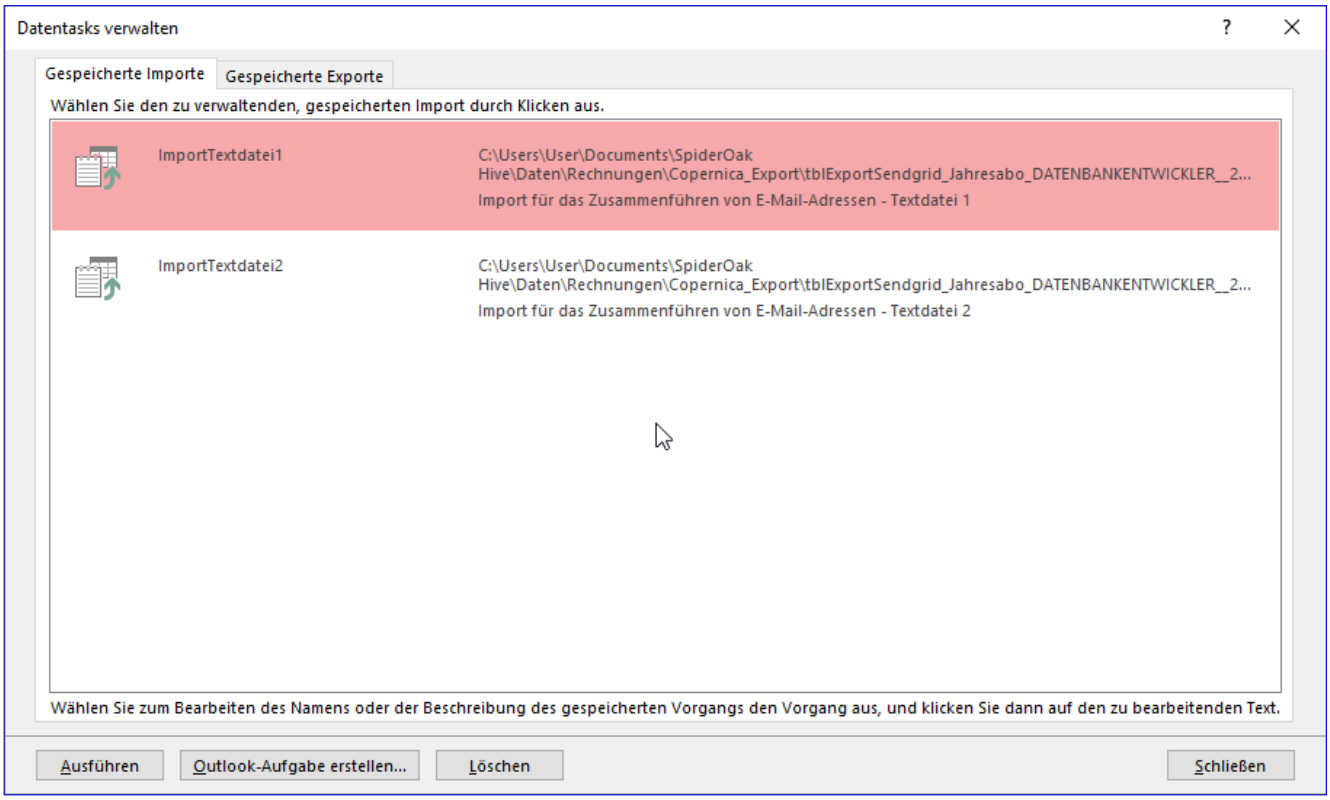

Bild 1: Übersicht der gespeicherten Import- und Export-Schritte

ge löschen. Außerdem können Sie über die Schaltfläche **Outlook-Aufgabe erstellen...** eine Outlook-Aufgabe mit dem geplanten Vorgang als Ziel erstellen.

Vorgang mit Outlook-Aufgabe

Wenn Sie eine solche Aufgabe erstellen, sieht diese wie in Bild 2 aus. Hier finden Sie lediglich eine Anleitung, welche Schritte auszuführen sind – wer also gedacht hat, er könnte damit die Ausführung eines der gespeicherten Vorgänge automatisch starten, wird enttäuscht.

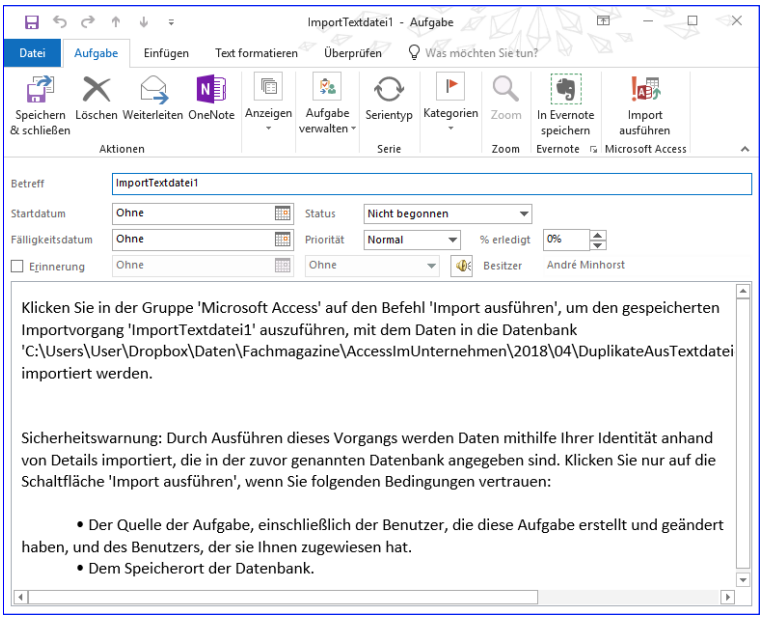

Bild 2: Aufgabe mit Anweisungen zum Ausführen eines gespeicherten Vorgangs

Vorgang anstoßen per VBA

Uns interessiert ohnehin mehr, wie wir per VBA auf die gespeicherten Vorgänge zugreifen können. Das interessanteste ist eine Möglichkeit, einen gespeicherten Import oder Export per VBA-Code anzustoßen. Dieser Befehl heißt **RunSavedImportExport** und ist Teil des Objekts **DoCmd**. Um einen gespeicherten Vorgang etwa namens **ExportTextdatei** anzustoßen, verwenden Sie also etwa den folgenden Aufruf:

```
DoCmd.RunSavedImportExport "ExportTextdatei"
```

Dies führt den Vorgang aus. Sollte es sich um einen Import handeln, werden eventuell bereits vorhandene Zielobjekte einfach überschrieben. Beim Export werden vorhandene Dateien gleichen Namens ebenfalls ignoriert.

Gespeicherte Vorgänge verwalten per VBA

Aber es gibt noch mehr Elemente in der Access-Bibliothek, mit denen Sie auf die gespeicherten Vorgänge zugreifen können. Die wesentlichen Elemente sind die Auflistung **ImportExportSpecifications** und die darin enthaltenen Elemente des Typs **ImportExportSpecification**.

Hier muss man nun aufpassen, um Missverständnisse zu vermeiden: Es gibt zwar während des Ausführens eines Imports oder Exports die Möglichkeit, eine Import- oder Export-Spezifikation anzulegen, die man später wieder laden oder als Teil eines Aufrufs etwa der Methode **TransferText** des **DoCmd**-Objekts nutzen kann und deren Daten in den Systemtabellen **MSysIMEXColumns** und **MSysIMEXSpecs** gespeichert werden. Mit der Auflistung **ImportExportSpecifications** referenzieren wir stattdessen die im oben erwähnten Dialog gespeicherten Importe und Exporte.

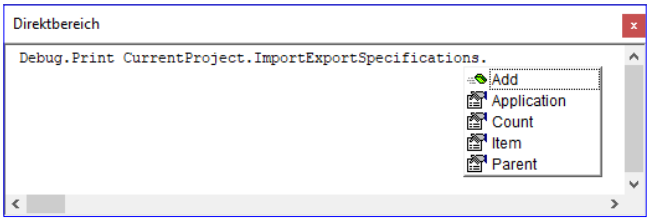

Bild 3: Elemente von **ImportExportSpecifications**

Auflistung ImportExportSpecifications durchlaufen

Dazu schauen wir uns zunächst diese Auflistung an. Diese ist ein Element des mit **CurrentProject** gelieferten Objekts. Die Eigenschaften können wir uns beispielsweise ansehen, wenn wir die Methode im Direktfenster eingeben. Nach Eingabe eines weiteren Punkts zeigt IntelliSense alle Methoden und Eigenschaften dieser Auflistung an (siehe Bild 3).

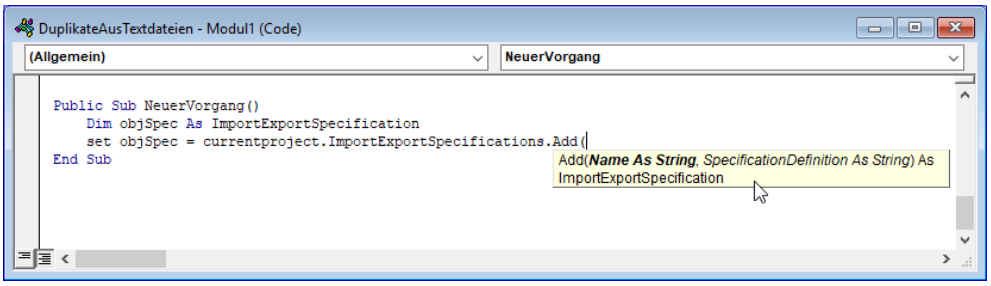

Bild 4: Hinzufügen eines neuen Vorgangs

Die folgende Prozedur durchläuft alle in dieser Auflistung gespeicherten Elemente und gibt diese im Direktfenster aus:

```
Public Sub VorgaengeDurchlaufen()
    Dim objSpec As ImportExportSpecification
    For Each objSpec In 7
                CurrentProject.ImportExportSpecifications
        Debug.Print objSpec.Name
    Next objSpec
End Sub
```

Mit unseren drei Beispiel-Vorgängen erhalten wir dann etwa die folgende Ausgabe:

```
ImportTextdatei1
ImportTextdatei2
ExportTextdatei
```

Neuen Vorgang anlegen

Wenn Sie per VBA einen neuen Vorgang anlegen wollen, nutzen Sie die **Add**-Methode der Auflistung **ImportExportSpecifications** wie in Bild 4. Der erste Parameter **Name** ist nicht erklärungsbedürftig, aber was sollen wir für den zweiten Parameter **SpecificationDefinition** angeben? Dafür schauen wir uns zunächst an, welchen Inhalt diese Eigenschaft für einen der bereits gespeicherten Einträge aufweist.

Dies erledigen wir wieder in der bereits vorgestellten Prozedur **VorgaengeDurchlaufen**, nur dass wir hier nach einer Eigenschaft suchen, welche dem Inhalt des Parameters **SpecificationDefinition** entspricht. Allerdings finden wir hier zumindest namentlich keine passende Eigenschaft (siehe Bild 5). Also probieren wir es einfach einmal mit der Eigenschaft **XML** aus. Und das liefert direkt einige Informationen – das Beispiel aus Listing 1 zeigt die Definition eines Imports im Überblick.

