Kapitel 3

Objekte und Visual Basic

Ich erinnere mich an einen Kurs in meinem zweiten Studienjahr am College, der ICS2 hieß (Information and Computer Science #2 – tatsächlich). Der Kurs ging über mehrere Wochen, in denen ich damit zu kämpfen hatte, das Konzept von »Klassen« in einer Sprache zu verstehen, die *Simula* hieß. Wenn Sie es unbedingt wissen wollen: Simula ist eine der Sprachen, die College-Professoren zu lieben scheinen, aber nie verwenden – kein Mensch weiß, warum. Stellen Sie sich diese Sprache als eine Art »besseres« Pascal vor (aus der Zeit, bevor Pascal in so viele verschiedene Dialekte zerfiel, wie es Programmierer gibt).

Bis dahin hatte ich in Basic programmiert – auch Basic war in viele Dialekte zerfallen, was aber niemanden zu stören schien. Der Professor hat das Konzept der Klassen wohl viele Male erklärt, aber ich hab's einfach nicht begriffen. Wie kann bloß eine Datenstruktur Funktionen enthalten? Es schien einfach keinen Sinn zu ergeben.

Doch eines Tages war ich wie vom Blitz getroffen – ich hatte es verstanden. Ich verstand nicht nur, worum es bei Klassen ging, sondern begriff auch, daß sie ein phänomenales Werkzeug zum Erstellen von Anwendungen sind. Ich hatte die objektorientierte Programmierung entdeckt und seitdem immer angewendet, sobald eine Sprache das zuließ.

Wenn sich Leute über die Wichtigkeit von in Visual Basic 4.0 eingeführten Features unterhalten, reden sie oft über die 32-Bit-Unterstützung, die Unterstützung von OLE-Automation (Entschuldigung, ActiveX natürlich), DLLs, Performance-Verbesserungen und dergleichen mehr. Für mich war das einzig wichtige, herausragende Feature in Visual Basic 4.0 die Einführung der Klassen-Module. Nicht etwa deswegen, weil diese Server oder OLE-Automation ermöglichten, sondern weil ich wieder meine Klassen hatte. Ich konnte nun endlich objektorientierte Techniken in meiner Lieblingssprache zur Anwendung bringen.

Nun, wenn Sie sich mit dem Programmieren mit Klassen auskennen, wenn Sie deren Bedeutung zu schätzen wissen, und wenn Ihr erster Schritt beim Entwurf einer jeden Visual-Basic-Anwendung darin besteht, mit welchen Klassen Sie die Daten und die Funktionalität Ihrer Anwendung abstrahieren können, dann können Sie dieses Kapitel überfliegen oder sogar überspringen. Falls Sie allerdings in der Regel Anwendungen ohne die Verwendung von Klassen erstellen, falls Sie denken sollten, daß Klassen nur für programmierbare Objekte (EXE- oder DLL-Server) von Belang seien, oder wenn Sie sich einfach nicht vorstellen können, warum ich so von Klassen-Modulen begeistert bin, daß ich sie als den einzig wahren Fortschritt von VB3 zu VB4 betrachte, dann ist dieses Kapitel genau richtig für Sie – erst recht, wenn für Sie Visual-Basic-Programmierung etwas vollkommen Neues ist.

3.1 Die Theorie der objektorientierten Programmierung

Ein Objekt im Sinne der objektorientierten Programmierung zeichnet sich durch folgende drei Merkmale aus:

- Kapselung
- Polymorphie
- Vererbung

Stopp! Halten wir einmal einen Moment inne.

Objektorientierte Programmierung ist einer der Begriffe in der Computer-Wissenschaft, der mit ein paar unglücklichen Begleiterscheinungen verbunden ist:

- Er wirkt einschüchternd – Programmieranfänger sehen es oft als eine fortgeschrittene Technik an, mit der nur Experten etwas anfangen könnten.
- Er klingt nach Fortgeschrittenheit – nach etwas, das ein Anfänger oder ein Programmierer mittlerer Erfahrung später noch lernen kann, wenn er erst einmal mit der Sprache vertraut genug ist.

Irrtum – in beiden Fällen.

Es wundert niemanden, daß der Begriff diese Reaktionen hervorruft, wenn mit Begriffen wie Kapselung, Polymorphie und Vererbung hantiert wird.

Sie können sich vorstellen, daß ich die »wahre« Definition von objektorientierter Programmierung, die Notwendigkeit von Polymorphie, die Vor- und Nachteile von Vererbung oder Aggregation dutzende Male gelesen habe. Aber irgendwie kann ich mich nie an all die akademischen Argumente erinnern. Da Visual Basic aber auch keine »echte« objektorientierte Sprache ist (im Sinne der akademischen Definition), tut das wohl kaum etwas zur Sache, denke ich.

Tatsächlich scheint mir, je mehr ich darüber nachdenke, das Studium der Theorie der objektorientierten Programmierung der ungeeigneteste Weg zu sein, um sie zu lernen. Vielleicht finde ich ja noch einen anderen Zugang und werde dann später auf die Theorie zurückkommen, falls nötig.

