Kapitel 5
VBA – Grundlagen

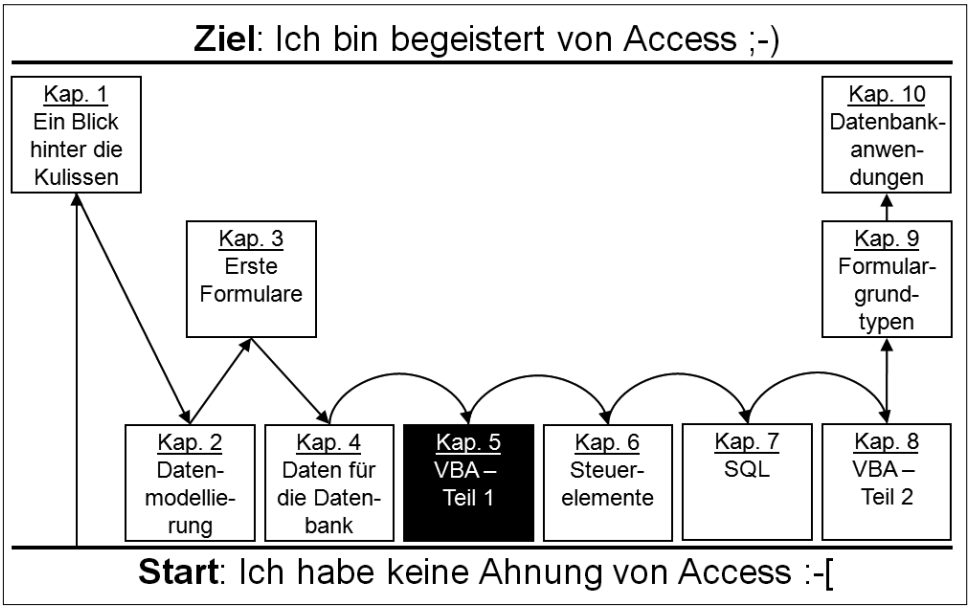

Abbildung 5.1: Das Kapitel 5, »VBA – Teil 1«.

In diesem Kapitel

... versuche ich, Ihnen das Allernotwendigste zum Thema »Programmieren« zu erläutern. Ich gehe dabei nur so weit, dass Sie einfache, unbedingt notwendige Programmieraktionen selbstständig ausführen können. Dabei geht es zunächst nur um ganz allgemeine Programmierbefehle, die es in jeder Programmiersprache gibt, die also noch nichts mit einer Datenbank zu tun haben. Das wird Gegenstand eines späteren Kapitels sein.

Ach ja: »VBA« heißt übrigens »Visual Basic for Applications«. Das ist eine Programmiersprache, die Sie in allen Office-Produkten verwenden können – also nicht nur in Access, sondern auch in Word, PowerPoint und Excel.

Programmieren – muss das sein?

Die Antwort auf die in der Überschrift gestellte Frage lautet: Ja! Man kann Bücher über Word und PowerPoint schreiben, ohne auf Programmierung einzugehen. Man kann sogar hervorragende Bücher über Excel schreiben, ohne auf Programmierung einzugehen. Man kann aber – das ist jedenfalls meine Meinung – kein Buch über Access schreiben, ohne auf Programmierung einzugehen. Und das ist sicherlich auch ein Grund dafür, dass Access solch ein Nischendasein in der Office-Suite fristet: Die meisten Benutzer möchten mit Programmierung nichts zu tun haben.

Warum eigentlich? Weil es so schwer ist? Weil es so eine geheimnisvolle Kunst der »Hacker« ist? Vielleicht darum: Programmieren zwingt zum absolut logischen und folgerichtigen Denken. Denn – wie Sie ja sicher schon öfter gehört haben – der Computer ist eigentlich dumm. Er macht immer nur genau das, was ihm gesagt wird. Von wem? Vom Programmierer!

Dafür schreibt der Programmierer ein Programm in einer Programmiersprache. Das Programm enthält Dutzende, Hunderte oder gar Tausende Zeilen mit Befehlen, die der Computer nacheinander ausführen soll. Dafür muss der Programmierer das Problem, das von dem Programm bearbeitet werden soll, vorher analysiert haben und sich einen Lösungsweg ausgedacht haben.

Habe ich »Problem« gesagt? Das klingt schon wieder so kompliziert. Aber meist sind es ganz banale Dinge, deren Erledigung der Programmierer mithilfe eines Programms organisieren muss.

Ein Beispiel

Obwohl dies ein Buch über Access ist, erläutere ich in diesem Abschnitt den Grundgedanken des Programmierens an einem Excel-Beispiel, weil ich denke, dass Excel Ihnen im Moment vielleicht noch vertrauter ist als Access.

Das Beispiel: Die Anzahl der Namen in einer Kundenliste soll gezählt werden. Für Sie als Mensch gar kein Problem: Sie schauen mit den Augen auf die erste Zeile der Liste, beginnen gedanklich mit »1«, schauen auf die nächste Zeile, denken »2« usw. Sie hören auf zu zählen, wenn Sie mit den Augen auf eine leere Zeile treffen.

Der Computer hat (noch?) keine Augen und kann (noch?) nicht denken. Also muss der Programmierer ihm mithilfe eines Programms Folgendes beibringen:

- Reserviere Speicherplatz für das Zählen der Zeilen.
- Speichere dort erst mal eine »1« ab.
- Merke dir diese Stelle im Programm.
- Versuche, die nächste Zeile in der Liste zu lesen.
- Wenn dort etwas steht, erhöhe die Zahl in dem Speicherplatz um eins und gehe im Programm zurück an die Stelle, die du dir gemerkt hast.
- Wenn dort nichts steht, bist du fertig.

Das meinte ich weiter vorn mit dem Zwang zum logischen Denken: Selbst banalste Dinge, die wir Menschen völlig unbewusst machen, müssen dem Computer Schritt für Schritt wie einem Kleinkind gesagt werden. Und das Nachdenken darüber, welche Schritte in welcher Reihenfolge gemacht werden müssen, nennt man »Programmieren«. Dafür benötigen Sie – ebenso wie für das Sprechen mit dem Kleinkind (»Jetzt aber schön heia machen!«) – eine spezielle Sprache, die der Computer versteht. In der Sprache VBA sieht die obige Schrittfolge in Excel dann so aus:

```
1   Dim intKundenanzahl As Integer
2   intKundenanzahl = 1
3   Do While Cells(intKundenanzahl, 2) <> ""
4         intKundenanzahl = intKundenanzahl + 1
5   Loop
6   intKundenanzahl = intKundenanzahl - 1
```

In Zeile 1 wird ein Speicherplatz (= eine »Variable«) für das Zählen der Zeilen reserviert (*Dim* = Dimension, *Integer* = ganzzahlig). Dort wird zunächst eine »1« abgelegt (Zeile 2). Die Programmzeile 3 besagt: Solange die Excel-Zelle in der Zeile *intKundenanzahl* und der Spalte 2 nicht leer ist (<> ""), führe alle Befehle zwischen *Do While* und *Loop* aus. Dort steht in Zeile 4 nur ein einziger Befehl, der die Kundenanzahl um eins erhöht. Wenn die erste leere Excel-Zelle gefunden wird, bricht die *While*-Schleife ab, und der darauffolgende Befehl in Zeile 6 wird ausgeführt: Die Kundenanzahl wird um eins reduziert, weil mit dem obigen Algorithmus immer ein Kunde zu viel gezählt wird, denn das Programm »merkt« erst in der ersten Leerzeile nach dem Ende der Liste, dass die Liste zu Ende ist.

»Hackermentalität«

Wie gesagt – beim Programmieren müssen Sie eine Problemlösung bis herunter auf die elementarsten Einzelschritte durchdenken und aufschreiben: »reserviere einen Speicherplatz«, »addiere eine Eins«, »prüfe, ob der Inhalt einer Zelle leer ist«. Das macht das Programmieren aus der Sicht vieler Menschen so mühsam und unakzeptabel. Für andere wiederum ist es die eigentlich Hohe Schule der Computernutzung – sie wollen nicht nur auf bunten Bildchen herumklicken, sondern dem Computer mithilfe eines selbst geschriebenen Programms ihren Willen aufzwingen, bis er endlich das macht, was sie sich ausgedacht haben! Es ist immer wieder ein erhebender Augenblick, wenn ein Programm endlich funktioniert!

Sie sind dann nicht mehr darauf angewiesen, ein Programm ausschließlich so zu benutzen, wie Sie es bekommen haben. Sie können das Aussehen der Fenster und die Funktionen nach eigenem Ermessen verändern und das Programm ganz Ihren Wünschen anpassen! Sollte das Programm Fehler machen, können Sie diese selbst beseitigen und müssen nicht auf ein Update warten oder sich ein anderes Programm besorgen!

Wenn Sie diese »Hackermentalität« in keiner Weise nachempfinden können, haben Sie dieses Buch eventuell vergebens gekauft. Aber treffen Sie bitte jetzt noch keine Entscheidung und werfen Sie das Buch noch nicht in die Ecke – ich will erst versuchen, Ihnen das Programmieren in VBA ein wenig schmackhaft zu machen.

Fehler finden und korrigieren

So frustrierend es für den Anfänger klingen mag – aber jeder, der schon mal programmiert hat, weiß:

Programmieren heißt: Fehler machen, finden und korrigieren!

Ein Programmierfehler ist schnell gemacht: Aus »End« wird »Endr«, weil der Fingernagel zu lang ist und beim Druck auf »d« das »r« mit erwischt. Oder: Sie denken, die Variable heißt »adresse« – sie muss aber »anschrift« heißen. Viele Funktionen haben eine ganze Reihe von Parametern, von denen man schon mal einen vergessen kann. Meist ist auch die Reihenfolge von Befehlen entscheidend. Und so weiter, und so weiter! Es gibt zahllose Gelegenheiten, Programmierfehler zu machen.

Man kann sogar absichtlich Fehler machen – und zwar so: Angenommen, es gibt zwei oder drei mögliche Arten, einen Programmierbefehl zu schreiben und Sie haben gerade vergessen, welche davon die richtige ist. Dann dauert es viel zu lange, erst irgendwo nachzuschlagen. Probieren Sie einfach die verschiedenen Möglichkeiten nacheinander aus, und Sie werden schon merken, welche die richtige ist. Bei den falschen erhalten Sie nämlich eine Fehlermeldung von Ihrem Computer!

Es ist nun schwierig, etwas über Fehler beim Programmieren zu schreiben, wenn der Leser noch gar nicht programmieren kann. Andererseits können Sie das Programmieren nur lernen, wenn Sie es tun – und dabei werden Sie unweigerlich Fehler machen und müssen dann wissen, was zu tun ist. Da haben wir also das übliche »Henne-Ei-Problem«!

Ich will versuchen, dieses Dilemma zu lösen, indem ich Ihnen zunächst ohne VBA-Vorkenntnisse erläutere, welche Arten von Programmierfehlern es gibt, wie Sie sie finden und was Sie dagegen tun können. Dazu benutzen wir den VBA-Code aus der Beispielanwendung *Verein*.

Sie finden die Datei *Verein.accdb* im Internet (Adresse in der Einleitung, dort im Ordner *\Kap01*). Übrigens finden Sie Anhang A (»Wichtige Standardaktionen durchführen«) auch als PDF-Datei im Ordner *\KapA*, die ich Ihnen zum Ausdrucken als Arbeitshilfe sehr empfehle.

Das VBA-Fenster

Um die folgenden Erläuterungen zu den verschiedenen Arten von Programmierfehlern direkt am Computer nachvollziehen zu können, öffnen Sie bitte die Beispielanwendung *Verein* und drücken die Tastenkombination Alt+F11. Daraufhin öffnet sich das VBA-Fenster (Abbildung 5.2).

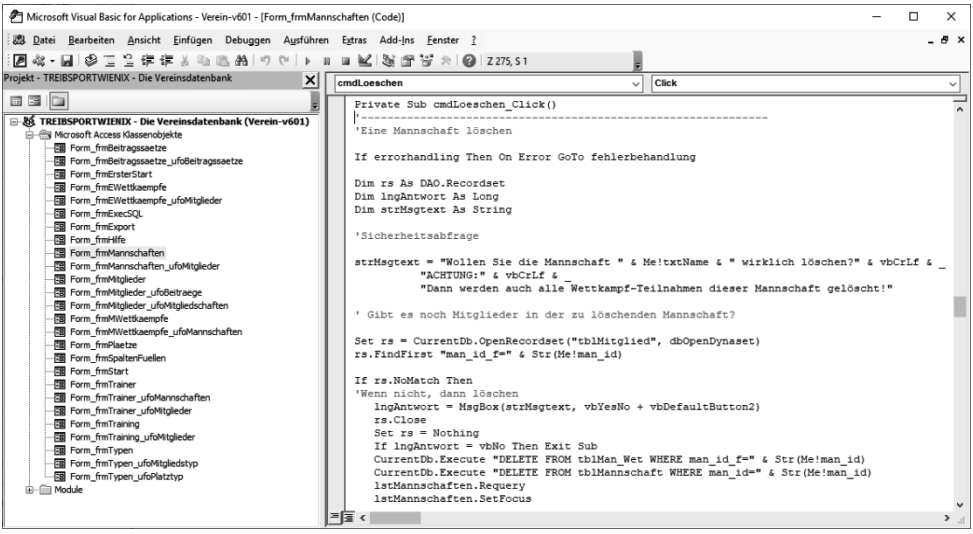

Abbildung 5.2: Die Tastenkombination Alt + F11 *öffnet das VBA-Fenster.*

Sollte das VBA-Fenster bei Ihnen nicht genau so aussehen wie in Abbildung 5.2, machen Sie bitte einen Doppelklick auf *frmMannschaften* in der linken Fensterhälfte und scrollen in der rechten Fensterhälfte ungefähr bis zur Mitte.

Eventuell ist der linke schmale Teil des Fensters bei Ihnen noch einmal unterteilt. Den unteren Teil davon (*Eigenschaften*) können Sie durch einen Klick auf die *Schließen*-Schaltfläche erst einmal schließen. Sollten Sie den linken Teil des VBA-Fensters (den Projekt-Explorer) aus Versehen geschlossen haben, können Sie ihn mit *Ansicht/Projekt-Explorer* wieder öffnen.

Bitte lassen Sie sich jetzt nicht schocken von dem unverständlichen »Chinesisch rückwärts« in der rechten Fensterhälfte. Dort sehen Sie nämlich den Text von VBA-Programmen (die Sie bald selbst schreiben werden!). Diese Programme werden ausgeführt, wenn

- in einem bestimmten Formular (z. B. *frmMannschaften*)
- auf einem bestimmten Objekt (z. B. Schaltfläche *cmdLoeschen*)
- ein bestimmtes Ereignis stattfindet (z. B. *Click*).

In der linken Fensterhälfte sehen Sie die Liste der Formulare, die Sie schon aus vorangegangenen Kapiteln kennen. Nach einem Doppelklick auf einen Formularnamen erscheinen in der rechten Fensterhälfte die zu diesem Formular gehörigen VBA-Programme. Am Namen des jeweiligen Programms (das ist die Zeile, die mit *Private Sub* beginnt) erkennen Sie, welches Ereignis (meistens ein Mausklick – also *Click*) auf welchem Objekt des Formulars (häufig Schaltflächen) den Start des Programms auslöst.

Man nennt das »ereignisorientierte Programmierung«, d. h., VBA »lauert« auf das Stattfinden von Ereignissen (Mausbewegungen, Mausklicks, Tastendrücken) und reagiert darauf mit dem Start von bestimmten Programmen. Sie können als Programmierer also entscheiden, was passieren soll, wenn der Benutzer eine bestimmte Aktion an einer bestimmten Stelle in einem bestimmten Fenster ausführt – oder anders ausgedrückt: wenn er dort ein Ereignis auslöst.