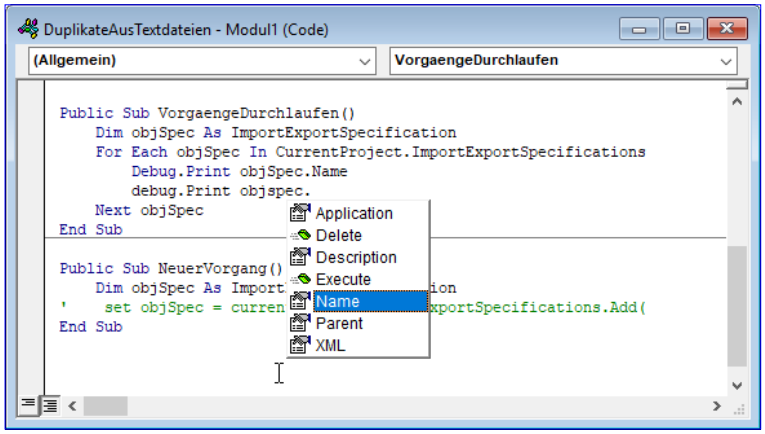

Bild 5: Eigenschaften eines **ImportExportSpecification**-Elements

XML-Definition eines Imports

Welche Elemente finden wir hier vor? Das Root-Element namens **ImportExportSpecification** enthält als wichtigstes Attribut den Pfad der Quell- oder Zieldatei. Das erste Unterelement heißt in unserem Fall **ImportText** und gibt so an, um was für einen Import es sich handelt. Es hat die folgenden Attribute:

- **TextFormat**: Gibt das Format an, hier **Delimited**.
- **FirstRowHasNames**: Gibt an, ob die erste Zeile die Feldnamen enthält, hier **true**.
- **FieldDelimiter**: Trennzeichen zwischen den einzelnen Feldern, hier das Semikolon (**;**)

Tabellenimport per VBA

Im Beitrag »Gespeicherte Importe und Exporte per VBA« zeigen wir, wie Sie etwa den Export oder Import von Textdateien per VBA steuern können. Mit diesen grundlegenden Techniken ausgestattet können wir noch einen Schritt weitergehen und beispielsweise auch komplette Tabelle aus einer anderen Datenbank importieren und diesen Vorgang per VBA ausführen. Wenn Sie etwa regelmäßig immer wieder die gleichen Tabellen aus einer Vorlagendatenbank in die Zieldatenbank importieren, müssen Sie normalerweise immer den entsprechenden Import-Assistenten dazu nutzen. Das können Sie von nun an auch mit ein paar Zeilen VBA-Code erledigen.

Wenn Sie einen Import von Access-Objekten wie hier von Access-Tabellen aus einer anderen Datenbank automatisieren wollen, können Sie dies ganz einfach erledigen.

Manueller Import als Voraussetzung

Als Erstes führen Sie den Import einmal mit dem dafür vorgesehenen Assistenten durch. Dazu wählen Sie im Ribbon den Eintrag **Externe Daten|Importieren und Verknüpfen|Neue Datenquelle|Aus Datenbank|Access** aus (siehe Bild 1).

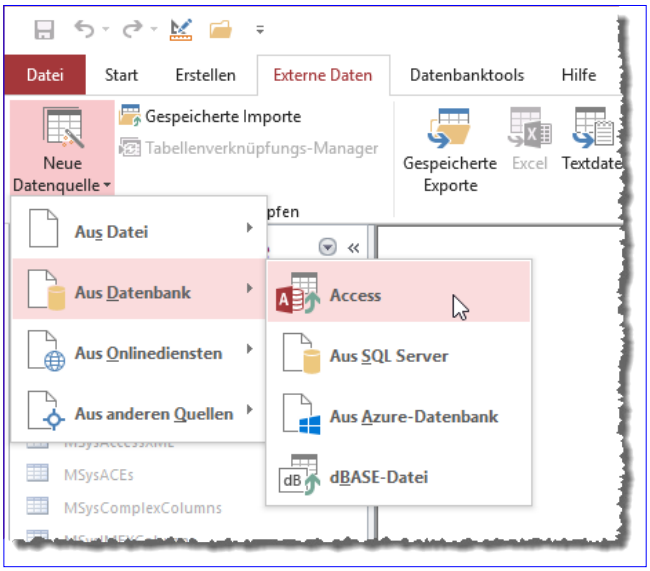

Bild 1: Aufruf des Importassistenten für Daten aus einer Access-Datenbank

Im nun erscheinenden Dialog **Externe Daten – Access-Datenbank** geben Sie oben zunächst den Namen der als Quelle zu verwendenden Access-Datenbank ein oder wählen diesen über die **Durchsuchen...**-Schaltfläche aus. Danach entscheiden Sie sich, ob Sie die Access-Objekte importieren wollen oder, wenn Sie nur Tabellen in den Import einbeziehen wollen, ob diese verknüpft werden sollen (siehe Bild 2). Klicken Sie dann auf die Schaltfläche **OK**. In diesem Beispiel wollen wir einige Tabellen der Datenbankdatei **Suedsturm.accdb** in die aktuell geöffnete Datenbank importieren.

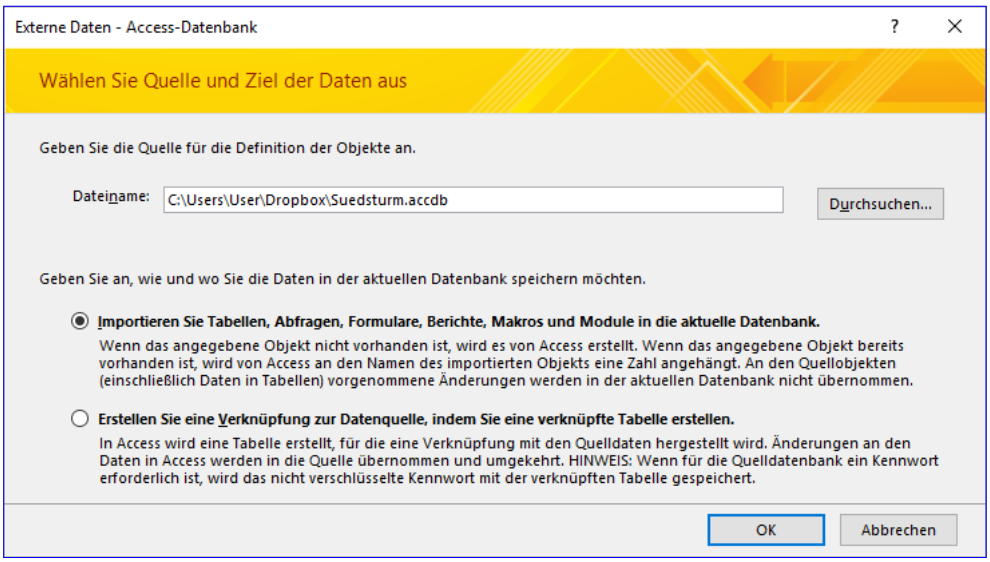

Bild 2: Auswahl der Import-Art und der Quelldatei für den Import

Im folgenden Dialog finden wir nun verschiedene Registerseiten mit den verschiedenen Objekttypen als Überschriften vor. Wir interessieren uns nur für die erste Seite mit den Tabellen. Hier können wir mit einem Klick auf die Schaltfläche **Optionen >>** weitere Optionen anzeigen, die im unteren Bereich des Dialogs eingeblendet werden. Wir behalten die Optionen bei und wählen alle Tabellen aus, die wir in die Zieldatenbank importieren wollen. Damit die Definitionen samt den enthaltenen Daten importiert werden, behalten wir im Bereich Tabellen importieren die Option **Definitionen und Daten** bei (siehe Bild 3).

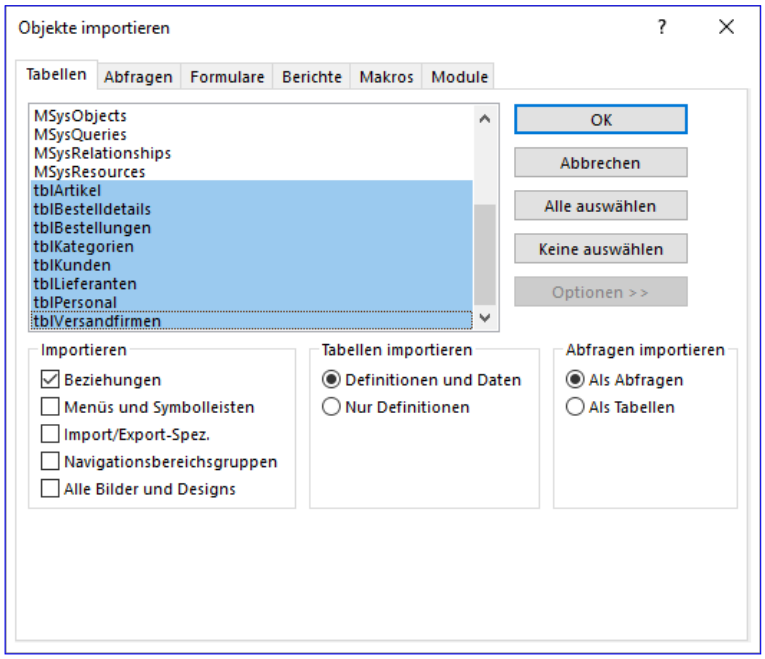

Bild 3: Dialog zum Auswählen der zu importierenden Daten und zu Einstellen weiterer Optionen

Es fehlt noch der letzte Schritt, der den Benutzer fragt, ob dieser die Importschritte speichern möchte (**Importschritte speichern**). Nach dem Aktivieren dieser Option blendet der Dialog einige weitere Optionen ein, von denen wir nur das Textfeld **Speichern unter:** nutzen wollen (siehe Bild 4). Hier geben wir einen sinnvollen Namen für den zu speichernden Import ein und speichern diesen dann durch einen Klick auf die Schaltfläche **Import speichern**.

