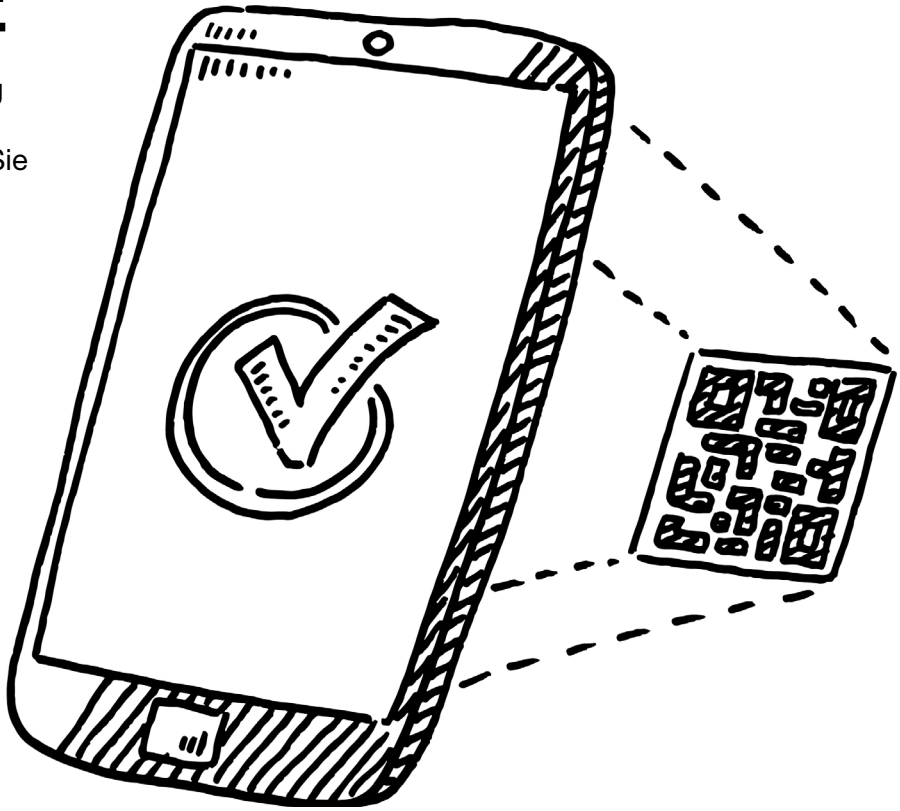
HAUFE.
Ausgabe 05/2022

ACCESS
IM UNTERNEHMEN

RECHNUNGEN BEZAHLEN PER QR-CODE

Statten Sie Ihre Rechnungserstellung mit Funktionen zum Hinzufügen von QR-Bezahlcodes aus und ersparen Sie Ihren Kunden viel Zeit (ab S. 65).



In diesem Heft:

FORMULARPOSITION MERKEN

Sorgen Sie dafür, dass sich Access die Positionen von Objekten beim Schließen merkt und beim Öffnen wiederherstellt.

SEITE 2

SETUP MIT DER ACCESS-RUNTIME

Statten Sie Benutzer ohne Access-Vollversion mit der kostenlosen Runtime aus – ganz einfach per Anwendungssetup.

SEITE 43

RECHNUNGEN VERWALTEN

Unsere Musterlösung zur Rechnungsverwaltung wächst weiter – diesmal um ein Bestellformular und ein Kundendetailformular.

SEITE 14

Rechnungen einfacher bezahlen

In letzter Zeit wird die Vereinfachung der Bezahlung und auch Verarbeitung von Rechnungen forciert. Erst war es die XRechnung, über die wir bereits in Ausgabe 6/2020 berichtet haben. Die XRechnung ist eine Version einer Rechnung im XML-Format, die leicht beispielsweise in eine Datenbank eingelesen werden kann. Nun folgt mit dem EPC-QR-Code eine weitere Vereinfachung auch für Privatpersonen. Dieser Code wird auf üblichen Rechnungen aufgebracht, damit der Empfänger die Überweisungsdaten mit seiner Banking-App einlesen und die Überweisung schnell ausführen kann.

Beides greifen wir in der aktuellen Ausgabe auf. Während wir in Ausgabe 6/2020 berichtet haben, wie Sie XRechnungen aus den Daten Ihrer Datenbank erstellen können, beschreiben wir im Beitrag **XRechnung, Teil 2: Rechnungen einlesen** ab Seite 52, wie Sie die Daten einer XRechnung in die Tabellen Ihrer Datenbank einlesen können.

Die zweite Lösung dieser Ausgabe dreht sich um das Vereinfachen des Bezahlvorgangs nach dem Erhalt einer Rechnung. Viele Onlinebanking-Apps bieten bereits Schaltfläche mit Texten wie **Fotoüberweisung** oder **QR-Code scannen**. Nach dem Anklicken bieten diese die Möglichkeit, einen auf der Rechnung abgebildeten QR-Code zu scannen, der alle für die Überweisung wichtigen Daten wie Verwendungszweck, Empfängerdaten und den zu überweisenden Betrag enthält. Und damit beginnt unser Teil der Aufgabe: Das Bereitstellen eines solchen QR-Codes auf den mit Access erstellten Rechnungsberichten. Die notwendigen Schritte zum Erstellen eines solchen QR-Codes beschreiben wir ab Seite 65 im Beitrag **EPC-QR-Code per COM-DLL erstellen**.

Wer viele Formulare und andere Elemente in seiner Access-Datenbank so nebeneinander anordnet, dass er optimal damit arbeiten kann, ärgert sich vielleicht, wenn er nach dem Schließen und erneuten Öffnen seine Wunschkonfiguration immer wieder neu herstellen muss. Wenn das bei Ihnen so ist, haben wir perfekte Lösung! Im Beitrag **Objektpositionen speichern und wiederherstellen** beschreiben wir ab Seite 2, wie Sie Ihrer Anwendung eine Funktion hinzufügen, die Ihr Problem dauerhaft löst. Diese Lösung enthält eine Tabelle sowie zwei Prozeduren. Die eine wird beim Schließen der Datenbank ausgelöst und schreibt die aktuellen Positionen der geöffneten Objekte in die Tabelle. Die andere wird beim Öffnen der Anwendung gestartet und öffnet die in der Tabelle gespeicherten Objekte wieder – samt Wiederherstellung von Position und Größe.

In unserer Beitragsreihe zur Rechnungsverwaltung, die später darin gipfeln wird, dass wir Rechnungen mit dem oben erwähnten QR-Code erstellen werden, finden Sie zwei neue Teile. Im ersten Teil namens **Rechnungsverwaltung: Bestellformular** erläutern wir ab Seite 14, wie wir per Formular neue Bestellungen und die entsprechenden Bestellpositionen erfassen können.

Der zweite neue Teil mit dem Titel **Rechnungsverwaltung: Kundendetails** beschreibt ab Seite 31, wie wir die Kundendetails inklusive der bereits für diesen Kunden erfassten Bestellungen verwalten können.

Und schließlich zeigt unser Autor Chris Jüngling ab Seite 43 im Beitrag **Access-Applikation mit Runtime installieren**, wie Sie für Benutzer, die kein Access auf dem Rechner installiert haben, noch die kostenlose Runtime-Version gemeinsam mit der zu installierenden Datenbankanwendung in einem Setup verpacken.

Nun viel Spaß beim Lesen!

Ihr André Minhorst

Objektpositionen speichern und wiederherstellen

Neulich fragte ein Leser, ob und wie man die Position von Objekten im Access-Fenster speichern und wiederherstellen könne. Der Hintergrund ist, dass er immer wieder mühsam Tabellen, Abfragen und andere Objekte zu einem Arbeitsbereich zusammengestellt hat und wenn er die Anwendung schließt, ist die ganze Arbeit dahin – und am nächsten Tag muss er die Objekte erneut anordnen. Ich fühlte mich ein wenig an Zeiten erinnert, wo man zwar einen Homecomputer zum Programmieren, aber kein Gerät zum Speichern der eingetippten Spiele aus den Computermagazinen hatte ... Da sich die Zeiten zum Glück geändert haben, zeige ich in diesem Beitrag, wie Sie die Position und Größe der beim Schließen einer Datenbank geöffneten Objekte abspeichern und beim nächsten Öffnen wieder herstellen können.

Aufgabenstellung

Aus den Anforderungen des Lesers ergeben sich die folgenden Aufgabenstellungen:

- Herausfinden, wie wir die Position und Größe aller zu einem bestimmten Zeitpunkt geöffneten Objekte ermitteln können
- Definieren einer Tabelle zum Speichern der geöffneten Objekte mit Name und Objekttyp und ihrer Position und Größe sowie der aktuellen Ansicht
- Prozedur zum Speichern dieser Informationen in einer geeigneten Tabelle in der jeweilige Datenbankanwendung
- Herausfinden, wie wir die abgespeicherten Objekte wieder öffnen und die Position und die Größe zu einem bestimmten Zeitpunkt wiederherstellen können
- Festlegen eines Automatismus, die Position und Größe der Objekte beim Schließen der Datenbank zu speichern.
- Festlegen eines weiteren Automatismus, um die Position und Größe bei Öffnen der Datenbankanwendung wiederherzustellen

Größe und Position der geöffneten Objekte ermitteln

Die erste Aufgabe ist bereits die anspruchsvollste: Wie können wir alle geöffneten Objekte ermitteln und wie finden wir die aktuelle Position und Größe dieser Objekte heraus?

Für diese Aufgabe gibt es keine Lösung, mit der wir alle Objekte gleichermaßen behandeln können. So gibt es zwar Auflistungen namens **Forms** und **Reports**, mit denen direkt auf die aktuell geöffneten Formulare und Berichte zugegriffen werden kann.

Entsprechende Auflistungen für Tabellen und Abfragen beispielsweise namens **Tables** oder **Queries** suchen wir jedoch vergeblich.

Wenn wir jedoch nach diesen Schlüsselwörtern im Objektkatalog des VBA-Editors suchen, finden wir schnell passende Einträge, nämlich **AllTables** und **AllQueries** (siehe Bild 1).

Damit haben wir zumindest schon einmal Auflistungen für alle Objekttypen gefunden – nun schauen wir uns an, wie wir an die jeweils gewünschten Informationen wie Name, aktuelle Ansicht, Position vom linken und oberen Rand, Höhe und Breite gelangen.

Formulare und Berichte analysieren

Bei Formularen und Berichten macht Access es uns ein wenig leichter als bei Tabellen und Abfragen, die wir uns im Anschluss anschauen. Mit der **Forms**- und der **Reports**-Auflistung können wir direkt die aktuell geöffneten Objekte referenzieren. Mit der folgenden Prozedur durchlaufen wir beispielsweise alle Formulare, die aktuell geöffnet sind:

```
Public Sub Formulare()
    Dim frm As Form
    For Each frm In Forms
        Debug.Print frm.Name, _
          frm.WindowLeft, frm.WindowTop, _
          frm.WindowWidth, frm.WindowHeight
    Next frm
End Sub
```

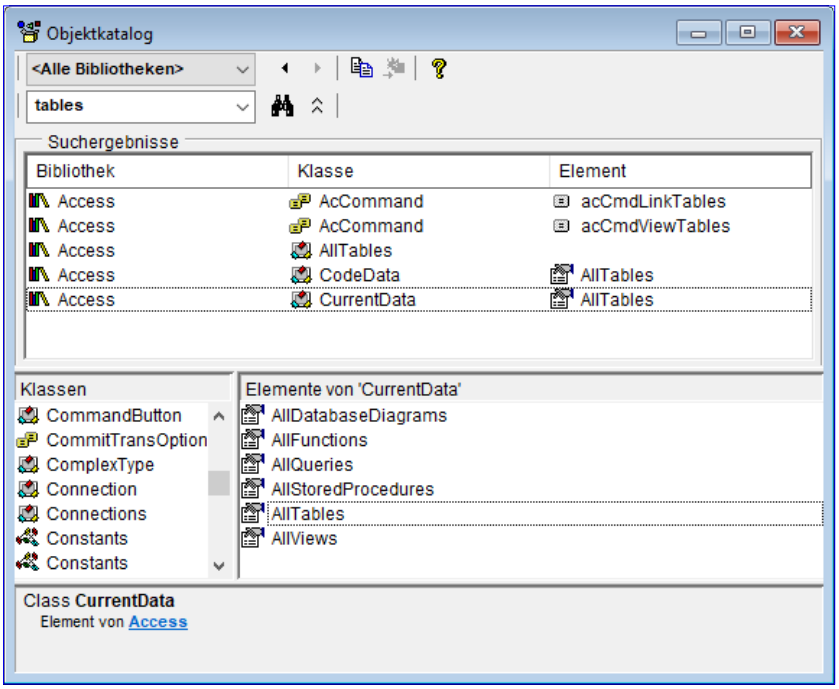


Bild 1: Finden von geeigneten Auflistungen für den Zugriff auf Tabellen und Abfragen

Die Werte der Eigenschaften **WindowLeft**, **WindowTop**, **WindowWidth** und **WindowHeight** liefert Access in der Einheit **Twips**. Vorneweg: Diese Eigenschaften sind schreibgeschützt und wir können diese später nicht nutzen, um die Größe und die Position der Formulare und Berichte wiederherzustellen. Allerdings arbeitet auch die Methode, die wir dazu später nutzen werden, mit Angaben in dieser Einheit. Daher verzichten wir an dieser Stelle auf eine Beschreibung, was Twips genau sind.

Den Namen des jeweiligen Formulars oder Berichts lesen wir einfach aus der Eigenschaft **Name** aus. Spannend ist nun noch, die aktuelle Ansicht zu ermitteln.

Formulare können in den Ansichten **Einzelnes Formular**, **Endlosformular**, **Datenblatt** oder **Geteiltes Formular** angezeigt werden (siehe Bild 2). Bei Berichten kommen noch die Ansichten **Berichtssicht** und **Seitenansicht** hinzu.

Diese Eigenschaft können wir per VBA einerseits über **CurrentView** ermitteln. **CurrentView** liefert die folgenden Werte:

- **0** (**acCurViewDesign**): Entwurfsansicht

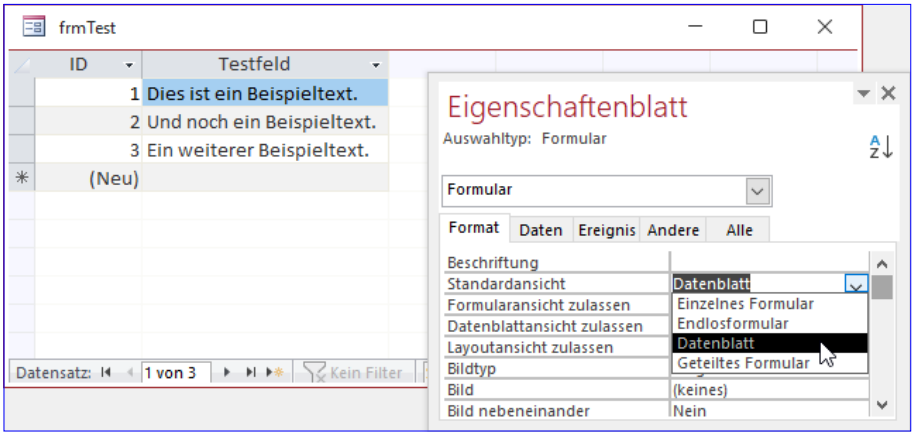


Bild 2: Mögliche Ansichten für ein Formular

- **1** (**acCurViewFormBrowse**): Formularansicht
- **2** (**acCurViewDatasheet**): Datenblattansicht
- **5** (**acCurViewPreview**): Seitenansicht (bei Berichten)
- **6** (**acCurViewReportBrowse**): Berichtssicht (bei Berichten)
- **7** (**acCurViewLayout**): Layoutansicht

Aber Moment – das sind ja gar nicht die Werte, die wir für die Eigenschaft **Standardansicht** aus dem Eigenschaftenblatt auswählen können. Diese können wir der VBA-Eigenschaft **DefaultView** eines **Form**-Objekts entnehmen und die Werte stimmen nicht mit den Zahlenwerten für die Eigenschaft **CurrentView** überein.

Im Gegenteil – es gibt sogar für Formulare und Berichte teilweise gleiche Werte mit unterschiedlicher Bedeutung:

- **0**: Einzelnes Formular
- **1**: Endlosformular
- **2**: Datenblatt
- **3**: PivotTable
- **4**: PivotChart
- **5**: Geteiltes Formular

Für Berichte gibt es die folgenden beiden Werte:

- **0**: Seitenansicht
- **1**: Berichtsansicht

Wie aber kommen wir an die aktuelle Ansicht? Immerhin kann der Benutzer diese ja, wenn die möglichen Ansichten nicht auf eine Ansicht eingeschränkt ist, wechseln. Daher verwenden wir die Eigenschaft **CurrentView**, um die aktuelle Ansicht zu ermitteln.

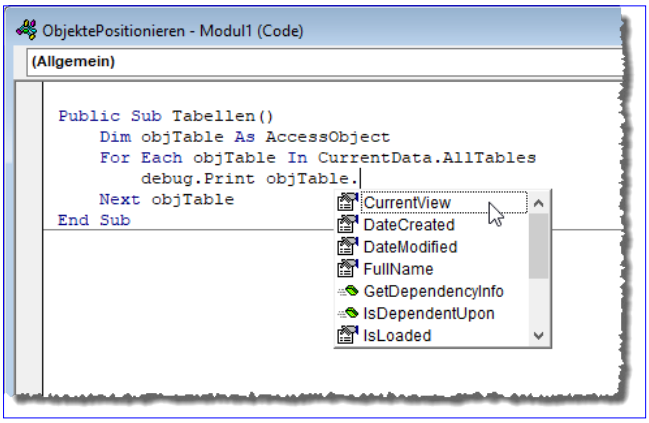


Bild 3: Erkunden der Eigenschaften von **AccessObject**-Elementen

Tabellen und Abfragen analysieren

Etwas komplizierter wird die Analyse bei Tabellen und Abfragen. Wie bereits erwähnt, gibt es keine Auflistung, die alle derzeit geöffneten Tabellen oder Abfragen auflistet.

Also schauen wir, was wir mit den beiden Auflistungen **AllTables** und **AllQueries**, die wir weiter oben im Objektkatalog gefunden haben, für unsere Zwecke nutzen können.