Die folgenden Ausführungen zu Programmierfehlern beziehen sich alle auf das Programm *Private Sub cmdLoeschen_Click()*. Wenn Ihr VBA-Fenster also so aussieht, wie in Abbildung 5.2 dargestellt, können Sie die von mir diskutierten Fehler dort einbauen und die einzelnen Aktionen zum Auffinden und Korrigieren der Fehler nachvollziehen.

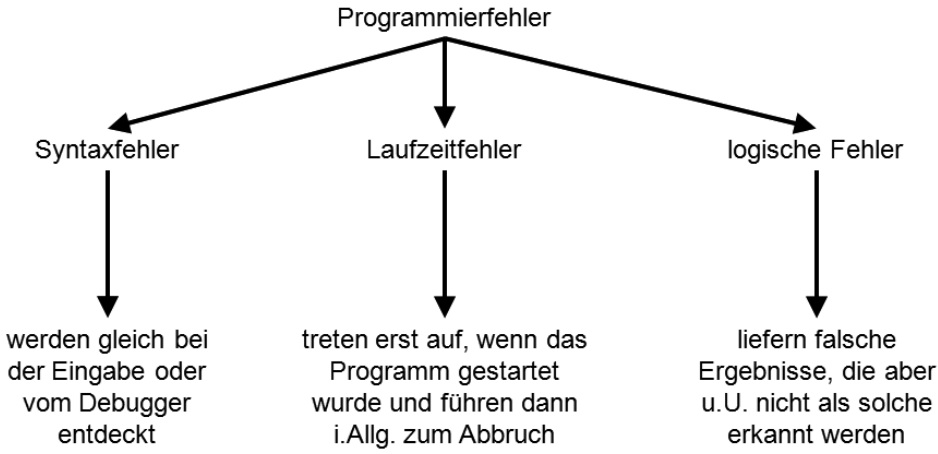

Abbildung 5.3: Die logischen Fehler sind am gemeinsten!

Syntaxfehler

Beim Programmieren benutzen Sie eine Programmiersprache, die – wie jede andere Sprache auch – bestimmten Regeln gehorcht. Es gibt bestimmte Wörter, die in einer wohldefinierten Weise zu schreiben sind und die man auch in bestimmter Weise aneinanderreihen muss, damit sie alle zusammen einen Sinn ergeben. Machen Sie dabei etwas falsch, spricht man von einem Syntaxfehler. Das ist genau so, als würden Sie im Englischen »Leike you windos?« statt »Do you like Windows?« schreiben.

Ihr Computer reagiert unmittelbar nach der Eingabe des falschen Worts mit einer Fehlermeldung (Abbildung 5.4). Die Zeile, in der der Fehler gefunden wurde, wird rot dargestellt, und die Stelle innerhalb der Zeile, an der der Fehler vermutet wird, wird blau hinterlegt.

Die blau hinterlegte Stelle muss nicht mit dem Fehler identisch sein; es ist nur die Stelle, an der VBA den Fehler bemerkt hat!

Im Beispiel in Abbildung 5.4 wurde das VBA-Schlüsselwort *Dim* falsch geschrieben. (Es bedeutet *Dimension* und dient der Definition von Variablen.) VBA merkt aber erst beim Wort *As*, dass da etwas faul ist, und markiert dieses Wort. Sie können sich daher nicht darauf verlassen, dass die blau markierte Stelle den Fehler enthält. Sie müssen immer in der ganzen Zeile suchen!

Auch auf die bei *Erwartet* von VBA vorgeschlagene Lösung des Problems können Sie sich nicht unbedingt verlassen, da die Ursache manchmal falsch lokalisiert wird.

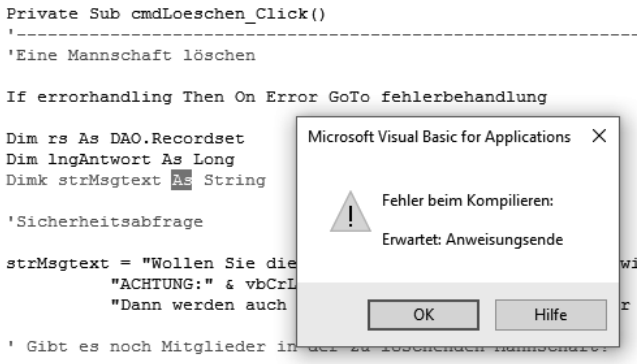

Abbildung 5.4: Das VBA-Schlüsselwort Dim *wurde falsch geschrieben.*

Im Gegensatz zu einer Umgangssprache wie Englisch oder Deutsch gibt es aber in einer Programmiersprache nicht nur die fest vordefinierten Wörter (»Schlüsselwörter«). Sie können als Programmierer eigene Wörter hinzudefinieren – z.B. *lngAntwort* oder *strMsgText*. Dafür gibt es einen speziellen Programmierbefehl – nämlich das in Abbildung 5.4 falsch geschriebene Wort *Dim*. In Abbildung 5.5 wird auf diese Weise das Wort *strMsgtext* definiert. Dieses Wort können Sie dann im darauffolgenden Programmtext benutzen.

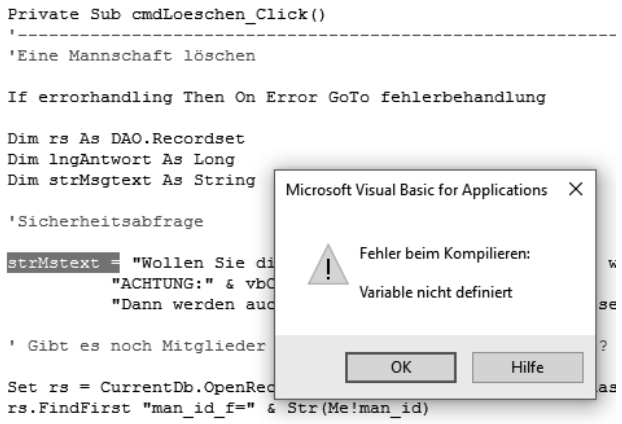

Abbildung 5.5: Der selbst definierte Variablenname strMsgtext *wurde falsch geschrieben.*

Wenn Sie es aber nicht so schreiben, wie Sie es vorher selbst definiert haben (siehe Abbildung 5.5), entsteht wieder ein Syntaxfehler. Dieser wird aber im Unterschied zu vorher nicht sofort beim Eintippen bemerkt, sondern erst nach dem Start des Programms. Damit müsste es eigentlich ein Laufzeitfehler sein (siehe Abbildung 5.3 und den nächsten Abschnitt). Aber: Sie können diese Art von Fehlern auch entdecken, ohne das Programm zu starten – und zwar mithilfe des sogenannten Debuggers. Das englische Wort »bug« heißt auf Deutsch einerseits »Laus«, andererseits aber auch »Programmierfehler«. Mit »debug« meint man also entweder »entlausen« oder »Programmierfehler suchen und beseitigen«.

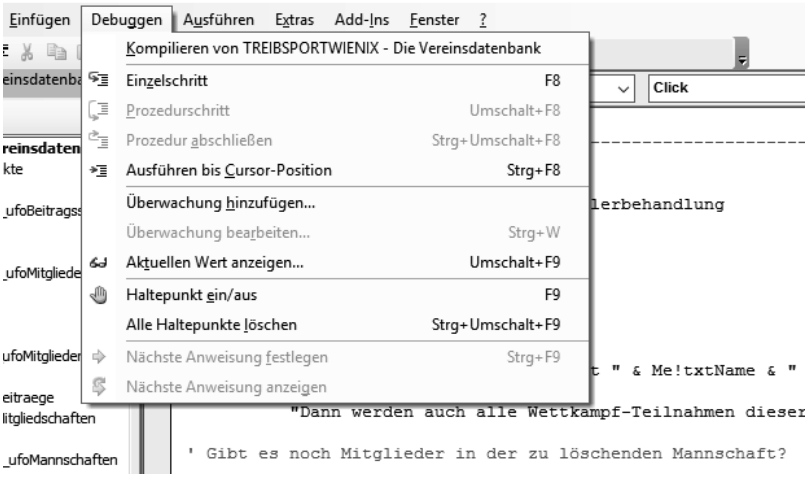

Abbildung 5.6: So »entlausen« Sie Ihr VBA-Programm.

Dafür gibt es dann auch ein entsprechendes Menü im VBA-Fenster (Abbildung 5.6). Mit einem Klick auf *Kompilieren von TREIBSPORTWIENIX – Die Vereinsdatenbank* starten Sie den Debugger. Er überprüft Ihr Programm auf syntaktische Richtigkeit und zeigt gegebenenfalls eine Fehlermeldung wie die in Abbildung 5.5.

Der Debugger hört auf zu suchen, nachdem er den ersten Fehler gefunden hat! Wenn Sie diesen Fehler behoben haben, sollten Sie den Debugger also noch einmal starten, um gegebenenfalls den nächsten Fehler zu finden. Das müssen Sie so lange wiederholen, bis der Debugger keine Syntaxfehler mehr meldet.

Abschließend noch eine Bemerkung zur Ehrenrettung von VBA. Ich hatte weiter oben gesagt, dass VBA in der Zeile *Dimk strMsgtext As String* erst beim Wort *As* merkt, dass etwas faul ist. Da haben Sie vielleicht gedacht: »Na toll! Warum sagt man mir nicht gleich, dass ich *Dim* falsch geschrieben habe?«

Jaaa nun, es könnte ja sein, dass *Dimk* ein von Ihnen selbst definiertes Wort ist! Das kann VBA natürlich nicht wissen! Wenn es aber ein selbst definiertes Wort wäre, dürfte es nicht in der Zusammenstellung *Dimk strMsgtext As String* benutzt werden. Und das merkt VBA eben genau an der Stelle, an der der von Ihnen geschriebene Text die Syntaxregeln der Programmiersprache verletzt.

Die Fehlermeldung in Abbildung 5.4 müsste also eigentlich so gelesen werden: »Wenn *Dimk* und *strMsgtext* von Ihnen selbst definierte Wörter sind, müsste die Anweisung hier (also bei *As*) zu Ende sein. Wenn nicht, liegt der Fehler woanders.«

Laufzeitfehler

Wenn Ihr Programm syntaktisch richtig ist, muss es noch lange nicht so funktionieren, wie Sie sich das gedacht haben. Das ist ähnlich wie in einer Umgangssprache: Der Satz »Would you please give me the fork?« ist syntaktisch richtig, und Sie merken erst, dass Sie eigentlich

ein Messer haben wollten, wenn Sie den Satz als Aufforderung an Ihr Gegenüber richten und es Ihnen die Gabel reicht.

Der Fehler tritt also erst bei Ausführung des Programms – zur Laufzeit – in Erscheinung. Er macht sich dann mit einem Warnfenster, wie in Abbildung 5.7 dargestellt, bemerkbar. Gleichzeitig wird die Ausführung des Programms angehalten.

Sie haben dann die Möglichkeit,

- auf *Beenden* zu klicken und damit die Programmausführung abzubrechen oder
- auf *Debuggen* zu klicken. Damit wird die Programmausführung auch abgebrochen; zusätzlich öffnet sich aber das VBA-Fenster, und die Stelle, an der sich der Fehler vermutlich befindet, wird farbig markiert (siehe Abbildung 5.8).

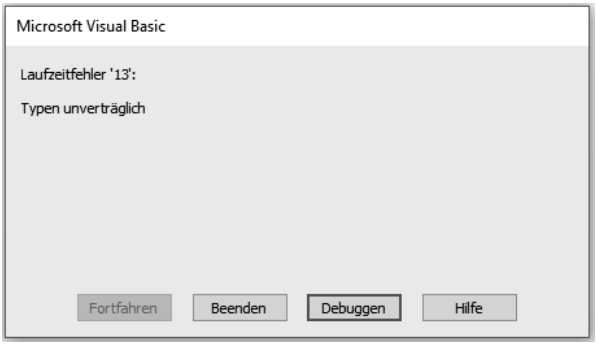

Abbildung 5.7: Nach dem Start des Programms wird ein Laufzeitfehler gemeldet.

Ich habe vorsichtshalber »vermutlich« geschrieben, denn auch hier gilt wieder das weiter oben bereits Gesagte: Eventuell merkt VBA erst eine oder sogar mehrere Zeilen nach der Zeile mit dem Fehler, dass etwas faul ist. Die farbig markierte Zeile ist also nur ein Hinweis darauf, dass sich der Fehler in dieser Zeile oder in einer Zeile davor befindet. Dadurch kann das Finden des Fehlers zu einer ziemlich kniffligen Aufgabe werden. Dazu jedoch später mehr!

In diesem Fall befindet sich der Fehler allerdings direkt in der farbig markierten Zeile: Mittendrin fehlen zwei Kommata.

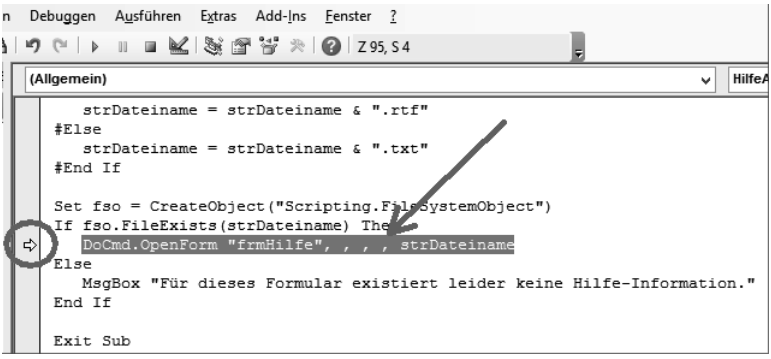

Abbildung 5.8: Die Stelle mit der (möglichen!) Ursache für den Laufzeitfehler wird markiert.

Sie können den in Abbildung 5.8 dargestellten Laufzeitfehler nachvollziehen, indem Sie in der farbig markierten Zeile zwei Kommata löschen, zu Access wechseln, dort das Formular *frmMannschaften* öffnen und auf die Schaltfläche *Hilfe* (rechts oben mit dem Fragezeichen) klicken. Daraufhin erscheint die Meldung eines Laufzeitfehlers (siehe Abbildung 5.7). Die zu bearbeitende Codezeile befindet sich übrigens in *VBA/Module/Hilfsprozeduren* in der Prozedur *Public Sub HilfeAnzeigen(strObjektname As String)*.

Um die in Abbildung 5.8 farbig markierte Programmzeile zu finden, benutzen Sie am besten die Suchfunktion (Aufruf mit Strg+F, *Suchen nach: frmHilfe, Suchen in: aktuellem Projekt)*.

Wenn Ihr VBA-Fenster so aussieht, wie in Abbildung 5.8 dargestellt, können Sie in Access nichts mehr tun, was die Ausführung von VBA-Code erfordert, d.h., Sie können z.B. keine Schaltflächen auf Ihren Formularen mehr anklicken!

Um in Access wieder ungehindert arbeiten zu können, müssen Sie den Debugger anhalten, indem Sie im VBA-Fenster (nicht in Access!) auf die Schaltfläche *Zurücksetzen* klicken (Abbildung 5.9).

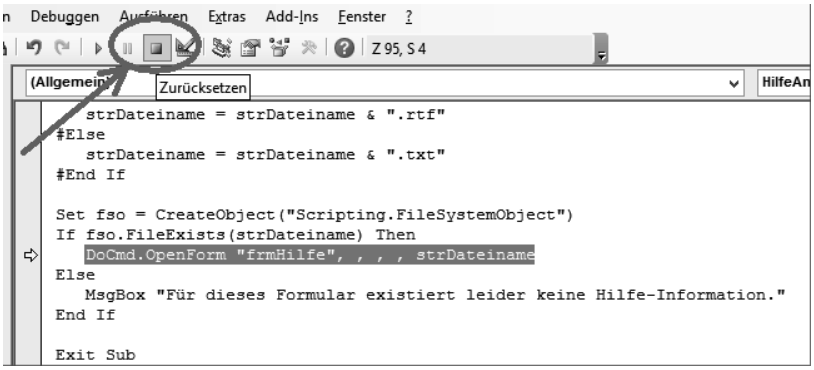

Abbildung 5.9: Durch Anklicken der Schaltfläche Zurücksetzen *wird der Debugger angehalten.*

Wenn auf dem Bildschirm ein Fenster wie das in Abbildung 5.7 erscheint, ist das für den Benutzer der Datenbankanwendung sehr unerfreulich. Eine Meldung der Art »Laufzeitfehler 13: Typen unverträglich« ist unverständlich, und der Benutzer weiß nicht, was er tun soll. Letzten Endes ist er unzufrieden mit der gesamten Anwendung, und das fällt auf Sie als Entwickler zurück.