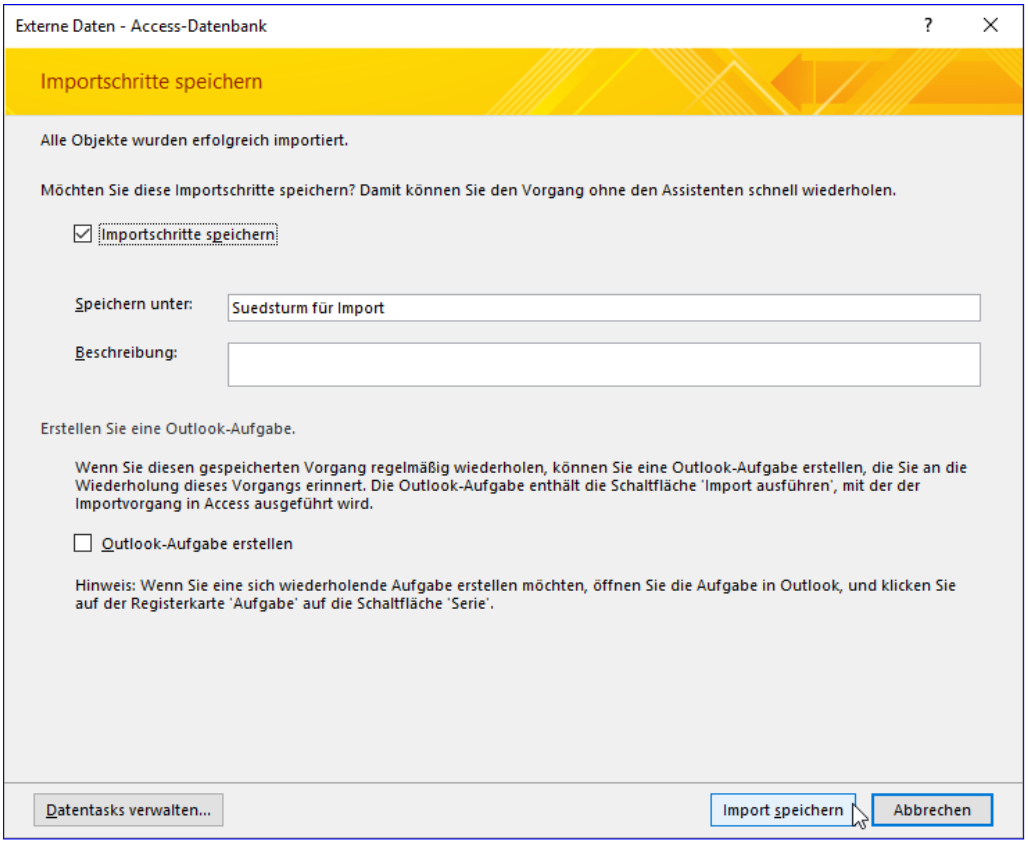

Bild 4: Optionen zum Speichern der Importvorlage

Gespeicherten Import aufrufen

Nachdem wir dies erledigt haben, können wir den gespeicherten Import über einen Dialog einsehen, den wir

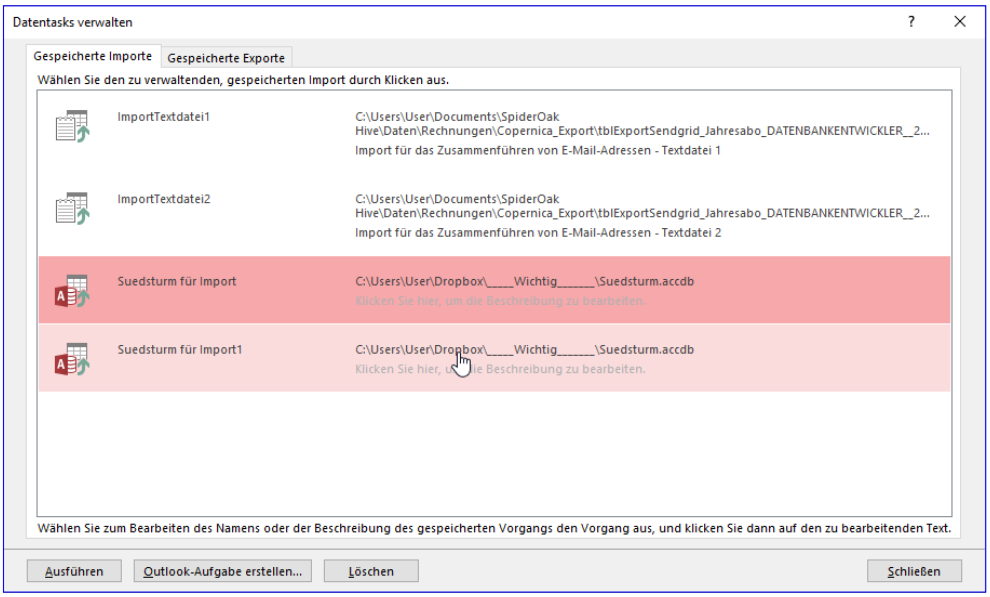

Bild 5: Die gespeicherte Importvorlage in der Liste der vorhandenen Importvorlagen

mit dem Ribbonbefehl **Externe Daten|Importieren und Verknüpfen|Gespeicherte Importe** einsehen können. Dieser öffnet den Dialog aus Bild 5, wo wir auch unseren gespeicherten Import vorfinden. Durch Markieren des gewünschten Eintrags und anschließendes Betätigen der Schaltfläche **Ausführen** können Sie diesen nun erneut ausführen. Die im Navigationsbereich angezeigten Tabellen werden nun nicht direkt aktualisiert, dies geschieht erst nach dem Schließen des Dialogs.

Wenn Sie einen Import mehrmals ausführen oder aus anderen Gründen bereits Tabellen vorhanden sind, welche die Namen der zu importierenden Tabellen aufweisen, werden diese unter einem neuen Namen importiert, der aus dem vorhandenen Namen plus einer Zahl als Zusatz besteht – bei **tblArtikel** also **tblArtikel1**, **tblArtikel2** und so weiter.

Direkter Zugriff auf den gespeicherten Import

Sie können nun direkt per VBA auf die Einstellungen des gespeicherten Imports zugreifen. Um etwa den Namen auszugeben, können Sie auch den Namen als Index für die Auflistung **ImportExportSpecifications** verwenden:

```
? CurrentProject.ImportExportSpecifications("Suedsturm für Import").Name
```

Die Definition erhalten Sie über die Eigenschaft **XML** dieses Elements der Auflistung **ImportExportSpecifications**. Sie sieht wie in Listing 1 aus.

Informationen der Import-Definition

Das XML-Dokument mit den Import-Daten enthält im Root-Element unter **Path** wieder den Pfad zu der Datei mit den zu importierenden Daten. Darunter folgt das Element **ImportAccess**. Dieses weist die folgenden Attribute auf, die im Wesentlichen die Informationen über die festgelegten Optionen enthalten:

- **ImportExportSpecs**: Import/Export-Spezifikationen importieren
- **MenusAndToolbars**: Menüs und Symbolleisten importieren
- **Relationships**: Beziehungen importieren
- **NavPane**: Navigationsbereichsgruppen importieren
- **StructureAndData**: Definitionen und Daten importieren oder nur Definitionen
- **QueriesAsTables**: Abfragen als Abfragen oder Tabellen importieren
- **Resources**: Alle Bilder und Designs importieren (nur für neuere Datenbank mit der Dateiendung **.accdb**)

```
<?xml version="1.0"?>
<ImportExportSpecification Path="C:\Users\User\Dropbox\_____Wichtig_______\Suedsturm.accdb"
        xmlns="urn:www.microsoft.com/office/access/imexspec">
    <ImportAccess ImportExportSpecs="false" MenusAndToolbars="false" Relationships="true" NavPane="false"
            StructureAndData="true" QueriesAsTables="false" Resources="false">
        <AccessObject Source="tblArtikel" ObjectType="Table"/>
        <AccessObject Source="tblBestelldetails" ObjectType="Table"/>
        <AccessObject Source="tblBestellungen" ObjectType="Table"/>
        <AccessObject Source="tblKategorien" ObjectType="Table"/>
        <AccessObject Source="tblKunden" ObjectType="Table"/>
        <AccessObject Source="tblLieferanten" ObjectType="Table"/>
        <AccessObject Source="tblPersonal" ObjectType="Table"/>
        <AccessObject Source="tblVersandfirmen" ObjectType="Table"/>
    </ImportAccess>
</ImportExportSpecification>
```

Listing 1: Definition des **ImportExportSpecification**-Objekts

Damit sind alle Optionen des Dialogs **Objekte importieren** abgedeckt.

Darunter folgen dann mehrere Elemente mit dem Namen **AccessObject**. Hier gibt es zwei Attribute:

- **Source**: Name des Access-Objekts
- **ObjectType**: Objekttyp, also **Table**, **Query**, **Form** für Tabellen, Abfragen, Formulare und so weiter

Eigenen Import für nur eine Tabelle bauen

Schauen wir uns an, ob wir den Import für eine einzige Tabelle selbst per VBA nachbauen können. Dazu benötigen wir nur wenige Zeilen Code, wir wie in Listing 2 sehen.

Hier deklarieren wir eine Variable namens **strXML** des Typs **String** sowie eine namens **objSpec** mit dem Typ **ImportExportSpecification**. Die Variable **strXML** füllen wir mit dem Code der zuvor erstellten Export-Spezifikation – mit dem Unterschied, dass wir diesmal nur eine Tabelle importieren wollen (hier **tblArtikel**).

objSpec füllen wir über die Methode **Add** der Auflistung **ImportExportSpecifications**, der wir als ersten Parameter den Namen der neu zu erstellenden Spezifikation übergeben (**Import_tblArtikel**) und als zweiten Parameter

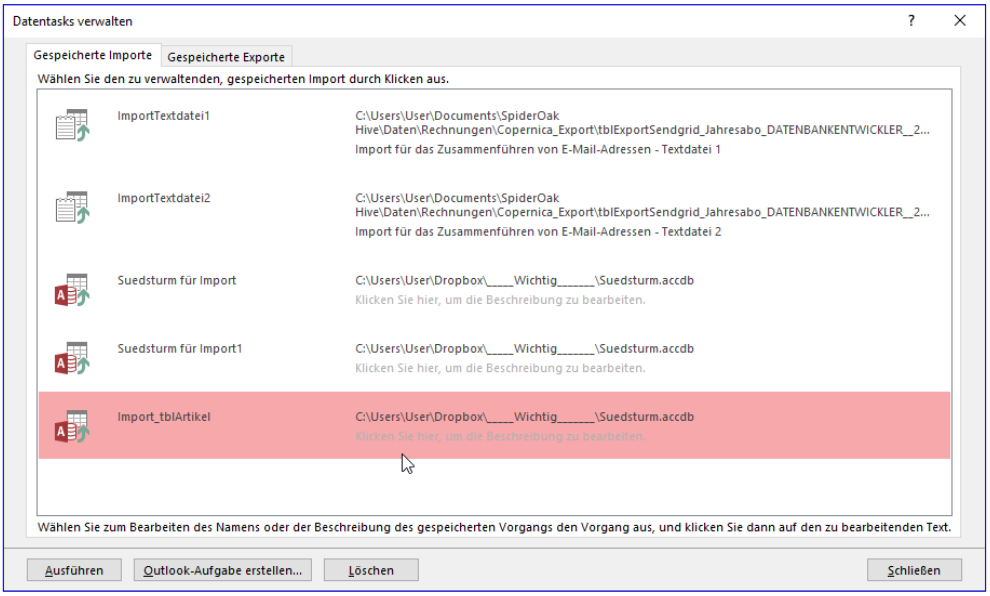

Bild 6: Die neu erstellte Importvorlage im Dialog **Datentasks verwalten**

Dateien schnell suchen mit Everything

Verlieren Sie auch hin und wieder den Überblick über Ihr Dateisystem und finden partout bestimmte Dateien nicht mehr auf? Dann ist neben der Windows-Suche ein Tool wie Everything ein guter Freund. Doch nicht nur die Anwendung mit ihrer Oberfläche selbst ist ein ausgereifter Helfer. Denn als Pluspunkt können Sie zusätzlich eine API-Schnittstelle verbuchen, welche sich aus Access heraus ansprechen lässt.