Um diese Auflistungen zu durchlaufen, wollen wir zunächst herausfinden, welchen Datentyp die damit referenzierten Elemente überhaupt aufweisen. Dazu greifen wir einfach auf das erste Element einer solchen Auflistung zu und lassen uns den Objekttyp mit der Funktion **Typename** im Direktbereich ausgeben:

```
? Typename(CurrentData.AllTables(0))
AccessObject
```

Jetzt, da wir den Objekttypen kennen, können wir die Elemente direkt in einer **For Each**-Schleife durchlaufen und auch per IntelliSense auf die Eigenschaften der Elemente zugreifen (siehe Bild 3).

Um herauszufinden, welche Objekte geöffnet sind und in welcher Ansicht sie dann angezeigt werden, verwenden wir die folgende Prozedur:

```
Public Sub GeoeffneteTabellenDurchlaufen()
    Dim objTable As AccessObject
    For Each objTable In CurrentData.AllTables
        Debug.Print objTable.Name;
        If objTable.IsLoaded Then
            Debug.Print objTable.CurrentView
        Else
            Debug.Print
        End If
    Next objTable
End Sub
```

Damit erhalten wir beispielsweise folgendes Ergebnis, wenn die beiden Tabellen **tblObjecttypes** und **tblTest** in der Datenblattansicht und die Tabelle **tblObjects** in der Entwurfsansicht geöffnet sind:

```
MSysAccessStorage
MSysACEs
... weitere nicht geöffnete Systemtabellen
MSysRelationships
MSysResources
tblObjects 0
tblObjecttypes 2
tblTest 2
```

Interessant sind für Tabellen die folgenden Ansichten:

- **0**: Entwurfsansicht
- **2**: Datenblattansicht

Abfragen durchlaufen wir auf ähnliche Weise – wir verwenden hier lediglich die Auflistung **AllQueries**:

```
Public Sub GeoeffneteAbfragenDurchlaufen()
    Dim objQuery As AccessObject
    For Each objQuery In CurrentData.AllQueries
        Debug.Print objQuery.Name;
        If objQuery.IsLoaded Then
            Debug.Print objQuery.CurrentView
        Else
            Debug.Print
        End If
    Next objQuery
End Sub
```

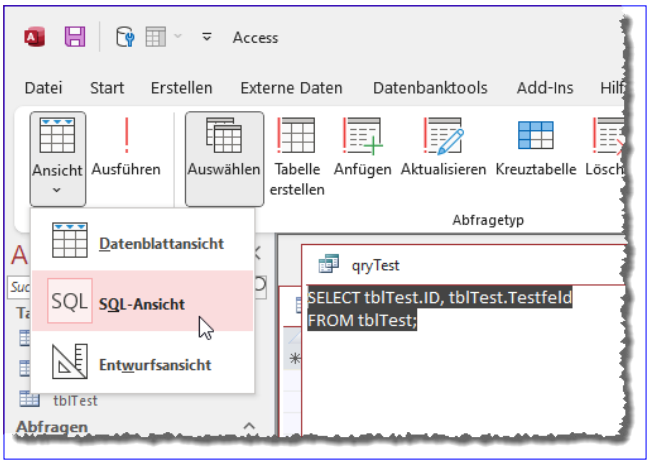


Bild 4: Die SQL-Ansicht einer Abfrage

Interessant ist hier, dass es eigentlich noch eine weitere Ansicht gibt, nämlich die SQL-Ansicht (siehe Bild 4). Wenn wir eine Abfrage in dieser Ansicht öffnen und die Prozedur **GeoeffneteAbfrageDurchlaufen** starten, gibt diese ebenfalls den Wert **0** für die Eigenschaft **CurrentView** zurück.

Davon ausgehend, dass der Benutzer nicht den Zustand von in der SQL-Ansicht geöffneten Abfragen speichern möchte, stellt das aber auch im Rahmen dieses Beitrags kein Problem dar.

Position und Größe von Tabellen und Abfragen ermitteln

Nachdem wir die Größe und Position von Formularen und Berichten direkt über die Eigenschaften der **Form**- beziehungsweise **Report**-Objekte ermitteln können, schauen wir uns nun die Tabellen und Abfragen an. Da das **AccessObject**-Element keine diesbezüglichen

Eigenschaften offeriert, müssen wir einen kleinen Umweg gehen, um die Informationen zu beschaffen.

Ausgehend davon, dass wir ohnehin nur geöffnete Elemente verwenden wollen, können wir dann das **Screen**-Objekt nutzen, welches uns Informationen über die jeweils aktiven Objekte liefert, also die Objekte, die aktuell den Fokus besitzen. Mit dem **Screen**-Objekt können wir über die folgenden Eigenschaften auf die jeweiligen Objekte zugreifen:

- **Screen.ActiveDatasheet**: Aktive Tabellen und Abfragen in der Datenblattansicht
- **Screen.ActiveForm**: Aktives Formular
- **Screen.ActiveReport**: Aktiver Bericht

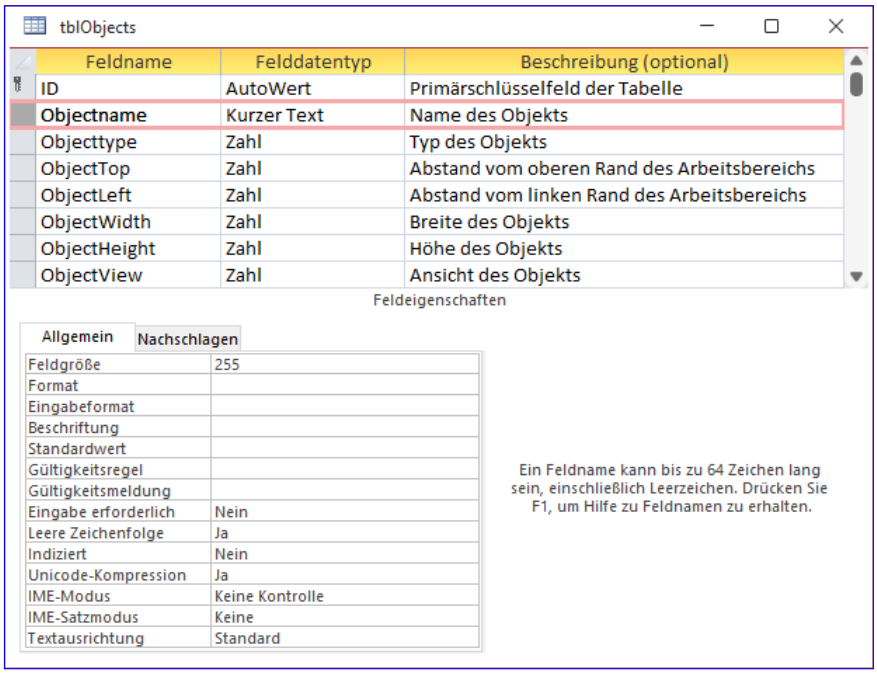


Bild 5: Tabelle zum Speichern der Objekteigenschaften

Hier sehen wir also eine weitere Einschränkung bezüglich der Ansichten, die wir beim Speichern von Position und Größe berücksichtigen können: Die Entwurfsansicht von Tabellen oder Abfragen fällt weg, da wir diese nicht mit den **Active...**-Eigenschaften referenzieren können.

Die Eigenschaft **ActiveDatasheet** liefert tatsächlich nur einen Objektverweis zurück, wenn aktuell eine Tabelle oder Abfrage in der Datenblattansicht geöffnet ist. Aber auch das ist kein Problem, zumindest nicht gemessen an dem Wunsch des Lesers, die Position und Größe von in der Datenblattansicht geöffneten Elementen zu ermitteln und zu speichern.

Allerdings hilft uns **Screen.ActiveDatasheet** noch nicht weiter, wenn das Objekt, das wir untersuchen wollen, nicht den Fokus hat. Deshalb müssen wir, bevor wir auf eine geöffnete Tabelle in der Datenblattansicht zugreifen wollen, zunächst den Fokus auf diese Tabelle verschieben. Dass sie überhaupt geöffnet ist, können wir voraussetzen, denn wir wollen ja nur die Position und Größe von Tabellen ermitteln, die derzeit im Arbeitsbereich sichtbar sind.

Den Namen der zu untersuchenden Tabelle kennen wir über die **Name**-Eigenschaft auch. Also können wir die **DoCmd.SelectObject**-Methode nutzen, um die zu untersuchende Tabelle oder Abfrage in den Fokus zu setzen – beispielsweise so:

```
DoCmd.SelectObject acTable, objTable.Name
```

Tabelle zum Speichern der Objekteigenschaften

Als Nächstes erstellen wir die Tabelle, in der wir die Größe, Position und Ansicht der aktuell geöffneten Objekte speichern wollen.

Welche Informationen wir speichern wollen, haben wir bereits ausführlich besprochen, daher hier nur der Verweis auf den Entwurf der Tabelle **tblObjects** in Bild 5.

Rechnungsverwaltung: Bestellformular

Nachdem wir das Datenmodell für unsere Rechnungsverwaltung angelegt sowie die Tabellen mit Beispieldaten gefüllt haben, kommt als Nächstes die Benutzeroberfläche zum Verwalten der Kunden-, Produkt- und Bestelldaten an die Reihe. Die dazu notwendigen Formulare stellen wir in mehreren Teilen dieser Beitragsreihe vor. Die Basis ist das Formular zum Anzeigen der Bestellungen, mit dem wir den Kunden auswählen, die Bestelldaten eingeben und die Bestellpositionen hinzufügen können. Die Programmierung dieses Formulars zeigen wir im vorliegenden Beitrag – inklusive Validierung und mehr.

Leichter programmieren mit Testdaten

Das Schöne ist, dass wir im Beitrag **Rechnungsverwaltung: Beispieldaten** (**www.access-im-unternehmen.de/1381**) bereits einige Beispieldatensätze angelegt haben, sodass wir beim Programmieren der Formulare nicht immer noch mühselig von Hand Testdaten eingeben müssen. Dabei sollten wir aber nicht vergessen, dass der Kunde die Anwendung gegebenenfalls ohne Daten erhält. In diesem Fall müssen wir die Formulare auch noch mit leeren Tabellen testen, um zu prüfen, ob das initiale Anlegen von Daten ebenfalls funktioniert und ob die Formulare komplett ohne Daten genauso gut funktionieren.

Reihenfolge beim Anlegen der Formulare

Wenn Sie keine Testdaten zur Verfügung hätten, würden Sie die Formulare logischerweise in einer Reihenfolge erstellen, in der auch die Daten eingegeben werden. Wir würden also zuerst ein Formular zur Eingabe von Anreden, Einheiten und Mehrwertsteuersätzen benötigen, dann für Kunden und Produkte und schließlich für Bestellungen und Bestelldetails (wobei letztere in einem Formular plus Unterformular untergebracht werden).

Mit Testdaten können wir das Erstellen der Formulare allerdings in beliebiger Reihenfolge gestalten. Also beginnen wir doch gleich mal mit dem aufwendigsten Formular – dem zur Eingabe der Bestellungen.

Formulare zur Eingabe von Bestellungen

Zur Eingabe von Bestellungen benötigen wir eigentlich nur ein einfaches Formular, aber zu Bestellungen gehören ja auch noch Bestellpositionen. Und da diese über eine 1:n-

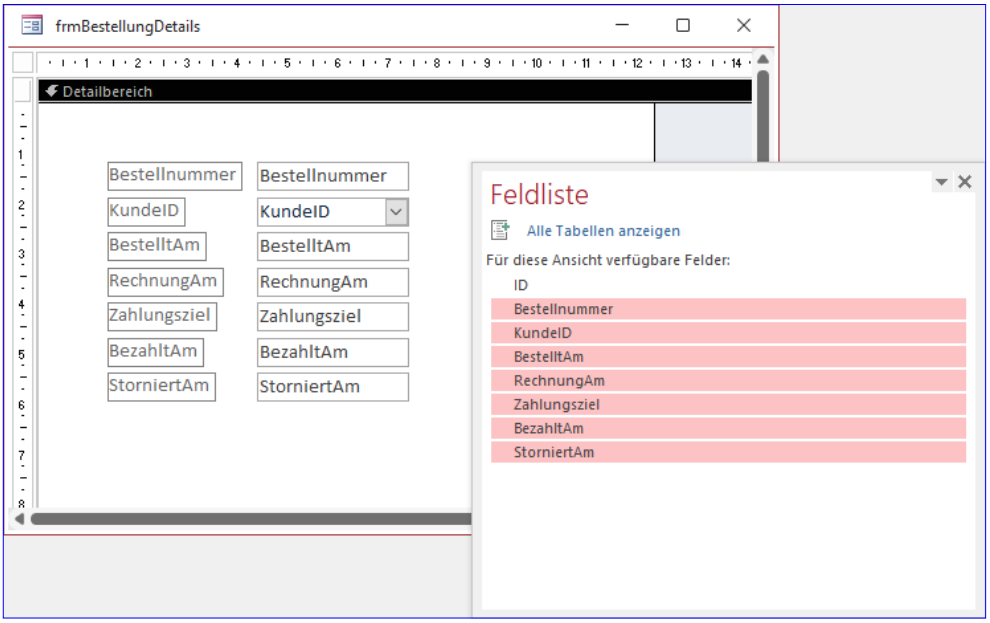


Bild 1: Das Formular **frmBestellungDetails** in der Entwurfsansicht

Beziehung mit den Bestellungen verknüpft sind, bietet sich die Verwendung eines Unterformulars an.

Wir erstellen zuerst das Hauptformular und öffnen es in der Entwurfsansicht. Dieses nennen wir **frmBestellungDetails** und weisen ihm für die Eigenschaft **Datensatzquelle** die Tabelle **tblBestellungen** zu. Danach wechseln wir zur Feldliste und ziehen alle Felder außer **ID** in den Detailbereich des Formularentwurfs (siehe Bild 1). Warum nicht das Feld **ID**? Weil dieses ein rein für die Herstellung von Beziehungen verwendetes Feld ist und der Benutzer dieses ohnehin nicht ändern kann und soll. Die Beschriftungen müssen wir noch ein wenig anpassen, sodass beispielsweise aus **KundeID** die Beschriftung **Kunde** wird oder aus **BestelltAm** die Beschriftung **Bestellt am**. Außerdem fehlen überall noch Doppelpunkte.

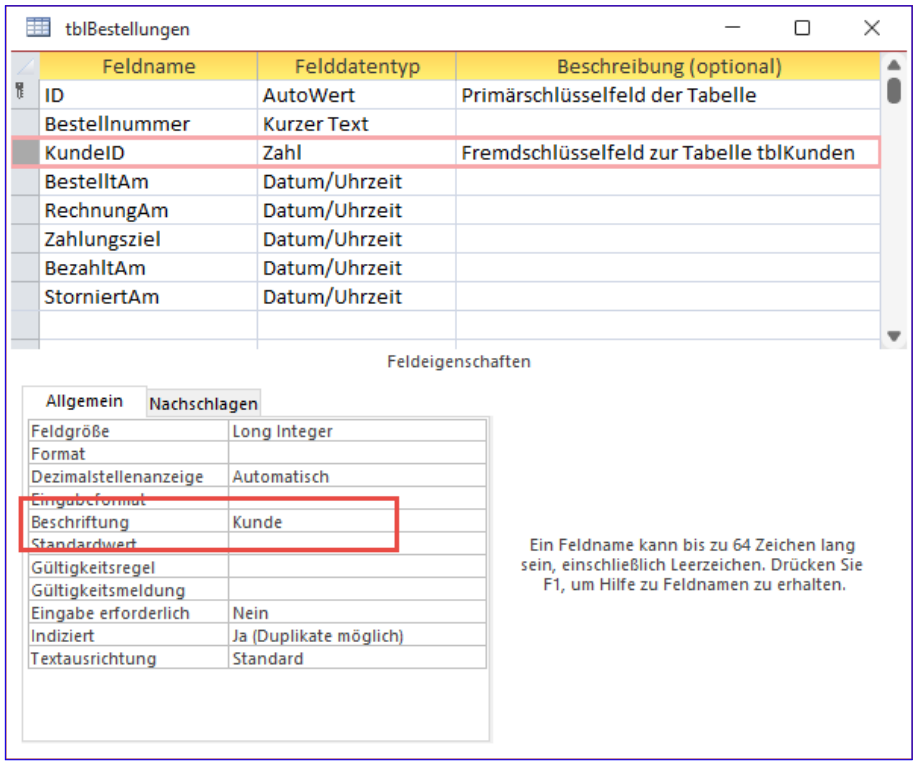


Bild 2: Voreinstellung für die Beschriftung von Feldern, auch in Formularen oder Berichten

Diese Änderungen hätten wir auch schon zu einem früheren Zeitpunkt vorbereiten können, nämlich im Tabellenentwurf. Dort hätten wir diese Bezeichnungen für die Eigenschaft **Beschriftung** der jeweiligen Felder eintragen können. Da wir nicht wissen, ob wir noch weitere Formulare oder Berichte auf Basis dieser Felder erstellen, nehmen wir diese Änderungen noch schnell vor. Dazu öffnen wir die Tabelle **tblBestellungen** nochmals in der Entwurfsansicht und stellen dort für die verschiedenen Felder die gewünschten Beschriftungen in der gleichnamigen Eigenschaft ein (siehe Bild 2).

Doppelpunkte zu Beschriftungen hinzufügen

Und auch den Doppelpunkt hinter der Beschriftung müssen wir nicht von Hand anlegen. Wir können dies für das aktuelle Formular auch so einstellen, dass jedes Feld automatisch einen Doppelpunkt erhält. Dazu müssen Sie jedoch bereits vor dem Hinzufügen der Felder der Datensatzquelle eine bestimmte Eigenschaft einstellen.

Klicken Sie dazu in der Entwurfsansicht des Formulars im Ribbon auf **Formularentwurf|Steuerelemente|Textfeld**, aber fügen Sie kein Textfeld zum Formular hinzu. Das Eigenschaftsfeld zeigt nun einige Eigenschaften an, die nach dem Hinzufügen von Textfeldern nicht mehr erscheinen – zum Beispiel **Mit Doppelpunkt**. Diese Eigenschaft stellen Sie, falls dies noch nicht der Fall ist, auf **Ja** ein (siehe Bild 3).

Anschließend betätigen Sie die **Esc**-Taste, um die Auswahl des Textfeldes abzubrechen. Das Feld **KundeID** haben wir in der Tabelle als Nachschlagefeld ausgelegt. Das heißt, dieses Feld wird beim Ziehen aus der Feldliste in den Formularentwurf als Kombinationsfeld erstellt. Da wir die Änderung zum Anzeigen des Doppelpunkts soeben nur für Textfelder eingestellt haben, müssen wir dies auch noch

für Kombinationsfelder erledigen.