Vergessen wir also den Rest, der in diesen Abschnitt hinein gehört hätte.

3.2 Objektfreies Programmieren

Ich habe mir hier ein ehrgeiziges Ziel gesteckt. Ich möchte Ihnen nicht einfach bloß die objektorientierte Programmierung erklären. Ich möchte Sie vielmehr zu diesem Programmier-Stil bekehren. Ich bin mir sicher, daß sich das nicht nur auf erfolgreichere Ansätze bei der ActiveX-Programmierung auswirken wird, sondern auf Ihre gesamte Programmierarbeit.

Theorie kann hochinteressant sein. Aber ich glaube, sie wäre hier fehl am Platze. Steigen wir statt dessen einfach mal in ein Beispiel ein, das typisch für die Art und Weise ist, in der viele Leute programmieren. Und versuchen wir dabei herauszufinden, wie und an welcher Stelle objektorientierte Programmierung einen Vorteil versprechen könnte.

Mit anderen Worten: lassen Sie uns ein Programm schreiben.

Eine simple Anwendung sollte reichen. Nehmen wir eine einfache Registrierkassen-Anwendung, wie Sie sie vielleicht für Fast-Food-Restaurants programmieren würden. In Abbildung 3.1 sehen Sie das Hauptformular des Projekts `Retail1`. Es enthält vier Schaltflächen, für jedes Gericht eine eigene. Das Betätigen einer der Schaltflächen fügt das entsprechende Gericht zur Rechnung hinzu. Die Clear-Schaltfläche löscht die aktuelle Bestellung. Geben Sie einen gezahlten Betrag ein, zeigt die Change-Anzeige das herauszugebende Wechselgeld an. Mit einem Doppelklick auf einen Eintrag in der Liste entfernen Sie genau diesen aus der Liste.

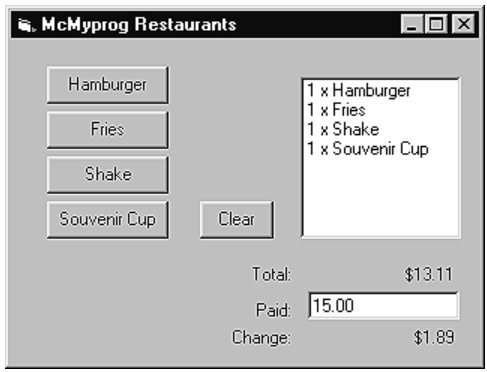

Abb. 3.1: Das Hauptformular der Anwendung Retail1

Es ist eine simple Anwendung, und sie ist ziemlich narrensicher, auch wenn sie wohl einige Prüf- und Sicherheitsmechanismen vermissen läßt, die bei einer kommerziellen Qualitätslösung notwendig wären. Geben Sie beispielsweise nur ein Dollar-Zeichen in das Feld für den Zahlbetrag ein, wird der Wert nicht korrekt ermittelt.

Dieses Beispiel entstand, indem zuerst die Benutzeroberfläche gestaltet, eine Reihe von Arrays zur Aufnahme von Preisen und Mengen angelegt und schließlich Code für die entsprechenden Ereignisse eingefügt wurde. Listing 3.1 zeigt den Code dieser Implementierung. Zwei null-basierte Arrays werden in dieser Anwendung verwendet. Null-basiert bedeutet einfach, daß unter dem Index 0 relevante Werte abgelegt werden. Das Array `Prices(3)` enthält demnach vier Einträge (0 bis 3), in die jeweils der Preis eines Gerichts eingelesen wird. Die vier Schaltflächen sind in einem ebenfalls null-basierten Control-Array zusammengefaßt. Die Beschriftungen der Schaltflächen zeigen die Bezeichnungen der Gerichte der Speisekarte an. Das Array `Items` enthält Zähler für jedes Gericht, der die Anzahl der Bestellungen nachhält.

```
' Retail example #1
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.
```

```
Option Explicit

' Null-basiertes Array von Menüpunkten
Dim prices(3) As Currency  ' Preise der Menüpunkte
' Null-basierte Bestellmengen
Dim items(3) As Integer

Private Sub Form_Load()
   prices(0) = 1.99
   prices(1) = 0.89
   prices(2) = 1.29
   prices(3) = 8.94  ' Sonderangebot...
   UpdateAll
End Sub

' Einen Posten einer Bestellung hinzufügen
Private Sub cmdMenu_Click(Index As Integer)
   items(Index) = items(Index) + 1
   UpdateAll
End Sub

' Bestellung löschen
Private Sub cmdClear_Click()
   Dim itemnum%
   For itemnum = 0 To 3
      items(itemnum) = 0
   Next itemnum
   UpdateAll
End Sub

' Inhalt der ListBox aktualisieren
Public Sub UpdateListBox()
   Dim itemnum%
   lstItems.Clear
   For itemnum = 0 To 3
      If items(itemnum) > 0 Then
         lstItems.AddItem items(itemnum) & " x " _
          & cmdMenu(itemnum).Caption
         lstItems.ItemData(lstItems.NewIndex) = itemnum
      End If
   Next itemnum
End Sub

' Bestell-Posten per Doppelklick löschen
Private Sub lstItems_DblClick()
   Dim thisitem%
```

```
    thisitem = lstItems.ItemData(lstItems.ListIndex)
    items(thisitem) = items(thisitem) - 1
    UpdateAll
End Sub

' Gesamtbetrag berechnen
Public Function CalculateTotal() As Currency
    Dim itemnum%
    Dim total As Currency

    For itemnum = 0 To 3
        total = total + items(itemnum) * prices(itemnum)
    Next itemnum
    CalculateTotal = total
End Function

' Summe und Wechselgeld anzeigen
Public Sub UpdateTotals()
    Dim total As Currency
    Dim paid As Currency
    total = CalculateTotal()
    lblTotal.Caption = Format(total, "Currency")
    paid = Val(txtPaid.Text)
    If paid - total > 0 Then
        lblChange.Caption = Format(paid - total, _
         "Currency")
    Else
        lblChange.Caption = "Please pay"
    End If
End Sub

Private Sub txtPaid_Change()
    UpdateTotals
End Sub

Public Sub UpdateAll()
    UpdateListBox
    UpdateTotals
End Sub
```