Everything

Gerade als Entwickler nennen Sie wahrscheinlich einen Rechner ihr Eigen, welcher mit Programmen, Tools, Dokumenten, Quellcode und anderen Dateien vollgestopft ist. Ich verzeichne aktuell über zwei Millionen Dateien auf meinem Hauptrechner. Da heißt es Ordnung halten und das Dateisystem gut strukturieren. Dennoch kommt es häufig vor, dass der Verbleib mancher Dateien schleierhaft bleibt – vor allem dann, wenn sie mehrfach in verschiedenen Verzeichnissen abgelegt wurden. Nun können Sie zum Auffinden selbstverständlich die Suche benutzen, die Windows von Haus aus mitbringt und, etwa unter **Windows 10**, die Suche namens **Cortona** befragen. **Windows Search** indiziert im Hintergrund fortwährend das System und füllt damit eine Datenbank, die im Ergebnis eine relativ schnelle Suche ermöglicht. Allerdings kann ich mich weder mit den möglichen Suchparametern, noch mit der Präsentation der Ergebnisse so richtig anfreunden. Deshalb gibt es eine Menge Tools, die die Lücke schließen und eine Alternative anbieten.

Everything von **voidtools.com** sticht dabei unter mehreren Aspekten heraus. Zwar gibt es mittlerweile diverse Freeware-Tools, die ebenso schnell arbeiten, weil sie als Basis die **MFTs** der Festplatten heranziehen, doch gerade die ausgefeilte **API**-Schnittstelle von **Everything** stellt ein Alleinstellungsmerkmal dar. Zudem haben Sie bei Installation die Wahl zwischen einer Systeminstallation und der **Portable Version**, bei der Sie den Download nur in ein Verzeichnis Ihrer Wahl entpacken.

Sollten Sie **Everything** noch nicht kennen, so lohnt sich zunächst ein Blick in die **FAQ** auf den Seiten von **voidtools**. Unter dem Menüpunkt **Support** finden Sie dort dann eine ausführliche Beschreibung aller Möglichkeiten und alles zum Umgang mit der Anwendung. Möchten Sie das Programm nutzen, so begeben Sie sich zu den **Downloads** und wählen unter verschiedenen Optionen die passende Version.

Es spielt, soviel vorweggenommen, keine Rolle, ob Sie die **32-Bit**- oder die **64-Bit**-Version nehmen. Normalerweise müssen Sie bei **Office**, das Sie sicherlich als 32-Bit-Version vorliegen haben, darauf achten, dass ein Programm, welches über eine **API**-Schnittstelle angesprochen wird, ebenfalls in 32 Bit vorliegt. Doch mit **Everything** wird gar nicht direkt kommuniziert, sondern über eine intermediäre DLL, die einen standardisierten Transportmechanismus von Windows nutzt, nämlich **IPC**. Der kann sich zwischen 32-Bit- und 64-Bit-Programmen gleichermaßen austauschen. Vielleicht liegt Ihnen die portable Version eher, als die Installation über eine **Setup**- oder **MSI**-Datei. Allerdings installiert letztere zunächst nicht einen Windows-Service, der das System automatisch im Hintergrund indiziert. Sie müssen dann die Anwendung erst unter Administratorrechten starten, was den bekannten **UAC-Prompt** unumgänglich macht. Erst dann scannt das Tool das System.

Wie funktioniert Everything?

Wollten Sie das Dateisystem mit Bordmitteln von Windows auslesen und sich dabei solcher **API**-Funktionen, wie **FindFileFirst**, et cetera, bedienen, oder unter VBA die **Dir**-Funktion nutzen, oder einen Verweis auf ein **File System Object** setzen, um mit jenem zu arbeiten, so können Sie auch auf einem schnellen Rechner bei

Millionen von Dateien eine lange Kaffeepause einlegen. Solche Verfahren sind zum Scannen des gesamten Dateisystems schlicht ungeeignet.

Everything nimmt sich jedoch direkt die **MFT** (**Master File Table**) einer Festplattenpartition zur Brust, die ein Verzeichnis aller Dateien und Ordner darstellt. Die Indizierung läuft deshalb unglaublich schnell ab. Bei den erwähnten zwei Millionen Dateien ist das hier in etwa 30 Sekunden erledigt. Ist die interne **Everything**-Datenbank einmal angelegt, so berücksichtigt das Tool nur noch Änderungen am Dateisystem.

Nach dem Start dauert dies dann gerade einmal zwei bis fünf Sekunden, je nachdem, wieviel sich seit dem letzten Indizieren am Dateisystem geändert hat. Voraussetzung allerdings ist, dass das Dateisystem mit **NTFS** partitioniert ist. **FAT** kann **Everything** natürlich ebenfalls lesen, doch dann kommen die altbekannten Windows-API-Verfahren zum Zug, die erheblich langsamer sind.

Sinnvoll ist es deshalb, eine Verknüpfung zum Tool in den **Autostart**-Ordner von Windows zu legen, wenn Sie die portable Version verwenden. Nach dem Start der Anwendung minimiert sich diese in den **Tray** von Windows. Mit Doppelklick auf das entsprechende Lupen-Icon öffnet sich dann die Oberfläche des Tools. Ist **Everything** als Service installiert, so ist das **Tray-Icon** automatisch immer aktiv. Solange die Anwendung läuft, bekommt sie alle Änderungen am Dateisystem mit und modifiziert demgemäß ihre Datenbank. Das lässt sich sogar Live beobachten, wenn Sie die Oberfläche von **Everything** geöffnet haben.

Das **GUI** von **Everything** zeigt Bild 1. Im Textfeld oben geben Sie einen Suchbegriff ein, und sofort filtert sich die Ergebnisliste unten entsprechend. Die Syntax für den Suchbegriff ist dabei weitaus mächtiger, als die von Windows. Hier können spezielle Platzhalter zum Einsatz kommen und allerlei Kombinationen derselben bis hin zu regulären Ausdrücken. Das Kombinationsfeld rechts neben dem Suchfeld erlaubt die einfache Filterung nach Kategorien, wie etwa Dokumente, Bilder oder Videos. Die Einträge in dieser Combobox können Sie zuvor selbst festgelegt haben.

Die Ergebnisliste ist nicht einfach nur eine Liste, sondern hat alle Eigenschaften der Windows-**Shell**. Das bedeutet,

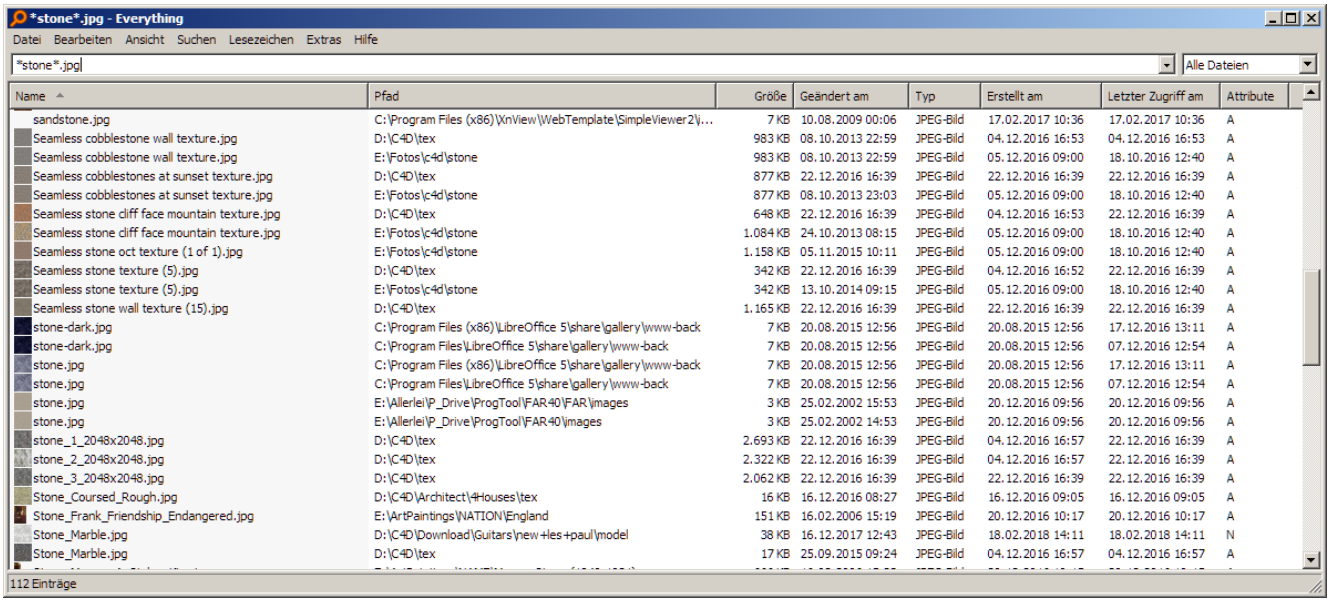

Bild 1: Das **GUI** von **Everything** wirkt auf den ersten Blick recht unspektakulär und eher spartanisch, hat unter der Haube aber viel zu bieten.

dass die gelisteten Dateien etwa dasselbe Kontextmenü anzeigen, wie der Explorer, und Kopier- sowie **Drag And Drop**-Operationen zulassen. Doch Gegenstand dieses Beitrags ist nicht die Beschreibung des Tools selbst, sondern seine Fernsteuerung. Zum Umgang mit der Anwendung finden Sie alle Informationen unter dem Menüpunkt **Support** auf **voidtools**. Sie werden staunen, was alles mit **Everything** möglich ist!

API-Schnittstelle

Unter dem Stichwort **SDK** können Sie auf **voidtools** die interaktive Schnittstelle zu **Everything** herunterladen. Die Funktionen der Schnittstelle sind zudem auf den Webseiten genau dokumentiert. Über diese Schnittstelle können Sie dann dieselben Vorgänge realisieren, wie auch manuell über die **GUI:** Sie setzen einen Suchbegriff ab, übergeben einige Suchoptionen, und erhalten als Resultat eine Liste mit den gefundenen Dateien oder Ordner. Das **GUI** von **Everything** muss während dieses Vorgangs nicht geöffnet sein.

Nun hatte **David Carpenter**, der Entwickler von **voidtools**, mit diesem **SDK** nicht unbedingt **Visual Basic**-Entwickler als Zielgruppe im Sinn. Die Dokumentation stützt sich deshalb auf **C**. Zwar finden Sie im Netz – genauer: im Forum zu **Everything** – auch eine VB-Portierung der **C-Header**-Dateien, doch die ist etwas veraltet und hat nur eine Teilmenge der Methoden übersetzt.

Also hat **Access im Unternehmen** die Fleißarbeit für Sie übernommen und praktisch alle **API**-Aufrufe nach **VBA** übersetzt, sodass Sie nun auch aus einer Datenbank heraus Zugriff auf **Everything** haben – mit den gleichen Möglichkeiten, wie ein **C++**-Programmierer.

Alles, was Sie dazu benötigen, ist eine aktuelle Version der **everything32.dll**, die als Bestandteil des **SDK** den **IPC**-Mittler zwischen **API** und **Everything**-Anwendung oder -Dienst darstellt. Sie finden die DLL direkt im Download zur Demodatenbank. Und schließlich brauchen Sie noch alle API-Deklarationen zur DLL, die sich im Klassenmodul **clsEverything** der Demo befinden. Die kleine Testprozedur in Listing 1 zeigt zunächst, wie Sie dieses Modul einsetzen. Hier soll **Everything** untersuchen, wo sich alle Dateien mit dem Namen **mscomctl.ocx** befinden. Die Prozedur rufen Sie einfach aus dem VBA-Direktfenster heraus auf.

