HAUFE.

Ausgabe 06/2014

ACCESS IM UNTERNEHMEN

FLEXIBLE SUCHFUNKTION

Eine Suchfunktion, die mit nur einem Textfeld arbeitet und die gezielte Suche nach den Inhalten bestimmter Felder bietet? Mit der Sie aber auch alle Felder gleichzeitig durchsuchen können? Dies alles zeigen wir Ihnen ab S. 56 und S. 66.

In diesem Heft:

OUTLOOK-MAILS IN DATENBANK DROPPEN

Ziehen Sie Mails einfach von einem Outlook-Ordner in den anderen, um die Inhalte in einer Datenbank zu speichern.

SEITE 26

SCHRIFTEN IN DER BENUTZEROBERFLÄCHE

Erfahren Sie alles Wissenswerte über den Einsatz der verschiedenen Schriftarten in Formularen und Berichten.

SEITE 2

FORMULARDATEN IN VORLAGEN SPEICHERN

Speichern Sie den aktuellen Formularinhalt zwischen und stellen Sie ihn per Mausklick wieder her.

SEITE 10

Mat-Nr. H1583-5086

ERGONOMIE	Schriften in Formularen und Berichten	2
FORMULARE UND STEUERELEMENTE	Formularinhalte speichern und wieder abrufen	10
	Kombinationsfelder mit vielen Einträgen	20
INTERAKTIV	Mails per Drag and Drop in der Datenbank speichern	26
VBA UND PROGRAMMIERTECHNIK	VBA unter Access mit 64 Bit	34
	Das Windows Management Instrumentarium	40
	Mit Arrays arbeiten	49
	Suchen mit der clsFlexSearch-Klasse	56
LÖSUNGEN	Flexible Suche in Formularen	66
SERVICE	Impressum	U2
	Editorial	1

DOWNLOAD

Die Downloads zu dieser Ausgabe finden Sie wie gewohnt unter:

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

Benutzername: **((((((&lsp dĞp ?`mllclrcl qgafr`_p'**

Kennwort: **(((((**

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

Impressum

ISBN 978-3-8092-1496-0 ISSN: 1616-5535 Mat.-Nr. 01583-5086

Access im Unternehmen
© Haufe-Lexware GmbH & Co. KG, Munzinger Str. 9, 79111 Freiburg
Kommanditgesellschaft, Sitz Freiburg
Registergericht Freiburg, HRA 4408
Komplementäre: Haufe-Lexware Verwaltungs GmbH, Sitz Freiburg,
Registergericht Freiburg, HRB 5557; Martin Laqua
Geschäftsführung: Isabel Blank, Markus Dränert, Jörg Frey, Birte Hackenjos, Randolf Jessl, Markus Reithwiesner, Joachim Rotzinger, Dr. Carsten Thies

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

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

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

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

Auslieferung und Vertretung für die Schweiz: H.R. Balmer AG, Neugasse 12, CH-6301 Zug, Tel. 041/711 4735, Fax: 041/711 0917
Das Update-Heft und alle darin enthaltenen Beiträge und Abbildungen sind urheberrechtlich geschützt. Jede Verwertung, die nicht ausdrücklich vom Urheberrechtsgesetz zugelassen ist, bedarf der vorherigen Zustimmung des Verlags. Das gilt insbesondere für Vervielfältigungen, Bearbeitungen, Übersetzungen, Mikroverfilmung und für die Einspeicherung in elektronische Systeme.
Wir weisen darauf hin, dass die verwendeten Bezeichnungen und Markennamen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen. Die im Werk gemachten Angaben erfolgen nach bestem Wissen, jedoch ohne Gewähr. Für mögliche Schäden, die im Zusammenhang mit den Angaben im Werk stehen könnten, wird keine Gewährleistung übernommen.

Wer sucht, der findet!

Die beste Datenbank nützt nichts, wenn man die Daten nicht findet. Deshalb finden Sie in Access im Unternehmen immer wieder neue Vorschläge für Suchfunktionen. In dieser Ausgabe zeigen wir uns hier sehr flexibel: Mit einer Suchfunktion, die wie Google und Co. mit nur einem Suchfeld auskommt, aber dennoch sowohl die gezielte Suche in bestimmten Feldern als auch die Suche über alle Daten des Formulars ermöglicht.

Dieser Suchfunktion widmen wir gleich zwei Beiträge: Der erste heißt **Flexible Suche in Formularen** (ab S. 66) und zeigt, wie Sie die komplett in einer einzigen Klasse enthaltene Suchfunktion in bestehende Formulare integrieren. Außerdem erfahren Sie dort, wie Sie die Suchfunktion nutzen. Der Beitrag **Suchen mit der clsFlexSearch-Klasse** zeigt ab S. 56, wie wir diese Suchfunktion programmiert haben. Dabei stellen wir nicht nur den für die Suche verwendeten Algorithmus vor, sondern auch noch die Techniken, mit denen Sie diese in Form eines einzigen Klassenmoduls implementieren.

Wer wirklich professionelle Access-Anwendungen erstellen möchte, kommt nicht umhin, die Schriftarten in seinen Formularen und Berichten zu prüfen. Immerhin steht und fällt damit die Lesbarkeit der enthaltenen Texte! Worauf es dabei ankommt, zeigt Sascha Trowitzsch in seinem Beitrag **Schriften in Formularen und Berichten**, den Sie ab S. 2 lesen können. Der gleiche Autor steuert noch einen weiteren Beitrag zu dieser Ausgabe bei. Er heißt **Das Windows Management Instrumentarium** und stellt darin ab S. 40 eine Schnittstelle vor, mit der Sie auf einfache Weise Informationen über System, Hardware oder Software erhalten.

Wer eine Anwendung nutzt, in der er oft ähnliche Daten in ein Formular eingeben muss, profitiert von **Formularinhalte speichern und abrufen** (ab S. 10). Darin stellen wir eine Funktion vor, die per Mausklick den kompletten Inhalt des aktuellen Formulars in einer speziell dafür angelegten Tabelle speichert. Die auf diese Weise angelegten Vorlagen kann der Benutzer dann per Kombinationsfeld auswählen und so die gespeicherten Daten in einen neuen Datensatz eintragen.

Gelegentlich werden Sie auf die Anforderung stoßen, Kombinationsfelder mit sehr großen Datenmengen zu füllen. Dabei sprechen wir von mehr als 65.536 Einträgen, was der maximal möglichen Anzahl entspricht. Der Beitrag **Kombinationsfelder mit vielen Einträgen** zeigt ab S. 20, wie Sie mit solchen Kombinationsfeldern umgehen. Um es kurz zu machen: Durch Abfangen der Eingabe des Benutzers reduzieren wir die Anzahl der anzuzeigenden Datensätze und verhindern so nicht nur das Erreichen der maximalen Anzahl von Einträgen, sondern tragen auch etwas zur einer besseren Performance bei.

Wer seinen E-Mail-Verkehr über Microsoft Outlook (ab Version 2007) abwickelt, möchte vielleicht die Inhalte bestimmter E-Mails in einer Datenbank speichern – beispielsweise, um die Nachrichten von Kunden direkt im Bearbeitungsformular für den Kunden verfügbar zu machen. Der Beitrag **E-Mails per Drag and Drop in der Datenbank speichern** zeigt ab S. 26, wie Sie dies auf einfachste Art und Weise erledigen: durch Drag and Drop der gewünschten E-Mail in einen dafür vorbereiteten Mail-Ordner. Eine in Outlook angelegte Ereignisprozedur erkennt dies und erfasst den Inhalt der E-Mail in einer Tabelle der angegebenen Datenbank.

Außerdem schauen wir uns im Beitrag **VBA unter Access mit 64 Bit** ab S. 34 an, welche Probleme bei der Verwendung von Access in der 64bit-Version auftreten können. Einige davon können sie nicht beheben – so bleibt etwa die Nutzung des TreeView-Steuerelements den Anwendern mit der 32bit-Version vorbehalten. Aber für den Zugriff etwa auf API-Funktionen sind nur einige Änderungen im Code notwendig, die dieser Beitrag im Detail erläutert.

Schließlich zeigen wir im Beitrag **Mit Arrays arbeiten** ab S. 49, wie Sie Arrays etwa zum Debuggen auf verschiedene Arten ausgeben können, prüfen, ob ein Array überhaupt gefüllt ist oder wie ein Array mit den Daten einer Tabelle befüllt wird.

Und nun: Viel Spaß bei der Lektüre der neuen Ausgabe!

M. Forster

Ihr Michael Forster

Schriften in Formularen und Berichten

Unter Textverarbeitungssystemen sind Erwägungen zu den verwendeten Schriften selbstverständlich. Hier bestimmen neben der Lesbarkeit auch Gesamtlayout und Schriftbild den Einsatz verschiedener Schriftarten. Welche Überlegungen für die Anforderungen an Schriften unter Access-Formularen und -Berichten anzustellen sind, soll in diesem Beitrag diskutiert werden.

Beispieldatenbank

Die Beispiele dieses Artikels finden Sie in der Datenbank **1409_Fonts.mdb**.

Was unterscheidet Schriften?

Holen wir uns zunächst ins Bewusstsein, durch welche Parameter eine Schrift im Groben charakterisiert ist.

Technisch gesehen gibt es einmal die Raster-Fonts und dann die Vektor-Fonts. Raster-Fonts sind Schriftdateien, in denen die einzelnen Zeichen als Bitmaps gespeichert sind, wobei meist für verschiedene Größen auch verschiedene Bitmaps eingebettet sind. Die grafische Skalierbarkeit dieser Fonts ist daher eingeschränkt und das Schriftbild bescheiden. Zum Einsatz kommen diese Fonts nur noch selten.

In Vektor-Fonts wird jedes Zeichen durch einen Satz von Liniensegmenten definiert, deren Stützpunkte jeweils die Hülle ergeben, innerhalb derer ausgefüllt wird. Sie lassen sich beliebig verlustlos skalieren. Die beiden Hauptvertreter sind **TrueType** (**.ttf**) und **OpenType**-Schriften (**.otf**).

Stilistisch gesehen gibt es unterschiedliche Kategorien. Die eine ist die Zeichenbreite. Im Allgemeinen hat jedes Zeichen eine eigene Breite – ein **i** ist nun mal schmaler als ein **W**. Schriften, die dem gerecht werden, sind Proportionalschriften.

Für spezielle Zwecke jedoch werden auch nichtproportionale Schriften benötigt, bei denen jedes Zeichen dieselbe Breite aufweist, erkennbar oft schon am Namen der Schrift, der häufig ein **Mono** enthält, abgeleitet von **Mono Space**.

Rein optisch ist das weniger schön, aber immer dann sinnvoll, wenn untereinanderstehende Zeilen Bezug zueinander haben. Etwa in Rechnungen, wo Zahlen untereinander addiert werden. Oder auch der Übersichtlichkeit halber in Programmcode, wie in den Code-Modulen von VBA. An dieser Stelle, da es zum Thema passt, ein kleiner Einschub:

Die Schriftart des VBA-Editors können Sie in dessen Optionen-Dialog einstellen – s. Bild 1. Testen Sie einmal, wie der Programmcode je nach eingestellter Schriftart wirkt und welche Varianten die Lesbarkeit fördern. Es stellt sich heraus, dass jenen mit fester Zeichenbreite meist der Vorzug zu geben ist. Es ist common sense für Code-Editoren, solche Schriftarten mit **fixed pitch** zu verwenden. Tatsächlich ist das Kombinationsfeld für die Schriftarten im Optionen-Dialog auch gefiltert; nicht alle installierten Schriftarten erscheinen dort.

Allerdings erschließt sich uns nicht, nach welchen Kriterien die Schriften hier gefiltert sind. Es sind keineswegs nur nichtproportionale Schriften aufgeführt. Zu allem Überfluss stürzt VBA, und damit Access, übrigens auch noch bei der Auswahl einiger Schriftarten ab – ein Bug, der schon seit Urzeiten von VBA mitgeschleppt wird. Sollte dies passiert sein, so bleibt Ihnen nur die Korrektur über die Registry, weil VBA jedes Mal erneut abstürzt, sobald es geöffnet wird. An den Optionen-Dialog

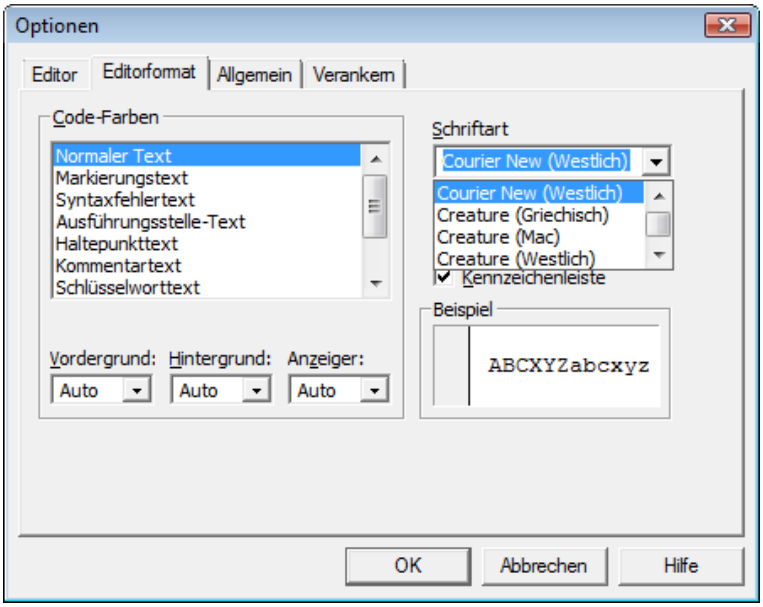

Bild 1: Auswahl der Schriftart für den VBA-Code im Optionen-Dialog

kommen Sie somit nicht mehr heran. Ändern Sie dann in der Registry diesen Zweig

```
HKEY_CURRENT_USER\Software\Microsoft\VBA\7.0\Common
```

und überschreiben Sie den Wert für **FontFace** mit einer Schriftart wie **Courier New**.

Zurück zu den Schriftartkategorien. Neben der Breite der Zeichen ist deren Gestalt relevant. So gibt es die zwei Haupttypen Serifen- und serifenlose Schriften. Serifen sind die Füßchen am Ende der Balken. Sie haben den Sinn, bei Textblöcken optisch virtuelle Leitlinien für das Auge zu bilden und damit den Lesefluss zu verbessern.

In Printmedien sind diese Schriften, deren Hauptvertreter die Times darstellt, nach wie vor weit verbreitet.

Die Füßchen haben jedoch im Auslauf so kleine Abmessungen, dass sie auf Monitoren auch bei den heute gängigen Auflösungen und Pixelgrößen nicht hinreichend dargestellt werden können. Entweder werden sie krakelig oder durch das Antialiasing verwaschen, was störend wirkt. Aus diesem Grund werden hier für normalen Text eher serifenlose Schriften bevorzugt. Daher gilt dies auch für das vorliegende Dokument: **Access im Unternehmen** verwendet die serifenlose **Helvetica**, die unter anderem E-Books einfach lesbarer macht.

Oder fänden Sie diese Zeilen hier etwa besser zu lesen, wenn die Schriftart **Times New Roman** angewandt würde?

Bild 2: Einzelne Dimensionsteile einer Schriftart

Es bleibt festzuhalten, dass am Monitor die serifenlosen Schriften günstiger sind, weshalb Microsoft diese unter Access auch als Standard festgelegt hat. Einst war es **MS Sans Serif**, dann **Tahoma**, **Arial**, und seit Office 2007 sind **Calibri** und **Segoe UI** das Maß der Dinge.

Kaum der Erwähnung wert ist die Tatsache, dass auch Schriftarten wie **Handwriting**, barocke Stile oder andere verschnörkelte Typen auf dem Monitor in einer nüchternen Umgebung wie Access nichts zu suchen haben.

Mit dem Problem der ungenügenden Auflösung hat auch eine andere Kategorie zu kämpfen: die Lage der Schrift. In Printmedien werden bestimmte Ausdrücke etwa dadurch hervorgehoben, dass sie kursiv gedruckt sind. Auf Monitoren führt die Steillage von Balken jedoch zu Sprüngen der Pixel, also einer Treppe statt einer Linie, weshalb Antialiasing notwendig wird.

Dieses wiederum ist nur ein Notbehelf, weil die Schrift dann verwaschener wird, da die Interpolation zu Pixeln führt, die grau statt schwarz sind. Je kleiner die Schrift, desto mehr tritt dieser Effekt in Erscheinung.

Aus diesem Grund verwendet auch **Access im Unternehmen** keine kursiven Zeichen, sondern nimmt zur Hervorhebung ausschließlich fett gedruckte Worte. Auch Sie sollten den Einsatz von kursiver Schrift unter Access genau abwägen. Brauchen Sie sie wirklich?

Soweit zu den Grundkategorien von Schriften. Bliebe noch das eigentliche Design. Die Kunst des Schriftendesigners ist darin begründet, diverse Anforderungen unter einen Hut zu bringen. Die Schrift soll konsistent wirken und den Eindruck einer gleichmäßigen Helligkeitsverteilung über einem Textblock hervorrufen. Auch kleine Zeichen sollen deutlich und schnell erkennbar sein.

Das Auge soll schnell entlang der Zeilen geführt werden und beim Springen im Text Passagen wiederfinden. Hierfür gibt es kein Erfolgsrezept, denn diese Erfordernisse sind je nach Textinhalt und Bestimmungszweck unterschiedlich zu gewichten.

Es ist eben etwas Anderes, ob die Schrift für einen kurzen Text eines Bezeichnungsfelds herhalten soll, oder für einen Erläuterungsblock, für eine Tabelle, oder für den Ausdruck über einen Bericht. Bevor wir uns damit im Folgenden beschäftigen, kurz noch einige Definitionen zur Gestalt der Schriftzeichen. Betrachten Sie hierzu Bild 2.

Der Abstand zwischen Klein- und Großbuchstaben sind die Oberlängen. Der größte Kleinbuchstabe bildet die Mittellänge. Darunter finden sich für die Zeichen **g**, **j**, **y** und so weiter die Unterlängen. Ganz unten schließt der sogenannte Durchschuss die Zeichen ab. Das ist im Prinzip der Abstand zwischen den Zeilen. Unter Word etwa spielt dieser in der Schriftdatei definier-

te Durchschuss keine so große Rolle, weil Sie da den Zeilenabstand gesondert für Absätze einstellen können, unter Access haben Sie darauf jedoch keinen Einfluss. Und hier liegt der Hund begraben! Leider sind 10 Punkt in der einen Schriftart nicht dasselbe wie in einer anderen.

Zwar ist eigentlich genau definiert, was die Punktgröße einer Schrift ausmacht. Es ist die Summe von Ober-, Mittel- und Unterlänge ohne Durchschuss. Demnach müsste jede Schriftart der Größe 10 Punkt exakt die gleiche Höhe haben.

In der Realität wird davon jedoch abgewichen und für die Punktgröße häufig auch ein Teil des Durchschusses abgezwackt. In der Folge variiert die reelle Höhe je nach Schriftart locker um den Faktor 3. Sie können sich also bei der Schrifthöhe nicht allein auf die Punktgröße verlassen.

Ein Punkt ist übrigens definiert durch das 72tel eines Inchs und ist damit etwa 0,35 mm groß. Ein 10pt-Font müsste also Zeichen von maximal 3,5 mm Höhe darstellen. Geht man von den gängigen Pixelabmessungen heutiger Monitore aus, muss demnach so ein Zeichen durch etwa 13-14 Pixel gerendert werden. Daran wird deutlich, dass für Serifen nicht ausreichend Spielraum besteht.

Fontname	Fontsize	Bold	Width2
Blue Highway Condensed	10	☐	3886
HPDXC	10	☐	4692
Utah Condensed	10	☐	4730
Franklin Gothic Medium Cond	10	☐	5059
Arial Narrow	10	☐	5086
Tunga	10	☐	5090
Nina	10	☐	5153
Calibri	10	☐	5410
Gill Sans MT	10	☐	5594
Sassoon Sans	10	☐	5602
Corbel	10	☐	5609
Andale Sans	10	☐	5652
Myriad Web	10	☐	5765
Franklin Gothic Book	10	☐	5779
Miramonte	10	☐	5782
Abadi MT	10	☐	5803
Segoe UI	10	☐	5921
Trebuchet MS	10	☐	5971
Tahoma	10	☐	6000
Malgun Gothic	10	☐	6132
MS Sans Serif	10	☐	6158
Arial	10	☐	6158
Small Fonts	10	☐	6158
Frutiger Linotype	10	☐	6178
RHGDT	10	☐	6194
Arial Unicode MS	10	☐	6216
MS Gothic	10	☐	6262
Futura Md BT	10	☐	6300
Arial Rounded MT	10	☐	6336
Lucida Sans Unicode	10	☐	6583
Mangal	10	☐	6696
Bitstream Vera Sans	10	☐	6809
Consolas	10	☐	6821

Bild 3: Datenblattformular mit vertikaler Ausdehnung

Schriften in Access-Berichten

Es hängt vom Zweck eines Access-Berichts ab, welche Schriften hier zum Einsatz kommen können. Soll der Bericht nur am Bildschirm angezeigt werden, so gilt für die Schriftarten weitgehend das Gleiche, wie für Formulare. Serifenlose Schriften sind zu bevorzugen. Die Lesbarkeit steht an erster Stelle.