Listing 3.1: Listing des Formulars frmRetail1.frm

Woran mangelt es bei diesem Code? Der größte offensichtliche Fehler ist, daß das Menü fest codiert (»hart-codiert«) ist. Das bedeutet, daß die Menü-Einträge nicht dynamisch geändert werden können und jede Änderung eine Änderung im Code der Anwendung mit anschließender erneuter Kompilierung erfordern würde. Dieser Weg mag für einen Quick-and-Dirty-Schnellschuß brauchbar sein, ist aber ansonsten ein fürchterlicher Entwurf.

Zur Benutzeroberfläche gibt es nicht viel zu sagen. Sie besticht durch ihre Schlichtheit. Es dürfte nur wenige Minuten dauern, jemanden in die Bedienung des Programms einzuweisen.

Und sie hätte schlechter ausfallen können – viel schlechter.

Obwohl der Code schnell zusammengestoppelt wurde, enthält er dennoch ein paar ganz gute Ideen. Die Preise befinden sich in einem Array. Dieses ist zwar hart-codiert, hält aber die Möglichkeit für Änderungen zur Laufzeit vielleicht in einer späteren Version offen. Die Schaltflächen sind ebenfalls als Array angelegt. Könnten so vielleicht Menüänderungen zur Laufzeit möglich werden? Wie steht es mit Sonderangeboten? Vielleicht möchten Sie ein separates Stammkunden-Menü einführen – die Preise in einem Array ermöglichen das.

Der Code ist teilweise modular aufgebaut. Die Aktualisierung der ListBox ist zentral zusammengefaßt, ebenso wie die Berechnung der Summen. Das bedeutet, wenn an irgendeiner Stelle im Code die aktuelle Reihenfolge geändert wird, genügt der Aufruf der Funktion UpdateAll – man braucht sich keine direkten Gedanken um den Inhalt der ListBox oder der Summenfelder zu machen.