Nun werden Sie vielleicht sagen: »Ich entwickle ja für mich selbst!« Trotzdem bleibt eine solche Fehlermeldung unerfreulich, denn Sie müssen sich nun erst mal um die Fehlerbehebung statt um Ihre eigentliche Arbeit kümmern. Deshalb ist es außerordentlich wichtig, schon bei der Entwicklung der Datenbankanwendung darauf zu achten, dass Laufzeitfehler möglichst gar nicht erst auftreten können. Und wenn es doch einmal passiert, sollte die Anwendung wenigstens nicht »abstürzen«.

Was Sie als Programmierer dafür tun können, erläutere ich weiter unten in diesem Kapitel im Abschnitt »Laufzeitfehler verhindern«.

Logische Fehler

Das sind die gemeinsten Fehler! Scheinbar ist alles in Ordnung: Der Debugger meldet keine Syntaxfehler, und es treten keine Laufzeitfehler auf. Alles funktioniert!

Alles funktioniert? Wenn damit gemeint ist, dass sich auf Wunsch Fenster öffnen, dass in den Listen Zahlen stehen, dass Sie in Textfeldern Daten eingeben können, dass die Schaltflächen Reaktionen zeigen – dann ja. Aber das ist nur das technische Funktionieren. Es müssen sich auch die **richtigen** Fenster öffnen. Es müssen in den Listen auch die **richtigen** Zahlen stehen. Die Schaltflächen müssen auch die **richtigen** Reaktionen zeigen.

Ein Beispiel: Angenommen, Sie hätten sich, wie in Abbildung 5.10 dargestellt, vertippt.

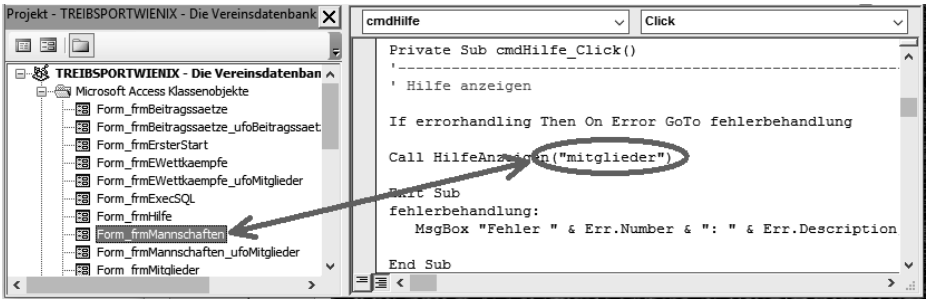

Abbildung 5.10: Logischer Fehler durch eine syntaktisch richtige, aber logisch falsche Programmzeile.

Das VBA-Programm, das nach einem Mausklick auf die Schaltfläche *cmdHilfe* im Formular *frmMannschaften* startet, öffnet also statt der Hilfe zu diesem Formular den Hilfetext zum Formular *frmMitglieder*. Syntaktisch ist alles richtig, und es tritt natürlich auch kein Laufzeitfehler auf – aber das falsche Fenster öffnet sich. In diesem Fall merken Sie das natürlich sofort. Es gibt aber auch ganz gemeine Fälle, in denen der Fehler nur schwer zu entdecken ist.

Nehmen wir mal an, Sie hätten sich bei der Mehrwertsteuer vertippt und würden mit 16 % rechnen (schön wär's ja ...). Das fällt sicher nicht sofort auf.

Oder: Sie haben in einer Datenbanktabelle Telefon- und Faxnummern von Kunden oder Bekannten gespeichert und vertauschen diese beim Drucken einer Kontaktliste.

Oder: Sie haben in einer Berechnung plus, minus, mal und geteilt durch so »geschickt« vertauscht, dass falsche Ergebnisse herauskommen, die aber zufällig die richtige Größenordnung haben. Es kommt also statt 120,45 zwar nicht 538.298,76 heraus – aber vielleicht 125,87.

Oder: Sie zählen in einer Liste von Einträgen immer einen Eintrag zu viel (siehe Codebeispiel ganz am Anfang dieses Kapitels).

Oder, oder, oder ...

In allen genannten Fällen haben Sie die richtigen Daten in Ihren Datenbanktabellen, und Ihre Anwendung zeigt auch rein technisch gesehen die richtige Reaktion. Es tritt kein auf den ersten Blick sichtbarer Fehler auf.

Was kann man dagegen tun? Da hilft eigentlich nur:

- Definieren Sie einen oder mehrere Sätze von Testdaten.
- Lassen Sie Ihre Datenbankanwendung aus diesen Testdaten Ergebnisse errechnen und/oder zusammenstellen.

- Erstellen oder berechnen Sie die Ergebnisse parallel dazu ohne Ihre Datenbankanwendung – zur Not mit Bleistift und Papier.
- Vergleichen Sie die von Hand erzielten Ergebnisse mit den Ergebnissen der Datenbankanwendung.

Wichtig ist dabei, **alle** nur denkbaren Fälle für die Testdaten zu überprüfen:

- Kann es ganz große oder ganz kleine Zahlen geben?
- Kann es negative Zahlen oder Nullen geben?
- Kann es keine, sehr wenige oder sehr viele Daten geben?
- Können die Daten besondere Zeichen enthalten? (z. B. Klammern oder Schrägstriche in Telefonnummern, Tausenderpunkte in Zahlen, Länderkennzeichen in Postleitzahlen ...)
- Können sich scheinbar feste Werte doch ändern? (z. B. Mehrwertsteuersatz)
- Welche Maßeinheiten können Daten haben? (Meter oder Fuß, Kilogramm oder Pfund, Liter oder Gallone ...)
- Bestehen Abhängigkeiten zwischen den Daten? (z. B. Beginn vor Ende, netto kleiner als brutto)

Die Entwicklungsumgebung

So, jetzt wollen wir also loslegen mit dem Programmieren. Da stelle ich mir gleich ganz erschrocken die Frage: VBA auf einigen Dutzend Seiten? Geht das überhaupt? Nein, das geht nicht! Zu dem Thema werden sehr dicke Bücher geschrieben, und in denen steht noch immer nicht alles drin, was man dazu sagen könnte. Dieses Buch heißt aber »Keine Angst vor Microsoft Access«, und in diesem Sinne möchte ich Ihnen anhand von Beispielen erläutern, wie VBA funktioniert. Wenn Sie ernsthaft auf diesem Gebiet weitermachen möchten, brauchen Sie ein oder mehrere weitere Bücher mit Details.

Sie finden die Übungsdatei *VBAlernen.accdb* im Internet (Adresse in der Einleitung, dort im Ordner *\Kap05*).

Es soll aber nicht so sein, dass Sie nach der Lektüre dieses Buchs zwar etwas mehr wissen und vielleicht auch den Mut gefasst haben, sich vertieft mit VBA zu beschäftigen – aber immer noch hilflos sind, wenn es um das Schreiben von VBA-Programmen geht. Nein, Sie sollen schon in gewissen Grenzen arbeitsfähig sein!

Dazu müssen wir uns als Erstes die Werkzeuge ansehen. Die nennen sich in diesem Fall »Entwicklungsumgebung« – ein etwas sperriges Wort, das eine Software beschreibt, die Sie in vielerlei Hinsicht beim Programmieren unterstützt.

So wie Sie mit Word einen Brief schreiben, so wollen Sie auch das VBA-Programm schreiben. Dafür gibt es in der Entwicklungsumgebung den Editor.

Wenn Sie Ihr Programm geschrieben haben, wollen Sie es auf Syntaxfehler untersuchen lassen. Dafür gibt es in der Entwicklungsumgebung den Debugger.

Der Editor

Beides haben wir in diesem Kapitel schon benutzt. Trotzdem kommt hier noch einmal eine kleine Wiederholung: Sie beginnen mit dem Start von Access und bearbeiten damit Ihre Tabellen und Formulare. Durch die Tastenkombination Alt+F11 starten Sie parallel dazu VBA. Es öffnet sich das VBA-Fenster (Abbildung 5.2), in dem Sie auf der rechten Seite den Editor zum Schreiben der Programme sehen. Diese Programme gehören jeweils zu einem bestimmten Formular. Deshalb sehen Sie auf der linken Seite des VBA-Fensters die Liste der Formulare. Durch einen Doppelklick auf einen Formularnamen wechselt der Inhalt des Editors und zeigt Ihnen die Programme an, die zu dem angeklickten Formular gehören.

Der Editor selbst ist im Grunde genommen ein einfaches Textverarbeitungsprogramm. Wenn Sie über *Start/Programme/Zubehör/Editor* schon mal Notepad gestartet haben, wissen Sie, was ich meine. Mit dem Editor können Sie unformatierten Text schreiben, d. h., es gibt im Unterschied zu Word kein *fett*, *kursiv*, *rechtsbündig* usw. Aber das brauchen Sie auch gar nicht. Beim Programmieren kommt es nur auf eine saubere Logik an – weniger aufs schöne Aussehen!

Das heißt, ein ganz klein wenig »schön« macht der Editor den von Ihnen geschriebenen Programmtext schon: Er färbt Schlüsselwörter (also von der Programmiersprache definierte Begriffe) blau und Kommentare (die mit einem Apostroph beginnen) grün. Alles andere bleibt schwarz.

Beim Schreiben Ihrer Programme sollten Sie sich unbedingt Folgendes angewöhnen:

- Alles kleinschreiben!
- In den Namen von Variablen und Programmen keine Umlaute wie ä, ö, ü, kein ß und keine Leerzeichen verwenden!

Warum alles klein? Fast alle Schlüsselwörter enthalten eine Mischung aus Groß- und Kleinbuchstaben (z. B. *If, Then, MsgBox)*. Wenn Sie das Wort kleinschreiben (also *msgbox* statt *MsgBox*), wandelt der Editor nach dem Drücken der Leertaste automatisch die großzuschreibenden Buchstaben um. Tut er das nicht, erkennen Sie sofort, dass Sie sich verschrieben haben.

Der Editor erkennt sofort beim Schreiben Syntaxfehler in Schlüsselwörtern (siehe oben).

Warum sollen Sie keine Umlaute verwenden? Programmiert wird schon seit Urzeiten auf Englisch – und da gibt es nun mal keine Umlaute. Wenn Sie jetzt sagen: »Mir doch egal! Ich schreibe deutsch!«, rate ich Ihnen dringend, trotzdem auf Umlaute zu verzichten. Irgendwann bekommen Sie deswegen Probleme beim Programmieren. Glauben Sie's mir!

Und noch einige Dinge sollten Sie sich angewöhnen:

- Strukturieren Sie Ihren Programmtext durch Einrückungen. Was ich damit meine, sehen Sie sich am besten in meinen Beispieldatenbanken an. Die Programme werden dadurch sehr viel übersichtlicher!
- Versehen Sie Ihre Programme mit möglichst vielen und ausführlichen Kommentaren. Sie glauben gar nicht, wie schnell Sie selbst vergessen haben, wie Ihr eigenes Programm funktioniert! Und was soll da erst jemand anderes sagen, der Ihr Programm vielleicht weiterbearbeiten will oder muss?
- Wenn Sie beim Programmieren bemerken, dass eine bestimmte Lösung so nicht funktioniert, löschen Sie die betreffenden Programmzeilen nicht einfach weg. Lassen Sie sie stehen und verwandeln Sie sie durch vorangestellte Apostrophe in Kommentare. Schreiben Sie außerdem dazu, warum das nicht funktioniert. Sie sparen sich damit später viel

Zeit! Wie oft habe ich schon gedacht: »Das müsste doch auch so gehen ...« – um dann frustriert festzustellen, dass ich in dieser Sackgasse schon mal gewesen bin!

Wenn eine Codezeile einmal zu lang wird und Sie sie auf mehrere Textzeilen verteilen möchten, schreiben Sie am Ende der fortzusetzenden Zeile ein Leerzeichen und einen Unterstrich (aufpassen – das Leerzeichen ist wichtig!).

Aber Achtung – so funktioniert es **nicht**:

```
MsgBox "Bitte geben Sie in dem Feld Enddatum einen Wert ein, sonst kann _
        die Transportzeit nicht berechnet werden."
```

Sie müssen die fortzusetzende Zeile mit einem Operatorzeichen (&, +, -, *, /) oder einem Komma beenden:

```
MsgBox "Bitte geben Sie in dem Feld Enddatum einen Wert ein, sonst " & _
       " kann die Transportzeit nicht berechnet werden."
```

Und noch ein kleiner Service unterscheidet den VBA-Editor von einem Textverarbeitungsprogramm à la Notepad. Er unterstützt Sie bei der Eingabe von VBA-Schlüsselwörtern auf zweierlei Arten:

- **QuickInfo** Wenn Sie einen Funktionsnamen wie z. B. *MsgBox* eintippen, erscheint in dem Augenblick, in dem Sie die öffnende Klammer hinter dem Funktionsnamen tippen, ein hellgelbes Informationsfeld direkt unterhalb der Einfügemarke. Es zeigt Ihnen, welche Parameter diese Funktion in welcher Reihenfolge erfordert. Beim Weitertippen wird jeweils der gerade von Ihnen eingetippte Parameter fett dargestellt, sodass Sie immer wissen, wo Sie gerade sind.

  Probieren Sie das bitte gleich einmal aus, indem Sie an einer beliebigen Stelle in einem VBA-Programm *msgbox(* eintippen – einschließlich der öffnenden runden Klammer.

  Wenn Sie die Funktion bereits fertig getippt haben, können Sie die QuickInfo auch über einen Rechtsklick auf den Funktionsnamen aufrufen. In dem sich öffnenden Kontextmenü wählen Sie den Befehl *QuickInfo*.

- **Wort vervollständigen** Beim Eintippen bestimmter VBA-Schlüsselwörter öffnet sich nach den ersten Buchstaben automatisch ein Listenfeld, in dem Ihnen verschiedene Möglichkeiten für die Vervollständigung des Worts angeboten werden.

  Beispiel: Lassen Sie sich einmal im VBA-Fenster (Abbildung 5.12) durch einen Doppelklick auf den Formularnamen *S6_EnabledVisibleLocked* den Programmcode für dieses Formular anzeigen. Dort tippen Sie direkt unter der Zeile, die mit *Private Sub …* beginnt, Folgendes ein: *chkKontrollkaestchen.vi*. Bereits nach dem Punkt erscheint das Listenfeld. Sowie Sie das »v« eintippen, wird die Zeile *ValidationRule* markiert. Nach Eingabe des *i* wird *Visible* markiert. Jetzt brauchen Sie nur noch die ↵-Taste zu drücken, und der Editor ergänzt Ihre Eingabe zu *chkKontrollkaestchen.Visible*.

Zwischen den beiden Fenstern, die Sie jetzt geöffnet haben – eines mit Access und eines mit VBA –, besteht eine Vielzahl von Wechselbeziehungen. Sie werden sich ab jetzt an eine Arbeitsweise gewöhnen müssen, die Ihnen vielleicht neu ist: Sie müssen gleichzeitig in zwei Fenstern mit zwei ganz unterschiedlichen Programmen arbeiten und ständig zwischen die-

sen beiden Fenstern hin- und herwechseln. (Da ist es **sehr** hilfreich, wenn man sich den Luxus erlauben kann, mit zwei Bildschirmen zu arbeiten!)