Anschließend ziehen wir erneut die gewünschten Felder aus der Feldliste in den Detailbereich des Formularentwurfs. Das Ergebnis sieht schon viel besser aus – die Beschriftungen aus dem Tabellenentwurf wurden übernommen und auch die Doppelpunkte wurden hinzugefügt (siehe Bild 4).

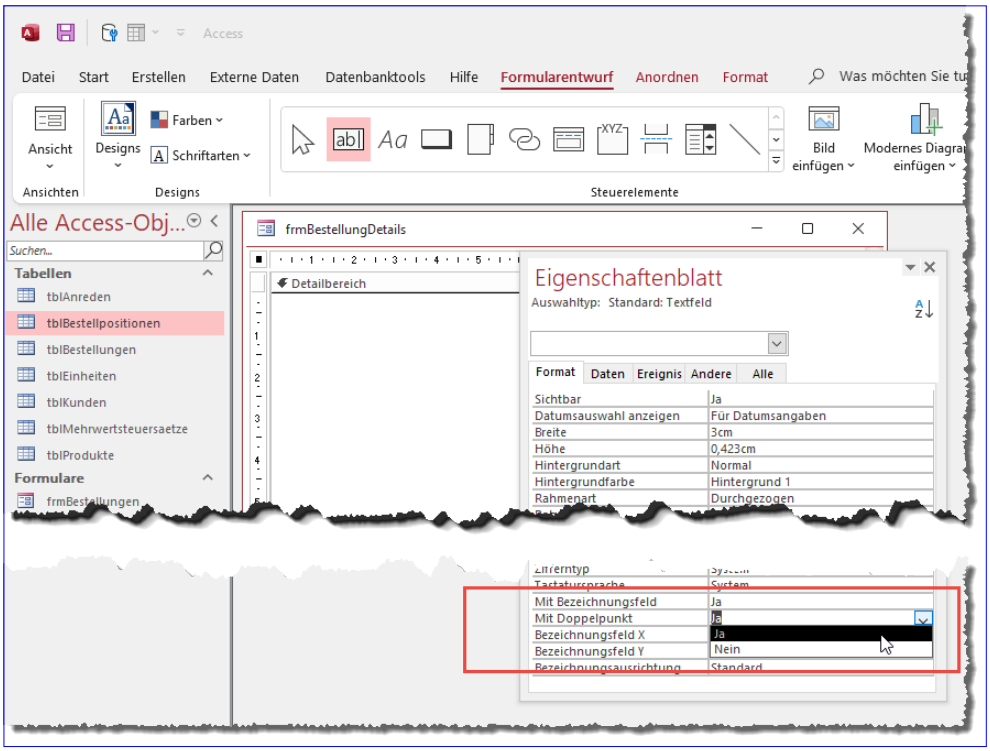


Bild 3: Aktivieren des Doppelpunkts für Beschriftungsfelder

Damit sind die Arbeiten am Hauptformular vorerst erledigt. Sie können nun bereits in die Datenblattansicht wechseln und sehen dort die Testdaten zum Durchblättern. Hier erkennen Sie auch, dass wir die Breite des Kombinationsfeldes **KundeID** noch vergrößert haben (siehe Bild 5).

Unterformular für Bestellpositionen

Damit kommen wir zum Unterformular, das die Bestellpositionen zur jeweiligen Bestellung anzeigen soll. Dieses legen wir unter dem Namen **sfmBestellungDetails** an. Als Datensatzquelle fügen wir die Tabelle **tblBestellpositionen** zu. Diese passen wir zuvor ähnlich wie die Tabelle **tblBestellungen** noch an, indem wir passende Beschriftungen für die Felder **ProduktID** (**Produkt**), **Mehrwertsteuersatz** (**MwSt.-Satz**) und **EinheitID** (**Einheit**) einstellen.

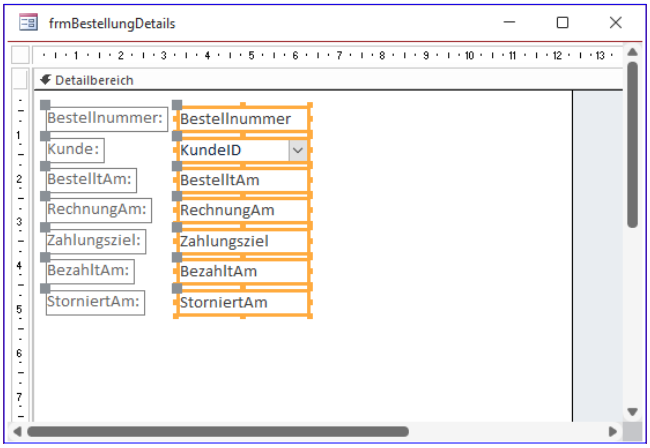


Bild 4: Bezeichnungsfelder mit Doppelpunkten

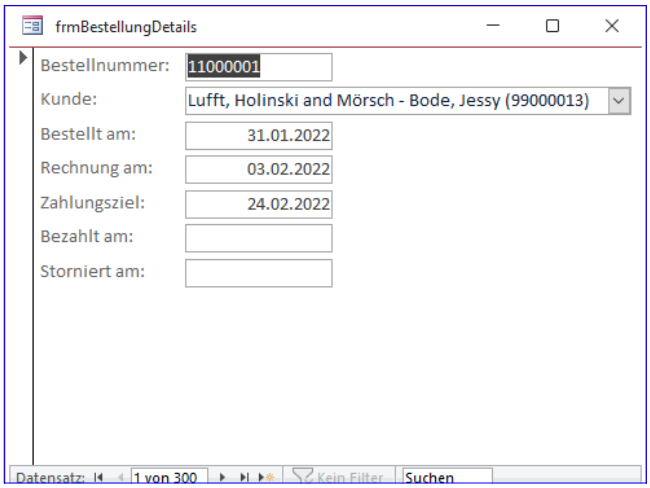


Bild 5: Steuerelemente mit angepasster Breite

Danach ziehen wir alle Felder außer **ID** und **BestellungID** in den Detailbereich des Formulars. ID benötigen wir nicht, weil es das Primärschlüsselfeld der Tabelle ist, und **BestellungID** dient nur dem Herstellen einer Beziehung zum jeweils im Hauptformular angezeigten Datensatz.

Außerdem stellen wir die Eigenschaft **Standardansicht** des Formulars auf **Datenblatt** ein. Die Bestellpositionen sollen im Unterformular tabellarisch dargestellt werden. Danach schließen wir das als Unterformular zu verwendende Formular.

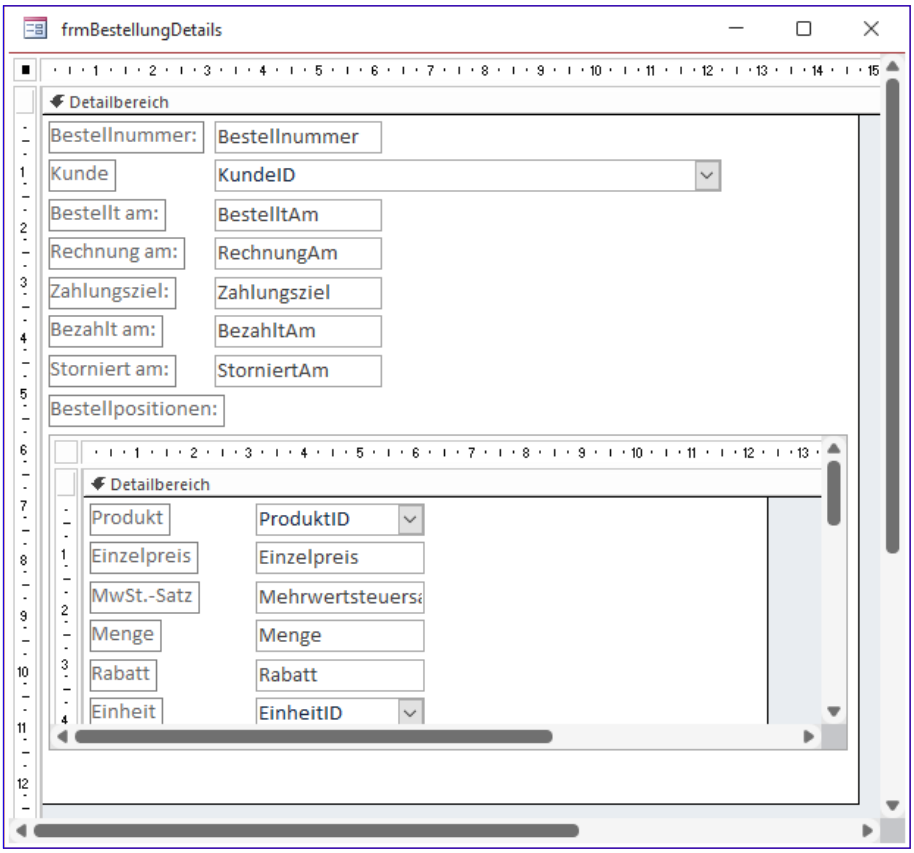


Bild 6: Haupt- und Unterformular

Unterformular zum Hauptformular hinzufügen

Nun öffnen wir wieder das Formular **frmBestellungDetails** in der Entwurfsansicht und fügen das Unterformular **sfmBestellungDetails** zum Hauptformular hinzu, indem wir es aus dem Navigationsbereich in den Detailbereich des Hauptformulars ziehen – und es unter den dort bereits befindlichen Steuerelementen fallenlassen.

Dort platzieren wir es wie in Bild 6 und ändern seine Beschriftung auf **Bestellpositionen**. Für die bessere Bedienbarkeit nehmen wir noch weitere Änderungen vor. So stellen wir die Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** des Unterformular-Steuerelements jeweils auf **Beide** ein.

Dies ändert automatisch die entsprechenden Eigenschaften des Bezeichnungsfeldes der Unterformular-Steuerelemente, die wir wieder auf **Oben** beziehungsweise **Links** zurückstellen.

Wie soll das Unterformular nun wissen, dass es nicht alle Bestellpositionen anzeigen soll, sondern nur die Bestellpositionen, die zum aktuell im Hauptformular angezeigten Datensatz gehören? Die notwendige Einstellung haben wir indirekt bereits getroffen, indem wir eine Beziehung zwi-

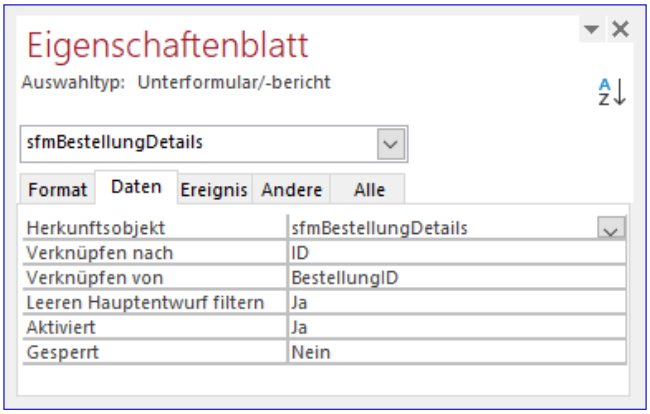


Bild 7: Herstellen der Beziehung der Daten aus Haupt- und Unterformular

schen den Feldern **BestellungID** der Tabelle **tblBestellpositionen** und **ID** der Tabelle **tblBestellungen** definiert haben. Access erkennt diese Beziehung beim Hinzufügen eines Unterformulars und trägt die relevanten Daten direkt in die beiden Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformular-Steuerelements ein. Das Ergebnis sieht wie in Bild 7 aus.

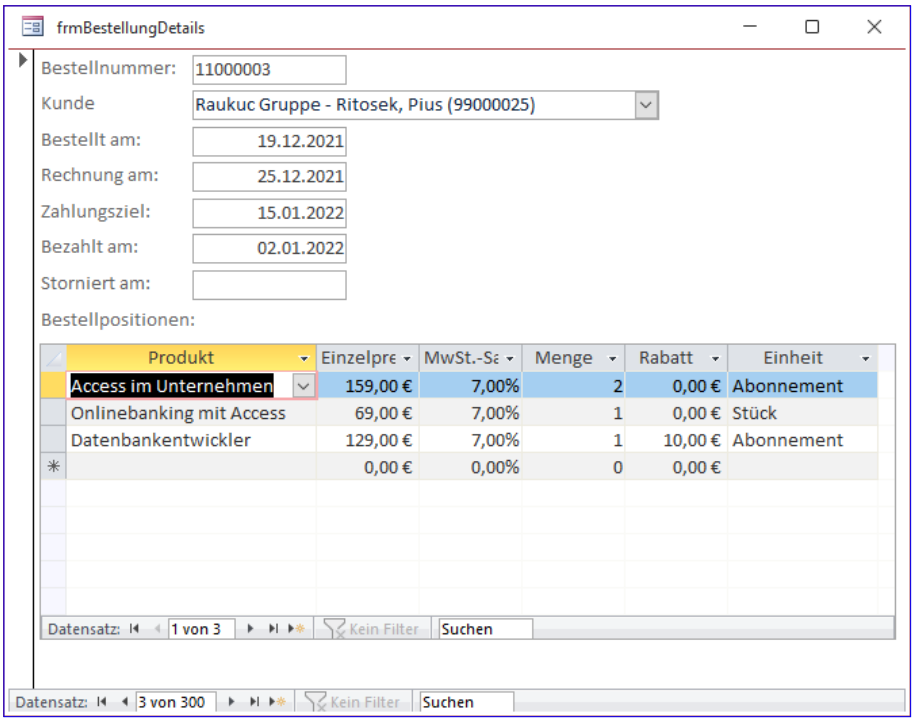


Bild 8: Haupt- und Unterformular in der Formularansicht

Ausprobieren des Bestellungen-Formulars

Nun wechseln wir in die Formularansicht des Formulars **frmBestellungDetails** und finden das Ergebnis aus Bild 8 vor.

Wenn wir durch die Datensätze des Hauptformulars navigieren, zeigt das Unterformular auch jeweils die passenden Datensätze an.

Damit können wir nun auch einen der weiter oben erwähnten Tests durchführen, der das Anlegen einer neuen Bestellung betrifft. Dazu wechseln wir im Hauptformular zu einem neuen, leeren Datensatz. Dieser zeigt sowohl im Hauptformular als auch im Unterformular noch keine Daten an. Wenn wir nun den herkömmlichen Weg gehen und zuerst die Daten der Bestellung im Hauptformular anlegen und dann über das Unterformular Bestellpositionen hinzufügen, läuft alles wie erwartet.

Aber es kann ja auch sein, dass der Kunde anruft und direkt sagt, er möchte Produkt X und Produkt Y erhalten und der neue Mitarbeiter trägt erst einmal die entsprechenden Bestellpositionen ein, ohne die übrigen Bestelldaten hinzuzufügen.

Dann geschieht Folgendes: Da im Hauptformular noch kein Datensatz angelegt wurde, trägt Access in die neuen Datensätze im Unterformular mit den Bestellpositionen den Wert in das Fremdschlüsselfeld **BestellungID** ein, der auch gerade im damit verknüpften Feld **ID** im Hauptformular enthalten ist – und dieser lautet **Null** (siehe Bild 9).

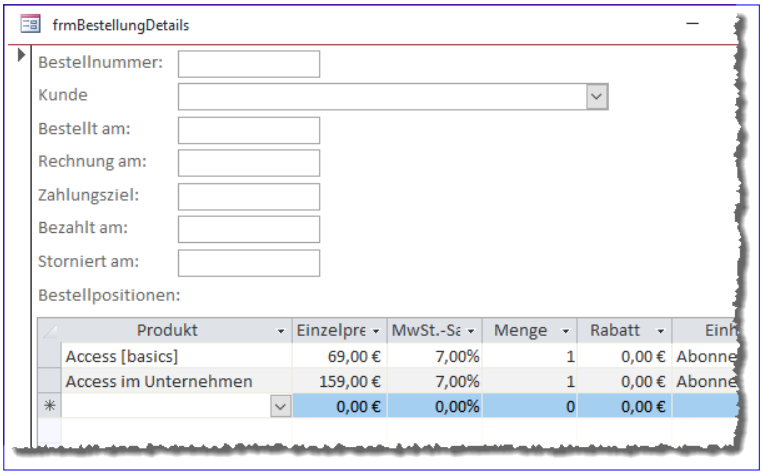


Bild 9: Anlegen von Daten im Unterformular ohne Datensatz im Hauptformular

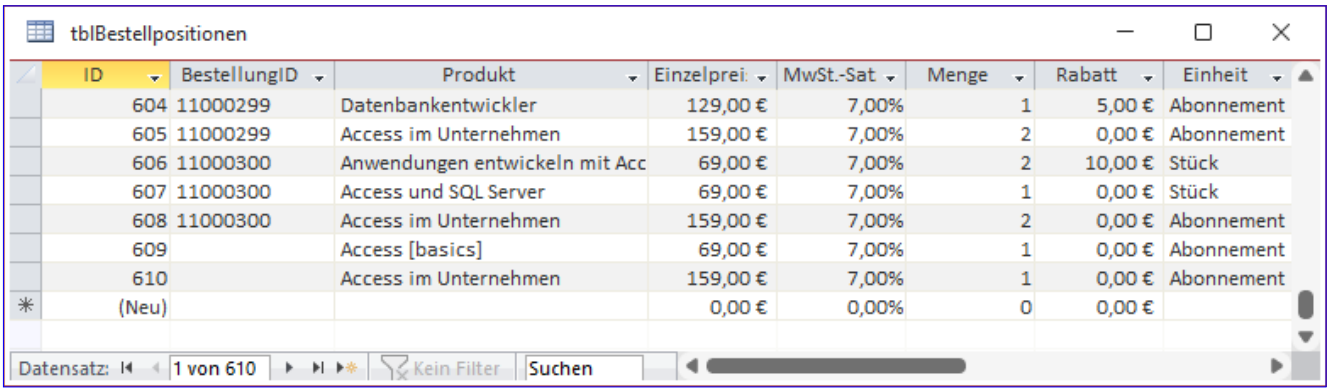


Bild 10: Bestellpositionen ohne **BestellungID**

Wir haben also ein paar Bestellpositionen ohne Zuweisung zu einer Bestellung (siehe Bild 10).