Soll der Bericht auch, oder in erster Linie, gedruckt werden, so kommen auch Serifenschriften infrage. Dann können Sie dieselben Überlegungen anstellen, wie für ein Word-Dokument.

Allerdings hängt es maßgeblich vom Inhalt des Berichts ab, welche Schriftart zum Einsatz kommt. Geht es um eine Tabellendarstellung oder gegliederte Ausgabe von Daten, so ist ein gleichmäßiges Schriftbild bei komprimierter Schriftgröße günstig, was gleichzeitig die Verwendung mehrerer Schriftarten im Dokument infrage stellt. Letzteres ist jedoch ohne Weiteres statthaft, wenn der Bericht der Ausgabe von Rechnungen dient, wobei man sich gemeinhin an die Vorlagen anlehnt, die auch unter Word für ein Geschäftsschreiben verwendet werden.

Schriften in Formularen

Für den Einsatz von Schriftarten in Formularen gelten andere Erfordernisse. Zu unterscheiden wäre dabei zunächst zwischen normaler Formularansicht und Datenblattansicht.

Schriften für die Datenblattansicht

Ähnlich, wie unter Excel, stellt sich bei tabellarischem Layout meist die Frage, wie sich Scrollen möglichst vermeiden lässt und viele Spalten oder Zeilen auf dem Monitor platziert werden können, ohne dass der Anwender zur Lupe greifen muss oder sich nach der Arbeit zur Teilnahme an einer Rückenschule genötigt sieht. Also ist eine möglichst kleine

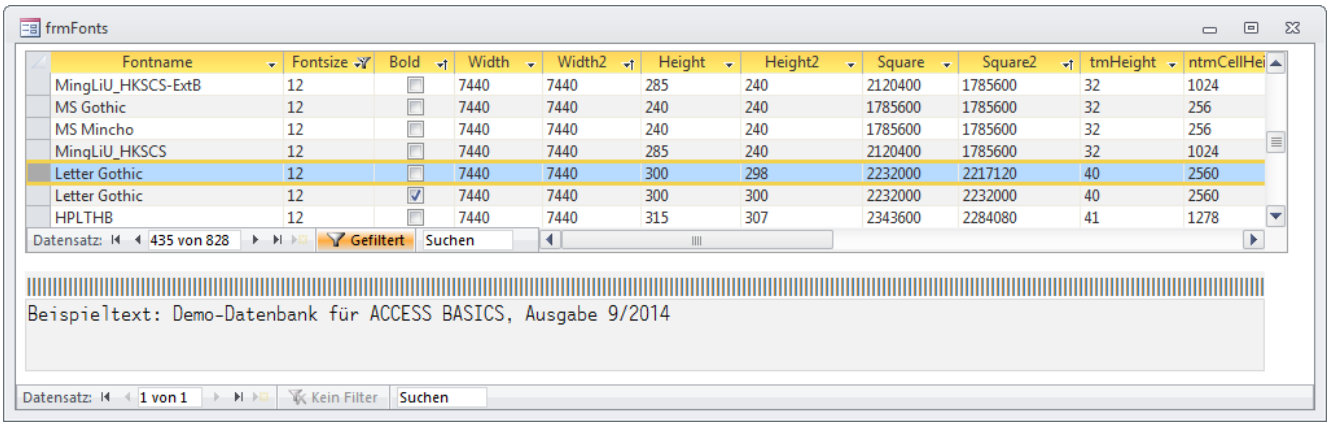

Bild 4: Unterdatenblatt im Formular mit horizontaler Ausdehnung

Schriftgröße im Verein mit optimaler Lesbarkeit gesucht. Da ein Datenblatt unter Access für alle Zellen nur eine Schrift vorsieht, braucht man sich um Konsistenz der Schrift bei verschiedenen Größen und Wichtungen nicht zu kümmern. Von Inhalt und Anzahl der Spalten hängt es ab, ob eine geringe Zeilenhöhe im Fokus steht oder eine Schriftart mit komprimierter Laufweite.

Nehmen Sie etwa die Tabelle aus Bild 3 auf der folgenden Seite. Die horizontale Ausdehnung des Datenblatts ist gering, da es nur vier Spalten aufweist. Folglich liegt das Augenmerk weniger auf Schriftarten, die eine geringe Laufweite haben, also horizontal komprimiert sind, als vielmehr darauf, möglichst viele Zeilen auf den Bildschirm zu bringen, um lästiges Scrollen zu vermeiden. Als Schriftarten kommen solche infrage, die eine geringe Höhe, einen kleinen Durchschuss aufweisen und dennoch gut lesbar sind.

Anders der Fall im Formular **frmFonts** der Beispieldatenbank (s. Bild 4). Das eingebettete Datenblatt hat dort zahlreiche Spalten und somit vertikale Ausdehnung. Das horizontale Scrollen wird zwar unvermeidlich, sollte aber minimiert werden. Also benutzen wir hier eine Schrift, die möglichst klein ist, dennoch gut lesbar, aber deren Höhe weniger relevant ist als die Laufweite. Horizontal sollten möglichst viele Zeichen in die Spalten passen. Welche Schriftarten das sind, finden wir später noch heraus, wenn es um die Routinen der Beispieldatenbank geht.

Schriften für Eingabeformulare

Eingabemasken bestehen für gewöhnlich aus einem Mix von Bezeichnungsfeldern, Textfeldern und eventuell Listenfeldern. Für jedes dieser Elemente gelten andere Anforderungen. Während man für Listenfelder dieselben Überlegungen wie für Datenblätter anstellen kann, möchte man Labels eher unscheinbar wirken lassen und wenig Platz für sie vergeuden.

Auf den Eingabefeldern, also Textboxen und Comboboxen, liegt der Fokus. Sie sollen deutlich hervorstechen. Natürlich gibt es neben der Schriftart auch noch weitere Gestaltungsmöglichkeiten, wie Schriftfarben, Hintergrundfarben und Rahmen der Steuerelemente zur ansprechenden Formatierung. Dennoch möchte man auch Textfelder so gestalten, dass möglichst viel Text in deren Eingaberechteck unterzubringen ist. Also ist auch hier die Laufweite der Schrift von Belang.

Grundsätzlich ist dabei anzuraten, nur eine Schriftart für das ganze Formular zu verwenden und auch die Schriftgrößen nicht zu variieren. Ausnahmen wären Überschriften und Fußzeilen. Ansonsten können Hervorhebungen durch Fettdruck erfolgen. Das Ganze ist im Interesse eines ruhigen Gesamteindrucks der Benutzeroberfläche.

Schriftgröße

Die Schriftgröße ist wahrscheinlich ein Punkt, über den sich streiten lässt. Über lange Zeit war von Microsoft aus 8pt-Schrift als Standard für Formulare voreingestellt. Ab Access 2007 war es plötzlich die 11-Punkt-Schrift.

In Printmedien war lange Zeit 10,5 als Standard auszumachen. Inzwischen scheint die Lesbarkeit nicht mehr oberste Priorität zu haben und man sieht sich, beeinflusst von vermeintlich (?) modernem Design, häufig auch mal mit einer Schriftgröße von 7pt konfrontiert – die Lesebrille hat man ja ab dem 35ten Geburtstag schließlich immer im Gepäck...

Formularinhalte speichern und wieder abrufen

Webbrowser kennen diese Funktion dank Add-Ons schon länger: Wenn Sie dort Daten in ein Formular eintragen, können Sie diese speichern und bei der nächsten Verwendung des Formulars per Mausklick erneut einfügen. Warum gibt es das unter Access eigentlich noch nicht? Wie auch immer: Dieser Beitrag zeigt, wie Sie die Inhalte eines Access-Formulars zwischenspeichern und bei Bedarf wieder abrufen. Das hilft sowohl dem Entwickler als auch dem Benutzer, denn sowohl bei der Entwicklung als auch bei der Anwendung der Datenbank kann dieses Feature sehr nützlich sein – wenn auch mit leicht unterschiedlichem Ansatz.

Wir wollen die Technik zum Speichern und Wiederherstellen eines Formularinhalts von zwei Seiten betrachten: Aus Entwicklersicht und aus Benutzersicht. Der Entwickler benötigt das Feature hauptsächlich, um beim Testen eines Formulars nicht die gleichen Daten immer wieder neu einzugeben, sondern diese gegebenenfalls per Mausklick zu speichern und wiederherzustellen. Für den Entwickler soll diese Funktion eher flexibel und für alle Formulare anwendbar sein – am besten also in Form eines Add-Ins, das die Inhalte mehrerer Formulare verwalten kann.

Für den Benutzer ist eine eingebaute Funktion sinnvoll, welche zum einen eine Schaltfläche zum Speichern der aktuellen Eingabe bietet, zum anderen eine Auswahlmöglichkeit für die bisher gespeicherten Daten.

Formularinhalte speichern für Benutzer

In diesem Beitrag schauen wir uns die Variante für Benutzer an. Dazu erstellen wir zunächst ein einfaches Formular, das die Daten der Tabelle **tblArtikel** in der Formularansicht anzeigt. Diesem fügen wir ein Kombinationsfeld namens **cboVorlageAuswaehlen** hinzu sowie eine Schaltfläche mit der Beschriftung **Speichern** und dem Namen **cboVorlageSpeichern**. Im Entwurf sieht das Formular nun wie in Bild 1 aus.

Bevor wir uns nun an die Programmierung machen können, müssen wir uns ein paar Gedanken um die Speicherung selbst machen. Die erste Frage dabei lautet: Welche Inhalte sollen überhaupt gespeichert werden? Und die zweite: Wohin und wie speichern wir diese Inhalte?

Die erste Frage ist eigentlich einfach zu beantworten: Natürlich alle Daten, die der Benutzer in das Formular eingegeben hat! Nur müssen wir noch herausfinden, in welchen Steuerelementen sich diese Daten befinden. Aber wie erkennen wir, welche Steuerelemente die Daten enthalten, die in der zugrunde liegenden Datenherkunft gespeichert werden?

Und: Gibt es überhaupt immer eine Datenherkunft? Kann es nicht sein, dass ein Formular zur Eingabe von Daten dient, die noch nicht einmal in der Datenbank gespeichert werden? Dies könnte ja geschehen, wenn Sie etwa die Daten für eine Banküberweisung in ein Formular eingeben und die Überweisung anstoßen. Die Daten müssen nicht zwangsläufig gespeichert werden, denn wir können diese später nach erfolgter Überweisung vom Bankserver einlesen.

Allein die Bindung eines Steuerelements ein Feld der Datenherkunft lie-

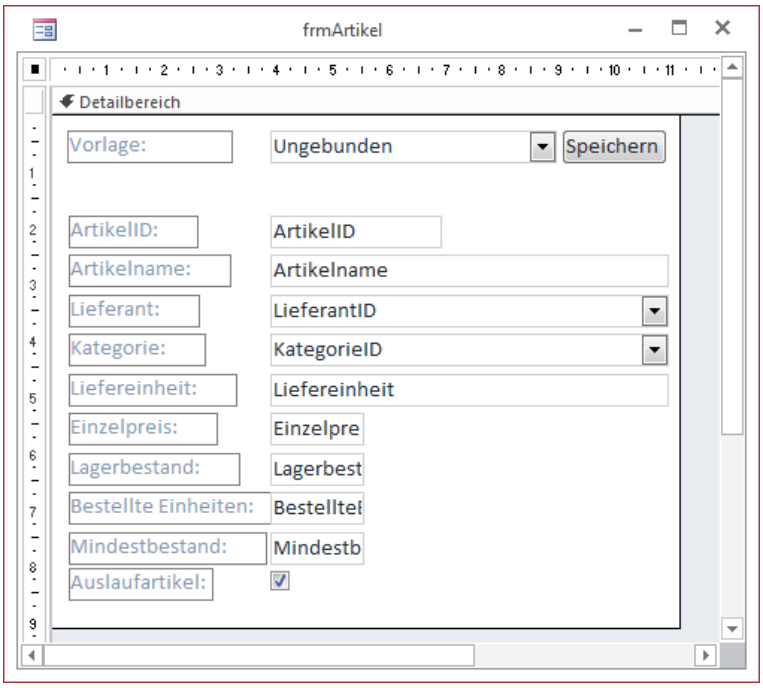

Bild 1: Das Beispielformular in der Entwurfsansicht

fert also kein Indiz dafür, dass der Inhalt des Steuerelements gespeichert und wiederhergestellt werden soll.

Also gestalten wir die Funktion so flexibel, dass wir sowohl einfach den Inhalt aller gebundenen Steuerelemente speichern können als auch speziell zum Speichern markierte Steuerelemente.

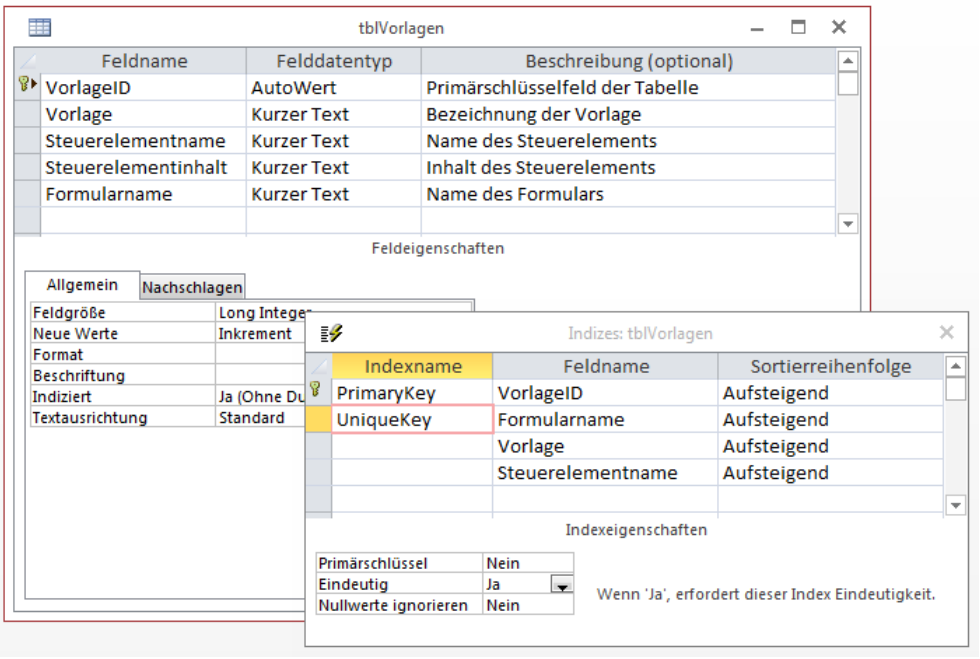

Bild 2: Entwurf der Tabelle **tblVorlagen**

Tabelle zum Speichern der Daten

Die Tabelle zum Speichern der Inhalte der Steuerelemente soll ebenso flexibel sein. Wir möchten diese also nicht auf eine einzige Datenherkunft auslegen und somit nicht etwa die Artikeltabelle abbilden, sondern beliebig viele Felder für unterschiedliche Vorlagen in der Tabelle speichern.

Damit wir verschiedene Vorlagen für das gleiche Formular speichern können, benötigen wir ein Feld, das den Namen der Vorlage aufnimmt. Wenn wir die Vorlagen-Funktion für mehrere Formulare in der gleichen Datenbank verwenden möchten, müssen wir außerdem noch ein Feld vorsehen, das angibt, zu welchem Formular die Vorlage gehört.

Schließlich benötigen wir ein Feld für den Steuerelementnamen und eines für den Inhalt des Steuerelements. Welchen Datentyp soll das Feld zum Speichern der Inhalte haben? Da wir einfach nur die in den Steuerelementen enthaltenen Werte speichern wollen, die ja in verschiedenen Datentypen auftreten können, verwenden wir den Datentyp **Text**. Diesem können wir sowohl Texte also auch Zahlen, Datumsangaben oder **Boolean**-Werte zuweisen (**0**/**-1**).

Die Tabelle **tblVorlagen** sieht im Entwurf schließlich wie in Bild 2 aus. Dort erkennen Sie auch, dass es einen aus den drei Feldern **Formularname**, **Vorlage** und **Steuerelementname** zusammengesetzten, eindeutigen Index gibt.

Speichern des Formularinhalts

Nun benötigen wir eine Prozedur, die durch das Ereignis **Beim Klicken** der Schaltfläche **cmdVorlageSpeichern** ausgelöst wird. Zuvor müssen wir noch festlegen, welche Felder gespeichert werden sollen. Wir wollen zwei Arten ermöglichen:

- Speichern aller Felder, die an ein Feld der Datenherkunft gebunden sind
- Speichern aller Felder, deren **Tag**-Eigenschaft einen bestimmten Wert enthält, zum Beispiel **AlsVorlageSpeichern**.

Die erste Variante ist für gebundene Formulare interessant, die zweite für gebundene und ungebundene Formulare. Welche Methode zum Einsatz kommt, legen wir durch eine **Boolean**-Variable fest, die beim Laden des Formulars auf den entsprechenden Wert eingestellt wird.

Werfen wir nun einen Blick auf die Schaltfläche **cmdVorlageSpeichern**. Diese löst beim Anklicken die Prozedur aus Listing 1 (Teil I) aus. Die Prozedur liest zunächst den Namen des aktuellen Formulars aus der Eigenschaft **Name** in die Variable **strFormular** ein. Dann prüft sie, ob aktuell im Kombinationsfeld **cboVorlageAuswaehlen** ein Eintrag ausgewählt ist. Falls ja, geht sie davon aus, dass die aktuellen Inhalte des Formulars unter dem Namen der ausgewählten Vorlage gespeichert werden sollen. Das

```
Private Sub cmdVorlageSpeichern_Click()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim ctl As Control
    Dim strName As String
    Dim varWert As Variant
    Dim strVorlage As String
    Dim strFormular As String
    Dim bolFeldSpeichern As Boolean
    Dim strTest As String
    strFormular = Me.Name
    If Not IsNull(Me!cboVorlageAuswaehlen) Then
        strVorlage = Me!cboVorlageAuswaehlen
    Else
        strVorlage = "[Neue Vorlage]"
    End If
    strVorlage = InputBox("Geben Sie eine Bezeichnung für die Vorlage ein.", "Vorlage speichern", strVorlage)
    Set db = CurrentDb
    db.Execute "DELETE FROM tblVorlagen WHERE Vorlage = '" & strVorlage & "' AND Formularname = '" & strFormular _
        & "'", dbFailOnError
    Set rst = db.OpenRecordset("SELECT * FROM tblVorlagen", dbOpenDynaset)
    For Each ctl In Me.Controls
        bolFeldSpeichern = False
        Select Case bolGebundeneFelder
            Case True
                On Error Resume Next
                strTest = ctl.ControlSource
                If Err.Number = 0 Then
                    bolFeldSpeichern = True
                End If
                On Error GoTo 0
            Case False
                If ctl.Tag = "AlsVorlageSpeichern" Then
                    bolFeldSpeichern = True
                End If
        End Select
        ...
```

Listing 1: Speichern des aktuellen Formularinhalts als Vorlage, Teil I

bedeutet natürlich, dass die vorhandenen Daten überschrieben werden. Der Name der aktuell ausgewählten Vorlage landet dann zunächst in der Variablen **strVorlage**. Ist das Kombinationsfeld leer, trägt die Prozedur den Platzhalter **[Neue Vorlage]** in die Variable **strVorlage** ein.

Dieser Wert wird nun in einer **InputBox** angezeigt, die neben dem Text **Geben Sie eine Bezeichnung für die Vorlage ein.** den in **strVorlage** gespeicherten Wert als Standardwert verwendet (s. Bild 3). Anschließend löscht die Prozedur mit einer per **Execute**-Methode ausgeführten **DELETE**-Anweisung alle Datensätze aus der Tabelle **tblVorlagen**, deren Feld **Vorlage** den Wert aus **strVorlage** und deren Feld **Formularname** den Wert aus **strFormular** enthält.