Der Zusammenhang zwischen Access und VBA entsteht über die Ereignisse, die auf den Objekten der Formulare stattfinden.

Objekte und Ereignisse

Wir arbeiten weiterhin mit der Beispielanwendung *VBAlernen*. Öffnen Sie einmal das Formular *S2_Listenfeld* **in der Entwurfsansicht**, klicken Sie mit der rechten Maustaste auf das Listenfeld und wählen Sie im sich öffnenden Kontextmenü den Befehl *Eigenschaften*. Es erscheint das Eigenschaftenblatt (siehe Abbildung 5.11). Bitte wählen Sie dort die Registerkarte *Ereignis*.

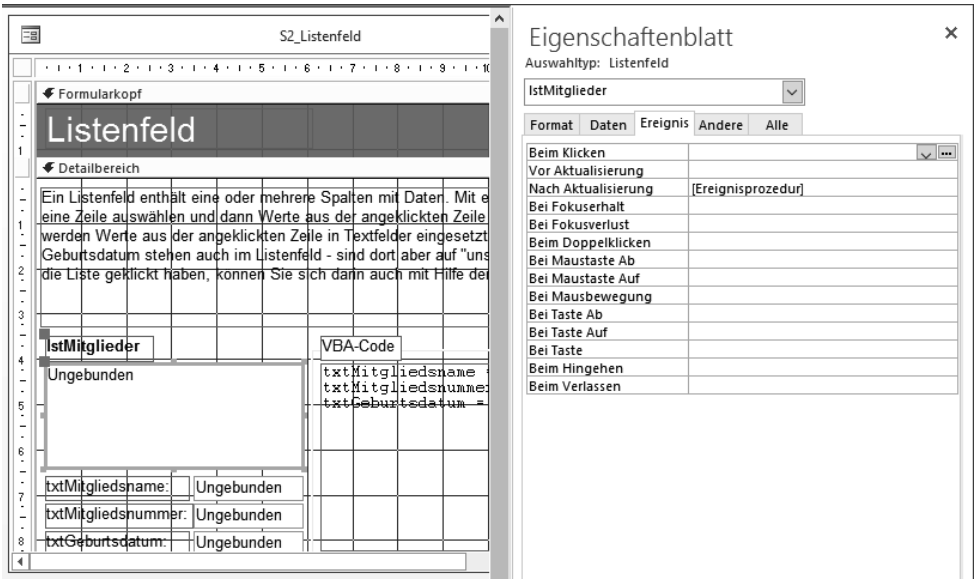

Abbildung 5.11: Auf einem Listenfeld können viele verschiedene Ereignisse stattfinden.

Dort sehen Sie, was auf dem Listenfeld alles passieren kann: Sie können mit der Maus einfach oder doppelt darauf klicken, Sie können die Maus auch nur ohne zu klicken darüber bewegen, Sie können eine Taste auf der Tastatur drücken, während sich die Einfügemarke im Listenfeld befindet, usw. Alles das sind Ereignisse, die auf dem Objekt *lstMitglieder* vom Typ *Listenfeld* stattfinden können.

Und jetzt kommt es: Jedes dieser Ereignisse kann den Start eines VBA-Programms auslösen! Wenn also ein bestimmtes Ereignis auf einem bestimmten Objekt eines Formulars stattfindet, startet ein bestimmtes VBA-Programm. Das bedeutet, Sie können als Programmierer detailliert festlegen, was bei bestimmten Aktionen des Benutzers passieren soll. Klickt er auf eine Schaltfläche, öffnet sich ein neues Formular. Klickt er in ein Listenfeld, werden die Daten des angeklickten Kunden angezeigt. Klickt er auf eine andere Schaltfläche, wird ein neuer Datensatz angelegt usw.

In dem in Abbildung 5.11 dargestellten Fall passiert etwas bei dem Ereignis *Nach Aktualisierung*, denn rechts daneben steht *Ereignisprozedur*. Damit ist ein VBA-Programm gemeint.

Dieses Programm startet, wenn das Ereignis *Nach Aktualisierung* auf dem Listenfeld stattfindet, d. h., wenn der Benutzer einen neuen Eintrag in der Liste gewählt hat – sei es durch Mausklick oder durch Drücken der Taste ↑ oder ↓. Wollte man Letzteres ausschließen, hätte man das Ereignis *Beim Klicken* wählen müssen.

Wenn Sie jetzt wissen wollen, was für ein VBA-Programm denn ausgeführt wird, müssen Sie auf die kleine Schaltfläche mit den drei Punkten am rechten Rand klicken. Achtung: Diese Schaltfläche ist nur sichtbar, wenn die Einfügemarke in der Zeile *Nach Aktualisierung* steht! Daraufhin öffnet sich das VBA-Fenster (Abbildung 5.12) – womit wir wieder bei dem Zusammenhang zwischen Access und VBA sind!

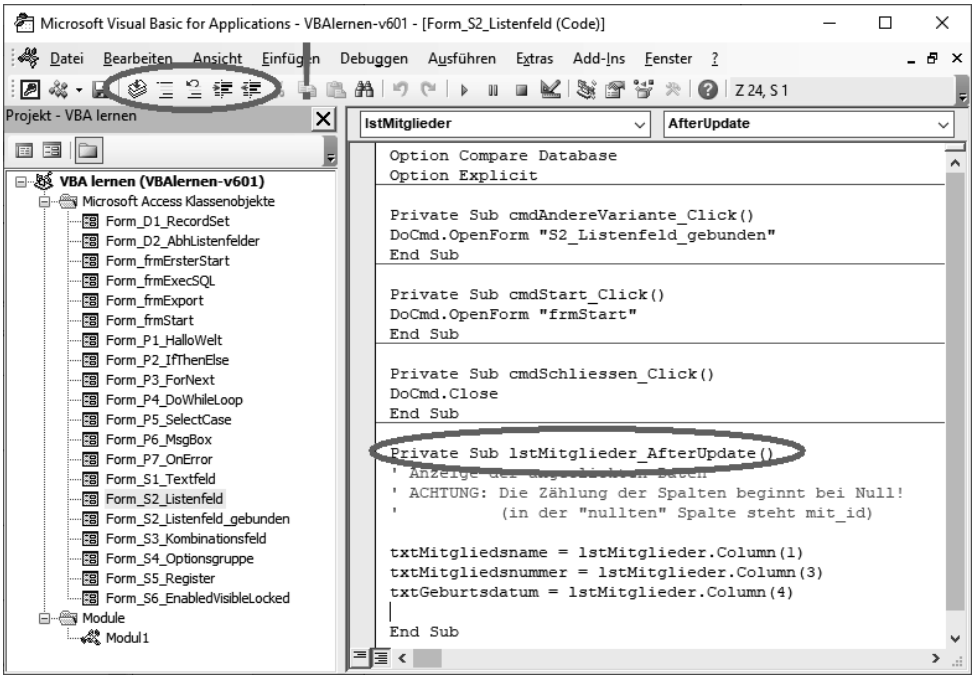

Abbildung 5.12: VBA-Code für das Ereignis AfterUpdate auf dem Objekt lstMitglieder des Formulars Form_S2_Listenfeld.

Das jetzt angezeigte VBA-Programm heißt *lstMitglieder_AfterUpdate*. Das ist ganz wichtig, denn so werden die Programmnamen generell gebildet: zuerst der Name des Objekts, dann ein Unterstrich und dann der Name des Ereignisses.

Dazu gleich wieder ein wichtiger Tipp:

Sollte einmal bei einem bestimmten Ereignis gar nichts passieren oder nicht das, was Sie wollten, vergleichen Sie den Namen des Objekts mit dem Namen des Programms, das ausgeführt werden sollte. Häufig ist es nämlich so, dass man **nur einen von beiden** geändert hat, und dann startet die Prozedur nicht mehr, weil der Name des Objekts nicht mit dem Namen der Prozedur übereinstimmt. (Den Namen eines Objekts finden Sie im Eigenschaftenblatt auf der Registerkarte *Andere*.)

Ganz oben im VBA-Fenster sehen Sie zwei Kombinationsfelder, in denen in Abbildung 5.12 *lstMitglieder* bzw. *AfterUpdate* steht. Wenn Sie diese Kombinationsfelder einmal aufklappen, sehen Sie darin links die Namen aller Objekte des ausgewählten Formulars (in Abbildung 5.12 ist das *S2_Listenfeld*) bzw. rechts die Bezeichnungen aller Ereignisse, die auf dem ausgewählten Objekt möglich sind. Das ist eine weitere Hilfe, die der Editor Ihnen bietet: Wollen Sie ein bestimmtes Programm schreiben, wählen Sie im linken Kombinationsfeld das Objekt und rechts das Ereignis auf diesem Objekt. Anschließend bildet der Editor Ihnen daraus die Titelzeile des entsprechenden Programms. Wenn ein solches Programm schon existiert, wandert die Einfügemarke an diese Stelle.

Abschließend noch einmal der wichtige Hinweis auf den Zusammenhang zwischen Access und VBA:

In Access entwickeln Sie Formulare. Auf den Formularen gibt es Objekte, auf denen Maus- und Tastaturereignisse stattfinden können. Daraufhin kann ein VBA-Programm starten, dessen Name sich aus dem Namen des Objekts und dem Namen des Ereignisses zusammensetzt.

Bitte sehen Sie sich zur Übung einmal den VBA-Code für verschiedene Ereignisse auf verschiedenen Objekten an und schalten Sie dabei jeweils zwischen Access und VBA hin und her.

Der Debugger

Über den Debugger hatte ich weiter oben im Abschnitt über die Programmierfehler schon einiges gesagt. Er ist Teil der VBA-Entwicklungsumgebung und unterstützt Sie beim Finden und Korrigieren von Syntax- und Laufzeitfehlern in Ihrem Programm.

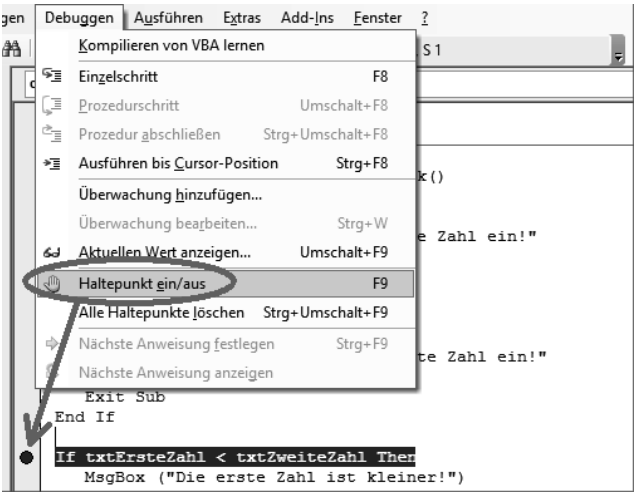

Abbildung 5.13: Mit dem Debugger können Sie Fehler in Ihrem VBA-Programm finden und korrigieren.

Mit dem Debugger können Sie Ihr Programm auch Zeile für Zeile testen (Befehl *Einzelschritt* im Menü *Debuggen*) oder nur bis zu einer bestimmten Stelle laufen lassen – z. B. weil Sie den

Rest dahinter noch nicht ganz fertig haben und erst mal den ersten Teil testen wollen. Dazu müssen Sie an der entsprechenden Stelle einen sogenannten Haltepunkt setzen – entweder über den entsprechenden Befehl im Menü *Debuggen* oder indem Sie einfach mit der Maus in den grauen Streifen links neben dem VBA-Code klicken (in Abbildung 5.13 dunkel hinterlegt mit einem Punkt links daneben). Gelöscht wird der Haltepunkt durch einen erneuten Klick auf dieselbe Stelle.

An dieser Stelle möchte ich noch einmal an meine Ausführungen zu der Frage »Findet der Debugger immer die Stelle mit dem Fehler?« erinnern. Ich hatte weiter oben schon gesagt, dass er das gar nicht immer kann. Sie müssen also immer auch vor der vom Debugger markierten Fehlerstelle suchen. Wenn Sie durch Ansehen des VBA-Codes den Fehler nicht entdecken, hilft nur die schrittweise Programmausführung und genaue Beobachtung: »In welcher Zeile beginnt das Fehlverhalten?«

Ich habe auch schon in ganz verzweifelten Situationen Zeile für Zeile bereits geschriebenen VBA-Code wieder weggelöscht und dabei vor mich hin gemurmelt: »Verd..., es ging doch schon mal!« Plötzlich geht es dann wirklich wieder, und damit hat man die verantwortliche Zeile gefunden!

Symbolleiste anpassen

Bei der Entwicklung von VBA-Prozeduren werden Sie so arbeiten: Code schreiben – kompilieren (d.h. Syntaxfehler suchen) – gegebenenfalls Fehler korrigieren – noch einmal kompilieren – speichern – Wechsel zu Access und Prozedur ausprobieren – Wechsel zu VBA und nicht funktionierenden Code probehalber auskommentieren (nicht löschen, sonst müssen Sie ihn nachher eventuell neu eintippen!) – kompilieren – speichern – usw. – usw.

Sie brauchen also ständig die Editorfunktionen *Speichern*, *Kompilieren*, *Code in Kommentar umwandeln* und *Kommentar in Code umwandeln*. Diese Funktionen sollten Sie sich daher in der Symbolleiste des VBA-Fensters zurechtlegen. Das spart Zeit und Nerven!

Wie das geht, wissen Sie vielleicht schon aus anderen Microsoft Office-Anwendungen: Wählen Sie im VBA-Fenster im Menü *Ansicht* die Option *Symbolleisten/Anpassen/Befehle*. Dann wählen Sie die Kategorie *Bearbeiten* aus und ziehen die Befehle *Block auskommentieren* und *Auskommentierung des Blocks aufheben* per Drag-and-drop in die Symbolleiste rechts neben das Symbol für *Speichern*. Das Gleiche wiederholen Sie noch einmal mit der Kategorie *Debuggen* und dem Befehl *Projekt kompilieren*. Anschließend sieht die Symbolleiste so aus wie die in Abbildung 5.12 und Abbildung 5.14 .

Jetzt haben Sie vier wichtige Befehle für die Programmierarbeit übersichtlich beieinander in der VBA-Symbolleiste. Vielleicht finden Sie auch noch mehr Symbole für Befehle, die Sie ständig brauchen. Sie wissen jetzt ja, wie Sie sich die Symbolleiste ganz nach Belieben selbst einrichten können!

Übrigens: Sie können Symbole auch auf dem umgekehrten Weg wieder aus der Symbolleiste entfernen. Dazu müssen Sie einfach das entsprechende Symbol per Drag-and-drop aus der Symbolleiste in das geöffnete *Anpassen*-Fenster ziehen.

Programmierbefehle

Nun sind wir eigentlich immer noch nicht beim »richtigen« Programmieren angekommen. Jetzt aber!

Um mit VBA in Access programmieren zu können, müssen Sie dreierlei wissen bzw. können:

- Sie müssen einige grundlegende Programmierbefehle kennen, die es in jeder Programmiersprache gibt und die erst einmal gar nichts mit Access zu tun haben. Dem ist dieses Kapitel gewidmet.
- Sie müssen die Steuerelemente kennen, die benötigt werden, um Formulare zu erstellen. Darum geht es im nächsten Kapitel.
- Sie müssen spezielle Programmiertechniken kennen, mit denen man den Inhalt von Datenbanktabellen lesen und auf dem Formular darstellen kann bzw. – umgekehrt – mit denen man die Benutzereingaben aus dem Formular lesen und in die Datenbanktabellen eintragen kann. Darum geht es in Kapitel 8.

Das Drumherum

Mit Abbildung 5.14 werfen wir schon mal einen Blick voraus auf den ersten Programmierbefehl, der erst später besprochen werden soll. Zunächst wollen wir uns aber nur für das Drumherum interessieren.

Ganz oben steht *Option Compare Database*. Das steht immer automatisch da, und wir wollen das jetzt einfach mal so hinnehmen, denn die Erläuterung würde Sie momentan nur verwirren und von Wichtigerem ablenken.