Implementieren wir doch ein paar der soeben angestellten Überlegungen zu Verbesserungen, wie sie in Listing 3.2 zu sehen sind.

```
' Retail example #2
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.

Option Explicit

' Null-basiertes Array für Menüpunkte
Dim prices() As Currency  ' Preise der Menüpunkte
' Null-basierte Bestellmengen
Dim items() As Integer
' ID des nächsten Menüpunkts
Dim NextItem As Integer

' Bestellung löschen
Private Sub cmdClear_Click()
   Dim itemnum%
   For itemnum = 0 To UBound(items)
      items(itemnum) = 0
   Next itemnum
   UpdateAll
End Sub

' Posten der Bestellung hinzufügen
Private Sub cmdMenu_Click(Index As Integer)
   items(Index) = items(Index) + 1
```

```
   UpdateAll
End Sub

Private Sub Form_Load()
   mnuStandard_Click
End Sub

' Inhalt der ListBox aktualisieren
Public Sub UpdateListBox()
   Dim itemnum%
   lstItems.Clear
   For itemnum = 0 To UBound(items)
      If items(itemnum) > 0 Then
         lstItems.AddItem items(itemnum) & " x " _
          & cmdMenu(itemnum).Caption
         lstItems.ItemData(lstItems.NewIndex) = itemnum
      End If
   Next itemnum
End Sub

' Gesamtbetrag berechnen
Public Function CalculateTotal() As Currency
   Dim itemnum%
   Dim total As Currency

   For itemnum = 0 To UBound(items)
      total = total + items(itemnum) * prices(itemnum)
   Next itemnum
   CalculateTotal = total
End Function

' Summe und Wechselgeld anzeigen
Public Sub UpdateTotals()
   Dim total As Currency
   Dim paid As Currency
   total = CalculateTotal()
   lblTotal.Caption = Format(total, "Currency")
   paid = Val(txtPaid.Text)
   If paid - total > 0 Then
      lblChange.Caption = Format(paid - total, _
       "Currency")
   Else
      lblChange.Caption = "Please pay"
   End If
End Sub

' Bestell-Posten per Doppelklick löschen
Private Sub lstItems_DblClick()
```

```
   Dim thisitem%
   thisitem = lstItems.ItemData(lstItems.ListIndex)
   items(thisitem) = items(thisitem) - 1
   UpdateAll
End Sub

' Standard-Menü setzen
Private Sub mnuStandard_Click()
   ClearMenuItems
   AddMenuItem "Hamburger", 1.99
   AddMenuItem "Curly Fries", 1.32
   AddMenuItem "Side Salad", 2.39
   AddMenuItem "Chicken", 2.32
   AddMenuItem "Stuff Delux", 5.39
   UpdateAll
End Sub

' Stammkunden-Menü
Private Sub mnuSenior_Click()
   ClearMenuItems
   AddMenuItem "Hamburger", 1.79
   AddMenuItem "Fries", 1.12
   AddMenuItem "Chicken Soup", 2.39
   AddMenuItem "Salad", 2.12
   AddMenuItem "LoCal Delux", 5.19
   UpdateAll
End Sub

' Programmierer-Menü
Private Sub mnuProgrammer_Click()
   ClearMenuItems
   AddMenuItem "Hamburger", 2.19
   AddMenuItem "Pizza", 2.52
   AddMenuItem "Cheetos", 0.89
   AddMenuItem "Jolt", 1.19
   AddMenuItem "More Jolt", 2.19
   UpdateAll
End Sub

Private Sub txtPaid_Change()
   UpdateTotals
End Sub

Public Sub UpdateAll()
   UpdateListBox
   UpdateTotals
End Sub
```

```
' Neuen Menüpunkt anlegen
Public Sub AddMenuItem(ByVal itemname$, _
 ByVal itemprice As Currency)
   Dim currenttop%
   If NextItem > 0 Then
      ' bloß nicht Schaltfläche mit Index 0 anlegen
      Load cmdMenu(NextItem)
   End If
   ReDim Preserve items(NextItem)
   ReDim Preserve prices(NextItem)
   prices(NextItem) = itemprice
   items(NextItem) = 0
   cmdMenu(NextItem).Caption = itemname
   cmdMenu(NextItem).Top = cmdMenu(0).Top + _
    NextItem * 450
   cmdMenu(NextItem).Visible = True

   NextItem = NextItem + 1
End Sub

' Alle Menüpunkte löschen
Public Sub ClearMenuItems()
   Dim itemnum%
   Dim currenttop%
   If NextItem = 0 Then Exit Sub ' Bereits gelöscht
   currenttop = UBound(items)
   For itemnum = 1 To currenttop
      Unload cmdMenu(itemnum)
   Next itemnum
   ReDim items(0)
   ReDim prices(0)
   NextItem = 0
End Sub
```

Listing 3.2: Listing des Formulars frmRetail2.frm der Anwendung Retail2.vbp

Einige der Änderungen sind offensichtlich. Alle Schleifen hängen nun von den Obergrenzen der Arrays ab, so daß sie auch weiterhin einwandfrei funktionieren werden, falls sich die Zahl der Menüeinträge ändern sollte. Die Menüeinträge selbst werden dynamisch über die Funktionen AddMenuItem und ClearMenuItems angelegt bzw. gelöscht. Ein PopUp-Menü (ein Windows-Menü, keine Speisekarte) ist hinzugekommen, um mehrere Menüs zu unterstützen.

Vertiefen Sie sich ein paar Minuten in diese beiden Beispiele, und starten Sie sie von der CD. Sie sollten verstanden haben, wie sie funktionieren, wenn Sie sich mit dem Szenario im folgenden Abschnitt beschäftigen wollen.

3.2.1 Ist Ihnen das auch schon passiert?

So, nun haben Sie also Ihr simples Kassen-Programm abgeliefert, selbstverständlich pünktlich und zum vereinbarten Preis. Der Kunde (oder Ihr Chef) schaut es sich an und testet es mit realem Personal durch, und jedermann ist begeistert. Doch dann hören Sie die Worte, bei denen es jedem Programmierer kalt den Rükken hinunterläuft: »Ich hätte gerne ein paar kleine Änderungen.«

Es wird gesagt: »Sehen Sie, wir denken, daß das Ganze effizienter werden könnte, wenn mehrere Aufträge gleichzeitig festgehalten werden könnten. Dann kann eine weitere Bestellung aufgegeben werden, solange die vorhergehende noch bearbeitet wird. Sie brauchen ja bloß ein paar Knöpfe einzubauen, mit denen der Kassierer zwischen den Bestellungen wechseln kann. Aber keine Bange, es werden kaum mehr als drei oder vier Bestellungen gleichzeitig anfallen, und ich sehe, daß da auf der Fläche noch jede Menge Platz ist.«

Was brauchen Sie, um die gewünschten »kleinen« Änderungen umsetzen zu können?