Noch schlimmer wird es, wenn der Mitarbeiter nun nach der Eingabe der Bestellpositionen die übrigen Daten der Bestellung nachträgt. Das sieht zu Beginn noch so aus, als würde es funktionieren (siehe Bild 11).

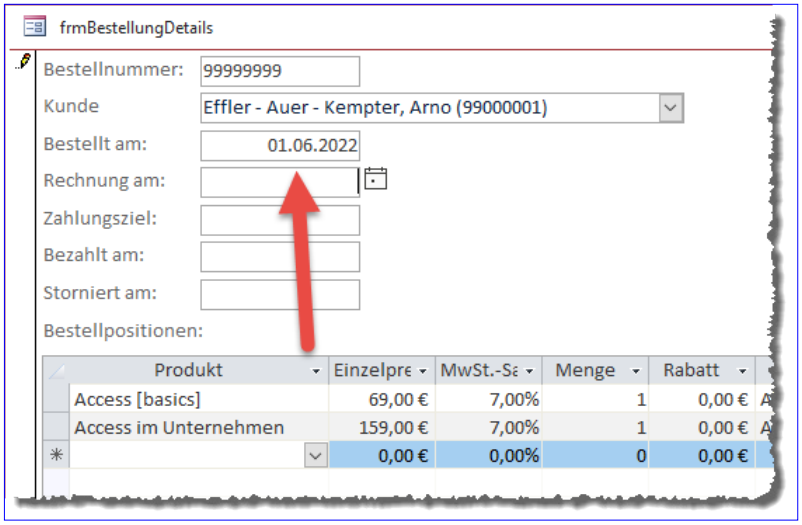


Bild 11: Eingabe der Bestelldaten nach den Bestellpositionen ...

Wenn der Benutzer dann allerdings zu einem anderen Datensatz wechselt und dann zum vorherigen Datensatz zurückkehrt, ist das Unterformular mit den Bestellpositionen plötzlich leer (siehe Bild 12).

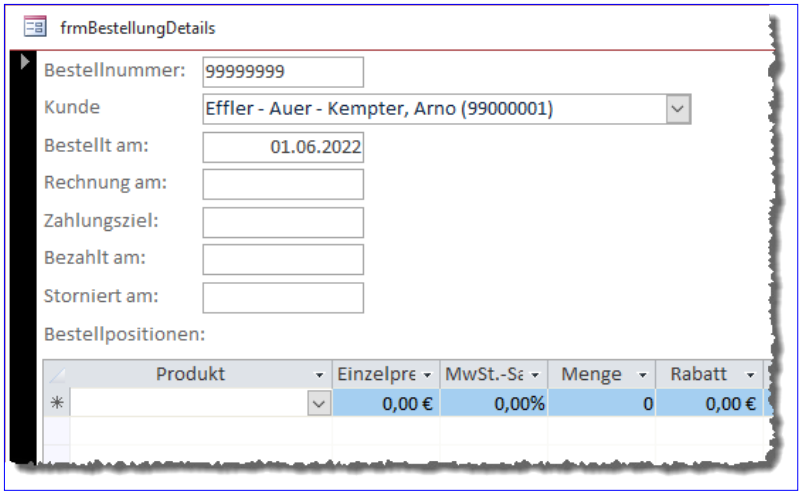


Bild 12: ... führt nach dem Datensatzwechsel zum Verschwinden der vermeintlich verknüpften Daten im Unterformular

Was tun? Wir müssen irgendwie dafür sorgen, dass der Benutzer nur Daten in das Unterformular eingeben kann, wenn der Datensatz im Hauptformular bereits angelegt wurde.

Daten erst ins Hauptformular eingeben

Um sicherzustellen, dass zuerst Daten ins Hauptformular und dann erst ins Unterformular eingegeben werden, wollen wir den Benutzer auf irgendeine Weise davon abhalten, die Daten umgekehrt einzugeben.

Die erste Idee wäre, das Unterformular einfach zu sperren, wenn der Benutzer gerade einen neuen Datensatz anlegt und versucht, in das Unterformular zu springen, bevor es einen Primärschüsselwert im Hauptformular gibt.

Das können wir mit einer Ereignisprozedur erledigen, die durch das Ereignis **Beim Anzeigen** des Hauptformulars ausgelöst wird. Diese prüft mit der Eigenschaft **NewRecord**, ob das Formular gerade einen neuen, leeren Datensatz enthält und stellt in diesem Fall die Eigenschaft **Enabled** des Unterformular-Steuerelements auf den Wert **False** ein, anderenfalls auf den Wert **True**:

```
Private Sub Form_Current()
    If Me.NewRecord Then
        Me!sfmBestellungDetails.Enabled = False
    Else
        Me!sfmBestellungDetails.Enabled = True
    End If
End Sub
```

Dies können wir auch wesentlich kürzer schreiben:

```
Private Sub Form_Current()
    Me!sfmBestellungDetails.Enabled = Not Me.NewRecord
End Sub
```

Was aber, wenn der Benutzer dann zum Beispiel den Kunden auswählt, was ja faktisch dazu führt, dass das Autowertfeld **ID** der Tabelle **tblBestellungen** einen Wert erhält und wir nun Bestellpositionen zum Unterformular hinzufügen könnten? Dann wäre das Unterformular immer noch gesperrt, weil die obigen Prozedur nur feuert, wenn ein Datensatz angezeigt wird. Aber auch das Ändern eines neuen Datensatzes löst ein Ereignis aus, nämlich **Bei Geändert**. Für dieses Ereignis hinterlegen wir die folgende Prozedur, die das Unterformular aktiviert:

```
Private Sub Form_Dirty(Cancel As Integer)
    Me!sfmBestellungDetails.Enabled = True
End Sub
```

Somit kann der Benutzer nun auch Daten in das Unterformular eingeben.

Benutzer sind allerdings einfallsreich und die nächste Schwachstelle, die sie finden könnten, ist Folgende: Wenn man nämlich den Datensatz im Hauptformular »dirty« macht und dann die **Esc**-Taste betätigt, werden die Änderungen wieder zurückgekommen und wir haben wieder einen unbefleckten Bestelldatensatz ohne **ID**. Das Anlegen von Bestellpositionen im Unterformular ist allerdings aktuell möglich und würde zu den oben beschriebenen Problemen führen.

Um dies zu verhindern, greifen wir das nächste spannende Ereignis ab, nämlich **Bei Rückgängig**. Dieses wird ausgelöst, wenn der Benutzer beispielsweise mit der **Esc**-Taste die bereits vorgenommen Änderungen verwirft. Hier deaktivieren wir einfach wieder das Unterformular, sofern sich im Hauptformular wieder ein neuer, leerer Datensatz befindet:

```
Private Sub Form_Undo(Cancel As Integer)
    Me!sfmBestellungDetails.Enabled = Not Me.NewRecord
End Sub
```

Damit kann der Benutzer nun zumindest keine nicht verknüpften Bestellpositionen mehr anlegen.

Bezeichnungen der Steuerelemente anpassen

Bevor wir gleich mit weiteren VBA-Prozeduren fortfahren, in denen wir unter anderem auf die Inhalte von Steuerelementen zugreifen wollen, legen wir für diese zunächst sinnvolle Namen mit Präfix fest. Das heißt, Textfelder erhalten das Präfix **txt**, Kombinationsfelder das Präfix **cbo** und so weiter.

Dies sollten Sie zumindest für solche Steuerelemente durchführen, auf die Sie per VBA zugreifen. Auf diese Weise unterscheiden Sie die Steuerelemente von den gleichnamigen, an das Formular gebundenen Felder der Datensatzquelle.

Rechnungsverwaltung: Kundendetails

Eine Rechnungsverwaltung, mit der Rechnungen an verschiedene Kunden geschickt werden sollen, benötigt eine Tabelle zum Speichern dieser Kunden. Logisch, dass wir dieser Tabelle auch ein Formular zum komfortablen Bearbeiten der Kunden an die Seite stellen. Dieses enthält allerdings nicht nur die reinen Kundendaten, sondern wir wollen damit auch noch die Bestellungen des jeweiligen Kunden in einem Unterformular anzeigen – und darüber die Anzeige der Bestelldetails ermöglichen.

Unterformular sfmKundeDetails für die Bestellungen

Wir beginnen direkt mit dem Entwurf des Unterformulars zur Anzeige der Bestellungen des Kunden. Dieses wollen wir **sfmKundeDetails** nennen. Diesem fügen wir über die Eigenschaft **Datensatzquelle** gleich die Tabelle **tblBestellungen** hinzu.

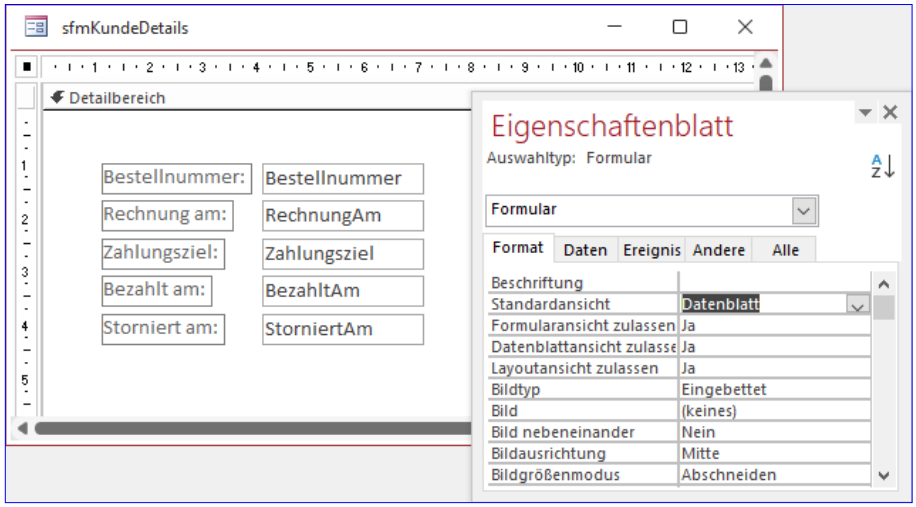


Bild 1: Entwurf des Unterformulars **sfmKundeDetails**

Im Gegensatz zum Unterformular aus dem Beitrag **Rechnungsverwaltung: Bestellübersicht** (**www.access-im-unternehmen.de/1384**), wo wir alle Bestellungen inklusive der Angabe des jeweiligen Kunden in einem Unterformular darstellen, benötigen wir hier nicht mehr die Anzeige des Kunden – dieser wird ja schon im Hauptformular angezeigt, das wir gleich noch erstellen werden. Also fügen wir nun die Felder **Bestellnummer**, **RechnungAm**, **Zahlungsziel**, **BezahltAm** und **StorniertAm** zum Detailbereich des Formularentwurfs hinzu.

Dieser sieht anschließend wie in Bild 1 aus. Damit die Daten in der Datenblattansicht angezeigt werden, legen wir die Eigenschaft **Standardansicht** dort auf den Wert **Datenblatt** fest. Außerdem wollen wir, dass der Kunde die Daten in diesem Unterformular nicht direkt bearbeiten kann. Daher legen wir die Eigenschaften **Anfügen zulassen**, **Löschen zulassen** und **Bearbeitungen zulassen** jeweils auf den Wert **Nein** ein. Damit können wir die Arbeiten an diesem Formular vorerst beenden und dieses schließen.

Hauptformular frmKundeDetails anlegen

Danach legen wir ein weiteres Formular namens **frmKundeDetails** an. Bevor wir diesem das Unterformular **sfmKundeDetails** hinzufügen, müssen wir die Datensatzquelle für das Hauptformular festlegen. So kann Access direkt erkennen, dass es zwischen den Datensatzquellen von Haupt- und Unterformular eine Beziehung gibt und dies entsprechend in den Eigenschaften **Verknüpfen von** und **Verknüpfen nach** des Unterformular-Steuerelements vermerken.

Wenn wir schon die Datensatzquelle definiert haben, können wir auch direkt die gewünschten Felder aus der

Feldliste in den Detailbereich des Formulars ziehen. Dabei berücksichtigen wir alle Felder mit Ausnahme des Feldes **ID**, das nur zu Verknüpfungszwecken gepflegt wird und für den Benutzer unsichtbar bleiben soll (siehe Bild 2).

Falls Sie sich wundern, dass in unserem Formular beispielsweise für das Feld **AnredeID** ein Beschriftungsfeld mit dem Text **Anrede** angelegt wurde: Wir haben direkt im Tabellenentwurf die für die Bezeichnungsfelder gewünschten Texte für die Eigenschaft **Beschriftung** der jeweiligen Felder hinterlegt. Mehr dazu erfahren Sie im Beitrag **Bezeichnungsfelder im Griff** (**www.access-im-unternehmen.de/1380**).

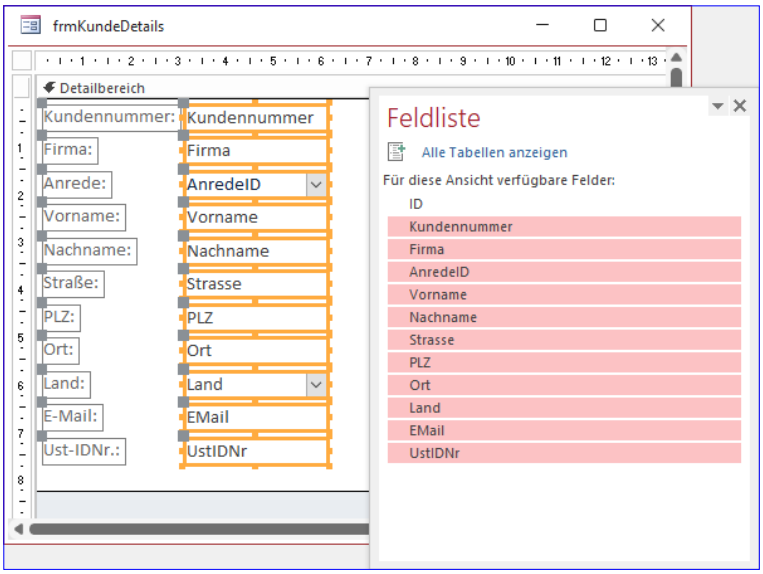


Bild 2: Entwurf des Hauptformulars **frmKundeDetails**

Unterformular zum Hauptformular hinzufügen

Danach teilen wir die Felder auf zwei Spalten auf, sodass wir unten das Unterformular **sfmKundeDetails** platzieren können. Dieses ziehen wir aus dem Navigationsbereich in den Formularentwurf und erhalten nach wenigen Anpassungen das Ergebnis aus Bild 3. Zu diesen Anpassungen gehört neben der Ausrichtung und der Einstellung der Größe das Festlegen der Eigenschaften **Horizontaler Anker** und **Vertikaler Anker** jeweils auf den Wert **Beide**. Damit wird das Unterformular mit dem Hauptformular vergrößert.

Gegebenenfalls können Sie sich nun noch davon überzeugen, dass Access automatisch das Primärschlüsselfeld **ID** der Tabelle **tblKunden** für die Eigenschaft **Verknüpfen nach** und das Fremdschlüsselfeld **KundeID** der Tabelle **tblBestellungen** für die Eigenschaft **Verknüpfen von** eingetragen hat. Dies stellt sicher, dass das Unterformular immer nur die Datensätze der Tabelle **tblBestellungen** anzeigt, die mit dem Datensatz der Tabelle **tblKunden** aus dem Hauptformlar verknüpft sind.

Da wir bereits einige Beispieldatensätze angelegt haben, wie im Beitrag **Rechnungsverwaltung: Beispieldaten** (**www.access-im-unternehmen.de/1381**) beschrieben,

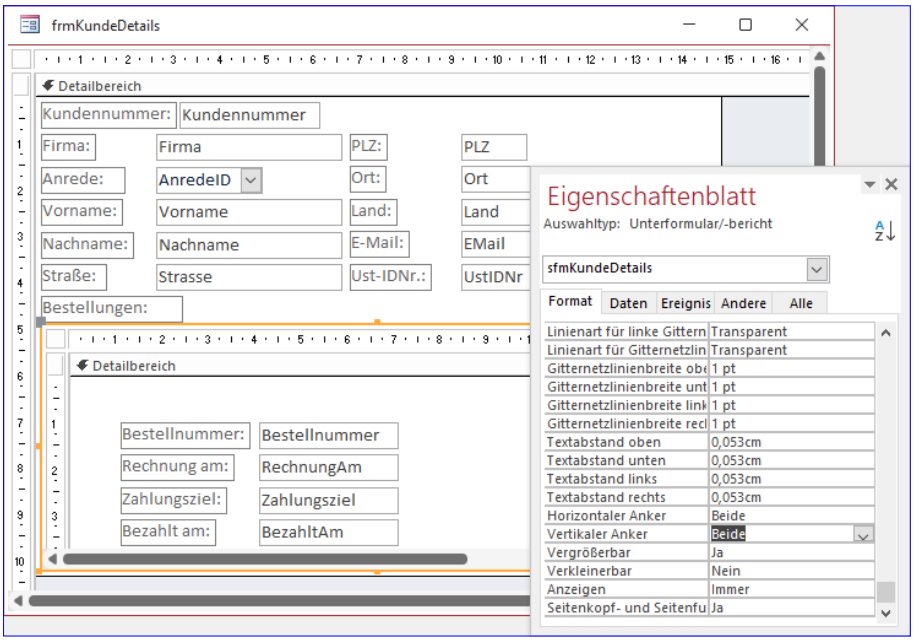


Bild 3: Das Hauptformular **frmKundeDetails** mit dem Unterformular

finden wir beim Wechsel in die Formularansicht bereits einige Beispieldaten vor (siehe Bild 4).