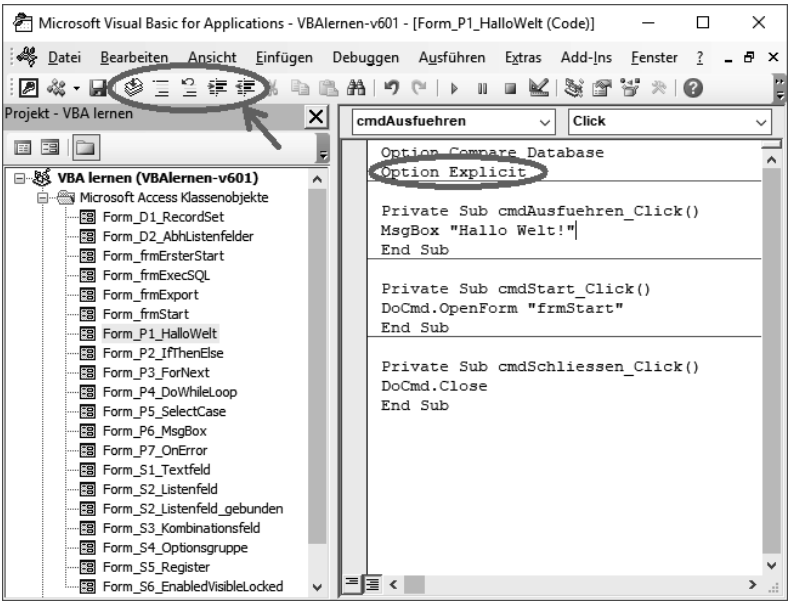

Abbildung 5.14: VBA-Code für das Ereignis Click auf dem Objekt cmdAusfuehren des Formulars P1-HalloWelt.

Das *Option Explicit* in der nächsten Zeile steht zunächst einmal nicht automatisch da, und ich empfehle Ihnen dringend, es immer als zweite Zeile einzugeben, also oberhalb aller VBA-Programme. Und was bedeutet das? Ich hatte weiter oben in diesem Kapitel im Abschnitt über die Syntaxfehler gesagt, dass der Debugger es merkt, wenn Sie mithilfe der *Dim*-Anweisung selbst definierte Variablennamen falsch schreiben. Das tut er aber genau genommen nur, wenn Sie ihn mittels *Option Explicit* dazu auffordern. Damit ist also gemeint: Variablen müssen explizit definiert werden. Das Gegenteil dazu wäre eine implizite Variablendefinition, bei der Sie einfach ohne vorheriges *Dim strVorname As String* schrieben: *strVorname = "Paul"*.

Die explizite Variablendefinition mit der *Dim*-Anweisung ist unbedingt vorzuziehen, weil falsch geschriebene Variablennamen sonst nicht als Syntaxfehler entdeckt werden.

Ein Beispiel dafür, was passieren kann, wenn Sie das *Option Explicit* weglassen und dann einen Variablennamen falsch schreiben, zeige ich Ihnen später im Abschnitt über die *While*-Anweisung.

Zwei Absätze weiter oben hatte ich geschrieben, dass *Option Explicit* **zunächst** einmal nicht automatisch dasteht. Sie können aber dafür sorgen, dass das automatisch immer dann geschieht, wenn Sie ein neues Modul anlegen. Dazu wählen Sie im VBA-Fenster im Menü *Extras* die Option *Optionen/Editor* und aktivieren dann die Option *Variablendeklaration erforderlich*. Das wirkt sich **nicht** auf bereits vorhandene Module aus, sondern nur auf neu anzulegende.

Kommen wir zur nächsten Zeile in Abbildung 5.14, die mit *Private Sub* beginnt. Das ist die erste Zeile eines VBA-Programms. Wir wollen dieses ab jetzt fachgerecht »Prozedur« nennen! Eine Prozedur beginnt also mit *Private Sub* und endet mit *End Sub* – dazwischen steht dann das von Ihnen geschriebene VBA-Programm. Beide Zeilen **brauchen** Sie nicht von Hand einzugeben, und das **sollten** Sie auch nicht tun, um Tippfehlern erst gar keine Chance zu geben.

Es gibt zwei Arten, diese beiden Zeilen automatisch zu erzeugen. Die erste hatte ich weiter oben im Abschnitt über Objekte und Ereignisse schon beschrieben:

1. Sie öffnen ein Formular in Access in der Entwurfsansicht,
2. erstellen ein Steuerelement (z. B. eine Schaltfläche),
3. klicken mit der rechten Maustaste darauf,
4. wählen im Kontextmenü den Befehl *Eigenschaften*,
5. wählen in dem sich öffnenden Eigenschaftenblatt die Registerkarte *Ereignis*,
6. klicken mit der linken Maustaste in die Zeile mit dem Ereignis, das Ihre VBA-Prozedur auslösen soll (z. B. *Beim Klicken*), und
7. klicken dann schließlich auf die kleine Schaltfläche mit den drei Punkten.

Daraufhin öffnet sich das VBA-Fenster, und die beiden gewünschten Zeilen stehen da! Jetzt können Sie damit beginnen, Ihren VBA-Code einzugeben.

Sie können aber auch anders vorgehen, um die beiden Codezeilen *Private Sub …* und *End Sub* zu erzeugen:

1. Öffnen Sie das **linke** der beiden Kombinationsfelder am oberen Rand des VBA-Editors (in Abbildung 5.14 steht dort *cmdAusfuehren*) und
2. wählen Sie darin ein Objekt Ihres Formulars aus (z. B. *Bezeichnungsfeld1*).

Daraufhin wird automatisch der Rahmen für eine VBA-Prozedur für das Klicken auf dieses Objekt erzeugt (z. B. *Private Sub Bezeichnungsfeld1_Click()*).

3. Wenn Sie aber gar keine Prozedur schreiben wollten, die mit einem Mausklick ausgelöst wird, öffnen Sie das **rechte** der beiden Kombinationsfelder am oberen Rand des VBA-Editors (in Abbildung 5.14 steht darin *Click*) und
4. wählen das gewünschte Ereignis aus (z. B. *MouseMove*).

Daraufhin wird ein weiterer Rahmen für eine andere VBA-Prozedur erzeugt, die startet, wenn sich der Mauszeiger (ohne zu klicken!) nur über das ausgewählte Objekt hinwegbewegt. Die von VBA in übereifriger Dienstbereitschaft erzeugte Click-Prozedur können Sie jetzt wieder löschen. Sie wird übrigens auch automatisch gelöscht, wenn Sie den Code kompilieren.

Zur Übung sollten Sie die in den vorangegangenen Zeilen bei »z. B.« genannten Aktionen einmal ausführen. Dann werden Sie sehen, dass die erste Zeile für die *MouseMove*-Prozedur so aussieht:

```
Private Sub Bezeichnungsfeld1_MouseMove(Button As Integer, Shift As Integer,
X As Single, Y As Single)
```

Das ist ein weiteres Argument dafür, diese Codezeilen nicht selbst einzutippen, sondern automatisch erzeugen zu lassen. Einige Ereignisse erfordern nämlich die Angabe einer ganzen Reihe von Parametern, die Sie sonst sicher erst nachschlagen müssten.

Wenn Sie jetzt

```
MsgBox "Hallo!"
```

in die Leerzeile zwischen *Private Sub* und *End Sub* schreiben, können Sie die so erzeugte Prozedur auch gleich ausprobieren. Dazu wechseln Sie zu Access, lassen das Formular *P1_HalloWelt* in der Formularansicht anzeigen und bewegen die Maus ohne zu klicken über die Formularüberschrift *Hallo-Welt*. Sie werden erleben, dass sich daraufhin ein Meldungsfeld öffnet!

Sehr lehrreich für den wechselseitigen Zusammenhang zwischen Access und VBA ist es jetzt, wenn Sie das Formular *P1_HalloWelt* in der Entwurfsansicht anzeigen lassen und sich die Registerkarte *Ereignis* im Eigenschaftenblatt der Formulartitelzeile anschauen. Dann werden Sie nämlich entdecken, dass dort in der Zeile *Bei Mausbewegung* der Eintrag *Ereignisprozedur* steht. Access hat also gemerkt, dass Sie ihm per VBA eine Ereignisprozedur untergejubelt haben!

Hallo Welt!

So, jetzt haben wir endlich alles Wissen beisammen, um wirklich mit dem Programmieren beginnen zu können. Da seit Urzeiten jeder Programmierkurs damit beginnt, die fröhliche Botschaft »Hallo Welt!« auf den Bildschirm zu zaubern, wollen wir es auch so halten.

Wie das geht, habe ich schon mit Abbildung 5.14 verraten. Na ja, es ist auch nicht so überaus schwierig!

Bitte probieren Sie zunächst die fertige Prozedur in der Beispielanwendung *VBAlernen* aus und erzeugen Sie dann zur Übung eine weitere Schaltfläche auf dem Formular, die eine andere wichtige Botschaft verkündet! Ist es nicht ein schöner Erfolg, wenn das erste Miniprogramm funktioniert? (Siehe weiter oben in diesem Kapitel unter »Hackermentalität«!)

If-Then-Else

Nach dem endlos langen Vorlauf geht's nun zügig weiter mit dem Programmieren. Als Nächstes lernen Sie einen ganz wichtigen Programmierbefehl kennen: die Entscheidung – im Hackerjargon »If-Then-Else«. Ich denke, es ist unmittelbar klar, was dieser Befehl bewirkt. Schauen Sie sich dazu bitte das Beispiel in *VBAlernen* an, klicken Sie im Startformular auf die Schaltfläche *If-Then-Else*, geben Sie zwei Zahlen ein und klicken Sie dann auf *ungesicherte Ausführung* bzw. auf *gesicherte Ausführung*.

Normalerweise würde man ein Textfeld in einem Formular nicht »txtErsteZahl« nennen, sondern »erste Zahl«. Ich habe das hier nur aus didaktischen Gründen getan, damit Sie den Zusammenhang zwischen Access (Name des Textfelds) und VBA (gleichlautender Name einer Variablen) besser erkennen!

Sehen wir uns dazu zunächst den Code für die ungesicherte Ausführung an:

```
If txtErsteZahl < txtZweiteZahl Then
   MsgBox ("Die erste Zahl ist kleiner!")
Else
   MsgBox ("Die zweite Zahl ist kleiner oder gleich der ersten!")
End If
```

Sie müssten sich jetzt die Frage stellen: Warum gibt es trotz *Option Explicit* keine *Dim*-Anweisung für die beiden Variablen *txtErsteZahl* und *txtZweiteZahl*? Damit sind wir wieder einmal bei dem Zusammenhang zwischen Access und VBA: Diese beiden Variablen werden nicht in VBA definiert, sondern in Access. Es sind nämlich die Namen der beiden Textfelder auf dem Formular *P2_IfThenElse* (Abbildung 5.15)! Das bedeutet: Die Namen von Formularobjekten können in VBA als Variablen verwendet werden!

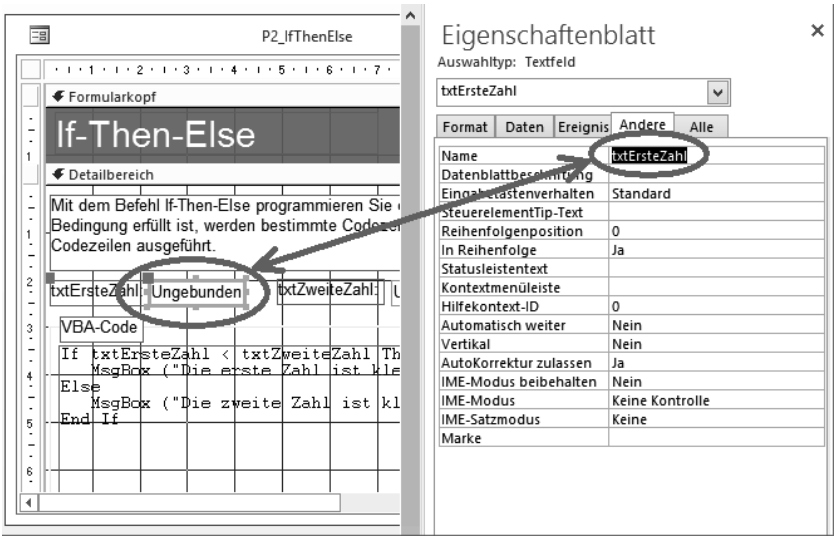

Abbildung 5.15: Die Namen von Formularobjekten können als Variablen in VBA verwendet werden!

Damit wird es ganz einfach,

- Benutzereingaben auf dem Formular in VBA-Prozeduren zu verwenden: Wenn der Benutzer in ein Textfeld mit dem Namen *txtErsteZahl* einen Wert eingibt, können Sie diesen Wert in Ihrer VBA-Prozedur einfach über den Namen des Textfelds für *If-Then-Else*-Vergleiche, für Berechnungen oder für irgendetwas anderes benutzen.
- Ergebnisse aus VBA-Prozeduren auf dem Formular darzustellen: Sie können einer Variablen mit dem Namen des Textfelds einfach einen Wert zuweisen – z. B. *txtErsteZahl = 5*. Probieren Sie das doch einfach mal aus, indem Sie auf dem Formular *P2_IfThenElse* ein Textfeld mit dem Namen *txtKleinereZahl* erstellen und dort mithilfe der Prozedur *cmdAusfuehrenUnsicher_Click()* die kleinere der beiden Zahlen *txtErsteZahl* und *txtZweiteZahl* hineinschreiben!

Ich hoffe, jetzt stellt sich langsam das »Hackerfeeling« bei Ihnen ein ...!? Und: **txt**ErsteZahl, **cmd**AusfuehrenUnsicher ... wie war das noch? Richtig! Die Namenskonvention!

Sie finden die Namenskonvention aus Anhang B auch im Internet (Adresse in der Einleitung, dort im Ordner *\KapA)* als PDF-Datei zum Ausdrucken.

Aber schauen wir uns einmal den Code der Prozedur *cmdAusfuehrenSicher_Click()* an. Dort sehen Sie vor dem eigentlichen *If-Then-Else* mit dem Vergleich der beiden vom Benutzer eingegebenen Zahlen noch zwei weitere *If-Then-Else*-Anweisungen:

```
If IsNull(txtErsteZahl) Then
   MsgBox "Bitte geben Sie eine erste Zahl ein!"
   txtErsteZahl.SetFocus
   Exit Sub
End If
```

```
If IsNull(txtZweiteZahl) Then
   MsgBox "Bitte geben Sie eine zweite Zahl ein!"
   txtZweiteZahl.SetFocus
   Exit Sub
End If
```

Diese beiden Anweisungen überprüfen, ob der Benutzer überhaupt eine Zahl eingegeben hat. Das ist ein ganz wichtiger Punkt, auf den ich vorher schon mehrfach hingewiesen habe:

Als Programmierer müssen Sie immer vorausdenken: »Was könnte der Benutzer meines Programms alles falsch machen?« Dann müssen Sie durch geschicktes Programmieren dafür sorgen, dass sich fehlerhafte Bedienungen durch den Benutzer nicht schädlich auf die Programmausführung auswirken.

In diesem Fall bedeutet das: Der Vergleich zwischen zwei Zahlen kann nur stattfinden, wenn auch zwei Zahlen vorliegen. Das klingt ungeheuer banal – aber genau so funktioniert das Programmieren: Sie müssen als Programmierer **alles** vorausdenken, was passieren kann – auch ganz primitive und scheinbar unsinnige Sachen!

Was tun wir also, wenn der Benutzer keine Zahl eingegeben hat? Wir weisen ihn in verständlicher Form darauf hin (also nicht etwa »Eingabefehler!« oder »Fehler 135: Fragen Sie Ihren Systemadministrator!«), wir setzen die Einfügemarke in das Feld, in dem noch keine Zahl steht *(SetFocus)* und beenden das Programm *(Exit Sub),* denn wenn eine der beiden Zahlen fehlt, ergibt ein Vergleich der beiden Zahlen keinen Sinn.