Nun, Sie brauchen mehrere Bestellungen. Und da jede Bestellung in einem Array abgelegt ist, werden es mehrdimensionale Arrays sein müssen – wahrscheinlich zweidimensionale. Aber wie kriegen Sie dann die Redimensionierung einwandfrei hin? Bedenken Sie, daß die einzelnen Bestellungen unter Umständen mit verschiedenen Menüs verknüpft sind, so daß bei jeder Umschaltung der Bestellung auch das Menü umgeschaltet werden muß.

Sie werden sehr schnell merken, daß Sie in ganz großen Schwierigkeiten stecken, da Sie Ihr Programm entworfen haben, ohne an diese Möglichkeit zu denken. Sicher werden Sie das Problem lösen können, jedoch werden die Änderungen wohl jede einzelne Funktion in Ihrem Programm betreffen. Genauso gut könnten Sie das Programm auch von Grund auf neu schreiben.

3.3 Auf ein Neues – nun mit Objekten

Schauen wir uns das Problem noch einmal näher an, aber dieses Mal mit ein paar gründlichen Gedanken für einen vernünftigen Entwurf im Hinterstübchen.

Diesmal werden wir uns beim Entwurf die Fähigkeit von Visual Basic zunutze machen, Objekte anlegen zu können. Ich gehe in diesem Abschnitt davon aus, daß Sie schon soweit mit Visual Basic vertraut sind, daß Sie Ihrer Anwendung ein Klassen-Modul hinzufügen und darin Methoden und Eigenschaften einfügen können. Falls das aber noch völliges Neuland für Sie sein sollte, empfehle ich Ihnen, sich ein wenig mit der Einführung in Visual Basic der Online-Dokumentation zu beschäftigen.

Dort finden Sie im Kapitel »Programmieren mit Objekten« des zweiten Teils den Abschnitt »Erstellen eigener Klassen/Klassenmodule Schritt für Schritt«, der Sie durch das Anlegen eines Klassen-Moduls und das Einfügen einer Methode führt. Der Vorgang, Funktionen und Prozeduren in ein Klassen-Modul einzufügen, ist

identisch mit dem bei Formularen. Somit dürften Sie den Code der nun folgenden Beispiele verstehen können, auch wenn Sie noch nie zuvor mit Klassen-Modulen gearbeitet haben sollten.

Bei unserer Registrierkassen-Anwendung kann ein Kassierer Punkte aus einem festgelegten Menü auswählen, um eine Bestellung zusammenzustellen. Jeder Punkt hat einen Preis und einen Namen. Eine Bestellung kann einen oder mehrere Punkte aus dem Menü umfassen. Der Gesamtbetrag der Bestellung kann aus der Liste der Einzelposten ermittelt werden. Vom Grundkonzept her sehen wir drei Objekte: ein Menü, Menüpunkte und eine Bestellung.

Jedes Menü kann einen oder mehrere Menüpunkte haben, die jeweils einen Namen und einen zugeordneten Preis haben. Eine Bestellung kann einen oder mehrere Menüpunkte umfassen, von denen jeder einmal oder mehrfach auftreten kann.

Behalten Sie immer im Auge, daß hier von Menüs im Sinne von Speisekarten die Rede ist, und nicht von Visual-Basic-Menüs und -Menüobjekten. Ich denke zwar, daß das offensichtlich ist – aber ich gehe jede Wette darauf ein, daß ich eine Menge E-Mails bekommen würde, worin laufend nach dem Zusammenhang zwischen Hamburgern und Popup-Menüs gefragt werden würde, wenn ich das jetzt nicht noch einmal ausdrücklich klarstelle ...

Spaß beiseite – wenden wir uns wieder unserem ernsteren Thema zu. Ebenso wie bei den meisten Beispielanwendungen in diesem Buch gibt es nicht die eine, ideale Lösung für diese Anwendung. Es existieren fast immer mehrere Lösungswege, selbst bei einem objektorientierten Ansatz.

Bei unserer Beispielaufgabe hier lautet die entscheidende Frage für die Implementierung des Menü-Objekts, ob ein Menüpunkt ein Objekt oder ein Element in einem Array sein soll. In den beiden Beispielversionen Retail1 und Retail2 wurden die zu den Menüpunkten gehörenden Preise in einem Array und die Namen der Menüpunkte in den `Caption`-Eigenschaften der jeweiligen Schaltflächen gespeichert. Das ist keine besonders geschickte Lösung, um es milde auszudrükken. Ich sehe da zwei naheliegendere Lösungen:

1. Die Menüpunkte werden in einem gemeinsamen Array für Namen *und* Preise abgelegt. Dieser Ansatz ist effizient (das Redimensionieren eines Arrays geht schnell), doch wird der Umgang mit den einzelnen Menüpunkten außerhalb der Klasse schwieriger oder gar unmöglich.

