3 Anwendungen mit Benutzerschnittstelle (UI) entwickeln

Programmierer, deren Software aus Bibliotheken von benutzerdefinierten Funktionen besteht, brauchen sich für die Benutzerschnittstelle nicht zu interessieren. Da die UDF von Excel aus aufgerufen werden, dient das jeweilige Excel-Tabellenblatt, von dem aus der Aufruf erfolgt, als Benutzerschnittstelle. Es kann zwar sein, dass der Excel-Benutzer mit der schmucklosen Präsentation des UDF-Ergebnisses nicht zufrieden ist und dieses vielleicht noch durch Rahmen oder Farben verschönern will, aber dies ist seine eigene Angelegenheit. Eine UDF kann zur Präsentation nichts beitragen, sie liefert nur das nackte Ergebnis.

Nun können UDF zwar mächtig sein, aber auch mächtige Funktionen können nicht alle Ansprüche erfüllen, die ein Benutzer an Software haben kann. Dabei geht es hier nicht nur um die Präsentation der Ergebnisse. Viel wichtiger ist die Steuerung des gesamten Vorgehens. Natürlich kann ein geschickter Excel-Benutzer auch komplexe Aufgaben mit Hilfe von UDF-Bibliotheken lösen, aber der Aufwand kann dafür sehr groß sein. Auch wäre es aus wirtschaftlicher Sicht nicht sinnvoll, wenn jeder Benutzer mit einem ähnlichen Problem seine eigene Lösung erarbeiten müsste.

Für komplexe Aufgaben benötigen wir also Software mit Benutzerschnittstelle. Diese hat die Aufgabe, die Kommunikation zwischen Programm und Benutzer abzuwickeln, also Eingaben entgegenzunehmen und Programmergebnisse zu präsentieren. Darüber hinaus steuert sie, mehr oder weniger strikt, das gesamte Vorgehen bei der Lösung der Aufgaben.

3.1 Möglichkeiten zur Gestaltung des UI

In einer vereinfachten Sicht können wir zwei Hauptformen des UI von Excel-VBA-Anwendungen unterscheiden:

- die **Excel-Oberfläche**, bestehend aus den Arbeitsblättern oder Tabellenblättern (worksheets) und den Registerkarten, in denen diverse Optionen zur Bearbeitung der Daten in den Arbeitsblättern zur Verfügung gestellt werden. Neben den Tabellenblättern gibt es noch die Diagrammblätter (chartsheets), aber diese haben in der Praxis geringere Bedeutung, weil Diagramme auch auf den Tabellenblättern dargestellt werden können.
- **Masken**, bei Microsoft auch **Formulare** genannt. Masken enthalten vor allem sog. Steuerelemente. Dazu gehören Felder zur Eingabe und Ausgabe, aber auch Elemente zum Einleiten von Verarbeitungsschritten, insbesondere Schaltflächen (Buttons). Auch Diagramme können in Masken gezeigt werden.

Im Übrigen können nicht nur Formulare Steuerelemente enthalten, sondern auch Tabellenblätter. Eine VBA-Anwendung mit Excel-Oberfläche enthält gewöhnlich mindestens ein Steuerelement, in den meisten Fällen ist es eine Schaltfläche, die der Benutzer betätigen kann, um die Berechnung auszulösen.

Überhaupt ist das UI einer typischen Excel-VBA-Anwendung ein gemischtes. Beispielsweise könnte ein Teil der Eingaben in einem Tabellenblatt geschehen und ein zweiter Teil in einer Maske; die Aus-

gaben könnten auf einem Tabellenblatt erscheinen. Welche Mischung sinnvoll ist, hängt immer von der Aufgabenstellung ab, zum Teil aber auch von den Präferenzen der Entwickler und der künftigen Benutzer. Anwendungen, deren UI nur aus Masken besteht, sind in Excel-VBA selten. Hierfür würde man eine andere Programmiersprache benutzen.

Um den optimalen Mix zu finden, müssen wir versuchen, die starken Seiten der beiden Hauptalternativen zu nutzen. Mit Masken kann man den Benutzer enger führen als mit Tabellenblättern. Tabellenblätter haben andererseits Vorteile, wenn die Eingabe oder Ausgabe großer Datenmengen nötig ist, oder wenn Daten aus anderen Anwendungen übernommen oder anderen Anwendungen zur Verfügung gestellt werden sollen.

3.2 Was ist ein gutes UI?

Von einem UI wünschen wir uns, dass es

- optimal auf die Aufgabe und die Arbeitsweise des typischen Benutzers zugeschnitten ist,
- leicht zu erlernen ist,
- Fehler vermeidet und robust auf Eingabefehler reagiert,
- motivierend wirkt.

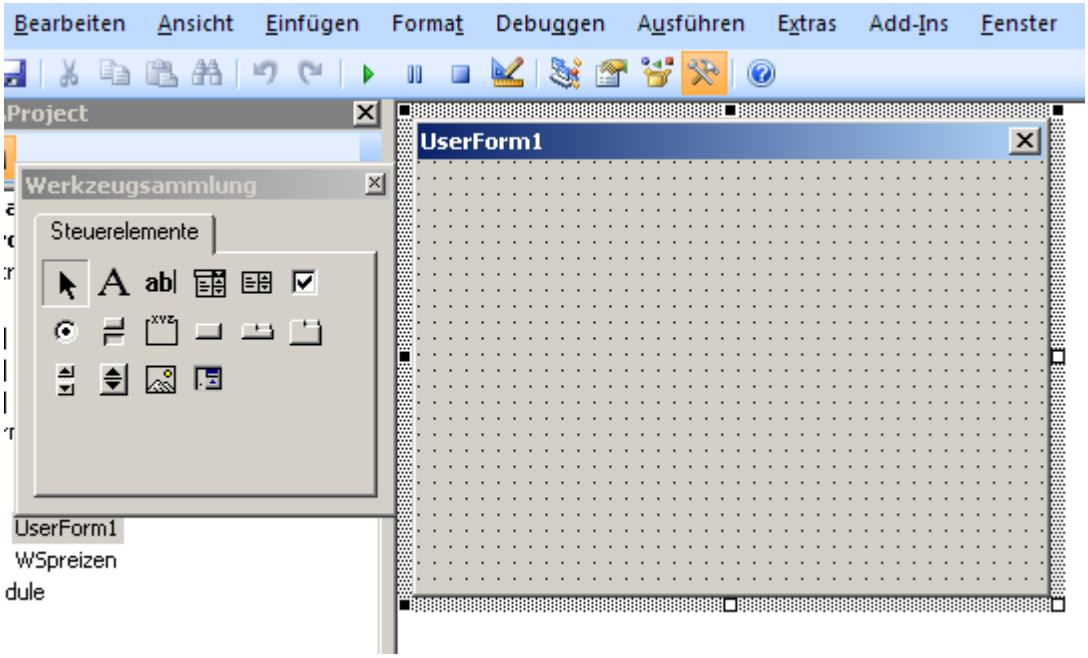
Diese Ziele sind unabhängig davon, welche der oben dargestellten Hauptformen des UI wir verwenden. Große Unterschiede gibt es aber hinsichtlich des konkreten Vorgehens, das wir als Entwickler beschreiten müssen, um die Ziele zu verwirklichen. Besonders groß sind die Unterschiede beim Ziel der Fehlervermeidung. Um sicher zu stellen, dass die Benutzer nur sinnvolle Eingaben machen, wenden wir bei Eingabe in ein Tabellenblatt die in Excel vorhandenen Möglichkeiten der Datenüberprüfung an, während wir für Maskeneingabe eigene Prüfprozeduren in VBA schreiben.

3.3 UI mit Formularen (Masken)

In diesem Hauptabschnitt wird zunächst gezeigt, wie man **Formulare anlegt und gestaltet**. Dann gehen wir auf das Prinzip der **ereignisgesteuerten Programmierung** ein, das die Verbindung zwischen den Handlungen des Benutzers und den Aktionen des Programms herstellt. Anschließend betrachten wir die verschiedenen **Methoden,ein Formular zu öffnen** und die daraus resultierenden Möglichkeiten der UI-Gestaltung. Danach werden einige **Prinzipien** präsentiert, die dem Entwickler helfen sollen, die im vorhergehenden Abschnitt propagierten Ziele der UI-Gestaltung zu verwirklichen. Es folgt dann ein Unterabschnitt, in dem gezeigt wird, wie man die Gestaltung der **Dialoge zwischen Benutzer und Formular planen** kann. Zum Abschluss beschäftigen wir uns mit der sog. **Eingabevali-dierung**, also damit, wie wir als Entwickler sicherstellen können, dass die Benutzer nur sinnvolle Eingaben machen und dass nur solche Eingaben vom UI akzeptiert werden.

3.3.1 Ein Formular anlegen und gestalten

Aus dem VBE legt man ein neues Formular an, indem man im Menü zuerst die Option „Einfügen“ wählt und danach die Option „UserForm einfügen“. Alternativ kann man gleich in der Kommandoleiste auf den Button <UserForm einfügen> klicken, falls dieser sichtbar ist. Es wird dann das gewünschte neue Formular geöffnet (s. Bild unten). Daneben erscheint ein kleines, mit „Werkzeugsammlung“ betiteltes Fenster. Es enthält die gängigsten Steuerelemente (Controls), die zur Gestaltung des Formulars zur Verfügung stehen.



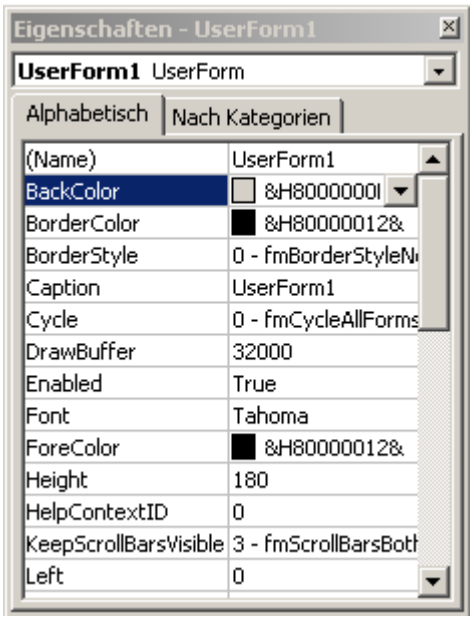
Das neue Formular ist zunächst mit einem generierten Namen und einer gleichlautenden Überschrift ausgestattet. Man sollte diese generierten Namen nie übernehmen, sondern gleich sinnvolle anwendungsbezogene Namen vergeben. Dies gilt auch für die einzelnen Steuerelemente, die auf dem Formular platziert werden.

Namen (Name), Überschrift (Caption) und alle sonstigen Eigenschaften ändert man im Eigenschaftsfenster, das man entweder mit F4 oder über den entsprechenden Button öffnen und beliebig auf dem Bildschirm platzieren kann (s. Bild unten).

Danach kann man beginnen, das Formular mit den benötigten Steuerelementen zu besetzen. Man klickt die Elemente in der Werkzeugsammlung an und kann sie dann anschließend in der gewünschten Größe auf das Formular setzen. Danach passt man wieder mit Hilfe des Eigenschaftsfensters den Namen und andere relevante Eigenschaften an.

Alle Namen von Formularen und Steuerelementen sollte man mit einem **Zusatz** versehen, der auf die Art des Elements schließen lässt. Dies kann ein Präfix oder, wie in den hier gezeigten Beispielen, ein Suffix sein (Beispiele: PersonenForm, SaveBtn, NameTBx). Solche Zusätze erleichtern die Programmierung des UI und helfen, Fehler zu vermeiden.

Die angezeigten Elemente der Werkzeugsammlung sind nur die am häufigsten gebrauchten. Man kann unschwer weitere Elemente hinzufügen, indem man mit der rechten Maustaste auf das Fenster der Werkzeugsammlung klickt und dann <zusätzliche Steuerelemente> wählt.



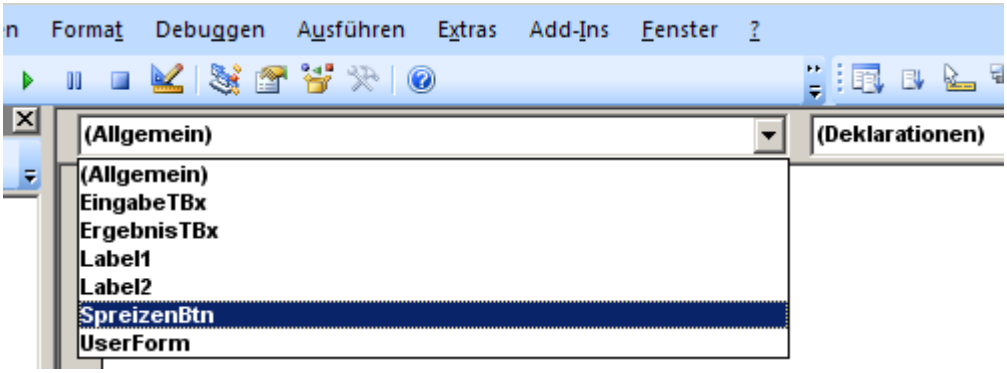
3.3.2 Das Prinzip der ereignisgesteuerten Programmierung

Wenn der Benutzer zur Laufzeit eines der Steuerelemente betätigt, dann löst er damit ein **Ereignis** aus. Für die wesentlichen Ereignisse, d.h. die, auf die das Programm reagieren soll, hat der Programmierer in dem Modul zum Formular jeweils eine **Ereignisprozedur** verfasst. Diese Prozedur wird gestartet, so bald das Ereignis eintritt. Damit reagiert das Programm auf die Aktion des Benutzers.

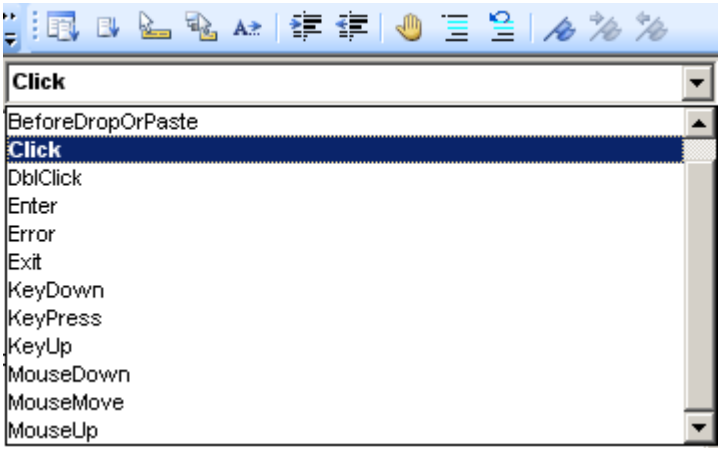
Anhand eines einfachen Beispiels (s. Bild unten) wird gezeigt, wie eine solche Ereignisprozedur angelegt wird. Der Zweck des Programms ist das Spreizen bzw. Sperren eines vom Benutzer einzugebenden Texts. Der Benutzer gibt zunächst den Text ein und drückt dann die unten befindliche Schaltfläche <Wort spreizen>. Erst nach diesem Klicken auf die Schaltfläche wird der gespreizte Text ausgegeben. Mit anderen Worten: das Klicken auf die Schaltfläche ist das auslösende Ereignis.

Zum Anlegen der Ereignisprozedur für das Anklicken der Schaltfläche muss man erst in die Ansicht wechseln, in der das Modul für das Formular angezeigt wird. Man kann diesen Wechsel in unterschiedlicher Weise vollziehen. Am einfachsten ist es, doppelt auf eine Stelle des Formulars zu klicken, an der sich kein Steuerelement befindet. Oder man klickt mit der rechten Maustaste auf den Formulareintrag im Projektexplorer und wählt dann „Code anzeigen“.

Oberhalb des noch leeren Moduls sieht man zwei Auswahlfelder (ComboBox). Aus dem linken wählt man zunächst das Steuerelement aus (hier SpreizenBtn).



Anschließend benutzt man das rechte Auswahlfeld („Deklarationen“), um das Ereignis auszuwählen, für das eine Prozedur geschrieben werden soll, in diesem Fall ist das „Click“.



Danach ist bereits der Rahmen für die Ereignisprozedur in den Modultext eingefügt. Der Programmierer braucht nur noch den Rumpf mit seinem Programmtext auszufüllen. Die fertige Prozedur für unser Beispiel lautet:

```
Private Sub SpreizenBtn_Click()
        Me.ErgebnisTBx.Text = StrBea.gespreizt(Me.EingabeTBx.Text)
End Sub
```

StrBea ist ein ebenfalls in der Arbeitsmappe befindliches Modul, das die Funktion gespreizt enthält. Diese Funktion fügt in einen beliebigen String jeweils ein Leerzeichen zwischen zwei Zeichen ein.

3.3.3 Formulardialoge starten und beenden

Man kann Formulardialoge auf ganz unterschiedliche Weise starten. Die ersten beiden sind eher für die Entwickler der Software geeignet, nicht aber für den Benutzer:

- Starten aus der VBA-Entwicklungsumgebung heraus über den Menüeintrag „Ausführen“. Diese Option wählt kann ein Entwickler wählen, um ein Formular auszuprobieren oder es zu testen.
- Starten aus der Registerkarte Entwicklertools heraus (Abschnitt Makros). Man kann ein Formular auf diesem Weg nicht direkt starten, man kann aber eine (selbst zu schreibende) Prozedur starten, die dann das Formular startet. Für den Endbenutzer einer von professionellen Entwicklern gefertigten Software ist dies nicht der beste Zugang, da erklärungsbedürftig und zu unbequem.

In beiden Fällen lässt sich das Formular leicht wieder schließen, indem man auf die mit einem Kreuz bebilderte quadratische Schaltfläche in der rechten oberen Ecke des Formulars klickt. Diese Schaltfläche ist in jedem Formular von vornherein vorhanden, der Entwickler muss sie also nicht explizit einfügen.

Der Endbenutzer einer Software, die ein anderer gefertigt hat, startet eine Maske, indem er die Software auf die Weise startet, die ihm vom Entwickler zu diesem Zweck eingerichtet worden ist. Die wichtigsten Möglichkeiten sind:

- automatisches Starten beim Öffnen der Arbeitsmappe
- durch Klicken auf eine vom Entwickler auf einem Arbeitsblatt platzierte Schaltfläche
- durch Klicken auf ein Symbol in einer Registerkarte (seit Excel 2007) oder einer Kommandoleiste, die sich über den Arbeitsblättern befindet. Auch diese Möglichkeiten müssen von den Entwicklern eingerichtet werden.

Bei größeren Programmen kommt es häufig vor, dass ein Benutzer von einem Formular zu einem anderen geleitet wird. Dieses Weiterleiten wird gewöhnlich durch das Betätigen einer bestimmten Schaltfläche im aktuellen Formular ausgelöst.Das zugrunde liegende Prinzip ist stets dasselbe, gleichgültig, ob es sich um die Startmaske oder eine Folgemaske handelt. Ein vom Benutzer ausgelöstes Ereignis (ein Klick) setzt eine vom Entwickler geschriebene Ereignisprozedur in Gang, welche die Maske öffnet.

Starten eines Formulars aus einer Prozedur heraus

Wir werden nun solche Prozeduren betrachten, mit denen Formulare gestartet werden. Wir benötigen hierfür aber einen kleinen Vorgriff auf die **objektorientierte Programmierung** (OOP), die erst in einem späteren Kapitel detailliert behandelt wird. Die Gestaltung und Durchführung von Formulardialogen ist in VBA nämlich nach dem Prinzip der OOP realisiert. Der Code, der zu einem Formular gehört, ist in einer **Klasse** (Klassenmodul) zusammengefasst, die als Grundlage zum Erzeugen von Formularen des betreffenden Typs dient. In den meisten Fällen benötigt man zu einem bestimmten Zeitpunkt nur ein Exemplar des Formulartyps, aber es ist durchaus möglich, auch mehrere Exemplare des Formulartyps gleichzeitig offen zu halten.

Wir betrachten zunächst als Beispiel eine Prozedur, welche ein Exemplar eines Formulars namens PersonenForm startet. Um diese Prozedur zu schreiben, legen wir zunächst ein Modul an und schreiben dann die Prozedur hinein:

```
Public Sub PersonenFormStart()
    Load PersonenForm
    PersonenForm.Show
End Sub
```

Nach der Ausführung des Befehls Load wird die Maske noch nicht angezeigt. Dies geschieht erst, wenn der Befehl Show ausgeführt wird. Wir könnten auf Load auch verzichten und gleich mit Show beginnen; das Laden würde dann automatisch vor dem Anzeigen des Formulars ausgeführt.

In den beiden Zeilen von PersonenFormStart wird jeweils für das Formular der Name verwendet, den der Entwickler dem Formular beim Zusammenstellen gegeben hat, nämlich PersonenForm. Dies ist eigentlich der Name der Klasse. Beachten Sie, dass der Zweck von PersonenFormStart aber nicht das Laden und Zeigen der Klasse, sondern eines Objekts der Klasse PersonenForm ist. PersonenForm in der obigen Prozedur bezeichnet also nicht die Formular-Klasse, sondern ein Formular-Objekt. Dies ist möglich, weil VBA standardmäßig das Objekt nach der Klasse benennt, wenn nur ein Formular benötigt wird. Wir werden weiter unten ein Beispiel betrachten, in dem mehrere Formulare derselben Formularklasse benutzt werden.

Formulare nichtmodal starten

Standardmäßig sind Formulare in VBA **modal**. Dies bedeutet, dass der Benutzer bei geöffnetem Formular nur in diesem Formular arbeiten kann, nicht in einem anderen Formular und auch nicht in einem Arbeitsblatt. Erst nach der Schließung des Formulars kann er an anderer Stelle weiterarbeiten.

Es ist aber durchaus möglich, Formulare **nichtmodal** zu öffnen, so dass man das Formular nicht schließen muss, um etwas in einem Arbeitsblatt nachzuschlagen oder dort etwas einzugeben. Man müsste die obige Prozedur hierfür folgendermaßen umformulieren. Beachten Sie den zusätzlichen Parameter vbModeless in der zweiten Anweisung.

```
Public Sub PersonenFormStartNichtModal()
    Load PersonenForm
    PersonenForm.Show   vbModeless
End Sub
```

Mehrere Formularexemplare zugleich offen halten

Nichtmodale Formulare können wir dann auch verwenden, wenn wir aus irgendwelchen Gründen in **mehreren Formularen gleichzeitig** arbeiten wollen. Das folgende Beispiel zeigt, wie zwei Exemplare des Personenformulars, d.h. zwei Objekte der Klasse PersonenForm, gleichzeitig geöffnet werden können. Hier können wir natürlich nicht mehr mit dem Standardnamen PersonenForm für die beiden Objekte arbeiten, denn wir müssen sie ja voneinander unterscheiden können. Wir geben den beiden Variablen die Namen pf1 und pf2.

```
Public Sub PersonenFormStartZwei()
    Dim pf1 As PersonenForm
```

```
        Dim pf2 As PersonenForm
        Set pf1 = New PersonenForm
        Set pf2 = New PersonenForm
        pf1.Show   vbModeless
        pf2.Show   vbModeless
End Sub
```

Die dargestellte Technik mit der expliziten Zuweisung eines Formularexemplars an eine Variable ist im Übrigen auch dann zu empfehlen, wenn nur ein Exemplar des Formulars gebraucht wird. Man hat damit bessere Kontrolle darüber, wann das Formular geöffnet und geschlossen wird. Dies macht die Anwendung robuster.

Schließen eines Formulars über eine spezielle Schaltfläche

Wie bereits erwähnt, kann ein Formular über die jederzeit verfügbare Schaltfläche in der rechten oberen Ecke des Formularfensters geschlossen werden. Diese Möglichkeit, ein Formular zu jedem beliebigen Zeitpunkt des Programmablaufs zu schließen, birgt bei vielen Programmen aber beträchtliche Gefahren. Schließt nämlich der Benutzer das Formular zu einem ungünstigen Zeitpunkt, z.B. wenn erst die Hälfte der notwendigen Bearbeitungsschritte vollzogen ist, so kann nicht gewährleistet werden, dass das Programm mit korrekten Ergebnissen und Daten beendet wird.

Um dies zu vermeiden, kann der Entwickler die Standard-Schaltfläche zum Schließen deaktivieren und durch eine selbst installierte Schaltfläche zum Schließen des Formulars bzw. Beenden des Programms ersetzen. Diese Schaltfläche wird nur in den Situationen aktiviert, in denen das Schließen des Formulars gefahrlos möglich ist. Wie dies im Einzelnen gemacht wird, wird weiter unten gezeigt.

3.3.4 Prinzipien zur Gestaltung der Benutzerschnittstelle

Im Abschnitt 3.2 haben wir uns bereits damit beschäftigt, was ein gutes UI ausmacht. Demnach soll ein UI optimal auf die Arbeitsweise des Benutzers zugeschnitten und leicht zu erlernen sein, es soll Fehler zu vermeiden helfen und es soll motivierend wirken. Wie wir diese Ziele erreichen können, hatten wir zunächst offengelassen. Die folgenden Gestaltungsprinzipien helfen erfahrungsgemäß, die postulierten Ziele zu erreichen. Sie lauten:

- Einfachheit
- Struktur
- Einheitlichkeit
- Suggestivität
- Rückmeldung
- Exaktheit
- Ästhetik

Einige Anmerkungen dazu:

Einfachheit entsteht vor allem durch sorgsam geplante Beschränkung auf das unbedingt Notwendige; das Notwendige selbst wird beschränkt durch Förderung von Struktur, Einheitlichkeit und Suggestivität.

Struktur gewinnt man durch inhaltliche Gruppierung der Bildschirmelemente, Unterstützung der inhaltlichen Gruppierung durch geeignete räumliche Anordnung (Abstände und Ballung), GroupBoxes, Linien, Aufteilung in Sektoren. Die Struktur muss auch auf den Arbeitsablauf des Benutzers zugeschnitten sein.

Einheitlichkeit bedeutet, dass man Gleiches gleich behandelt. Beispielsweise sollte man für eine bestimmte Funktion in den einzelnen Formularen eines Programms stets dieselbe Art von Schaltfläche verwenden (dieselbe Größe, dasselbe Symbol) und diese Schaltfläche sollte stets etwa dieselbe Position haben. Es ist auch sinnvoll, das Symbol zu benutzen, das sich im Lauf der Zeit für die betreffende Funktion eingebürgert hat.

Suggestivität bedeutet, die Bedienung zu erleichtern, indem man die Intuition des Benutzers anspricht (durch Hervorhebungen, Sinnbilder, passende Steuerelemente, ...)

Rückmeldung bewirkt, dass der Benutzer die Effekte seiner Handlungen sieht und stets weiß, was von ihm erwartet wird. Rückmeldungen müssen nicht immer aus Texten bestehen. Auch deutlich sichtbare **Dialogzustände** sind Rückmeldungen.

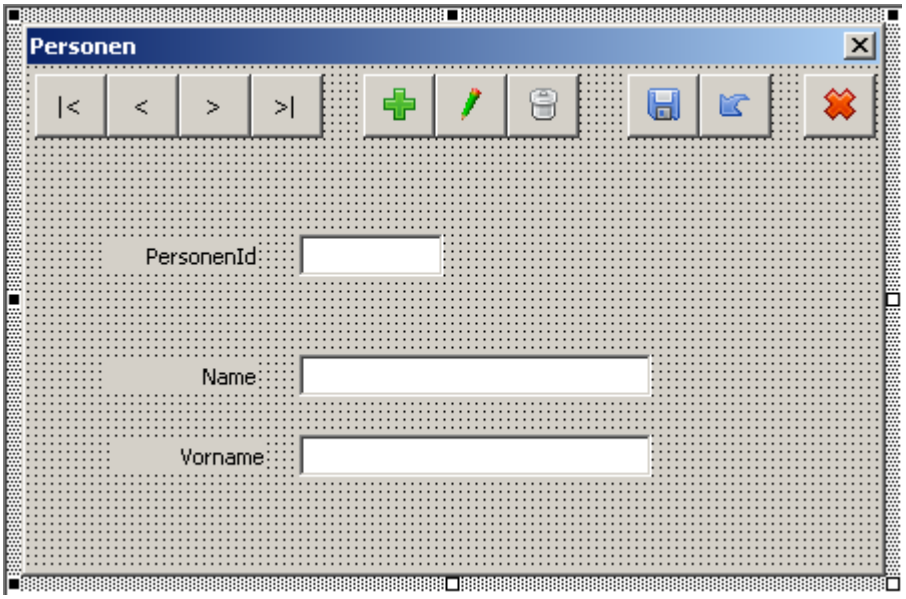
Exaktheit bedeutet, den Benutzer die Daten in der Form und Genauigkeit eingeben zu lassen, die benötigt wird und ihm dabei zeitraubende Fehler zu ersparen. Auch die Ausgabe soll genau auf die Bedürfnisse zugeschnitten sein. Exaktheit wird gefördert durch passende Eingabeformen, Formatvorgaben und Benennungen.

Ästhetik verlangt nach Harmonie, Struktur und Einfachheit. Eine ästhetische Benutzerschnittstelle ist eine, die dauerhaft gefällt. Bunt durcheinander gewürfelte Bildschirmelemente, grelle Farben und zappelnde Objekte sind Todsünden gegen die Ästhetik.

3.3.5 Planung von Dialogen

Mit Hilfe von Formularen lässt sich der Benutzer sehr viel enger und sicherer führen als mit Tabellenblättern alleine. Dies setzt aber voraus, dass man die möglichen Zustände eines Dialogs plant und konsequent realisiert. Wir betrachten hier zunächst nur die Planung. Die Programmierung, die dieser Planung folgt, wird dann am Ende des nächsten Abschnitts behandelt.