Danach öffnet die Prozedur ein Recordset auf Basis der Tabelle **tblVorlagen** und speichert den Verweis darauf in der

```
        ...
        If bolFeldSpeichern = True Then
            strName = ctl.Name
            varWert = ctl.Value
            Select Case strName
                Case "cboVorlageAuswaehlen", "cmdVorlageSpeichern"
                Case Else
                    rst.AddNew
                    rst!Vorlage = strVorlage
                    rst!Formularname = strFormular
                    rst!Steuerelementname = strName
                    rst!Steuerelementinhalt = varWert
                    rst.Update
            End Select
        End If
    Next ctl
    Me!cboVorlageAuswaehlen.Requery
    Me!cboVorlageAuswaehlen = strVorlage
End Sub
```

Listing 2: Speichern des aktuellen Formularinhalts als Vorlage, Teil II

Variablen **rst**. In einer **For Each**-Schleife durchläuft die Prozedur dann alle Steuerelemente des Formulars (**Me.Controls**) und speichert das aktuelle Steuerelement in der Variablen **ctl**.

Die folgenden Anweisungen ermitteln, ob der Inhalt des Steuerelements aus **ctl** gespeichert werden kann und soll. Zunächst geht die Prozedur vom Gegenteil aus und stellt die Variable **bolFeldSpeichern**, die letztlich darüber entscheidet, auf den Wert **False** ein.

Dann prüft die Prozedur den Wert der Variablen **bolGebundeneFelder**. Diese wird im Kopf des Klassenmoduls deklariert:

```
Dim bolGebundeneFelder As Boolean
```

Den Wert dieser Variablen legen wir in der Ereignisprozedur **Beim Öffnen** des Formulars fest – dazu später mehr. Hat die Variable den Wert **True**, sollen die Werte aller gebundenen Felder des Formulars in der Tabelle **tblVorlagen** gespeichert werden, sonst nur diejenigen Felder, deren **Tag**-Eigenschaft (im Eigenschaftsfenster unter **Marke** zu finden) den Wert **AlsVorlageSpeichern** enthält.

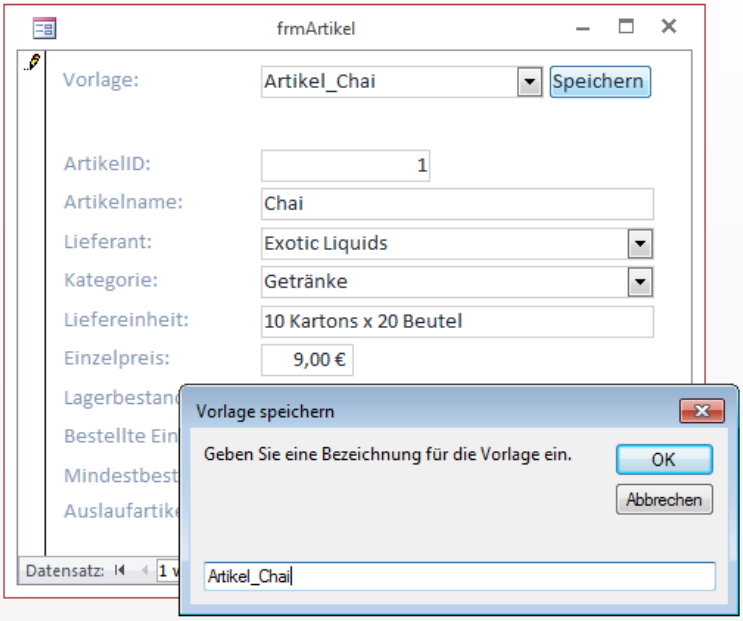

Bild 3: Abfrage des Namens für die neue Vorlage

Die Prozedur ermittelt den Wert der Variablen **bolGebundeneFelder** in einer **Select Case**-Bedingung und schlägt dort den entsprechenden Weg ein.

Im Fall der gebundenen Felder deaktiviert die Prozedur zunächst die Fehlerbehandlung und versucht dann, den Wert der Eigenschaft **ControlSource** des Steuerelements in die Variable **strTest** einzulesen. Dies führt zu einem Fehler, wenn das Steuer-

element nicht über die Eigenschaft **Steuerelementinhalt** (**ControlSource**) verfügt, der aber zu einer Fehlermeldung führt. Stattdessen prüft die Prozedur im nächsten Schritt, ob **Err.Number** den Wert **0** enthält, was dafür spricht, dass bei der Zuweisung kein Fehler aufgetreten ist und die Eigenschaft **Steuerelementinhalt** somit gefüllt ist.

Es handelt sich also um ein gebundenes Feld, dessen Inhalt in der Vorlage gespeichert werden soll, also stellt die Prozedur die Variable **bolFeldSpeichern** auf den Wert **True** ein.

Sollte **bolGebundeneFelder** den Wert **False** enthalten, sollen nur speziell gekennzeichnete Felder in der Vorlage gespeichert werden.

Die Prozedur prüft also für das aktuelle Steuerelement, ob die **Tag**-Eigenschaft den Wert **AlsVorlageSpeichern** aufweist. In diesem Fall erhält **bolFeldSpeichern** wiederum den Wert **True** (anderenfalls bleibt er **False**).

Da wir nun durch den Wert der Variablen **bolFeldSpeichern** wissen, ob der Inhalt des Steuerelements gespeichert werden soll, können wir die benötigen Werte ermitteln (s. Listing 2). Der Name des Steuerelements landet in **strName**, der Wert in **strWert**.

Wir prüfen das Steuerelement noch dahingehend, ob es sich dabei um das Kombinationsfeld zur Auswahl der Vorlagen oder um die Schaltfläche **cmdVorlageSpeichern** handelt. Falls nicht, erfolgt der eigentliche Speichervorgang für dieses Steuerelement.

Hier legen wir mit der **AddNew**-Methode einen neuen Datensatz in der Tabelle an und füllen die Felder **Vorlage**, **Formularname**, **Steuerelementname** und **Steuerelementinhalt** mit den Werten der Variablen **strVorlage**, **strFormularname**, **strName** und **strWert**. Schließlich speichert die **Update**-Methode den Datensatz.

Auf die gleiche Weise durchläuft die Prozedur alle Steuerelemente des Formulars und speichert die Inhalte der infrage kommenden Steuerelemente. Schließlich aktualisiert sie die Datensatzherkunft des Kombinationsfeldes **cboVorlageAuswaehlen** und stelle dieses auf die soeben gespeicherte Vorlage ein.

Der Inhalt der Tabelle **tblVorlagen** sieht für eine Vorlage des Beispielformulars **frmArtikel** nun wie in Bild 4 aus.

tblVorlagen

VorlageID	Vorlage	Steuerelementname	Steuerelementinhalt	Formularname
437	Artikel_Chai	ArtikelID	1	frmArtikel
438	Artikel_Chai	Artikelname	Chai	frmArtikel
439	Artikel_Chai	LieferantID	1	frmArtikel
440	Artikel_Chai	KategorieID	1	frmArtikel
441	Artikel_Chai	Liefereinheit	10 Kartons x 20 Beutel	frmArtikel
442	Artikel_Chai	Einzelpreis	9	frmArtikel
443	Artikel_Chai	Lagerbestand	39	frmArtikel
444	Artikel_Chai	BestellteEinheiten	0	frmArtikel
445	Artikel_Chai	Mindestbestand	10	frmArtikel
446	Artikel_Chai	Auslaufartikel	0	frmArtikel
(Neu)				

Datensatz: 1 von 10 Kein Filter Suchen

Bild 4: Die Tabelle **tblVorlagen** mit einer Beispielvorlage

Kombinationsfeld zur Auswahl einer Vorlage

Schauen wir uns nun das Kombinationsfeld **cboVorlageAuswaehlen** an. Es erhält seine Datensatzherkunft beim Öffnen des Formulars durch die Ereignisprozedur, die durch das Ereignis **Beim Öffnen** ausgelöst wird (s. Listing 3). Dabei dient der Formularname als Filter für die anzuzeigenden Elemente. Damit jeder Vorlagenname nur einmal angezeigt wird, verwendet die Abfrage hinter **SELECT** das Schlüsselwort **DISTINCT**.

Auswahl einer Vorlage

Wenn der Benutzer einen der im Kombinationsfeld enthaltenen Einträge

```
Private Sub Form_Open(Cancel As Integer)
    bolGebundeneFelder = True
    Me!cboVorlageAuswaehlen.RowSource = "SELECT DISTINCT Vorlage FROM tblVorlagen WHERE Formularname = '" _
        & Me.Name & "' ORDER BY Vorlage;"
End Sub
```

Listing 3: Diese Prozedur wird bei Öffnen des Formulars ausgelöst.

Kombinationsfelder mit vielen Einträgen

Ein Kombinationsfeld soll zur schnellen Auswahl von Daten dienen. Wenn es aber sehr viele Daten enthält, stoßen Sie einerseits an die maximal mögliche Anzahl anzuzeigender Daten, zweitens geht die Performance beim Laden der Daten in die Knie. Wir haben ein Beispiel mit sehr vielen Datensätzen erstellt und zeigen Ihnen in diesem Beitrag, wie Sie mit den Begrenzungen umgehen und dem Benutzer dennoch ein ergonomisches Arbeiten ermöglichen.

Da nicht jeder Entwickler eine Tabelle mit um die 100.000 Datensätzen zur Verfügung hat, um damit die Grenzen eines Kombinationsfeldes auszuloten und passende Lösungen für solche Fälle zu finden, erstellen wir uns auf die Schnelle eine neue Tabelle mit der entsprechenden Menge Datensätze.

Die Tabelle heißt **tblBeispieldaten** und sieht im Entwurf wie in Bild 1 aus.

Nach dem Hinzufügen der Beispieldaten sieht die Tabelle in der Datenblattansicht wie in Bild 2 aus. Wir haben erst einmal 100.000 Datensätze hinzugefügt, um etwas Beispielmaterial zu haben. Aber wie haben wir das erledigt? Mit einer kleinen Prozedur, die sich einiger Hilfsfunktionen zum Erstellen von GUIDs bedient und so die entsprechenden Werte zum Einfügen in das Feld Beispielwert ermittelt hat.

Diese Prozedur finden Sie in Listing 1. Sie durchläuft eine Schleife mit Werten von 1 bis 100.000 und erzeugt bei jedem Durchlauf eine neue GUID mit der Funktion **CreateGUID** (siehe Modul **mdlTools**). Die folgende Anweisung

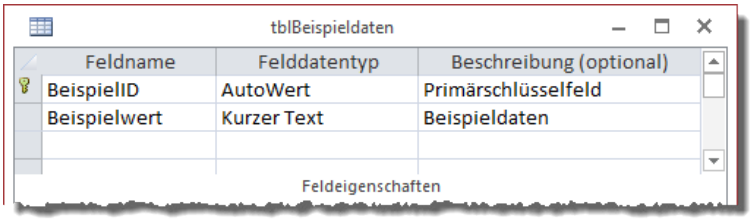

Bild 1: Die Tabelle mit den Beispieldaten in der Entwurfsansicht

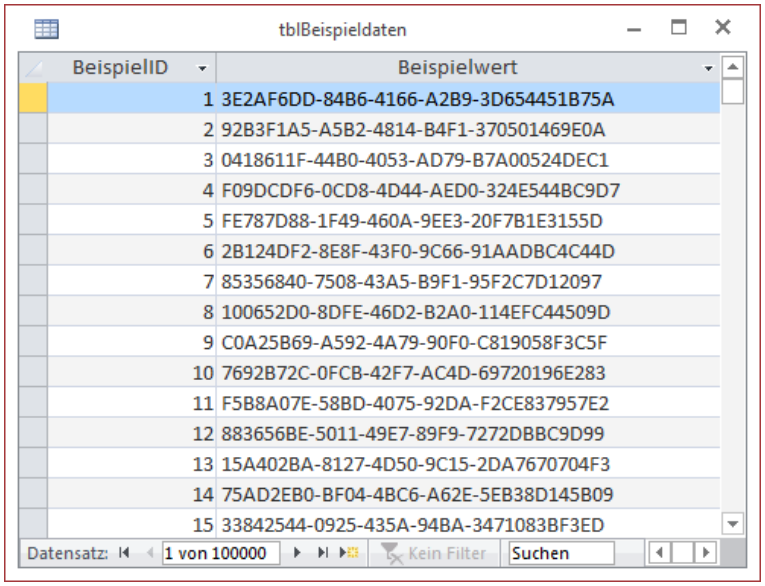

Bild 2: Die Tabelle mit den Beispieldaten

```
Public Sub BeispieltabelleFuellen()
    Dim db As DAO.Database
    Dim l As Long
    Dim strBeispielwert As String
    Set db = CurrentDb
    For l = 1 To 100000
        strBeispielwert = Mid(CreateGUID, 2, 36)
        db.Execute "INSERT INTO tblBeispieldaten(Beispielwert) VALUES('" & strBeispielwert & "')", dbFailOnError
    Next l
End Sub
```

Listing 1: Diese Prozedur legt schnell 100.000 verschiedene Datensätze in einer Tabelle an.

legt jeweils einen neuen Datensatz mit der frisch ermittelten GUID in der Tabelle **tblBeispieldaten** an. Und das dauerte noch nicht einmal besonders lang – gerade mal eine gute Minute.

Kombinationsfeld anlegen

Im ersten Versuch soll das Kombinationsfeld zunächst ohne Rücksicht alle Datensätze der Tabelle anzeigen. Dazu fügen Sie einem neuen, leeren Formular ein Kombinationsfeld namens **cboBeispielwerte** hinzu. Stellen Sie die Eigenschaft **Datensatzherkunft** auf den Wert **tblBeispieldaten** ein (s. Bild 3).

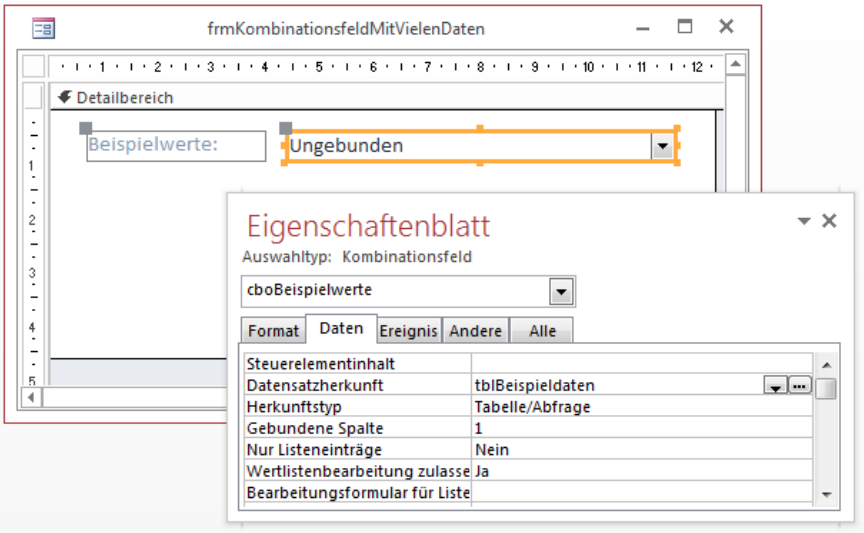

Bild 3: Das Kombinationsfeld zum Ausprobieren in der Entwurfsansicht

Klicken Sie dann auf die Schaltfläche mit den drei Punkten, um den Abfrage-Generator zu öffnen. Wir benötigen zwar die Daten der Tabelle **tblBeispieldaten**, aber wir wollen diese noch nach dem Inhalt des Feldes **Beispielwert** in aufsteigender Reihenfolge sortieren. Dazu passen Sie die Datensatzherkunft wie in Bild 4 an.

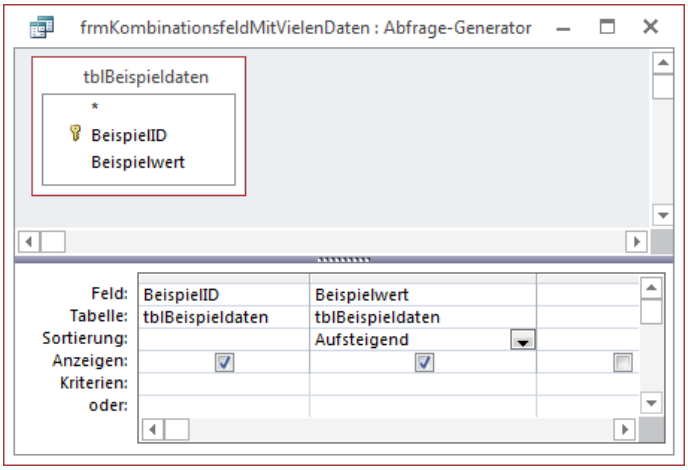

Bild 4: Die Datensatzherkunft des Kombinationsfeldes

Da die Datensatzherkunft nun aus dem Primärschlüsselfeld und dem Feld mit den anzuzeigenden Daten besteht, stellen wir noch die beiden Eigenschaften **Spaltenanzahl** auf **2** und **Spaltenbreiten** auf **0cm** ein. Auf diese Weise wird nur der Wert des Feldes **Beispielwert** angezeigt, gleichzeitig können wir aber den Primärschlüsselwert des Kombinationsfeldes auslesen.

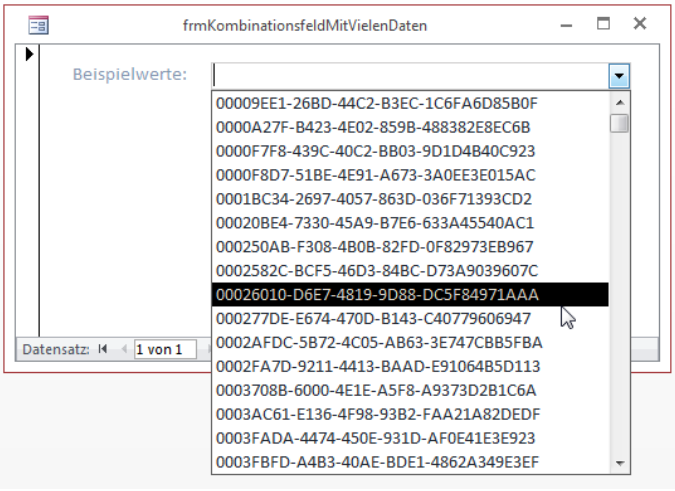

Bild 5: Kombinationsfeld mit Daten

In der Formularansicht zeigt das Kombinationsfeld nun die Einträge der Datensatzherkunft an (s. Bild 5).

Aber wie viele Datensätze zeigt es an? Dass es nicht alle Datensätze der Datensatzherkunft sind, können wir leicht ermitteln, indem wir im Kombinationsfeld mit der Bildlaufleiste nach ganz unten scrollen. Aber aufgepasst: Wenn Sie den Balken im ersten Anlauf ganz nach unten ziehen, springt es kurz darauf wieder weiter nach oben. Sie können den Balken noch mehrmals bis ganz nach unten ziehen – es dauert eine Weile, bis das Kombinationsfeld alle Datensätze erfasst hat und tatsächlich bis ganz nach unten springt.

Dies gelingt wesentlich schneller, wenn Sie das Kombinationsfeld aufklap-

pen (beispielsweise mit der Taste **F4**) und dann die Tastenkombination **Strg + Ende** betätigen.

In unserem Beispiel landen wir dann zwar ganz unten (weiter runter geht es nicht), aber wir treffen dort sicher nicht den letzten Datensatz an – dieser müsste eigentlich mit dem Buchstaben **F** beginnen.

Stattdessen sind wir aber noch nicht einmal über die Zahl 5 hinaus (s. Bild 6). Dabei sei angemerkt, dass die Tabelle nach einem Test nun 200.000 Datensätze enthält.

Um herauszufinden, wie viele Datensätze das Kombinationsfeld enthält, verwenden wir die Eigenschaft **ListCount** des Kombinationsfeldes.