2. Eine neue Klasse für Menüpunkte wird definiert. Dieser Ansatz erlaubt es, ein Menüpunktobjekt als Unterobjekt eines Menüs offenzulegen und es gegebenenfalls als Parameter an andere Prozeduren in der Anwendung zu übergeben.

Für dieses Beispiel belasse ich es dabei, Menüpunkte als internes Konstrukt der Klasse clsMenu1 anzulegen. Hier kommt es nicht so sehr darauf an, wie sie implementiert werden. Die Liste der Menüpunkte ist als Array eines benutzerdefinierten Datentyps angelegt, wodurch sich separate Arrays für die Punkte erübri-

gen. Die Verwendung von benutzerdefinierten Datentypen geht auch mit der objektorientierten Methodologie konform. Sie können sich nämlich einen benutzerdefinierten Datentyp als ein Einfachst-Objekt vorstellen, das nur über Eigenschaften verfügt. In Listing 3.3 sehen Sie die Implementierung eines Menüobjekts.

```
' Retail example #3
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.

Option Explicit
' Menüpunkt als benutzerdefinierter Datentyp
Private Type MenuItem
   ItemName As String
   ItemPrice As Currency
End Type

Private MenuList() As MenuItem   ' Array für Menüpunkte
Private NextItem As Integer ' Nächster zu ladender Punkt
' Methode zum Hinzufügen von Menüpunkten
Public Sub AddMenuItem(ByVal ItemName$, _
 ByVal ItemPrice As Currency)
   Dim currenttop%
   ReDim Preserve MenuList(NextItem)
   MenuList(NextItem).ItemPrice = ItemPrice
   MenuList(NextItem).ItemName = ItemName
   NextItem = NextItem + 1
End Sub

' Anzahl der Menüpunkte abfragen
Public Function ItemCount() As Integer
   ItemCount = NextItem
End Function

' Preis und Name eines Menüpunkts als Funktionen
Public Function ItemPrice(ByVal itemnum As Integer)
   ' Fehlerprüfung hier möglich:
   If itemnum >= NextItem Then Exit Function
   ItemPrice = MenuList(itemnum).ItemPrice
End Function

Public Function ItemName(ByVal itemnum As Integer)
   ' Fehlerprüfung hier möglich:
   If itemnum >= NextItem Then Exit Function
   ItemName = MenuList(itemnum).ItemName
End Function
```

Listing 3.3: Listing für clsMenu1, das Menü-Objekt

Das Objekt `clsMenu1` bietet die Methode `AddMenuItem` zum Einfügen eines einzelnen Menüpunkts an. Über die Funktionen `ItemName` und `ItemPrice` lesen Sie die Namen und Preise für die einzelnen Menüpunkte aus.

Eine Bestellung besteht aus einem Array mit Mengenangaben für jeden Punkt eines Menüs. Daher gehört zu jeder Bestellung ein assoziiertes Menü – in diesem Fall das gerade in der Anwendung in Gebrauch befindliche Menü. In den Beispielversionen Retail1 und Retail2 war es das `Items`-Array. Jedoch können Sie die Implementierung vom übrigen Programm isolieren, indem Sie sie in eine Klasse verpacken (siehe Listing 3.4).

```
' Retail example #3
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.

Option Explicit
Dim ItemList() As Integer
' Array enthält wenigstens ein Element
Private Sub Class_Initialize()
   ReDim ItemList(0)
End Sub

' Aktuelle Bestellung löschen
Public Sub Clear()
   Dim entry%
   For entry = 0 To UBound(ItemList)
      ItemList(entry) = 0
   Next entry
End Sub

' Bestellmenge eines einzelnen Menüpunkts
Public Property Get Quantity(ByVal itemnum%) As Integer
   ' Ungültiger Wert - lediglich Null zurückgeben
   If itemnum > UBound(ItemList) Then Exit Property
   Quantity = ItemList(itemnum)
End Property

Public Property Let Quantity(ByVal itemnum%, _
 ByVal iNewQuantity As Integer)
   If itemnum > UBound(ItemList) Then
      ReDim Preserve ItemList(itemnum)
   End If
   ItemList(itemnum) = iNewQuantity
End Property
```

Listing 3.4: Listing für clsOrder1, das Bestellung-Objekt

Die Klasse `clsOrder1` enthält eine Reihe von Methoden und Eigenschaften, die das interne Array für das übrige Programm offenlegen. Über die Eigenschaft `Quantity` wird die Menge zu einem einzelnen Menüpunkt gesetzt oder ausgelesen. Die Verwendung des Eigenschaften-Prozedurenpaares Get/Let erlaubt eine Fehlerprüfung und zugleich die Verwaltung des Arrays automatisch beim Zugriff auf die Eigenschaft.