Die folgende Maske (im Entwurfsmodus) soll zum Betrachten und zur Pflege von Personendaten dienen:



Die ersten vier Buttons auf der linken Seite dienen dem Navigieren zwischen den Datensätzen. Darauf folgen die Buttons zum Anlegen eines neuen Satzes (AddBtn), zum Ändern eines vorhandenen Satzes (ChangeBtn) und zum Löschen eines Satzes (DeleteBtn). Die folgende Gruppe besteht aus den Buttons zum Speichern von Eingaben (SaveBtn) und zum Verwerfen einer Eingabe (CancelBtn). Ganz rechts dann der Button zum Schließen der Maske (CloseBtn).

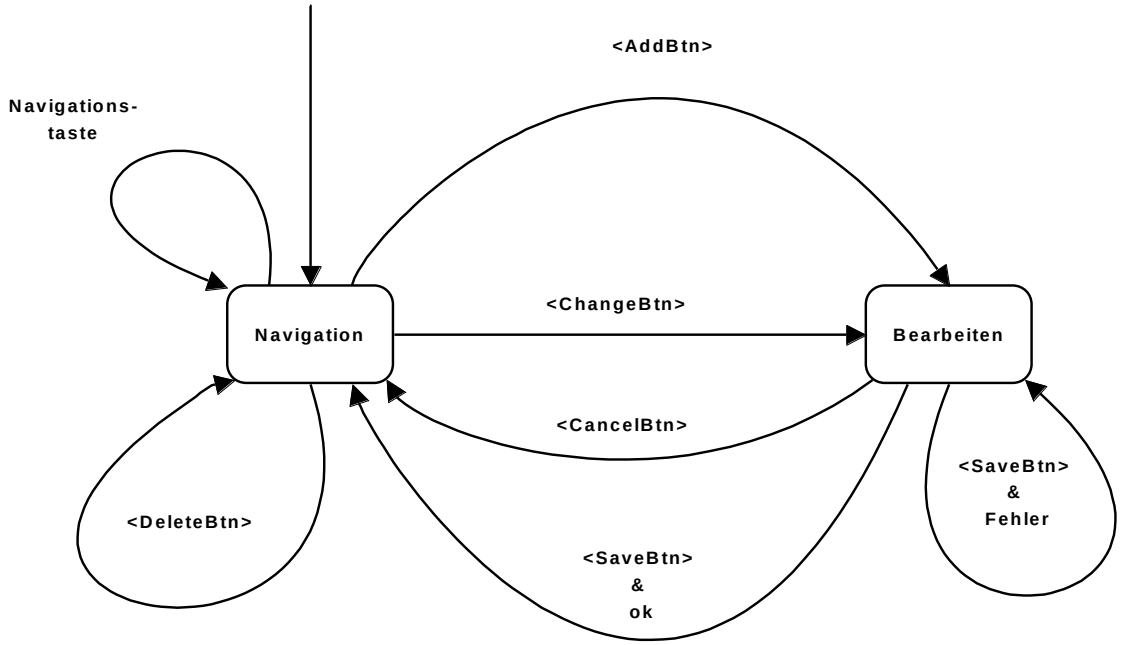
Die Buttons sind mit Hilfstexten (ToolTips) unterlegt. Diese werden sichtbar, wenn man zur Laufzeit mit dem Mauszeiger auf die Buttons deutet:

Indem man dem Benutzer jeweils nur die Bildschirmelemente anbietet, die in der jeweiligen Situation sinnvoll sind, erspart man ihm Irrwege und vermeidet Fehler.

Dialogzustände und Zustandsübergänge

Zunächst plant man die relevanten Zustände des Dialogs mit Hilfe eines **Zustandsdiagramms** (s. unten). Die **Zustände** sind als Kästchen mit abgerundeten Ecken dargestellt. Pfeile repräsentieren Übergänge zwischen den Zuständen. Sie sind mit dem Ereignis beschriftet, das den Zustandsübergang auslöst. Wird z.B. im Zustand Navigation die Änderungstaste (<ChangeBtn>) gedrückt , so geht das System in den Zustand Bearbeiten über. Mit anderen Worten, es erlaubt dem Benutzer, den aktuellen Datensatz zu bearbeiten. In manchen Fällen ist der Effekt eines Ereignisses auch von weiteren Faktoren abhängig. Drückt der Benutzer im Zustand Bearbeiten die Taste „Speichern“ (<SaveBtn>), so gibt es zwei Möglichkeiten: sind die eingegebenen Daten korrekt, so wird in den Zustand Navigation übergegangen, im anderen Fall verbleibt das System im Zustand Bearbeiten, so dass der Benutzer seine Eingabefehler korrigieren kann.

.



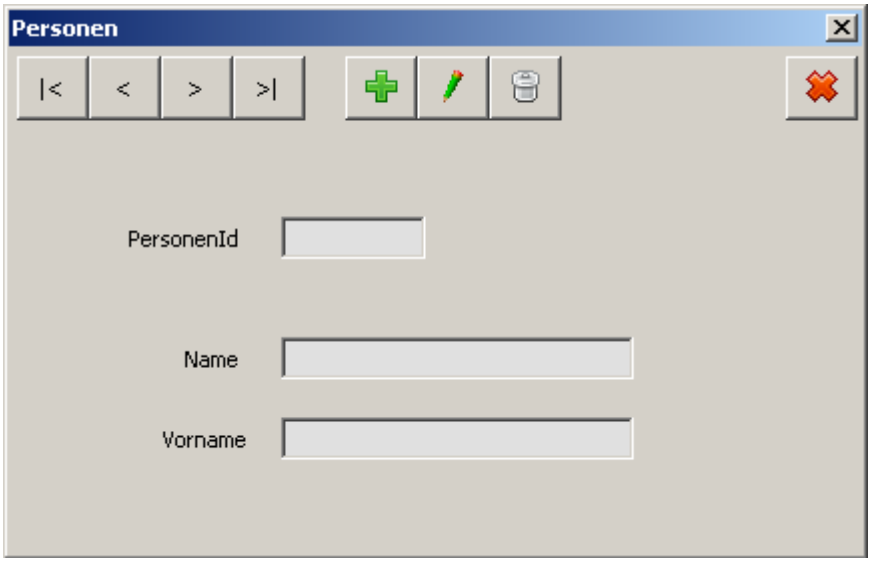
.

Für jeden Zustand wird festgelegt, wie sich die Maske dem Benutzer präsentiert. Im Wesentlichen geht es dabei um die Aktivierung und Deaktivierung von Formularelementen. Die folgende Tabelle fasst die Festlegungen zusammen. A bedeutet, dass das betreffende Element aktiviert ist, N steht für Nichtaktivierung (Sperrung).

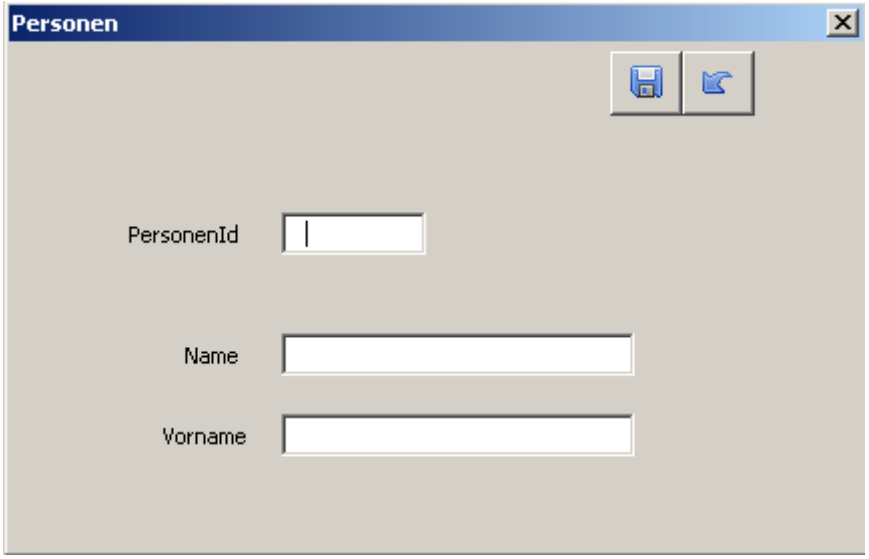
Maskenelement	**Navigation**	**Bearbeiten**
Navigationsbuttons	A	N
Eingabefelder	N	A
AddBtn	A	N
ChangeBtn	A	N
DeleteBtn	A	N
SaveBtn	N	A
CancelBtn	N	A
CloseBtn	A	N

Das folgende Bild zeigt die Maske im Zustand Navigation. Die Eingabefelder sind gesperrt (LOCKED) und grau unterlegt; die Schaltflächen für das Speichern und das Verwerfen der Eingabe sind ebenfalls gesperrt und darüber hinaus unsichtbar.

Alternativ hätte man auch die Eingabefelder deaktivieren können (ENABLED = FALSE). Dies ist dem völligen Verstecken der Felder im Prinzip sogar vorzuziehen, weil dem Benutzer mehr Orientierung gegeben wird. Das Verstecken wurde hier nur gewählt, weil die Deaktivierung nicht gut sichtbar war.



Im Zustand der Bearbeitung hat der Benutzer nur die Buttons zum Speichern und zum Verwerfen der Eingabe zur Verfügung:



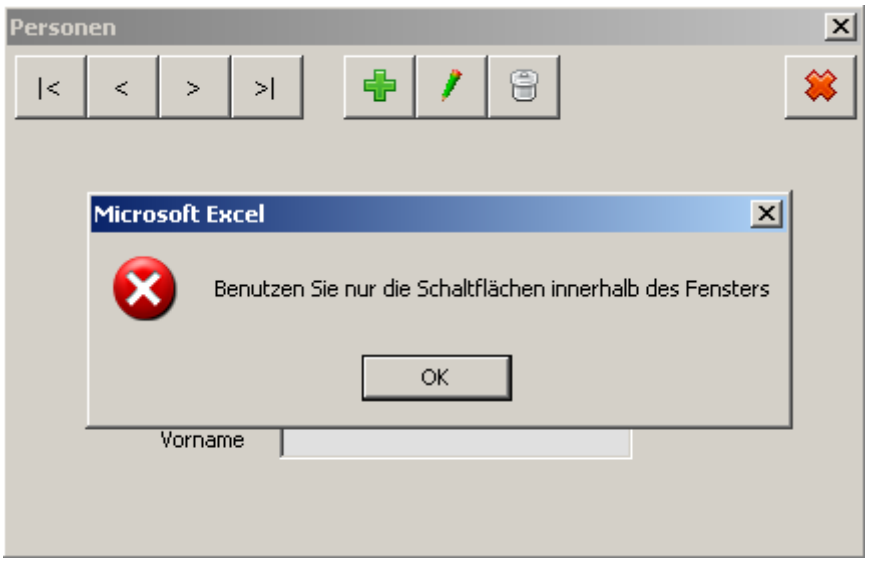
Die Fluchtmöglichkeit über die ControlBox (X) am rechten Ende der Fensterüberschrift ist ebenfalls versperrt. Der Benutzer soll nur die Buttons im Inneren des Fensters benutzen. Versucht er, das Fenster über die ControlBox zu schließen, wird eine entsprechende Meldung präsentiert und er wird zum Hauptformular zurück geführt (Bild unten).

Zur Realisierung der Zustände im Programm fügt man für jeden Zustand eine Sub-Prozedur ein, welche für jedes einzelne Maskenelement den gewünschten Zustand festlegt. Diese Prozeduren, und wie sie aufgerufen werden, zeigt der Code weiter unten.

Die ControlBox unschädlich machen

Dem Benutzer steht in Windows standardmäßig eine Schaltfläche zum Schließen des Fensters in der Fensterüberschrift zur Verfügung (X). Diese Fluchtmöglichkeit muss man versperren, wenn man vermeiden will, dass der Benutzer durch vorzeitiges Schließen Fehler verursacht. Am besten deaktiviert

man diese ControlBox gleich beim Laden des Formulars (s. Code weiter unten). Das folgende Bild zeigt die Reaktion des Programms auf einen Versuch, das Fenster über die ControlBox zu verlassen:



3.3.6 Simple Validierung von Eingaben

Vor dem Weiterverarbeiten und dem Speichern von eingegebenen Daten müssen diese auf ihre Richtigkeit hin überprüft werden. Dies ist eine aufwendige Programmieraufgabe, die nicht selten den größten Teil des Programmtexts ausmacht. Hier sei nur eine einfache Lösung gezeigt, und auch diese nur fragmentarisch (s. Code weiter unten). Bei einer sehr kleinen Maske genügt es, die Richtigkeit der Daten summarisch nach dem Klicken der Speichern-Schaltfläche zu überprüfen. Bei umfangreichen Eingaben empfiehlt sich zusätzlich die Überprüfung nach Verlassen der einzelnen Eingabefelder. Darauf wird in einem späteren Abschnitt eingegangen.

Wir wollen annehmen, dass es im vorliegenden Fall die Prüfung genügt, ob der Benutzer für die zu erfassende Person eine PersonenId eingegeben hat. Falls nicht, soll eine MsgBox mit einem entsprechenden Hinweis gezeigt werden.

Wir betrachten nun den Code des Formularmoduls PersonenForm. Der Code ist insofern unvollendet, als wir mit dem Formular nicht wirklich Daten verwalten können. Außer der Benutzeroberfläche ist vom Programm noch nichts vorhanden. Der Benutzer kann jedoch Daten eingeben und Buttons anklicken, und das Formular wird sich so verhalten, wie wir es oben mit Hilfe des Zustandsübergangs geplant haben. Auch die oben beschriebene Eingabevalidierung wird durchgeführt.

Realisierung der Dialogzustände mit Zustandsprozeduren

Der Code enthält fragmentarische Ereignisprozeduren für die wesentlichen Ereignisse, also das Anklicken der Schaltflächen.

Für jeden der beiden Zustände (Navigation und Bearbeitung) ist im Code eine Prozedur enthalten, welche diesen Zustand herstellt, hier also die gebrauchten Elemente aktiviert und sichtbar macht, die

nicht gebrauchten aber deaktiviert und unsichtbar macht. Die Zustandsprozeduren werden von den Ereignisprozeduren aufgerufen, denn das Anklicken von Schaltflächen ist der Anlass für Zustandsübergänge. Beim Laden des Formulars Prozedur wird der Zustand Navigation mit Hilfe der Prozedur UserForm_Initialize eingestellt. Diese Prozedur wird beim Laden automatisch ausgeführt.

Realisierung der Eingabevalidierung mit einer speziellen Funktion

Mit der Funktion ValidierungOK wird überprüft, ob der Benutzer eine PersonenId eingegeben hat. Ist dies nicht der Fall, so gibt die Funktion eine Fehlermeldung in einer MsgBox aus und liefert den Wert False an die aufrufende Stelle. Die aufrufende Stelle ist hier die Ereignisprozedur, die beim Anklicken der Schaltfläche Speichern ausgelöst wird (SaveBtn_Click). Nur wenn die Eingabe nicht beanstandet wird, liefert ValidierungOK den Wert True und nur dann wird nach dem Anklicken der Speichern-Schaltfläche der Zustand Navigation hergestellt. Bei fehlerhafter Eingabe wird der Zustand Bearbeiten nicht verlassen, so dass der Benutzer die fehlende Eingabe nachholen kann.

Beachten Sie, dass ValidierungOK ein ganz anderer Typ von Funktion ist, als Sie es bisher gewohnt waren. Dass eine Funktion eine Meldung ausgibt, ist ungewöhnlich, und wir wollen dies auch auf Funktionen beschränken, die wir speziell für die Eingabevalidierung schreiben. Bei allen anderen Funktionen würden Ausgaben irgendwelcher Art die Wiederverwendung verhindern.

Fragmentarischer Code des Beispiels

```
Option Explicit

Private Sub UserForm_Initialize()
        ZustandNavigation
End Sub

'verhindert das Schließen des Formulars über die ControlBox
Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
        If  CloseMode = vbFormControlMenu  Then
            MsgBox "Benutzen Sie nur die Schaltflächen innerhalb des Fensters",  _
                    vbCritical
            Cancel = True
        End If
End Sub

Private Sub ZustandNavigation()
        Me.FirstBtn.Enabled = True
        Me.PreviousBtn.Enabled = True
        Me.NextBtn.Enabled = True
        Me.LastBtn.Enabled = True
        Me.AddBtn.Enabled = True
        Me.ChangeBtn.Enabled = True
        Me.DeleteBtn.Enabled = True
        Me.SaveBtn.Enabled = False
        Me.CancelBtn.Enabled = False
        Me.CloseBtn.Enabled = True

        Me.FirstBtn.Visible = True
```

```
        Me.PreviousBtn.Visible = True
        Me.NextBtn.Visible = True
        Me.LastBtn.Visible = True
        Me.AddBtn.Visible = True
        Me.ChangeBtn.Visible = True
        Me.DeleteBtn.Visible = True
        Me.SaveBtn.Visible = False
        Me.CancelBtn.Visible = False
        Me.CloseBtn.Visible = True

        Me.PersonIdTBx.Locked = True
        Me.NameTBx.Locked = True
        Me.VornameTBx.Locked = True

        Me.PersonIdTBx.BackColor = &HE0E0E0
        Me.NameTBx.BackColor = &HE0E0E0
        Me.VornameTBx.BackColor = &HE0E0E0
End Sub

Private Sub ZustandBearbeiten()
        Me.FirstBtn.Enabled = False
        Me.PreviousBtn.Enabled = False
        Me.NextBtn.Enabled = False
        Me.LastBtn.Enabled = False
        Me.AddBtn.Enabled = False
        Me.ChangeBtn.Enabled = False
        Me.DeleteBtn.Enabled = False
        Me.SaveBtn.Enabled = True
        Me.CancelBtn.Enabled = True
        Me.CloseBtn.Enabled = False

        Me.FirstBtn.Visible = False
        Me.PreviousBtn.Visible = False
        Me.NextBtn.Visible = False
        Me.LastBtn.Visible = False
        Me.AddBtn.Visible = False
        Me.ChangeBtn.Visible = False
        Me.DeleteBtn.Visible = False
        Me.SaveBtn.Visible = True
        Me.CancelBtn.Visible = True
        Me.CloseBtn.Visible = False

        Me.PersonIdTBx.Locked = False
        Me.NameTBx.Locked = False
        Me.VornameTBx.Locked = False

        Me.PersonIdTBx.BackColor = &HFFFFFF
        Me.NameTBx.BackColor = &HFFFFFF
        Me.VornameTBx.BackColor = &HFFFFFF
End Sub

Private Sub AddBtn_Click()

        ZustandBearbeiten
End Sub
```

```
Private Sub CancelBtn_Click()

    ZustandNavigation
End Sub

Private Sub ChangeBtn_Click()

    ZustandBearbeiten
End Sub


Private Sub CloseBtn_Click()

    Unload Me
End Sub

Private Sub DeleteBtn_Click()

    ZustandNavigation
End Sub

Private Sub FirstBtn_Click()

    ZustandNavigation
End Sub

Private Sub LastBtn_Click()

    ZustandNavigation
End Sub


Private Sub NextBtn_Click()

    ZustandNavigation
End Sub

Private Sub PreviousBtn_Click()

    ZustandNavigation
End Sub

Private Sub SaveBtn_Click()
    If Not ValidierungOk Then Exit Sub
    ZustandNavigation
End Sub

Private Function ValidierungOk() As Boolean
    If  Me.PersonIdTBx.Text = "" Then
        MsgBox "Sie müssen eine PersonenId eingeben"
        ValidierungOk = False
        Exit Function
    End If
    ValidierungOk = True
```

```
End Function
```

3.3 UI auf Tabellenblättern (Worksheets)

Wir beschäftigen uns in diesem Hauptabschnitt zunächst mit der Gestaltung von UI, die auf Tabellenblättern eingerichtet werden. Dann wird auf die Validierung der Eingaben eingegangen, die hier vorwiegend mit Excel-Mitteln geschieht.

3.4.1 Designprinzipien und Zellenformatierung

Die Prinzipien zur Gestaltung der Benutzerschnittstelle, die im Zusammenhang mit Formularen vorgestellt wurden, gelten ebenso für UI auf Tabellenblättern. Es kommen jedoch noch einige konkrete Ratschläge hinzu, die sich aus den besonderen Gegebenheiten der Tabellenblätter ergeben:

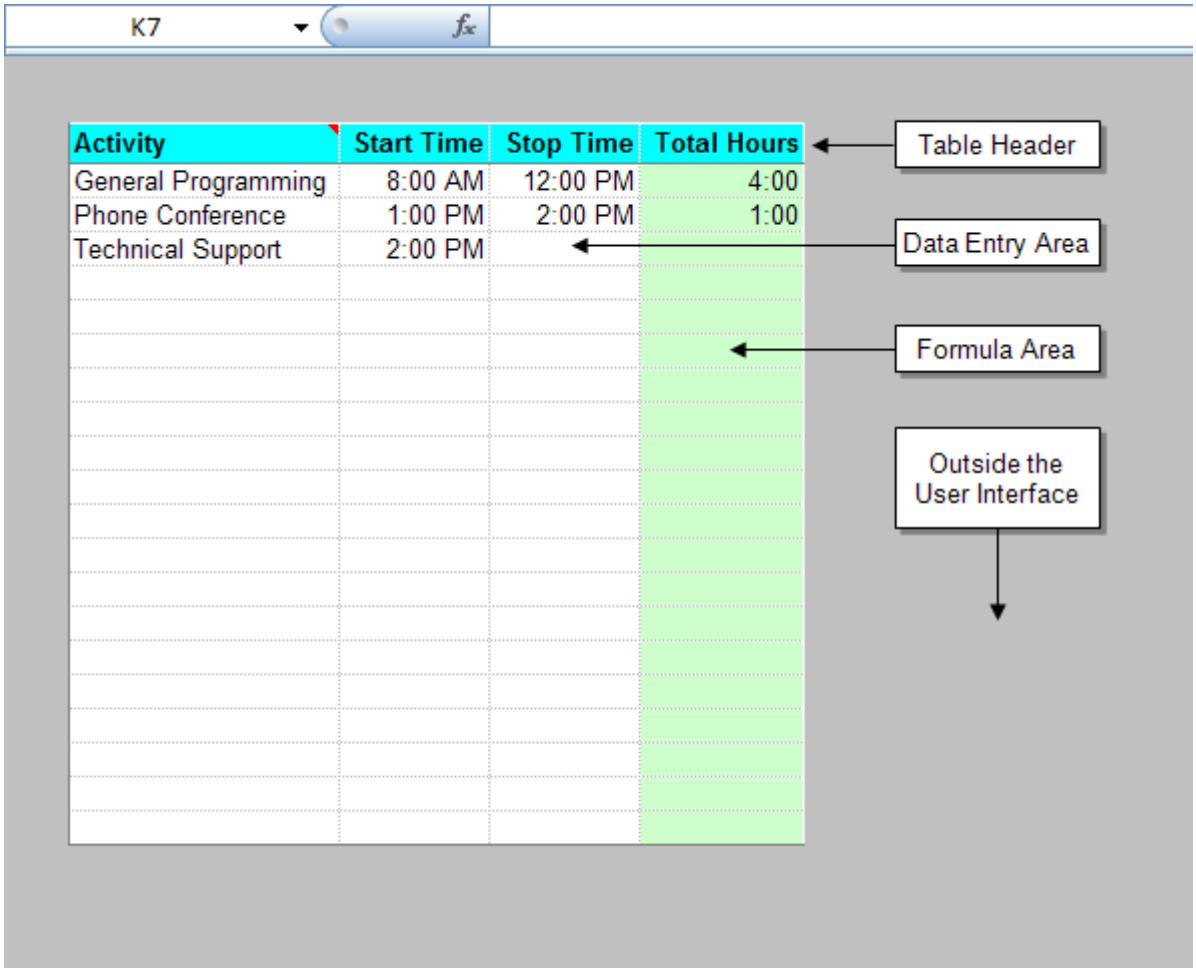
1. Visuellen Kontrast zwischen Zellen unterschiedlichen Zwecks herstellen und gewählte Zellstile in der ganzen Arbeitsmappe beibehalten (Konsistenz der Stile)
2. Keine grellen Farben verwenden
3. Logischen Arbeits- und Lesefluss herstellen. Den Benutzer nicht wahllos hin- und herspringen lassen. Anwendung auf Arbeitsblätter verteilen, z.B. Eingabe auf erstem Blatt, Ausgabe/Präsentation auf zweitem, dahinter Hilfsdaten und Hilfsberechnungen.
4. Oberfläche gut strukturieren und übersichtlich gestalten.
5. Katastrophale Fehler des Benutzers verhindern. Zellen, die nicht für Eingaben gedacht sind, schützen.
6. Verhindern, dass der Benutzer sich verirrt. Den Arbeitsbereich begrenzen und hervorheben.

Das folgende Bild (Bovey et al.: Professional Excel Development) demonstriert, wie die Orientierung des Benutzers durch geeignete Formatierung verbessert werden kann und Fehler vermieden werden können. Einige aus den oben genannten Prinzipien können hier gut nachvollzogen werden:

- Visuellen Kontrast herstellen
- Konsistenz der Stile
- Katastrophale Fehler verhindern (alle Zellen außer den Eingabezellen sind geschützt)
- Verhindern, dass der Benutzer sich verirrt. (nicht benötigte Bereiche sind grau gefärbt)

Man kann eine solche Formatierung mit Hilfe der Formatierungsoptionen der Registerkarte Start realisieren. Dort sind auch schon Formatvorlagen für häufig gebrauchte Fälle verfügbar. Eine Formatvorlage ist eine benannte und gespeicherte Kombination von Formatierungsfestlegungen.

Es ist auch möglich, selbst Formatvorlagen zu erstellen, wenn die vorhandenen Standardvorlagen nicht ausreichen. Dies ist besonders im Rahmen eines größeren Projekts sinnvoll.



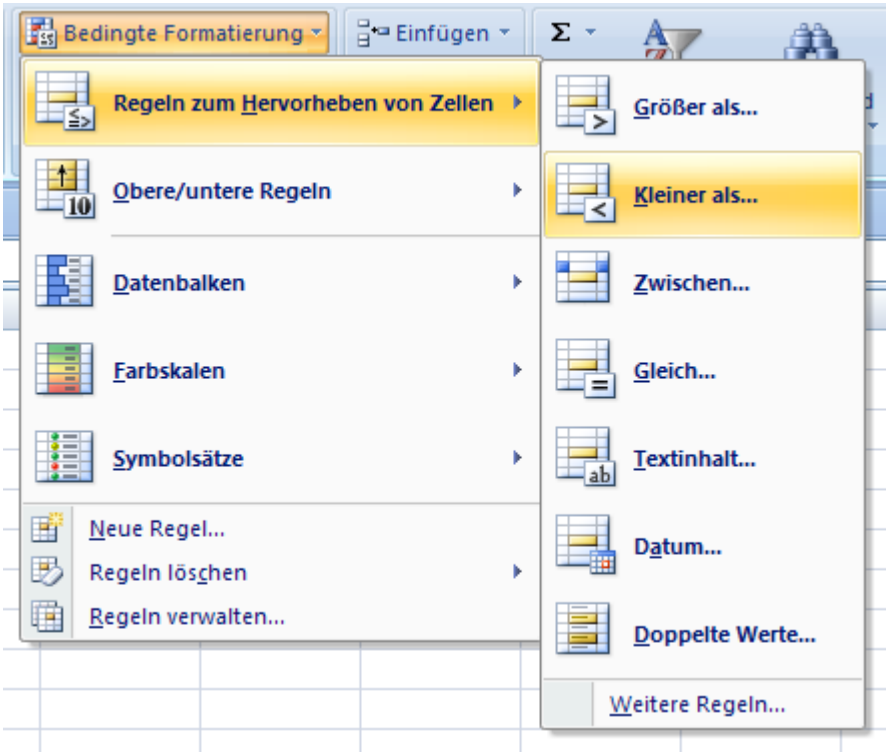
3.4.2 Bedingte Zellenformatierung

Bedingte Formatierung erlaubt, die Formatierung von Zellen und Bereichen an Bedingungen zu knüpfen. Sobald die Bedingungen erfüllt sind, nehmen die Zellen das vorgesehene Format an; eine etwaige vorherige Formatierung wird dabei überschrieben. Ist danach die Bedingung nicht mehr gegeben, wird das ursprüngliche Format wieder hergestellt.

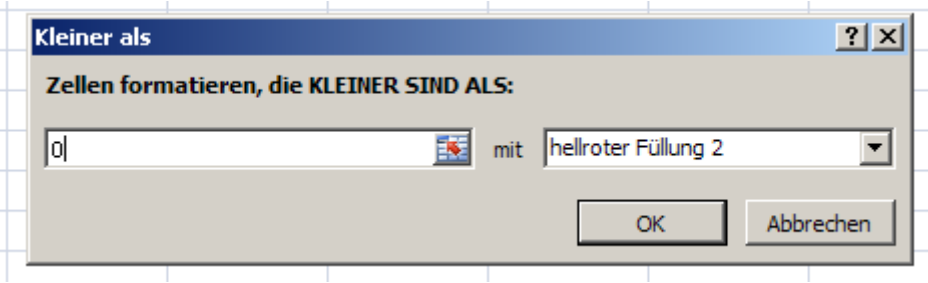
Häufige Anwendungen der bedingten Formatierung sind das Hervorheben von Fehlern und von unterschiedlichen Ergebniswerten (Gewinn oder Verlust, ...). Im folgenden Bild sind alle Zellen rot gekennzeichnet, die einen negativen Wert enthalten.

	A	B	C
1		Ergebnis	
2		100	
3		340	
4		-20	
5		-45	
6		300	
7			

Um diesen Effekt zu erreichen, markiert man den gesamten Bereich, in dem Werte stehen können. Dann ruft man den Dialog Bedingte Formatierung in der Registerkarte Start auf. Man wählt nacheinander „Regeln zum Hervorheben von Zellen“ und „Kleiner als ...“ (s. Bild unten).