Damit die Klasse funktioniert muss eine Instanz von **Everything** bereits auf dem Rechner laufen. Haben Sie es als Service installiert oder eine Verknüpfung zur **everything.exe** in den **Autostart**-Ordner von Windows gelegt, so ist dies bereits gewährleistet. Dann können Sie die erste Zeile der Prozedur löschen oder auskommentieren.

Die nämlich startet die im Datenbankverzeichnis befindliche **VBScript**-Datei **starteverything.vbs** über die API-Anweisung **ShellExecute**. Vielleicht kommt Ihnen das seltsam vor, weist doch **VBScript** keinerlei Methoden auf, die VBA selbst nicht auch enthielte. Man könnte also die Zeilen des Skripts ja auch in ein VBA-Modul schreiben?

Tatsächlich gibt es damit ein Problem: **Everything** soll mit erhöhten Administratorrechten laufen. Ein Start aus Access heraus würde dies nicht gewährleisten können, da Access selbst nur unter Benutzerrechten läuft. Diesem Umstand hilft das Skript ab. Aus Platzgründen listen wir es hier nicht auf, erläutern jedoch dessen Funktionsweise. Der Trick ist nämlich, dass das Skript automatisch zweimal startet. Es ruft sich selbst auf, nachdem es über das **WMI**, je nach Betriebssystem, erfahren hat, wo sich die für die Skript-Ausführung verantwortliche Datei **wscript.exe** befindet.

Und die wird dann ein zweites Mal ausgeführt, wobei nun als Argument ein **runas** übergeben wird. Das weist den Kommandozeileninterpreter an, das Skript mit erhöhten Rechten zu startet. Es kommt nun zum **UAC-Prompt**. In der Folge ermittelt die Routine den Ort zur **everything.exe** und startet eine Instanz derselben

```
Sub TestEverything(Optional ByVal Search As String = "mscomctl.ocx")
    Dim C As New clsEverything
    Dim ret() As Variant
    Dim sCols As String
    Dim i As Long, j As Long
    ShellExecute 0&, "open", CurrentProject.Path & _
        "\starteverything.vbs", vbNullString, CurrentProject.Path, 0&
    With C
        If .Ready Then
            Debug.Print .Version
            .CaseSensitive = False
            .ResultLimit = 2000
            .ResultProperties = EVERYTHING_REQUEST_FULL_PATH_AND_FILE_NAME Or _
                                EVERYTHING_REQUEST_DATE_CREATED
            .ShowMessages = True
            .SortOrder = EVERYTHING_SORT_PATH_ASCENDING
            If .Search(Search, ret, sCols) Then
                Debug.Print Replace(sCols, ";", vbTab)
                For i = 0 To UBound(ret, 2)
                    For j = 0 To UBound(ret, 1)
                        Debug.Print ret(j, i),
                    Next j
                    Debug.Print
                Next i
            End If
        End If
    End With
End Sub
```

Listing 1: Die Verwendung des Klassenmoduls **clsEverything** demonstriert diese Test-Routine

unter Zuhilfenahme des **Windows Scripting Host** und der Methode **Run**. **Everything** ist nun automatisch mit Admin-Rechten gestartet. Das Skript findet dabei selbständig heraus, ob die **32-Bit**- oder die **64-Bit-Version** von **Everything** installiert ist. Nach diesem Start kann die Klasse **clsEverything** in Gestalt der Objektvariablen **C**, auf die ein **With**-Block gesetzt wird, in Aktion treten.

Mit der Eigenschaft **Ready** ermittelt die Prozedur erst, ob **Everything** als Prozess auch wirklich korrekt ansprechbar ist. Dann werden einige Suchoptionen eingestellt. **CaseSensitive** steht auf **False**, wodurch nicht zwischen Groß- und Kleinschreibung unterschieden wird. Die maximale Anzahl an Rückgabewerten (**ResultLimit**) wird auf **2.000** festgelegt. Das ist für diesen Testaufruf eigentlich nicht notwendig, da Sie wohl kaum mehr Dateien mit dem Namen **mscomctl.ocx** finden werden. (Bei mir waren es aber immerhin 94.)

Übergeben Sie jedoch im Parameter **Search** der Prozedur etwa ***.jpg**, so ist die Wahrscheinlichkeit groß, dass dieses Limit überschritten wird. Dann stehen im Ergebnis nur die ersten **2.000** gefunden JPEG-Dateien.

Die nächste Einstellung ist da schon wichtiger. **ResultProperties** können Sie eine Kombination der im Klassenmodul definierten Enumerationskonstanten **eEVERYTHING_REQUEST** übergeben. Die steuern, was alles im Ergebnis enthalten sein soll. Die Standardeinstellung ist **FULL_PATH_AND_FILE_NAME**, die alle Dateien oder Ordner mit vollem Pfad zurückgibt. Weitere Angaben fehlen. Mit der zusätzlichen Einstellung **DATE_CREATED** weist das Ergebnis dann eine zusätzliche Spalte mit dem Erstelldatum der Dateien auf.

ShowMessages weist an, dass etwaige Fehlermeldungen ausgegeben werden sollen. Die Sortierung des Ergebnisses geschieht aufsteigend nach Namen, weil dies in **SortOrder** so eingestellt ist. Statt **PATH_ASCENDING** können Sie eine Menge anderer Sortierungen angeben. Mit **SIZE_DESCENDING** etwa würden die Ergebnisse absteigend nach Größe der Dateien sortiert.

Nach dieser (optionalen) Einstellung der Sucheigenschaften geht es mit dem Aufruf der Methode **Search** erst wirklich zur Sache. Dieser übergeben Sie im ersten Parameter den Suchausdruck, im zweiten ein Variant-Array (**ret**) und im dritten eine String-Variable **sCols**, welche die ermittelten Spalten im Ergebnis wiederspiegeln wird. Das Klassenmodul füllt diesen String selbst und gibt in unserem Beispiel über **Debug.Print** dies aus:

```
Kind; Path; File; DateCreated
```

Die erste Ergebnisspalte enthält demnach die Eigenschaft **Kind** der Dateien. Das ist ein Dateiattribut, welches aus einem Buchstaben besteht. Mit **F** ist eine Datei gekennzeichnet (**File**), mit **D** ein Ordner (**Directory**) und mit **V** ein Laufwerk (**Volume**). **Path** verweist auf den Dateipfad, **File** auf den Dateinamen. **DateCreated** gibt in der letzten Spalte das Erstelldatum aus. Nützlich ist dieser String dann, wenn Sie die Elemente des Rückgabe-Arrays etwa den Spalten einer Tabelle zuordnen möchten, wie wir noch sehen werden.

Das in **ret** zurückgegebene Array ist damit zweidimensional. Die zweite mit **UBound(ret, 2)** ermittelte Dimension ist die Anzahl der Zeilen, die erste die Zahl der Spalten darin. Zwei verschachtelte Schleifen mit den Zählern **i** und **j** auf diese Dimensionen durchlaufen nun das Array und geben die Werte im Direktfenster aus. Das abschließende Komma in **Debug.Print** weist VBA an, den nächsten Eintrag jeweils an die nächste TAB-Position zu stellen. Nach der inneren Schleife führt das **Debug.Print** ohne Parameter hingegen zum Zeilenumbruch.

Im Test dauerte der Aufruf von **Search** etwa ein bis zwei Sekunden. Vergegenwärtigen Sie sich, dass dabei das komplette Dateisystem durchsucht wird! Am schnellsten geschieht das, wenn Sie lediglich die Dateipfade zurückgeben lassen. Mit jedem weiteren Parameter steigt die Rechenzeit an. Besonders kritisch ist das beim Einsatz der Suchoption **SIZE**, die auch die Größe der Dateien ermittelt. Das übernimmt im Klassenmodul nämlich nicht **Everything** selbst, sondern eine Hilfsfunktion. Würden Sie als Suchausdruck etwa ***.*** angeben, so müsste für alle zwei Millionen im System gefundenen Dateien deren Größe ermittelt werden – vorausgesetzt, Sie hätten für **ResultLimit** einen entsprechenden Wert angegeben.

Abgesehen davon, dass ein Array mit 2 Millionen Werten wohl den Speicher von VBA sprengen würde, dürfte die Ermittlung dieser Dateigrößen locker eine Stunde benötigen. Ausprobiert haben wir das nicht.

ResultLimit ist übrigens auf **100.000** voreingestellt, wenn Sie die Eigenschaft nicht selbst setzen. Trotz der Performance von **Everything** kann die Ermittlung des Ergebnisses, je nach Situation, etwas dauern, weshalb es sich im Zweifel gut macht, vor dem Aufruf von **Search** ein **DoCmd.Hourglass True** abzusetzen, um die Sanduhr erscheinen zu lassen.

Klassenmodul clsEverything

Die Hauptaufgabe des Moduls ist die Übersetzung der **C-Header** von **Everything** nach VBA. Im Kopf finden Sie deshalb eine Menge von Konstantendefinitionen und API-Deklarationen. Die werden dann von den Prozeduren des Moduls verwendet. Außerhalb des Moduls brauchen Sie sich keine Gedanken über diese API-Deklarationen zu machen, die ja auch alle als **Private** daherkommen.

Im **Initialize**-Ereignis der Klasse (siehe Listing 2) werden einige Voreinstellungen für die Suche vorgenommen. Das Ergebnis-Limit steht auf **100.000**, die Rückgabespalten bestehen nur aus Pfad und Name der Datei, die Sortierung geschieht nach Namen, und zwischen Groß- und Kleinschreibung soll nicht unterschieden werden. Diese Einstellungen speichern die **Member**-Variablen des Moduls, welche namentlich jeweils mit **m_** beginnen.