Die `Clear`-Methode löscht eine Bestellung. Könnte man nicht auch eine Prozedur auf Modul- oder Formular-Ebene anlegen, die eine Bestellung einfach löscht, in der alle `Quantity`-Eigenschaften auf Null gesetzt würden? Natürlich ginge das. Aber hinter der objektorientierten Programmierung steht doch die Idee, daß ein bestimmter Satz an Daten mit Code verbunden wird. Wenn also eine Operation nur die Daten innerhalb eines Objekts betrifft, sollte diese Operation immer als Methode des Objekts angelegt werden.

Aus der Perspektive des Anwenders sehen die Beispielversionen Retail2 und Retail3 gleich aus. Sie arbeiten auf die gleiche Weise. Jedoch wurde die Version Retail3 unter Verwendung der Objekte clsMenu1 und clsOrder1 implementiert (siehe Listing 3.5).

```
' Retail example #3
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.

Option Explicit

Dim HighestButtonIndex As Integer
' Das aktuell zu verwendende Menü
Dim CurrentMenu As clsMenu1

' Die drei Standard-Menüs
Dim StandardMenu As New clsMenu1
Dim SeniorMenu As New clsMenu1
Dim ProgrammerMenu As New clsMenu1

' Die aktuelle Bestellung
Dim CurrentOrder As New clsOrder1

' Menüauswahlen hier laden.
' Hier geht das einfach so - in einer echten Anwendung
' würden diese vielleicht aus einer Datenbank geladen.
Private Sub Form_Load()
   With StandardMenu
      .AddMenuItem "Hamburger", 1.99
      .AddMenuItem "Curly Fries", 1.32
      .AddMenuItem "Side Salad", 2.39
      .AddMenuItem "Chicken", 2.32
      .AddMenuItem "Stuff Delux", 5.39
```

```
    End With
    With SeniorMenu
        .AddMenuItem "Hamburger", 1.79
        .AddMenuItem "Fries", 1.12
        .AddMenuItem "Chicken Soup", 2.39
        .AddMenuItem "Salad", 2.12
        .AddMenuItem "LoCal Delux", 5.19
    End With

    With ProgrammerMenu
        .AddMenuItem "Hamburger", 2.19
        .AddMenuItem "Pizza", 2.52
        .AddMenuItem "Cheetos", 0.89
        .AddMenuItem "Jolt", 1.19
        .AddMenuItem "More Jolt", 2.19
    End With

    ' StandardMenu zu Anfang (Vorgabe)
    Set CurrentMenu = StandardMenu
    ' Menü-Schaltflächen für aktuelles Menü laden
    LoadCommandButtons CurrentMenu
    UpdateAll
End Sub

' Bestellung löschen
Private Sub cmdClear_Click()
    CurrentOrder.Clear
    UpdateAll
End Sub

' Posten zur Bestellung hinzufügen
Private Sub cmdMenu_Click(Index As Integer)
    CurrentOrder.Quantity(Index) = _
     CurrentOrder.Quantity(Index) + 1
    UpdateAll
End Sub

' Inhalt der ListBox aktualisieren
Public Sub UpdateListBox()
    Dim itemnum%
    lstItems.Clear
    For itemnum = 0 To CurrentMenu.ItemCount
        If CurrentOrder.Quantity(itemnum) > 0 Then
            lstItems.AddItem _
             CurrentOrder.Quantity(itemnum) & " x " &_
             CurrentMenu.ItemName(itemnum)
            lstItems.ItemData(lstItems.NewIndex) = itemnum
```

```
      End If
   Next itemnum
End Sub

' Gesamtbetrag berechnen
Public Function CalculateTotal() As Currency
   Dim itemnum%
   Dim total As Currency

   For itemnum = 0 To CurrentMenu.ItemCount
      total = total + CurrentOrder.Quantity(itemnum) *_
      CurrentMenu.ItemPrice(itemnum)
   Next itemnum
   CalculateTotal = total
End Function

' Summe und Wechselgeld anzeigen
Public Sub UpdateTotals()
   Dim total As Currency
   Dim paid As Currency
   total = CalculateTotal()
   lblTotal.Caption = Format(total, "Currency")
   paid = Val(txtPaid.Text)
   If paid - total > 0 Then
      lblChange.Caption = Format(paid - total, _
       "Currency")
   Else
      lblChange.Caption = "Please pay"
   End If
End Sub

' Bestell-Posten per Doppelklick löschen
Private Sub lstItems_DblClick()
   Dim thisitem%
   thisitem = lstItems.ItemData(lstItems.ListIndex)
   With CurrentOrder
      .Quantity(thisitem) = .Quantity(thisitem) - 1
   End With
   UpdateAll
End Sub

' Gewünschtes Menü wählen
Private Sub mnuStandard_Click()
   Set CurrentMenu = StandardMenu
   MenuChanged
End Sub
```

```
Private Sub mnuSenior_Click()
   Set CurrentMenu = SeniorMenu
   MenuChanged
End Sub

Private Sub mnuProgrammer_Click()
   Set CurrentMenu = ProgrammerMenu
   MenuChanged
End Sub

Private Sub txtPaid_Change()
   UpdateTotals
End Sub

Public Sub UpdateAll()
   UpdateListBox
   UpdateTotals
End Sub

' Schaltflächen für aktuelles Menü laden
Public Sub LoadCommandButtons(mnu As clsMenu1)
   Dim buttonindex%
   buttonindex = 1
   ' Vorhandene Schaltflächen entladen
   Do While buttonindex <= HighestButtonIndex
      Unload cmdMenu(buttonindex)
      buttonindex = buttonindex + 1
   Loop
   ' Neue Schaltflächen laden
   cmdMenu(0).Visible = False ' Falls Menü leer

   ' Menüpunkte laden und anzeigen
   For HighestButtonIndex = 0 To mnu.ItemCount - 1
      If HighestButtonIndex > 0 Then
         Load cmdMenu(HighestButtonIndex)
      End if
      With cmdMenu(HighestButtonIndex)
         .Visible = True
         .Top = cmdMenu(0).Top + HighestButtonIndex _
          * 450
         .Caption = mnu.ItemName(HighestButtonIndex)
      End With
   Next HighestButtonIndex

   ' HighestButtonIndex erniedrigen, falls hinzugefügt
   If HighestButtonIndex > 0 Then
      HighestButtonIndex = HighestButtonIndex - 1
```

```
    End If
End Sub

' Menü-Schaltflächen aktualisieren und
' aktuelle  Bestellung löschen
Public Sub MenuChanged()
    CurrentOrder.Clear
    LoadCommandButtons CurrentMenu
    UpdateAll
End Sub
```