Es wird nun ein kleines Fenster geöffnet (Bild unten), in das man den Schwellenwert (hier 0) und die Farbe der Markierung eingeben kann.

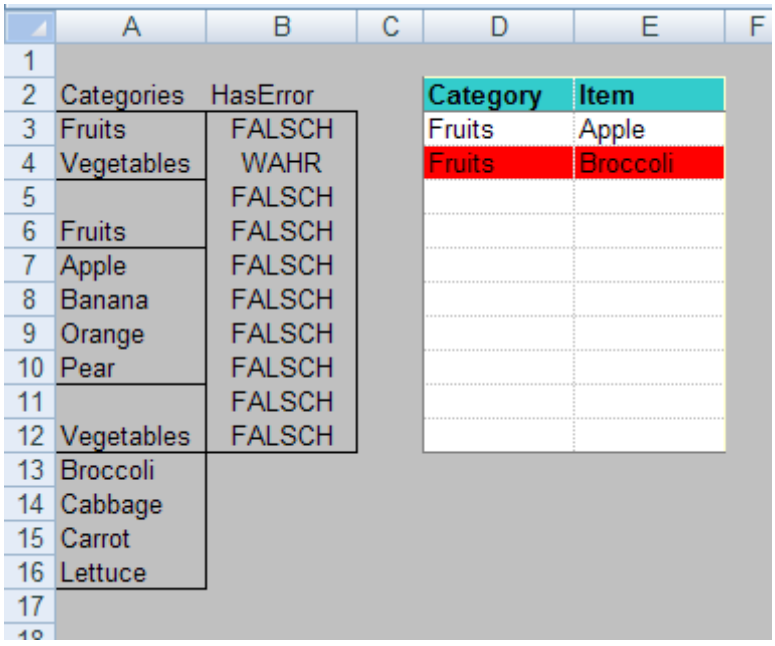


Ein aufwändigeres Beispiel bedingter Formatierung enthält das folgende Bild. Es zeigt einen Ausschnitt aus der Einsatzplanung eines Museums. Die Standardöffnungstage sind durch bedingte Formatierung farbig markiert, Wochenenden blau und Mittwoche grün. Außerdem weisen Symbole am rechten Rand darauf hin, wie der Planungsstand für den jeweiligen Tag ist. Ein grüner Haken markiert vollständig geplante Tage, ein gelbes Ausrufezeichen weist auf begonnene, aber unvollständige Planung hin, ein rotes Kreuz besagt, dass für den betreffenden Tag überhaupt noch niemand eingeplant ist.

Datum	Wochentag	Schließdienst	Aufsicht 1	Aufsicht 2
01.01.2010	Freitag			
02.01.2010	Samstag	Meier	Dinkel	
03.01.2010	Sonntag		Etzel	
04.01.2010	Montag			
05.01.2010	Dienstag			
06.01.2010	Mittwoch	Kunze	Berberich	Hofmann
07.01.2010	Donnerstag			
08.01.2010	Freitag			
09.01.2010	Samstag		Fabian	
10.01.2010	Sonntag		Ganter	Dinkel
11.01.2010	Montag			
12.01.2010	Dienstag			
13.01.2010	Mittwoch			
14.01.2010	Donnerstag			

3.4.3 Versteckte Bereiche

Hierunter versteht man Spalten am linken Rand eines Tabellenblatts bzw. Zeilen am oberen Ende des Tabellenblatts, die zur Speicherung von Hilfsdaten, zur Eingabevalidierung oder für Nebenrechnungen verwendet werden. Man nennt sie auch Programmspalten bzw. Programmzeilen, weil sie vom Programm genutzt werden (und nicht vom Benutzer). Den Benutzer würden sie nur verwirren. Sie werden deshalb vor Auslieferung der Software ausgeblendet (Registerkarte Start/Format).



Categories	HasError
Fruits	FALSCH
Vegetables	WAHR
	FALSCH
Fruits	FALSCH
Apple	FALSCH
Banana	FALSCH
Orange	FALSCH
Pear	FALSCH
	FALSCH
Vegetables	FALSCH
Broccoli	
Cabbage	
Carrot	
Lettuce	

Category	Item
Fruits	Apple
Fruits	Broccoli

Das Bild oben zeigt ein Tabellenblatt mit zwei Programmspalten (Quelle: Bovey et al.). Die linke enthält Listen mit zulässigen Eingaben, die rechte enthält Formeln, welche die Eingabe auf der Basis dieser Listen überprüfen.

3.4.4 Definierte Namen

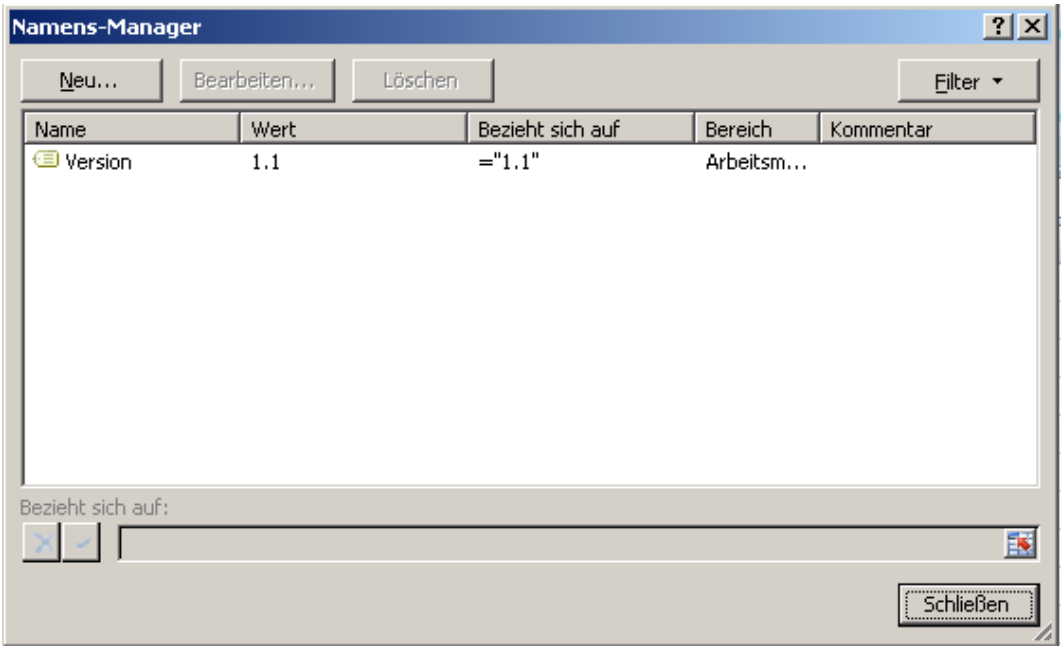
Namen können für Konstanten und Arbeitsblattbereiche vergeben werden. Außerdem sind zwei verschiedene Geltungsbereiche möglich: geltend für ein bestimmtes Tabellenblatt oder für eine ganze Arbeitsmappe. Definierte Namen machen die Arbeit mit Excel komfortabler und sie tragen auch dazu bei, Fehler zu vermeiden. Besondere Bedeutung haben sie im Zusammenhang mit der Eingabevalidierung (s. unten).

Benannte Konstanten

Sie beziehen sich auf einen konstanten Wert beliebigen Typs. Man kann Sie z.B. benutzen, um wichtige Kenndaten der Anwendung oder der Arbeitsmappe zu speichern, wie die Versionsnummer, die zugrunde liegende Version von Excel, usw.

Definieren eines Namens:
Registerkarte Formeln > Namen definieren > Namen definieren. Danach erscheint das oben abgebildete Fenster. Beachten Sie die Anführungszeichen um den eingegebenen Wert!

Ändern oder Entfernen von definierten Namen:
Registerkarte Formeln > Namensmanager. Danach erscheint das folgende Fenster:

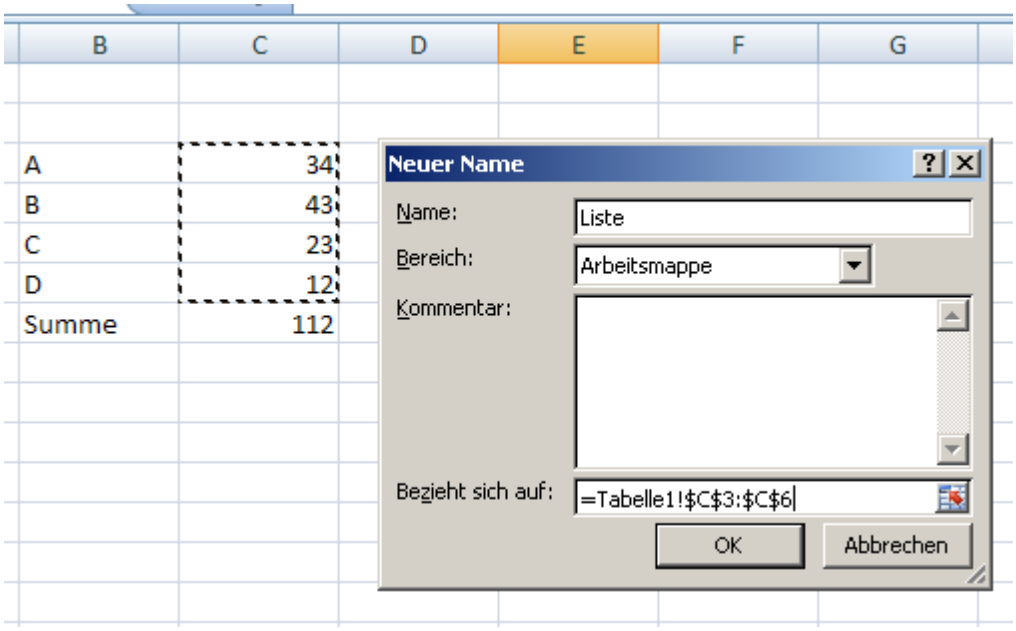


Benannte Arbeitsblattbereiche

Der Nutzen benannter Arbeitsblattbereiche liegt vor allem auf zwei Gebieten:

1. Sie vereinfachen das Gestalten der Anwendung, weil man einen inhaltlichen Bezug zu den relevanten Bereichen hat. Gleichzeitig wird damit das Gestalten sicherer, denn beim Eintippen eines Namens macht man nicht so leicht Fehler wie beim Eingeben eines Bezugs mit Zeilen- und Spaltenindizes.
2. Sie schaffen Unabhängigkeit vom aktuellen Zustand des Arbeitsblatts. Ein Verweis, der sich nicht auf einen festen Bereich bezieht, sondern auf einen Namen, muss nicht geändert werden, wenn sich der Bereich verschiebt, den der Name bezeichnet.

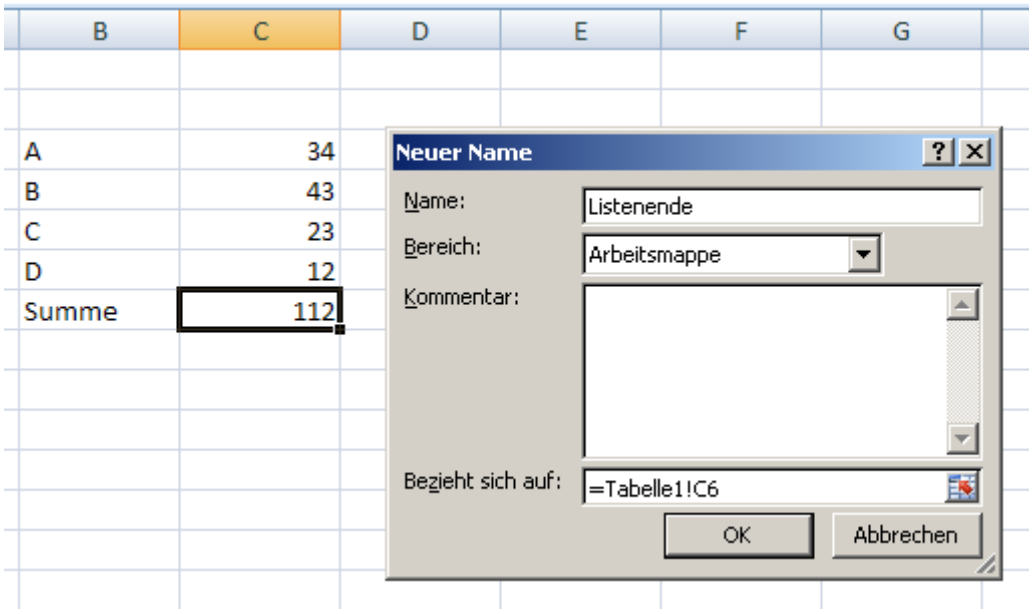
Benannte Bereiche können absolut oder relativ definiert werden. Das folgende Bild zeigt die Definition eines **absoluten** Bereichs. Man sieht es an den Dollarzeichen im Feld Bezieht sich auf. Nach der Definition des Namens kann dieser auch in Formeln verwendet werden. Man kann also z.B. in eine beliebige Zelle die Formel = Summe(Liste) platzieren.



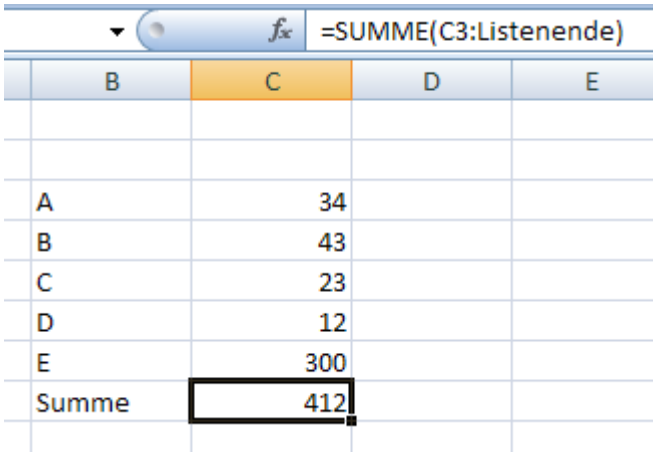
Relative benannte Bereiche sind solche, deren Lage in Bezug auf die Lage einer bestimmten Zelle bestimmt wird. Man definiert einen solchen benannten Bereich, indem man vor Aufruf des Dialogs einen Bereich bzw. eine Zelle markiert und im Eingabefeld „Bezieht sich auf“ des Dialogs die Dollarzeichen weglässt.

Im folgenden Beispiel ist der definierte relative Bereich die begrenzende, d.h. letzte Zelle eines zu summierenden Bereichs. Nehmen wir an, im obigen Tabellenblatt könnte der Benutzer weitere zu summierende Zeilen einfügen, die dann mit summiert werden sollen. Zwar versucht Excel in solchen Fällen, die Summenformel automatisch anzupassen, aber dies gelingt nicht in allen Fällen. Außerdem könnte es vorkommen, dass die entsprechende Option deaktiviert ist.

Man kann das Problem lösen, indem man einen Namen für das Ende des zu summierenden Bereichs definiert und in der Summenformel diesen Namen verwendet. Die letzte zu summierende Zelle ist im Beispiel die Zelle über der Summe. Wir vergeben also für sie einen Namen. Das unten stehende Bild zeigt, wie dies geschieht. Vor dem Aufruf des Namensvergabedialogs markiert man die Summenzelle. Sie bildet den Bezugspunkt des relativen Bezugs. Im dann aufgerufenen Dialog vergibt man den Namen („Listenende“) und gibt bei „Bezieht sich auf“ die Zelle über der Summenzelle an (C6), lässt aber die Dollarzeichen weg.



Nun muss man nur noch die Summenformel anpassen. Man gibt als Obergrenze den vergebenen Namen Listenende an. Listenende ist stets die Zelle direkt über der Summe, gleichgültig, wie viele Zeilen noch eingefügt werden.

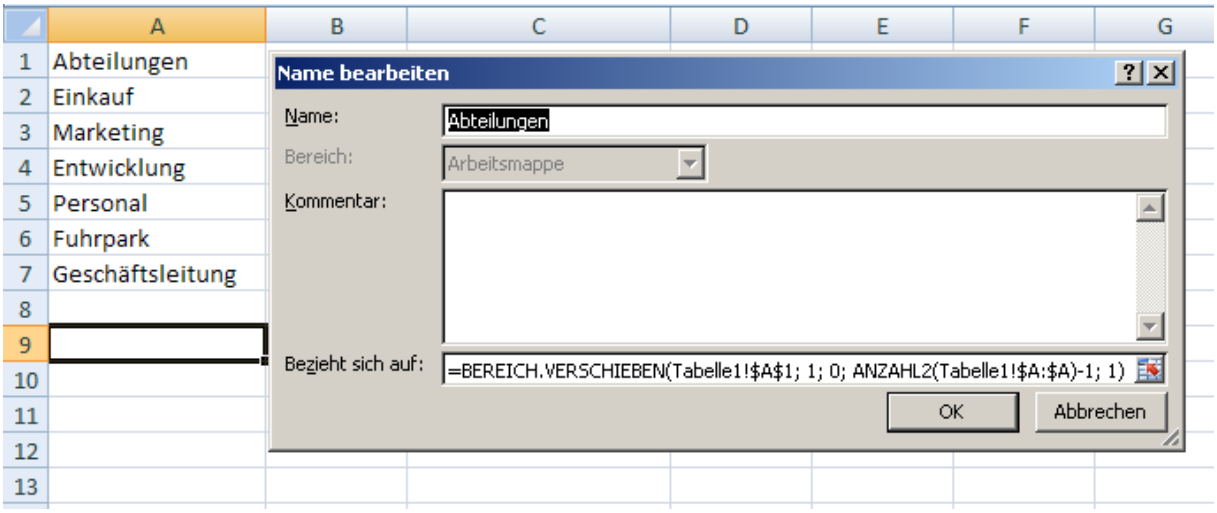


Benannte dynamische Bereiche

Eine nützliche Anwendung definierter Namen ist auch die Definition **dynamischer Bereiche**. Dies sind Bereiche, welche im Verlauf der Arbeitsblattbenutzung ihre Ausdehnung ändern können. Man definiert einen solchen Bereich, indem man eine Formel definiert, welche den Bereich liefert.

Mit solchen benannten Formeln können nahezu beliebig lange Rechnungen unter einem Namen zusammengefasst werden. Auch benannte Formeln können **absolut** oder **relativ** definiert werden.

Das unten stehende Bild zeigt, wie ein dynamischer Bereich definiert werden kann, der eine Liste von Abteilungsnamen enthält. Der vergebene Name („Abteilungen“) bezeichnet stets die gesamte Abteilungsliste ohne die Überschrift „Abteilungen“, auch wenn noch weitere Abteilungen hinzugefügt werden oder Einträge aus der Liste herausgenommen werden. Beachten Sie, dass das Beispiel voraussetzt, dass sich außer der Abteilungsliste in der Spalte A keine weiteren Einträge befinden.



Die eingegebene Formel verwendet die beiden Funktionen BEREICH.VERSCHIEBEN und ANZAHL2. ANZAHL2 liefert für einen Bereich die Anzahl der Zellen, in denen sich Werte befinden. BEREICH.VERSCHIEBEN (in der englischen Version OFFSET) liefert einen Bereich, der sich durch eine vorgegebene Bewegung von einem Startpunkt aus ergibt. Die Funktion hat 5 Parameter. Von links nach rechts sind dies:

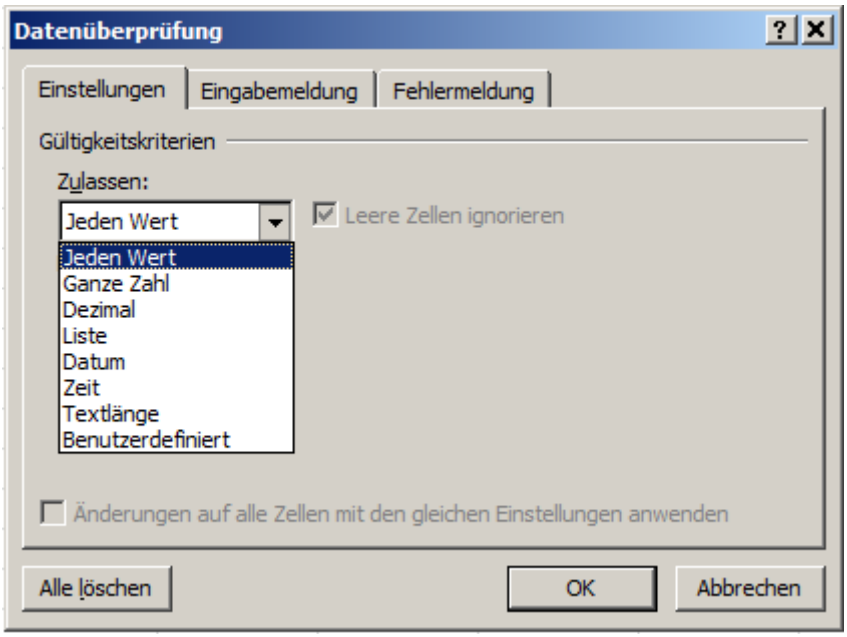
- Ein Startbereich (im Beispiel ist dies nur eine einzige Zelle: die Zelle A1)
- Eine Zeilenverschiebung, wobei positive Zahlen Verschiebungen nach unten sind, negative Werte Verschiebungen nach oben. Im Beispiel verschieben wir um eine Zeile nach unten.
- Eine Spaltenverschiebung. Der Parameterwert im Beispiel ist 0; es findet also keine Verschiebung statt.
- die Anzahl der Zeilen, die der zu liefernde Bereich umfassen soll. Im Beispiel liefert die Funktion ANZAHL2 diese Zeilenzahl. Der übergebene Parameter $A:$A bezeichnet die gesamte Spalte A. Dementsprechend liefert die Funktion die Anzahl der Zellen in der Spalte A, die Werte enthalten. Dies ist aber genau die Anzahl der Zellen, die von der Abteilungs-tabelle beansprucht wird. Zieht man noch eine Zelle für die Überschrift ab, so ergibt sich die Anzahl der Abteilungseinträge.
- Die Anzahl der Spalten, die der zu liefernde Bereich umfassen soll. Wir setzen im Beispiel 1, weil die Abteilungstabelle eine Spalte hat.

Wir wollen die Berechnung des Funktionsergebnisses nachvollziehen: Ausgehend vom Startpunkt A1 gehen wir eine Zeile nach unten und 0 Spalten nach rechts. Damit befinden wir uns beim obersten Eintrag der Liste. Dies ist die linke obere Ecke des zu liefernden Bereichs. Dieser Bereich umfasst soviele Zeilen, wie es Listeneinträge gibt, und genau eine Spalte (s. oben).

Weiter unten wird gezeigt, wie der benannte dynamische Bereich zur Eingabevalidierung verwendet werden kann.

3.4.5 Validierung von Eingaben

Um Daten zu überprüfen, die in Formulare (Masken) eingegeben werden, muss man VBA zu Hilfe nehmen. Daten, welche direkt in Tabellenblätter eingegeben werden, kann man weitgehend mit Excel-Mitteln überprüfen. Hierfür gibt es auf der Registerkarte Daten im Abschnitt Datentools sogar eine spezielle Schaltfläche „Daten überprüfen“. Diese öffnet ein Fenster zur Eingabe von Überprüfungsregeln.



Für einige häufige Fälle, wie Ganzzahligkeit der Eingabe, Beschränkung der Eingabewerte auf die Werte aus einer bestimmten Liste oder Beschränkung von Texteingaben auf eine maximale Länge, sind einfach anzuwendende Optionen verfügbar. Im folgenden Bild wird die Eingabe in den gerade markierten Bereich auf ganze Zahlen von 1 bis 14 beschränkt:

Für kompliziertere Fälle, insbesondere kombinierte Bedingungen, kann man die Option Benutzerdefiniert verwenden. Wir betrachten hierzu den Fall einer kombinierten Bedingung, die wir schrittweise formulieren. Sie umfasst die Prüfung auf Eindeutigkeit, auf Wertebereich > 0 und auf Ganzzahligkeit.

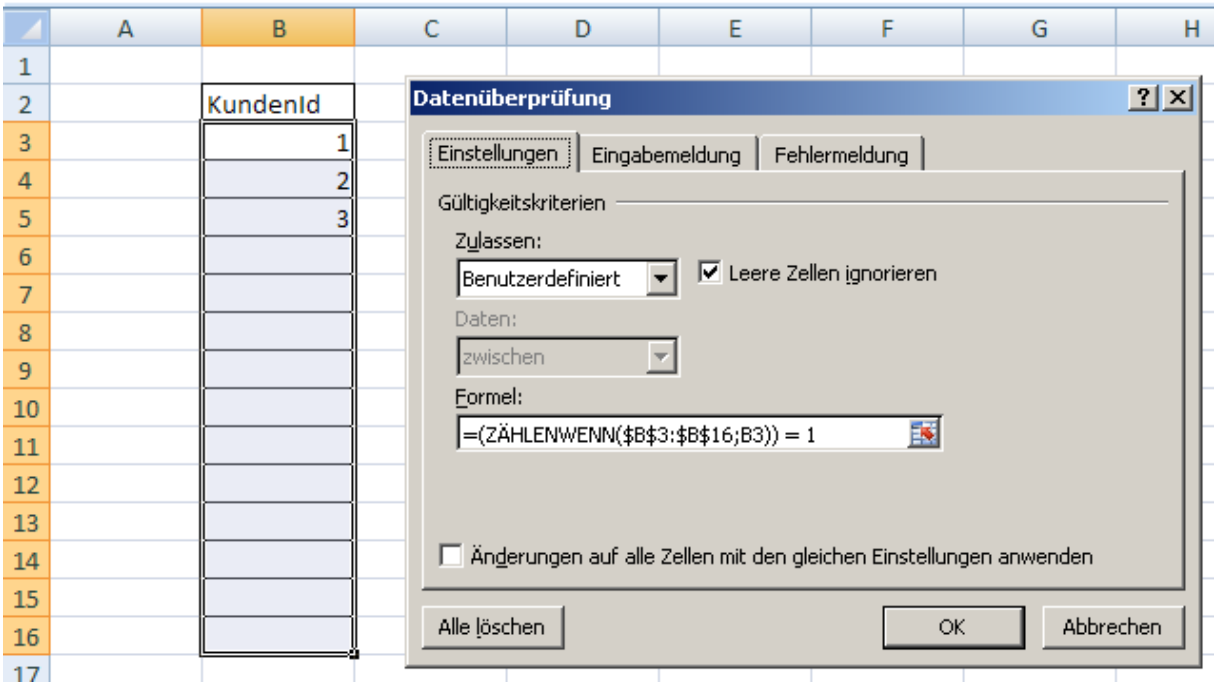
Prüfung auf Eindeutigkeit

Die erste Bedingung soll sein: Bei der Eingabe einer Kundenliste soll jede KundenId nur einmal vorkommen.

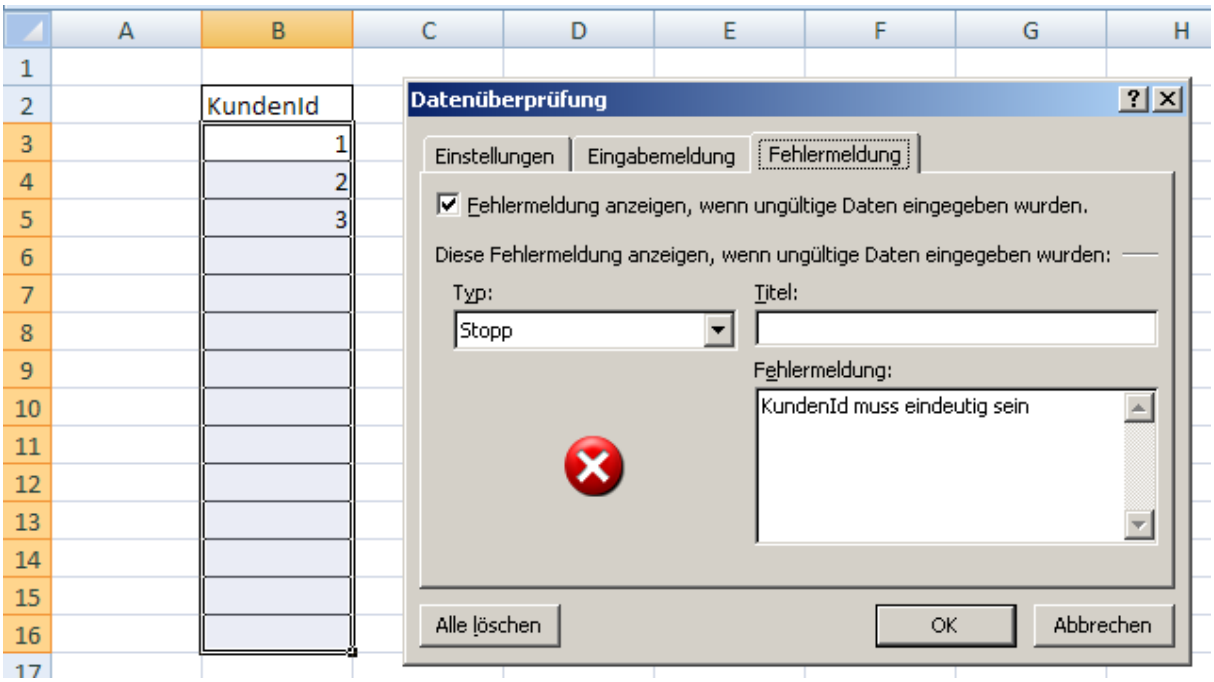
Man markiert zunächst den gesamten Eingabebereich für die KundenIds. Dann öffnet man das Fenster Datenüberprüfung und wählt bei Gültigkeitskriterien Benutzerdefiniert. Anschließend kann man die Überprüfungsregel eingeben.