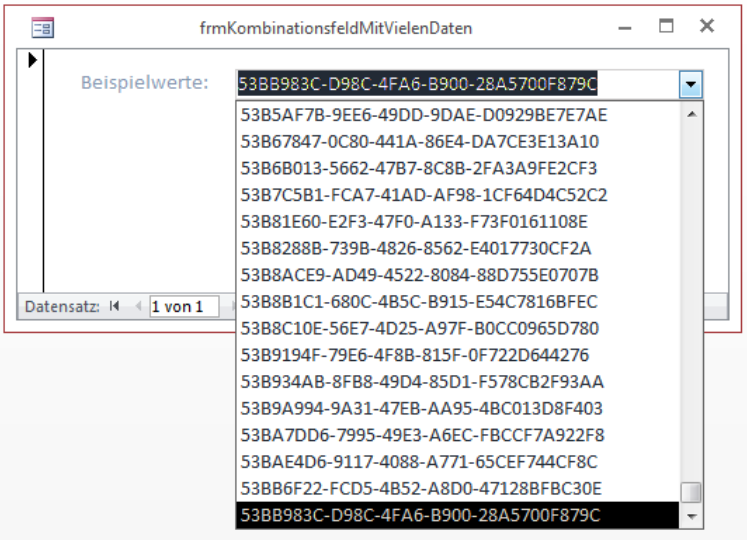

Bild 6: Der letzte angezeigte Datensatz im Kombinationsfeld

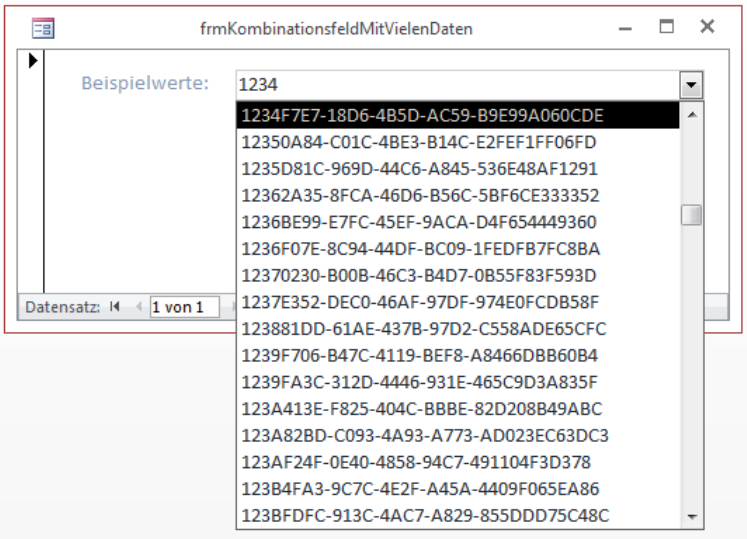

Bild 7: Nach Eingabe einiger Zeichen zeigt das Kombinationsfeld die Liste ab dem ersten möglichen Wert an.

```
Debug.Print Forms!
frmKombinationsfeldMitVielenDaten!
cboBeispielwerte.ListCount
 65536
```

Wenn Sie, wie ich, auch keine Lust haben, solch lange Ausdrücke einzugeben, können Sie das Kombinationsfeld auch direkter referenzieren – Sie müssen lediglich sicherstellen, dass es im aktuellen Access-Fenster den Fokus hat:

```
Debug.Print Screen.ActiveControl.ListCount
 65536
```

Und siehe da – wir finden genau 65.536 Datensätze vor, also 2 hoch 16. Nun haben wir ein Problem: Dank der Begrenzung kann der Benutzer die Datensätze im hinteren Bereich der Datensatzherkunft überhaupt nicht auswählen, denn diese sind schlicht nicht im Kombinationsfeld enthalten.

Datensatzherkunft nach Wortanfang

Doch natürlich bemühen wir uns, auch für dieses Problem eine Lösung zu finden. Diese sieht so aus, dass wir den Benutzer auffordern, ein oder mehrere Zeichen einzutippen, mit denen der gesuchte Ausdruck beginnt (s. Bild 7).

Das ist eigentlich sogar das Standardverhalten eines Kombinationsfeldes, zumindest, wenn die Eigenschaft **Automatisch ergänzen** auf den Wert **Ja** eingestellt ist. So zeigt das Kombinationsfeld automatisch den richtigen Eintrag an, wenn der Benutzer etwa die Zeichenfolge **1234** eingibt.

Das ändert allerdings nichts an der Tatsache, dass das Kombinationsfeld aktuell noch nicht alle Einträge der Datensatzherkunft anzeigt. Also müssen wir wohl per VBA nachhelfen und jeweils die Datensätze anzeigen, die der aktuell in das Kombinationsfeld eingetragenen Zeichenkette entsprechen.

Daten dynamisch laden

Wir wollen also nun einen Weg finden, die Daten möglichst effizient zu laden – und zwar alle Daten, die der Benutzer anfragt. Das heißt, dass wir auf die Eingabe eines ersten Zeichens durch den

Mails per Drag and Drop in der Datenbank speichern

Mails landen gewöhnlicherweise im Posteingangsordner und werden dann, entweder automatisch oder manuell, in den jeweiligen Zielordner verschoben – wenn sie nicht gar gelöscht werden. Aber kann das schon alles sein? Nein! In diesem Artikel zeigen wir Ihnen, wie Sie beim Verschieben einer Mail in einen bestimmten Ordner weitere Aktionen auslösen können, und zwar per VBA-Ereignis. Auf diese Weise können Sie dann beispielsweise den Inhalt der Mail und gegebenenfalls auch Anlagen direkt in einer dafür vorgesehenen Datenbank speichern.

Welchen Sinn aber soll es haben, den Inhalt von E-Mails in einer Datenbank zu speichern? Sie können auf diese Weise etwa die Mails zu bestimmten Kunden in einer entsprechenden Tabelle ablegen und dem Kunden in der Datenbank zuordnen.

So brauchen Sie von der Kundendatenbank aus nicht immer noch Outlook zu öffnen, um die Kommunikation mit diesem Kunden einzusehen.

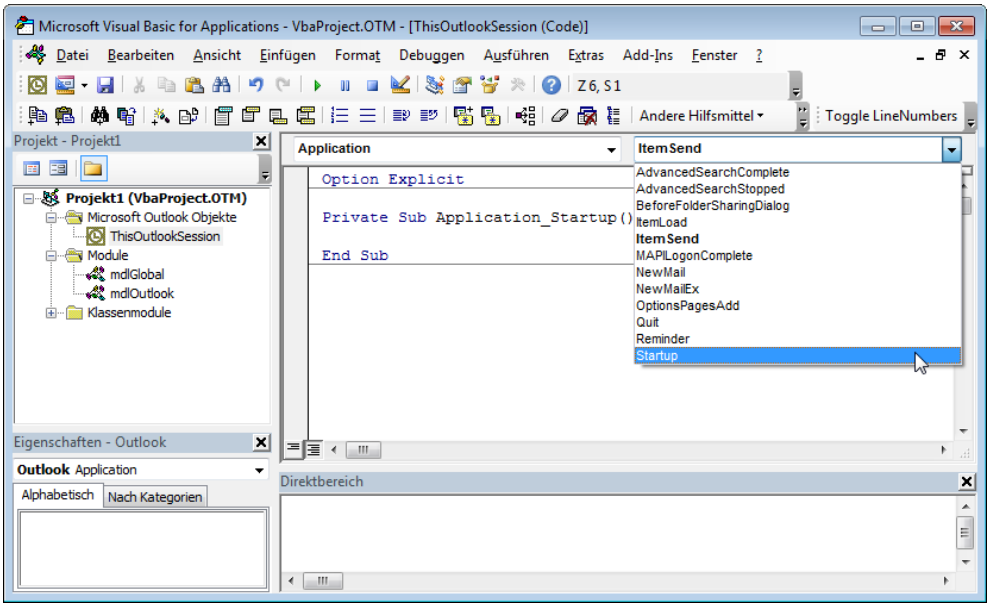

Bild 1: Anlegen der Ereignisprozedur **Application_Startup**

Voraussetzungen

Für die nachfolgend vorgestellten Techniken benötigen Sie Microsoft Office mit Outlook und Access in der Version 2007 oder neuer.

Ordner anlegen

Als Erstes wollen wir einen Ordner unter Outlook anlegen, dem Sie die in der Datenbank zu speichernden E-Mails hinzufügen. Diesen können Sie manuell anlegen, wir können dies aber auch per Code erledigen.

Dazu öffnen Sie den VBA-Editor von Outlook (bei geöffnetem Outlook-Fenster mit **Alt + F11**) und fügen dem bereits vorhandenen Klassenmodul **ThisOutlookSession** über die beiden Kombinationsfelder im Kopf des Codefensters die Ereignisprozedur **Application_Startup** hinzu (s. Bild 1). Dieses Ereignis wird bei jedem Start von Outlook ausgelöst.

Hier können wir einen Code einfügen, der prüft, ob das gewünschte Verzeichnis sich bereits an Ort und Stelle befindet oder ob es noch angelegt werden soll. Letzteres führt die Prozedur dann gegebenenfalls noch durch.

Um beim Starten von Outlook zu prüfen, ob bereits ein Verzeichnis namens **Kundenmails** vorhanden ist, und dieses gegebenenfalls anzulegen, fügen Sie der Prozedur **Application_Startup** noch die Zeilen aus Listing 1 hinzu. Diese Prozedur füllt ein Objekt namens **objNamespace** mit dem **MAPI**-Namespace-Objekt von Outlook. Dieses liefert über die **GetDefaultFolder**-Eigenschaft mit dem Parameter **olFolderInbox** einen Verweis auf den Posteingangsordner.

Dann versucht die Prozedur, einen darin enthaltenen Ordner namens **Kundenmails** mit der Variablen **objKundenmails** zu referenzieren. Gelingt dies nicht, bleibt **objKundenmails** leer, was die nachfolgende **If...Then**-Bedingung prüft.

Ist **objKundenmails** dann nicht gefüllt, legt die Prozedur diesen Ordner an und trägt einen Verweis auf diesen Ordner in die Variable **objKundenmails** ein.

Ereignis beim Hinzufügen einer Mail auslösen

Das waren die Vorarbeiten. Tatsächlich müssen wir das **Folder**-Objekt, mit dem wir das Verzeichnis **Posteingang** referenzieren, aber außerhalb der Prozedur **Application_Startup** referenzieren – und zwar mit der folgenden Anweisung:

```
Dim WithEvents objInbox As Outlook.Folder
```

Das Schlüsselwort **WithEvents** ist nötig, damit wir für die Variable **objInbox** Ereignisprozeduren im Klassenmodul **ThisOutlookSession** implementieren können. Dies erledigen Sie nun wiederum über die beiden Kombinationsfelder im Kopf des Codefensters. Wählen Sie dort links den nun verfügbaren Eintrag **objInbox** aus und rechts den Eintrag **BeforeItemMove** aus, sodass der VBA-Editor die Ereignisprozedur **objInbox_BeforeItemMove** anlegt (s. Bild 2).

```
Public Sub Application_Startup()
    Dim objInbox As Outlook.Folder
    Dim objKundenmails As Outlook.Folder
    Dim objNamespace As Outlook.NameSpace
    Set objNamespace = Outlook.GetNamespace("MAPI")
    Set objInbox = objNamespace.GetDefaultFolder(olFolderInbox)
    On Error Resume Next
    Set objKundenmails = objInbox.Folders("Kundenmails")
    On Error GoTo 0
    If objKundenmails Is Nothing Then
        Set objKundenmails = objInbox.Folders.Add("Kundenmails")
    End If
End Sub
```

Listing 1: Prüfen, ob das Verzeichnis **Kundenmails** schon vorhanden ist

Hier hinterlegen Sie nun einige weitere Codezeilen, die wie in Listing 2 aussehen. Damit haben wir nun die Basis für weitere Funktionen geschaffen: Wir haben eine Ereignisprozedur, die durch das Verschieben einer E-Mail aus dem Verzeichnis Posteingang ausgelöst wird. Diese Prozedur prüft innerhalb einer **Select Case**-Bedingung das Zielverzeichnis. Nur wenn dieses den Namen **Kundenmails** trägt, wird ein Meldungsfenster angezeigt, das über die Übermittlung der Mail in das Verzeichnis Kundenmails informiert.

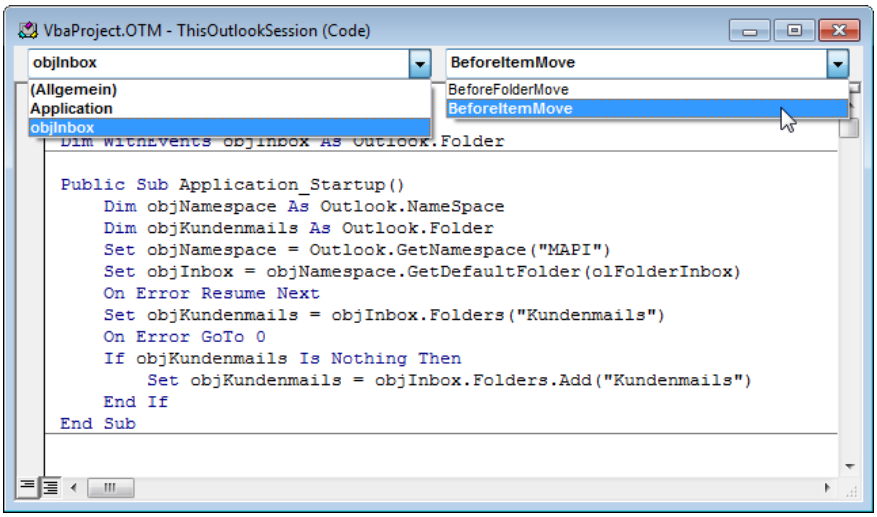

Bild 2: Hinzufügen einer Ereignisprozedur, die beim Verschieben einer Mail ausgelöst wird

Statt des Meldungsfensters können wir dort nun weitere Anweisungen hinzufügen – etwa, um die Inhalte der E-Mail in der Tabelle einer Datenbank zu speichern.

Tabelle zum Speichern der E-Mails

In einer neuen Datenbankdatei namens **Kundenmails.accdb** legen wir nun eine neue Tabelle mit der Bezeichnung **tblKundenmails** an. Diese soll die Felder aus Bild 3 enthalten. Neben dem Primärschlüsselfeld **KundenmailID** nimmt

```
Private Sub objInbox_BeforeItemMove(ByVal Item As Object, ByVal MoveTo As MAPIFolder, Cancel As Boolean)
    Select Case MoveTo.Name
        Case "Kundenmails"
            MsgBox "Mail ist angekommen."
    End Select
End Sub
```

Listing 2: Meldung ausgeben, wenn die Mail in den gewünschten Ordner verschoben wurde

die Tabelle die grundlegenden Informationen und Inhalte der E-Mail auf – so den Absender, den Empfänger, den Betreff, den Inhalt und das Versanddatum.

Die Anlagen speichern wir in einer weiteren Tabelle namens **tblMailanlagen**. Diese finden Sie in Bild 4 vor. Die Tabelle enthält ein Fremdschlüsselfeld namens **KundenmailID** zum Verknüpfen der Datensätze dieser Tabelle mit denen der Tabelle **tblKundenmails**. Außerdem speichert die Tabelle den Dateinamen der Anlage im Feld **Anlagebezeichnung** sowie die Datei selbst im Feld **Mailanlage**.

Grundsätzlich hätten wir auch einfach nur ein Feld mit dem Datentyp **Anlage** in der Tabelle **tblKundenmails** anlegen können. Wir möchten da Anlage-Feld jedoch nicht dazu nutzen, mehrere Dateien zu speichern, da Informationen wie etwa der Dateiname dann nicht so einfach über die Benutzeroberfläche (etwa über Formulare) einsehbar sind.

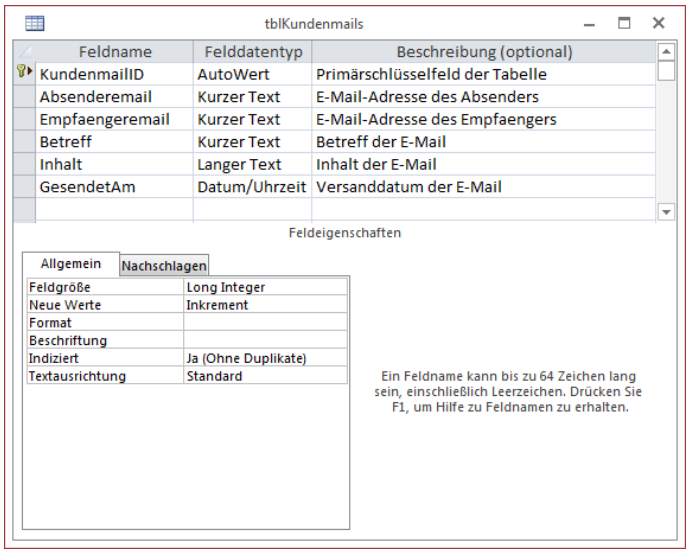

Bild 3: Die Tabelle **tblKundenmails** speichert die grundlegenden Informationen

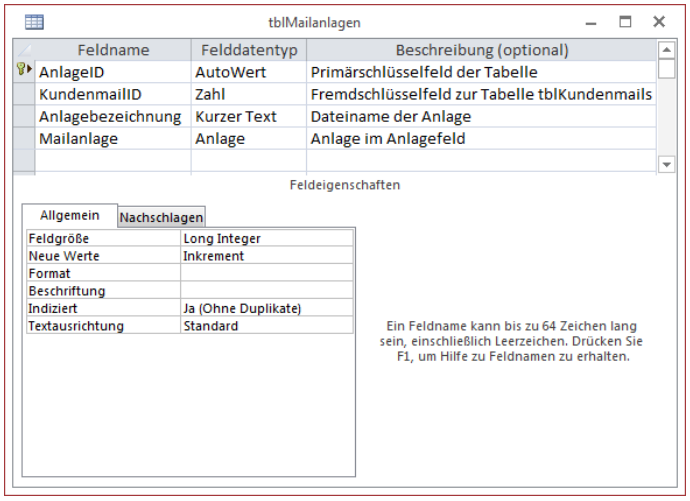

Bild 4: Die Tabelle **tblMailanlagen** speichert die Mailanlagen

E-Mails und Anlagen speichern

Listing 3 liefert zunächst zwei Konstanten namens **cStrPath** und **cStrDB**, die den Pfad und den Namen der Datenbank enthalten. Sicher gibt es elegantere Möglichkeiten, als Pfad und Dateiname hier fest zu verdrahten, aber aktuell reicht uns diese Variante aus.

Die Ereignisprozedur **objInbox_BeforeItemMove** prüft wiederum den Zielordner des Drag-and-Drop-Vorgangs und ruft im Falle des Ordners **Kundenmails** die Prozedur **MailSpeichern** auf. Dieser übergibt sie mit dem Parameter **Item** einen Verweis auf das verschobene E-Mail-Objekt. Sicher haben Sie auch

```
Const cStrPath As String = "Z:\Dokumente\Daten\Fachmagazine\AccessImUnternehmen\2014\06\OutlookDrop\"
Const cStrDB As String = "Kundenmails.accdb"

Private Sub objInbox_BeforeItemMove(ByVal Item As Object, ByVal MoveTo As MAPIFolder, Cancel As Boolean)
    If Not MoveTo Is Nothing Then
        Select Case MoveTo.Name
            Case "Kundenmails"
                MailSpeichern Item
        End Select
    End If
End Sub
```

Listing 3: Konstanten mit Pfad und Datenbankname und die Ereignisprozedur **objInbox_BeforeItemMove**

erkannt, dass wir der Prozedur noch eine **If...Then**-Bedingung hinzugefügt haben. Diese prüft, ob **MoveTo** überhaupt einen Wert enthält. Sollte der Benutzer nämlich eine E-Mail vollständig löschen (also bei gedrückter Umschalttaste und ohne Umweg über den Ordner **Gelöschte Objekte**), dann ist das **MoveTo**-Objekt gar nicht mehr mit einem Verweis auf das entsprechende **MailItem**-Objekt gefüllt und die Prozedur kann beendet werden.

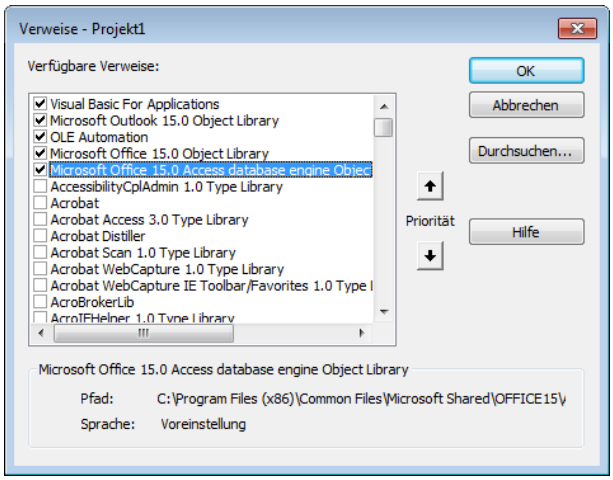

Bild 5: Verweis auf die DAO-Bibliothek hinzufügen

Vorbereitungen für den Zugriff auf Access

Das VBA-Projekt von Outlook kümmert sich normalerweise nicht um Access-Datenbanken. Dementsprechend wird auch der Zugriff auf die Tabellen einer Datenbank mit den Methoden der DAO-Bibliothek nicht nativ unterstützt. Damit wir beim Programmieren den gewohnten Komfort mit IntelliSense auch für die DAO-Objekte erhalten, fügen wir dem Projekt einen Verweis auf die Bibliothek **Microsoft Office x.0 Access database engine Objects** hinzu – und zwar über den **Verweise**-Dialog aus Bild 5, den Sie mit dem Menüeintrag **Extras|Verweise** öffnen.

Speichern der E-Mail

Das Speichern der E-Mail erledigt die Prozedur **MailSpeichern**, die Sie in Listing 4 finden.