Damit die API-Deklarationen greifen, muss die für die Kommunikation mit Everything verantwortliche Datei **everything32.dll** geladen werden. Das muss nur einmal geschehen. **GetModuleHandle** fragt ab, ob das Modul

bereits im Prozessraum von Access steht. Dann ist alles ok und **hMod** ist ungleich Null. Ansonsten schaut die Routine erst einmal nach, ob die **everything32.dll** überhaupt im Datenbankverzeichnis liegt (**CurrentProject.Path**), was die Voraussetzung für die Klasse ist. Sie können diesen Zweig natürlich modifizieren und die DLL etwa in einem anderen Verzeichnis unterbringen. Ist die Datei vorhanden, so lädt **LoadLibrary** sie und speichert das erhaltene **ModulHandle** in **hMod**.

```
Private Sub Class_Initialize()
    m_sort = EVERYTHING_SORT_PATH_ASCENDING
    m_requestprops = EVERYTHING_REQUEST_FILE_NAME Or _
        EVERYTHING_REQUEST_PATH
    m_msg = True
    m_limit = 100000
    m_case = False

    hMod = GetModuleHandle("everything32")
    If hMod = 0 Then
        If Len(Dir(CurrentProject.Path & "\everything32.dll")) = 0 Then
            MsgBox _
                "Die Datei everything32.dll muss im Datenbankverzeichnis liegen!", _
                vbCritical, "Everything"
        Else
            hMod = LoadLibrary(CurrentProject.Path & "\everything32.dll")
        End If
    End If
End Sub
```

Listing 2: Im **Initialize**-Ereignis werden Suchparameter voreingestellt und die **everything**-DLL geladen

Selbst dann, wenn die DLL geladen ist, muss zusätzlich überprüft werden, ob das Anwendungsfenster von **Everything** schon aufgebaut ist. Über dieses Fenster geschieht nämlich die Kommunikation in Form von **IPC-Messages**. Sichtbar muss es dazu allerdings nicht sein. Es reicht, wenn **Everything** als Icon im **Tray** zu finden ist. Die Eigenschaft **Ready** des Klassenmoduls sucht nun erst nach dem Fenster und anschließend nach dem Zustand von **Everything**, welcher über die API-Funktion **Everything_IsDBLoaded** abgefragt werden kann. Starten Sie **Everything** etwa über das besprochene **VBS-Script**, so beginnt **Everything** zunächst mit der Indizierung der modifizierten Dateien des Systems, um diese Änderungen in seiner Datenbank abzulegen. Das kann einige Zeit in Anspruch nehmen und erst nach Ablauf dieser gibt **Everything_IsDBLoaded** einen anderen Wert aus, als Null:

```
If Everything_IsDBLoaded Then
    Ready = True
Else
    MsgBox "Die Everything-Datenbank ist noch nicht" & _
       "vollständig geladen. Bitte später nochmals " & _
       "versuchen.", vbCritical, "Everything"
End If
```

Die ganzen **Property**-Prozeduren des Moduls übernehmen im Wesentlichen die Aufgabe, die übergebenen Parameter in **Member**-Variablen abzuspeichern, die dann erst beim Aufruf von **Search** ausgewertet werden. Und diese Prozedur ist das Kernstück des Moduls. Deklariert ist sie so:

```
Function Search( _
    ByVal QueryString As String, _
    ResultSet() As Variant, _
    ByRef Columns As String) As Boolean
```

In **QueryString** übergeben Sie den Suchausdruck, in **ResultSet** das undimensionierte Rückgabe-Array vom Typ **Variant**, und in **Columns** eine String-Variable, die anschließend die Spalten des Ergebnisses enthält. Gültig ist das Ergebnis indessen nur dann, wenn die Funktion

True zurückgibt. Haben Sie etwa **ShowMessages** auf **False** eingestellt, dann unterbleiben Fehlermeldungen. In diesem Fall gibt nur der Rückgabewert der Funktion Aufschluss über den Erfolg der Suche.

In der Testprozedur von **Listing 1** wird dieser Rückgabewert auch ausgewertet.

Search stellt nun anhand der über die Eigenschaft **ResultProperties** eingestellten Suchoptionen über die **Member**-Variable **m_requestprops** die Spalten des Ergebnisses zusammen:

```
Columns = "Kind;Path;File;"
If (m_requestprops And REQUEST_EXTENSION) Then
    bExt = True: Columns = Columns & "Extension;"
If (m_requestprops And REQUEST_DATE_CREATED) Then
     bDateCr = True: Columns = Columns & "DateCreated;"
If (m_requestprops And REQUEST_DATE_MODIFIED) Then
    bDateMod = True: Columns = Columns & "DateModified;"
...
```

Die Boolean-Variablen **bExt**, **bDateCr**, **bDateMod** et cetera steuern den weiteren Ablauf in der Prozedur. Der Suchausdruck wird nun an **Everything** übergeben:

```
Everything_SetSearchW (StrPtr(QueryString))
ret = Everything_QueryW(1)
```

QueryW ermittelt hier, ob der übergebene Suchausdruck korrekt ist und den Spezifikationen von **Everything** genügt. Nur dann steht der Rückgabewert in **ret** auf **1**. Tatsächlich führt dieser Aufruf bereits zur Suche und das Ergebnis kann anschließend ausgewertet werden:

```
cnt = Everything_GetNumResults
```

In **cnt** steht jetzt die Zahl der ermittelten Dateien oder Ordner. Ist sie gleich **0**, so wird die Prozedur verlassen. Andernfalls fährt Sie mit der Überführung der Ergebnisse in das Array **ResultSet** fort:

```
If cnt > 0 Then
    Search = True: ReDim ResultSet(n - 1, cnt - 1)
    For i = 0 To cnt - 1
        (...)
    Next i
End If
```

Das Array wird anhand der Ergebniszahl **cnt** dimensioniert. Das gilt für die zweite Dimension. Die erste bestimmt die Zahl der Spalten, die vorher in **n** bei der Zusammenstellung derselben ermittelt wurde.

Den Dateinamen erhält die Routine nun etwa so:

```
sFile = String(260, 0)
ret = Everything_GetResultFullPathNameW( _
          i, StrPtr(sFile), 260&)
If ret > 0 Then sFile = Left(sFile, ret)
```

Die String-Variable **sFile** muss zuerst mit **Null-Characters** gefüllt werden, wie dies häufig bei API-Funktionen erforderlich ist. Der String wird nicht direkt übergeben, sondern als Zeiger per **StrPtr**. In **ret** steht dann die Buchstabenzahl, die die API-Funktion errechnete. **Left** schneidet die nun überflüssigen **Null-Characters** ab.

Übrigens fährt die Prozedur mit den weiteren Schritten grundsätzlich nur dann fort, wenn **ret** hier auf **>0** steht. Denn der Datei- oder Ordnername sind das Mindeste, was im Ergebnis auftauchen muss.

Soll auch das Erstelldatum abgefragt werden, so tritt dieser Zweig in Aktion:

```
If bDateCr Then
    ret = Everything_GetResultDateCreated(i, FilTime)
    If ret <> 0 Then
        ResultSet(col, i) = APITimeToDate(FilTime)
    End If
    col = col + 1
End If
```

ActiveX und COM ohne Registrierung verwenden

ActiveX-Komponenten müssen, so sie unter VBA ansprechbar sein sollen, im System registriert werden. Dies erfordert administrative Rechte unter Windows, weshalb viele Administratoren solche externen Komponenten leider gar nicht gern sehen. Dass Sie manche ActiveX-Dateien jedoch auch ganz ohne Registrierung verwenden können, zeigen die Techniken dieses Beitrags.

VBA und die COM-Registry

Es gibt zunächst unter VBA zwei Möglichkeiten, um ein Objekt einer ActiveX-Komponente zu erzeugen. Die eine nutzt die Methode **CreateObject**, der Sie die sogenannte **ProgID** übergeben und das Resultat einer Objektvariablen zuweisen. Bei dieser kann es sich um den allgemeinen Typ **Object** handeln. Dennoch lassen sich die Funktionen des Objekts namentlich über den Punkt-Operator ansprechen, obwohl **Object** selbst ja keinerlei Methoden aufweist. Hier verwenden Sie **Late Binding**.

Late deshalb, weil Windows hier den Methodennamen erst beim Aufruf auswerten muss, um an den Funktionszeiger zu gelangen, der schließlich angesprungen wird. Dieser Vorgang benötigt einige Zeit, weshalb **Late Binding** die Performance erheblich herabsetzt. Dafür aber brauchen Sie immerhin keinen Verweis auf die Komponente zu setzen!

Anders bei der zweiten Methode, die sich des Operators **New** bedient. Nach dem Schreiben dieses Ausdrucks in eine VBA-Routine listet **IntelliSense** gleich alle Klassen auf, die sich damit instanziieren lassen. Dabei kann VBA aber nur auf jene zugreifen, die es in den Verweisen des aktuellen VBA-Projekts findet. Demzufolge muss auf die gefragte ActiveX-Komponente ein Verweis gesetzt sein. VBA kennt dann aus dieser **Type Library** den genauen Aufbau des Objekts, weshalb Sie es es einer Objektvariablen des dezidierten Klassentyps zuweisen können. Nun haben Sie **Early Binding** vor sich. Technisch bedeutet dies, dass VBA nun bereits beim Kompilieren die Funktionszeiger ermitteln kann und eben diese in das **PCode**-Kompilat einsetzt. Das steigert die Performance ungemein.

Doch beiden Methoden zum Erzeugen des Objekts ist eines gemein: Sie funktionieren nicht ohne die Registrierung der ActiveX-Komponente unter Windows. Der Verweis auf eine Komponente macht es nötig, dass deren **Type Library** (Bibliothek) und deren **Interfaces** (Klassen) in den Zweigen **HKEY_CLASSES_ROOT** und den Unterschlüsseln **typelib**, **interface** und **clsid** eingetragen sind. Für **CreateObject** braucht es zusätzlich Einträge unter **HKEY_CLASSES_ROOT** direkt, wobei die **ProgID** selbst den Schlüssel stellt und wiederum auf eine zugehörige **clsid** verweist.

Beispiel: Sie möchten den Ordnerpfad zum Windows-Desktop ermitteln. Dafür können Sie etwa die **Shell**-Bibliothek verwenden. Sie wählen diese in der Liste der Verweise unter dem Eintrag **Microsoft Shell Controls And Automation** aus. Die Bibliothek zeigt sich dann im Objektkatalog unter der Rubrik **Shell32**. Mit diesem Code-Schnipsel erhalten Sie dann den Pfad:

```
Dim oShell As Shell32.Shell
Set oShell = New Shell32.Shell
Debug.Print oShell.Namespace(0&).Self.Path
```

Ohne VBA-Verweis auf die **Shell**-Bibliothek geht es mit **Late Binding** auch so:

```
Dim oShell As Object
Set oShell = CreateObject("Shell.Application")
Debug.Print oShell.Namespace(0&).Self.Path
```

Was passiert hier? VBA sucht in der Registry unter **HKEY_CLASSES_ROOT** und dem Schlüssel **Shell.Application**. Dort wertet es den Unterschlüssel

CLSID aus, der die **GUID {13709620-C279-11CE-A49E-444553540000}** zurückgibt. Im Folgenden schreitet es zum Zweig **HKEY_CLASSES_ROOT\clsid** und sucht dort nach dieser **GUID**. Es wird fündig und liest nun den Unterschlüssel **InProcServer32** aus, der auf die ActiveX-Datei **%SystemRoot%\system32\shell32.dll** verweist. Die kann VBA nun theoretisch schon laden und das Objekt über die **GUID** und die allen COM-Komponenten eigene API-Funktion **DLLGetClassObject** erzeugen. Allerdings weiß es dann noch nichts über das Objekt selbst. Dazu muss es einen zweiten Unterschlüssel **Typelib** auswerten, der die **GUID** zur **Shell**-Bibliothek zurückgibt. Diese Bibliothek muss es nun ebenfalls laden, um dort nach der Klasse mit der zuvor ermittelten **CLSID** zu suchen. Aus **HKCR\TypeLib\<GUID>\1.0\0\win32** erfährt es, dass die Bibliothek sich konkret unter **C:\Windows\SysWOW64\shell32.dll** befindet. In der geladenen Bibliothek klappert es nun die sogenannten **CoClasses** ab, die in ihr enthalten sind. Das sind all jene, die sich als Objekt neu erzeugen lassen.

Es findet die zur **CLSID** gehörige Klasse und erfährt nun erst, welches Standard-**Interface** ihr eigen ist. Dabei sollte es sich ab Windows 7 um **IShellDispatch5** oder **IShellDispatch6** handeln. Nun nimmt es sich die Definition dieses Interfaces vor und kennt damit dessen Methoden. Das Interface selbst holt es sich über einen Aufruf der Standard-**COM**-Methode **QueryInterface** auf das zuvor per **DLLGetClassObject** erzeugte Objekt. Erst jetzt kann VBA das Objekt einer Variablen zuweisen.