Diese Regel beruht hier auf der Funktion ZÄHLENWENN (engl. COUNTIF). Diese Funktion zählt die nichtleeren Zellen eines Bereichs, die einer bestimmten Bedingung genügen. Die Bedingung ist hier durch die gerade eingegebene KundenId bestimmt. Die Funktion zählt, wie oft sie im Bereich vorkommt. Ist die Anzahl der Vorkommen nicht gleich 1, so ist die Bedingung verletzt, die Eingabe also nicht zulässig.

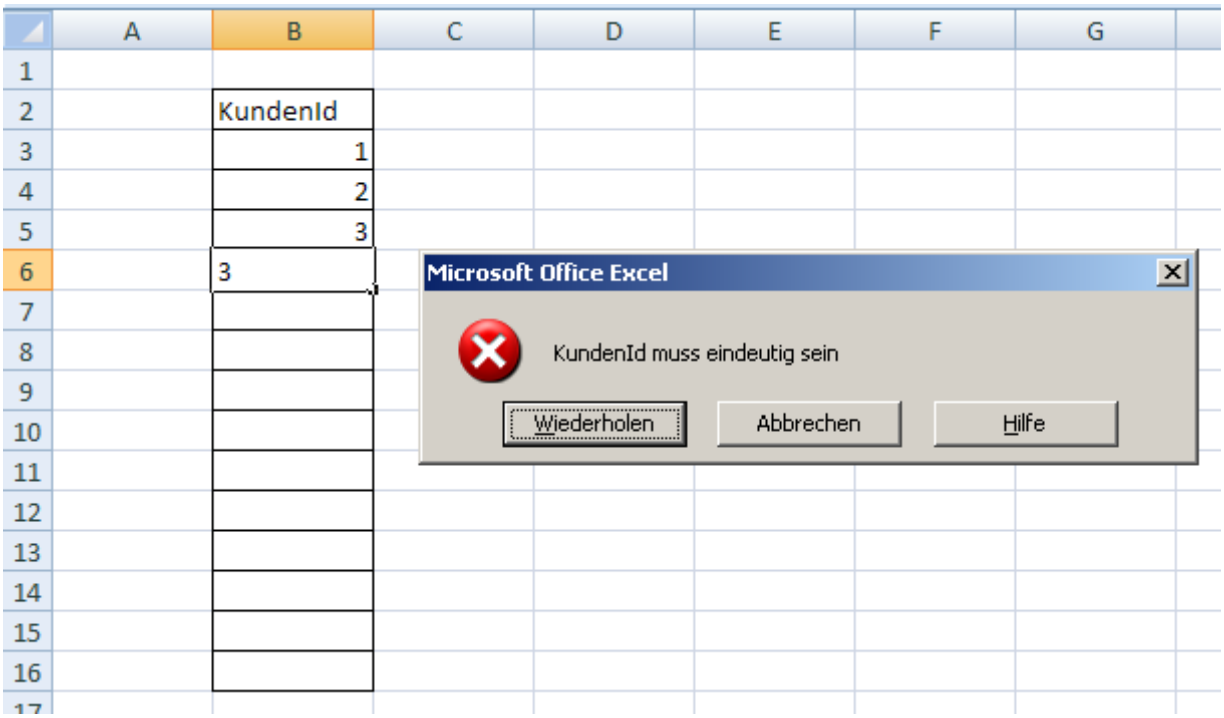
Beachten Sie das Setzen bzw. Weglassen der Dollarzeichen bei den Parametern von ZÄHLENWENN. B3 ist ohne Dollarzeichen eingegeben, weil die Eingabezelle nicht fix bleibt, sondern jede Zelle im Bereich sein kann.



Die Reaktion des Systems auf Eingabefehler kann in der Registerkarte Fehlermeldung genau definiert werden (Bild unten). Die Wahl des Reaktionstyps Stopp bewirkt, dass der fehlerhafte Eingabewert nicht akzeptiert wird. Zusätzlich kann der Text für eine Fehlermeldung festgelegt werden.

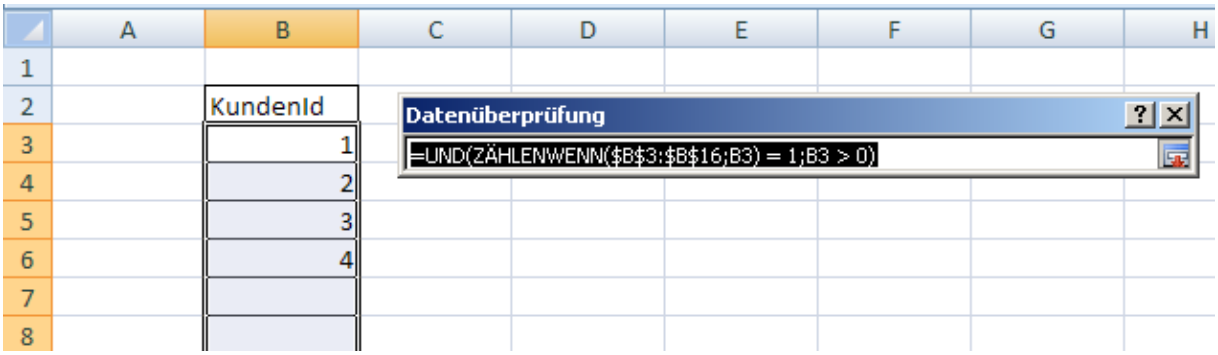


Das folgende Bild zeigt die Reaktion auf eine fehlerhafte Eingabe des Benutzers.



Ausweitung der Regel: Wertebereich > 0 und Ganzzahligkeit

Um zusätzlich zu erreichen, dass der Benutzer keine Kundennummern ≤ 0 eingibt, können wir die eingegebene Formel einfach erweitern. Um eine bereits eingegebene Regel zu verändern, markiert man nochmals den zu überprüfenden Bereich und drückt dann die Schaltfläche Datenüberprüfung. Es erscheint dann der Dialog mit der aktuell gültigen Regel. Diese kann man nun einfach ergänzen um die Regel für Wertebereich > 0 (s. unten) unter Verwendung der logischen Funktion UND.

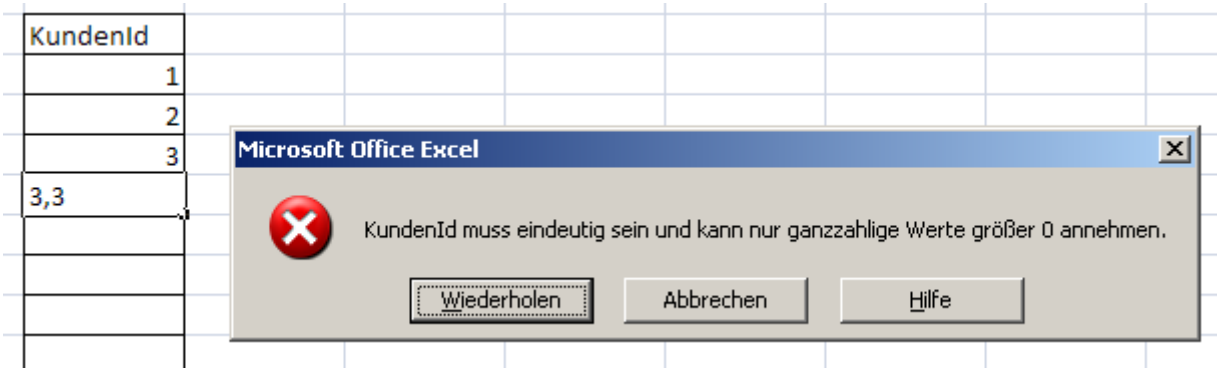


Ganzzahligkeit zu prüfen, ist im Prinzip kein Problem, denn es ist dafür im Dialog Datenüberprüfung eine eigene Option vorgesehen (s. oben).Das Problem in unserem Fall ist nur, dass bei Wahl dieser Option die anderen Bedingungen wegfielen. Wir müssen also einen anderen Weg wählen und nehmen stattdessen in die bestehende Regel die Ganzzahligkeitsbedingung zusätzlich auf. Da es in Excel keine spezielle Funktion zur Überprüfung der Ganzzahligkeit gibt, verwenden wir eine Kombination aus der Funktion GANZZAHL, welche den ganzzahligen Anteil einer Zahl liefert, und der Funktion REST, welche den Rest bei einer Division liefert. Eine ganze Zahl liegt demnach vor, wenn die Division einer Zahl durch ihren ganzzahligen Anteil den Rest 0 hat.

Mit Hilfe eines weiteren UND arbeiten wir die zusätzliche Bedingung in die vorhandene Regel ein. Es ergibt sich:

=UND(UND(ZÄHLENWENN($B$3:$B$16;B3) = 1;B3 > 0);REST(B3;GANZZAHL(B3))=0)

Man muss nun nur noch die Fehlermeldung anpassen. Das folgende Bild zeigt die Validierungsregel in Aktion.



Datenvalidierung durch Nachschlagen in Listen

Wenn der Benutzer nur Werte aus einer vorgegebenen Menge von Werten eingeben soll, kann man die Eingabe sehr gut mit Hilfe von Wertelisten validieren. Wie bereits oben gezeigt, existiert hierfür in der Option Daten überprüfen der Registerkarte Daten eine Auswahlmöglichkeit (Liste). Wir betrachten hier den etwas komplizierteren Fall, in dem die Eingabe in eine Zelle von der Eingabe in eine andere Zelle abhängt.

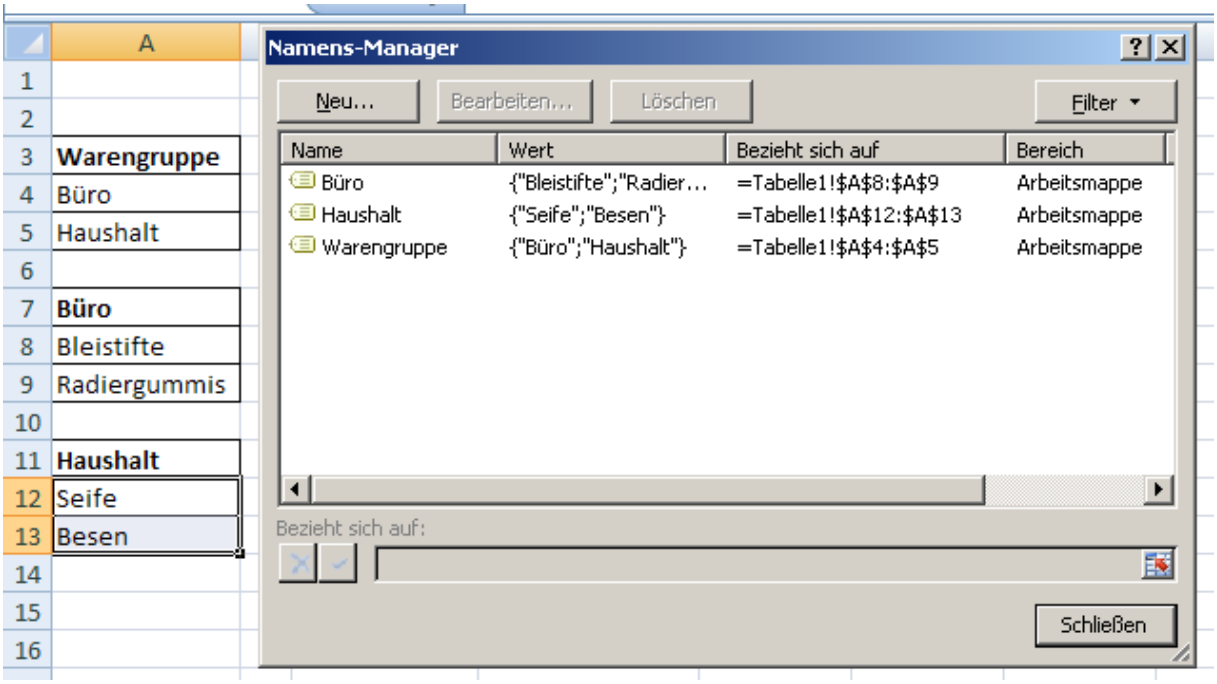
Das folgende Bild zeigt in den Spalten C und D eine Tabelle zur Eingabe von Warengruppen und Artikeln, in der Spalte A die Wertelisten, an denen sich die Eingabe orientieren soll. Die Wertelisten dienen nur der Validierung und werden zur Laufzeit versteckt.

	A	B	C	D
1				
2				
3	**Warengruppe**		**Warengruppe**	**Artikel**
4	Büro		Büro	Radiergummis
5	Haushalt			
6				
7	**Büro**			
8	Bleistifte			
9	Radiergummis			
10				
11	**Haushalt**			
12	Seife			
13	Besen			

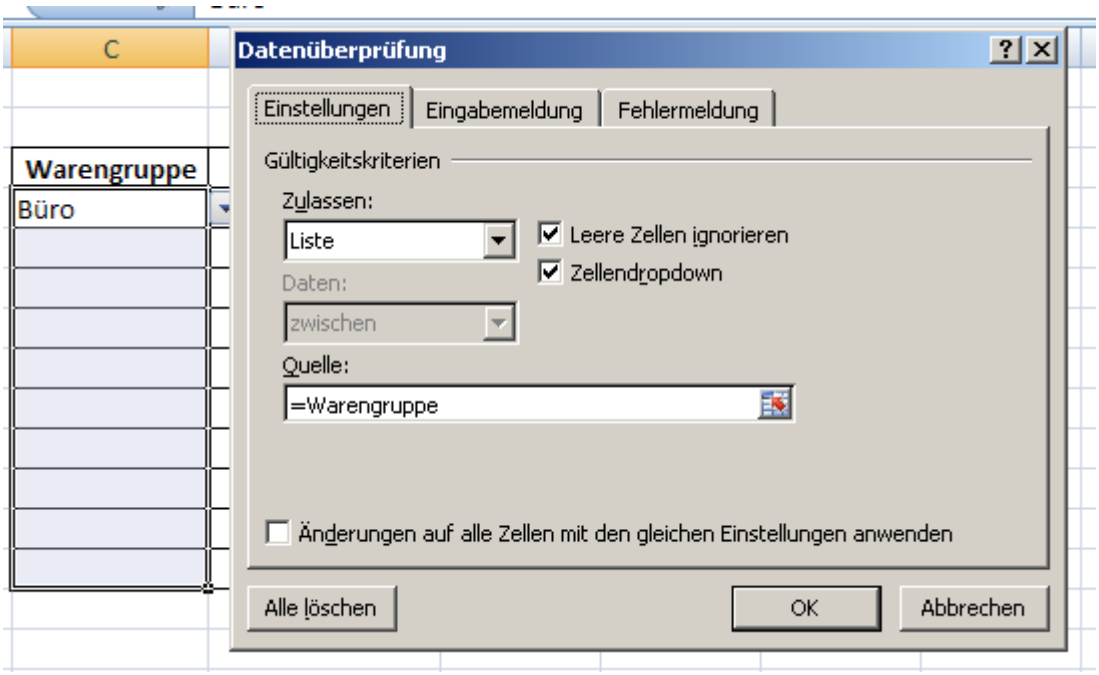
Die Validierung findet auf zwei Ebenen statt:

1. Es wird sichergestellt, dass der Benutzer nur Warengruppen eingibt, die in der Liste der Warengruppen vorkommen,
2. Es wird überprüft, dass der Artikel, der rechts von einer Warengruppe eingegeben wird, auch zu der Warengruppe gehört.

Zunächst werden für die drei Wertelisten Namen vergeben, welche mit den Überschriften der Listen übereinstimmen. Das folgende Bild zeigt bereits das Ergebnis dieser Namensvergabe im Namens-manager:

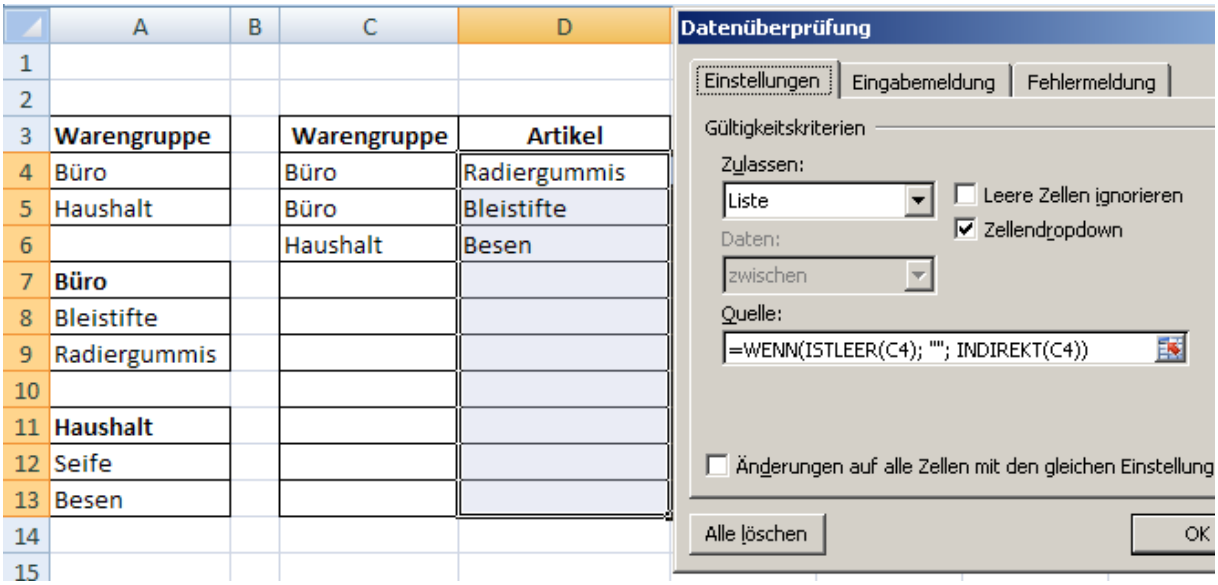


Um die Validierung für die Warengruppen einzugeben, markiert man den Bereich, wählt anschließend im Datenüberprüfungsdialog die Option Liste und gibt als Quelle den Namen Warengruppe an:



Die Validierung der Artikeleingaben gestaltet sich etwas schwieriger (s. unten). Die eingegebene Formel unterscheidet zwei Fälle. Ist die links von der Artikelzelle befindliche Warengruppenzelle leer, so soll die Artikelzelle auch leer sein. Enthält die Warengruppenzelle aber eine der Warengruppen, so sollen in der Artikelzelle nur die dazu gehörigen Artikel zulässig sein, also das, was in der betreffenden Werteliste steht. Die Funktion INDIREKT erlaubt, auf die zutreffende Liste zuzugreifen. INDIREKT liefert den Wert eines Bezugs. Der Ausdruck INDIREKT(C4) liefert also das, was in der Zelle C4 steht, nämlich den vom Benutzer eingegebene Name einer Warengruppe. Da diese Namen aber den von

uns vergebenen Namen für die Wertelisten gleich sind, können wir auf diese Weise auf die benannten Wertelisten zugreifen. Beachten Sie, dass die Option „Leere Zeilen ignorieren“ hier nicht aktiviert ist, denn dann wäre nicht sichergestellt, dass nur dann ein Artikel eingegeben werden kann, wenn auch die Warengruppe besetzt ist.



	A	B	C	D
1				
2				
3	**Warengruppe**		**Warengruppe**	**Artikel**
4	Büro		Büro	Radiergummis
5	Haushalt		Büro	Bleistifte
6			Haushalt	Besen
7	**Büro**			
8	Bleistifte			
9	Radiergummis			
10				
11	**Haushalt**			
12	Seife			
13	Besen			
14				
15				

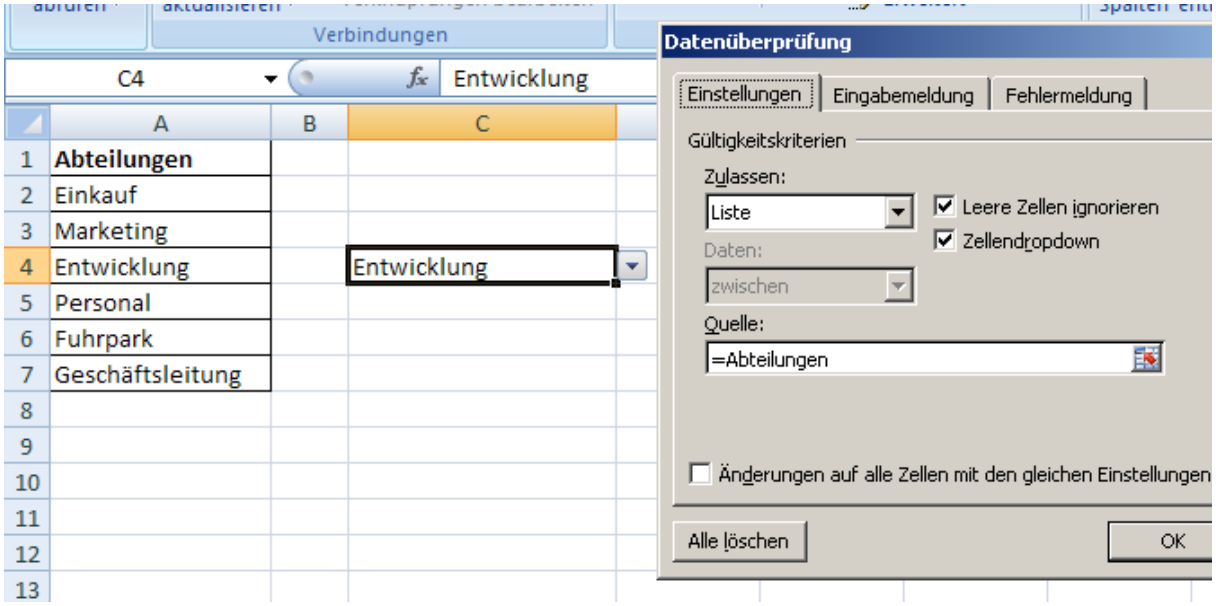
Das folgende Bild zeigt die Validierungsregel in Aktion. Der Benutzer hat in der Zeile 6 die Warengruppe Haushalt eingetragen. Dementsprechend werden ihm nur die dazu gehörigen Artikel Seife und Besen angeboten.

2				
3	**Warengruppe**		**Warengruppe**	**Artikel**
4	Büro		Büro	Radiergummis
5	Haushalt		Büro	
6			Haushalt	
7	**Büro**			Seife
8	Bleistifte			Besen
9	Radiergummis			
10				
11	**Haushalt**			
12	Seife			
13	Besen			
14				

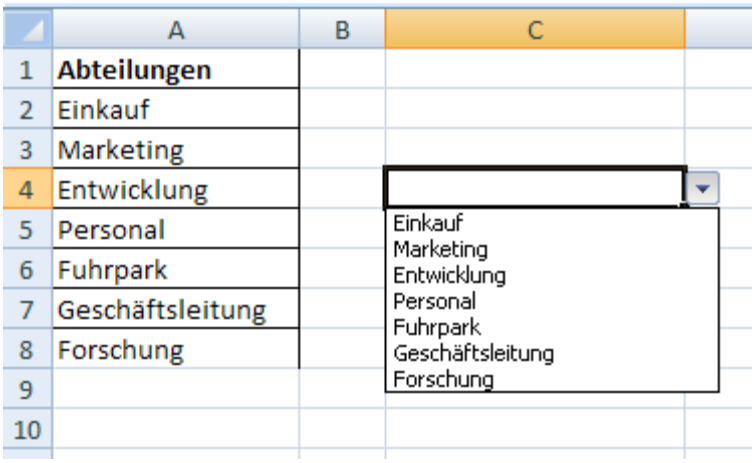
Validierung mit dynamischen Listen bzw. dynamischen Bereichen

Falls die Listen der zulässigen Werte selbst häufigen Änderungen unterworfen sind, empfiehlt es sich, die Validierung auf einen dynamischen Bereich zu beziehen. Weiter oben wurde gezeigt, wie man einen solchen Bereich definieren kann. Wir greifen das Beispiel noch einmal auf, um zu zeigen, wie dieser Bereich in die Validierung eingehen kann.

Im Datenüberprüfungsdialog wählen wir die Option Liste und beziehen uns bei Quelle auf den bereits vergebenen Namen Abteilungen, der einen dynamischen Bereich kennzeichnet.



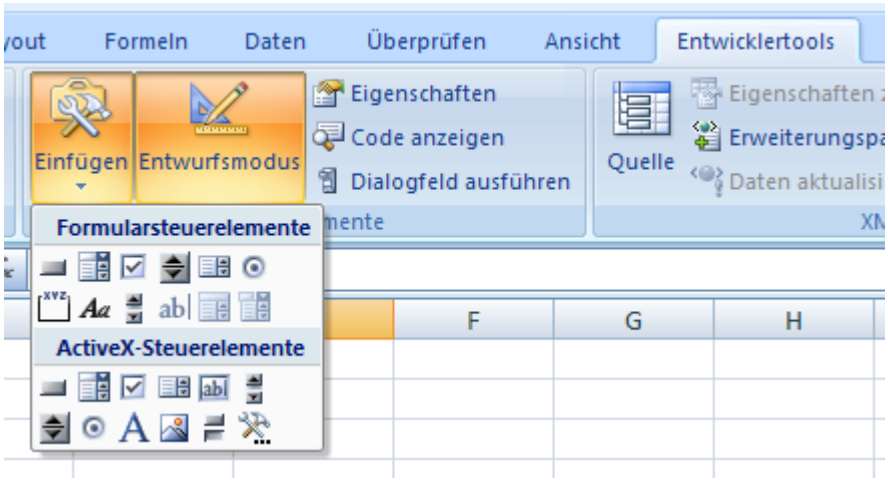
Wir können uns überzeugen, dass es funktioniert, indem wir in die Werteliste eine weitere Abteilung eingeben und danach eine Eingabe in das Feld C4 machen. In der Drop-Down-Liste neben dem Eingabefeld ist die neue Abteilung bereits berücksichtigt.



3.4.6 Benutzung von Steuerelementen auf Tabellenblättern

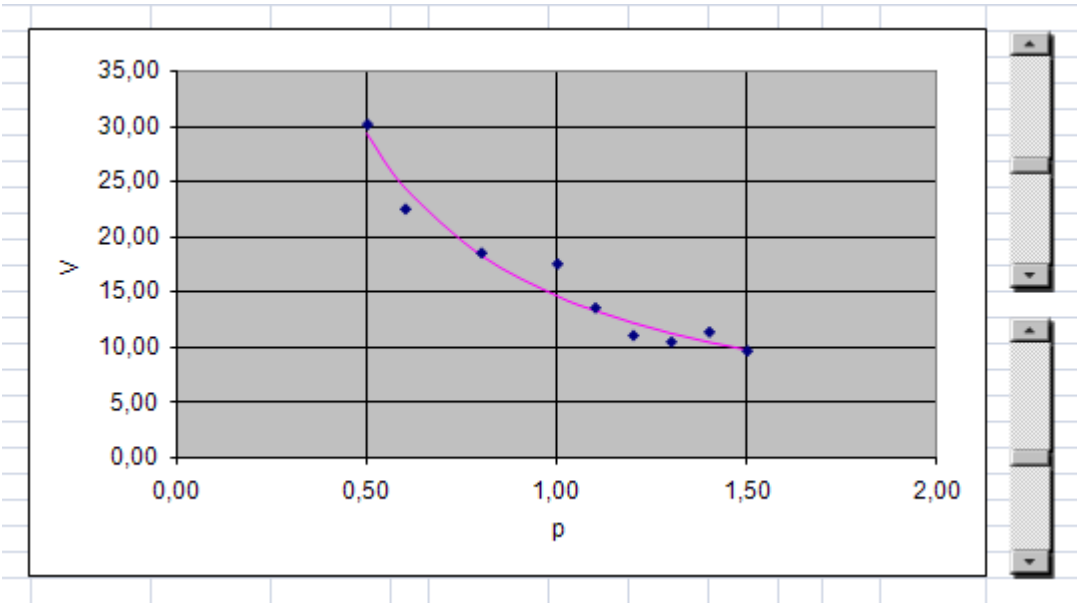
Man kann auf Tabellenblättern auch Steuerelemente (Controls) platzieren. Diese können Eingaben unterstützen oder Aktionen auslösen. Zwei Arten von Steuerelementen stehen zur Verfügung: klassische Formularsteuerelemente und ActiveX-Steuerelemente. Formularsteuerelemente sind einfacher in der Handhabung, aber weit weniger flexibel als ActiveX-Steuerelemente. Für VBA-Programmierer empfiehlt es sich, gleich die ActiveX-Elemente zu verwenden, weil sie im Rahmen von VBA vorzugsweise benutzt werden.

Hat eine Anwendung ein UI, das nur Tabellenblätter benutzt, also nicht von Formularen Gebrauch macht, so kommt man um die Anwendung von Steuerelementen nicht herum. Man benötigt zumindest eine Schaltfläche, um die Berechnung zu starten.