Die Prozedur erstellt zunächst einen Verweis auf die Zieldatenbank namens **Kundenmails.accdb** in dem angegebenen Zielordner. Dort, wo Sie sonst die **CurrentDb**-Funktion verwenden, um auf die aktuelle Datenbank zuzugreifen, nutzen wir nun die Funktion **OpenDatabase**, um von Outlook aus die Datenbank **Kundenmails.accdb** zu nutzen. Dann erstellt die Prozedur ein Recordset-Objekt auf Basis der Tabelle **tblKundenmails**, wobei allerdings keine Datensätze zurückgegeben werden sollen – daher die Bedingung **1=2**.

Die Prozedur fügt einen neuen Datensatz hinzu und füllt einige Felder:

Die Eigenschaft **SenderEMailAddress** landet im Feld **AbsenderEMail**, **Subject** in **Betreff**, **Body** in **Inhalt** und **SentOn** im Feld **GesendetAm**. Etwas komplizierter ist es mit der Empfängeradresse:

```
Private Sub MailSpeichern(objMail As Outlook.MailItem)
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim lngKundenmailID As Long
    Set db = OpenDatabase(cStrPath & cStrDB)
    Set rst = db.OpenRecordset("SELECT * FROM tblKundenmails WHERE 1 = 2", dbOpenDynaset)
    rst.AddNew
    rst!Absenderemail = objMail.SenderEmailAddress
    rst!Empfaengeremail = objMail.Recipients.Item(1)
    rst!Betreff = objMail.Subject
    rst!Inhalt = objMail.Body
    rst!GesendetAm = objMail.SentOn
    lngKundenmailID = rst!KundenmailID
    rst.Update
    If objMail.Attachments.Count > 0 Then
        AnlagenSpeichern db, lngKundenmailID, objMail.Attachments
    End If
End Sub
```

Listing 4: Prozedur zum Speichern des Inhalts einer E-Mail in der Tabelle **tblKundenmails**

VBA unter Access mit 64 Bit

Die 64bit-Version von Access sollten Sie schon deshalb nicht nutzen, weil diese ActiveX-Steuerelemente wie das TreeView-, das ListView- oder das ImageList-Steuerelement nicht unterstützt. Aber es gibt noch weitere Gründe: Zum Beispiel könnte es Probleme bei der Verwendung von API-Funktionen geben. Manchmal bleibt einem Entwickler allerdings keine anderen Möglichkeit, weil etwa am Arbeitsplatz nur die 64bit-Version vorliegt oder aber der Kunde mit dieser Version arbeiten muss. Der vorliegende Beitrag liefert Informationen, wie Sie nicht kompatible VBA-Anweisungen für die Zielversion anpassen müssen und wie Sie programmieren müssen, um die 32bit- und die 64bit-Version gleichermaßen zu bedienen.

Unterschiede zwischen 32bit- und 64bit-Version

Warum entsteht das in der Einleitung beschriebene Dilemma überhaupt und was ist mit 32bit-Version und 64bit-Version überhaupt gemeint?

Hierbei beziehen wir uns zunächst auf das Betriebssystem. Wenn Sie ein 32bit-Betriebssystem nutzen, kann dieses maximal 2 hoch 32 Bytes Arbeitsspeicher adressieren, also ca. 4.096 Megabyte. Bei einem 64bit-Betriebssystem kann ein Arbeitsspeicher von 2 hoch 64 Byte adressiert werden – dies wird noch eine Weile ausreichen.

Wer also mehr als vier Gigabyte Arbeitsspeicher in seinem Rechner verbaut hat und sich wundert, dass davon nur vier angezeigt werden, fährt also vermutlich noch ein 32bit-Betriebssystem.

Möchten Sie herausfinden, wie viel Arbeitsspeicher Sie nutzen und ob Sie ein 32bit- oder ein 64bit-Betriebssystem nutzen, können Sie dies über die Systemsteuerung ermitteln. Unter Windows 7 etwa klicken Sie dazu auf **Start|Systemsteuerung** und dann auf System. Es erscheint der Dialog aus Bild 1, dem Sie die notwendigen Informationen entnehmen können. In diesem Fall handelt es sich um ein 64bit-System mit acht Gigabyte Arbeitsspeicher.

Bild 1: Hinweise auf Arbeitsspeicher und Betriebssystem in den Systemeinstellungen von Windows

Office mit 32bit und 64bit

Damit kommen wir zum Thema Office und somit auch zu Access. Access 2010 etwa wird standardmäßig in der 32bit-Variante installiert.

Wer die 64bit-Variante wünscht, muss dies explizit bei der Installation angeben. Warum das? Weil es sehr viele externe Tools und Steuerelemente für Access in der 32bit-Version gibt. Würden Sie Access in der 64bit-Version installieren und versuchen, ein mit der 32bit-Version etwa von VB6 entwickeltes Add-In zu starten, führt dies zu einem Fehler.

Das Gleiche gilt, wenn Sie etwa ein 32bit-ActiveX-Steuerelement wie das **TreeView**-Steuerelement verwenden möchten – mehr dazu weiter unten.

Einfache Lösung

Wenn Sie eine neue Office-Version installieren, können Sie diese und anderen Probleme auf einfache Art und Weise umschiffen: Installieren Sie einfach die 32bit-Variante. Windows in der 64bit-Variante bietet eine Kompatibilitätsumgebung namens **WOW64**, mit der Sie 32bit-Anwendungen wie in diesem Fall Office und Access ganz normal ausführen können.

In diesem Fall wird Office nicht in dem dafür vorgesehenen Verzeichnis **C:\Program Files\Microsoft Office\Office15** installiert, sondern etwa unter **C:\Program Files (x86)\Microsoft Office\Office15**.

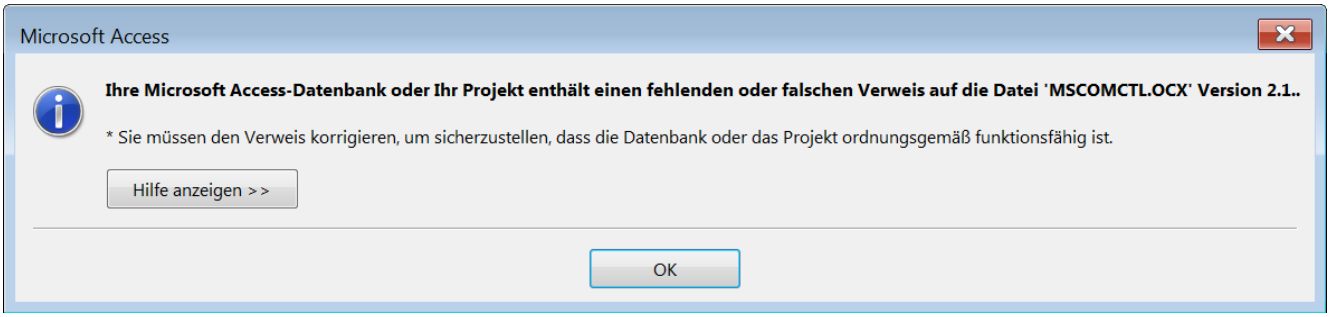

Bild 2: Der Verweis auf die Datei **MSCOMCTL.ocx** kann unter Access in der 64bit-Version nicht gefunden werden.

Probleme mit ActiveX-Steuerelementen

Das vermutlich kritischste Problem, das bei der Nutzung von Access in der 64bit-Version auftaucht, sind ActiveX-Steuerelemente, die nur in der 32bit-Version verfügbar sind.

Wir haben testweise eine frische Datenbank unter der 32bit-Version mit einem TreeView-Steuerelement versehen. Wenn Sie diese Datenbank unter Access in der 64bit-Version öffnen, löst dies zunächst die Meldung aus Bild 2 aus.

Wenn Sie diese Meldung ignorieren und dann das Formular mit dem **TreeView**-Steuerelement öffnen, erhalten Sie die Meldung aus Bild 3.

Was tun? Nun: Es gibt keine 64bit-Version dieses Steuerelements, also können Sie nur zur 32bit-Version von Access wechseln oder eine Alternative für das **TreeView**-Steuerelement suchen.

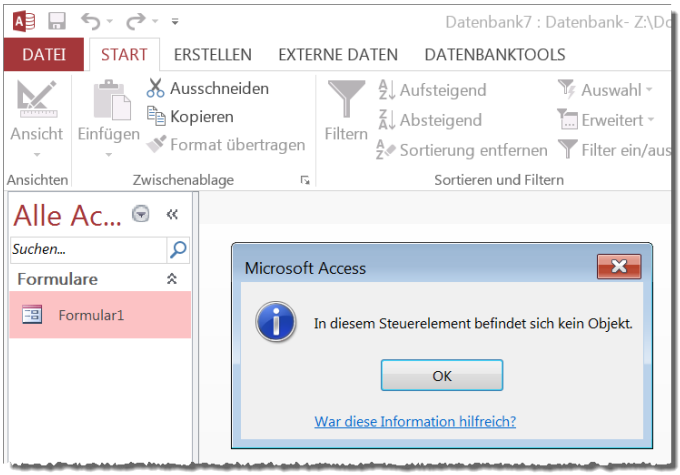

Bild 3: Der Container mit dem **TreeView**-Steuerelement ist leer.

Probleme mit mde/accde

Wenn Sie eine **.mde**- oder **.accde**-Datei unter Access in der 32bit-Version erzeugt haben, können Sie diese nur mit der entsprechenden Access-Version öffnen. Wenn Sie etwa eine 32bit-accde-Datei unter Access in der 64bit-Version öffnen, erhalten Sie die Fehlermeldung aus Bild 4. In diesem Fall besteht Hoffnung: Wenn Sie Zugriff auf die unkompilierte Version der Datenbankdatei haben, können Sie diese unter der 64bit-Version von Access kompilieren und die **.mde**/**.accde**-Datei dann unter Access in der 64bit-Version nutzen.

Probleme mit API-Deklarationen

Ein weiteres Problem tritt auf, wenn Sie API-Funktionen, die mit der 32bit-Version von Access funktionieren, unter der 64bit-Version einsetzen möchten. Wir schauen uns dazu die GDI-/OGL-Bibliothek von Sascha Trowitzsch an, die eine Menge API-Funktionen verwendet, und wollen diese für die 64bit-Version lauffähig machen.

Wenn Sie das Modul **mdlOGL0710** unter Access 2013 in der 64bit-Version öffnen, erhalten Sie einige rot markierte Zeilen wie in Bild 5. Der erste Schritt, um die API-Deklarationen für die 64bit-Version nutzbar zu machen, liegt im Hinzufügen eines einfachen Schlüsselworts, und zwar **PtrSafe**. Dieses Schlüsselwort fügen Sie unmittelbar hinter der **Declare**-Anweisung der API-Funktion ein,

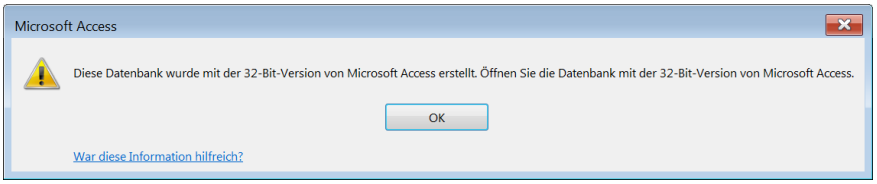

Bild 4: Fehler beim Öffnen einer .mde- oder .accde-Datei, die mit Access in der 32bit-Version erstellt wurde

sodass etwa die erste in der Abbildung sichtbare Funktion wie in Bild 6 aussieht.

Auf diese Weise statten Sie nun alle API-Funktionen mit diesem Schlüsselwort aus. Die Suchen/Ersetzen-Funktion des VBA-Editors ist dabei eine willkommene Erleichterung: Ersetzen Sie zunächst **Declare Function** durch **Declare PtrSafe Function** und dann **Declare Sub** durch **Declare PtrSafe Sub** – klicken Sie dabei jeweils auf **Alle ersetzen** (siehe Bild 7).

Sollten sich noch weitere nicht kompatible API-Deklarationen finden, können Sie diese durch Betätigen des Menübefehls **Debuggen|Kompilieren von <Datenbankname>** auffinden. Es erscheint dann die Meldung aus Bild 8.

Damit wären schon einmal die Probleme mit der Variablendeklaration behoben.

Probleme mit Variablentypen

Nach diesem Problem folgt auch gleich das nächste – wenn auch nicht offensichtlich. Dazu betätigen wir nochmals den **Debuggen|Kompilieren von <Datenbankname>**.

In unserem Beispiel meldet Access nun **Fehler beim Kompilieren: Typen unverträglich** und markiert einen Parameter der API-Funktion **GdipCreateBitmapFromFile** (siehe Bild 9). Hier wird offensichtlich eine Funktion namens **StrPtr** verwendet, um der Funktion die Adresse der Variablen **sFileName** zu übermitteln.

Was geht hier schief? Schauen wir uns die Funktion **StrPtr** an, indem wir das Kontextmenü zu diesem Befehl öffnen und den Eintrag **Definition** auswählen. Dies führt zunächst zur Fehlermeldung aus Bild 10.

Das ist aber kein Problem: Wir aktivieren im Objektkatalog mit einem Klick auf den Eintrag **Verborgene Einträge anzeigen** des Kontextmenüs die Anzeige der verborgenen Elemente.

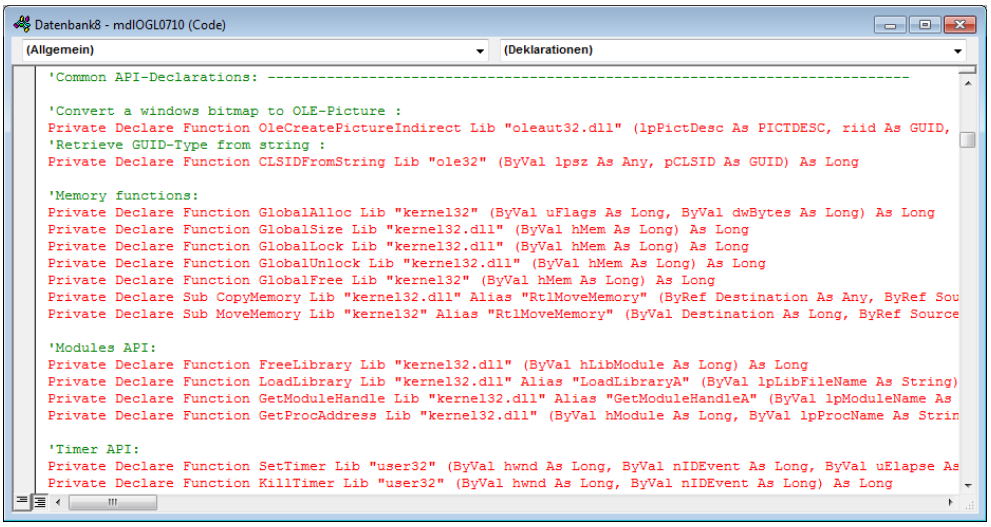

Bild 5: Fehlerhafte API-Deklarationen

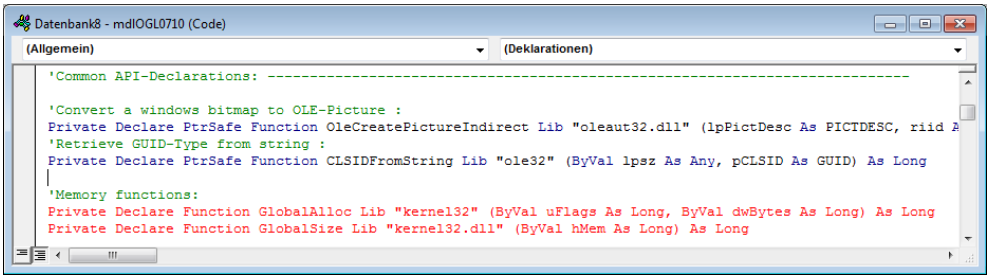

Bild 6: Hinzufügen des Schlüsselworts **PtrSafe**

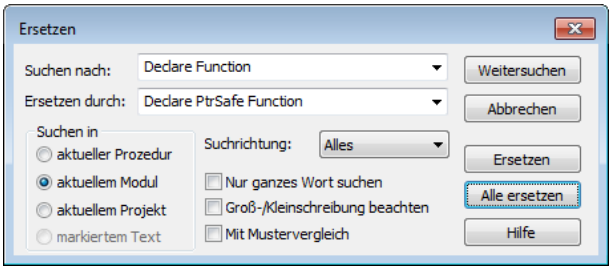

Bild 7: **PtrSafe** per Suchen/Ersetzen hinzufügen

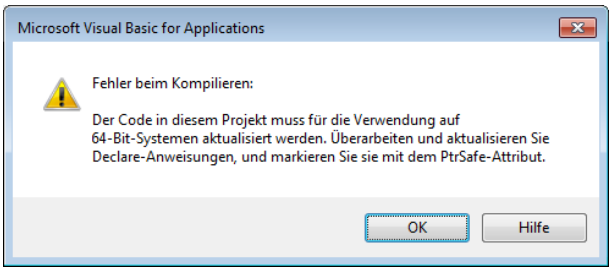

Bild 8: Meldung, wenn beim Debuggen noch inkompatible API-Deklarationen gefunden werden

Das Windows Management Instrumentarium

Wenn Sie in Ihrer Datenbank externe Informationen über System, Hardware oder Software benötigen, so lassen sich diese unter VBA meist nur über umfangreiche API-Routinen gewinnen. Dabei gibt es mit dem Windows Management Instrumentarium, kurz WMI, eine Schnittstelle, die fast keine Wünsche offen lässt und Ihnen enorm viel Programmierarbeit abnehmen kann. Das ist Grund genug, diese Schnittstelle einmal genauer unter die Lupe zu nehmen.

Beispieldatenbank

Die Beispiele dieses Artikels finden Sie in der Datenbank **1409_WMI.mdb**.

Wie funktioniert das WMI?

Windows bringt WMI als einen Service mit, der in der Systemverwaltung auf den Namen **Windows-Verwaltungs-instrumentation** hört und automatisch gestartet wird. Wirklich aktiv wird es allerdings erst, wenn eine Anfrage an den Service gestellt wird.

Dann findet man im Taskmanager den neuen Prozess **WmiPrvSE.exe**, eine Datei, die aus dem Unterverzeichnis **\wbem** des Windows-Systemordners heraus gestartet wird. Der Prozess selbst verwaltet nur die Anfragen und hat deshalb den Namen **WMI Provider Host**.

Bei Anfragen entscheidet er, welche zusätzlichen Dlls zu laden sind, in denen die eigentlichen Funktionen zur Ermittlung der Systemkomponenten untergebracht sind. Diese Dlls sind die **Provider**, also Datenlieferanten, und ebenfalls im **\wbem**-Verzeichnis beheimatet.

Sie finden dort außerdem zahlreiche Dateien mit der Endung **.mof**. Das sind Textdateien, die die Struktur jedes Providers beschreiben.

Wir kommen noch darauf zu sprechen, wie eine Anfrage an den Service gestellt wird. Der Ablauf sieht schematisch danach jedenfalls so aus:

- Anfrage an WMI-Service
- Der Service befragt den Provider Host.
- Der Provider Host durchsucht beim ersten Start das Verzeichnis **\wbem** nach allen Provider-Definitionen und baut daraus im Speicher eine Art Datenbank auf. Aus dieser wird je nach Anfrageschlüsselwörtern der geeignete Provider herausgesucht und die entsprechende Dll geladen.
- Über eine COM-Schnittstelle wird ein Klassenobjekt aus der Dll erzeugt, das zu einem gesuchten Objekt, etwa einer Druckerinstanz, die Eigenschaften zurückliefert. Die Dll weiß selbst, welche weiteren Dlls sie zum Lösen der Aufgabe benötigt, und verwendet dazu normale API-Methoden.
- Die Ergebnisse laufen den Weg zurück bis zum Auslöser der Anfrage

Der gesamte Vorgang ist standardisiert, sodass es keine Rolle spielt, ob Sie Hardware- oder Software-Komponenten abfragen wollen – der Ablauf ist immer identisch.

Da das WMI intern eine Datenbank über alle möglichen abfragbaren Objekte aufbaut, liegt es nahe, dass diese über einen SQL-Dialekt angesprochen werden kann. Das ist auch der Fall: Die Abfragesprache nennt sich **WQL**. Damit ist es möglich, ein Abbild der WMI-Datenbank in das Datenmodell einer Access-Datenbank zu überführen. Die Lösung dazu finden Sie in der Beispieldatenbank zum Beitrag. Das Datenmodell sieht aus wie in Bild 1 und zeigt eine hierarchische Struktur. Um es zu erläutern und die Entsprechungen zur WMI-Datenbank deutlich zu machen, nehmen wir ein einfaches WQL-Beispiel zur Hand:

```
SELECT Caption FROM Win32_Printer
```

Analog zu einer Access-Abfrage handelt sich bei **Caption** offenbar um ein Feld, das aus der Datenherkunft **Win32_Printer** stammt. Gesucht werden hier alle Bezeichnungen der im System installierten Drucker. **Win32_Printer** scheint demnach eine Tabelle oder selbst eine Abfrage zu sein. Tatsächlich trifft eher Letzteres zu. Was unter Access eine Tabelle oder Abfrage ist, nennt sich unter WMI allerdings Objektklasse.