Sie sehen, dass VBA unter der Haube allerlei zu bewältigen hat, wenn es um **Late Binding** geht. Dabei ist der Vorgang in Wirklichkeit noch deutlich komplizierter, als geschildert. Nachvollziehbar: die Performance geht dabei in die Knie.

Doch die beiden Methoden zum Erzeugen von Objekten sind ja eigentlich nicht Thema dieses Beitrags. Die Erläuterungen sind aber für das Verständnis der weiteren Ausführungen nützlich.

ActiveX-Objekte per API erstellen

Der Sachverhalt ist jener: Es gibt nur wenig, was Sie unter **C++** realisieren können, was sich nicht auch unter **Visual Basic 6** oder VBA umsetzen ließe! Der Aufwand dafür allerdings ist ungleich höher. Denn mit normalem VBA-Code kommen Sie nicht weiter. Gerade dann, wenn es um **COM** geht, benötigen Sie in der Regel wenigstens eine Bibliothek, wie die **oleexp.tlb**, welche die Definition der grundlegenden **Interfaces** enthält. Wir stellten diese Bibliothek bereits vor (**Shortlink 1096**) und werden weitere Beiträge zum Umgang mit ihr veröffentlichen. Hier aber geht es ja gerade darum, die Zahl der Verweise im VBA-Projekt minimal zu halten, weshalb in der vorliegenden Lösung auf externe Bibliotheken komplett verzichtet wurde.

Zum Einsatz kommt dafür eine Handvoll standardisierter Windows-API-Funktionen und ... etwas **Assembler-Code**! Ja, Sie hören richtig! Tatsächlich kann mit einigen Tricks auch unter VBA Maschinen-Code ausgeführt werden. Sehr komfortabel ist das indessen nicht, aber diese Code-Teile sind in der Lösung auch nicht sonderlich umfangreich.

Untergebracht ist alles im Modul **mdlCOMLoaderAsm** der Demodatenbank **Regless.accdb**. Die Routine zum Erzeugen eines Objekts nennt sich darin **GetCOMInstance**. Sie übergeben ihr als Parameter einerseits den Pfad zur ActiveX-Datei – die nicht registriert sein muss! – und andererseits die **ProgID** des gewünschten Objekts. Alternativ können Sie statt der **ProgID** auch direkt die **CLSID** der Komponente angeben, wenn Sie diese kennen.

Wollten wir hier die genaue Funktionsweise der komplexen Prozedur erläutern, so könnte damit sicher die gesamte Ausgabe gefüllt werden. Deshalb beschränken wir uns nur auf die prinzipiellen Techniken. Ansonsten verwenden Sie das Modul einfach unbesehen und importieren es bei Bedarf in Ihre eigenen Datenbankprojekte. Es weist keine weiteren Abhängigkeiten auf und lief

bisher in der vorliegenden Version in unseren eigenen Projekten anstandslos. Die Anwendung erfolgt also nach diesem Schema:

```
Dim O As Object
Dim sFile As String
Dim sProgIDCLSID As String

sFile = "c:\Windows\SysWOW64\shell32.dll"
sProgIDCLSID = "Shell"
Set O = GetCOMInstance(sFile, sProgIDCLSID)
Debug.Print Typename(O)   '-> IShellDispatch6
O.TueDiesUndDas ...
```

Die Objektvariable **O** soll das Objekt aufnehmen, welches wir instanziieren möchten. In **sFile** steht der volle Pfad zur ActiveX-Datei die das Objekt beherbergt. **sProgID-CLSID** speichert die **ProgID** oder wahlweise die CLSID der Komponente. Hier verwenden wir die **shell32.dll** und in ihr die Objektklasse **Shell**. Damit erhalten wir über den Aufruf von **GetCOMInstance** dasselbe Objekt, wie mit dem eingangs vorgestellten Code. **Typename** gibt korrekt die **Interface**-Klasse **IShellDispatch6** zurück. Statt **TueDiesUndDas** könnten Sie nun eben den Pfad zum Desktop ermitteln:

```
Debug.Print O.Namespace(0&).Self.Path
Set O = Nothing
```

Die Routine läuft schneller ab, wenn Sie statt der **ProgID** die **CLSID** einsetzen:

```
sProgIDCLSID = "{13709620-C279-11CE-A49E-444553540000}"
```

Sie finden diese in der Registry im Zweig **HKCR\Shell.Application\CLSID**. Die Funktion muss dann nicht erst umständlich die **ProgID** in die benötigte **CLSID** umwandeln. Des Weiteren können Sie den vollen Pfad zur Datei auch abkürzen:

```
sFile = "shell32.dll"
```

Das funktioniert in folgenden Fällen: Die Datei befindet sich entweder im Verzeichnis der Datenbank, oder im Suchpfad von Windows. Der besteht aus den Windows-Verzeichnissen und allen, die in der Umgebungsvariablen **PATH** abgelegt sind.

Dass das Ganze funktioniert, können Sie übrigens mit der Funktion **GetDesktopPath** im Modul **mdlTest** der Demodatenbank nachvollziehen.

Einen dritten optionalen Parameter von **GetCOMInstance** haben wir noch unterschlagen: Steht **ClearCache** auf **True**, dann ermittelt die Routine aus einer **ProgID** die **CLSID** immer neu.

Wird **False** übergeben (Standard), so speichert das Modul die ermittelte **CLSID** beim ersten Mal ab und verwendet sie fortan bei jedem Aufruf der Funktion mit derselben **ProgID**.

Das beschleunigt die Sache dann deutlich. (Allerdings werden Sie die Instanziierung von Objekten auch nicht etwa fortlaufend in einer Schleife ausführen, so dass der Performance-Gewinn wohl nicht sonderlich zu Buche schlägt.)

GetCOMInstance-Funktion im Detail

Die erste Aktion besteht darin, die ActiveX-Datei zu laden. Das übernimmt an sich die API-Funktion **LoadLibrary**, welcher Sie den Pfad übergeben und ein **Handle** auf das Modul erhalten. Das aber ist nicht immer notwendig, denn die Datei kann sich bereits im Prozessraum von Access befinden. Bei der **shell32.dll** etwa ist dies der Fall. Access benötigt sie selbst für seine internen Funktionen. Dann aber reicht die API-Funktion **GetModuleHandle** aus:

```
hModule = GetModuleHandle(sFile)
```

Bei API-Funktionen ereignet sich in der Regel kein VBA-Fehler, wenn falsche Parameter übergeben wurden.

Stattdessen definiert der Rückgabewert den Erfolg. Hier verhält es sich so, dass die in **hModule** (Long-Wert) gespeicherte Rückgabe **0** ist, wenn die Funktion fehlschlug. Dann kommt **LoadLibrary** zum Zug:

```
If hModule = 0 Then hModule = LoadLibrary(sFile)
```

Findet die die Funktion auch hier die Datei nicht, weil deren Pfad nicht stimmt oder sie sich nicht im Windows-Suchpfad befindet, so kommt es zum letzten Versuch:

```
If hModule = 0 Then
  hModule = LoadLibrary(CurrentProject.Path & "\" & sFile)
End if
```

Hier wird das Datenbankverzeichnis auf Existenz der Datei befragt. Schlagen alle Methoden fehl, so löst die Routine einen VBA-Fehler aus und verlässt die Prozedur:

```
If hModule = 0 Then
  Err.Raise vbObjectError, , sFile & " module not loaded"
  Exit Function
End If
```

Wir gehen davon aus, dass die Datei erfolgreich geladen wurde. Nun geht es an den nächsten Test, der untersucht, ob die Datei ActiveX-kompatibel ist und die die API-Funktion **DllGetClassObject** exportiert:

```
pFunc = GetProcAddress(hModule, "DllGetClassObject")
If pFunc = 0 Then
    Err.Raise vbObjectError, , _
        sFile & " does not export DllGetClassObject"
    FreeLibrary hModule
    Exit Function
 End If
```

Der API-Funktion **GetProcAddress** übergeben Sie das zuvor erhalten **Handle** (**hModule**) und den Namen einer Funktion. In der Long-Variablen **pFunc** ist nun der Funktionszeiger gespeichert. Steht hier eine Null, so unterstützt die Datei diese Funktion nicht und es kommt ebenfalls zu einer Fehlermeldung und zum Verlassen der Routine.

Anhand des Funktionszeigers kann im Folgenden die Funktion **DllGetClassObject** im Prinzip aufgerufen werden. Das allerdings kann VBA im Unterschied zu C++ keinesfalls! Hier kommt eine Hilfsfunktion **CallPointerASM** zum Einsatz, die Ihnen garantiert kryptisch vorkommen wird, weil sie auf Umwegen **Assembler**-Code generiert, der von der API-Funktion **CallWindowProc** angesprungen wird. Sie ist so deklariert:

```
Private Function CallPointerASM( _
    ByVal fnc As Long, ParamArray Params()) As Long
    ...
End Function
```

Mit dem Parameter **fnc** setzen Sie den Funktionszeiger und in **Params()** eine beliebige Anzahl von zu übergebenden Parametern, da diese Variable vom Typ **ParamArray** ist. Die Prozedur erzeugt nun mit API-Funktionen, wie **VirtualAlloc**, einen Speicherblock im **RAM** und bringt dort die Parameter unter. Genauer **pusht** es diese auf den virtuellen **Stack**. Abschließend gibt es diesen Speicherblock auch gleich wieder frei (**VirtualFree**). Zuvor jedoch wird die API-Funktion **CallWindowProc** zweckentfremdet. Eigentlich ist sie dafür gedacht, um die Standard-Routine eines Windows-Fensters anzuspringen. Möglicherweise kennen Sie sie deshalb schon von **Subclassing**-Routinen, wo **Windows-Messages** an eine per **AddressOf** erhaltene VBA-Modulfunktion geleitet werden und von dort aus weiter an die ursprüngliche Funktionsadresse über **CallWindowProc**, damit das Windows-Fenster weiter funktioniert, wie gedacht. Tatsächlich aber weiß Windows nicht, ob die Funktionsadresse die eines Fensters ist, oder irgendetwas Anderes. Wir verwenden stattdessen nun den eben erzeugten Speicherblock als Adresse, welcher den **Assembler**-Code enthält. Windows führt nun unmittelbar etwa die Funktion **DllGetClassObject** aus!

Es kann sein, dass Sie ähnlichen Code im Netz finden, wenn Sie nach **CallPointer VB** googeln. Die meisten Lösungen verwenden dabei einfach ein Byte-Array, in dem der **Assembler**-Code abgespeichert wird. An **CallWindowProc** wird dann die Adresse des Byte-Arrays geleitet. Das hat früher auch funktioniert, seit **VBA 7** (**Office 2010**) jedoch nicht, weil nun aus Sicherheitsgründen der allen Variablen zugrundeliegende RAM-Speicher nicht mehr mit dem Attribut **PAGE_EXECUTE_READWRITE** ausgestattet ist, um Ungemach zu verhindern. Deshalb wurde die Routine **CallPointerASM** komplett neu entwickelt.