Sind das jetzt alle Fehler, die der Benutzer machen kann? Überlegen Sie mal! Richtig – er könnte ja statt einer Zahl auch etwas anderes eingeben: einen Buchstaben oder drei Fragezeichen. Das würde dann ebenfalls zu einem Laufzeitfehler führen, denn zwischen einem Buchstaben und einer Zahl können Sie keinen Größer-kleiner-Vergleich durchführen. Geben Sie doch mal in eines der beiden Eingabefelder einen Buchstaben ein und beobachten Sie, was passiert.

Sowohl bei der gesicherten als auch bei der ungesicherten Ausführung kommt die Meldung: »Sie haben einen Wert eingegeben, der für dieses Feld nicht gültig ist.« Diese Meldung müssen Sie nicht selbst programmieren, das macht Access automatisch. Aber woher weiß Access, dass eine Zahl und kein Buchstabe eingegeben werden soll?

Dazu öffnen Sie bitte das Formular *P2_If_Then_Else* in der Entwurfsansicht und schauen sich die Registerkarte *Format* im Eigenschaftenblatt des Textfelds *txtErsteZahl* an. Dort steht unter *Format*: *Allgemeine Zahl*. Sie können also schon beim Anlegen des Textfelds festlegen, dass es nur mit Zahlen gefüllt werden darf. Wenn Sie einmal das Kombinationsfeld in der Zeile *Format* öffnen, sehen Sie, dass Sie hier noch eine Reihe weiterer Festlegungen treffen können: *Datum, Zeit, Euro* usw.

An dieser Stelle können wir wieder ein lehrreiches Experiment durchführen. Dazu löschen Sie bitte den Eintrag *Allgemeine Zahl* auf den Eigenschaftenblättern beider Textfelder und öffnen das Formular dann in der Formularansicht. Jetzt können Sie in die Textfelder beliebige Zeichen eingeben, z. B. auch *txtErsteZahl* = *a* und *txtZweiteZahl* = *b*. Das führt dann zu dem Ergebnis: »Die erste Zahl ist kleiner!« Jetzt werden nämlich nicht Zahlen, sondern Zeichenketten miteinander verglichen. Das ist dann wie im Telefonbuch, in dem »Neumann« vor »Schulz« kommt.

Wenn Sie jetzt allerdings *txtErsteZahl = 11* und *txtZweiteZahl = 5* eingeben, führt das zu dem überraschenden Ergebnis: »Die erste Zahl ist kleiner!« Und das stimmt sogar! Die Zeichenkette »11« kommt nämlich vor der Zeichenkette »5«. Sie können sich sicherlich vorstellen, dass so etwas zu schwer aufzufindenden Fehlern im Programm führen kann. Sie denken die ganze Zeit, dass Sie mit der Zahl »11« operieren – in Wirklichkeit arbeitet Access aber mit der Zeichenkette »11«. Darum:

Legen Sie bei Textfeldern für die Eingabe von Daten immer den einzugebenden Datentyp fest (Eigenschaft *Format* im Eigenschaftenblatt). So verhindern Sie, dass der Benutzer z.B. Buchstaben statt Zahlen oder Eurobeträge statt Prozentangaben eingibt.

Ein weiteres Experiment: Geben Sie mal für die zweite Zahl nichts ein und für die erste irgendeine positive Zahl – z. B. »55«. Wenn Sie jetzt auf die Schaltfläche *ungesicherte Ausführung* klicken, erhalten Sie die Auskunft: »Die zweite Zahl ist kleiner oder gleich der ersten!« Ändern Sie die erste Zahl auf »-55«, meint Access wieder: »Die zweite Zahl ist kleiner oder gleich der ersten!« »Nichts« ist also kleiner als jede beliebige Zahl? Dann wäre »nichts« so etwas wie »minus unendlich«?

Nein – es ist so: Nach dem *If* kommt eine Frage, die mit *Ja* oder *Nein* zu beantworten ist. Wird sie mit *Ja* beantwortet, werden die Anweisungen zwischen *Then* und *Else* ausgeführt – andernfalls die Anweisungen zwischen *Else* und *End If*. Unser letztes Experiment hat uns darüber hinaus gezeigt: Kann die Frage gar nicht beantwortet werden, gilt das als *Nein*. Auch das kann wieder zu schwer auffindbaren Programmierfehlern führen.

Ich erläutere diesen einfachen Programmierbefehl *If-Then-Else* so ausführlich, weil ich Sie zu eigenen Experimenten anregen will.

Probieren Sie einfach alles aus, was Ihnen einfällt! Machen Sie dabei doch auch mal ganz »blöde« Sachen, die eigentlich gar nicht vorkommen dürften! Dann beobachten Sie, was passiert, und versuchen herauszubekommen, **warum** es passiert.

Die dabei gewonnenen Erkenntnisse sollten Sie schriftlich in einem kleinen Programmiertagebuch festhalten (natürlich auf dem Computer und nicht mit Füller und Büttenpapier!). Das hilft dann im Notfall eventuell schneller als aufwendiges Nachschlagen in Büchern und in der Onlinehilfe! Und es hilft auch dem Programmieranfänger, ein Gefühl fürs Hacken zu bekommen!

Ganz wichtig: Bei solchen Experimenten sollten Sie immer mit einem Absturz von Access bzw. sogar von Windows rechnen – also vorher alle anderen auf dem Computer laufenden Programme beenden und die Testdatenbank unter einem anderen Namen sichern. Eventuell geht Ihr Experiment nämlich so extrem schief, dass Sie die Testdatenbank hinterher gar nicht mehr öffnen können!

Wenn nach dem *Then* nur **eine einzige** Anweisung kommt, können Sie diese direkt dahinter in **dieselbe** Zeile schreiben und auf das *End If* verzichten:

```
If (Bedingung) Then (Anweisung)
```

Wenn nach dem *Then* **mehrere** Anweisungen folgen, müssen Sie diese in jeweils eine **neue** Zeile schreiben und mit *End If* abschließen. Dabei können Sie – wenn es keine Alternative gibt – auf das *Else* verzichten:

```
If (Bedingung) Then
   (Anweisung)
   (Anweisung)
   (Anweisung)
End If
```

Wenn Sie das *Else* jedoch benutzen, muss es ganz allein in einer Zeile stehen!

Abschließend zu *If-Then-Else* möchte ich Ihnen noch eine besondere Variante dieses Programmierbefehls zeigen. So können Sie aus drei oder mehr Alternativen statt nur aus zweien auswählen:

```
If (Bedingung) Then
     (Anweisungen)
ElseIf (Bedingung) Then
     (Anweisungen)
ElseIf (Bedingung) Then
     (Anweisungen)
ElseIf (Bedingung) Then
     (Anweisungen)
Else
     (Anweisungen)
EndIf
```

Eleganter ist jedoch die *Select-Case*-Anweisung (siehe weiter unten).

For-Next

Kommen wir jetzt zum nächsten elementaren Programmierbefehl, den es in jeder Programmiersprache gibt: die Schleife, d.h. die wiederholte Ausführung von Befehlen. Schauen Sie sich dazu bitte das Beispiel in *VBAlernen* an, klicken Sie im Startformular auf die Schaltfläche *For-Next*, geben Sie drei Zahlen ein und klicken Sie dann auf *ungesicherte Ausführung* bzw. auf *gesicherte Ausführung*.

Wählen Sie dabei den Abstand zwischen der ersten und der letzten Zahl nicht zu groß, sonst sind Sie ziemlich lange mit dem Wegklicken der Meldungsfelder beschäftigt und kommen nicht mehr zum Lesen!

Sehen wir uns zunächst wieder den Code für die ungesicherte Ausführung an:

```
Dim lngLaufzahl As Long
For lngLaufzahl = txtErsteZahl To txtLetzteZahl Step txtSchrittweite
   MsgBox "Jetzt bin ich bei " & Str(lngLaufzahl) & " !"
Next lngLaufzahl
```

Es wird eine Variable *lngLaufzahl* definiert, die in der *For*-Anweisung zunächst den Wert von *txtErsteZahl* bekommt (z. B. »1«). Dann werden die Anweisungen zwischen *For* und *Next* ausgeführt, und beim Erreichen der *Next*-Anweisung wird der Wert von *txtSchrittweite* (z. B. »5«) zu *lngLaufzahl* hinzuaddiert (z. B. »6«). Daraufhin kehrt die Programmausführung zur Zeile mit der *For*-Anweisung zurück. Jetzt wird überprüft, ob der Wert von *lngLaufzahl* größer als *txtLetzteZahl* (z. B. »20«) ist. Ist das nicht der Fall, werden die Anweisungen zwischen *For* und *Next* erneut ausgeführt. Durch das ständige Erhöhen von *lngLaufzahl* um *txtSchrittweite* ist *lngLaufzahl* aber irgendwann gleich oder größer als *txtLetzteZahl*. In dem Fall werden die Anweisungen zwischen *For* und *Next* nicht mehr ausgeführt – das Programm setzt stattdessen mit der Anweisung hinter *Next* fort. So einfach ist das!

Experimente:

- Was passiert bei ungesicherter Ausführung, wenn die erste Zahl größer als die letzte Zahl ist?
- Was passiert, wenn die Schrittweite größer als der Abstand zwischen erster und letzter Zahl ist?
- Was passiert, wenn die Schrittweite negativ ist?

Im Code für die gesicherte Ausführung finden Sie mehrere *If-Then-Else*-Anweisungen. Sie bedürfen keiner weiteren Erläuterung – bis auf die letzte. Sie setzt für die Schrittweite einen Standardwert ein, wenn der Benutzer keine Angabe gemacht hat. Sie können diesen Effekt beobachten, wenn Sie für die erste und die letzte Zahl etwas eingeben, für die Schrittweite aber nichts. Wenn Sie dann auf *gesicherte Ausführung* klicken, erscheint im Textfeld *txtSchrittweite* eine »1«.

Do-While-Loop

Mit *Do-While-Loop* wird ebenfalls eine Schleife definiert – im Unterschied zu *For-Next* steht aber nicht von vornherein fest, wie oft die Schleife durchlaufen wird. Stattdessen wird nach jedem Schleifendurchlauf eine Bedingung getestet. Ist die Bedingung erfüllt, wird die Schleife erneut durchlaufen, andernfalls setzt die Programmausführung mit der Anweisung fort, die nach *Loop* steht.

Schauen Sie sich dazu bitte das Beispiel in *VBAlernen* an, klicken Sie im Startformular auf die Schaltfläche *Do-While-Loop*, geben Sie eine Zahl für den Entscheidungswert ein und klicken Sie dann auf *ungesicherte Ausführung* bzw. auf *gesicherte Ausführung*.

Wählen Sie dabei den Entscheidungswert nicht zu groß, sonst sind Sie ziemlich lang mit dem Wegklicken der Meldungsfelder beschäftigt!

Sehen wir uns wieder den Code für die ungesicherte Ausführung an:

```
Dim lngLaufvariable As Long
lngLaufvariable = 1
Do While lngLaufvariable < txtEntscheidungswert
   MsgBox "Jetzt bin ich bei " & Str(lngLaufvariable) & " !"
   lngLaufvariable = lngLaufvariable + 1
Loop
```

Hier sind zwei Dinge wichtig:

- Sie müssen für die Laufvariable einen Startwert festlegen. Das habe ich hier einfach im Programm mit der Zeile *lngLaufvariable = 1* getan. Der Wert könnte natürlich auch berechnet oder aus einem Textfeld des Formulars entnommen werden.
- Sie müssen den Wert der Laufvariablen innerhalb der Schleife – also zwischen *Do While* und *Loop* – ändern, und zwar so, dass irgendwann die Bedingung in der *Do While*-Zeile nicht mehr erfüllt ist. Sonst entsteht eine der gefürchteten Endlosschleifen!

Wenn Sie mathematisch ein wenig interessiert sind, wird Ihnen vielleicht die Zeile *lngLaufvariable = lngLaufvariable + 1* sauer aufstoßen. Kann das überhaupt sein? Was folgt denn mathematisch aus x = x + 1? 0 = 1!

Auflösung des Preisrätsels: Sie dürfen diesen Befehl nicht mathematisch deuten. Es handelt sich um eine Programmiertechnik, die den Computer anweist: »Nimm den Inhalt des Speicherbereichs mit dem Namen *lngLaufvariable* (darum der *Dim*-Befehl, damit wird ein Speicherbereich mit diesem Namen reserviert!), erhöhe ihn um eins und speichere das Ergebnis wieder im Speicherbereich mit dem Namen *lngLaufvariable* ab.«

Experimente

Jetzt kommen Experimente, in denen Endlosschleifen produziert werden. Da kommen Sie nur wieder raus, indem Sie Access gewaltsam über den Task-Manager abbrechen. Es könnte auch sein, dass Access oder sogar Windows abstürzt – also bitte vorher alle anderen auf dem Computer laufenden Programme beenden und die Testdatenbank unter einem anderen Namen sichern.

Wir wollen uns jetzt das typische Problem einer Schleife ansehen – dass sie nämlich nicht beendet wird, sondern endlos weiterläuft. Dazu brauchen Sie als Entscheidungswert nur einen Buchstaben statt einer Zahl einzugeben und dann die ungesicherte Ausführung zu starten. Aus dem Abschnitt über *If-Then-Else* wissen Sie schon, dass die Bedingung *lngLaufvariable < txtEntscheidungswert* dann nicht auswertbar ist, was von Access als *Nein* interpretiert wird. Da die Bedingung also nie erfüllt wird, läuft die *While*-Schleife endlos weiter. Damit das funktioniert, habe ich im Eigenschaftenblatt des Eingabefelds für den Entscheidungswert das Format diesmal nicht auf *Allgemeine Zahl* gesetzt. Es ist also möglich, in dieses Feld irgendwelche Zeichen einzugeben.

Im VBA-Code für die gesicherte Ausführung steht deshalb:

```
If Not (IsNumeric(txtEntscheidungswert)) Then
   MsgBox "Bitte geben Sie eine Zahl ein!"
   txtEntscheidungswert = ""
```

```
    txtEntscheidungswert.SetFocus
    Exit Sub
End If
```

Ich hoffe, Sie haben vor dem Start Ihres Experiments alle Vorsichtsmaßnahmen getroffen. Sie müssten nun eine endlose Folge von »Jetzt bin ich bei ...«-Meldungsfeldern sehen. Ihre VBA-Prozedur läuft und läuft und läuft ...

Rettung bringt jetzt nur noch der Task-Manager, den Sie mit der Tastenkombination Strg+Alt+Entf aufrufen. Dort sehen Sie auf der Registerkarte *Anwendungen* in der Liste den Eintrag *Microsoft Access*. Den klicken Sie einmal an und klicken dann auf die Schaltfläche *Task beenden*. Damit müsste der Spuk vorbei sein. Sollte sich bei Ihnen etwas Schlimmeres ereignet haben (ein Access- oder sogar Windows-Absturz), sollten Sie Ihren Computer jetzt neu starten.

Wenn Ihnen ein solches Experiment zu gefährlich ist, können Sie folgendermaßen eine »Notbremse« einbauen:

```
If lngLaufvariable > 50 Then Exit Sub
```

Geben Sie diese Zeile zwischen *Do While* und *Loop* ein. Dann kann Ihre Laufvariable nicht größer als 50 werden.

Ein weiteres Experiment:

Ich hatte im Abschnitt »Das Drumherum« die *Option Explicit* erwähnt, die den Debugger zwingt, zu untersuchen, ob alle Variablen mittels *Dim*-Anweisung deklariert wurden. Was kann passieren, wenn Sie das nicht tun?