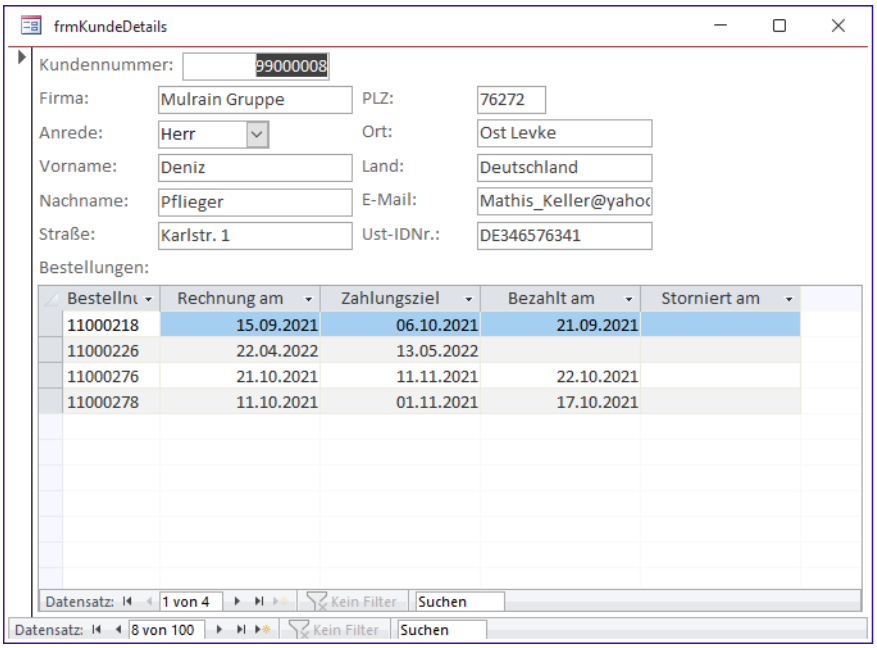


Bild 4: Ein Kunde und seine Bestellungen

Validierung im Hauptformular

Damit können wir uns nun um die Validierung der Eingabefelder im Hauptformular kümmern.

Hier sind Einschränkungen bei folgenden Feldern nötig:

- **Anrede**: Pflichtfeld
- **Vorname**, **Nachname**, **Straße**, **Ort**, **Land**: Pflichtfelder
- **PLZ**: Ausgehend von der vereinfachenden Annahme, wir hätten es mit Adressen aus dem Bereich Deutschland, Österreich und der Schweiz zu tun, muss diese fünf Stellen (für Deutschland) oder vier Stellen (für Österreich und Schweiz) aufweisen.
- **E-Mail**: Diese soll grob validiert werden, also auf ein enthaltenes @-Zeichen und einen Punkt.
- **Ust-IDNr.**: Soll für Deutschland und Österreich geprüft werden auf **DE** plus neun Ziffern beziehungsweise auf **ATU** plus acht Ziffern, für Schweiz muss das Feld leer sein.

Da Pflichtfelder nur beim Speichern des kompletten Datensatzes geprüft werden können und dies für die abhängigen Felder ohnehin der Fall ist, können wir uns hier auf das Ereignis **Vor Aktualisierung** des Formulars konzentrieren. Für dieses hinterlegen wir die Prozedur aus Listing 1. Das Ereignis wird nur beim Versuch ausgelöst, den Datensatz nach vorherigen Änderungen zu speichern – also etwa beim Wechsel zu einem anderen Datensatz oder beim Speichern mit der Tastenkombination **Strg + S**.

Vorab die Information, dass wir alle gebundenen Steuerelemente mit Präfixen versehen haben, in diesem Fall die Textfelder mit **txt** und die Kombinationsfelder mit **cbo**. Auf diese Weise kann man sauber zwischen den Feldnamen der Datensatzquelle und den daran gebundenen Steuerelementen unterscheiden.

Diese Prozedur enthält einige **If...Then**-Bedingungen, die jeweils eine Überprüfung für ein Feld vornehmen. Die erste untersucht beispielsweise, ob das Feld **txtFirma** leer ist. Ist das der Fall, erscheint eine entsprechende Meldung, der Fokus wird auf das Textfeld eingestellt und der Rückgabeparameter **Cancel** auf den Wert **True**.

Außerdem verlassen wir an dieser Stelle mit **Exit Sub** die Prozedur. Das Einstellen des Parameters **Cancel** auf **True** sorgt dafür, dass der **Speichern**-Vorgang, der das Ereignis **Vor Aktualisierung** ausgelöst hat, abgebrochen wird.

Validieren der E-Mail-Adresse

Die E-Mail-Adresse können wir direkt nach der Eingabe validieren und den Benutzer darauf hinweisen, falls die E-Mail-Adresse nicht gültig ist. Deshalb reicht es auch aus,

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
    If Len(Nz(Me!txtKundennummer, "")) = 0 Then
        MsgBox "Bitte geben Sie eine Kundennummer ein.", vbOKOnly + vbExclamation, "Kundennummer fehlt"
        Me!txtKundennummer.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Nz(Me!cboAnredeID, 0) = 0 Then
        MsgBox "Bitte wählen Sie eine Anrede aus.", vbOKOnly + vbExclamation, "Anredefehlt"
        Me!cboAnredeID.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtVorname, "")) = 0 Then
        MsgBox "Bitte geben Sie einen Vornamen ein.", vbOKOnly + vbExclamation, "Vorname fehlt"
        Me!txtVorname.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtStrasse, "")) = 0 Then
        MsgBox "Bitte geben Sie eine Straße ein.", vbOKOnly + vbExclamation, "Straße fehlt"
        Me!txtStrasse.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtPLZ, "")) = 0 Then
        MsgBox "Bitte geben Sie eine PLZ ein.", vbOKOnly + vbExclamation, "PLZ fehlt"
        Me!txtPLZ.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtOrt, "")) = 0 Then
        MsgBox "Bitte geben Sie einen Ort ein.", vbOKOnly + vbExclamation, "Ort fehlt"
        Me!txtOrt.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!cboLand, 0)) = 0 Then
        MsgBox "WählenSie ein Land aus.", vbOKOnly + vbExclamation, "Land fehlt"
        Me!cboLand.SetFocus
        Cancel = True
        Exit Sub
    End If
    If Len(Nz(Me!txtEMail, "")) = 0 Then
        MsgBox "Bitte geben Sie eine E-Mail-Adresse ein.", vbOKOnly + vbExclamation, "E-Mail-Adresse fehlt"
        Me!txtEMail.SetFocus
        Cancel = True
        Exit Sub
    End If
End Sub
```

Listing 1: Validieren beim Speichern des Datensatzes

dass wir in der Prozedur **Form_BeforeUpdate** nur auf eine leere Zeichenkette prüfen. Sobald der Benutzer einmal ein Zeichen für die E-Mail eingegeben hat, kann er dieses nur nach Eingabe einer gültigen E-Mail-Adresse verlassen.

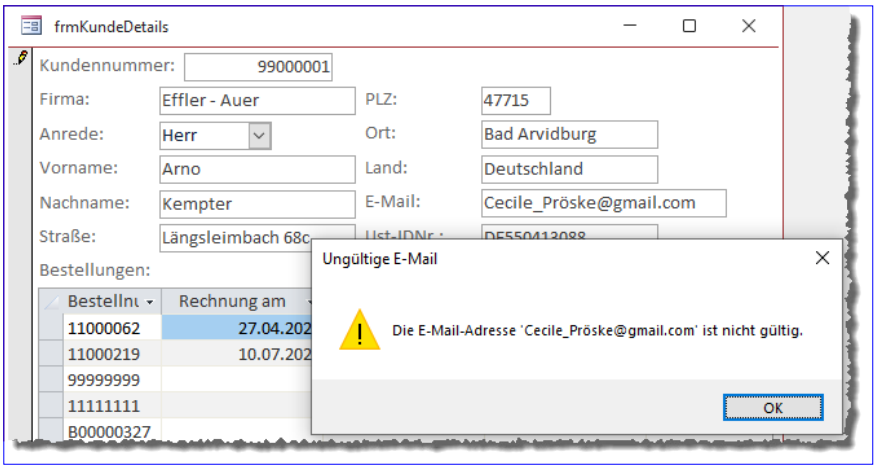


Bild 5: Die E-Mail-Adresse konnte nicht validiert werden.

Wie Sie die Gültigkeit einer E-Mail-Adresse überprüfen, haben wir im Beitrag **E-Mail-Adressen validieren per VBA** (**www.access-im-unternehmen.de/1376**) beschrieben. Die dortige Funktion **CheckEMailSyntax** verwenden wir auch in diesem Formular, um die E-Mails nach der Eingabe zu prüfen. Dazu hinterlegen wir für das Ereignis **Vor Aktualisierung** des Textfelds **txtEMail** die Prozedur aus Listing 2.

Die Prozedur liest den Inhalt des Textfeldes **txtEMail** in die Variable **strEMail** ein. Falls das Feld den Inhalt **Null** hat, landet eine leere Zeichenkette in **strEMail**. Danach ruft die Prozedur die Funktion **CheckEMailSyntax** auf und übergibt dieser die E-Mail-Adresse zur Prüfung. Ist das Ergebnis nicht **True**, zeigt die Prozedur eine Meldung an. Außerdem sorgt das Einstellen des Parameters **Cancel** auf den Wert **True** dafür, dass das Feld nicht verlassen werden kann. Das Ergebnis sieht beispielsweise wie in Bild 5 aus.

Postleitzahl validieren

Das Feld **PLZ** ist ohnehin schon im Datenmodell auf fünf Zeichen begrenzt. Das reicht aus, wenn der Benutzer – wie vorgegeben – nur die Postleitzahl eingibt und nicht, wie hier und da noch zu sehen, die Postleitzahl mit führendem Länderkennzeichen, also beispielsweise **D-47137**. Es sollen also nur Zahlen eingegeben werden. Das Feld **PLZ** ist allerdings mit dem Felddatentyp **Kurzer Text** versehen.

Warum das, wenn doch nur Zahlen eingegeben werden können sollen? Weil es auch Postleitzahlen mit führender Null gibt (**01234**) und diese gern wegfällt, wenn man das Text als Zahlenfeld definiert. Um dennoch dafür zu sorgen, dass nur Zahlen eingegeben werden können, haben wir zwei Möglichkeiten.

Die erste ist, nach der Eingabe zu prüfen, ob der Benutzer nur Zahlen eingegeben hat. Das erledigen wir mit der Ereignisprozedur **Vor Aktualisierung** für das Feld **txtPLZ**:

```
Private Sub txtEMail_BeforeUpdate(Cancel As Integer)
    Dim strEMail As String
    strEMail = Nz(Me!txtEMail, "")
    If Not CheckEMailSyntax(strEMail) Then
        MsgBox "Die E-Mail-Adresse '" & strEMail & "' ist nicht gültig.", vbOKOnly + vbExclamation, "Ungültige E-Mail"
        Cancel = True
    End If
End Sub
```

Listing 2: Prozedur zum Prüfen von E-Mails direkt nach der Eingabe

```
Private Sub txtPLZ_BeforeUpdate(Cancel As Integer)
    If Not IsNumeric(Me!txtPLZ) Then
        MsgBox "Die PLZ darf nur aus Zahlen bestehen.", _
                        vbOKCancel + vbExclamation, _
                            "Ungültige Postleitzahl"
        Cancel = True
    End If
End Sub
```

Dadurch erscheint eine Meldung, wenn der Benutzer andere Zeichen als Zahlen eingibt.

Die Alternative ist, direkt die Eingabe von Zahlen zu unterbinden. Wie das gelingt, zeigen wir im Beitrag **Textfeld nur mit bestimmten Zeichen füllen** (**www.access-im-unternehmen.de/1379**).

Wir könnten hier noch einen Schritt weitergehen und die Validierung der Postleitzahl vom angegebenen Land abhängig machen. Darauf verzichten wir an dieser Stelle jedoch.

Das Feld Kundennummer

Das Feld **Kundennummer** nimmt eine besondere Stellung ein. Es soll in unserem Beispiel eine Kundennummer angeben, die mit **99** beginnt und danach sechs Ziffern, bestehend aus Nullen und dem Wert des Primärschlüsselfeldes. Für den Primärschlüsselwert **123** also beispielsweise **99000123**.

Dieser Wert soll möglichst automatisch erzeugt werden, wenn der Benutzer die ersten Daten in den Kundendatensatz einträgt. Wie man dies realisieren kann, lernen Sie im Beitrag **Nummern für Bestellungen generieren** (**www.access-im-unternehmen.de/1377**).

Dort entnehmen wir die folgenden beiden Prozeduren, von denen die erste ausgelöst wird, wenn der Benutzer ein beliebiges Feld bearbeitet. Dieses prüft, ob es sich um einen neuen, leeren Datensatz handelt und stellt die Eigenschaft **TimerInterval** auf **100** ein:

```
Private Sub Form_Dirty(Cancel As Integer)
    If Me.NewRecord Then
        Me.TimerInterval = 100
    End If
End Sub
```

Dadurch wird das Ereignis **Bei Zeitgeber** 100 Millisekunden später ausgelöst. Dieses stellt die Kundennummer auf einen Wert ein, welche aus **99000000** und dem Primärschlüsselwert zusammengesetzt ist.

Danach stellt sie die Eigenschaft **TimerInterval** wieder auf **0** ein, was dazu führt, dass die Prozedur **Form_Timer** vorerst nicht erneut ausgelöst wird:

```
Private Sub Form_Timer()
    Me!txtKundennummer = Format(Me!ID, "99000000")
    Me.TimerInterval = 0
End Sub
```

Der Hintergrund ist, dass der Autowert für das Primärschlüsselfeld erst einen Augenblick nach dem Auslösen von **Bei Geändert** gesetzt wird und wir erst dann auf diesen zugreifen können, um ihn als Grundlage für die Kundennummer zu verwenden.

Kundennummer vor Zugriff schützen

Wenn wir die Kundennummer wie in diesem Fall beim Anlegen eines neuen Kunden per Code festlegen, sollten wir das irgendwie für den Benutzer kenntlich machen.

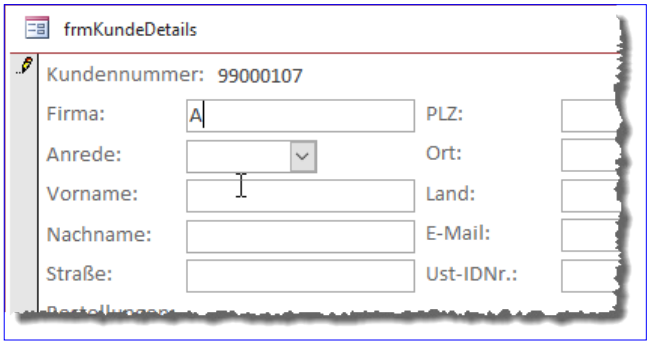


Bild 6: Automatisches Anlegen der Kundennummer

Access-Applikation mit Runtime installieren

Christoph Jüngling, https://www.juengling-edv.de

Office-Dokumente wie Word- oder Excel-Dateien lassen sich mittlerweile auf fast allen Geräten lesen. Wenn das nicht möglich ist, kann man diese oft in die jeweils vorhandene Textverarbeitung oder Tabellenkalkulation importieren. Bei Datenbankanwendungen ist das anders: Dass der Entwickler eine Vollversion von Microsoft Access auf dem Rechner hat, ist Voraussetzung. Aber was ist, wenn wir eine Datenbankanwendung in einem Unternehmen an viele Arbeitsplätze verteilen oder diese online an Kunden verkaufen wollen? Muss in dem Fall für alle User ebenfalls eine Vollversion von Access beschafft werden? Glücklicherweise lautet die Antwort nein. Es gibt eine kostenlose Runtime-Version von Access, die das Nötigste für den Betrieb von Access-Anwendungen mit sich bringt. Der vorliegende Artikel zeigt, welche Vorbereitungen dafür in unserer Applikation erforderlich sind und wie man die Runtime in ein eigenes Setup integriert.

Microsoft Access ist ein tolles Produkt, aber es hat – global gesehen – leider einen entscheidenden Nachteil: Es kostet Geld. Na gut, das kann man verstehen, wir alle müssen ja Miete zahlen, und das geht nicht vom guten Willen (will heißen „Spenden“) allein. Doch glücklicherweise gibt es für einige von uns eine Möglichkeit, wenigstens diese Kosten zu vermeiden – und dabei rede ich natürlich nicht von „Hacks“, nein, das geht ganz legal!

Historisches

Wegen der anfallenden Lizenzkosten kann man nicht grundsätzlich davon ausgehen, dass Access beim User schon installiert ist. Je nach aktueller Lizenzpolitik seitens Microsoft gibt es vielleicht eine Sparversion von Office, die Access außen vor lässt und dafür günstiger ist. Oder die IT-Administration hat entschieden, dass ohnehin niemand Access braucht, weil Excel ja genauso gut ist und auch „Datenbank kann“.

Was auch immer nun der Grund dafür ist, Access nicht zu installieren, für uns Entwickler stellt sich die Frage, wie wir damit umgehen. Immerhin stellt Microsoft schon seit längerem eine „Access-Runtime-Version“ bereit. In der Vergangenheit musste der Entwickler diese gelegentlich bezahlen, damit er sie dann kostenfrei an seine Anwender weitergeben konnte. Seit längerem jedoch ist die Runtime kostenlos herunterladbar. Dies verursacht dem Entwickler also keine Zusatzkosten mehr, und vor allem dürfen wir diese Runtime auch überall kostenfrei installieren.

Auch in Hinsicht dessen, was man als Entwickler darf, scheinen sich die Ansichten bei Microsoft etwas gelockert zu haben. Ich erinnere mich noch an meine früheren Recherchen darüber, nach denen ich die Runtime meinen Kunden nicht hätte bereitstellen dürfen; sie mussten sie sich damals selbst herunterladen (einschließlich der Registrierung). Dazu habe ich aktuell nichts mehr auf der Microsoft-Website gefunden. Selbstverständlich soll dies keine Rechtsberatung darstellen.

Das Problem: Die Kosten

Wie wir alle wissen, ist Microsoft Access eine lizenzpflichtige Software. Das bedeutet, dass jeder, der sie nutzen will, dafür einen gewissen Betrag investieren muss. Das schließt auch alle Anwender mit ein, und das kann für ein kleines Unternehmen ganz schön ins Geld gehen! Natürlich gibt es verschiedene Möglichkeiten, diese Kosten in Grenzen zu halten, zum Beispiel das Action Pack (**https://**

partner.microsoft.com/de-de/membership/action-pack) oder Volumenlizenzen, aber darum soll es hier nicht gehen.

Das mit den Kosten ist übrigens auch dann der Fall, wenn die Anwender eine individuell entwickelte Access-Applikation benutzen, die ja bereits mit einer lizenzierten Access-Version erstellt wurde! Klingt vielleicht unfair, ist aber so, daran können wir nichts ändern. Oder doch?

Die Lösung: Runtime-Version verwenden

Die Erklärung ist einfach: Diese als **.accdb**, **.accde** oder **.accdr** bei den Anwendern installierte (oder einfach aufgespielte) Applikation ist ohne weitere Hilfe leider nicht lauffähig. Wir können mit Access keine eigenständigen Programme schreiben, denn es gibt keinen Compiler/Linker wie beispielsweise bei den Programmiersprachen C++ oder VB.Net.

Es entsteht also keine **.exe**-Datei. Stattdessen muss Access immer mit im Boot sein, es interpretiert die Datenbankdatei und führt die darin beschriebenen Aktionen aus.

Wenn wir allerdings die Runtime-Version von Access nutzen, wird kein Access mehr benötigt, um die Applikation laufen zu lassen. Allerdings gibt es dabei noch ein paar Einschränkungen, die wir uns genauer anschauen müssen.

Unterschiede zur Vollversion

Natürlich gibt es Unterschiede zwischen Runtime und Vollversion, sonst würde ja niemand mehr die Lizenz von Access kaufen, sondern immer gleich zur kostenlosen Runtime-Version greifen. Der wichtigste Unterschied ist der, dass man mit der Runtime eine Datenbank zwar benutzen, aber nicht entwickeln kann. Entwickeln soll hier stellvertretend für alles stehen, was ein Entwickler so macht, unter anderem also Formulare, Berichte und andere Datenbankobjekte bearbeiten und den Navigationsbereich anzeigen und nutzen. Das alles geht in der Runtime nicht mehr, dafür braucht man die Vollversion von Access.

Ein echtes Problem ist das sicher nicht, denn genau dies alles wird der reine Anwender in aller Regel sowieso nicht machen. Und mal ehrlich: Als Entwickler würden wir es sogar begrüßen, wenn der Anwender das alles auch gar nicht kann.

Nur der Vollständigkeit halber sei noch gesagt, dass die Bearbeitung der Daten auch mit der Runtime natürlich ohne Probleme funktioniert!

Vorüberlegungen

Bevor man eine Access-Applikation für die Runtime-Version freigibt, sollten also einige Überlegungen angestellt werden. Wir benötigen sinnvollerweise:

- Frontend-/Backend-Trennung,
- ein eigenes Ribbon,
- einen Mechanismus zum automatischen Start aller benötigten Formulare etc.,
- 32-Bit oder 64-Bit,
- ein Setup
- und wir sollten unser Setup testen.

Frontend-/Backend-Trennung

Die Frontend-/Backend-Trennung und alle damit verbundenen Aktionen (zum Beispiel automatische Tabelleneinbindung) ist generell sinnvoll. Der wichtigste Grund ist, dass damit die Applikation einfach durch einen Austausch der Datei aktualisiert werden kann (mit oder ohne Setup), ohne die bereits eingegebenen Daten zu verlieren.

Eine mögliche Vorgehensweise dazu ist denkbar einfach:

- Im ersten Schritt duplizieren wir mit Hilfe des Windows-Explorers die Datenbankdatei (Copy/Paste).

- In einer der beiden Dateien löschen wir nun alle Tabellen. Diese Datei nennen wir **Frontend**.

- In der anderen Datei löschen wir alles außer den Tabellen. Diese Datei nennen wir **Backend**.

- Nun verknüpfen wir die Tabellen aus dem Backend in das Frontend (**Externe Daten|Neue Datenquelle|Aus Datenbank|Access**).

Weitergehende Informationen finden Sie im Beitrag **Setup für Access-Anwendungen** (**www.access-im-unternehmen.de/1316**) unter **Exkurs: Frontend-/Backend-Trennung**.

Ribbon

Da das Access-eigene Ribbon in der Runtime-Umgebung nicht angezeigt wird, müssen wir als Entwickler ein eigenes vorbereiten. Hierzu gibt es bereits zahlreiche Artikel zu Ribbons. Die Alternative besteht darin, dass wir gar nicht mit Ribbons arbeiten, sondern alle Funktionen der Anwendung aus einem beim Anwendungsstart geöffneten Formular (nicht datengebunden) heraus starten.

Dieses Ribbon sollte mindestens die Icons beinhalten, die der Anwender während der üblichen Arbeiten benötigt, zum Beispiel die Gruppen **Zwischenablage**, **Sortieren und Filtern**, **Datensätze** und **Suchen**. Der Vorteil bei Verwendung der eingebauten Bestandteile ist, dass sie auch ihre Automatik (aktiv/inaktiv je nach Kontext) mitbringen.

In Bild 1 sehen Sie ein Beispiel dafür. Natürlich sind Sie in Ihrer eigenen Applikation frei in der Gestaltung dieses Ribbons. Wenn Sie jedoch keines vorbereiten, wird in der Runtime auch keines enthalten sein.

Automatischer Start

Der automatische Start aller benötigten Objekte (zum Beispiel Ribbon, Formulare et cetera) muss berücksichtigt werden, da der Navigationsbereich nicht zugänglich ist, wenn die Applikation in der Runtime-Umgebung läuft. Wir brauchen also entweder ein **Autoexec**-Makro, ein Startformular mit Code für das **Form_Open**-Ereignis, oder wir nutzen den in dem Artikel **Code beim Öffnen der Anwendung: Ribbon** (**www.access-im-unternehmen.de/1369**) beschriebenen Trick.

Der Start des Ribbons erfolgt durch Access automatisch, wenn wir dieses in der Datenbank korrekt eingetragen haben. Dazu muss die Tabelle **USysRibbons** mit der vorgegebenen Struktur angelegt und mit der Beschreibung der Ribbon-Definition gefüllt sein. Außerdem muss im Menüpunkt **Datei|Optionen|Aktuelle Datenbank|Menüband- und Symbolleistenoptionen|Name des Menübands** der Name des initial zu ladenden Ribbons eingetragen sein. Wie das genau geht, ist in den im vorigen Abschnitt verlinkten Artikeln bereits beschrieben.

Wenn Sie ein Programm wie zum Beispiel den **Ribbon-Admin 2016** (**https://shop.minhorst.com/access-tools/309/ribbon-admin-2016?c=78**) zur Erstellung eines Ribbons verwenden, wird alles Benötigte bereits von diesem erledigt.

Der Start der weiteren Objekte unterscheidet sich in der Runtime-Version nicht von dem, was Sie wahrscheinlich auch in der **.accdb**/**.accde**-Datei längst tun.

Bitbreite

Mit „Bitbreite“ ist selbstverständlich nicht im wörtlichen Sinne die Breite eines Bits gemeint. Näheres dazu gibt es in der Wikipedia. Es geht konkret um die Frage 32-Bit oder

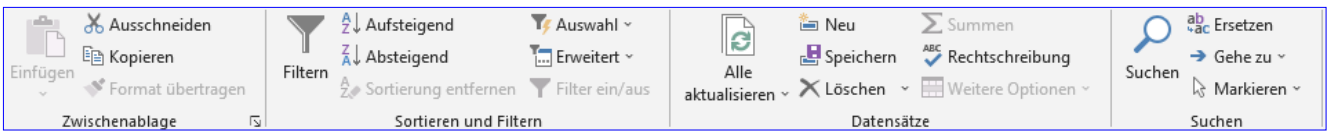


Bild 1: Wichtige Befehle des Ribbons

64-Bit, und zwar im Hinblick auf Access, nicht Windows. Wir müssen im Code einiges vorbereiten, wenn unsere Access-Applikation in beiden Welten lauffähig sein soll. Dieses Thema ist in dem Artikel **VBA unter Access mit 64 Bit** (**www.access-im-unternehmen.de/961**) ausführlich besprochen worden.

Allerdings gibt es im Hinblick auf unsere Installation einen Fallstrick: Die **.accdb**-Datei mag nach sorgfältigen Vorbereitungen unter beiden Bitbreiten laufen, jedoch stimmt das für die kompilierte **.accde**-Datei nicht: Diese muss für 32-Bit und 64-Bit in der jeweiligen Access-Variante erzeugt werden, was unseren Buildprozess ein wenig komplizierter macht. Das ist allerdings nicht neu, sondern schon seit einem Vierteljahrhundert so.

Setup

Damit sind wir gedanklich beim Setup angekommen. Es erleichtert dem Anwender, die Applikation zu installieren, andernfalls müsste er sich um das Aufspielen an die richtige Stelle, die Verknüpfung mit Startmenü und Desktop-Icon und vielleicht noch einiges mehr selbst kümmern. Bei der späteren Deinstallation wiederum müsste er ebenfalls an all dies denken. Wenn der Benutzer das nicht aus eigener Kraft schafft, müsste jedes Mal ein Administrator aktiv werden.

Durch ein Setup bekommen auch unerfahrene Anwender schnell einen lauffähigen Zustand. Sinnvollerweise schließen wir die Installation der Runtime-Version von Access gleich mit ein, sofern diese benötigt wird.

Es gibt noch eine Reihe anderer Vorteile und Möglichkeiten, die wir in den Beiträgen **Setup für Access-Anwendungen** (**www.access-im-unternehmen.de/1316**), **Setup für Access: Umsetzung mit InnoSetup** (**www.access-im-unternehmen.de/1326**), **Setup für Access: Vertrauenswürdige Speicherorte** (**www.access-im-unternehmen.de/1333**) und **Setup für Access-Applikationen, Restarbeiten** (**www.access-im-unternehmen.de/1355**) ausführlich besprochen haben. Im Rahmen dieses Artikels soll nur darauf eingegangen werden, wie man die Runtime-Version in unser Setup integriert.

Test

Für den Test unseres Setups ist es erforderlich, dass wir einen zweiten Rechner haben, egal ob nun physisch oder als virtuelle Maschine. Denn eine Parallelinstallation von Vollversion und Runtime auf einem einzigen Rechner ist problematisch.

Eine Windows-Installation in einer virtuellen Maschine (zum Beispiel Microsoft Virtual PC, VMWare Workstation oder Oracle VirtualBox) bietet sich hierbei an, da mittels eines Snapshots der Grundzustand sehr einfach gesichert und danach jederzeit wiederhergestellt werden kann.

Wenn wir dann auch noch 32-Bit und 64-Bit unterstützen wollen, brauchen wir mehrere virtuelle Maschinen. Beachten Sie dabei, dass Sie die erforderliche Anzahl Lizenzen besitzen. Ich denke aber, das oben verlinkte Actionpack ist schon eine gute Grundlage dafür.

Zu testen wäre in erster Linie, ob

- die Runtime immer installiert wird, wenn sie gebraucht wird.
- die Runtime nicht installiert wird, wenn diese oder die Vollversion bereits installiert sind.
- die Applikation mit der richtigen Bitbreite installiert wird (falls wir diese Unterscheidung machen).

Das bedeutet, dass die folgenden VMs vorhanden sein sollten:

- Mit Access, 32 Bit
- Mit Access-Runtime, 32 Bit
- Mit Access, 64 Bit

XRechnung, Teil 2: Rechnungen einlesen

Nachdem wir im ersten Teil dieser Beitragsreihe gezeigt haben, wie Sie aus Daten wie Kundeninformationen, Rechnungsdatum und Rechnungspositionen ein XML-Dokument im XRechnung-Format erstellen, wollen wir in diesem Beitrag den umgekehrten Weg gehen: Wir wollen die Daten aus einer so generierten Rechnung auslesen und zurück in das Datenmodell schreiben. Dazu sind vor allem Fähigkeiten im Auslesen von XML-Dokumenten erforderlich – und der Umgang mit Namespace-Deklarationen in diesen Dokumenten. Nach der Lektüre dieses Beitrags sind Sie in der Lage, die Daten aus einer XRechnung automatisiert in ein entsprechendes Datenmodell einzulesen.

Ausgangssituation

Für bestimmte Empfänger müssen Rechnungen mittlerweile in einem automatisiert lesbaren Format vorliegen, zum Beispiel im Format **XRechnung**. Dieses Format hat gegenüber Rechnungen im PDF-Format den Vorteil, dass alle Informationen an der entsprechenden in der vorgegebenen XML-Struktur zu finden sind und diese somit maschinell verarbeitet werden können. Im Beitrag **XRechnung, Teil 1: Rechnungen generieren** (**www.access-im-unternehmen.de/1277**) haben wir die Daten aus einer Rechnungsverwaltung per Knopfdruck in ein solches Dokument geschrieben.

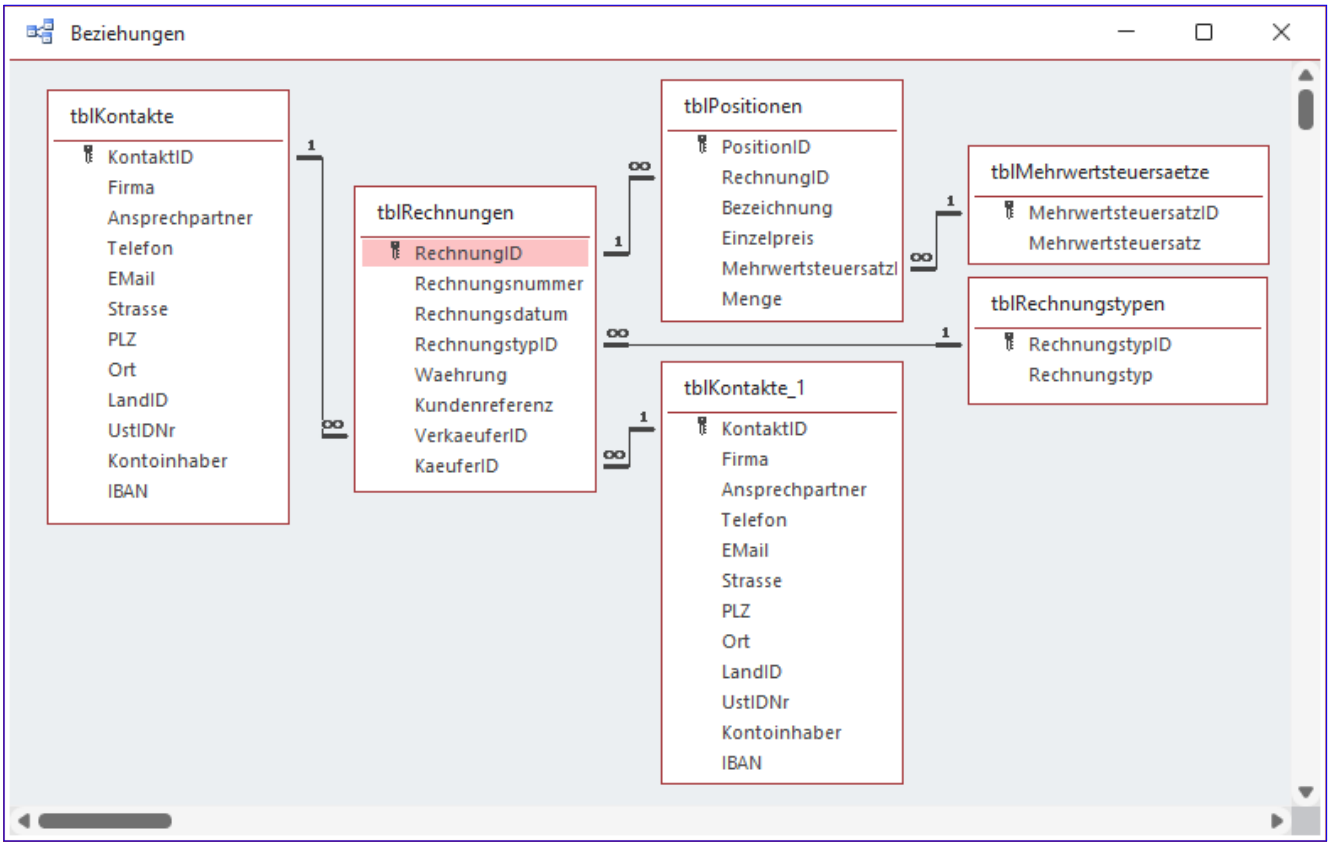


Bild 1: Datenmodell für die Daten aus einer XRechnung

Nun ist es zu erwarten und auch wünschenswert, dass sich solche Formate allgemein durchsetzen, damit niemand mehr Rechnungen in gedruckter Form oder als PDF entgegennehmen und diese händisch verarbeiten muss. Deshalb werden früher oder später alle Unternehmen in der Lage sein müssen, solche Rechnungen zu verarbeiten.

Im vorliegenden Beitrag wollen wir die Daten aus einem solchen XRechnung-Dokument in ein vorgegebenes Datenmodell einlesen können. In einem weiteren Beitrag schauen wir uns dann an, wie wir die eingelesenen Daten in einem Bericht als herkömmliche Rechnung, lesbar für das menschliche Auge, präsentieren können.

Änderung im Datenmodell

Im Vergleich zum Datenmodell des ersten Teils der Beitragsreihe haben wir die Tabelle **tblRechnungen** leicht angepasst. Wir haben ein Feld namens **Rechnungsnummer** zum Speichern der jeweiligen Rechnungsnummer hinzugefügt und das Feld **RechnungID** mit dem Datentyp **Autowert** versehen. Zuvor mussten wir die Beziehung zwischen dem Feld **RechnungID** der Tabelle **tblPositionen** und der Tabelle **tblRechnungen** löschen und diese anschließend erneut anlegen.

Außerdem haben wir der Tabelle **tblPositionen** noch ein Feld namens **Position** hinzugefügt, mit dem die in der XRechnung angegebenen Positionsnummern gespeichert werden können, sowie ein Feld namens **Einheit**.

Beispielhafter Import

Der Standard der XRechnung umfasst natürlich alle denkbaren Informationen. Diese können wir im Rahmen dieses Beitrags nicht alle erfassen. Es geht hier allein darum, die grundlegenden Techniken für die Erfassung der gängigsten Informationen zu präsentieren.

Wenn Sie oder Ihre Kunden es mit Daten in XRechnungen zu tun bekommen, deren Übertragung hier nicht berücksichtigt wurde, so lernen Sie dennoch die Techniken, die nötig sind, die notwendigen Ergänzungen vorzunehmen.

Datenmodell für die Daten aus der XRechnung

Das Datenmodell finden Sie in Bild 1. Jede Rechnung wird grundsätzlich in der Tabelle **tblRechnungen** erfasst und enthält dort einige grundlegende Daten wie das Rechnungsdatum, den Rechnungstyp, die Währung, eine Kundenreferenz sowie Verweise auf den Käufer und den Verkäufer. Käufer und Verkäufer werden in je einem Datensatz der Tabelle **tblKontakte** gespeichert und über Fremdschlüsselfelder der Tabelle **tblRechnungen** zugewiesen. Der Rechnungstyp wird aus einer Tabelle namens **tblRechnungstypen** ausgewählt.

Die einzelnen Rechnungspositionen landen schließlich in der Tabelle **tblPositionen**, die über das Fremdschlüsselfeld **RechnungID** dem jeweiligen Datensatz aus der Tabelle **tblRechnungen** zugewiesen werden.

Jede Position enthält Bezeichnung, Einzelpreis, Mehrwertsteuersatz und Menge, wobei der Mehrwertsteuersatz wiederum über ein Fremdschlüsselfeld aus der Tabelle **tblMehrwertsteuersaetze** ausgewählt wird.

Einlesen der Basisdaten

Der Anfang des XRechnung-Dokuments sieht wie in Listing 1 aus. Hier sehen wir zunächst das Root-Element **ubl**, das einige Namespaces aufweist, um die wir uns später explizit kümmern werden.

Als erste untergeordnete Elemente folgen nun bereits die grundlegenden Rechnungsdaten. Das Element **cbc:CustomizationID** liefert das Format, in dem die Rechnung verfasst wurde, hier in der Version 1.2.

Das Element **cbc:ID** liefert die Rechnungsnummer, **cbc:IssueDate** das Rechnungsdatum. Mit dem Wert **380** im Feld **cbc:InvoiceTypeCode** erhalten wir einen Hinweis auf die Art der Rechnung. **380** steht für **Commercial Invoice**.

Das Element **cbc:DocumentCurrencyCode** liefert das Kürzel für die Währung der Rechnung. In der Regel

finden wir hier die Einstellung **EUR** für Euro vor. Das Feld **cbc:BuyerReference** nimmt die sogenannte Leitweg-ID auf.

Außerdem gibt es noch einige weitere Elemente für verschiedene Referenzen wie **OrderReference**, **ContractDocumentReference**, **ProjectReference** et cetera, die wir an dieser Stelle jedoch nicht verarbeiten wollen.

Wir wollen den Einlesevorgang zunächst VBA-gesteuert ausführen, daher erstellen wir zuerst eine Prozedur, mit der wir die einzulesende Datei per Dateiauswahl-Dialog ermitteln und dann die eigentliche Prozedur zum Einlesen aufrufen. Die Prozedur zum Initialisieren des Einlesevorgangs sieht wie folgt aus:

```
Public Sub Test_XRechnungEinlesen()
    Dim strPfad As String
    strPfad = OpenFileName(CurrentProject.Path, "XRechnung auswählen", "XRechnung (*.xml)")
    XRechnungEinlesen strPfad
End Sub
```

Die Funktion **OpenFileName** finden Sie im Modul **mdlFileDialog**.

Die Funktion XRechnungEinlesen

Danach starten wir mit der Funktion **XRechnungEinlesen** den eigentlichen Einlesevorgang. Dazu prüfen wir in einer **If...Then**-Bedingung zunächst, ob ein Dateipfad mit **strPfad** übergeben wurde und ob die Datei überhaupt vorhanden ist (siehe Listing 2).

Ist das der Fall, erstellt die Funktion ein neues Objekt des Typs **MSXML2.DOMDocument60**. Um diese verwenden zu können, benötigen wir einen Verweis auf die Bibliothek

```
<ubl:Invoice xmlns:ubl="urn:oasis:names:specification:ubl:schema:xsd:Invoice-2"
    xmlns:cac="urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2"
    xmlns:cbc="urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="urn:oasis:names:specification:ubl:schema:xsd:Invoice-2
    http://docs.oasis-open.org/ubl/os-UBL-2.1/xsd/maindoc/UBL-Invoice-2.1.xsd">
    <cbc:CustomizationID>urn:cen.eu:en16931:2017#compliant#urn:xoev-de:kosit:standard:xrechnung_1.2</cbc:CustomizationID>
    <cbc:ID>1234</cbc:ID>
    <cbc:IssueDate>2020-09-24</cbc:IssueDate>
    <cbc:InvoiceTypeCode>380</cbc:InvoiceTypeCode>
    <cbc:DocumentCurrencyCode>EUR</cbc:DocumentCurrencyCode>
    <cbc:BuyerReference>1234</cbc:BuyerReference>
    <cac:OrderReference>
        <cbc:ID/>
    </cac:OrderReference>
    <cac:ContractDocumentReference>
        <cbc:ID/>
    </cac:ContractDocumentReference>
    <cac:ProjectReference>
        <cbc:ID/>
    </cac:ProjectReference>
    ...
</ubl>
```

Listing 1: Basisdaten der Rechnung im XML-Dokument

```
Public Function XRechnungEinlesen(strPfad As String) As Long
    Dim objXML As MSXML2.DOMDocument60
    Dim objInvoice As MSXML2.IXMLDOMNode
    Dim db As DAO.Database
    Set db = CurrentDb
    If Not Len(Dir(strPfad)) = 0 Then
        Set objXML = New MSXML2.DOMDocument60
        objXML.SetProperty "SelectionNamespaces", "xmlns:ubl='urn:oasis:names:specification:ubl:schema:xsd:Invoice-2'" _
            & "xmlns:cac='urn:oasis:names:specification:ubl:schema:xsd:CommonAggregateComponents-2' " _
            & "xmlns:cbc='urn:oasis:names:specification:ubl:schema:xsd:CommonBasicComponents-2' " _
            & "xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'"
        objXML.Load strPfad
        Set objInvoice = objXML.childNodes.Item(0)
        XRechnungEinlesen = RechnungsdatenEinlesen(objInvoice, db)
    End If
End Function
```

Listing 2: Startprozedur zum Einlesen der XRechnung

Microsoft XML, v6.0, die wir im **Verweise**-Fenster, das wir mit dem Menübefehl **Extras|Verweise** des VBA-Editors öffnen, markieren müssen (siehe Bild 2).

Nun kommt ein entscheidender Schritt, ohne den wir dieses mit einigen Namespace-Präfixen gespickte XML-Dokument nicht einlesen können: Wir stellen die Eigenschaft **SelectionNamespaces** mit der Methode **SetProperty** auf einen Wert ein, der die Definition der im Element **ubl:Invoice** des XML-Dokuments angegebenen Namespaces enthält.

Danach können wir die **Load**-Methode des Objekts **objXML** nutzen, um die unter **strPfad** angegebene XRechnung zu laden. Dann referenzieren wir das Root-Element, das wir mit **objXML.childNodes.Item(0)** ermitteln, mit der Variablen **objInvoice** und übergeben diese an die erste Unterfunktion **RechnungsdatenEinlesen**. Dieser übergeben wir auch den zuvor mit **CurrentDb** ermittelten Verweis auf das **Database**-Objekt der aktuellen Datenbankdatei.

Die Funktion **RechnungsdatenEinlesen** soll nach erfolgreichem Einlesen in die Tabelle **tblRechnungen** den Primärschüsselwert des neu erstellten Datensatzes zurückliefern, den wir wiederum als Rückgabewert der aufrufenden Funktion **XRechnungEinlesen** festlegen.

Funktion zum Einlesen der Rechnungsdaten

Die Funktion **RechnungsdatenEinlesen** erwartet das Root-Element sowie einen Verweis auf das aktuelle **Database**-Objekt als Parameter (siehe Listing 3).

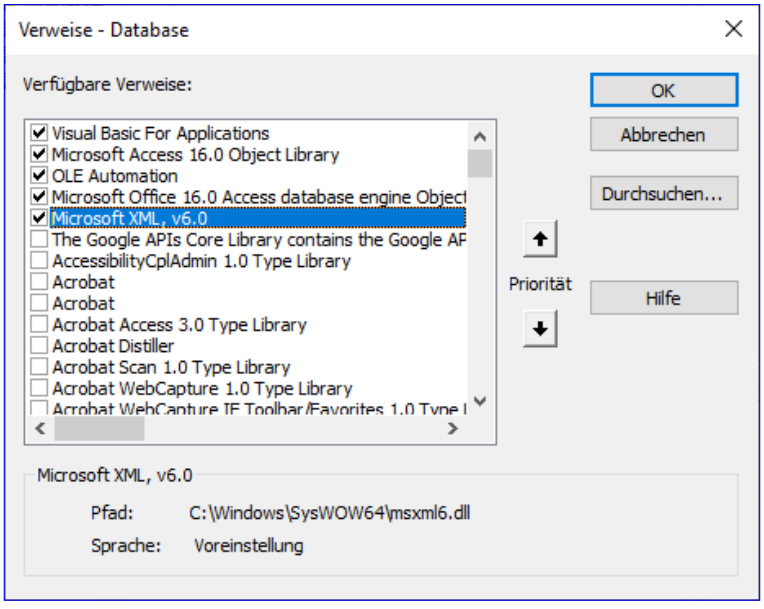


Bild 2: Hinzufügen eines Verweises auf die XML-Bibliothek

Sie deklariert einige Variablen des Typs **IXMLDOMNode**, um Unterelemente von **objInvoice** aufzunehmen, sowie ein Element des Typs **IXMLDOMNodeList** für die Rechnungspositionen in der XRechnung. Außerdem benötigen

```
Private Sub RechnungsdatenEinlesen(objInvoice As MSXML2.IXMLDOMElement, db As DAO.Database) As Long
    Dim rst As DAO.Recordset
    Dim strID As String
    Dim datIssueDate As Date
    Dim lngInvoiceTypeCode As Long
    Dim strDocumentCurrencyCode As String
    Dim strBuyerReference As String
    Dim objKaeufer As MSXML2.IXMLDOMNode
    Dim objVerkaeufer As MSXML2.IXMLDOMNode
    Dim objPaymentMeans As MSXML2.IXMLDOMNode
    Dim objInvoiceLines As MSXML2.IXMLDOMNodeList
    Dim lngVerkaeuferID As Long
    Dim lngKaeuferID As Long
    Dim lngRechnungID As Long
    strID = TextEinlesen(objInvoice, "cbc:ID")
    datIssueDate = TextEinlesen(objInvoice, "cbc:IssueDate")
    lngInvoiceTypeCode = TextEinlesen(objInvoice, "cbc:InvoiceTypeCode")
    strDocumentCurrencyCode = TextEinlesen(objInvoice, "cbc:DocumentCurrencyCode")
    strBuyerReference = TextEinlesen(objInvoice, "cbc:BuyerReference")
    Set objKaeufer = ElementEinlesen(objInvoice, "//cac:AccountingCustomerParty/cac:Party")
    lngKaeuferID = AdresseEinlesen(objKaeufer, db)
    Set objVerkaeufer = ElementEinlesen(objInvoice, "cac:AccountingSupplierParty/cac:Party")
    lngVerkaeuferID = AdresseEinlesen(objVerkaeufer, db)
    Set objPaymentMeans = ElementEinlesen(objInvoice, "cac:PaymentMeans")
    ZahlungsinformationenSchreiben objPaymentMeans, db, lngVerkaeuferID
    Set rst = db.OpenRecordset("SELECT * FROM tblRechnungen WHERE 1=2", dbOpenDynaset)
    With rst
        .AddNew
        !Rechnungsnummer = strID
        !Rechnungsdatum = datIssueDate
        !RechnungstypID = lngInvoiceTypeCode
        !Waehrung = strDocumentCurrencyCode
        !Kundenreferenz = strBuyerReference
        !VerkaeuferID = lngVerkaeuferID
        !KaeuferID = lngKaeuferID
        lngRechnungID = !RechnungID
        .Update
    End With
    Set objInvoiceLines = ElementeEinlesen(objInvoice, "cac:InvoiceLine")
    PositionenEinlesen objInvoiceLines, db, lngRechnungID
    RechnungsdatenEinlesen = lngRechnungID
End Sub
```

Listing 3: Hauptprozedur zum Einlesen der Rechnungsdaten

wir eine **Recordset**-Variable namens **rst**, der wir die Tabelle **tblRechnungen** zuweisen und über die wir den neuen Rechnungsdatensatz anlegen. Daneben dienen einige weitere Variablen wie **strID**, **datIssueDate** et cetera dazu, die Werte aus der XRechnung nach dem Auslesen und vor dem Eintragen in den neuen Datensatz temporär aufzunehmen.

```
<cac:AccountingCustomerParty>
    <cac:Party>
        <cac:PostalAddress>
            <cbc:StreetName>Teststr. 1</cbc:StreetName>
            <cbc:AdditionalStreetName/>
            <cbc:CityName>Berlin</cbc:CityName>
            <cbc:PostalZone>12121</cbc:PostalZone>
            <cac:Country>
                <cbc:IdentificationCode>DE</cbc:IdentificationCode>
            </cac:Country>
        </cac:PostalAddress>
        <cac:PartyTaxScheme>
            <cbc:CompanyID>DE232323232</cbc:CompanyID>
            <cac:TaxScheme>
                <cbc:ID>VAT</cbc:ID>
            </cac:TaxScheme>
        </cac:PartyTaxScheme>
        <cac:PartyLegalEntity>
            <cbc:RegistrationName>Müller AG</cbc:RegistrationName>
        </cac:PartyLegalEntity>
        <cac:Contact>
            <cbc:Name>Klaus Müller</cbc:Name>
            <cbc:Telephone>0123-2121212</cbc:Telephone>
            <cbc:ElectronicMail>klaus@mueller.de</cbc:ElectronicMail>
        </cac:Contact>
    </cac:Party>
</cac:AccountingCustomerParty>
```

Listing 4: Element mit den Daten des Rechnungsempfängers

Basisdaten der Rechnung aus der XRechnung einlesen

Die ersten Anweisungen lesen die Werte der entsprechenden Elemente aus der XRechnung ein. Dazu nutzen diese verschiedene Hilfsfunktionen, die wir am Ende des Beitrags beschreiben. Die erste heißt **TextEinlesen** und sucht aus einem **IXMLDomNode**-Element den Text des mit dem zweiten Parameter übergebenen Wertes aus – in der ersten Anweisung beispielsweise das Element **cbc:ID**. Dieser Wert landet in der Variablen **strID**. Das Gleiche geschieht mit den Elementen **cbc:IssueDate**, **cbc:InvoiceTypeCode**, **cbc:DocumentCurrencyCode** und **cbc:BuyerReference**. Damit erhalten wir die Basisdaten der Rechnung.

Danach nutzen wir eine weitere Hilfsfunktion namens **ElementEinlesen**, um das Unterelement mit dem Namen **cac:AccountingCustomerParty/cac:Party** mit der Variablen **objKaeufer** zu referenzieren. Dieser Teilbereich der XRechnung sieht beispielsweise wie in Listing 4 aus.

Nachdem wir dieses Element aufgerufen haben, rufen wir die Funktion **AdresseEinlesen** auf und übergeben dieser das **IXMLDOMNode**-Element aus **objKaeufer** sowie einen Verweis auf das **Database**-Objekt. Der Übersicht halber beschreiben wir diese Funktion erst später. An dieser Stelle interessiert uns nur das Ergebnis dieser Funktion. Diese schreibt nämlich die Käuferdaten in die Tabelle **tblKontakte** und liefert den Wert des Primärschlüssels des neu hinzugefügten Datensatzes zurück. Diesen können wir dann in der Variablen **lngKaeuferID** zwischenspeichern und beim Erstellen eines neuen Rechnungsdatensatzes direkt in die Tabelle **tblRechnungen** schreiben.

Auf die gleiche Weise lesen wir auch noch die Daten des Verkäufers ein. Diese finden wir im Element **cac:AccountingSupplierParty/cac:Party**. Auch für dieses Element rufen wir die Funktion **AdresseEinlesen** auf, was dazu führt, dass auch der Verkäufer als neuer Datensatz in der

EPC-QR-Code per COM-DLL erstellen

Spätestens seit sich das Onlinebanking immer mehr auf das Smartphone verschiebt, wird das Eingeben von Rechnungsdaten wie langen IBANs oder Verwendungszwecken zu einer undankbaren Aufgabe. Und auch wenn die Papierrechnungen weniger werden und sich die Daten von PDF-Rechnungen leicht per Copy und Paste übertragen lassen, so ist doch der EPC-QR-Code eine tolle Erleichterung: Dieser QR-Code enthält alle für eine Überweisung benötigten Daten und viele Onlinebanking-Apps bieten mittlerweile die Möglichkeit, solche Codes mithilfe der Smartphone-Kamera einzulesen. Um dieses Feature in Access-Berichten bereitzustellen, benötigen wir erst einmal eines: Ein Tool, mit dem wir solche QR-Codes erstellen können. Dieser Beitrag zeigt, wie wir eine .NET-DLL programmieren, die uns diese Aufgabe abnimmt.

Für viele Anwendungsbereiche bietet .NET Bibliotheken und Tools, die dem VBA-Entwickler auf direktem Wege nicht zugänglich sind. Zum Glück wird es immer einfacher, diese Helferlein in Form beispielsweise von COM-DLLs auf der Basis von .NET zu programmieren und diese mit einer Schnittstelle auszustatten, auf die wir auch von Access aus leicht zugreifen können.

Für die Aufgabe, die wir uns in diesem Beitrag vornehmen, benötigen wir die folgenden Dinge:

- eine Definition, wie die Informationen zusammengestellt werden, die für das Einlesen in eine Überweisung nötig sind,
- eine Bibliothek, die das Erstellen von QR-Codes erlaubt und
- ein VB.NET-Projekt, das die ersten beiden zusammenführt und in Form einer COM-DLL für den Zugriff von VBA aus verfügbar macht.

Schließlich benötigen wir noch eine Datenbank, welche die damit zu erstellenden Bilddateien mit QR-Codes nutzt und in Rechnungsberichte einbettet, sodass diese vom Rechnungsempfänger per Smartphone-Kamera erfasst und für das schnelle Eintragen der Rechnungsdaten in das Überweisungsformular genutzt werden kann. Diese beschreiben wir in einem weiteren Beitrag namens **Rechnungsbericht mit EPC-QR-Code** (**www.access-im-unternehmen.de/1400**) in der nächsten Ausgabe.

Wie sieht der Inhalt des EPC-QR-Codes aus?

Als Erstes schauen wir uns an, welchen Aufbau der Text hat, der als Grundlage für das Erstellen des EPC-QR-Codes genutzt wird. EPC steht für European Payment Code.

Ein Beispiel für den Inhalt finden wir auf Wikipedia, die praktischerweise direkt die Bezahldaten für eine Spende an Wikipedia selbst im Beispielcode untergebracht haben. Der Inhalt sieht wie folgt aus:

```
BCD
001
1
SCT
BFSWDE33BER
Wikimedia Foerdergesellschaft
DE33100205000001194700
EUR123.45