Zurück zur Prozedur **GetCOMInstance**! Bevor die API-Funktion **DllGetClassObject** in ihr angesprungen werden kann, muss erst noch die **CLSID** bereitstehen. Denn die API-Funktion hat folgende prototypische Definition:

```
Declare Function DllGetClassObject( _
        Byval CLSID As GUID, _
        ByVal IID As GUID, _
        pResultingInterface As Long) As Long
```

Sie benötigt also die **CLSID** des Objekts, welches man erhalten möchte. Außerdem muss die **IID** eine **GUID** aufweisen, die zu einem **Interface** gehört, welches die Funktion zurückgeben soll. Für ein Objekt reicht hier das Standard-**Interface IUnknown** aus, für die Instanziierung muss aber das **Interface IClassFactory** her, weil erst dieses die endgültige Funktion **CreateInstance** beherbergt.

Die Aufgabe, aus einer **ProgID** eine **CLSID** zu machen ohne dass die Klasse registriert ist (!), ist allerdings alles andere, als trivial. Die Hilfsroutine **ScanCoClasses** übernimmt sie. Ihre Darstellung sprengte trotz der nur etwa 40 Zeilen endgültig den Rahmen dieses Beitrags. Sie simuliert über benutzerdefinierte Typen Interfaces der Abteilung **ITypelib** und **ITypeInfo**, deren Methoden sie abermals über **CallPointerASM** aufruft. Dabei ermittelt sie alle **CoClasses** der ActiveX-Datei und deren zugehörigen **CLSIDs**, sowie **ProgIDs**. Nachdem diese in einem Array abgespeichert sind, kann die Hauptroutine dieses durchlaufen und mit der übergebenen **ProgID** vergleichen. Trifft ein Element zu, so haben Sie die **CLSID** zur **ProgID**. **DllGetClassObject** kann nun ausgeführt und eine Instanz des gewünschten Objekts über die **CreateInstance**-Methode von **IClassFactory** erhalten werden.

Das ist alles nicht so einfach. Glückwunsch, wenn Sie das Kapitel bis zu diesem Punkt durchgehalten haben! Es ist tatsächlich nur für die Hartgesottenen unter Ihnen gedacht. Für den praktischen Einsatz des Moduls ist dieses Wissen nicht notwendig.

Beispiel ActiveX-Datei zum Ähnlichkeitsvergleich
Im Download zum Beitrag finden Sie eine ActiveX-Datei **mSimilarity.dll**, die unter anderem Strings auf Ähnlichkeit untersuchen kann und aus eigener Feder stammt. Das ist ein Erfordernis, das gerade in Datenbanken oft benötigt wird. Aus einer Adressentabelle filtern Sie etwa jene Namen heraus, die bei falscher Schreibweise eines Suchbegriffs die ähnlichsten Treffer enthalten. Eine ähnliche Komponente kam im Beitrag **Fehlertolerantes Suchen** (**Shortlink 733**) bereits zum Einsatz. Dort ist auch der Umgang mit ihren Funktionen erschöpfend beschrieben. Die aktuelle DLL ist eine Weiterentwicklung mit mehr Methoden und verbesserter Performance. Eine bessere Alternative werden Sie wahrscheinlich vergeblich im Netz suchen... Und, bevor es wieder zu Nachfragen per E-Mail kommt: Ja, die DLL ist komplett **Freeware** und darf ohne weitere Hinweise in eigenen Projekten eingesetzt und weitergegeben werden, ob privat oder kommerziell. Eine kleine Angabe zum Autor **mossSOFT** an der einen oder anderen Stelle kann jedoch nicht schaden.

Sie registrieren die DLL über die begleitende Datei **register_mSimilarity.bat**, welche Sie im Administrator-Kontext ausführen. Bei Weitergabe Ihres Datenbankprojekts entfiele dieser Schritt, weil wir Klassen der DLL

regless instanziieren werden. Für die Demodatenbank ist die Registrierung allerdings nötig, weil deren Routinen teilweise ihren Bibliotheksverweis ansprechen.

Hier ein Code-Schnipsel, welches die Ähnlichkeitsmethode **JaroWinkler** demonstriert:

```
Dim C As CSimilarity
Dim S1 As String, S2 As String
Set C = New CSimilarity
S1 = "Trowitzsch"
S2 = "Trowitsch"
Debug.Print C.JaroWinkler(S1,S2)   '-> 92
```

Die gefragte Klasse in der DLL nennt sich **CSimilarity**. Eine Objektinstanz auf sie landet in der Variablen **C**. Zwei Begriffe in **S1** und **S2** weisen kleine Unterschiede in der Schreibweise auf. (Die erste ist korrekt!) **JaroWinkler** vergleicht beide und gibt **92** als Ähnlichkeitsfaktor aus. **100** wäre der Wert für komplette Übereinstimmung. Als Limit für hinreichende Ähnlichkeit nehmen Sie übrigens einen Wert zwischen **70** und **80**. Alles darunter ist eher fragwürdig.

Die Routine in Listing 1 testet nun die Performance der Methoden. Neben **JaroWinkler** zieht sie zusätzlich die Alternative **Ratcliff** heran. Die Variable **n** speichert jeweils den Ähnlichkeitswert. Beide Methoden werden in einer Schleife 1 Million Mal aufgerufen. Die dafür benötigte Zeit messen der **VBA.Timer** und die Variable **T**. Das Ergebnis auf meinem **I7-5820**-Rechner:

```
0,84375 s       92
0,9414063 s     91
```

JaroWinkler ist also geringfügig schneller, als **Ratcliff**. Die zurückgegebenen Ähnlichkeitswerte sind bei beiden fast identisch. Die Performance ist auch nicht schlecht: Jede Schleife auf die Funktionen wird in weniger als einer Sekunde durchlaufen. Sie können damit also eine Million Strings pro Sekunde vergleichen. Die Dauer, die

```
Sub TestSimilarity1()
    Dim C As New CSimilarity
    Dim I As Long, n As Integer
    Dim S1 As String, S2 As String
    Dim T As Single

    S1 = "Trowitzsch": S2 = "Trowitsch"
    n = C.JaroWinkler(S1, S2)
    n = C.Ratcliff(S1, S2)
    DoEvents
    T = VBA.Timer
    For I = 1 To 1000000
        n = C.JaroWinkler(S1, S2)
    Next I
    Debug.Print VBA.Timer - T, n

    T = VBA.Timer
    For I = 1 To 1000000
        n = C.Ratcliff(S1, S2)
    Next I
    Debug.Print VBA.Timer - T, n

    Set C = Nothing
End Sub
```

Listing 1: Dies ist der Standard-Code bei registrierter ActiveX-DLL

VBA für die Schleife selbst benötigt, spielt übrigens keine Rolle. Ein Test auf eine Quasi-Null-Schleife in der folgenden Form gibt hier **0,0078 ms** für die Ausführungszeit aus. Das sind eine Milliarde Durchläufe pro Sekunde! VBA ist also alles andere als langsam:

```
For i = 1 To 1000000
    n = n
Next i
```

Nun ersetzen wir in einer weiteren Routine die Instanziierung des Klassenobjekts per **New** durch die **regless**-Variante per **GetCOMInstance**, wie in Listing 2. Sie übergeben hier in **sFile** den Pfad zur Datei **mSimilarity.dll** im Datenbankverzeichnis und verwenden die **ProgID CSimilarity**. Den **CLSID**-Cache des Moduls **mdlCOMLoaderASM** leeren Sie mit Übergabe von **True** an den dritten Parameter **ClearCache**. Der Rest der Prozedur ist

```
Sub TestSimilarity2()
    Dim sFile As String
    Dim C As CSimilarity
    Dim sCLSIDProgID As String
    Dim I As Long, n As Integer
    Dim S1 As String, S2 As String
    Dim T As Single

    S1 = "Trowitzsch": S2 = "Trowitsch"
    sCLSIDProgID = "CSimilarity"
    sFile = CurrentProject.Path & "\mSimilarity.dll"
    Set C = GetCOMInstance(sFile, sCLSIDProgID, True)
    '= CSimilarity
    n = C.JaroWinkler(S1, S2)
    n = C.Ratcliff(S1, S2)
    DoEvents
    T = VBA.Timer
    For I = 1 To 1000000
        n = C.JaroWinkler(S1, S2)
    Next I
    Debug.Print VBA.Timer - T, n

    T = VBA.Timer
    For I = 1 To 1000000
        n = C.Ratcliff(S1, S2)
    Next I
    Debug.Print VBA.Timer - T, n

    Set C = Nothing
End Sub
```

Listing 2: Und das ist die **regless**-Variante mit **GetCOMInstance**

identisch mit dem aus **TestSimilarity1**. Wenig erstaunlich: die Performance ist auch nicht schlechter!

Das liegt daran, dass erstens die Instanzierung des Objekts aus der Messung herausgelassen wurde – dafür messen wir gesondert etwa **90 ms** – und zweitens die Objektvariable **C** als **CSimilarity**-Typ deklariert ist, wodurch implizit **Early Binding** auf die Verweisbibliothek **mSimilarity** zum Einsatz kommt.

Ohne Verweis können Sie **C** nur noch **As Object** deklarieren, wodurch der Ausführungsmodus nach **Late Binding** wechselt. Die hier nicht gelistete Routine **TestSimilarity3** vollzieht dies. Für die Performance messen wir nun folgende Werte:

```
3,929688 s      92
4,199219 s      91
```

Tja, die ist nun um Größenordnungen geringer, als bei **Early Binding**. Aber immerhin lassen sich auch damit etwa **250.000** Strings pro Sekunde vergleichen, was wohl noch hinzunehmen ist.

Es gibt jedoch noch eine alternative Methode, um die Funktionen der DLL aufzurufen und das **Late Binding** zu umgehen. Die allerdings ist wieder ähnlich kompliziert, wie die Routine **GetCOMInstance** ...

Direkter Methodenaufruf in ActiveX-Objekten

Das Modul **mdlCOMLoaderASM** enthält noch eine zusätzliche Prozedur **CallCOMFunction**. Ihr übergeben Sie ein Objekt, die Adresse einer Methode und die ihr zu übergebenden Funktionsparameter. Als Ergebnis (Variant) erhalten Sie entweder nichts bei einer **Sub**-Methode oder díe Rückgabe der Methode bei einer **Funktion**. Listing 3 bildet die Prozedur ab.

Im Kern bedient sie sich der API-Funktion **DispCallFunc**. Die erwartet einen Zeiger auf das Objekt (**InterfacePointer**), den sogenannten **VTable-Offset** der gewünschten Methode, die Aufrufkonvention der DLL und die Parameter in zwei Arrays, wobei das erste die Werte abbildet, das zweite deren Datentypen. Sie verstehen nur Bahnhof? Das ist nicht verwunderlich, weil wir es hier wieder mit Interna von Objekten und deren **IDispatch**-Interface zu tun haben.

Ein Objekt stellt imgrunde nur einen **Benutzerdefinierten Typ** dar. In diesem steht ganz zu Beginn eine Tabelle mit den Funktionszeigern auf die Methoden (**VTable**). Sie erhalten einen Speicherverweis auf diese Tabelle über die VBA-Funktion **ObjPtr()** auf das Objekt. In dieser Tabelle stehen jedoch nicht etwa die Namen der Methoden,