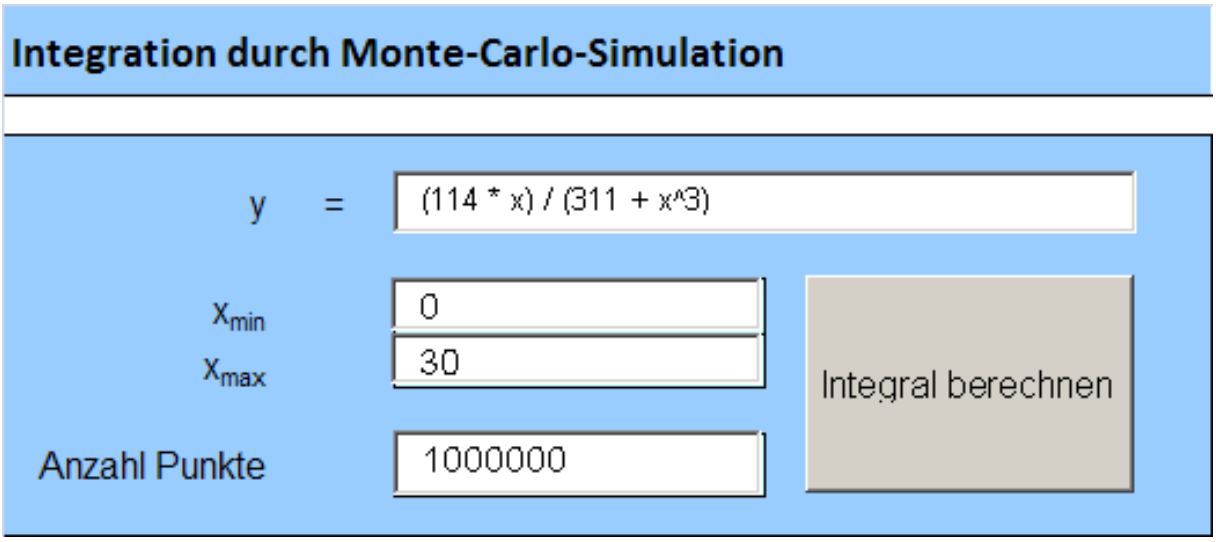
Zu den Steuerelementen gelangt man über die Registerkarte Entwicklertools (falls sie nicht gezeigt wird, über Excel-Optionen hinzufügen) und den darin befindlichen Abschnitt Steuerelemente.

Gut geeignet sind Steuerelemente, wenn es darum geht, auf dem Tabellenblatt abgebildete Objekte zu manipulieren, wie die Lage der farbigen Linie in der folgenden Graphik.



Ein weiteres Beispiel für einen nutzbringenden Einsatz von Steuerelementen zeigt das folgende Bild. Eingabefelder (sog. Textboxes) werden verwendet, um vom Benutzer die notwendigen Eingabedaten abzufragen. Eine Schaltfläche (Button) nimmt dann das Kommando zum Starten der Berechnung entgegen.

Die Validierung der Eingaben geschieht bei dieser Form der Eingabe in VBA. Sie unterscheidet sich wenig von der Validierung im Rahmen von Formularen.



Die Benutzung von Textboxes anstelle von Tabellenblattzellen hat den Vorteil, dass die Eingabe auf die Textbox konzentriert bleibt und nicht, wie bei Eingabe über eine Zelle, bei Änderungen in der Eingabezeile des Tabellenblatts gearbeitet werden muss. Man kann den Benutzer also enger führen.

Wenn die Benutzer allerdings im Rahmen einer Anwendung ganze Tabellen eingeben müssen, empfiehlt sich die Verwendung von Textboxes nicht. Man sollte dies über das Tabellenblatt machen, aber die Eingaben sorgfältig validieren.

3.5 Der Aufbau einer Anwendung

Programme, die nur aus einer Funktion oder einer Prozedur bestehen, gibt es meist nur als Lehrbeispiele, aber selten in der Praxis. Praxisrelevante Programme bestehen oft aus hunderten von Funktionen und Prozeduren, manchmal sogar aus tausenden.

Damit die Programmierer in einem so großen Programm den Überblick behalten können, müssen sie ihm eine sinnvolle Struktur geben. Gut strukturierte Programme sind gewöhnlich zuverlässiger und wartungsfreundlicher als schlecht strukturierte oder gänzlich unstrukturierte. Auch lässt sich nur mit einer sinnvollen Struktur ein hohes Maß an Wiederverwendbarkeit erreichen.

Wie eine solche sinnvolle Struktur (ein solcher Programmaufbau) aussehen kann, wird in diesem Abschnitt gezeigt. Die Beispiele, die wir hierzu betrachten, haben einen sehr kleinen Umfang, sind also im Hinblick auf die Größe nicht typisch für die Praxis. Die Beschränkung auf kleine Programme erlaubt jedoch besser, die Strukturierungsprinzipien herauszuarbeiten, die ihren Nutzen besonders bei umfangreichen Programmen entfalten. Außerdem müssen wir nur geringen Gebrauch vom sog. Excel-Objektmodell machen, das im nächsten Hauptabschnitt behandelt wird.

3.5.1 Funktionen, Prozeduren und Module

Ein Excel-VBA-Programm besteht gewöhnlich aus mehreren, oft auch aus vielen Modulen. In diesen Modulen sind die Funktionen und Prozeduren des Programms enthalten. Man kann weder eine

Funktion noch eine Prozedur außerhalb eines Moduls schreiben. Neben Funktionen und Prozeduren können Module aber auch noch Deklarationen von Variablen oder von Datentypen enthalten.

Module fassen also Funktionen und Prozeduren zu größeren Einheiten zusammen. Module in VBA sind nicht aktiv, wie Funktionen oder Prozeduren es sein können. Während eine Prozedur eine andere Prozedur oder eine Funktion aufrufen kann, gibt es eine solche Zusammenarbeit zwischen Modulen nicht. Kein Modul kann ein anderes aufrufen. Wenn Entwickler davon sprechen, dass ein bestimmtes Modul ein anderes benutzt bzw. aufruft, dann muss man sich bei einer solchen Formulierung im Klaren darüber sein, dass eigentlich etwas anderes gemeint ist: eine Prozedur bzw. Funktion des einen Moduls ruft eine Prozedur bzw. Funktion des anderen Moduls auf.

In Excel-VBA gibt es verschiedene **Arten von Modulen**, die für unterschiedliche Zwecke eingesetzt werden:

- Der Arbeitsmappe und jedem Arbeitsblatt ist ein **spezielles Modul** zugeordnet. In diese speziellen Module werden gewöhnlich keine Anweisungen geschrieben, welche das Programm selbst betreffen. Häufig benutzt man sie für Ereignisprozeduren, welche mit Steuerelementen verbunden sind, die sich auf einem Arbeitsblatt befinden. Beispiel: Auf einem Arbeitsblatt namens Start befindet sich eine Schaltfläche, welche zum Starten eines Programms dient. Der Code im Modul des Arbeitsblatts stellt damit die Verbindung zwischen der Schaltfläche und dem eigentlichen Programm her.
- **Standardmodule** sind die Module, in denen sich gewöhnlich der Kern eines Programms, also der Hauptteil, befindet. Anders als die oben erwähnten speziellen Arbeitsblatt- und Arbeitsmappenmodule müssen Standardmodule vom Programmierer explizit angelegt werden.
- **Klassenmodule** werden nur in der objektorientierten Programmierung (OOP) eingesetzt. Ein Klassenmodul enthält den Code einer Klasse. In Excel-VBA sind reinrassige objektorientierte Programme selten, aber fast alle Programme enthalten Elemente der objektorientierten Programmierung.
- **Formularmodule** enthalten den Code, der die Benutzung eines Formulars (einer Bildschirmmaske) betrifft, also insbesondere die Ereignisprozeduren, welche auf die vom Benutzer ausgelösten Ereignisse reagieren. Formularmodule sind spezielle Klassenmodule, denn Formulare werden in Excel-VBA prinzipiell als Klassen implementiert.

Die Struktur eines Programms ist durch den Inhalt seiner Module und die Zusammenarbeit zwischen ihnen bestimmt. Es ist offensichtlich, dass die Aspekte des Modulinhalts und der Modulzusammenarbeit eng zusammenhängen. Man kann sie aber trotzdem getrennt diskutieren, wenn man dies in allgemeiner Form tut. Dies soll in den folgenden drei Abschnitten geschehen. Zunächst werden in 3.5.2 zwei bewährte Ansätze zur Bildung von Modulen vorgestellt. Die daran anschließenden Abschnitte 3.5.3 und 3.5.3 beschäftigen sich mit allgemein anerkannten Prinzipien zur Gestaltung der Modulzusammenarbeit.

Bei den Überlegungen zum Modulinhalt bereiten die speziellen Module für die Arbeitsmappe und die Arbeitsblätter und die Formularmodule am wenigsten Kopfzerbrechen. Beide Arten haben vorwiegend Brückenfunktion. Die Ereignisprozeduren in den speziellen Modulen schlagen die Brücke zwischen den Benutzer und der Programmbenutzung, die Formularmodule sorgen danach dafür, dass die vom Benutzer gewollten Programmfunktionen vom Kern des Programms genau so ausgeführt werden, wie der Benutzer dies wünscht. Wir beziehen uns deshalb im Abschnitt 3.5.2 ausschließlich auf Standard- und Klassenmodule.

3.5.2 Modulbildung: Was soll in ein Modul?

Generell wünschen wir uns von einem Modul, dass es mit den Zielen der Softwareentwicklung harmoniert. Insbesondere sollte ein Modul gut wiederverwendbar und leicht wartbar sein, und durch seine Verständlichkeit und Übersichtlichkeit eine zügige und korrekte Programmierung fördern. Die folgenden Modultypen haben sich als besonders nützlich und zielkonform erwiesen:

Typ A: Modul, das genau eine Funktion enthält

Im zweiten Teil dieses Kurses (Mächtige benutzerdefinierte Funktionen entwickeln) wurde ein Typ von Funktion beschrieben, der im Hinblick auf Wartbarkeit und Wiederverwendung ideal ist. Die wichtigsten Eigenschaften einer solchen Funktion seien kurz wiederholt:

- Die Parameter sind vollständig spezifiziert, insbesondere ist ein Datentyp angegeben, Der Datentyp des Funktionsergebnisses ist ebenfalls angegeben.
- In die Ermittlung des Funktionsergebnisses gehen neben den in der Parameterklammer angegebenen Parametern keine weiteren Informationen ein.
- Die Ermittlung des Funktionsergebnisses ist der einzige Effekt der Funktion, d.h. sie hat keine Nebenwirkungen.

Offensichtlich sind Module, die nur aus einer solchen Funktion bestehen, hinsichtlich der Wiederverwendung und der Wartbarkeit optimal. Im Hinblick auf die Wartbarkeit des gesamten Programms muss man aber darauf achten, dass die Funktionen eine gewisse Mindestgröße haben. In einem Programm, das aus Tausenden von kleinen Modulen zusammengesetzt wäre, könnte leicht die Übersicht verloren gehen.

Eine Funktion, aus der ein Modul gebildet wird, muss also hinreichend „bedeutend“ sein. Hier einige Beispiele:

- Eine Funktion, die eine Matrix sortiert.
- Eine Funktion, die ein magisches Quadrat mit der Kantenlänge *n* liefert.
- Eine Funktion, welche die konvexe Hülle für eine Menge von Punkten ermittelt.

Typ B: Modul, das ein Objekt oder eine Datenstruktur repräsentiert

Module dieses Typs sind immer zweiteilig. Der eine Teil besteht aus den Deklarationen der „globalen“, d.h. im gesamten Modul sichtbaren Variablen, in denen der Zustand des zu verwaltenden Objekts gespeichert wird. Der andere Teil wird von den Prozeduren und Funktionen gebildet, welche zur Verwaltung des Objekts benötigt werden. Wichtig ist, dass die globalen Variablen immer als „private“ deklariert werden. Wenn dieser Grundsatz eingehalten wird, können alle Prozeduren und Funktionen des Moduls auf diese Variablen zugreifen, aber es sind keine Zugriffe von außerhalb möglich. Auf diese Weise bilden die Prozeduren des Moduls einen Schutzschild um die globalen

Variablen. Das Objekt kann von außen nur über Aufrufe der Prozeduren und Funktionen verändert werden („Kapselung“). Beispiele für Module des Typs B:

- Modul „Geldautomat“. Dieses Modul repräsentiert bzw. ersetzt innerhalb einer Simulation den realen Automaten. Zu den globalen Variablen gehören z.B. die aktuellen Bestände an Geldscheinen.
- Modul „Roboter“. Auch dieses Modul wird innerhalb von Simulationen eingesetzt. Es repräsentiert dort einen realen Roboter. Die wichtigsten globalen Variablen sind die, welche den aktuellen Zustand des Roboters festhalten, also z.B. die Ausrichtung und der Standort.
- Modul „Tokenizer“. Tokenizer kommen innerhalb von Übersetzern zur Anwendung. Der Tokenizer liefert jeweils ein Wort (ein „Token“) des Programmtexts, eines nach dem anderen. Es werden insbesondere zwei globale Variablen gebraucht, eine für den Text selbst und eine zur Speicherung der aktuellen Position im Text, so dass bestimmt werden kann, welches Wort als nächstes geliefert werden soll.

Generell lassen sich beide Modultypen sowohl als Standardmodule als auch als Klassenmodule realisieren. Während aber beim Typ A beide Realisierungsformen gleichwertig sind, gibt es beim Typ B einen wesentlichen Unterschied. Klassenmodule erlauben, innerhalb eines Programms gleichzeitig mehrere Exemplare des vom Modul repräsentierten Objekts zu benutzen, also mehrere Roboter oder mehrere Geldautomaten gleichzeitig. Bei Standardmodulen ist dies nicht möglich.

Typ C: Modul, das verwandte Funktionen vereint

Ein Modul von diesem Typ besteht gewöhnlich aus einer Menge von Funktionen eines bestimmten Sachgebiets. Das Modul ist „Experte“ auf diesem Gebiet. Deklarationen globaler Variablen sind in einem Modul vom Typ C nicht angebracht. Die enthaltenen Funktionen arbeiten völlig selbstständig voneinander. Beispiele:

- In einem Modul „Matrixexperte“ wird eine Sammlung von Matrixoperationen (Addition, Multiplikation, Inversion, Transposition, …) bereitgestellt.
- Ein Modul „Stringexperte“ enthält Funktionen zur Bearbeitung von Zeichenketten (Strings)
- Das Modul „Zeitexperte“ enthält Funktionen, welche erlauben, Zeitangaben aus einer Zeitzone in Zeitangaben aus einer anderen Zeitzone umzurechnen.

Es soll nicht verschwiegen werden, dass Module dieses Typs in der Theorie der Softwareentwicklung einen zweifelhaften Ruf haben. Allerdings beruht diese Geringschätzung auf einer anderen, engeren Definition des Moduls, nämlich der einer separat übersetzbaren Einheit. Module dieser Art gibt es in VBA aber nicht; VBA-Module sind bloße Verwaltungseinheiten.

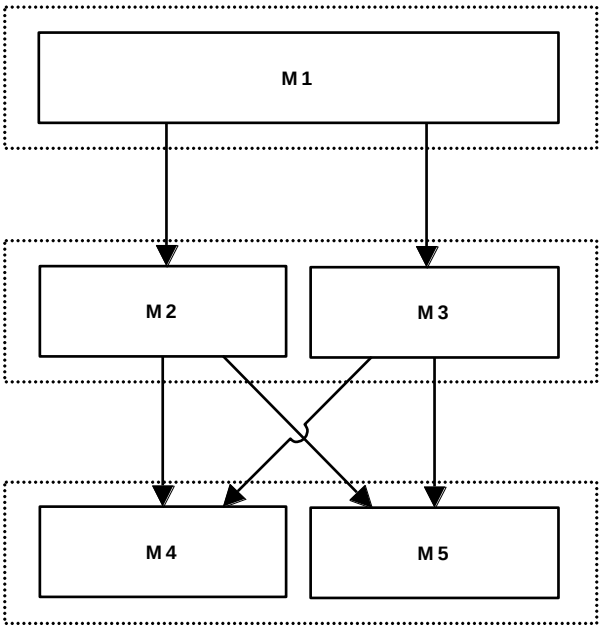
3.5.3 Wie sollen Module zusammenarbeiten? (1) Schichtenarchitekturen

Schichtenarchitekturen sind in der Softwaretechnik unentbehrlich bei der Entwicklung von großen Systemen. Sie fördern Wartbarkeit und Wiederverwendbarkeit der Programme, und sie sind eine

gute Grundlage für arbeitsteiliges Entwickeln. Das folgende Bild zeigt eine aus fünf Modulen und drei Schichten bestehende Schichtenarchitektur.

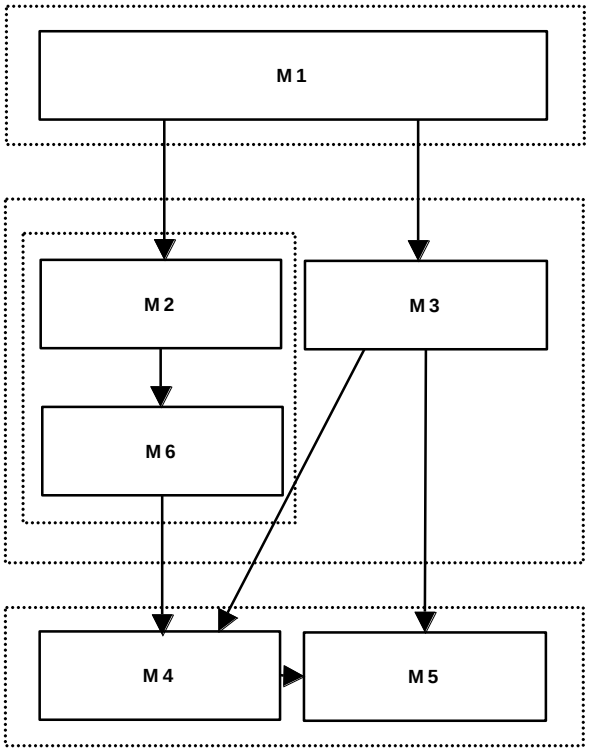
Eine Schichtenarchitektur bildet stets eine **Zugriffshierarchie**. Jede Schicht kann nur auf darunter liegende Schichten zugreifen, niemals auf eine darüber liegende. In der **strengen Form** des Schichtenprinzips sind nur Zugriffe auf Einheiten der direkt darunterliegenden Schicht gestattet, in der **weiteren Form** sind Sprünge über Ebenen hinweg und Zugriffe auf Einheiten derselben Ebene erlaubt.

Das folgende Bild zeigt eine Ausprägung der strengen Schichtenarchitektur. Die Pfeile in der Graphik zeigen die Richtung der darin stattfindenden Aufrufe.



Das nächste Bild enthält ein weiteres Beispiel einer dreistufigen Schichtenarchitektur. Gegenüber dem vorherigen Beispiel fallen zwei Unterschiede auf:

- In der untersten Schicht finden schichteninterne Zugriffe von Modul M4 nach Modul M5 statt. Es wird also die weitere Interpretation des Schichtenprinzips verwendet.
- Die zweite Besonderheit ist mit beiden Interpretationen des Schichtenprinzips vereinbar. Es geht um die Bildung von Subsystemen innerhalb der Schichten. In der mittleren Schicht ist ein Modul M6 enthalten, welches ausschließlich Aufrufe aus dem Modul M2 entgegennimmt. Die beiden Module bilden deshalb ein Subsystem innerhalb dieser Schicht.



Schichtenarchitekturen sind bei der Entwicklung großer Softwaresysteme die Norm. Sie haben folgende Vorteile:

- Die Software bekommt dadurch einen übersichtlichen Aufbau und ist leichter „in den Griff zu bekommen“.
- Es ist leichter, arbeitsteilig zu entwickeln. Erstens bilden die Schichten natürliche Arbeitseinheiten, und zweitens erlaubt die Schichtenbildung isoliertes Arbeiten an den einzelnen Schichten. Voraussetzung dafür ist allerdings, dass die Schnittstellen zwischen den Schichten sorgfältig geplant und dann festgelegt werden.
- Die Wartbarkeit wird verbessert, weil isoliertes Warten der einzelnen Schichten möglich ist.

Das strenge Schichtenprinzip bindet die Entwicklung natürlich stärker als das weitere und sorgt damit für noch mehr Strukturierung und Übersichtlichkeit. Je komplexer die Aufgabe ist, umso mehr spricht für die strenge Interpretation.

3.5.4 Wie sollen Module zusammenarbeiten? (2) Trennung von UI und Programmlogik

Eine besondere, aber allgemein übliche und wichtige Ausprägung der Schichtenarchitektur ist die Trennung der Benutzerführung (UI) von der Programmlogik, also dem Kern des Programms. Trennung bedeutet hier, dass diese Komponenten verschiedenen Schichten angehören sollen. Bei vielen Systemen kommt noch eine dritte Schicht hinzu, welche für die Speicherung der Daten auf einer Datenbank und das Wiedereinlesen dieser Daten zuständig ist.

Eine solche Schichtenarchitektur besteht aus demnach aus drei Schichten:

- der Benutzeroberfläche bzw. der Benutzerführung (UI),

- einer „logischen Schicht“ für den Verarbeitungskern
- einer Datenbankzugriffsschicht.

Bei komplexen Systemen kann jede dieser drei Schichten selbst wieder in Schichten aufgeteilt sein. Kleine Anwendungen ohne Datenbankzugriff sollten zumindest zwei Schichten besitzen, die Benutzerführung und die logische Schicht.

Die Trennung von UI und Programmlogik ist besonders wichtig, weil Änderungen an der Benutzerführung bei unverändert bleibender Programmlogik sehr häufig anfallen.

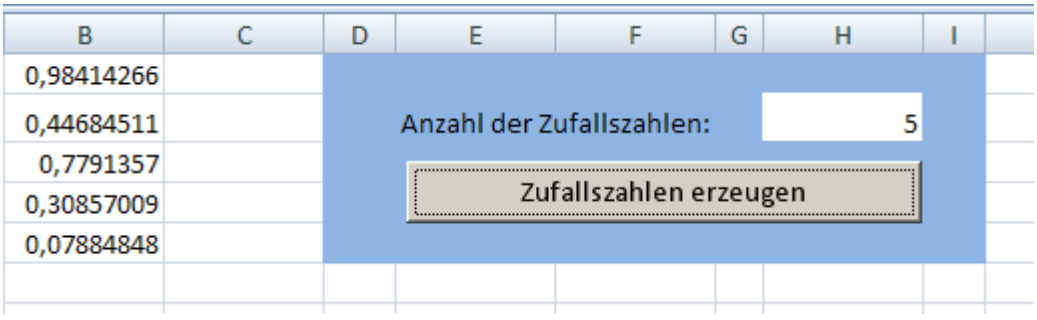
Wir betrachten in den folgenden drei Abschnitten 3.5.5 bis 3.5.7 verschiedene Varianten einer zweischichtigen Anwendung ohne Datenbankzugriff. Das Programm erlaubt dem Benutzer, eine natürliche Zahl *n* zwischen 1 und 1 Million einzugeben. Anschließend werden *n* Zufallszahlen ausgegeben. Drei Versionen dieses Programms werden vorgestellt, die sich in der Gestaltung des UI unterscheiden:

- Version 1: UI vollständig auf Tabellenblatt (Excel-UI)
- Version 2: UI vollständig auf Formular
- Version 3: gemischtes UI

3.5.5 Eine Anwendung mit Excel-UI

Das Bild unten zeigt rechts den blau gefärbten Eingabebereich, links (Spalte B) die Ausgabe der Zufallszahlen. Das Eingabefeld für die Anzahl ist eine gewöhnliche Zelle des Tabellenblatts. Die Schaltfläche darunter erlaubt dem Benutzer, die Zufallszahlenberechnung anzustoßen.

Für das Eingabefeld H2 wurde mit Hilfe der Excel-Option Daten > Datenprüfung festgelegt, dass die Eingabe eine Ganzzahl im Bereich von 1 bis 1 Million sein soll.



Die Ereignisprozedur für die Schaltfläche wurde im Modul des Tabellenblatts platziert. Dieses Modul bildet die **obere Schicht** der zweischichtigen Architektur.

```
Private Sub ZufallStartBtn_Click()
  Dim n As Long
  Dim r() As Double
  If Not IsEmpty(Worksheets("Tabelle1").Cells(2, 8).Value) Then
    n = CLng(Worksheets("Tabelle1").Cells(2, 8).Value)
  Else
    MsgBox "Sie müssen eine Zahl im Bereich 1 bis 1 Million eingeben"
    Exit Sub
```

```
    End If
    Worksheets("Tabelle1").Range("B:B").Clear
    r = Zufall.ZufallszahlenErzeugen(n)
    Worksheets("Tabelle1").Range("B1:B" & n) = r
End Sub
```

Beachten Sie, dass die Prozedur zunächst überprüft, ob das Eingabefeld nicht leer ist. Die eingestellte Datenüberprüfung sorgt zwar dafür, dass eine eingegebene Zahl sich im erwünschten Bereich befindet, sie kann aber nicht sicher stellen, dass überhaupt etwas eingegeben wird. Erweist sich das Eingabefeld als nicht leer, so werden im weiteren Verlauf dann die Zufallszahlen durch einen Aufruf der Funktion ZufallszahlenErzeugen im Modul Zufall abgerufen und dann in die Spalte B von Tabelle1 eingestellt.

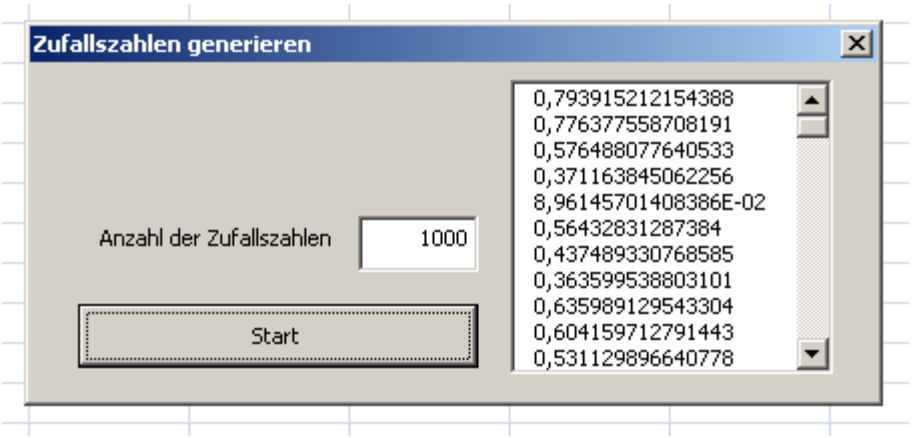
Das Modul Zufall, das nur die Funktion ZufallszahlenErzeugen enthält, stellt die **zweite Schicht** unserer Zwei-Schichten-Anwendung dar. Die Funktion liefert n Zufallszahlen in einem Double-Array. Es handelt sich dabei formal um ein zweidimensionales Array, also eine Matrix. Allerdings besitzt diese Matrix nur eine Spalte. Der Grund für diesen Kunstgriff ist, dass ein eindimensionales Array nicht so gut in eine Spalte eines Tabellenblatts eingefügt werden könnte.

```
Public Function ZufallszahlenErzeugen(ByVal n As Long) As Double()
    Dim iAs Long
    Dim z() As Double
    ReDimz(1 To n, 1 To 1)
    For i = 1 To n
        z(i, 1) = Rnd
    Next i
    ZufallszahlenErzeugen = z
End Function
```

Es bleibt noch anzumerken, dass das Arbeiten mit dem UI hier nicht optimal ist. Es gibt einige gravierende Fehlerquellen. Es könnte z.B. vorkommen, dass der Benutzer die Schaltfläche drückt, ohne vorher eine Anzahl eingegeben zu haben. Auch wenn er eine Zahl eintippt, kann noch etwas passieren, nämlich wenn er die Eingabe nicht mit der Eingabetaste abschließt, bevor er die Schaltfläche drückt. Wir könnten diese Probleme zumindest teilweise entschärfen, indem wir als Eingabefeld ein Textfeld benutzen an Stelle einer Zelle.

3.5.6 Eine Anwendung mit Formular-UI

Die zweite Version wickelt sowohl die Eingabe als auch die Ausgabe über ein Formular ab (s. unten). Die Eingabe erfolgt in einer Textbox; nach dem Anklicken der Schaltfläche Start werden die Zufallszahlen generiert und in der Listbox auf der rechten Seite ausgegeben.



Auch diese Version des Programms ist zweischichtig, die obere Schicht besteht nun aus dem Modul (genauer: dem Klassenmodul) des Formulars, die zweite Schicht besteht aus zwei Moduln: wie bei der ersten Version enthält sie das Modul Zufall mit der Kernfunktion des Programms. Zusätzlich ist darin aber noch ein Modul HR mit Hilfsfunktionen, die bei der Validierung der Eingabe benötigt werden.