Spende fuer Wikipedia
```

In diesem Beispiel sehen Sie einige Leerzeilen, und auch die letzte Zeile könnte noch mit Inhalt gefüllt sein. Daher hier die Zeilen mit der Beschreibung der Inhalte:

- Zeile 1 (Beispiel: **BCD**): Service Tag (fester Wert)
- Zeile 2 (Beispielwert: **002**): Version (**001** oder **002**)
- Zeile 3 (Beispielwert: **2**): Zeichencodierung (1=UTF-8, 2=ISO 8859-1, 3=ISO 8859-2, 4=ISO 8859-4, 5=ISO 8859-5, 6=ISO 8859-7, 7=ISO 8859-10, 8=ISO 8859-15)
- Zeile 4 (Beispielwert: **SCT**): Identifikation, dreistelliger Code – derzeit nur **SCT** (**SEPA Credit Transfer**)
- Zeile 5 (Beispielwert: **BFSWDE33BER**): BIC der Empfängerbank (in Version **001** erforderlich; in Version **002** innerhalb des EWR optional)
- Zeile 6 (Beispielwert: **Wikimedia Foerdergesellschaft**): Name des Zahlungsempfängers (maximal 70 Zeichen Text)
- Zeile 7 (Beispielwert: **DE33100205000001194700**): Internationale Bankkontonummer (IBAN) des Zahlungsempfängers
- Zeile 8 (Beispielwert: **EUR123.45**): Zahlungsbetrag (Format **EUR#.##**, zwischen 0.01 und 999999999.99, optional)
- Zeile 9 (Beispielwert: **CHAR**): Zweck (max. vierstelliger Code analog dem Textschlüssel nach DTA-Verfahren, optional)
- Zeile 10 (Beispielwert **RF18 5390 0754 7034**): Referenz (strukturierter 35-Zeichen-Code gem. ISO 11649 RF Creditor Reference, optional)
- Zeile 11 (Beispielwert: **Spende fuer Wikipedia**): Verwendungszweck (unstrukturierter maximal 140 Zeichen langer Text, optional)
- Zeile 12 (Beispielwert: ./.): Hinweis an den Nutzer (maximal 70 Zeichen, optional)

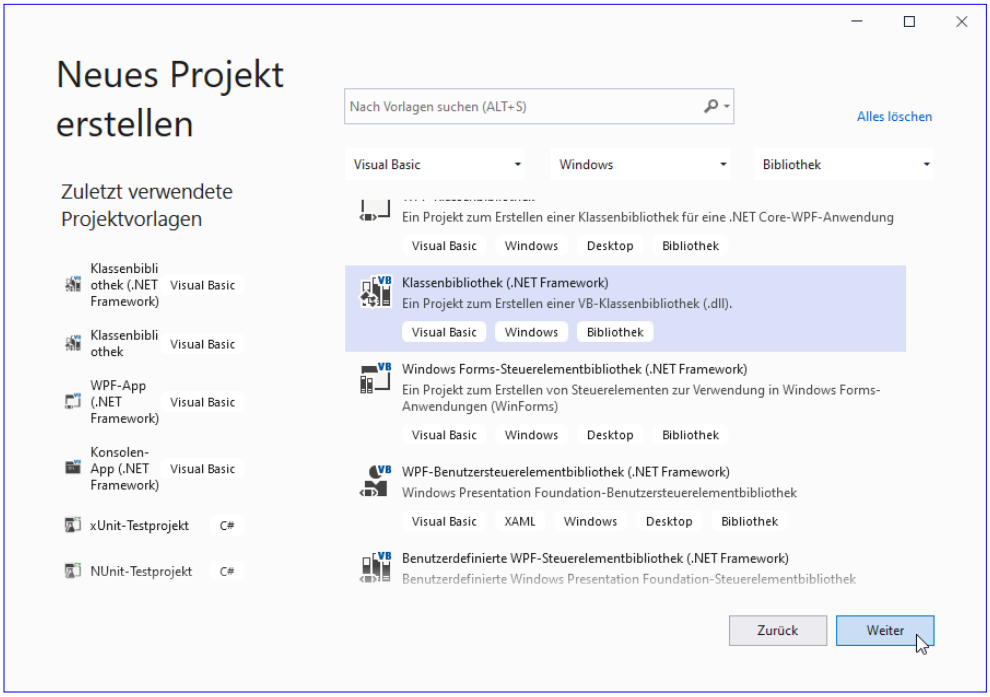


Bild 1: Auswahl des geeigneten Projekttyps

Wir wollen in diesem Beispielprojekt lediglich die wichtigsten Daten unterbringen. Wie Sie oben gesehen haben, legt man in der zweiten Zeile die Version des EPC-QR-Codes fest, der eigentlich nur einen Unterschied bewirkt: Die erste Version verlangt nach einem BIC, die zweite tut das nicht. Um das Beispiel einfach zu halten, nutzen wir hier die Version 2 und geben den BIC nicht an. Die ersten vier Zeilen enthalten Standardwerte, einige Zeilen bleiben leer,

so müssen wir nur die folgenden Daten mit der zu erstellenden COM-DLL entgegennehmen:

- Zahlungsempfänger
- IBAN
- Zahlungsbetrag
- Zweck

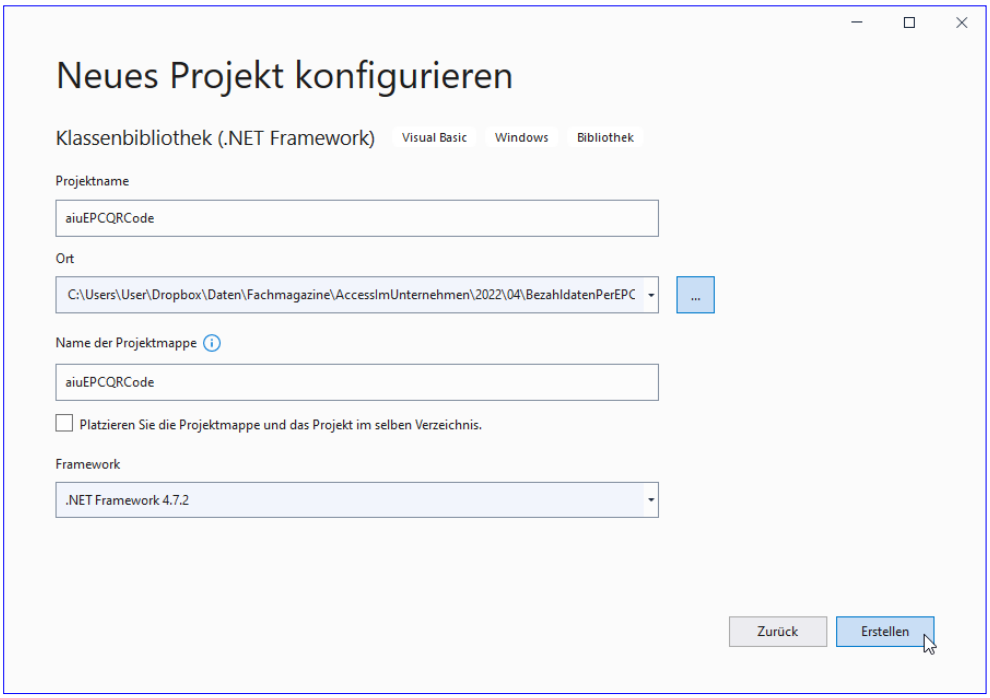


Bild 2: Eingabe von Zielordner und Projektname

.NET-Projekt für die COM-DLL erstellen

Damit können wir direkt mit der Erstellung der COM-DLL in Visual Studio beginnen. Visual Studio ist in der Community-Edition kostenlos. Scheuen Sie sich also nicht, es auszuprobieren – es macht Spaß, mal mit einer alternativen Entwicklungsumgebung zu arbeiten!

Nachdem Visual Studio gestartet ist, hier in der Version 2019, legen Sie ein neues Projekt an. Dazu verwenden wir die Vorlage **Klassenbibliothek (.NET-Framework)**, die Sie wie in Bild 1 auswählen und mit einem Klick auf die **Weiter**-Schaltfläche bestätigen.

Mit dem folgenden Dialog ist das Erstellen des Projekts bereits abgeschlossen: Hier geben Sie noch den Namen für das Projekt ein und legen den Zielordner fest.

Sie müssen keinen neuen Ordner anlegen, Visual Studio legt einen Ordner mit dem zuvor festgelegten Namen des Projekts an (siehe Bild 2).

Klassennamen ändern

Wenn Sie den Projektmappen-Explorer einsehen, finden Sie hier eine einzige Klasse namens **Class1.vb**. Den Namen dieser Klasse ändern wir auf **EPCQRCodeGenerator.vb** (siehe Bild 3).

Die anschließende Meldung, ob auch alle Verweise darauf geändert werden sollen, akzeptieren wir mit **Ja**. Damit ändert sich auch gleich der Name der Klassenbezeichnung im Modul selbst auf **EPCQRCodeGenerator**.

NuGet-Paket mit QR-Code-Bibliothek importieren

Bevor wir uns an das Programmieren begeben, fügen wir die für das eigentliche Erstellen des QR-Codes notwendige

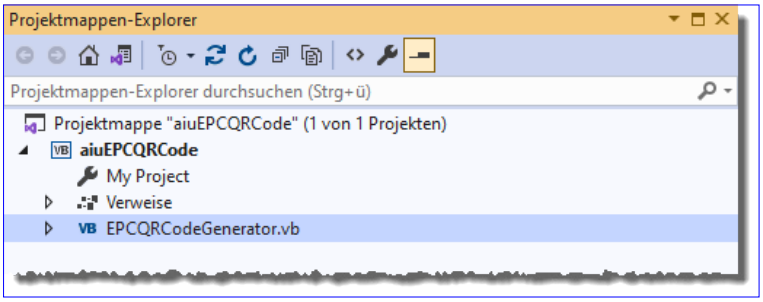


Bild 3: Ändern des Namens der einzigen Klasse

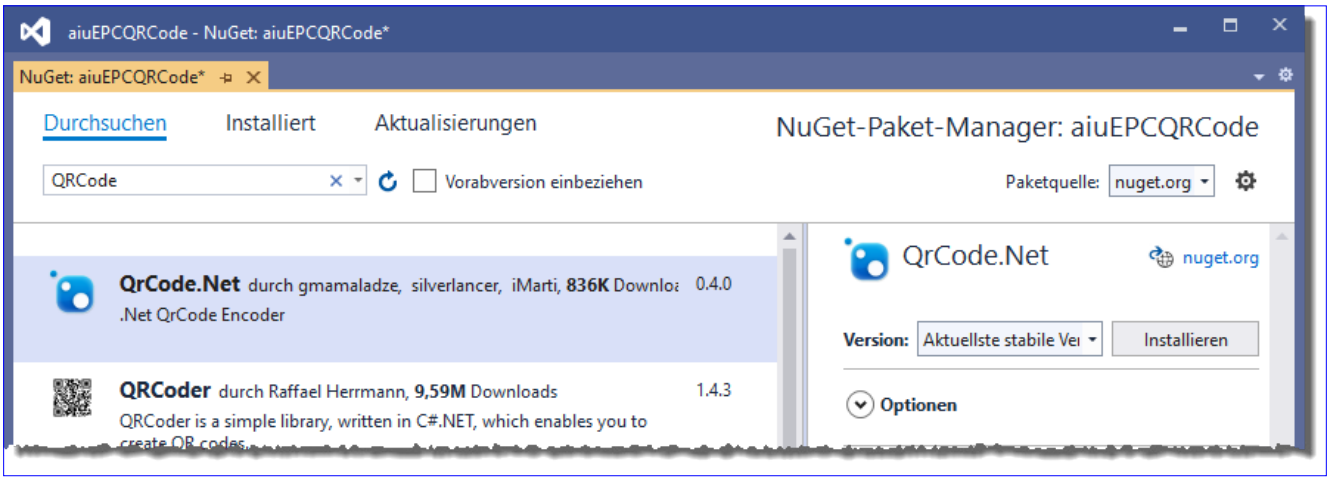


Bild 4: Hinzufügen des NuGet-Pakets **QrCode.Net**

Bibliothek zum Projekt hinzu. Dazu rufen Sie mit dem Menübefehl **Projekt|NuGet-Pakete verwalten ...** den NuGet-Manager auf. Hier klicken Sie auf den Bereich **Durchsuchen** und geben im Suchen-Feld den Text **QRCode** ein.

Daraufhin erscheint das Paket **QrCode.Net**, das wir anklicken und mit einem Klick auf die nun erscheinende Schaltfläche **Installieren** zum Projekt hinzufügen (siehe Bild 4). Dies legt ein paar Elemente im Projekt an, um die wir uns aber nicht weiter kümmern müssen.

Namespaces importieren

Um in der Datei **EPCQRCodeGenerator.vb** auf die Elemente des hinzugefügten NuGet-Pakets und auf einige andere Elemente zugreifen zu können, machen wir diese mit einigen **Imports**-Anweisungen verfügbar.

Diese landen direkt ganz oben in der Datei, also noch über der **Public Class**-Anweisung, und sehen wir folgt aus:

```
Imports System.Runtime.InteropServices
Imports Gma.QrCodeNet.Encoding
Imports Gma.QrCodeNet.Encoding.Windows.Render
Imports System.Drawing
```

Für den Import des Namespaces **System.Drawing** müssen wir überdies noch einen entsprechenden Verweis auf die gleichnamige Bibliothek hinzufügen. Dazu öffnen Sie mit dem Menübefehl **Projekt|Verweis hinzufügen...** den Dialog **Verweis-Manager** und suchen dort unter Assemblys nach dem Suchbegriff **System.Drawing**. Den gefundenen Eintrag fügen Sie durch Setzen eines Hakens

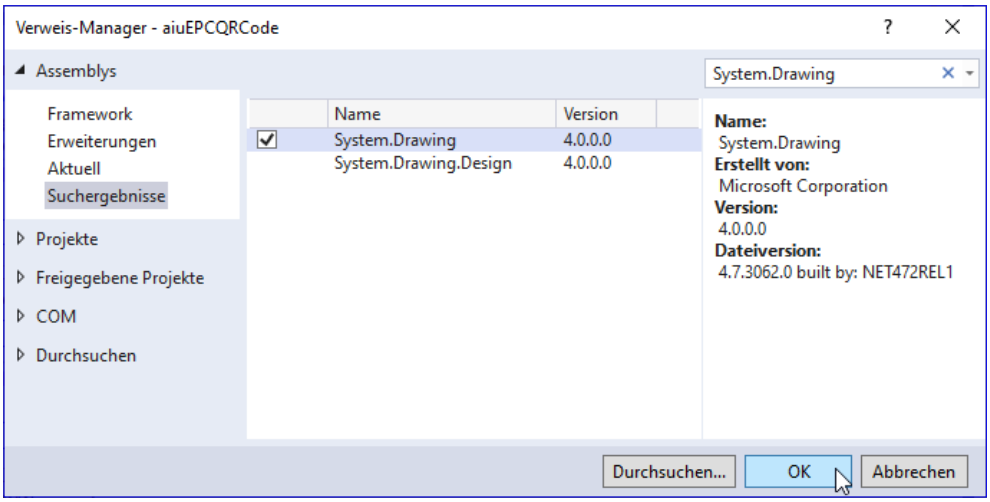


Bild 5: Hinzufügen von Verweisen zum Projekt

zum Projekt hinzu (siehe Bild 5).

Klasse zum Aufnehmen der Überweisungseigenschaften

Normalerweise würden wir nun einfach eine Klasse erstellen, welche einige öffentlich deklarierte Eigenschaften enthält, über die Sie vom VBA-Editor aus die für die Überweisung notwendigen Informationen an die DLL übergeben können. Diese Klasse würde jedoch auch einige Standardmember von .NET-Klassen bereitstellen wie **GetType** oder **ToString**.

Diese benötigen wir aber unter VBA nicht, also gehen wir einen kleinen Umweg. Dazu definieren wir eine öffentliche Schnittstelle, die wir dann implementieren.

Die Schnittstelle wird als solche mit dem Schlüsselwort **Public Interface** definiert, erhält eine spezielle Auszeichnung, damit sie über die COM-Schnittstelle für den Zugriff von außen bereitgestellt wird und sieht wie folgt aus:

```
<InterfaceType(ComInterfaceType.InterfaceIsDual)>
Public Interface IEPCQRCode
    Property Verwendungszweck As String
    Property IBAN As String
    Property Betrag As <Runtime.InteropServices.
                        MarshalAs(Runtime.InteropServices.
                           UnmanagedType.Currency)> Decimal
    Property Empfaenger As String
    Function Generate(strPfad As String,
                      ByRef strErgebnis As String) As Boolean
End Interface
```

Wir stellen also genau die gefragten Eigenschaften bereit plus eine Funktion namens **Generate**, mit der wir den QR-Code generieren und unter dem angegebenen Pfad speichern können.

```
<ClassInterface(ClassInterfaceType.None)>
Public Class EPCQRCodeGenerator
        Implements IEPCQRCode
    Public Property Verwendungszweck As String Implements IEPCQRCode.Verwendungszweck
    Public Property BIC As String Implements IEPCQRCode.BIC
    Public Property IBAN As String Implements IEPCQRCode.IBAN
    Public Property Betrag As Decimal Implements IEPCQRCode.Betrag
    Public Property Empfaenger As String Implements IEPCQRCode.Empfaenger
    ...
```

Listing 1: Erster Teil der Schnittstellen-Implementation

Implementierung der Schnittstelle

Die Schnittstelle implementieren wir unter VB.NET genauso, wie Sie es auch von VBA kennen. Sie erstellen eine Klasse und zeigen hier mit dem Schlüsselwort **Implements** und Angabe der Schnittstellenbezeichnung an, dass diese Klasse die Schnittstelle implementiert. Sollte dieses Thema noch neu für Sie sein: Die Implementierung einer Schnittstelle muss alle in der Schnittstellendefinition vorgegebenen Member umsetzen, also beispielsweise wie hier die **Property**-Eigenschaften und die Funktion **Generate**.

Den ersten Teil dieser Schnittstelle finden Sie in Listing 1. Wenn Sie schon einmal **Property**-Eigenschaften in Klassenmodulen mit VBA abgebildet haben, vermissen Sie möglicherweise die einzelnen **Property Get...**- und **Property Set/Let...**-Prozeduren.

Unter VB.NET könnte man eine solche Eigenschaft auch wie folgt definieren:

```
Dim _Verwendungszweck As String
Public Property Verwendungszweck As String _
        Implements IEPCQRCode.Verwendungszweck
    Get
        Return _Verwendungszweck
    End Get
    Set(value As String)
        _Verwendungszweck = value
    End Set
End Property
```