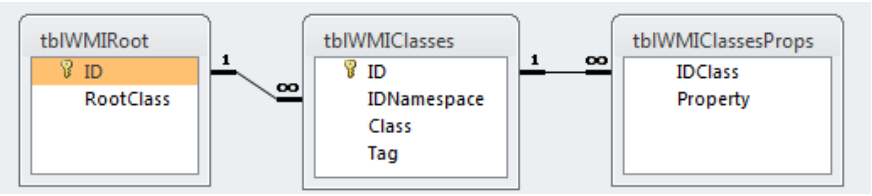

Bild 1: Analogie des WMI-Datenmodells unter Access

WMI-Datenmodell

Welches ist aber die Datenbank, aus der die Objektabfrage stammt? Um bei der Analogie zu bleiben, nehmen wir an dieser Stelle das Ergebnis der Routinen der Beispieldatenbank vorweg und verweisen auf Bild 2.

Das ist der Inhalt der im vorherigen Bild dargestellten Tabelle **tblWMIRoot**. Was unter Access als die aktuelle Datenbank **CurrentDb** bekannt ist, wäre im WMI die Objektklasse **root**.

Sie besteht aus einer Auflistung von Grundklassen, bestimmten Standardkategorien, auch Namespaces genannt, von denen der Provider **CIMV2** der wichtigste ist. **CIMV2** ist eine Abkürzung für **Common Information Model** Version 2. **root** wäre der Name der Datenbank, **CIMV2** eine Abfrage auf die Tabelle der Root-Provider. Also sähe die Syntax der Abfrage **Win32_Printer** eigentlich so aus:

tblWMIRoot

ID	RootClass
1	subscription
2	DEFAULT
4	MSAPPS11
5	CIMV2
6	Cli
7	nap
8	MicrosoftIISv2
10	snmp
11	SecurityCenter2
14	WMI
16	Policy
18	SecurityCenter
19	MSAPPS12
21	aspnet
(Neu)	

Datensatz: 1 von 14 Kein Filter

Bild 2: Root-Provider-Liste des WMI

```
SELECT * FROM root.CIMV2 _
  WHERE Name = 'Win32_Printer'
```

Dass diese Abfrage unter WQL so nicht funktioniert, werden wir noch sehen. Wichtig ist hier zunächst das Prinzip.

Ließe man die Bedingung in der Abfrage weg, so käme man auf eine Liste aller **CIMV2**-Klassen. In der Beispieldatenbank steht dafür die Tabelle **tblWMIClasses**. Ihr Inhalt ist auszugsweise in Bild 3 zu begutachten. **IDNamespace** ist ein Bezug zu **ID** der Root-Klasse; hier die **5** für **CIMV2**. Die Tabelle ist wiederum mit der Tabelle **tblWMIClassesProps** verlinkt, indem deren Feld **IDClass** mit der ID verknüpft und als Unterdatenblatt dargestellt wird. In **tblWMIClassesProps** stehen die Eigenschaften der jeweiligen Klasse und damit eigentlich deren Feldnamen. In der Abbildung sind für die Klasse **Win32_CurrentTime** (aktuelle Systemzeit) die Eigenschaftsfelder ausgeklappt. Wollte man etwa den Tag des aktuellen Datums abfragen, so wäre der Wert des Feldes **Day** der Abfrage **Win32_CurrentTime** zu ermitteln.

tblWMIClasses

ID	IDNamespace	Class	Tag
1000	5	Win32_ConnectionShare	ConnectionShare
1051	5	Win32_ControllerHasHub	ControllerHasHub
537	5	Win32_CreateFolderAction	CreateFolderAction
717	5	Win32_CurrentProbe	CurrentProbe
527	5	Win32_CurrentTime	CurrentTime
690	5	Win32_DCOMApplication	DCOMApplication

Property
Day
DayOfWeek
Hour
Milliseconds
Minute
Month
Quarter
Second
WeekInMonth
Year

Datensatz: 3095 von 3095 Ungefiltert Suchen

Bild 3: WMI-Objektklassen und die Eigenschaftsfelder einer Klasse

Das WMI abfragen

Schauen wir uns nun an, wie das Abfragen der Systemzeit unter VBA konkret realisiert werden kann. Grundsätzlich gibt es da zwei Wege, die beschritten werden können: eine Variante ohne und eine mit Verweis-Bibliothek. Zur verweislosen Lösung finden Sie nachfolgend zwei Routinen.

Sie können eine Instanz des WMI sehr einfach erhalten, indem Sie einen spezi-

```
Sub TestWMIDate1()
    Dim objWMI As Object
    Dim objSet As Object
    Dim obj As Object
    Set objWMI = GetObject("winmgmts://./root/cimv2")
    Set objSet = objWMI.ExecQuery("SELECT Day, Month, Year FROM Win32_CurrentTime")
    Set obj = objSet.ItemIndex(0)
    Debug.Print obj.GetObjectText_
End Sub
```

Listing 1: Erste Variante zum Ermitteln von Datumsangaben über das WMI

ellen sogenannten Moniker-String an die VBA-Funktion **GetObjekt** übergeben:

```
Set objWMI = GetObject("winmgmts:")
```

Sie erhalten damit jedoch nicht etwa die oberste Ebene der WMI-Datenbank, also **root**, sondern bereits den **CIMV2**-Provider. Denn der voll ausgeschriebene Moniker-String müsste so lauten:

```
GetObject("winmgmts://./root/cimv2")
```

WMI verwendet, wenn bestimmte Parameter im String weggelassen werden, automatisch Defaults. Zu diesen Defaults gehören der angesprochene Computer, die **root**-Datenbank und der **CIMV2**-Provider. Die Syntax des Moniker-Strings lautet genau genommen

```
winmgmts://[Computer]/root/[Provider]
```

Für **Computer** können Sie einfach einen Punkt verwenden, wenn Sie das lokale System ansprechen wollen. Damit wird bereits deutlich, dass man über das WMI auch andere vernetzte Rechner abfragen kann, soweit die Berechtigungen dazu ausreichen! Für **Computer** kann man also ebenfalls die IP-Nummer oder den Netzwerknamen des Rechners angeben:

```
winmgmts://192.168.0.2/root/cimv2
```

```
winmgmts://Harry2/root/cimv2
```

WMI ist übrigens unsensibel, was Groß- und Kleinschreibung angeht. Meist ist es egal, wie Sie einen Ausdruck schreiben.

In der Routine **TestWMIDate1** (s. Listing 1) aus dem Modul **mdlWMI** wird also zunächst ein Objekt **objWMI** gesetzt, das dem Klassenkatalog des Haupt-Providers **CIMV2** entspricht. In Analogie zu Access entspräche dieses Objekt erst einer Datenbank. Und diese kann wie unter Access über **OpenRecordset** abgefragt werden, nur dass sich der Befehl hier **ExecQuery** nennt. Im WQL-String werden also die Felder **Day**, **Month** und **Year** des Providers **Win32_CurrentTime** abgefragt.

Als Resultat bekommen Sie über diese Abfrage nicht ein Recordset, sondern ein WMI-**ObjectSet**, das in der Routine der Variablen **objSet** zugewiesen wird. Zwar kann ein Recordset viele Datensätze enthalten, doch in unserem Fall gibt es nun mal nur ein aktuelles Datum.

Das Resultat besteht also nur aus einem Datensatz (WMI-Objekt). Dieser Datensatz muss jetzt für die weitere Verwendung in einer weiteren Objektvariablen **obj** gespeichert werden. Man erhält ihn über die Funktion **ItemIndex**([Datensatznummer]) des **ObjectSets**. Und schließlich lässt sich der Inhalt des Datensatzobjekts extrem einfach über die Anweisung **GetObjectText_** ausgeben. Heraus kommt dabei etwa dies:

```
instance of Win32_LocalTime
{
    Day = 21;
    Month = 09;
    Year = 2014;
};
```

GetObjectText_ entspricht damit dem Pendant **GetString** eines ADO-Recordsets.

Da sich das Ergebnis in dieser Form schlecht weiterverwenden lässt, zeigt die Prozedur **TestWMIDate2** in Listing 2, wie man stattdessen die einzelnen Feldwerte des Datensatzes auslesen kann. Sie gleicht anfangs genau der besprochenen Routine, verwendet dann aber die Auflistung **Properties_** des Datensatzobjekts, um die Eigenschaftswerte zurückzugeben. Sie werden als String zu einem Ergebnis zusammengesetzt, der das aktuelle Datum wiedergibt:

```
21.09.2014
```

Warum erst hier, werden Sie fragen? Hätte man, wie in einer Access-Abfrage,

```
Sub TestWMIDate2()
    Dim objWMI As Object
    Dim objSet As Object
    Dim obj As Object
    Set objWMI = GetObject("winmgmts://./root/cimv2")
    Set objSet = objWMI.ExecQuery ("SELECT Day, Month, Year FROM Win32_CurrentTime")
    Set obj = objSet.ItemIndex(0)
    With obj.Properties_
        Debug.Print .Item("Day") & "." & .Item("Month") & "." & .Item("Year")
    End With
End Sub
```

Listing 2: Zweite Variante zum Ermitteln von Datumsangaben über das WMI

nicht gleich den Datums-String zusammensetzen können?

```
SELECT Day & '.' & Month & '.' &  Year
AS DatumAktuell FROM Win32_CurrentTime
```

Aber das lässt WQL eben nicht zu. Im SQL-String können nur einzelne Felder angegeben, aber nicht weiteren Operationen unterzogen werden, und die Zuweisung an einen Alias-Namen ist ebenso unmöglich.

WQL-Syntax

Die anderen relevanten Unterschiede von WQL zu JET-SQL sollen nicht verschwiegen werden. Was etwa auch nicht geht, ist die Verknüpfung mehrerer Tabellen in der Abfrage per **Joins**.

Zwar lassen sich Verknüpfungen zwischen Provider-Klassen auf etwas kompliziertere Weise auch herstellen, Erläuterungen dazu würden jedoch den Rahmen des Beitrags sprengen. Forschen Sie bei Interesse nach den WQL-Stichwörtern **Associators Of** oder **References Of**.

Möglich hingegen sind Bedingungen per **WHERE**-Statement und die Zusammensetzung von Bedingungen über **OR** und **AND**. Für unser Datumsbeispiel mit nur einem Ergebnisdatensatz macht eine bedingte Filterung keinen Sinn. Daher hier ein anderes Beispiel:

```
SELECT Caption FROM Win32_Printer _
WHERE Shared=True _
AND Name LIKE '%Epson%'
```

Hier wird nach der Bezeichnung von installierten Druckern gefragt, die erstens für die gemeinsame Verwendung freigegeben sind (**Shared**) und außerdem den Ausdruck **Epson** im Namen haben. Im Unterschied zu Jet-SQL ist beim **LIKE**-Operator das Zeichen % statt * zu verwenden.

Ansonsten können in Bedingungen auch die üblichen Vergleichsoperatoren =, >, <, >=, <=, <> sowohl für numerische Werte wie für Text verwendet werden.

Auch **IS NULL** und **IS NOT NULL** für die Unterscheidung von leeren Feldwerten lassen sich setzen:

```
SELECT Caption FROM Win32_Printer _
ServerName IS NOT NULL
```

Das ermittelt alle Drucker, bei denen ein Servername gesetzt ist, also etwa Netzwerkdrucker.

Alle weiteren Informationen zu WQL entnehmen Sie bitte der Referenz von Microsoft: WQL (SQL for WMI)

Geht es auch kürzer?

Die bisher vorgestellten Routinen lassen sich noch weiter komprimieren. In TestWMIDate3 (s. Listing 3) wurde auf eine Abfrage per **ExecQuery** verzichtet, weil der Provider für **Win32_Current-**

```
Sub TestWMIDate3()
    Dim obj As Object
    Set obj = GetObject("winmgmts:root/cimv2:Win32_CurrentTime")
    With obj.Instances_.ItemIndex(0).Properties_
        Debug.Print .Item("Day") & _
            "." & .Item("Month") & _
            "." & .Item("Year")
    End With
End Sub
```

Listing 3: Dritte Variante zum Ermitteln von Datumsangaben über das WMI

Mit Arrays arbeiten

Über Arrays unter VBA wurde schon vieles geschrieben und gesagt. Dennoch gibt es praktische Hinweise, die man gelegentlich benötigt, nicht so einfach im Internet. Also werfen wir in diesem Beitrag einen genaueren Blick auf die Möglichkeiten, die Arrays bieten, und liefern einige neue Funktionen rund um die Felder zum temporären Speichern und Bereitstellen von Daten. Unter anderem sehen wir uns an, wie Sie die Inhalte von Arrays zu Debugging-Zwecken im Direktbereich oder in einer Excel-Tabelle ausgeben oder wie Sie prüfen, ob ein Array leer ist.

Array aus Datensatz füllen

Manchmal möchten Sie vielleicht die Daten eines oder mehrerer Felder eines Datensatzes in einem Array speichern, um diese Daten performant bereitzustellen.

Wenn Sie nur ein Feld eintragen möchten, reicht ein eindimensionales Array aus. Dieses füllen Sie mithilfe der Funktion **RecordsetInArrayEindimensional** aus Listing 1. Die Funktion erwartet das einzulesende Recordset mit dem Parameter **rst**.

Es deklariert ein Array namens **arr** mit dem Datentyp **Variant**. In einer **Do While**-Schleife durchläuft die Funktion alle Datensätze des Recordsets und führt dabei zwei Anweisungen aus:

- das ReDimensionieren des Arrays entsprechend der aktuellen Position des Datensatzzeigers (**AbsolutePosition**) sowie
- das Zuweisen des Werts des ersten Felds des aktuellen Datensatzes an die entsprechende Position des Arrays.

Danach übergibt die Funktion das Array als Funktionswert an die aufrufende Routine. Diese sieht beispielsweise wie in Listing 2 aus. Die Routine füllt das Recordset-Objekt mit den Datensätzen der Tabelle **tblArtikel**, wobei nur das Feld **Artikelname** berücksichtigt wird.

```
Public Function RecordsetInArrayEindimensional(rst As DAO.Recordset) As Variant()
    Dim arr() As Variant
    Do While Not rst.EOF
        ReDim Preserve arr(rst.AbsolutePosition)
        arr(rst.AbsolutePosition) = rst.Fields(0).Value
        rst.MoveNext
    Loop
    RecordsetInArrayEindimensional = arr
End Function
```

Listing 1: Einlesen eines Feldes eines Recordsets in ein eindimensionales Array

```
Public Sub Test_RecordsetInArray()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim i As Integer
    Dim arr() As Variant
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT Artikelname FROM tblArtikel", dbOpenDynaset)
    arr = RecordsetInArrayEindimensional(rst)
    For i = LBound(arr) To UBound(arr)
        Debug.Print arr(i)
    Next i
End Sub
```

Listing 2: Test zum Einlesen eines eindimensionalen Arrays aus einem Recordset

Dann ruft sie die Funktion **RecordsetInArrayEindimensional** auf und übergibt das Recordset als Parameter.

Um zu prüfen, ob alle Werte wie gewünscht in das Array eingelesen wurden, durchläuft die Routine anschließend in einer **For...Next**-Schleife alle Elemente des Arrays und gibt den jeweils aktuellen Wert im Direktbereich des VBA-Editors aus.

ReDim vorziehen

Wenn schon vorher klar ist, wie viele Elemente im Array landen sollen, können Sie dieses natürlich auch bereits vor dem Eintritt in die **Do While**-Schleife redimensionieren.

Der Versuch, dies direkt beim Deklarieren des Arrays zu erledigen, schlägt

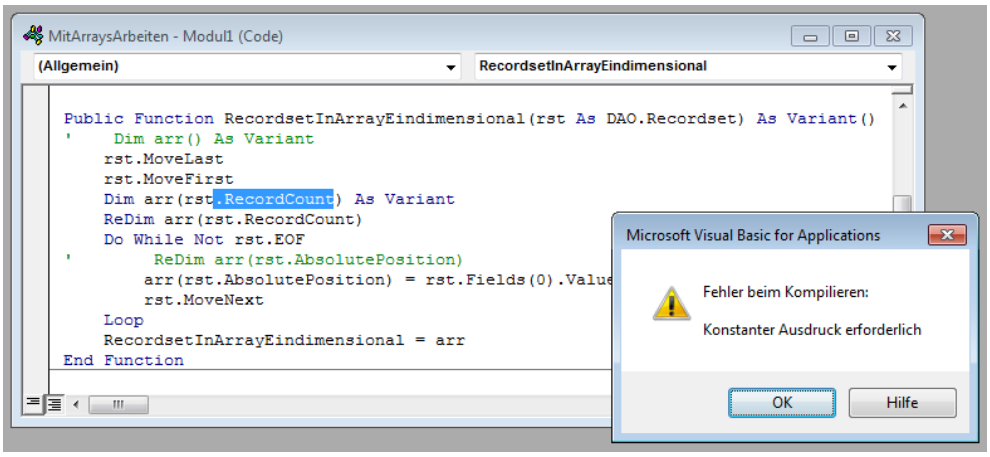

Bild 1: Automatische Ergänzung eines Feldnamens

```
Public Function RecordsetInArrayEindimensional(rst As DAO.Recordset) As Variant()
    Dim arr() As Variant
    rst.MoveLast
    rst.MoveFirst
    ReDim arr(rst.RecordCount - 1)
    Do While Not rst.EOF
        arr(rst.AbsolutePosition) = rst.Fields(0).Value
        rst.MoveNext
    Loop
    RecordsetInArrayEindimensional = arr
End Function
```

Listing 3: Einlesen eines eindimensionalen Arrays mit vorheriger Dimensionierung

```
Public Function RecordsetInArrayMehrdimensional(rst As DAO.Recordset) As Variant()
    Dim arr() As Variant
    Dim fld As DAO.Field
    Dim i As Integer
    rst.MoveLast
    rst.MoveFirst
    ReDim arr(rst.RecordCount - 1, rst.Fields.Count - 1)
    Do While Not rst.EOF
        For Each fld In rst.Fields
            i = 0
            arr(rst.AbsolutePosition, i) = fld.Value
            i = i + 1
        Next fld
        rst.MoveNext
    Loop
    RecordsetInArrayMehrdimensional = arr
End Function
```

Listing 4: Einlesen eines mehrdimensionalen Arrays mit vorheriger Dimensionierung

allerdings fehl – mit der Fehlermeldung **Konstanter Ausdruck erforderlich**, wie in Bild 1 zu erkennen.

Also deklarieren wir das Array in einer neuen Version der Funktion (s. Listing 3) zunächst ohne Angabe der Anzahl der enthaltenen Elemente und ändern diese dann später mit der **ReDim**-Anweisung. Dieser übergeben wir den Ausdruck **rst.RecordCount**. Damit diese Eigenschaft des Recordsets immer den richtigen Wert liefert, verschieben wir den Datensatzzeiger zuvor mit **MoveLast** einmal an das Ende der Datensatzgruppe und mit **MoveFirst** wieder an die erste Position.

Im Gegensatz zum ReDimensionieren innerhalb der **Do While**-Schleife entfällt hier natürlich das **Preserve**-Schlüsselwort der **ReDim**-Anweisung, denn das Array ist ja ohnehin noch leer.

Gleichzeitig beheben wir noch einen kleinen Fehler bei der Dimensionierung: Dort haben wir nämlich zunächst den Wert von **rst.RecordCount** als Obergrenze angegeben. Die Untergrenze ist jedoch, wenn nicht anders angegeben, **0**. Das heißt, dass wir das Array um ein Element zu groß dimensionieren würden. Also ändern wir den Ausdruck für die obere Grenze in **rst.RecordCount - 1**.

Mehrdimensionale Arrays mit Daten füllen

Wenn Sie nun mehrere Felder in das Array einlesen möchten, müssen Sie dieses entsprechend anders dimensionieren – nämlich als zweidimensionales Array. Dazu deklarieren wir in der Funk-

tion **RecordsetInArrayMehrdimensional** zunächst wieder das Array **arr** (s. Listing 4). Anschließend redimensioniert die Funktion es, und zwar mit der Anzahl der Datensätze in der ersten Dimension und mit der Anzahl der Felder in der zweiten. Danach durchläuft die Funktion in einer äußeren **Do While**-Schleife die Datensätze des Recordsets und in einer inneren Schleife die Felder – und zwar über die **Fields**-Auflistung des Recordsets. Innerhalb der Schleife füllt die Funktion das Array. Da **fld** keinen Index mitliefert, müssen wir immer noch umständlich eine Zählervariable namens **i** mitlaufen lassen, um die Position für die zweite Dimension anzugeben. Dies können wir auch noch eleganter erledigen, indem wir die innere Schleife als **For...Next**- statt als **For...Each**-Schleife implementieren:

```
Do While Not rst.EOF
    For i = 0 To rst.Fields.Count - 1
        arr(rst.AbsolutePosition, i) _
            = rst.Fields(i)
    Next i
    rst.MoveNext
Loop
```

Die Funktion zum Testen dieser Funktion finden Sie in Listing 5. Die Prozedur erstellt diesmal ein Recordset mit den beiden Feldern **ArtikelID** und **Artikelname** der Tabelle **tblArtikel** und übergibt dieses als Parameter an die Funktion **RecordsetInArrayMehrdimensional**.