Formularmodul	
Zufall	HR

Wir beginnen mit dem Formularmodul. Dieses enthält neben der Ereignisprozedur, welche in Aktion tritt, nachdem der Benutzer die Schaltfläche Start gedrückt hat, auch eine Funktion ValidierungOK, welche die Eingabe überprüft. Beachten Sie, dass diese Funktion selbst wieder Funktionen des Moduls HR aus der zweiten Schicht aufruft.

```
Private Sub StartBtn_Click()
  Dim n As Long
  If ValidierungOK Then
    n = CLng(Me.AnzahlTBx.Text)
    Me.ZZLBx.Clear
    Me.ZZLBx.List = Zufall.ZufallszahlenErzeugen(n)
  End If
End Sub


Private Function ValidierungOK() As Boolean
  If HR.istNatuerlicheZahl(AnzahlTBx.Text) Then
    If CLng(Me.AnzahlTBx.Text) <= 1000000 Then
      ValidierungOK = True
    Else
      ValidierungOK = False
    End If
  Else
    ValidierungOK = False
  End If
  If Not ValidierungOK Then
    MsgBox "nur ganze Zahlen von 1 bis 1 Million eingeben!"
```

```
        Me.AnzahlTBx.SetFocus
    End If
End Function
```

Die Funktion ValidierungOK fängt auch den Fall ab, dass der Benutzer die Schaltfläche Start drückt, ohne eine Anzahl eingegeben zu haben. Es erscheint in diesem Fall dieselbe Fehlermeldung wie bei der Eingabe einer unpassenden Zahl. Man könnte aber auch das Textfeld für die Anzahl mit einer häufig benutzten Zahl vorbesetzen. In diesem Fall würde das Programm ausgeführt, auch wenn der Benutzer keine Anzahl eingegeben hat. Das Vorbesetzen geschieht automatisch beim Laden des Formulars wenn man dem Formularmodul folgende Ereignisprozedur hinzufügt:

```
Private Sub UserForm_Initialize()
    Me.AnzahlTBx.Text = 1
End Sub
```

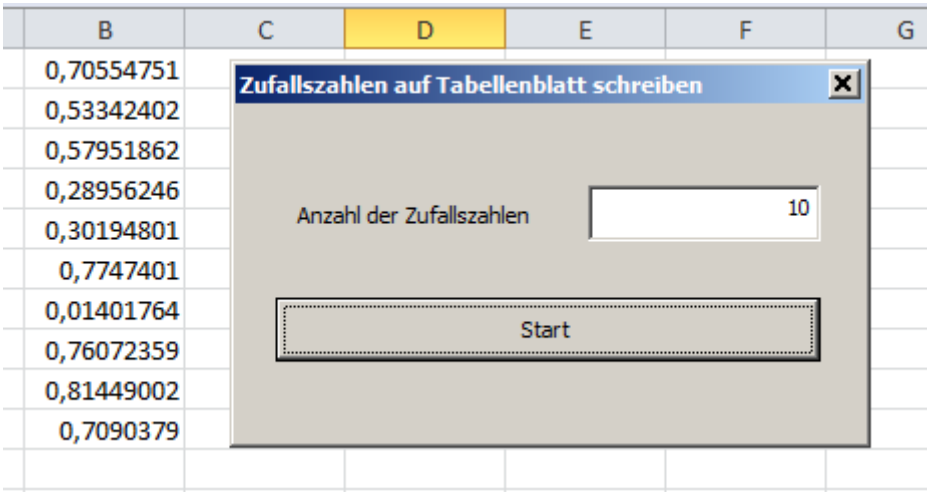
In der zweiten Schicht ist das Modul Zufall gegenüber der ersten Version völlig unverändert. Das ebenfalls zur zweiten Schicht gehörige Modul HR enthält allgemein verwendbare Funktionen, welche bei der Validierung von Eingaben hilfreich sind.

```
Public Function istGanzzahl(ByVal z As Variant) As Boolean
    If Not IsNumeric(z) Then
        istGanzzahl = False
    Else
        If z - Int(z) = 0 Then
            istGanzzahl = True
        Else
            istGanzzahl = False
        End If
    End If
End Function

Public Function istNatuerlicheZahl(ByVal z As Variant) As Boolean
    If Not istGanzzahl(z) Then
        istNatuerlicheZahl = False
        Exit Function
    End If
    istNatuerlicheZahl = CLng(z) > 0
End Function
```

3.5.7 Eine Anwendung mit gemischtem UI

Die dritte Version unseres Beispiels hat ein gemischtes UI. Der Benutzer gibt die gewünschte Anzahl von Zufallszahlen in ein Formular ein und stößt dort auch die Generierung an. Die Ausgabe der generierten Zahlen geschieht auf einem Tabellenblatt (s. Bild).



Gegenüber der ersten Version hat diese den Vorteil, dass die Benutzerführung straffer ist und weniger Fehlerquellen eröffnet. Aber auch gegenüber der zweiten Version gibt es Vorteile: die generierten Zahlen können besser betrachtet werden, und sie sind leichter weiter zu verwenden.

Wir haben es auch hier mit einer Zwei-Schichten-Architektur zu tun. In der Grobbetrachtung der Schichten stimmt das Programm mit der zweiten Version überein:

Formularmodul	
Zufall	HR

Die untere Schicht entspricht auch im Detail der unteren Schicht der zweiten Version. Die Unterschiede gegenüber der zweiten Version manifestieren sich im Code der Ereignisprozedur für die Schaltfläche Start im Formularmodul:

```
Private Sub StartBtn_Click()
  Dim n As Long
  Dim r() As Double
  If ValidierungOK Then
    n = CLng(Me.AnzahlTBx.Text)
    Worksheets("Tabelle1").Cells.Clear
    r = Zufall.ZufallszahlenErzeugen(n)
    Worksheets("Tabelle1").Range("B1:B" & n) = r
  End If
End Sub
```

Die Zuweisung des Arrays mit den erzeugten Zufallszahlen an den Bereich des Tabellenblatts, in dem sie angezeigt werden sollen, kann en bloc in einer einzigen Anweisung geschehen.

Beachten Sie, dass der inhaltliche Kern des Programms im Modul Zufall in allen drei Versionen derselbe ist. Dies ist der Schichtenarchitektur zu verdanken, welche die Benutzerführung strikt vom Kern (der Programmlogik) trennt.

3.5.8 Ein Formular starten

Haben wir eine Anwendung geschrieben, dessen UI von Formularen Gebrauch macht, so stellt sich die Frage, wie wir dem künftigen Benutzer dieses Formular zugänglich machen können. Wir können ihm sicherlich nicht zumuten, in den VBE zu wechseln und dort das Formular über die Option Ausführen zu starten.

Es gibt mehrere Möglichkeiten, und diese sind unterschiedlich schön und unterschiedlich aufwändig für den Entwickler. Die gängigsten sind:

1. Aufnahme einer Schaltfläche zum Start des Formulars in ein Tabellenblatt.
2. Automatisches Anzeigen des Formulars nach dem Öffnen der Arbeitsmappe. Diese Option wird meist mit der ersten kombiniert, denn der Benutzer benötigt eine Möglichkeit, das Formular wieder zu öffnen, falls er es geschlossen hat.
3. Aufnahme einer Menüleiste (command bar), einer Schaltfläche oder eines Drop-down-Menüs in die Registerkarte Add-Ins. Diese Möglichkeit ist eigentlich ein Überbleibsel aus Excel 2003. Sie wird aber weiterhin unterstützt, damit auch ältere Anwendungen mit den neuen Excel-Versionen harmonieren. In den neuen Excel-Versionen ab 2007 wird das eingefügte Steuerelement in der Registerkarte Add-Ins platziert.
4. Aufnahme eines Befehls zum Öffnen des Formulars in eine neue Registerkarte. Diese Option ist wohl die schönste, aber auch die aufwändigste.

Die Lösungen 1 und 2 sind schnell realisiert. Sie empfehlen sich vor allem für kleinere Programme. Lösung 3 erfordert schon erheblich mehr Aufwand, und Lösung 4 ist derart aufwändig, dass man dafür schon besser spezielle Werkzeuge zum Einsatz bringt. Man wird daher die vierte Möglichkeit speziell dann einsetzen, wenn man ein größeres Softwarepaket erstellt hat, das aus diversen Einzeloptionen besteht. Wir betrachten im Folgenden nur die beiden ersten Möglichkeiten.

Starten mit Schaltfläche

Eine Schaltfläche können wir mit Hilfe des Befehls Einfügen der Registerkarte Entwicklertools auf dem Tabellenblatt platzieren. Führen wir dann, nach dem Ändern des Namens und der Beschriftung, noch im Entwurfsmodus einen Doppelklick auf die Schaltfläche durch, so wird im Modul des Tabellenblatts der Rahmen für eine Ereignisprozedur generiert, also z.B.:

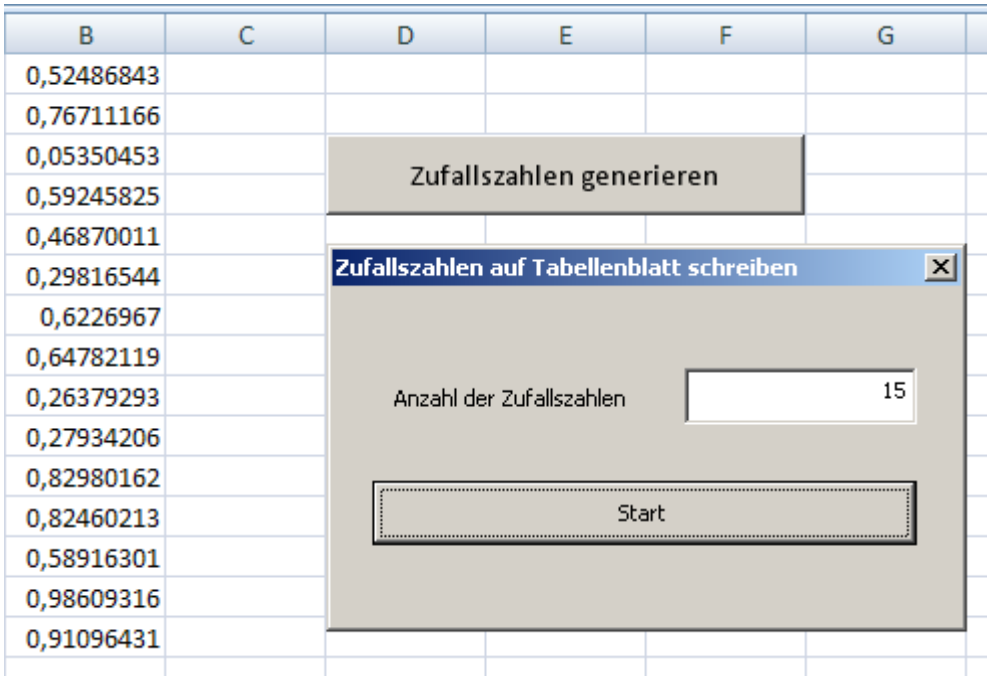
```
Private Sub ZZStartBtn_Click()

End Sub
```

Diesen Rahmen füllt man nun mit dem Code zum Aufbau und zum (hier modalen) Anzeigen des Formulars:

```
Private Sub ZZStartBtn_Click()
  Dim zzform As ZufallszahlenForm
  Set zzform = New ZufallszahlenForm
  zzform.Show
End Sub
```

Das folgende Bild zeigt die Schaltfläche inmitten der Version mit dem gemischten UI:



Starten beim Öffnen der Arbeitsmappe

Damit das Formular nach dem Öffnen der Arbeitsmappe ohne Zutun des Benutzers angezeigt wird, fügen wir die folgende Ereignisprozedur in das Modul der Arbeitsmappe ein:

```
Private Sub Workbook_Open()
  Dim zzform As ZufallszahlenForm
  Set zzform = New ZufallszahlenForm
  zzform.Show
End Sub
```

3.6 Objektorientierte Programmierung (OOP) und das Excel-Objektmodell

Das Excel-Objektmodell verschafft dem VBA-Programmierer Zugriff auf alle Bestandteile einer Excel-Arbeitsmappe. Wie der Name schon andeutet, beruht es (mehr oder weniger) auf den Grundsätzen der objektorientierten Programmierung (OOP). Wir machen deshalb zunächst einen kurzen Ausflug in die OOP, bevor wir uns speziell dem Objektmodell widmen.

3.6.1 Allgemeine Prinzipien der Objektorientierten Programmierung

Objekte sind die grundlegenden Bestandteile objektorientierter Programme. Ein Objekt kann Aufträge entgegennehmen und ausführen. Im Verlauf einer Programmausführung können viele solcher Auftragserteilungen erfolgen. Häufig fungiert dabei ein Objekt als Koordinator und delegiert Aufgaben an die anderen Objekte.

Bevor ein Objekt Aufträge erhalten kann, muss es erst mit Hilfe eines **Konstruktoraufrufs** aus einer (Objekt-)**Klasse** erzeugt werden. Die Klasse ist ein Baumuster für Objekte. In ihr ist genau festgelegt, welche Aufträge (**Methoden**) die Objekte der Klasse ausführen können und wie dies im Einzelnen geschieht. Auch ist in der Klasse festgehalten, welche Daten den Objekten zur Ausführung der Methoden übergeben werden müssen und welche Daten sie sich dauerhaft merken können.

Zur Verständnis des Objektbegriffs ist die **Analogie** mit einem menschlichen Sachbearbeiter nützlich. Wie ein Sachbearbeiter auch, hat ein Objekt einen bestimmten **Tätigkeitsbereich** (die Methoden) und kann zur Ausführung einer Tätigkeit (einer Methode) **Informationen** benötigen, welche ihm bei der Auftragserteilung übergeben werden (die Parameter der Methode). Und so wie ein Sachbearbeiter kann auch ein Objekt über ein **Gedächtnis** verfügen, das Informationen über die Ausführung eines einzelnen Auftrags hinaus speichern kann.

Das Gedächtnis eines Objekts wird durch seine Instanzenvariablen bzw. Eigenschaften (VBA: **Properties**) verkörpert. Diese behalten die ihnen zugewiesenen Werte, so lange das Objekt existiert, in den meisten Fällen also bis zur Beendigung der Programmausführung.

Objekte sind also nicht nur Aufgabenträger, sondern auch Datenspeicher. Man kann diese Dualität der Objekte griffig zur folgenden „Formel“ zusammenfassen:

Objekte = Daten + Methoden

Dieses Verständnis des Objektbegriffs hat im Übrigen eine enge Verwandtschaft mit dem Begriff des **Moduls vom Typ B**, der weiter oben eingeführt wurde. Module vom Typ B repräsentieren ein Objekt oder eine Datenstruktur. Man kann solche Module sowohl als Standardmodule als auch als Klassenmodule realisieren. Der entscheidende Unterschied zwischen den beiden Formen ist, dass es von einem Standardmodul immer nur ein Exemplar geben kann, während man aus einer Klasse beliebig viele Objekte erzeugen kann. Auf die Sachbearbeiter-Analogie bezogen heißt dies: nimmt man ein Standardmodul vom Typ B, so gibt es nur einen Sachbearbeiter mit Gedächtnis, nimmt man ein Klassenmodul (eine Klasse), so kann man viele Sachbearbeiter haben, von denen jeder sein eigenes Gedächtnis besitzt.

Nicht alle in einem Programm vorkommenden Klassen entsprechen dem Typ B, also der obigen Formel (Daten + Methoden). Manche Klassen sind auch vom **Typ A** oder vom **Typ C**, bestehen also nur aus einer Sammlung von Funktionen und haben keine Instanzenvariablen. In diesem Fall gibt es keinen substanziellen Unterschied zwischen Standardmodulen und Klassenmodulen. Lediglich die Benutzung innerhalb des Programms unterscheidet sich leicht.

3.6.2 Umgang mit Objekten in VBA

Wie Klassen von Grund auf programmiert werden können, wird in einigen Fallstudien am Ende des Kapitels gezeigt. Hier betrachten wir nur, wie man aus einer bereits vorhandenen Klasse ein Objekt erzeugt, ihm Aufträge erteilt und seinen Properties Werte zuweist bzw. diese Werte liest. Die betrachteten Objekte sind dabei Formulare der bereits aus einem früheren Beispiel bekannten Klasse PersonenForm. Formulare gehören zwar nicht zum Excel-Objektmodell (sie sind in allen Office-Anwendungen gleichermaßen verfügbar), aber der Umgang mit Formular-Objekten entspricht dem im Excel-Objektmodell. Eine wichtige Abweichung gibt es aber: die Art und Weise, wie neue Objekte erzeugt werden. Hierauf kommen wir später zu sprechen.

```
'zwei Variablen der Klasse PersonenForm werden deklariert
  Dim pf1 As PersonenForm
  Dim pf2 As PersonenForm

'den Variablen werden zwei durch Konstruktoraufrufe erzeugte Objekt zugewiesen
  Set pf1 = New PersonenForm
  Set pf2 = New PersonenForm

'es folgt der Auftrag an die Objekte, sich zu zeigen; dies geschieht durch Aufruf
'der Methode Show. Dem Aufruf wird hier der Parameterwert vbModeless mitgegeben
  pf1.Show vbModeless
  pf2.Show vbModeless

'die Property Width (Formularbreite) wird auf 500 gesetzt
  pf1.Width = 500

'die Property Width der Schaltfläche CloseBtn wird auf 120 gesetzt
  pf1.CloseBtn.Width = 120
```

3.6.3 Das Excel-Objektmodell

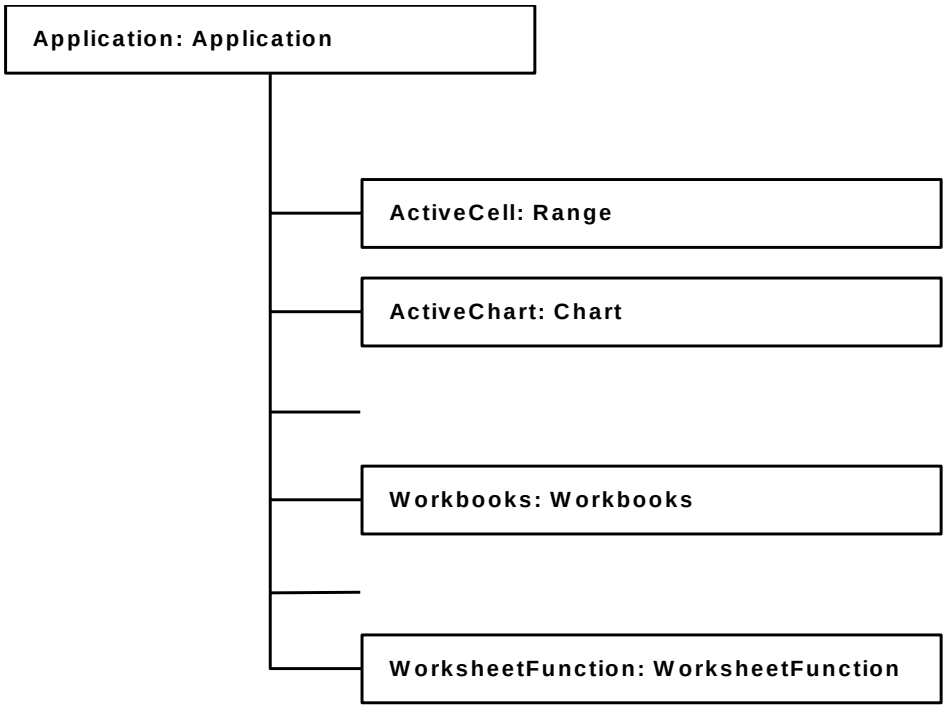
Das folgende Bild zeigt fragmentarisch den Aufbau des Excel-Objektmodells. Technisch gesehen und in der Sprache der OOP ausgedrückt ist das Objektmodell eine vielstufige Komposition. Von einer Komposition spricht man, wenn in ein Objekt (bzw. eine Klasse) andere Objekte eingebettet sind.

Im Excel-Objektmodell gibt es ein allumfassendes „Oberobjekt" mit dem Namen Application. Application repräsentiert die Gesamtheit aller Excel-Arbeitsmappen auf dem Rechnersystem. In dieses Oberobjekt sind neben diversen primitiven Daten auch viele Objekte als Komponenten eingebettet. Technisch ist diese Einbettung mit Hilfe von Instanzenvariablen realisiert. Im VBA-Sprachgebrauch heißen diese Instanzenvariablen auch **Properties** (englisch: property, hier im Sinne von Eigenschaft)

Diese Komponenten bzw. Properties können, falls sie Objekte sind, selbst wieder Objekte als Komponenten enthalten. Und diese Komponenten sind teilweise auch Objekte und enthalten

Objekte als Komponenten. So geht es weiter, über viele Stufen hinweg. Application ist also ein hierarchisch geschachteltes, höchst komplexes Objekt.

Das Bild veranschaulicht den Aufbau dieses Objekts. Kästchen repräsentieren Objekte, die Beschriftung enthält links jeweils den Namen des Objekts, rechts seine Klasse. Die Graphik zeigt nur einen winzigen Teil der Komponenten von Application, und sie zeigt auch nur die oberste Ebene der hierarchischen Aufteilung. Nicht enthalten sind auch die vielen Properties, die keine Objekte sind, sondern einen einfachen Datentyp besitzen.



Es fällt auf, dass im Modell manchmal der Name des Objekts (links) mit dem Namen der Klasse (rechts) identisch ist. Diese Namensgleichheit beginnt schon auf der obersten Ebene, beim Objekt Application. Die Schöpfer der Hierarchie haben diese Taktik der Namensgleichheit bewusst verfolgt, um den Umgang mit Objekten und Klassen zu „vereinfachen“. Namensgleichheit wird immer dann verwendet, wenn nicht die Gefahr der Verwechslung zwischen Objekten gleichen Namens droht. Dies ist in den Fällen so, wo zu jedem Zeitpunkt nur ein Objekt mit dem betreffenden Namen ansprechbar ist. Meiner Meinung nach ist dies keine gute Taktik, denn man behindert eher das Verständnis, wenn man die Dinge einfacher macht, als sie sind. Da wir aber keinen Einfluss auf diese Benennungen haben, müssen wir damit zurechtkommen.

Qualifizierte Benennung von Objekten mit Hilfe der Punktnotation

Gemäß Bild ist z.B. das Objekt ActiveCell eine Property des Objekts bzw. der Klasse Application. Eine wichtige Eigenschaft von ActiveCell ist ihr aktueller Wert (Value). Diesen Wert könnte man z.B. folgendermaßen auslesen (vorausgesetzt, die Zelle enthält eine Zahl):

```
Dim z As Double
z = CDbl(Application.ActiveCell.Value)
```

Mit der **Punktnotation**, wie hier in Application.ActiveCell.Value, kann man beliebige Objekte oder Werte aus der umfangrechen Hierarchie des Objektmodells benennen. Die Benennung beginnt mit dem Gesamtobjekt Application, nach einem Punkt folgt dann jeweils die benötigte Property auf der nächst niedrigeren Ebene. Da die Hierarchie sehr tief ist, können solche Benennungen sehr lang werden.

Diese Art, ein Objekt durch Punktnotation vollständig und eindeutig zu adressieren, wird auch **Qualifizierung** genannt.

Unqualifizierte Verwendung von Benennungen und die Objektsammlung <Global>

Eine Reihe von häufig gebrauchten Objekten wird zusätzlich in der Objektsammlung <**Global>** zur Verfügung gestellt. Diese Objekte kann man direkt ansprechen, muss also nicht den Weg über die ganze Hierarchie gehen. Da ActiveCell auch in <Global> enthalten ist, führt auch die folgende Version zum Ziel:

```
Dim z As Double
z = CDbl(ActiveCell.Value)
```

Bei Objekten, welche nicht in <Global> aufgeführt sind, soll man stets qualifiziert benennen. Tut man dies nicht, adressiert man also **unqualifiziert**, so wird das Programm unberechenbar. Manchmal wird es vielleicht das erwartete Ergebnis bringen, manchmal ein anderes, und in weiteren Fällen wird es mit einer Fehlermeldung aussteigen.

Betrachten wir die folgende Prozedur, welche den Inhalt einer Zelle eines Arbeitsblatts ausgeben soll:

```
Public Sub test()
    MsgBox Cells(1, 1)
End Sub
```

Das VBA-System führt diese Prozedur ohne Murren aus, aber mit wechselnden Ergebnissen. Weshalb ist das so? Die Property Cells, welche die Gesamtheit der Zellen eines Arbeitsblatts beinhaltet, gibt es in jedem Arbeitsblatt einer Arbeitsmappe. Wenn aber ein Name nicht eindeutig ist, behilft sich VBA mit einer Auswahlregel. In diesem Fall besagt diese Regel, dass die Zellen des gerade aktiven Arbeitsblatts genommen werden, wenn nichts anderes angegeben ist. Mit anderen Worten: das Ergebnis hängt davon ab, welches Arbeitsblatt Sie oder irgendwelche anderen Personen vorher betrachtet haben.

Wenn Sie solche Fehler vermeiden wollen, müssen Sie Namen immer qualifizieren. Eine passende Anweisung würde z.B. lauten:

```
MsgBox  Application.Worksheets("Tabelle1").Cells(1,1)
```

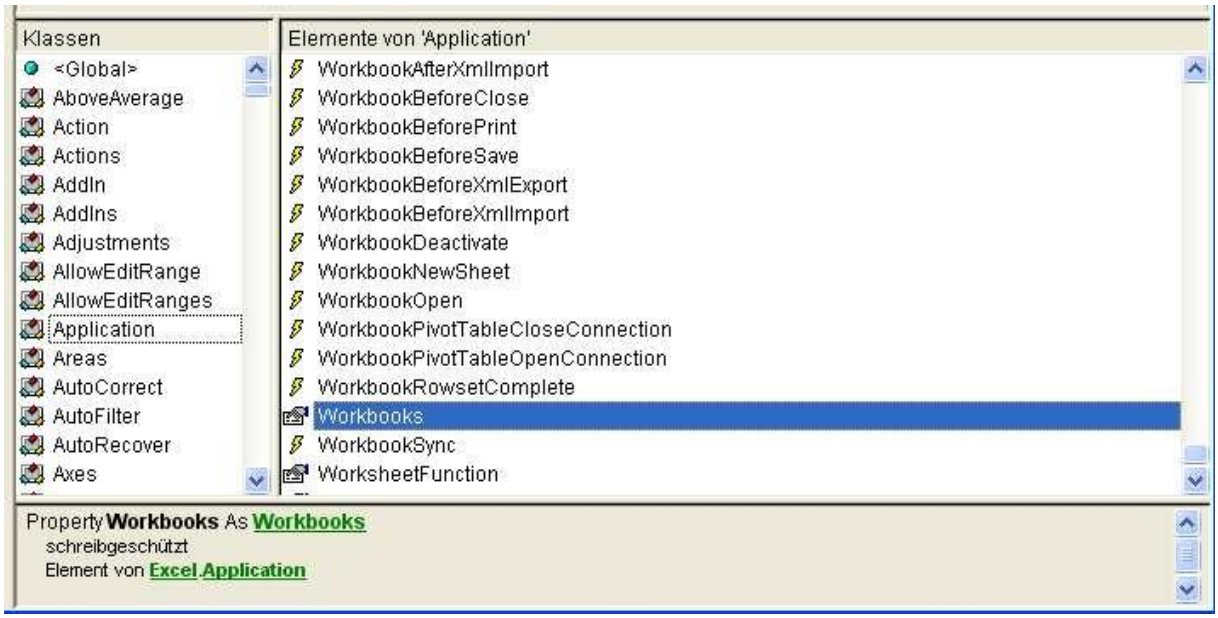
Da aber Worksheets in der Sammlung <globals> enthalten ist, können Sie guten Gewissens auch die folgende, kürzere Formulierung verwenden:

```
MsgBox  Worksheets("Tabelle1").Cells(1,1)
```

Ein Gang durch die Hierarchie mit Hilfe des Objektkatalogs

Auskunft über die vollständige Hierarchie gibt der sogenannte **Objektkatalog**, der in der Entwicklungsumgebung eingesehen werden kann. Der Objektkatalog, eigentlich ein Klassenkatalog, enthält nicht nur die Komponenten (properties) der einzelnen Klassen, sondern auch ihre Methoden und Ereignisse, ist also ein sehr umfassendes Nachschlagewerk.

Anhand der folgenden beiden Bilder wird veranschaulicht, wie man sich mit Hilfe dieses Katalogs den Weg durch die Hierarchie der Objekte bzw. Klassen suchen kann. Die Bilder geben auch einen Eindruck von der Kompliziertheit dieses Modells.



Die linke Spalte des Objektkatalogs enthält die Liste der Klassen, die in diesem Modell vorkommen. Selektiert man die **Klasse Application**, d.h. die Klasse des Objekts an der Spitzer der Hierarchie, so sieht man in der rechten Spalte alle Bestandteile dieser Klasse. Nicht nur die Properties sind hier aufgeführt, sondern auch die Methoden und die Ereignisse, die innerhalb dieser Klasse ausgelöst werden können.

Nehmen wir an, wir möchten mehr über die **Property Workbooks** erfahren. Diese Property ist eine sehr wichtige, denn sie enthält eine Kollektion (Collection) aller Arbeitsmappen, die auf dem Rechner liegen. Selektieren wir auf der rechten Seite den Eintrag Workbooks, so sehen wir im unteren Bereich des Katalogs einige Details über diese Property. Dazu gehört auch ihre Klasse. In diesem Fall hat die Klasse denselben Namen wie die Property selbst, also Workbooks.

Wie die **Klasse Workbooks** aufgebaut ist, erfahren wir, wenn wir auf den grünen unterstrichenen Eintrag Workbooks klicken. Die Auflistung der Klassenbestandteile auf der rechten Seite des Katalogs, der bisher die Bestandteile der Klasse Application angezeigt hat, wird nun durch eine Auflistung der Bestandteile der Klasse Workbooks ersetzt. Mit Hilfe dieser Auflistung können wir nun die Bestandteile dieser Klasse weiter erkunden.

Da gibt es z.B. die **Property Item**. Item bezeichnet ein Element der Workbook-Kollektion, die in der Klasse Workbooks enthalten ist. Dementsprechend lautet die am Fußende angegebene Klasse von Item auch **Workbook**. Hinter dem Namen Item versteckt sich allerdings nicht nur ein einziges Workbook, sondern die gesamte Kollektion. Die einzelnen Workbooks werden durch Indizes bzw. vom Benutzer vergebene Namen unterschieden, also Item(1), Item(2), … oder Item(“wrkbk1“), Item(“wrkbk2“) usw.