Dazu doppelklicken Sie bitte auf *Form_P4_DoWhileLoop* in der linken Hälfte des VBA-Fensters und löschen dann in der rechten Hälfte ganz oben die Zeile *Option Explicit*. Anschließend ändern Sie in der Prozedur *cmdAusfuehrenUnsicher_Click()* die Zeile *lngLaufvariable = lngLaufvariable + 1* in *lngLaufvariable = lngLauvariable + 1*. Wir simulieren damit also einen Tippfehler. Dieser wird jetzt vom Debugger nicht mehr bemerkt, und die Prozedur startet beim Klick auf die entsprechende Schaltfläche trotz Syntaxfehler. Das führt dann zu einem Laufzeitfehler in Form einer Endlosschleife. Sie sehen endlos oft die Meldung »Jetzt bin ich bei 1!«. Wegen Rettung siehe oben.

Select-Case

Ich hatte weiter oben im Abschnitt über *If-Then-Else* bereits erläutert, wie Sie eine Auswahl aus mehr als zwei Alternativen programmieren können. Eleganter geht es aber mit der *Select-Case*-Anweisung. Dazu klicken Sie im Startformular unserer Beispielanwendung *VBA-lernen* auf die Schaltfläche *Select-Case*, geben ein beliebiges Zeichen ein und klicken dann auf *ungesicherte Ausführung* bzw. auf *gesicherte Ausführung*.

Für die *Case*-Zeilen sind mehrere verschiedene Formen zugelassen – sowohl für Zahlen als auch für Texte:

```
Select Case lngZahl
   Case 1,2,3
      (Anweisungen)
   Case 4 To 10
      (Anweisungen)
   Case Is > 11
      (Anweisungen)
   Case Else
      (Anweisungen)
End Select
```

```
Select Case strText
   Case "Alles"
      (Anweisungen)
   Case "Nüsse" To "Suppe"
      (Anweisungen)
   Case Testtext
      (Anweisungen)
   Case Else
      (Anweisungen)
End Select
```

Was Sie sicherheitshalber immer tun sollten – auch wenn es Ihnen ganz und gar unmöglich erscheint, dass der zu testende Ausdruck (hinter *Select Case*) andere Werte annimmt, als von Ihnen in den einzelnen *Case*-Zeilen vorgesehen: Schreiben Sie Anweisungen hinter *Case Else* für den Fall, dass es doch passiert. Und es wird passieren! Das ist durch Murphys Gesetz garantiert!

MsgBox

Erinnern Sie sich? Die Messagebox war einer der ersten Programmierbefehle, den Sie kennengelernt haben. (Einer meiner Studenten hat übrigens mal in einer Klausur »Massagebox« geschrieben. Auch eine nette Idee!) Die Messagebox hat uns die Botschaft »Hallo Welt!« auf den Bildschirm gezaubert. Aber das war nur die allereinfachste Form der Messagebox. Sie hat lediglich eine einzige Schaltfläche *(OK)*; wenn man sie anklickt, schließt sich die Messagebox wieder. Das war's!

Eingabeparameter
Von der Benutzung der verschiedensten Windows-Programme her kennen Sie aber bestimmt noch andere Messageboxen mit mehr als einer Schaltfläche. Wenn Sie z. B. eine Datei unter einem Namen speichern wollen, den es schon gibt, erscheint sicherheitshalber eine Messagebox mit den beiden Schaltflächen *Ja* und *Nein* und der Frage, ob Sie die vorhandene Datei ersetzen wollen. In anderen Fällen gibt es noch eine dritte Wahlmöglichkeit: *Abbrechen*. Alles das kann auch die VBA-Funktion *MsgBox*. Dazu muss man sie nur statt so:

```
MsgBox "Hier kommt eine Nachricht!"
```

so schreiben:

```
Dim lngAntwort As Long
lngAntwort = MsgBox ("Wollen Sie wirklich löschen?", vbYesNo)
If lngAntwort = vbYes Then
   …
Else
   …
End If
```

Das ist sicherlich erklärungsbedürftig. Also: Die Funktion *MsgBox* kann nicht nur eine Information auf dem Bildschirm anzeigen, sie liefert auch einen Ergebniswert in Form einer ganzen Zahl. Das müssen Sie sich ähnlich vorstellen wie bei der Funktion *WURZEL*. Wenn Sie diese Funktion auf die Zahl 25 anwenden, liefert sie als Ergebnis die Zahl 5: *WURZEL(25)=5*. Eine Funktion bekommt also einen oder mehrere Parameter und berechnet daraus ein Ergebnis. Das ist im Fall der Quadratwurzelberechnung völlig klar, bei der Funktion *MsgBox* für den Programmieranfänger aber erst einmal gewöhnungsbedürftig.

Die Funktion *MsgBox* bekommt also im oben genannten Beispiel die beiden Parameter »Wollen Sie wirklich löschen?« und *vbYesNo*. Was macht sie damit? Sie erzeugt ein kleines Fenster auf dem Bildschirm mit der Frage »Wollen Sie wirklich löschen?« und mit zwei Schaltflächen, die mit *Ja* und *Nein* beschriftet sind. Letzteres legt der zweite Parameter *vbYesNo* fest. Das ist eigentlich eine Zahl, nämlich die 4. Wenn Sie also der Funktion *MsgBox* als zweiten Parameter eine »4« mitgeben, hat die auf dem Bildschirm erscheinende Messagebox die beiden Schaltflächen *Ja* und *Nein*. Hätten Sie als zweiten Parameter »5« gewählt, wären die beiden Schaltflächen mit *Wiederholen* und *Abbrechen* beschriftet worden. Da sich diese ganzen Zahlen aber niemand merken kann, stellt VBA dafür entsprechende Namen zur Verfügung: *vbYesNo*, *vbRetryCancel* usw. Eine vollständige Liste aller möglichen Parameter finden Sie in der VBA-Hilfe unter dem Stichwort *MsgBox*. Dort können Sie auch nachlesen, dass die Funktion *MsgBox* noch drei weitere Parameter hat, auf die ich hier aber nicht näher eingehen möchte.

Antwortwert

Jetzt müssen Sie als Entwickler noch dafür sorgen, dass die Benutzeraktion richtig ausgewertet wird. Das bedeutet, dass das Programm unterschiedlich fortfahren muss, und zwar abhängig davon, ob der Benutzer die *Ja*-Schaltfläche oder die *Nein*-Schaltfläche angeklickt hat. Woher wissen Sie, was der Benutzer angeklickt hat? Vom Wert der Funktion *MsgBox* nach dem Klick! Dieser Wert wird im obigen Beispiel in der Variablen *lngAntwort* gespeichert, die anschließend mit einem *If-Then-Else*-Befehl ausgewertet wird. Auch dieser Antwortwert ist eigentlich wieder eine Zahl, für die es eine Reihe von definierten Werten gibt: 6 bedeutet z. B. *Ja* und 7 *Nein.* Die Bedeutung der übrigen Werte zwischen 1 und 5 können Sie wiederum der VBA-Hilfe zum Stichwort *MsgBox* entnehmen.

Bitte schauen Sie sich in der Beispielanwendung *VBAlernen* jetzt das Formular *P6_MsgBox* und den dazugehörigen VBA-Code an (Abbildung 5.16). Da eine Messagebox auch mehr als zwei Schaltflächen haben kann, erfolgt die Auswertung der Benutzeraktion dort nicht mit *If-Then-Else* (nur zwei Alternativen), sondern mit *Select-Case* (beliebig viele Alternativen).

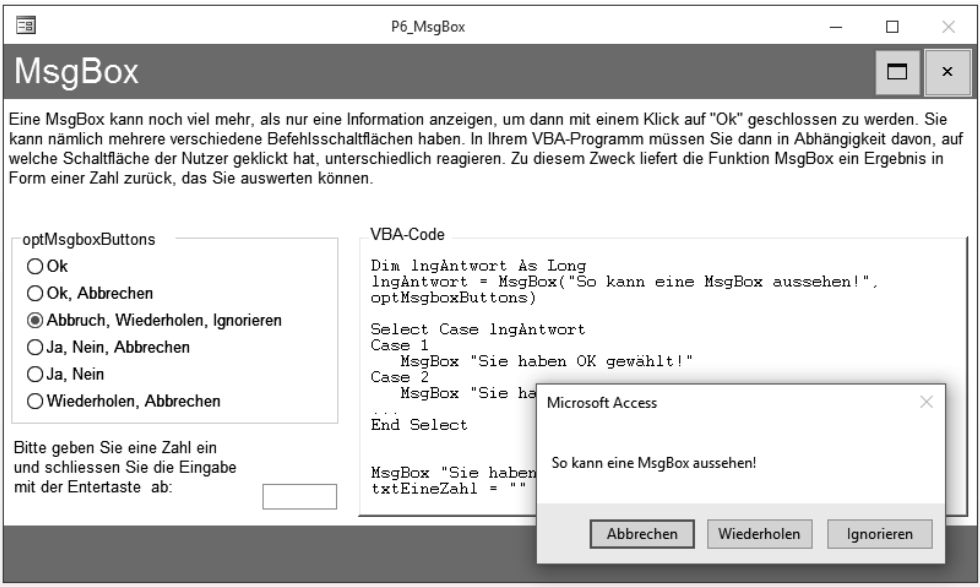

Abbildung 5.16: Eine Messagebox kann mit verschiedenen Schaltflächen programmiert werden.

Daten anzeigen

Ich erläutere die *MsgBox* hier so ausführlich, weil sie für vielerlei Zwecke eingesetzt werden kann – unter anderem auch, um Daten anzuzeigen. Dazu müssen die Daten (z. B. Zahlen) allerdings in das Textformat umgewandelt werden. Das geschieht mit dem VBA-Befehl *Str(…)* – von engl. »string«.

Der VBA-Befehl *Str(…)* fügt vor die in einen Text umzuwandelnden Zeichen ein zusätzliches Leerzeichen ein. Wenn Sie das nicht wollen, müssen Sie *Trim(Str(…))* schreiben oder die Funktion *Format(…)* verwenden.

Außerdem muss der gesamte auszugebende Text in **einem** Parameter enthalten sein. Das ist aber ganz einfach, weil Texte wie Zahlen addiert werden können: *"Paul"&" Meier"="Paul Meier"*. Theoretisch könnten Sie für die Textaddition auch das Zeichen »+« benutzen – üblicherweise nimmt man aber das »&«-Zeichen.

Im Formular *P6_MsgBox* wird dafür z. B. der Befehl

```
MsgBox "Sie haben eine " & Str(txtEineZahl) & " eingegeben!"
```

verwendet, um die Eingabe des Benutzers in das Textfeld *txtEineZahl* wieder auszugeben.

Wenn Sie die MsgBox nur in ihrer einfachsten Form für die Ausgabe einer Information verwenden wollen, können Sie den ersten Parameter einfach hinter das Schlüsselwort *MsgBox* schreiben. Möchten Sie dagegen den Klick des Benutzers auf eine der Schaltflächen auswerten, müssen Sie die beiden Parameter in runde Klammern setzen und den Ausgabewert der Funktion *MsgBox* einer Variablen zuweisen.

Noch einmal: Eingabeparameter

Ich hatte weiter oben schon beschrieben, dass der zweite Eingabeparameter der Funktion *MsgBox* darüber entscheidet, wie viele und welche Schaltflächen die Messagebox hat. Mit der richtigen Wahl dieses Parameters können Sie aber noch viel mehr erreichen: In der VBA-Hilfe zum Stichwort *MsgBox* finden Sie unter anderem die Auskunft »16 (= *vbCritical*) = Meldung mit Stopp-Symbol anzeigen«.

Und wie stellen Sie es nun an, wenn Sie die drei Schaltflächen *Abbrechen*, *Wiederholen* und *Ignorieren* (=*vbAbortRetryIgnore*=2) in der Messagebox haben wollen und **zusätzlich** das Stopp-Symbol (=*vbCritical*=16)? Ganz einfach: Sie geben als zweiten Parameter *18 (=16+2)* ein! Dann sieht die Messagebox nicht wie in Abbildung 5.16 aus, sondern so:

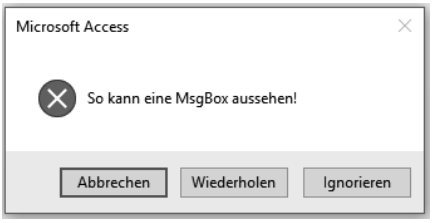

Abbildung 5.17: Eine Messagebox mit dem zweiten Parameter = 18.

Experimentieren Sie ruhig einmal etwas mit dieser Technik herum, d.h., ändern Sie in der Beispielanwendung *VBAlernen* in der VBA-Prozedur *Private Sub optMsgboxButtons_Click()* die dritte Zeile etwa folgendermaßen:

```
lngAntwort = MsgBox("So kann eine MsgBox aussehen!", & _
                    optMsgboxButtons + vbCritical)
```

Dabei bezeichnet *optMsgboxButtons* den Namen der sogenannten Optionsgruppe mit den verschiedenen Möglichkeiten für die Art und Anzahl der Schaltflächen in einer Messagebox (links in Abbildung 5.16).

Zeilenumbruch

Wenn Sie in dem von der Messagebox angezeigten Text einen Zeilenumbruch haben möchten, programmieren Sie

```
MsgBox "Bitte geben Sie in dem Feld Enddatum einen Wert ein," & vbCrLf & _
       "sonst kann die Transportzeit nicht berechnet werden."
```

Die merkwürdige Abkürzung *vbCrLf* kommt noch aus der Schreibmaschinenzeit: *Cr* bedeutet »Carriage Return« (Wagenrücklauf – also zurück an den Anfang der Zeile), und *Lf* bedeutet »Line Feed« (Zeilenvorschub – also weiter in der nächsten Zeile). Das bedeutet dann zusammengefasst: »Am Anfang der nächsten Zeile geht es weiter.«

Schauen Sie sich dazu einmal die ↵-Taste auf Ihrer Tastatur an. Darauf sehen Sie das Symbol eines Pfeils, der von oben nach unten und dann nach links zeigt. Das ist *CrLf*!

So, damit haben wir die wichtigsten Programmierbefehle erst einmal abgearbeitet. Noch mehr Seiten zu diesem Thema würden den Rahmen dieses kleinen Buchs sprengen. Es ging mir einerseits darum, dass Sie als Programmierneuling die Denkweise eines Programmierers (»Hackermentalität«) nachempfinden können. Andererseits möchte ich Sie aber auch befähigen, selbst erste VBA-Programme zu schreiben. Die Benutzung weiterer Programmierbe-

fehle können Sie sich bei Bedarf mithilfe eines anderen Buchs oder unter Benutzung der integrierten VBA-Hilfe erarbeiten.

Laufzeitfehler verhindern

Aber damit ist dieses Kapitel noch nicht zu Ende. Ich hatte es mit dem Thema »Programmierfehler« begonnen und möchte es damit auch beenden. Sie können nämlich bei der VBA-Programmierung

- durch geeignete Befehle verhindern, dass später Laufzeitfehler auftreten, und
- bewirken, dass Ihr Programm sich bei Auftreten eines Laufzeitfehlers benutzerfreundlich verhält, d.h., der Benutzer wird in verständlicher Form über die Art des aufgetretenen Laufzeitfehlers informiert, und das Programm stürzt nicht ab.

Benutzereingaben prüfen

Es gibt kaum ein Programm, in dem der Benutzer nicht aufgefordert wird, Informationen einzugeben. Entweder tippt er sie selbst in Textfelder ein, oder er wählt sie aus Listen-, Kombinations- oder Optionsfeldern aus. Diese Informationen verwendet das Programm dann als Grundlage für seine Aktionen. Eine ganz wichtige Aufgabe zur Vermeidung von Laufzeitfehlern besteht daher darin, die Benutzereingaben zu überprüfen.

Was kann der Benutzer denn alles falsch machen?