Das Ergebnis durchläuft die Prozedur dann in zwei verschachtelten **For... Next**-Schleifen. Bei der Ermittlung der Arraygrenzen der beiden Dimensionen verwenden wir natürlich eine andere Form der **LBound**- und **UBound**-Funktionen. Die untere Grenze der ersten

```
Public Sub Test_RecordsetInArrayMehrdimensional()
    Dim db As DAO.Database
    Dim rst As DAO.Recordset
    Dim i As Integer
    Dim j As Integer
    Dim arr() As Variant
    Set db = CurrentDb
    Set rst = db.OpenRecordset("SELECT ArtikelID, Artikelname FROM tblArtikel", _
        dbOpenDynaset)
    arr = RecordsetInArrayMehrdimensional(rst)
    For i = LBound(arr, 1) To UBound(arr, 1)
        For j = LBound(arr, 2) To UBound(arr, 2)
            Debug.Print arr(i, j);
        Next j
        Debug.Print
    Next i
End Sub
```

Listing 5: Testprozedur zum Einlesen mehrdimensionaler Arrays

Dimension ermitteln wir beispielsweise mit **LBound(arr, 1)**, die der zweiten Dimension mit **LBound(arr, 2)**.

Innerhalb der inneren Schleife sollen die beiden Feldwerte des aktuellen Datensatzes ausgegeben werden. Damit diese in einer Zeile erscheinen, versehen wir die **Debug.Print**-Anweisungen noch mit einem folgenden Semikolon (**;**). Dies verhindert den Sprung in die folgende Zeile.

Allerdings würden so alle Elemente hintereinander in eine einzige Zeile geschrieben werden. Also müssen wir für jeden Datensatz, der ja in der äußeren Schleife durchlaufen wird, noch eine **Debug.Print**-Anweisung ohne Parameter einfügen.

Funktion zur Ausgabe von Arrays

Damit haben wir nun zwei Funktionen geschaffen, mit denen wir Arrays etwa auf Basis eines Recordsets füllen können. Allein die Ausgabe zur Prüfung des Arrays mit dem Durchlaufen der Elemente der einzelnen Dimensionen steckt noch in den beiden Testprozeduren.

Dabei wäre doch gerade die Ausgabe des Inhalts eines Arrays bei Entwickeln hilfreich, denn damit könnte man auf die Schnelle herausfinden, ob das Array auch die erwarteten Daten enthält. Also programmieren wir uns eine kleine Funktion, welche ein Array als Parameter erwartet und dieses im Direktbereich von Access ausgibt.

Die Prozedur heißt **OutputVariant** und ist in Listing 6 zu finden. Die Prozedur deklariert zwei Variablen namens **i** und **j** zum Durchlaufen der verschiedenen Dimensionen des Arrays sowie vier Variablen namens **int1Min**, **int1Max**, **int2Min** und **int2Max**, um die Unter- und Obergrenzen der Dimensionen des Arrays aufzunehmen.

Diese Variablen füllt die Prozedur in den folgenden vier Zeilen, für die durch die Anweisung **On Error Resume Next** die Fehlerbehandlung deaktiviert wurde.

Suchen mit der clsFlexSearch-Klasse

Die Klasse clsFlexSearch lässt sich einfach in Formulare integrieren und fügt einem einfachen Textfeld umfassende Suchmöglichkeiten hinzu. Wie Sie diese Suche implementieren, erfahren Sie im Beitrag Flexible Suche in Formularen (www.access-im-unternehmen.de/965). Der vorliegende Beitrag erläutert die Klasse clsFlexSearch und stellt den Code vor, damit Sie diesen nachvollziehen und ihn gegebenenfalls an Ihre eigenen Bedürfnisse anpassen können.

Alles in einer Klasse

Die Klasse **clsFlexSearch** bietet nicht nur eine praktische Suchfunktion, sondern Sie ist außerdem auch noch ganz leicht zu implementieren.

Der Grund ist, dass der komplette Code in einer einzigen Klasse gekapselt ist und Sie dem Formular, das Sie mit der Funktion ausstatten möchten, lediglich einige Zeilen VBA-Code hinzufügen müssen.

Die Klasse hat die Aufgabe, ein Textfeld mit einer Suchfunktion zu versehen (s. Bild 1). Der Benutzer gibt dort das Suchkriterium ein und die in der Klasse enthaltenen Ereignisprozeduren liefern die Funktionen dazu:

- die automatische Ergänzung der Eingabe des Benutzers bezüglich der zu durchsuchenden Feldnamen und
- die Funktion, die den eingegebenen Ausdruck parst und eine entsprechende **Where**-Klausel formuliert und zurückgibt.

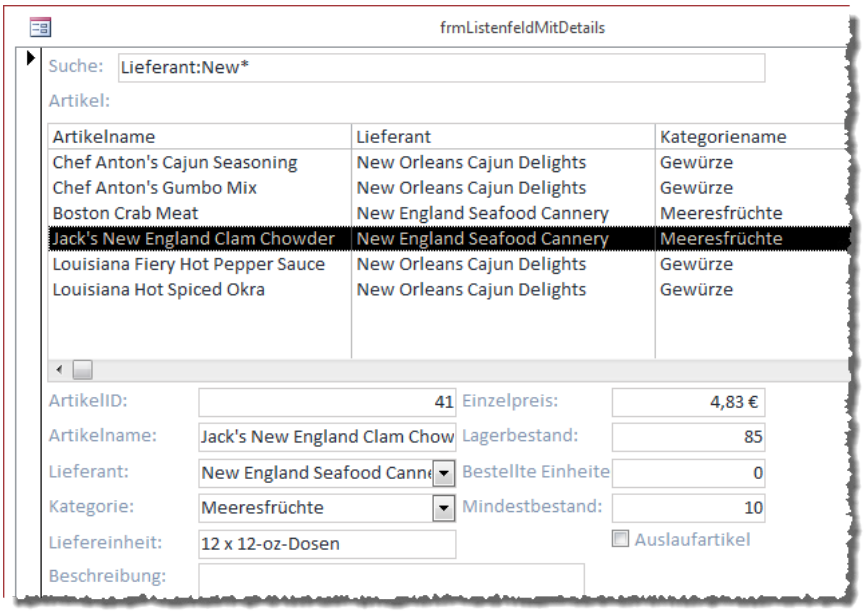

Bild 1: Die Klasse **clsFlexSearch** in Aktion

```
Public Property Let Suchabfrage(strSuchabfrage As String)
    m_Suchabfrage = strSuchabfrage
End Property
```

Listing 1: Öffentliche Eigenschaft für die Angabe der Suchabfrage

Die Klasse clsFlexSearch

Um die Klasse anzulegen, erstellen Sie im VBA-Editor über **Einfügen|Klassenmodul** ein neues Klassenmodul und speichern es unter dem Namen **clsFlexSearch**.

Die Klasse verwendet insgesamt vier Member-Variablen, denen der Benutzer vom Klassenmodul des Formulars aus die entsprechenden Eigenschaftswerte zuweist. Diese Member-Variablen werden wie folgt im Kopf des Klassenmoduls **clsFlexSearch** deklariert:

```
Private m_Suchabfrage As String
Private m_PKFeld As String
Private m_PKFeldEinschliessen As Boolean
Private WithEvents m_Suchfeld As TextBox
```

Eigenschaften zuweisen

Die in diesen vier Variablen gespeicherten Werte sollen über öffentliche Eigenschaften der Implementierung der Klasse **clsFlexSearch** zugewiesen werden, also etwa so:

```
With objFlexSearch
    .Suchabfrage = "qryArtikelsuche"
    .PKFeld = "ArtikelID"
    .PKFeldEinschliessen = False
    Set .Suchfeld = Me!txtSucheNach
End With
```

Diese Eigenschaften zum Entgegennehmen der Werte für die Member-Variablen programmieren wir in der Klasse

clsFlexSearch als **Public Property Let**- beziehungsweise **Public Property Set**-Prozeduren. Die erste sieht wie in Listing 1 aus und nimmt als Parameter einen Wert entgegen, der nach dem Zuweisen in die Variable **m_Suchabfrage** eingetragen wird.

Für Wert-Variablen etwa vom Datentyp **String** verwenden wir das Schlüsselwort **Let**, später für Objektvariablen das Schlüsselwort **Set**.

Die **Public Property Let**-Prozedur **PKFeld** aus Listing 2 erwartet den Namen des Primärschlüsselfeldes und weist diesen der Membervariablen **m_PKFeld** zu.

```
Public Property Let PKFeld(strPKFeld As String)
    m_PKFeld = strPKFeld
End Property
```

Listing 2: Öffentliche Eigenschaft für die Angabe des Primärschlüsselfeldes

```
Public Property Let PKFeldEinschliessen(bolPKFeldEinschliessen As Boolean)
    m_PKFeldEinschliessen = bolPKFeldEinschliessen
End Property
```

Listing 3: Hiermit legen Sie fest, ob der Primärschlüssel für die Suche berücksichtigt wird.

```
Public Property Set Suchfeld(txtSuchfeld As TextBox)
    Set m_Suchfeld = txtSuchfeld
    With m_Suchfeld
        .OnKeyDown = "[Event Procedure]"
        .OnKeyUp = "[Event Procedure]"
    End With
End Property
```

Listing 4: Zuweisen des Such-Textfeldes

Ähnlich arbeitet die **Public Property Let**-Prozedur **PKFeldEinschliessen**. Die an diese Eigenschaft übergebene Variable legt fest, ob der Name des Primärschlüsselfelds im Such-Textfeld automatisch ergänzt und ob es bei der Suche über alle Felder berücksichtigt werden soll (s. Listing 3).

Interessant wird es bei der **Public Property Set**-Prozedur **Suchfeld**. Diese nimmt einen Verweis auf das Textfeld entgegen, das zur Eingabe der Suchbegriffe verwendet werden soll.

Die Prozedur speichert diesen Verweis in der Membervariablen **m_Suchfeld**, welches den Datentyp **Textbox** aufweist und mit dem Schlüsselwort **WithEvents** deklariert wurde.

Auf diese Weise können Sie, auch wenn das Textfeld gar kein Objekt der aktuellen Klasse ist, in dieser Klasse auf seine Ereignisse reagieren. Dazu tragen wir für die den relevanten Ereignissen entsprechenden Eigenschaften jeweils den Wert **[Event Procedure]** ein – hier für die Eigenschaften **OnKeyUp** und **OnKeyDown** (s. Listing 4).

Auf die entsprechenden Ereignisse **Bei Taste ab** und **Bei Taste auf** reagieren wir, um bei der Eingabe von Zeichen erstens zu prüfen, ob hier eine automatische Vervollständigung angewendet werden muss, und zweitens, um auf die Tab-Taste und das Zeichen Doppelpunkt (:) zu reagieren.

Beides soll den als Autovervollständigungs-Wert angegebenen Ausdruck bestätigen und einen Doppelpunkt hinten anfügen, damit der Benutzer hier den Vergleichsausdruck eintragen kann.

Beim Herunterdrücken einer Taste

Nun müssen wir die Ereignisprozedur programmieren, die beim Herunterdrücken einer Taste im Suchen-Textfeld ausgelöst wird. Die Prozedur sieht wie in Listing 5 aus und liefert mit **KeyCode** einen der gedrückten Taste entsprechenden Integer-Wert sowie mit **Shift** einen Wert, der angibt, ob gleichzeitig die **Umschalt**-, die **Alt**- oder die **Strg**-Taste gedrückt wurde.

Die Prozedur speichert zunächst den **KeyCode** in der Variablen **intKeyCode**, die wir im Kopf des Klassenmoduls wie folgt deklarieren:

```
Dim intKeycode As Integer
```

Dies ist nötig, weil wir den **KeyCode** in einigen Fällen auch noch in der später aufgerufenen Ereignisprozedur **m_Suchfeld_KeyUp** benötigen, diese aber in **m_Suchfeld_KeyDown** auf den Wert **0** setzen, damit die eigentliche Wirkung dieses Tastendrucks unterdrückt wird.

Die folgende **Select Case**-Bedingung prüft einige Werte für **intKeyCode**. Der erste Zweig reagiert auf die Werte **9** und **190**. **9** entspricht der **Tab**-Taste, **190** dem Doppelpunkt.

```
Private Sub m_Suchfeld_KeyDown(KeyCode As Integer, Shift As Integer)
    Dim intSelStart As Integer
    Dim intSelLength As Integer
    Dim intStarttemp As Integer
    Dim i As Integer
    Dim intLenTemp As Integer
    Dim strFeldTemp As String
    Dim strSuche As String
    intKeycode = KeyCode
    Select Case intKeycode
        Case 9, 190
            If intKeycode = 190 And Not Shift = acShiftMask Then
                Exit Sub
            End If
            KeyCode = 0
            strSuche = m_Suchfeld.Text
            intSelStart = m_Suchfeld.SelStart
            intSelLength = m_Suchfeld.SelLength
            If Not ArrayGefuellt(strSuchfelder) Then
                SuchfelderEinlesen
            End If
            intStarttemp = InStrRev(strSuche, " ", intSelStart + intSelLength) + 1
            strFeldTemp = Mid(strSuche, intStarttemp, intSelStart + intSelLength)
            If Len(strFeldTemp) > 0 Then
                intLenTemp = Len(strFeldTemp)
                For i = LBound(strSuchfelder) To UBound(strSuchfelder)
                    If strFeldTemp = Left(strSuchfelder(i), intLenTemp) Then
                        m_Suchfeld.Text = Left(m_Suchfeld.Text, intSelStart + intSelLength - 1) & Mid(strSuchfelder(i), _
                            intLenTemp) & ":" & Mid(m_Suchfeld.Text, intSelStart + intSelLength + 1)
                        m_Suchfeld.SelStart = intSelStart + intSelLength + Len(strSuchfelder(i)) - intLenTemp + 1
                        Exit For
                    End If
                Next i
            End If
        Case 13
            Suchen m_Suchfeld.Text
            KeyCode = 0
        Case Else
            Debug.Print intKeycode
    End Select
End Sub
```

Listing 5: Ereignisprozedur, die bei Herunterdrücken einer Taste im Suchen-Textfeld ausgelöst wird

Eigentlich entspricht 190 der Taste, auf der sich der Punkt und der Doppelpunkt befinden, aber auf den Punkt wollen wir nicht reagieren. Also verlassen wir die Prozedur, wenn der zweite Parameter, **Shift**, nicht den Wert 1 (oder **acShift-Mask**) enthält.

In diesem Fall stellen wir **KeyCode** auf den Wert **0** ein, da etwa das Betätigen der **Tab**-Taste sonst zum Verlassen des Textfeldes führen würde. Die Prozedur speichert den aktuellen Inhalt des mit **m_Suchfeld** referenzierten Such-Textfeldes in der Variablen **strSuche**. Die

Variable **intSelStart** nimmt die aktuelle Position der Einfügemarke auf, die Variable **intSelLength** die Länge des aktuell markierten Textes.

Nun benötigen wir eine Liste aller Suchfelder, also derjenigen Felder, die in der für die Eigenschaft **Suchabfrage** angegebenen Abfrage enthalten sind.

Der Performance halber wollen wir diese nicht jedes Mal neu aus der Definition der Abfrage abrufen, deshalb speichern wir diese einmalig in einem Array namens **strSuchfelder**.

```
Private Sub SuchfelderEinlesen()
    Dim db As DAO.Database
    Dim qdf As DAO.QueryDef
    Dim fld As DAO.Field
    Dim i As Integer
    Set db = CurrentDb
    Set qdf = db.QueryDefs(m_Suchabfrage)
    For Each fld In qdf.Fields
        If Not (fld.Name = m_PKFeld And m_PKFeldEinschliessen = False) Then
            ReDim Preserve strSuchfelder(qdf.Fields.Count) As String
            strSuchfelder(i) = fld.Name
            i = i + 1
        End If
    Next fld
End Sub
```

Listing 6: Einlesen von Suchfeldern in ein Array

Um dieses nur einmal zu füllen, prüfen wir zunächst mithilfe der Funktion **ArrayGefuellt**, ob das Array **strSuchfelder** bereits gefüllt ist. Diese Funktion erläutern wir im Detail im Beitrag **Mit Arrays arbeiten** (**www.access-im-unternehmen.de/963**).

Die Variable **strSuchfelder** deklarieren wir übrigens wie folgt im Kopf des Klassenmoduls **clsFlexSearch**:

```
Dim strSuchfelder() As String
```

Wenn das Array noch nicht gefüllt ist, liest die Prozedur **SuchfelderEinlesen** die Felder der unter **m_Suchabfrage** angegebenen Abfrage in das Array **strSuchfelder** ein (s. Listing 6). Die Prozedur öffnet ein **QueryDef**-Objekt auf Basis der angegebenen Abfrage und durchläuft in einer **For Each**-Schleife alle Felder dieser Abfrage. Dabei prüft die Prozedur, ob es sich bei dem aktuellen Feld um das als Primärschlüssel angegebene Feld handelt und dieses gleichzeitig für die Suche berücksichtigt werden soll.

Ist dies der Fall oder handelt es sich um eines der übrigen Felder, fügt die Prozedur den Namen des Feldes zum Array hinzu, wobei zuvor die Anzahl der zulässigen Elemente angepasst wird.

Die Prozedur ermittelt dann ausgehend von der aktuellen Position der Einfügemarke die Position des ersten Leerzeichens links von der Einfügemarke und speichert diese in der Variablen **intStartTemp**. Sollte hierbei kein Leerzeichen gefunden werden, liefert die **InStrRev**-Funktion den Wert **0**. In jedem Fall addiert diese Zeile den Wert **1** zum Ergebnis.

In **strFeldTemp** landet dann die Zeichenkette, die sich zwischen dieser Position und der aktuellen Position der Einfügemarke befindet (inklusive Markierung, falls vorhanden).

Enthält **strFeldTemp** nun einen Wert (**Len(strFeldTemp) > 0**), beginnt die weitere Verarbeitung. Die Variable **intLenTemp** wird nun mit der Länge des in **strFeldTemp** gespeicherten Ausdrucks gefüllt.

In einer **For...Next**-Schleife durchläuft die Prozedur alle Einträge des Arrays **strSuchfelder**, also alle Feldnamen der Suchabfrage. Darin prüft die Prozedur, ob der in **strFeldTemp** gespeicherte Ausdruck mit den ersten **intLenTemp** Zeichen des Feldnamens übereinstimmt.

Wenn **strFeldTemp** also nun den Ausdruck **Art** enthält, weil der Benutzer

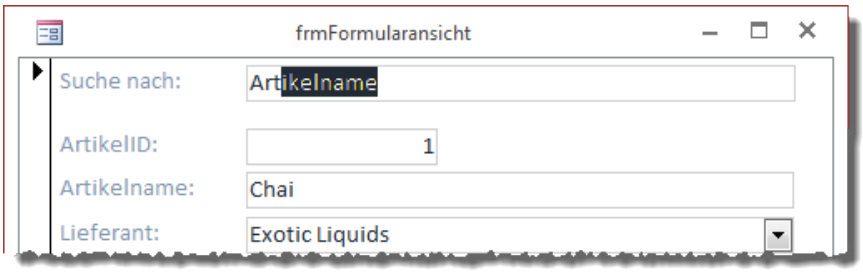

Bild 2: Automatische Ergänzung eines Feldnamens

Flexible Suche in Formularen

Es gibt viele Möglichkeiten, eine Suche über die Datensätze einer Tabelle zu programmieren. Wir wollen in diesem Beitrag eine sehr flexible Variante vorstellen. Damit können Sie sowohl einfach Suchbegriffe eingeben, mit denen dann alle für die Suche angegebenen Felder durchsucht werden. Andererseits soll die Suche auch das gezielte Ermitteln von Datensätzen ermöglichen, deren Felder bestimmte Werte enthalten. Um die Angabe der Feldnamen zu vereinfachen, haben wir der Suche auch noch eine Funktion zur Autovervollständigung hinzugefügt.

Vorweg ist zu erwähnen, dass wir hier keine Suchfunktion mit nicht überschaubaren Möglichkeiten produzieren möchten. Vielmehr soll sie sich an der Praxis orientieren – in diesem Fall an meiner Praxis. Mein Bedarf an einer Suchfunktion, speziell an der hier vorgestellten, resultiert aus einer wachsenden Kundendatenbank.