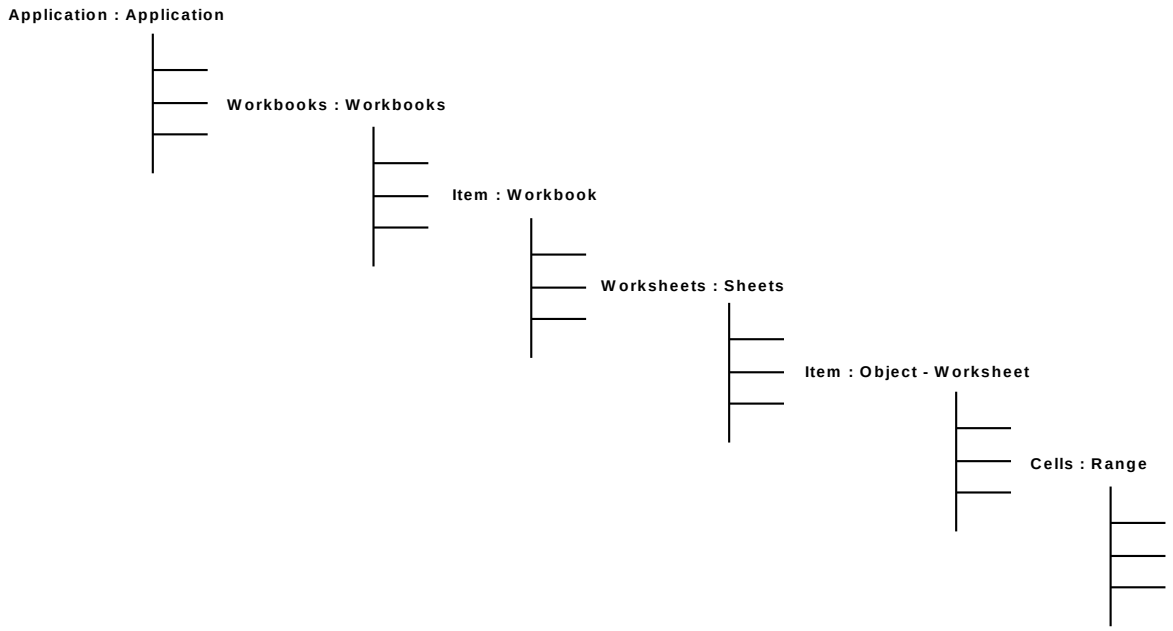
Wir können das Spiel nun weiter treiben und auch die **Klasse Workbook** erkunden, indem wir auf den Eintrag am Fußende des Katalogs klicken. Die rechte Seite des Katalogs zeigt dann die Bestandteile der Klasse Workbook.

Eine wichtige Property dieser Klasse ist Worksheets, denn Worksheets enthält eine Kollektion der Arbeitsblätter der Arbeitsmappe. Klicken wir in der Liste auf den Eintrag Worksheets, so sehen wir unten, dass die Klasse dieser Property Sheets heißt.

Die **Klasse Sheets** ist so konzipiert, dass sie nicht nur Arbeitsblätter (Klasse Worksheet) enthalten kann, sondern auch auch Diagrammblätter (Klasse Chart). Das merkt man auch, wenn man sich die Bestandteile der Klasse Sheets ansieht. Die **Property Item**, welche die Elemente der Kollektion repräsentiert, ist nicht von der Klasse Worksheet, wie man erwarten könnte, sondern von der Klasse Object. Object ist eine Art Überklasse, welche Objekte jeglicher Art umfasst, also quasi das Variant der OOP. Im Zusammenhang mit der Property Worksheets bezeichnet Item allerdings nur Objekte der Klasse Worksheet. Wollen wir also den Aufbau der einzelnen Items näher erkunden, so müssen wir die Klasse Worksheet ansteuern (s.Bild unten).

Suchen wir im linken Teil des Objektkatalogs die Klasse Worksheet auf und klicken auf den Eintrag, so sehen wir auf der rechten Seite die Bestandteile der Klasse. Unter diesen Bestandteilen ist auch die Poperty Cells. Cells bezeichnet die Gesamtheit der Zellen eines Arbeitsblatts. Cells ist von der **Klasse Range**.

Wie das unten stehende Bild andeutet, ist damit die Hierarchie noch nicht in voller Tiefe durchschritten. Auch die Klasse Range hat viele Bestandteile, darunter auch solche, die selbst wieder Bestandteile haben.



Angenommen, wir möchten innerhalb einer Prozedur die Werte aller Zellen eines bestimmten Arbeitsblatts löschen. Der Objektkatalog sagt uns, dass die Klasse Range eine Methode Clear besitzt, die genau das tut. Wie müssen wir die Anweisung formulieren? Nehmen wir an, die Arbeitsmappe heißt „Beispiel“, und bei dem Arbeitsblatt handelt es sich um „Tabelle1“. Dann müssten wir bei vollständiger Qualifizierung und wenn wir dem oben aufgezeichneten Pfad folgen, so formulieren:

```
Application.Workbooks.Item("Beispiel “).Worksheets.Item("Tabelle1").Cells.Clear
```

Glücklicherweise gibt es in vielen Fällen auch kürzere Formulierungen. Ist die Arbeitsmappe „Beispiel“ mit der gerade aktiven Arbeitsmappe identisch, dann können wir uns zunutze machen, dass die Sammlung <Global> einen Eintrag Worksheets enthält, welcher die Arbeitsblätter der aktiven Arbeitsmappe bezeichnet. Wir kommen damit zu der kürzeren und trotzdem eindeutigen Formulierung:

```
Worksheets.Item("Tabelle1").Cells.Clear
```

Im nächsten Abschnitt werden Sie sehen, dass man die Anweisung ohne Gefahr noch weiter verkürzen kann.

Standardeigenschaften und Standardwerte

Manche Properties werden viel häufiger gebraucht als andere. Dies hat die Schöpfer von VBA dazu bewogen, bei manchen Klassen jeweils eine solche Property zur Standard-Property (default property) zu erklären. Immer dann, wenn der Programmierer bei der Adressierung eines Objekts nicht explizit eine Property angibt, nimmt das System an, dass die Standard-Property gemeint ist. Für Objekte der Klasse Worksheets ist Item die Standard-Property. Dies bedeutet, dass die oben zitierte Anweisung

```
Worksheets.Item("Tabelle1").Cells.Clear
```

ohne Gefahr reduziert werden kann zu:

```
Worksheets("Tabelle1").Cells.Clear
```

Eine ebenfalls häufig in Anspruch genommene Standard-Property ist die der Klasse Range. Jede Zelle eines Arbeitsblatts und jeder Zellenbereich ist von dieser Klasse. Die Standard-Property von Range ist Value, bezeichnet also den Wert, der in einer Zelle steht. Die folgenden beiden Anweisungen bedeuten demnach dasselbe:

```
MsgBox Worksheets("Tabelle1").Cells(1,1).Value

MsgBox Worksheets("Tabelle1").Cells(1,1)
```

Auch in Bezug auf die Werte gibt es bei manchen Properties Standard-Annahmen. Beispielsweise hat die Property ScreenUpdating der Klasse Application, die vom Typ Boolean ist, den Standardwert True. Wenn der Programmierer nichts anderes verfügt, so gilt dieser Wert, und der Bildschirm wird nach jeder Veränderung aktualisiert. Nur wenn der Programmierer diesen Wert explizit ändert, z.B. durch die Anweisung

```
Application.ScreenUpdating = False
```

wird vom Standardwert abgegangen. Der Bildschirm wird dann erst wieder aktualisiert, so bald die Anweisung

```
Application.ScreenUpdating = True
```

abgesetzt wurde.

Allerdings sollte man sich als Programmierer nur dann auf Standardwerte verlassen, wenn auch sicher ist, dass die betreffende Property noch mit dem Standardwert besetzt ist. Im Zweifelsfall ist es besser, den Wert explizit durch eine zusätzliche Anweisung zu setzen. Andernfalls wird die **Programmdurchführung** zum Glücksspiel.

3.6.4 Programmierbeispiele

Es wird nun eine kleine Auswahl von Beispielen präsentiert, welche einen Eindruck vom Umgang mit der OOP und speziell dem Excel-Objektmodell vermitteln sollen. Der Umgang mit der besonders wichtigen Klasse Range folgt weiter unten in einem gesonderten Abschnitt.

Beispiel 1: Arbeitsblätter auflisten

Mit der folgenden Prozedur greifen wir auf die Property Worksheets des Application-Objekts zu und listen alle Worksheets des aktiven Workbooks mit ihren Namen auf. Da Worksheets eine Kollektion bezeichnet, können wir eine Schleife mit For each benutzen. Das enthebt uns der Notwendigkeit, die Anzahl der in der Kollektion enthaltenen Worksheets zu ermitteln.

```
Public Sub ArbeitsblaetterAuflisten()
  Dim w As Worksheet
  Dim s As String
  s = ""
  For Each w In Application.Worksheets
    s = s & w.Name & vbLf
  Next w
  MsgBox s
End Sub
```

Da die Kollektion Worksheets, welche die Arbeitsblätter der aktuellen Arbeitsmappe beinhaltet, auch in <Global> enthalten ist, kann man den Kopf der For-each-Schleife auch etwas kürzer so fassen:

```
For Each w in Worksheets
```

Beispiel 2: die aktive Arbeitsmappe schließen

In der folgenden Prozedur benutzen wir die Methode Close der Klasse Workbook, um die Arbeitsmappe zu schließen, in der wir gerade arbeiten. Wir sprechen dabei die Property ActiveWorkbook des Application-Objekts an, welche vom Typ Workbook ist. Da ActiveWorkbook in <Global> enthalten ist, müssen wir es im Aufruf nicht qualifizieren:

```
Public Sub schliessen()
    ActiveWorkbook.Close
End Sub
```

Beachten Sie, dass die Anwendung selbst nicht geschlossen wird, sondern nur die aktive Arbeitsmappe. Nach Durchführung der Prozedur befinden Sie sich vor einer neuen, leeren Arbeitsmappe.

Beispiel 3: ein neues Arbeitsblatt anlegen

Dies ist eine Aufgabe, die im Rahmen der VBA-Programmierung häufig ansteht. In einem früheren Abschnitt wurde gezeigt, wie in der OOP mit Hilfe des Konstruktors new ein Objekt einer Klasse angelegt werden kann.

Für das Anlegen eines Arbeitsblatts und auch für viele andere Objekte innerhalb des Excel-Objektmodells ist der Konstruktor new allerdings nicht anwendbar. Es muss ein anderes Verfahren herangezogen werden. Der Grund hierfür ist, dass Arbeitsblätter (und viele andere Excel-Objekte) nicht alleine für sich existieren können, sondern nur innerhalb der speziellen Kollektionen, welche im Objektmodell dafür vorgesehen sind.

Arbeitsblätter, also Objekte vom Typ Worksheet, sind in der Property Worksheets der Arbeitsmappe, bzw. der darin enthaltenen Kollektion von Arbeitsblättern, enthalten. Die Property Worksheets hat, wie bereits weiter oben dargelegt, die Klasse Sheets. Diese Klasse enthält eine Methode Add, welche erlaubt, ein neues Objekt der Art Worksheet zu erzeugen und dieses Objekt gleichzeitig der in Property Worksheets enthaltenen Kollektion von Arbeitsblättern hinzuzufügen.

In der folgenden Prozedur wird der Kollektion Worksheets mit Hilfe dieser Methode Add ein Arbeitsblatt mit dem Namen „neuesArbeitsblatt“ hinzugefügt. Add ist eine Funktion, so dass man das neu geschaffene Arbeitsblatt einer Variablen vom Typ Worksheet zuweisen kann. In unserem Beispiel ist dies wichtig, weil wir dem Arbeitsblatt noch keinen Namen gegeben haben und somit ohne die Variable w nicht mehr darauf zugreifen könnten. Erst in der Zeile darunter wird der Name zugewiesen. Danach könnten wir über Worksheets(“neuesArbeitsblatt“) darauf zugreifen.

```
Public Sub neuesArbeitsblatt()
  Dim w As Worksheet
  Set w = Application.ActiveWorkbook.Worksheets.Add
  w.Name = "neuesArbeitsblatt"
End Sub
```

Etwas kürzer, aber weniger eingängig, hätte man es auch folgendermaßen formulieren können:

```
Public Sub neuesArbeitsblattKurz()
  Application.ActiveWorkbook.Worksheets.Add.Name = "neuesArbeitsblatt"
End Sub
```

Bei dieser Fassung ersparen wir uns die Variable w, denn die Namensvergabe wird mit der Erzeugung des Arbeitsblatts kombiniert, so dass dieses sofort danach über den Namen adressierbar ist.

Beispiel 4: Umbenennung eines Arbeitsblatts

Die folgende Prozedur benennt das Arbeitsblatt „Tabelle1“ um. Der neue Name ist „Benutzerbereich“.

```
Public Sub ArbeitsblattUmbenennen()
  Dim w As Worksheet
  Set w = Worksheets("Tabelle1")
  w.Name = "Benutzerbereich"
End Sub
```

Kürzer und ohne Hilfsvariable geht es, wenn wir das umzubenennende Arbeitsblatt über seinen alten Namen (hier: „neuesArbeitsblatt“) ansprechen:

```
Public Sub ArbeitsblattUmbenennen2()
  Worksheets("neuesArbeitsblatt").Name = "altesArbeitsblatt"
End Sub
```

Beispiel 5: Weniger schreiben müssen mit With

Benennungen in Punktnotation können manchmal ziemlich lang werden. Mit der Anweisung With kann man sich die Arbeit vereinfachen, wenn nacheinander mehrere Properties oder Methoden adressiert werden sollen, welche denselben Vorspann (dieselbe Qualifizierung) aufweisen. Die

folgende Prozedur schreibt in die drei ersten Spalten des Arbeitsblatts „altesArbeitsblatt“ die natürlichen Zahlen von 1 bis 10 und rechts daneben ihre Quadrate und Kubikzahlen:

```
Public Sub ZellenFuellen()
   Dim i As Integer
   With Worksheets(“altesArbeitsblatt”)
      For i = 1 to 10
         .Cells(i, 1) = i
         .Cells(i, 2) = i * i
         .Cells(i, 3) = i * i * i
      Next i
   End With
End Sub
```

Ohne die With-Anweisung müssten wir schreiben:

```
Public Sub ZellenFuellen()
   Dim i As Integer
      For i = 1 to 10
         Worksheets(“altesArbeitsblatt”).Cells(i, 1) = i
         Worksheets(“altesArbeitsblatt”).Cells(i, 2) = i * i
         Worksheets(“altesArbeitsblatt”).Cells(i, 3) = i * i * i
      Next i
End Sub
```

3.6.5 Range und nochmals Range

Die Bezeichnung Range wird im Objektmodell in zweierlei Bedeutung gebraucht:

- als Name einer Klasse
- als Name von Properties, also von Objekten

Range als Klasse

Sie ist eine der wichtigsten Klassen des Objektmodells für den Programmierer. Jede Zelle eines Arbeitsblatts, jeder Verbund von Zellen ist von diesem Typ.

Objekte vom Typ Range kann man als Programmierer nicht direkt erzeugen. Der Konstruktor new ist nicht anwendbar und es gibt auch keine dem Add der Klasse Sheets vergleichbare Ersatzmethode dafür. Ursache dieser Beschränkung ist, dass Range-Objekte immer nur innerhalb anderer Objekte existieren, nämlich innerhalb von Arbeitsblättern und größeren Range-Objekten. Der eigentliche Konstruktor für Range-Objekte ist die Methode Add der Klasse Worksheets, denn mit jedem neuen Arbeitsblatt entstehen auch die darin enthaltenen Zellen.

Deklariert man eine Variable vom Typ Range, so kann diese Variable Range-Objekte ganz unterschiedlichen Umfangs bezeichnen. Das Spektrum reicht von einer einzigen Zelle bis zur Gesamtheit aller Zellen eines Arbeitsblatts:

```
Dim r as Range
Set   r = Worksheets(“Tabelle1“).Cells(1,1)
Set   r = Worksheets(“Tabelle1“).Cells
```

Range als Property

Diverse Klassen des Excel-Objektmodells enthalten eine Property namens Range, so die Klasse Application, die Klasse Worksheet und die Klasse Range selbst auch. Diese Properties sind selbst wieder vom Typ Range.

Was diese Properties enthalten und wie sie zu benutzen sind, wird am besten an einem Beispiel deutlich. Wir wollen zwei Variablen von Typ Range deklarieren und ihnen die Property Range eines Arbeitsblatts zuweisen:

```
Dim r1 as Range, r2 as Range
Set   r1 = Worksheets(“Tabelle1“).Range("B2:C4")
Set   r2 = Worksheets(“Tabelle1“).Range("B25")
```

Aus dieser Zuweisung wird deutlich, dass diese Property keinen fest definierten Zellenbereich bezeichnet. Wir müssen in der Anweisung genau spezifizieren, welchen Bereich wir meinen, indem wir seine Adresse in der Parameterklammer angeben.

Die Property Range der Klasse Application funktioniert im Prinzip genauso. Während aber bei der Property eines Worksheet-Objekts klar ist, dass der herauszugreifende Zellenbereich aus diesem Arbeitsblatt ist, kann der Zellenbereich im Fall der Klasse Application aus jedem der Arbeitsblätter der Arbeitsmappe stammen. Eine Spezifizierung wie ("B2:C4") wäre nicht eindeutig und würde zu einem Fehler führen. Möglich sind dagegen folgende Anweisungen

```
Dim r as Range
Set   r = Application.Range("Tabelle1!B2:C4")
```

Weitere Beispiele zur Klasse Range

Die folgende Prozedur hat keinen Zweck als die Demonstration des Umgangs mit Range-Objekten. Die Anweisungen sind kommentiert und sollten gut verständlich sein. Studieren Sie die Beispiele. Vollziehen Sie sie möglichst am Rechner nach.

```
Public Sub Range_Ex()

  Dim w As Worksheet, r1 As Range, r2 As Range
  Set w = Worksheets("Tabelle1")

  ‘die folgenden 4 Anweisungen haben denselben Effekt
  Set r1 = w.Range("B2:C4")
  Set r1 = Application.Range("Tabelle1!B2:C4")
  Set r1 = Application.Range(w.Cells(2, 2), w.Cells(4, 3))
  Set r1 = Range("Tabelle1!B2:C4 “)

  r1.Cells(2, 1) = 5
  MsgBox r1.Cells(2, 1)       ‘gibt 5 aus
  MsgBox w.Cells(3, 2)        ‘dieselbe Zelle!

  ‘Daten einlesen
  r1(1, 1) = "x": r1(1, 2) = "y"
  Dim i As Integer, j As Integer
  For i = 2 To r1.Rows.Count
    For j = 1 To r1.Columns.Count
        r1.Cells(i, j) = CLng(InputBox("" & i & j))
        r1(i, j) = CLng(InputBox("" & i & j))          ‘Effekt wie Vorzeile
    Next j
  Next i

  'r2 wird ein Unterbereich von r1 zugewiesen
  Set r2 = Range(r1.Cells(2, 1), r1.Cells(3, 2))

  ‘wir benutzen for each, um alle Zellen von r2 zu besuchen
  Dim cell As Range
  Dim sum As Long
  sum = 0
  For Each cell In r2
     sum = sum + cell.Value
  Next
  MsgBox sum

  'wir benutzen die Properties Columns & Rows
  MsgBox WorksheetFunction.sum(r2.Columns(1))
  MsgBox WorksheetFunction.sum(r2.Rows(1))

  'kopieren von r1, verschiedene Formulierungen sind möglich:
  r1.Copy w.Cells(10, 10)  ‘die Angabe der oberen linken Ecke genügt
  r1.Copy w.Range("J10")
  Range(w.Cells(10, 10), w.Cells(12, 11)).Value = r1.Value
```

```
    'Gebrauch der Property current region
    Dim r4 As Range
    Set r4 = w.Cells(3, 3).CurrentRegion
    MsgBox r4.Rows.Count  &  "  "  &  r4.Columns.Count

End Sub
```

3.6.6 Der Makroaufzeichner-nützlich, aber kein Ersatz für eigene Programmierung

Mit dem Makroaufzeichner kann man eine Abfolge von Schritten, die ein Benutzer mit Excel macht, aufzeichnen. Der Makroaufzeichner benutzt zur Formulierung der Schritte VBA, so dass durch die Aufzeichnung eine VBA-Prozedur (ein „Makro“) entsteht. Lässt man dieses Makro ablaufen, so wiederholt man damit automatisch die während der Aufzeichnung vollzogenen manuellen Schritte.

Für den Programmierer kann der Makroaufzeichner, richtig angewandt, ein sehr nützliches Instrument sein. Wenn man beispielsweise auf einem Arbeitsblatt einen bestimmten Effekt hervorrufen will und nicht weiß, welche Eigenschaften man dafür verändern muss, so kann man sich diese Information schnell verschaffen, indem man den Makroaufzeichner einschaltet, die Operation manuell durchführt und danach das aufgezeichnete Makro studiert.

Wir betrachten hierzu zwei Beispiele. Während das erste dazu angetan ist, die Nützlichkeit des Aufzeichners zu demonstrieren, hat das zweite nur den Zweck, vor einer blauäugigen Anwendung zu warnen.

Beispiel 1: Verschmelzung von Zellen

Vier Zellen sollen verschmolzen werden. In die Mitte des so geschaffenen Zellverbunds soll eine Beschriftung eingefügt werden.

Der Makroaufzeichner liefert folgenden Code (etwas verkürzt):

```
Sub Makro3()
    Range("C16:D17").Select
    With Selection
        .HorizontalAlignment = xlCenter
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .IndentLevel = 0
        .ShrinkToFit = False
        .ReadingOrder = xlContext
        .MergeCells = False
    End With
    Selection.Merge
```

```
    With Selection
      .HorizontalAlignment = xlCenter
      .VerticalAlignment = xlCenter
      .WrapText = True
      .Orientation = 0
      .AddIndent = False
      .IndentLevel = 0
      .ShrinkToFit = False
      .ReadingOrder = xlContext
      .MergeCells = True
    End With
    ActiveCell.FormulaR1C1 = "Titel"
    Range("C18").Select
End Sub
```

Die Informationen, die in unserem Fall relevant sind, sind von mir fett hervorgehoben. Alles, was vor der Anweisung Selection.Merge steht, ist irrelevant. Es handelt sich um Anweisungen, welche die Situation vor der Verschmelzung beschreiben. Davon besteht der größte Teil aus der Aufzählung von Standardwerten. Auch das, was nach der Verschmelzungsanweisung kommt, sind zum Teil solche Aufzählungen, die wir nicht verändern müssen.

Beschränken wir uns auf das Wesentliche und beachten wir die Grundsätze guter Programmierung, so können wir für uns z.B. folgenden Code daraus entwickeln:

```
Public Sub Zellverschmelzung(ByVal r As Range, ByVal s As String)
    With r
      .Merge
      .HorizontalAlignment = xlCenter
      .VerticalAlignment = xlCenter
      .WrapText = True
      .Value = s
    End With
End Sub
```

Beispiel 2:

Ein Programmgenerator wie der Makroaufzeichner liefert gewöhnlich keinen Code, den man als gelungen im Sinne guter Programmierung bezeichnen könnte. Dies liegt in der Natur dieser Generatoren. Dieses Beispiel soll ausschließlich den Unterschied zwischen dem Programmierstil des Makroaufzeichners und korrektem Programmierstil demonstrieren.

Nehmen wir an, während einer Makro- Aufzeichnung hat der Benutzer folgende Schritte unternommen:

1. er hat im gerade aktiven Arbeitsblatt die Zelle B3 markiert und dort die Zahl 4 eingetragen
2. er hat die darunter liegende Zelle B4 markiert und dort die Zahl 5 eingetragen

3. dann ist er in die Zelle C5 gegangen und hat dort die Formel = B3 * B4 eingetragen
4. abschließend hat er die Zelle C5 durch Drücken des Abwärtspfeils verlassen

Daraus resultiert das folgende Makro

```
Sub Makro6()
  Range("B3").Select
  ActiveCell.FormulaR1C1 = "4"
  Range("B4").Select
  ActiveCell.FormulaR1C1 = "5"
  Range("C5").Select
  ActiveCell.FormulaR1C1 = "=R[-2]C[-1]*R[-1]C[-1]"
  Range("C6").Select
End Sub
```

Wir können darin unschwer die einzelnen Schritte des Benutzers wiedererkennen. Wir haben es mit einer Abfolge von Zweierkombinationen zu tun: der Benutzer aktiviert wiederholt eine Zelle und trägt dann dort etwas ein. So bezieht sich die zweite Anweisung auf die Zelle, welche in der ersten aktiviert wurde, die vierte Anweisung auf die Zelle, die in der dritten aktiviert wurde.

Man kann diese Art der Programmformulierung, bei der jedes Objekt, mit dem etwas getan werden soll zunächst aktiviert wird und dann auf das aktive Objekt Bezug genommen wird, als den **Makroaufzeichner-Programmierstil** bezeichnen.

Für eine qualitätsorientierte Programmierung, also eine, die auf Robustheit, Wartbarkeit und Wiederverwendbarkeit Wert legt, ist ein solcher Stil **untauglich**. Er führt dazu, dass die Anweisungen in ihren Wirkungen hochgradig voneinander abhängig und schwer interpretierbar sind. Besonders gravierend wird es, wenn die Aktivierung eines anzusprechenden Objekts nicht in der Zeile davor, sondern zwanzig oder fünfzig Zeilen vorher geschieht.

Außerordentlich gefährlich beim Makroaufzeichner-Stil ist es, wenn sich Zugriffe auf aktive Objekte beziehen, die gar nicht explizit aktiviert wurden. Das oben gezeigte Makro setzt z.B. voraus, dass das relevante Arbeitsblatt gerade aktiviert ist. Vorausgesetzte Aktivierungen, die nicht explizit erfolgten, sind die Ursache für den Mangel an Robustheit in vielen kursierenden VBA-Anwendungen.

Die folgende Prozedur zeigt, wie man dieselbe Aufgabe kürzer, besser verständlich und ohne Beeinträchtigung der Robustheit programmieren kann:

```
Public Sub MakroErsatz()
  With Worksheets("Tabelle1")
    .Range("B3").Value = 4
    .Range("B4").Value = 5
    .Range("C5").Formula = "= B3 * B4"
  End With
End Sub
```

Ergänzende Anmerkungen:
Vergleicht man mit dem aufgezeichneten Makro, so fallen noch weitere Unterschiede neben der dort verwendeten Aktivierungstechnik auf. Im aufgezeichneten Makro werden alle vom Benutzer vorgenommenen Eingaben der Eigenschaft FormulaR1C1 der jeweiligen Zelle zugewiesen, obwohl es sich um völlig unterschiedliche Eingaben handelt. Auch werden alle Eingaben als Strings aufgezeichnet,

obwohl ein Teil davon Zahlenwerte sind. Offensichtlich hat der Makroaufzeichner darauf verzichtet , genau zu ergründen, was das Ziel des Benutzers war. Er hat stattdessen Eigenschaften und Formate gewählt, die flexibel verwendbar sind.

Die selbst programmierte Prozedur ist dagegen trotz ihrer Kürze viel exakter und deshalb auch verständlicher. Zahlenwerte erscheinen dort als Zahlenwerte, und die Formel von Zelle C5 ist in einer erheblich besser lesbaren Notation formuliert.

3.7 Validierung von Eingaben: Vertiefung

Die simple und pauschale Art der Überprüfung von Eingaben, die im Hauptabschnitt 3.3 vorgestellt wurde, ist nur für Formulare mit wenigen Eingabefeldern tauglich. Größere Formulare erfordern differenziertere Methoden.

3.7.1 Überblick

Bei der Eingabe von Daten passieren erfahrungsgemäß viele Fehler. Lässt man ein Programm mit fehlerhaften Daten arbeiten, so sind dessen Ergebnisse in der Regel falsch. Oft stürzen Programme auch ab, wenn man sie mit unpassenden Daten füttert.

Deshalb prüft ein solides Programm die Daten, bevor es sie verarbeitet. Wird bei dieser Prüfung festgestellt, dass unter den Eingaben fehlerhafte sind, so muss der Benutzer dahin geführt werden, diese Werte zu korrigieren. Das Programm prüft also nicht nur die Eingaben, sondern stellt ihre Zulässigkeit sicher. Dieses Sicherstellen bzw. Gewährleisten zulässiger Eingaben nennt man **Validierung**.

Genauso wichtig wie die Prüfung bereits eingegebener Werte ist eine Benutzerführung, welche fehlerhaften Eingaben von vornherein entgegenwirkt. Wir können hier von einer Validierung im Voraus bzw. **Ex-ante-Validierung** sprechen, im Gegensatz zu der oben beschriebenen Validierung im Nachhinein, der **Ex-post-Validierung**.

3.7.2 Ex-post-Validierung

Zur Ex-post-Validierung wollen wir alle Aktivitäten zählen, die darauf abzielen, die Richtigkeit der Werte nach der Eingabe sicher zu stellen. Aber wann genau sollen wir das tun und wie stark sollen wir den Benutzer dabei führen?

Varianten der Ex-post-Validierung

Ein extremer Ansatz ist, unmittelbar nach der Eingabe eines einzelnen Zeichens zu überprüfen, ob es zum vorgesehenen Typ passt. Wir können dies die **zeichenbezogene Validierung** nennen. Bei ganz-

zahligen Datentypen ist die zeichenbezogene Validierung noch einfach: nur Ziffern und ggfls. ein Vorzeichen am Anfang sind erlaubt. Aber schon bei Double-Zahlen ist sie sehr schwierig. Außerdem lässt diese Art der Überwachung dem Benutzer kaum Luft zum Atmen. Schon bei einem einfachen Tippfehler, den er normalerweise selbst bemerkt und sofort verbessert, wird er sofort zur Ordnung gerufen. Der Benutzer empfindet dies als nervig und verliert die Lust an der Arbeit mit dem Programm.

Das andere Extrem ist die **Validierung en bloc**. Dabei lassen wir dem Benutzer zunächst völlige Freiheit bei seiner Arbeit im Formular. Erst wenn er dieses mit einem Klick auf die Schaltfläche <Speichern> (oder eine ähnliche) verabschiedet hat, folgt die Überprüfung, die dann im Fehlerfall in einer mehr oder weniger detaillierten Fehlerliste mündet.

Zwischen der zeichenbezogenen Validierung und der Validierung en bloc liegt die **feldbezogene Validierung**. Dabei validieren wir, sobald der Benutzer ein Eingabefeld verlässt, in das er einen Wert eingegeben hat. Die Reihenfolge der Eingaben kann er selbst bestimmen, und wir können ihm auch die Freiheit lassen, ein Feld zunächst leer zu lassen und erst dann zu bearbeiten, wenn er alle anderen Felder schon ausgefüllt hat.