Benutzerfehler	Codebeispiel

Absichtlich oder unabsichtlich Eingabefelder leer gelassen.	Aus *P2_IfThenElse*:
	`If IsNull(txtErsteZahl) Then`
	`  MsgBox "Bitte geben Sie eine erste Zahl ein!"`
	`  txtErsteZahl.SetFocus`
	`  Exit Sub`
	`End If`

Einen falschen Datentyp eingegeben (z.B. Buchstaben statt Zahlen).	Aus *P4_DoWhileLoop*:
	`If Not (IsNumeric(txtEntscheidungswert)) Then`
	`  MsgBox "Bitte geben Sie eine Zahl ein!"`
	`  txtEntscheidungswert = ""`
	`  txtEntscheidungswert.SetFocus`
	`  Exit Sub`
	`End If`
	Bemerkung: Es gibt außerdem noch die Funktion *IsDate*.

Widersprüchliche oder unplausible Daten eingegeben (z.B. ein Enddatum, das zeitlich vor dem Anfangsdatum liegt).	Aus *P3_ForNext*:
	`If txtErsteZahl >= txtLetzteZahl Then`
	`  MsgBox "Die erste Zahl muss kleiner als " & _`
	`  "die letzte Zahl sein!"`
	`  txtErsteZahl.SetFocus`
	`  Exit Sub`
	`End If`

Tabelle 5.1: Möglichkeiten zur Überprüfung der Benutzereingaben

Benutzerfehler	Codebeispiel

Falsche Daten eingegeben.
Das ist natürlich im Allgemeinen schwer oder gar nicht zu erkennen, denn ob jemand Meier oder Maier heißt, kann das Programm nicht wissen. Wenn aber ein Geburtsdatum im 22. Jahrhundert eingegeben wird, ist das sicherlich falsch.
	`If txtGeburtsdatum > Now() Then`
	`  MsgBox "Das Geburtsdatum liegt in der Zukunft!"`
	`  txtGeburtsdatum = ""`
	`  txtGeburtsdatum.SetFocus`
	`  Exit Sub`
	`End If`

*Tabelle 5.1: Möglichkeiten zur Überprüfung der Benutzereingaben (*Fortsetzung*)*

In fast allen Codebeispielen in Tabelle 5.1 sehen Sie wieder zwei kleine Beiträge zur Benutzerfreundlichkeit:

- Der falsch eingegebene Wert wird gelöscht, damit der Benutzer das nicht selbst machen muss.
- Die Einfügemarke wird mit dem Befehl *SetFocus* in das Feld gesetzt, in dem eine neue Eingabe erwartet wird, sodass der Benutzer gleich losschreiben kann, ohne erst zur Maus greifen zu müssen.

Bedienreihenfolge erzwingen

Ein weiterer »beliebter« Benutzerfehler besteht darin, die Steuerelemente – vorzugsweise die Schaltflächen – in der falschen Reihenfolge zu bedienen. Schon allein für die drei Schaltflächen *Neuen Datensatz einfügen*, *Datensatz speichern* und *Datensatz löschen* gibt es viele Fehlermöglichkeiten: Der Benutzer könnte z. B.

- **dieselbe Schaltfläche zweimal nacheinander anklicken.**

  Dann müssen Sie dafür sorgen, dass das ungefährlich ist. Hat der Benutzer z. B. *Neuen Datensatz einfügen* angeklickt, Daten eingegeben und dann erneut auf *Neuen Datensatz einfügen* geklickt, muss zunächst der erste eingegebene Datensatz gespeichert werden, und dann erst darf der Nutzer die Möglichkeit haben, den nächsten neuen Datensatz einzugeben.

- ***Datensatz speichern* anklicken, obwohl er gar keine Daten eingegeben hat.**

  Daher müssen Sie vor dem Speichern überprüfen, ob die Eingabefelder nicht leer sind (*If IsNull()*).

- ***Datensatz löschen* anklicken, obwohl er gar keinen zu löschenden Datensatz ausgewählt hat.**

  Vor dem Löschen müssen Sie überprüfen, ob der Benutzer einen Datensatz im Listenfeld angeklickt hat.

- ***Neuen Datensatz einfügen* anklicken und unmittelbar darauf *Datensatz speichern* oder *Datensatz löschen*.**

  Hier greifen die bereits in den vorangegangenen beiden Punkten erläuterten Sicherheitsmaßnahmen.

- **einen neuen Datensatz eingeben und dann das Formular schließen, ohne auf *Datensatz speichern* zu klicken.**

  Programmieren Sie einen *Speichern*-Befehl in der Prozedur *Form_Close* oder deaktivieren Sie die Standard-*Schließen*-Schaltfläche (die Schaltfläche mit dem X ganz rechts oben)

und programmieren Sie im VBA-Code Ihrer *Formular schließen*-Schaltfläche einen *Speichern*-Befehl.

Ein weiteres sehr nützliches Hilfsmittel zum Erzwingen einer bestimmten Bedienreihenfolge sind die Eigenschaften *Locked*, *Enabled* und *Visible*. Damit können Sie programmieren, dass nach der Bedienung bestimmter Steuerelemente andere Steuerelemente gesperrt oder sogar unsichtbar werden. Sie werden dann erst wieder freigegeben bzw. sichtbar, wenn der Benutzer die von Ihnen gewünschte Aktion ausführt.

Beispiel: In den VBA-Code zu der Schaltfläche *Neuen Datensatz einfügen* schreiben Sie die Zeile

```
cmdSpeichern.Enabled = False
```

und in den VBA-Code für die Änderung eines Textfelds (z.B. *txtName_AfterUpdate()*) schreiben Sie die Zeile

```
cmdSpeichern.Enabled = True
```

Dann wird die *Speichern*-Schaltfläche nach dem Klick auf *Neuen Datensatz einfügen* zunächst deaktiviert und erst wieder freigegeben, wenn der Benutzer einen Kundennamen eingegeben hat.

Bitte sehen Sie sich zu diesem Thema auch die Beispieldatenbanken noch einmal aufmerksam an. Ich mache dort von dieser Technik – gerade nicht benötigte Steuerelemente zu deaktivieren – intensiv Gebrauch!

Fehlfunktionen vorhersehen

Wenn die Benutzereingaben alle überprüft und für richtig bzw. wenigstens für plausibel befunden wurden, kommt die nächste Hürde. Jetzt kann es nämlich passieren, dass mit den richtigen Daten falsch gearbeitet wird. Als Programmierer ist es Ihre Aufgabe, solche möglichen Fehlfunktionen vorherzusehen.

Division durch null

Wenn in Ihrem Programm Berechnungen durchgeführt werden, ist sicherlich aus der Mathematik bekannt, dass dabei nicht durch null dividiert werden darf. Bitte öffnen Sie das Formular *P7_OnError* aus der Beispielanwendung *VBAlernen*. Wenn Sie dort als zweite Zahl eine Null eingeben und dann auf die Schaltfläche *ungesicherte Ausführung* klicken, erhalten Sie den Laufzeitfehler 11, »Division durch null«. Dem kann man natürlich so vorbeugen:

```
If txtZweiteZahl = 0 Then
   MsgBox "Bitte geben Sie als zweite Zahl keine Null ein!"
   txtZweiteZahl = ""
   txtZweiteZahl.SetFocus
   Exit Sub
End If
```

Bitte beachten Sie den Unterschied zwischen *txtZweiteZahl = 0* und *IsNull (txtZweiteZahl)*! Ersteres bedeutet, dass in dem Textfeld *txtZweiteZahl* eine Null steht, Letzteres dagegen bedeutet, dass in dem Textfeld *txtZweiteZahl* nichts steht.

Nicht existierende Daten

Ein weiteres Beispiel für eine vorhersehbare Fehlfunktion während der Programmausführung: Es wird ja nicht nur mit Daten gearbeitet, die der Benutzer eingegeben hat, sondern auch mit Daten, die aus den Tabellen der Datenbank stammen. Hierfür gibt es spezielle VBA-Befehle, die die richtigen Daten aus den richtigen Tabellen herausholen und für die weitere Nutzung in Form von Variablen zur Verfügung stellen. Darauf werde ich in Kapitel 8, »VBA – Teil 2«, noch näher eingehen.

Sie können sich aber sicherlich auch so schon vorstellen, dass es passieren kann, dass die gesuchten Daten in der Datenbank gar nicht existieren. Daher müssen Sie als Programmierer dafür sorgen, dass Ihr Programm gar nicht erst versucht, mit nicht vorhandenen Daten weiterzuarbeiten. Wie das konkret funktioniert, kann ich jetzt noch nicht darstellen. Dafür muss ich in einem späteren Kapitel erst ein Beispiel entwickeln, an dem wir uns solche Probleme dann klarmachen werden.

OnError

Auch wenn Sie sich als Programmierer noch so viel Mühe geben – Sie werden nie alle möglichen Fehler voraussehen können. Glücklicherweise gibt es aber in VBA einen Befehl, mit dem sich auch unvorhersehbare Fehlfunktionen behandeln lassen. Dazu schauen Sie sich bitte einmal die Prozedur *cmdAusfuehrenSicher_Click()* des Formulars *P7_OnError* an. Dort wurden **keine** speziellen Vorkehrungen getroffen, um eine Division durch null zu verhindern. Stattdessen steht in der zweiten Zeile der Prozedur:

```
On Error GoTo fehlerbehandlung
```

Das bedeutet: Wenn bei der Ausführung der Prozedur in einer bestimmten Zeile ein (jetzt noch nicht näher definierbarer) Fehler auftritt, soll die Ausführung unterbrochen werden. Anschließend soll hinter der Zeile, in der *fehlerbehandlung* steht, weitergemacht werden:

```
Exit Sub
fehlerbehandlung:
   txtErgebnis = ""
   MsgBox "Es ist ein Fehler aufgetreten!" & vbCrLf & _
          "Beschreibung:" & vbCrLf & Err.Description
```

In unserem Beispiel besteht die Fehlerbehandlung aus zwei Aktionen:

- Das Textfeld, das eigentlich das Ergebnis anzeigen sollte, wird geleert. Das ist wichtig, denn dort könnte ja noch das Ergebnis einer vorangegangenen Berechnung stehen, das jetzt falsch wäre.
- Es wird eine einfache Messagebox erzeugt, die die Beschreibung des Fehlers (*Err.Description*) anzeigt.

Wichtig ist, dass vor der Zeile, in der die Fehlerbehandlung beginnt, der Befehl *Exit Sub* steht. Ansonsten würde die Fehlerbehandlung auch durchgeführt werden, wenn gar kein Fehler aufgetreten ist.

Sie können die im Fall eines Fehlers in der Messagebox angezeigten Informationen natürlich auch wesentlich umfangreicher gestalten.

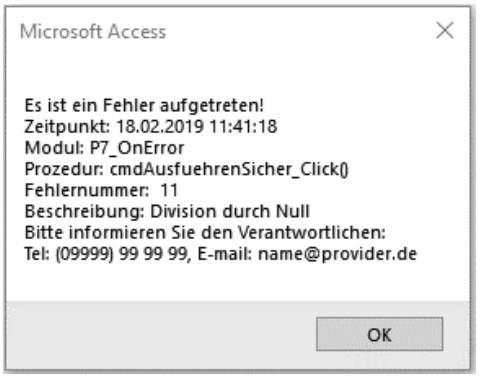

Abbildung 5.18: Damit können Benutzer und Entwickler etwas anfangen!

Abbildung 5.18 zeigt eine Fehlerinformation, mit der beide etwas anfangen können: Der Benutzer weiß, wo er Hilfe bekommt, und der Entwickler weiß ganz genau, wann an welcher Stelle in welcher Prozedur welcher Fehler aufgetreten ist.

Die Behandlung unvorhersehbarer Fehlfunktionen mit dem VBA-Befehl *OnError* sollte in keiner Ihrer Prozeduren fehlen!

Aber: Die Messagebox mit der Fehlerbeschreibung ist für den Endnutzer gedacht. Er soll nicht mit einem Sprung in den Quellcode und dem Anspringen des Debuggers konfrontiert werden.

Genau umgekehrt ist es für den Entwickler: Er möchte während der Programmentwicklung beim Auftreten eines Fehlers sofort an die richtige Stelle im Quellcode geführt werden, um den Fehler beheben zu können. Für ihn ist die Messagebox mit der Fehlerbeschreibung daher eher eine Behinderung!

Also sollte man die VBA-Prozeduren erst einmal ohne Fehlerbehandlung schreiben und diese dann später hinzufügen? Wer glaubt, dass er das dann nach Abschluss der Entwicklung wirklich noch macht, glaubt auch an den Weihnachtsmann! Üblicherweise wird der Termindruck gegen Ende der Entwicklung immer größer, während gleichzeitig die Lust sinkt, sich noch weiter damit zu beschäftigen. Daher sollten Sie besser gleich in jede Prozedur, die Sie neu anlegen, als Erstes die Zeilen für die Fehlerbehandlung hineinkopieren. Anschließend können Sie die Fehlerbehandlung für den Zeitraum der Entwicklung auf einfache Weise wieder ausschalten.

Dazu verwandeln Sie alle Programmzeilen

```
On Error GoTo fehlerbehandlung
```

mithilfe der Suchen-und-Ersetzen-Funktion in

```
If errorhandling Then On Error GoTo fehlerbehandlung
```

Dabei ist *errorhandling* die folgende Funktion:

```
Public Function errorhandling() As Boolean
errorhandling = False
End Function
```

Diesen kleinen Dreizeiler speichern Sie unter *Module* ab, und wenn Sie Ihre Entwicklungsarbeit beendet haben, ersetzen Sie einfach *False* durch *True* – schon ist die Fehlerbehandlung eingeschaltet!

Eine große Anzahl von VBA-Standardlösungen finden Sie in der Datei *Checkliste-Formulare.xlsx* im Internet (Adresse in der Einleitung, dort im Ordner *\KapA)*. Dort liste ich mehr als drei Dutzend Funktionalitäten auf, die jedes Ihrer Formulare aufweisen sollte, und beschreibe, wie sie realisiert werden können.

Was ist wichtig?

1. Finden Sie Syntaxfehler mit dem Debugger, beugen Sie Laufzeitfehlern durch entsprechende Programmierung vor und testen Sie Ihren Code mit Beispieldaten auf logische Fehler (siehe Abschnitt »Fehler finden und korrigieren« ab Seite 194 und Abschnitt »Laufzeitfehler verhindern« ab Seite 224).
2. Wenn Ihr Programm nicht mehr auf Mausklicks reagiert, könnte es sein, dass der Debugger läuft (siehe Abschnitt »Laufzeitfehler« ab Seite 198).
3. Tippen Sie die erste Zeile einer Prozedur niemals selbst ein, sondern lassen Sie sie von Access bzw. VBA automatisch generieren (siehe Abschnitt »Das Drumherum« ab Seite 209).
4. Finden Sie hartnäckige Programmierfehler durch schrittweise Programmausführung bzw. durch das Setzen von Haltepunkten (siehe Abschnitt »Der Debugger« ab Seite 207).
5. Passen Sie die VBA-Symbolleiste an, indem Sie ihr Symbole für häufig benötigte Befehle hinzufügen (siehe Abschnitt »Symbolleiste anpassen« ab Seite 208).
6. Zwingen Sie sich durch die Einstellung *Option Explicit* immer selbst zur expliziten Variablendeklaration (siehe Abschnitt »Das Drumherum« ab Seite 209 und Abschnitt »Experimente« ab Seite 218).
7. Benutzen Sie beim Testen Ihres Programms die Messagebox für die Ausgabe von Zwischenergebnissen (siehe Abschnitt »MsgBox« ab Seite 220).

8. Sehen Sie mit *OnError* auch Reaktionen auf nicht vorhersehbare Fehler vor und schalten Sie diese Fehlerbehandlung erst nach der Übergabe der Access-Anwendung an den Benutzer ein (siehe Abschnitt »OnError« ab Seite 227).
9. Benutzen Sie meine Standardlösungen aus *Checkliste-Formulare.xlsx* (im Ordner *\Buch\KapA)*.

Sie finden das Dokument *WasIstWichtig.pdf* zum Ausdrucken im Internet (Adresse in der Einleitung, dort im Ordner *\KapA*).