Früher verwendete ich eine Suchfunktion, die bestimmte Felder nach dem angegebenen Suchbegriff durchforstete. Wenn ich also die ersten Buchstaben des Nachnamens eines Kunden eingegeben habe, sollte das Formular nach den Kunden mit diesem Nachnamen gefiltert werden.

Da ich aber nicht nur das Feld **Nachname**, sondern auch noch andere Felder nach dem eingegebenen Suchbegriff durchsuchte, erschienen plötzlich Suchtreffer, bei denen etwa der Straßenname mit den gleichen Buchstaben begann wie der Nachname.

Bei einer überschaubaren Anzahl von Kundendatensätzen ist da kein Problem: Man blättert halt schnell durch die gefilterten Datensätze und landet so schnell bei dem gesuchten Kunden. Mit einer kleinen Anzahl Kunden klappte das recht gut. Mit wachsender Anzahl funktionierte es allerdings immer schlechter, da auch die Anzahl der Suchtreffer wuchs.

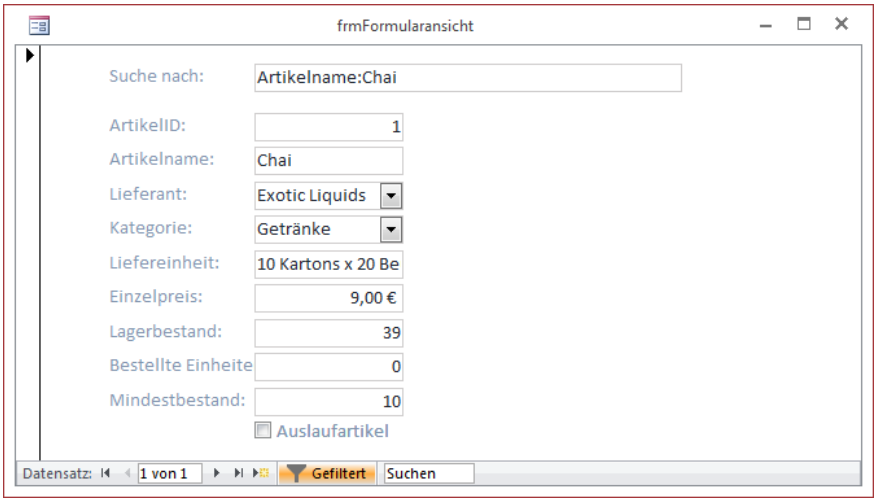

Bild 1: Beispiel für die Suche in einem Formular in der Formularansicht

Da ich keine Lust hatte, schon wieder eine neue Suchfunktion zu programmieren, machte ich aus der Not eine Tugend und suchte fortan nach Elementen in den Kundendaten, die eine kleinere Treffermenge versprachen.

Wenn ein Kunde also etwa den Nachnamen **Meier** trägt, habe ich erst gar nicht mehr nach dem Nachnamen gesucht, sondern den Vornamen als Suchbegriff eingegeben. Dumm nur, wenn der gesuchte Kunde etwa **Michael Meier** heißt – so waren auf jeden Fall eine Menge Treffer zu erwarten.

Auch interessant waren Kunden mit einem Nachnamen, der auch als Vorname verwendet wird – auch hier fallen eine Reihe Treffer mehr an als man eigentlich durchblättern möchte.

Also musste eine neue Suchfunktion her: Eine, die zwar immer noch die flexible Suche über alle festgelegten Suchfelder erlaubte, aber die auch die konkrete Angabe der zu durchsuchenden Felder erlaubte (s. Bild 1). Wenn man also etwa einfach den Nachnamen **Meier** eingibt, sollten alle festgelegten Felder nach dieser Zeichenkette durchsucht werden.

Um nur solche Datensätze zu finden, die den Wert **Meier** nur im Feld **Nachname** enthalten, soll der Benutzer diesen Ausdruck angeben können:

```
Nachname:Meier
```

Wenn die Suche zu viele Treffer liefert, kann der Benutzer das Ergebnis weiter einschränken – etwa durch zusätzliche Angabe des Vornamens:

Vorname:Michael Nachname:Meier

„Und" oder „Oder"

Die große Frage, die sich nun stellt, ist folgende: Benötigen wir eine Und-Verknüpfung für alle Vergleichskriterien oder eine Oder-Verknüpfung – oder gar beide? Und sollte man gegebenenfalls sogar Klammern zum Einstellen der Priorität der Verknüpfungen hinzufügen können? Normalerweise sollte eine Und-Verknüpfung ausreichen. Wenn man zusätzlich zum gesuchten Nachnamen noch den Vornamen eingibt, soll das Suchergebnis auch nur die Datensätze enthalten, bei denen beide Suchkriterien zutreffen. Dies gilt zumindest dann, wenn man nach genau einem Kundendatensatz sucht. Aber auch hier gibt es Ausnahmen: Es kann ja zum Beispiel sein, dass Sie nicht wissen, ob der Nachname des Kunden nun in der Lieferadresse (etwa im Feld **Nachname_Liefer**) oder in der Rechnungsadresse (zum Beispiel **Nachname_Rechnung**) gespeichert ist. Dann wäre es natürlich sinnvoll, wenn die Suchfunktion die beiden Suchausdrücke per **Or** verknüpfen würde. Also enthält die Lösung die Möglichkeit, **And** und **Or** als Verknüpfungsoperatoren zu verwenden.

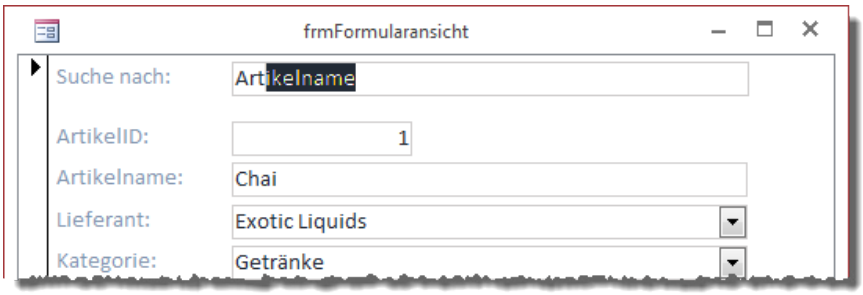

Bild 2: Automatische Ergänzung von Suchfeldern

In allen Feldern suchen

Und die Suchfunktion soll noch ein weiteres interessantes Feature liefern – nämlich die Suche über alle möglichen Felder. Dies soll immer dann geschehen, wenn der Benutzer einen Suchbegriff ohne Angabe des zu durchsuchenden Feldes angibt.

Die Eingabe des Ausdrucks **Chai** ohne vorheriges **Artikelname:** soll also alle Felder nach diesem Ausdruck durchsuchen.

Automatische Ergänzung

Eine der wichtigsten Funktionen der Suche ist die automatische Ergänzung bei der Eingabe der zu durchsuchenden Felder. Wenn die Datenherkunft also etwa ein Feld namens **Artikelname** enthält und der Benutzer gibt die Buchstaben **Art** ein, dann soll das Suchtextfeld automatisch die fehlenden Buchstaben (hier **ikelname**) anzeigen und gegebenenfalls nach Betätigen der **Tab**-Taste oder nach Eingabe des Doppelpunkts (**:**) hinzufügen (s. Bild 2).

Einschränkung

In der aktuellen Version funktioniert die Suche nach Textpassagen in Feldern des Datentyps **Text** oder **Memo** problemlos. Auch die Suche nach Zahlen oder Datumsangaben gelingt, allerdings nur für den Gleich-Operator. Größer/Kleiner und ähnliche Operatoren sind hier noch nicht umgesetzt.

Einsatz der Lösung

Ein Beispiel für den Einsatz der Lösung liefert das Formular **frmFormularansicht**. Hier haben wir ein Formular, das die Datensätze der Tabelle **tblArtikel** in der Formularansicht anzeigt, mit einem

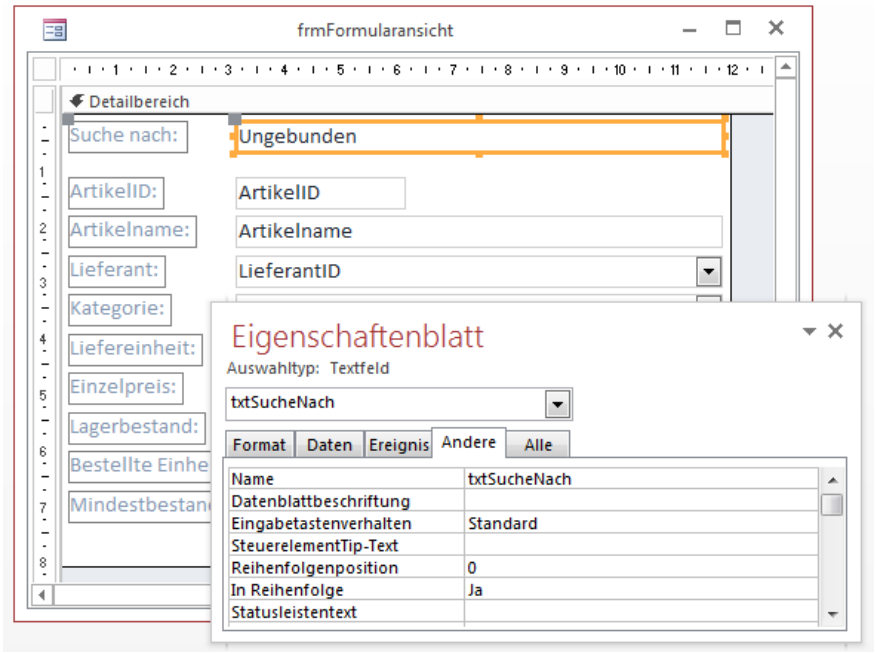

Bild 3: Hinzufügen eines Suchfeldes zum Formular

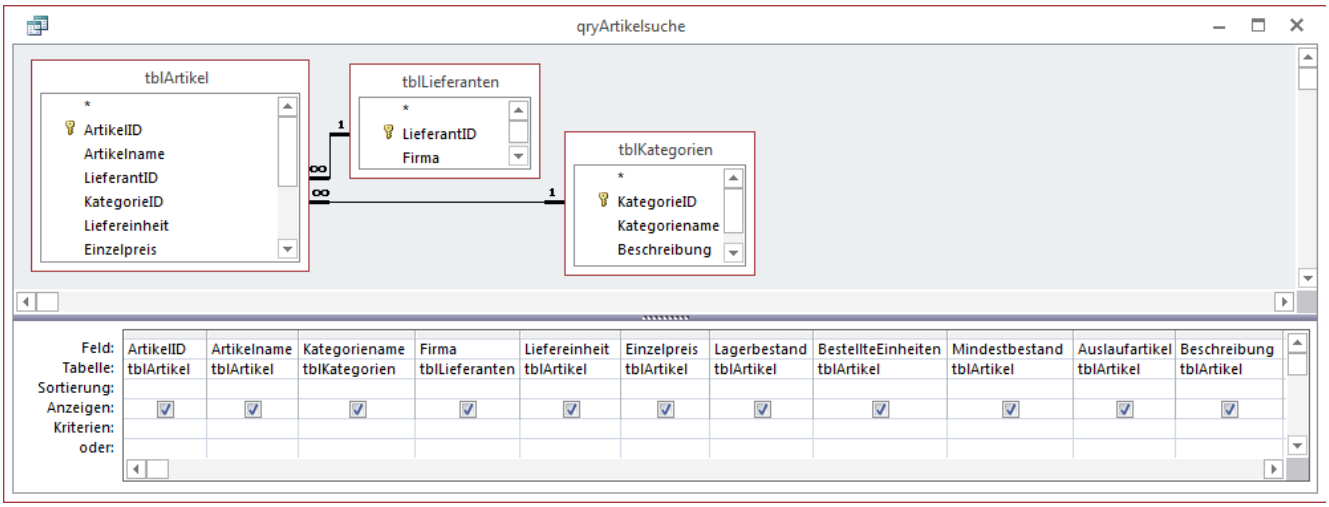

Bild 4: Abfrage mit den zu durchsuchenden Feldern

Suchfeld versehen. Im Beispiel soll die Suche alle Artikel liefern, deren Feld **Artikelname** den Wert **Chai** enthält – was genau einen Treffer bringt (siehe unten in der Abbildung: **Datensatz 1 von 1**).

Welche Schritte sind zum Anwenden der Lösung nötig? In Kurzfassung sind es die folgenden:

- Hinzufügen eines Textfeldes zur Eingabe des Suchbegriffs
- Erstellen einer Abfrage, die alle zu durchsuchenden Felder enthält
- Anlegen einer Ereignisprozedur für das Ereignis **Beim Laden**, das die Suchfunktion initialisiert
- Anlegen einer Ereignisprozedur für das Ereignis **Beim Entladen**, um die Suchfunktion zu beenden
- Anlegen einer Ereignisprozedur, die nach dem Start der Suche ausgelöst wird und die Aktion zum Filtern des Formulars, Unterformulars oder Listenfeldes enthält

Textfeld für die Suche hinzufügen

Im ersten Schritt fügen Sie dem Formular an der gewünschten Stelle ein Textfeld hinzu, das Sie zur Eingabe des Suchbegriffs nutzen möchten – beispielsweise im oberen Bereich des Formulars. Im ersten Beispiel des Formulars **frmFormularansicht**, das die Daten der Tabelle **tblArtikel** anzeigt, nennen wir dieses Steuerelement **txtSucheNach** (s. Bild 3).

Abfrage mit den zu durchsuchenden Feldern erstellen

Neben der Datenherkunft für das Formular benötigen wir noch eine Abfrage, welche die zu durchsuchenden Felder enthält. Warum aber sollte man nicht einfach die gleiche Abfrage verwenden, die auch als Datenherkunft für das zu durchsuchende Formular dient? Aus einem ganz einfachen Grund: Im Beispiel des Formulars **frmFormularansicht** etwa werden nur Daten der Tabelle **tblArtikel** angezeigt. Nun könnte man annehmen, dass dies auch für die Ermittlung der zu durchsuchenden Felder gilt. Das ist manchmal der Fall, aber nicht hier: Die Datenherkunft des Formulars liefert nämlich mit den beiden Feldern **LieferantID** und **KategorieID** zwei Nachschlagefelder, die ja nur die Werte der Primärschlüsselfelder der verknüpften Einträge aus den Tabellen **tblLieferanten** und **tblKategorien** enthalten. Wenn wir nun als Suchbegriff etwa für die Kategorie den Wert **Getränke** eingeben, finden wir in der Datenherkunft des Formulars natürlich keinen passenden Datensatz, weil das Feld **KategorieID** dort nur Zahlenwerte wie **1**, **2**, **3** et cetera enthält.

Wenn wir aber nach den eigentlichen Werten hinter den als Nachschlagefeld definierten Feldern suchen möchten, müssen wir die verknüpften Tabellen noch zur Datenherkunft für die Suchfunktion hinzufügen. Dies sieht dann wie in Bild 4 aus: Hier haben wir einer neuen Abfrage namens **qryArtikelsuche** nicht nur die Tabelle **tblArtikel**, sondern auch noch die beiden verknüpften Tabellen **tblKategorien** und **tblLieferanten** hinzugefügt. Nachdem alle benötigten Tabellen in der Abfrage enthalten sind, ist es wichtig, nur die Felder zum Entwurfsraster hinzuzufügen, die auch für die Suche herangezogen werden sollen – plus dem Primärschlüsselfeld der

eigentlich zu durchsuchenden Tabelle, also **ArtikelID**.

In diesem Fall fügen wir also zunächst die Felder **ArtikelID** und **Artikelname** hinzu. Dann folgt in der Tabelle **tblArtikel** das Feld **LieferantID**, welches wir aber nicht zur Datenherkunft hinzufügen – wir wollen ja nicht nach den Primärschlüsselwerten der Tabelle **tblLieferanten** suchen.

Stattdessen ziehen Sie das Feld **Firma** aus der Tabelle **tblLieferanten** in das Entwurfsraster. Auch das nächste Feld namens **KategorieID** benötigen wir nicht, sondern das Feld **Kategoriename** der verknüpften Tabelle **tblKategorien**. Die übrigen Felder der Tabelle **tblArtikel** können wir dann wieder direkt in das Entwurfsraster der Abfrage ziehen.

Hier fehlt nur noch eine kleine Optimierung: Im Formular wird etwa das Feld **KategorieID** mit der Bezeichnung **Kategorie** angezeigt. Der Name des Feldes in der Abfrage, die für die Suche verwendet werden soll, lautet hingegen **Kategoriename**. Ähnlich sieht es für das Feld **Lieferant** aus, das in der verknüpften Tabelle **Firma** heißt. In beiden Fällen versehen wir die Felder der Suchabfrage mit einem entsprechenden Alias: **Kategoriename** wird durch **Kategorie:Kategoriename** ersetzt, **Firma** durch **Lieferant:Firma** (s. Bild 5).

Diese Abfrage taucht nirgends im Formular auf. Sie wird lediglich für zwei Zwecke benötigt: Erstens als Lieferant für die Feldnamen, die im Suchtextfeld bei der Eingabe der Anfangsbuchstaben automatisch ergänzt werden sollen, zweitens als Abfrage, welche zur Ermittlung der gesuchten Datensätze dient.

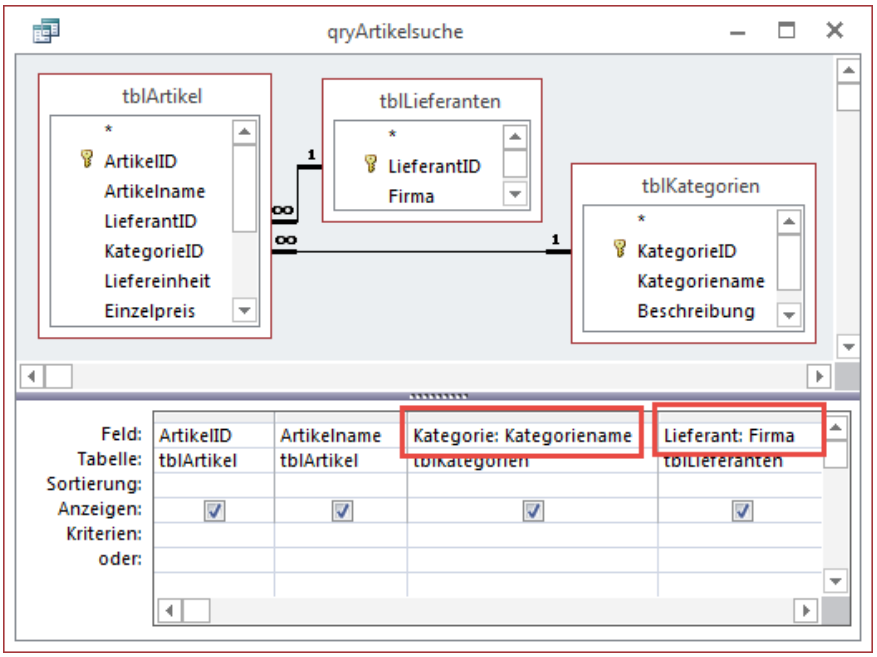

Bild 5: Optimierung: Anpassen der Felder der verknüpften Tabellen an den Feldnamen der Haupttabelle

Beim Laden: Suche initialisieren

Die Suchfunktion steckt komplett in einem einzigen Klassenmodul. Dieses müssen Sie im Klassenmodul des Formulars instanzieren und ihm einige Informationen übergeben, die es für die Implementierung der Suchfunktion für das Formular benötigt.

Dazu sind zwei Schritte nötig. Der erste ist die Deklaration einer Objektvariablen namens **objFlexSearch** im Kopf des Klassenmoduls des Formulars:

```
Dim WithEvents objFlexSearch
                    As clsFlexSearch
```

Der zweite ist das Hinzufügen einer Ereignisprozedur, die durch das Ereignis **Beim Laden** des Formulars ausgelöst wird. Für dieses hinterlegen Sie den folgenden Code:

```
Private Sub Form_Load()
    Set objFlexSearch = New clsFlexSearch
    With objFlexSearch
        .Suchabfrage = "qryArtikelsuche"
        .PKFeld = "ArtikelID"
        .PKFeldEinschliessen = False
        Set .Suchfeld = Me!txtSucheNach
    End With
End Sub
```

Die Eigenschaften der Klasse **clsFlexSearch** werden im VBA-Fenster per IntelliSense bereitgestellt und können so leicht gefüllt werden (s. Bild 6).

Die Eigenschaften haben die folgende Bedeutung:

- **Suchabfrage**: Name der gespeicherten Abfrage, die wir soeben erstellt haben
- **PKFeld**: Name des Primärschlüsselfeldes in der Suchabfrage, nach der die Datensätze der im Formular angezeigten Datenherkunft gefiltert werden sollen