Der **Nachteil der Validierung en bloc** gegenüber der feldbezogenen Validierung ist, dass der Benutzer erst sehr spät auf Fehler hingewiesen wird. Oft sind Felder in einem Eingabeformular einander ähnlich, so dass ein früher Fehlerhinweis beim ersten Feld dem Benutzer beim Ausfüllen der weiteren Felder geholfen hätte. Oder es gibt gar Abhängigkeiten zwischen Feldern. Andererseits kann die feldbezogene Validierung die Validierung en bloc nicht vollständig ersetzen. Nur diese kann Abhängigkeiten zwischen den Werten in den einzelnen Feldern berücksichtigen, und sie ist auch besser geeignet, das Ausfüllen von Muss-Feldern sicher zu stellen.

Optimal ist demnach eine **Kombination** aus feldbezogener Validierung und Validierung en bloc. Ausgenommen sind sehr kleine Formulare mit wenigen Eingabefeldern. Hier kann eine Validierung en bloc ausreichen. Im Grenzfall, wenn ein Formular nur ein Eingabefeld hat, fallen die beiden Validierungsarten ohnehin zusammen.

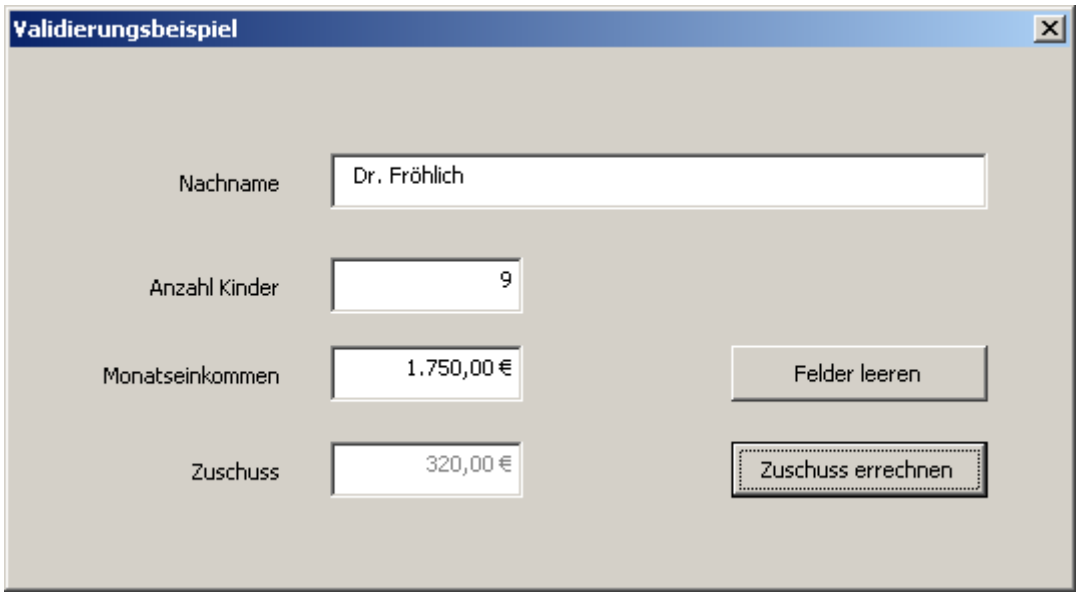
Beispiel: Zuschussberechnung

In diesem Beispiel wird eine Kombination aus feldbezogener Validierung und Validierung en bloc angewandt. Dabei weist die feldbezogene Validierung nur auf Fehler hin und erzwingt nicht deren Korrektur. Die abschließende Validierung en bloc prüft noch einmal alle Felder und stellt auch sicher, dass alle Mussfelder ausgefüllt sind.

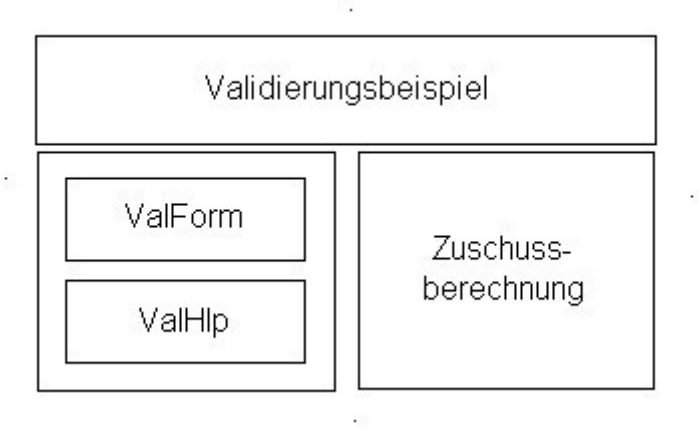
Das folgende Bild zeigt das einzige Formular des Programms. Es gibt drei Eingabefelder und ein Ausgabefeld (Zuschuss). Bezüglich der Eingaben gilt:

- beim Nachnamen wird ein String erwartet, der mit einem Buchstaben beginnen muss und außer Buchstaben nur wenige andere Zeichen enthalten darf (Leerzeichen, Bindestrich, Punkt, Apostroph)
- die Kinderzahl soll eine Ganzzahl im Bereich [0, 20] sein
- als Einkommen wird eine Kommazahl im Bereich [0, 100000] erwartet.

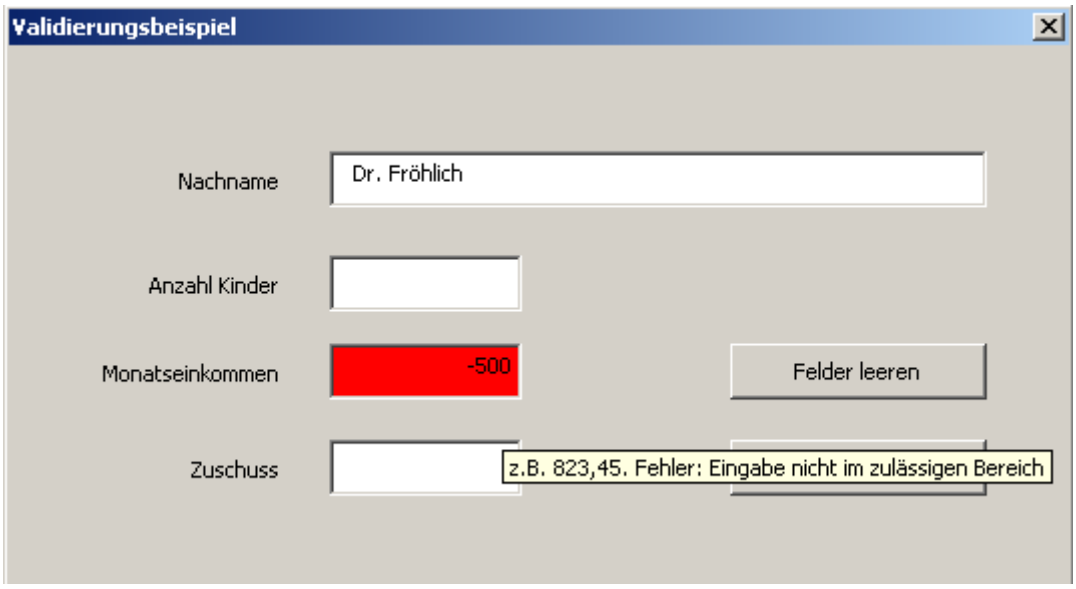
Das Eurozeichen in den Feldern Einkommen und Zuschuss wird den Zahlen vom Programm hinzugefügt.



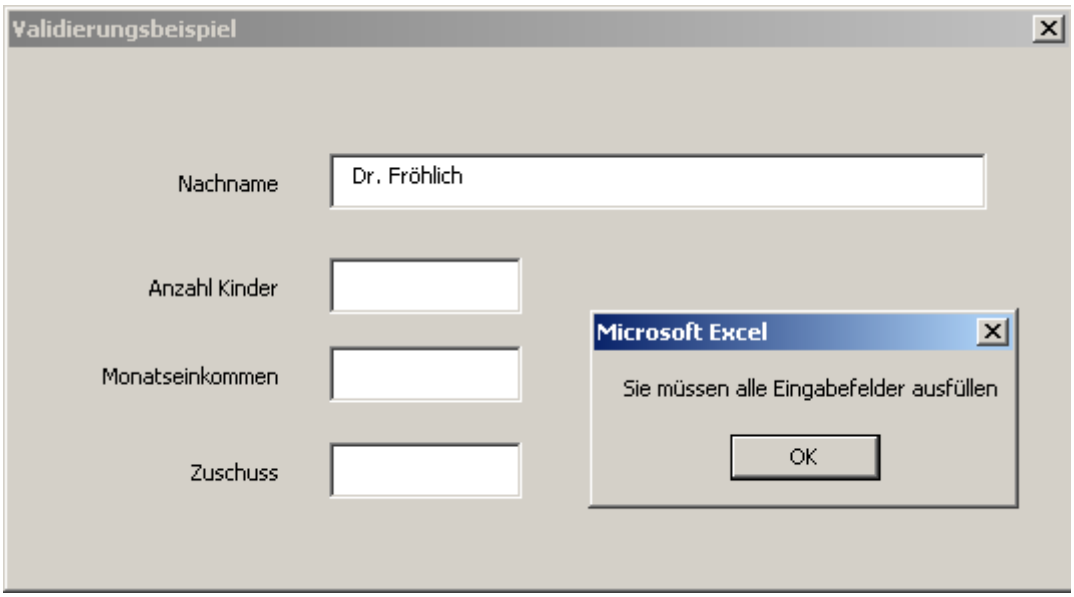
Das Programm besitzt eine **zweistufige Schichtenarchitektur** (nächstes Bild). Die obere Schicht bildet das Formular **Validierungsbeispiel** mit seinem Modul. Dieses Modul führt den Benutzer durch das Programm und steuert auch die Validierung. Die darunter liegende Schicht ist zweigeteilt. Im Modul **Zuschussberechnung** (rechts) liegt der inhaltliche Kern der Verarbeitung (sog. Programmlogik). Der Rest dieser Schicht besteht aus zwei Moduln, die bei der Validierung herangezogen werden. Diese beiden Moduln bilden selbst wieder eine kleine Hierarchie. **ValForm** enthält Funktionen, die bei der Validierung von Formular-Eingabefeldern herangezogen werden können; **ValHlp** enthält Funktionen, die rein wertbezogene Überprüfungen vornehmen.



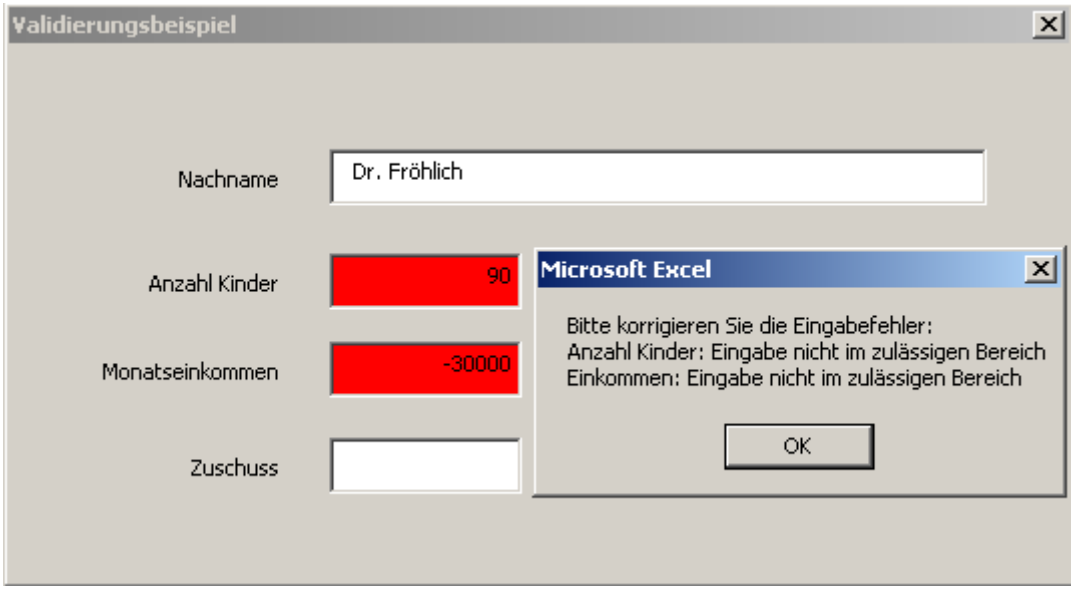
Wir wollen nun das **Verhalten des Programms bei Eingabefehlern** studieren. Das nächste Bild zeigt die Reaktion des Programms, wenn der Benutzer einen unzulässigen Wert in das Feld Monatseinkommen eingibt. Hier greift die **feldbezogene Validierung** ein: Das Feld wird rot gefärbt, der bereits vorher vorhandene Tooltiptext wird um einen Fehlerhinweis ergänzt. Der falsche Eingabewert bleibt zunächst erhalten und muss auch vom Benutzer nicht sofort korrigiert werden. Er kann seine Eingaben an beliebiger Stelle in der Maske fortsetzen.



Hat der Benutzer die Schaltfläche <Zuschuss errechnen> gedrückt, so greift die **Validierung en bloc**. Sie überprüft zunächst, ob alle Mussfelder ausgefüllt sind. Ist dies nicht der Fall, so erfolgt eine entsprechende Meldung (nächstes Bild).



Da die feldbezogene Validierung die Korrektur einzelner fehlerhafter Felder nicht erzwingt, prüft die Validierung en bloc auch nochmals die einzelnen Eingaben. Finden sich noch unzulässige Eingaben, so wird eine Korrekturaufforderung mit einer ausführlichen Fehlerliste ausgegeben (nächstes Bild). Nach dem Bestätigen der Korrekturaufforderung liegt der Fokus der Eingabe auf dem ersten fehlerhaften Feld. Die Berechnung des Zuschusses findet erst statt, wenn die Validierung en bloc keine Fehler mehr feststellt.



Wir betrachten nun das **Formularmodul Validierungsbeispiel**. Die drei Ereignisprozeduren, welche die **feldbezogene Validierung** anstoßen, befinden sich am Schluss. Jede der drei Prozeduren ist nur eine Zeile lang und besteht aus dem Aufruf einer auf den betreffenden Datentyp zugeschnittenen Validierungsfunktion im Modul ValForm. Beachten Sie, dass diesen Funktionen nicht nur der zu überprüfende Wert übergeben wird, sondern das gesamte Steuerelement, in das der Wert eingegeben wurde. Auf diese Weise können die Funktionen in ValForm voll über diese Steuerelemente verfügen und damit auch deren Hintergrundfarbe ändern und den Tooltiptext um die Fehlerhinweise ergänzen. Beachten Sie auch, dass diese Funktionen hier wie Prozeduren behandelt werden. Ihr Rückgabewert wird nicht aufgefangen, weil ihre relevante Aktion aus den Änderungen beim jeweiligen Steuerelement besteht.

Die **Validierung en bloc** geschieht innerhalb der Ereignisprozedur ErrechnenBtn_Click; sie nimmt den größten Teil dieser Prozedur ein. Zunächst wird die Vollständigkeit der Eingaben überprüft. Dann werden die einzelnen Eingabefelder validiert, wobei dieselben Funktionen des Moduls ValForm aufgerufen werden wie bei der feldbezogenen Validierung. Im Gegensatz zur feldbezogenen Validierung werden aber die Rückmeldungen der Funktionen aufgefangen und verwertet. Diese Rückmeldungen beschränken sich nicht auf die Funktionswerte (Eingabe korrekt oder nicht), sondern erstrecken sich auch auf die Parameter, die ByRef übergeben wurden. Dabei handelt es sich um die bereinigten Eingabewerte und, im Fehlerfall, einen Fehlerhinweis, der dann in die Liste der Fehlerhinweise aufgenommen wird.

```
'leert die Eingabefelder
'======================
Private Sub LeerenBtn_Click()
   Me.NameTBx.Text = ""
   Me.KinderTBx.Text = ""
   Me.EinkommenTBx.Text = ""
   Me.NameTBx.BackColor = vbWindowBackground
   Me.KinderTBx.BackColor = vbWindowBackground
   Me.EinkommenTBx.BackColor = vbWindowBackground
   Me.ZuschussTBx.Text = ""
   Me.NameTBx.SetFocus
```

```
End Sub

'Aktionen nach Drücken von <Zuschuss errechnen>
'=============================================
Private Sub ErrechnenBtn_Click()

  'Prüfung, ob obligatorische Felder ausgefüllt
  If Me.NameTBx.Text = "" Or Me.KinderTBx.Text = "" Or _
    Me.EinkommenTBx.Text = "" Then
    MsgBox "Sie müssen alle Eingabefelder ausfüllen"
    Exit Sub
  End If

  'Hilfsvariablen zur Fehlerbehandlung
  Dim Err As String
  Dim allErrors As String
  allErrors = ""
  Dim focusSet As Boolean

  'für Endform der Eingaben
  Dim Kinder As Long
  Dim Einkommen As Double
  Dim Name As String

  'Prüfung aller Eingabefelder

  If Not ValForm.checkName(Me.NameTBx, Name, Err) Then
    Me.NameTBx.SetFocus
    focusSet = True
    allErrors = allErrors & "Nachname: " & Err & vbLf
  End If

  If Not ValForm.checkLng(Me.KinderTBx, 0, 20, Kinder, Err) Then
    allErrors = allErrors & "Anzahl Kinder: " & Err & vbLf
    If Not focusSet Then
      Me.KinderTBx.SetFocus
      focusSet = True
    End If
  End If

  If Not ValForm.checkDbl(Me.EinkommenTBx, 0, 100000, Einkommen, Err) Then
    allErrors = allErrors & "Einkommen: " & Err & vbLf
    If Not focusSet Then
      Me.EinkommenTBx.SetFocus
      focusSet = True
    End If
  End If

  'Verarbeitung nach Prüfung
  If allErrors <> "" Then        'wenn Eingaben fehlerhaft
```

```
    MsgBox "Bitte korrigieren Sie die Eingabefehler:" & _
        vbLf & allErrors, vbOKOnly
  Else                    'wenn fehlerlos
    Dim zuschuss As Double
    zuschuss = Zuschussberechnung.zuschuss(Einkommen, Kinder)
    Me.EinkommenTBx.Text = FormatCurrency(Einkommen)
    Me.ZuschussTBx.Text = FormatCurrency(zuschuss)
  End If

End Sub

'Prüfung der Eingabefelder unmittelbar nach Update
'================================================

Private Sub KinderTBx_AfterUpdate()
  ValForm.checkLng con:=Me.KinderTBx, min:=0, max:=20
End Sub

Private Sub EinkommenTBx_AfterUpdate()
  ValForm.checkDbl con:=Me.EinkommenTBx, min:=0, max:=100000
End Sub

Private Sub NameTBx_AfterUpdate()
  ValForm.checkName con:=Me.NameTBx
End Sub
```

Das Modul **ValForm** enthält Validierungsfunktionen, deren Anwendung nicht auf dieses Programm beschränkt ist. Die Funktionen können in jedem Programm verwendet werden, das Validierungen von Formulareingaben vornehmen muss.

Die drei Funktionen des Moduls sind hinsichtlich der zu überprüfenden Datentypen spezialisiert. Die Funktion checkDbl prüft Double-Werte, die Funktion checkLng prüft ganzzahlige Werte und die Funktion checkName ist für die Prüfung von Strings anwendbar, welche Namen repräsentieren.

Eine Besonderheit der drei Funktionen sind ihre **Nebeneffekte**. Sie nehmen einige Parameter ByRef entgegen und verändern sie im Lauf der Validierung.

```
'prüft, ob der in con enthaltene Wert einer Doublezahl im Bereich [min, max] entspricht;
'=================================================================
Public Function checkDbl(ByRef con As MSForms.Control, _
                         ByVal min As Double, _
                         ByVal max As Double, _
                        Optional ByRef num As Double, _
                        Optional ByRef errorstr As String) _
            As Boolean

  Const ErrMsg As String = ". Fehler: "
  Dim ErrPos As Long
```

```
  ErrPos = InStr(1, con.ControlTipText, ErrMsg)
  If ErrPos > 0 Then con.ControlTipText = Left(con.ControlTipText, ErrPos - 1)

  If con.Text = "" Then
     num = 0
     errorstr = ""
     checkDbl = True
     con.BackColor = vbWindowBackground
  Else
     If ValHlp.istImDblBereich(con.Text, min, max) Then
        num = CDbl(con.Text)
        errorstr = ""
        checkDbl = True
        con.BackColor = vbWindowBackground
     Else
        checkDbl = False
        num = 0
        errorstr = "Eingabe nicht im zulässigen Bereich"
        con.BackColor = vbRed
        con.ControlTipText = con.ControlTipText & ErrMsg & errorstr
     End If
   End If
End Function


'prüft, ob der in con enthaltene Wert einer Long-Zahl im Bereich [min, max] entspricht;
'=====================================================================
Public Function checkLng(ByRef con As MSForms.Control, _
                         ByVal min As Long, _
                         ByVal max As Long, _
                         Optional ByRef num As Long, _
                         Optional ByRef errorstr As String) _
                  As Boolean

  Const ErrMsg As String = ". Fehler: "
  Dim ErrPos As Long
  ErrPos = InStr(1, con.ControlTipText, ErrMsg)
  If ErrPos > 0 Then con.ControlTipText = Left(con.ControlTipText, ErrPos - 1)

  If con.Text = "" Then
     num = 0
     errorstr = ""
     checkLng = True
     con.BackColor = vbWindowBackground
  Else
     If ValHlp.istImGanzzBereich(con.Text, min, max) Then
        num = CLng(con.Text)
        errorstr = ""
```

```
        checkLng = True
        con.BackColor = vbWindowBackground
      Else
        checkLng = False
        num = 0
        errorstr = "Eingabe nicht im zulässigen Bereich"
        con.BackColor = vbRed
        con.ControlTipText = con.ControlTipText & ErrMsg & errorstr
      End If
    End If

End Function


'prüft, ob der in con enthaltene Wert einem Namen entspricht
'================================================
Public Function checkName(ByRef con As MSForms.Control, _
                          Optional ByRef Name As String, _
                          Optional ByRef errorstr As String) _
                  As Boolean

  Const ErrMsg As String = ". Fehler: "
  Dim ErrPos As Long
  ErrPos = InStr(1, con.ControlTipText, ErrMsg)
  If ErrPos > 0 Then con.ControlTipText = Left(con.ControlTipText, ErrPos - 1)

  con.Text = Trim(con.Text)
  If con.Text = "" Then
     errorstr = ""
     checkName = True
     con.BackColor = vbWindowBackground
  Else
     If ValHlp.istName(con.Text) Then
        errorstr = ""
        checkName = True
        con.BackColor = vbWindowBackground
     Else
        checkName = False
        errorstr = "Eingabe nicht im zulässigen Bereich"
        con.BackColor = vbRed
        con.ControlTipText = con.ControlTipText & ErrMsg & errorstr
     End If
  End If

End Function
```

Die Überprüfung der Eingabewerte führen die Funktionen von ValForm nicht selbst durch, sondern delegieren sie an Funktionen des Moduls **ValHlp**. Dieses Modul stellt Funktionen zur Werteprüfung bereit, die nicht an Formularsteuerelemente gebunden sind.

```
'prüft, ob der Wert z einer Doublezahl im Bereich [min, max] entspricht
'=======================================================
Public Function istImDblBereich(ByVal z As Variant, _
                                ByVal min As Double, _
                                ByVal max As Double)
                    As Boolean
  istImDblBereich = False
  If IsNumeric(z) Then
    If CDbl(z) >= min And CDbl(z) <= max Then
      istImDblBereich = True
    End If
  End If
End Function


'ermittelt für den Wert z, ob er einer Ganzzahl entspricht; z kann auch ein String sein
'=================================================================
Public Function istGanzzahl(ByVal z As Variant) As Boolean
  If Not IsNumeric(z) Then
    istGanzzahl = False
  Else
    If CLng(z) - Int(z) = 0 Then
      istGanzzahl = True
    Else
      istGanzzahl = False
    End If
  End If
End Function


'ermittelt für den Wert z, ob er einer Ganzzahl entspricht und 'im Bereich [min, max]
´liegt; z kann auch ein String sein
'=================================================================
Public Function istImGanzzBereich(ByVal z As Variant, _
                                  ByVal min As Long, _
                                  ByVal max As Long)
                    As Boolean
  If Not istGanzzahl(z) Then
    istImGanzzBereich = False
  Else
    If CLng(z) >= min And CLng(z) <= max Then
      istImGanzzBereich = True
    Else
```

```
        istImGanzzBereich = False
      End If
    End If
End Function


'ermittelt für den Wert s, ob er den Anforderungen für einen 'Namen genügt;
,Achtung: fängt nicht alle Fehler ab!
'=============================================================
Public Function istName(ByVal s As String) As Boolean
  istName = True
  If Not istBuchstabe(Mid(s, 1, 1)) Then
    istName = False
  Else
    Dim i As Integer
    For i = 1 To Len(s)
      If Not (istBuchstabe(Mid(s, i, 1)) Or Mid(s, i, 1) = " " _
          Or Mid(s, i, 1) = "-" Or Mid(s, i, 1) = "." Or _
          Mid(s, i, 1) = "'") Then
        istName = False
      End If
    Next
  End If
End Function


'prüft, ob das übergebene Zeichen ein Buchstabe ist; bezieht
'dabei auch Umlaute mit ein
'================================================
Public Function istBuchstabe(ByVal c As String) As Boolean
  c = Mid(c, 1, 1)
  If c >= "A" And c <= "Z" Or _
    c >= "a" And c <= "z" Or _
    c = "Ä" Or c = "ä" Or c = "Ü" Or _
    c = "ü" Or c = "Ö" Or c = "ö" Then
    istBuchstabe = True
  Else
    istBuchstabe = False
  End If
End Function
```

Sie ist zwar im Hinblick auf die Validierung nicht relevant, aber zum Programm gehört sie trotzdem: die Berechnung des Zuschusses. Deshalb der Vollständigkeit halber noch das Modul **Zuschuss-berechnung**:

```
'ermittelt den Kinderzuschuss aus Einkommen und Kinderzahl
```

```
'=============================================
Public Function zuschuss(ByVal eink As Double, ByVal kids As Byte) As Double
  Dim anrechEink As Double
  anrechEink = IIf(eink - 450 > 0, eink - 450, 0)
  zuschuss = kids * 180 - anrechEink
  zuschuss = IIf(zuschuss > 0, zuschuss, 0)
End Function
```

Abschließende Bemerkungen zur Ex-post-Validierung

- Für die Ausarbeitung dieses Beispiels habe ich Anregungen aus dem ausgezeichneten Buch von Bovey, Wallentin, Bullen und Green aufgenommen (s. Literaturliste zu Excel-VBA).
- Die in den Modulen ValForm und ValHlp enthaltenen Funktionen decken nicht alle Eingabetypen ab; sie müssen bei Bedarf durch weitere Funktionen ergänzt werden.
- Manche Datentypen lassen sich mit den gezeigten primitiven Mitteln nur schwer überprüfen. Die im obigen Beispiel verwendete Funktion checkName versagt leider bei vielen Fehleingaben. So bleibt z.B. unbemerkt, wenn der Benutzer in einem Doppelnamen zu viele Bindestriche einträgt („Streit- - -Hammel“ statt „Streit-Hammel“). Um hier alle möglichen Fehler auszuschließen, muss man schwerere Geschütze auffahren, wie z.B. eine Prüfung anhand regulärer Ausdrücke.
- In manchen Fällen, in denen Ex-post-Validierung schwierig ist, hilft **Ex-ante-Validierung**. Dazu gehören insbesondere Datumseingaben.
- Validierung kann die Eingabe **zulässiger** Werte erzwingen, nicht aber die Eingabe **richtiger** Werte.

3.7.3 Ex-ante-Validierung

Zur Ex-ante-Validierung rechnen wir alle Maßnahmen, die eine zulässige Eingabe fördern. Während die Ex-post-Validierung aus der Sicht des Benutzers erst einsetzt, nachdem er einen Wert eingegeben hat, fördert oder erzwingt die Ex-ante-Validierung von vornherein die Eingabe zulässiger Werte.

Zu den Maßnahmen, die eine zulässige Eingabe **fördern, aber nicht erzwingen**, gehören (s. auch die Prinzipien der Formulargestaltung):

- präzise Feldbenennungen
- inhaltliche Gruppierung von Eingabefeldern
- Mussfelder visuell von Kannfeldern zu unterscheiden (z.B. durch Sternchen)
- Vorbesetzung von Mussfeldern mit Standardwerten
- Zusatzinformation, die bei Bedarf abgerufen werden kann. Im einfachsten Fall sind dies die Tooltips des Eingabefelds, bei komplizierteren Fällen kann man spezielle Icons neben den Eingabefeldern anbringen, die zu Hilfetexten führen.

Es gibt aber auch Maßnahmen, die eine zulässige Eingabe **erzwingen** können. Sie ersparen eine Ex-post-Validierung und können immer dann eingesetzt werden, wenn diese schwierig ist. Zu diesen Maßnahmen gehören:

- Wahl geeigneter Steuerelemente, welche die Eingabemöglichkeiten auf die zulässigen reduzieren. Die bekanntesten dieser Steuerelemente sind ComboBox, Option Button (Radio Button), CheckBox und ListBox. Manche Eingaben kann man „entschärfen“, indem man sie aufsplittet, z. B. Datumsangaben. Man kann sie durch drei ComboBox-Eingaben für Tag, Monat und Jahr ersetzen.
- Erzwingen einer Eingabefolge, wenn Eingaben von vorhergehenden Eingaben abhängig sind. Hierfür gibt es mehrere Möglichkeiten. Man kann die Eingaben z.B. auf mehrere Fenster aufteilen, die nacheinander präsentiert werden. Wenn es sich insgesamt nur um wenige Felder handelt, kann man auch die gerade gefragten Felder gezielt aktivieren und die anderen deaktivieren.