Listing 3.5: Listing für frmRetail3.frm, das Hauptformular des Retail3-Beispiel-Projekts

Die Anwendung legt ein aktuelles Menüobjekt (clsMenu1) und ein aktuelles Bestellungsobjekt (clsOrder1) an. In diesem Beispiel existiert nur ein einziges Bestellungsobjekt, wohingegen drei verschiedene Menüs definiert sind. Die Menüs werden während des `Form_Load`-Ereignisses geladen. Ich hoffe, daß Ihnen bewußt ist, daß die Menüobjekte auch problemlos aus einer Datenbank oder einer externen Datei geladen werden könnten, um die harte Codierung der Menüpunkte zu vermeiden. Vielleicht sollten Sie dann den Code zum Laden als Methode implementieren und nicht als Bestandteil des Formular-Codes.

Der größte Teil des Codes ist bewußt so weit und so nah wie möglich aus dem Retail2-Projekt übernommen worden. Der einzige größere Unterschied besteht in der Verwaltung der Schaltflächen. Das Formular benötigt einen Weg, die Schaltflächen für ein bestimmtes Menü festzulegen. Dies wird über die Funktion `LoadCommandButtons` realisiert.

3.3.1 Weitere kleine Änderungen

Wenn Sie das Beispiel-Projekt Retail3 sehen, werden Sie sich fragen, wozu der Schnickschnack mit der objektorientierten Programmierung gut sein soll. Mag ja sein, daß der Code nun ein wenig besser organisiert ist, ansonsten sieht es nur nach mehr Arbeit aus. Worin also liegt der Vorteil?

Nun, erinnern Sie sich an den netten Manager oder Kunden, dem auffiel, daß sich die Effizienz dadurch verdoppeln ließ, daß mehrere Bestellungen gleichzeitig im Register gehalten werden können. Die Änderungen in der Version Retail2 wären ziemlich mühselig geworden. Ist das nun bei der Version Retail3 anders?

Hier nun also unsere Änderungen an der Retail3-Version. Zunächst werden alle Referenzen auf `clsOrder1` in `clsOrder2` und auf `clsMenu1` in `clsMenu2` geändert. Dies hat nichts mit der Funktionalität der Anwendung zu tun – es dient nur zur Unterscheidung der Beispieldateien. Dann wird folgende Zeile im Modul `clsOrder2` eingefügt.

```
Public AssociatedMenu As clsMenu2
```

Diese Eigenschaft ist notwendig, damit das `clsOrder2`-Objekt sich merken kann, welches Menü es verwendet.

Im Allgemein-Abschnitt des Formulars ändern Sie die Deklaration `CurrentOrder` und fügen ein Array ein, das die in diesem Beispiel implementierten drei Bestellungen aufnehmen soll:

```
Dim CurrentOrder As clsOrder2
Dim Orders(2) As New clsOrder2
```

Im `Form_Load`-Ereignis ergänzen Sie folgendes:

```
Set CurrentOrder = Orders(0)
Set CurrentOrder.AssociatedMenu = CurrentMenu
```

Diese Anweisung initialisiert die aktuelle Bestellung.

Plazieren Sie nun ein `Label`-Control `lblOrder` auf dem Formular sowie ein Control-Array von drei `CommandButton`-Controls (`cmdOrder1`), wie Sie es in Abbildung 3.2 sehen.

Abb. 3.2: So sieht das Formular frmRetail4 zur Laufzeit aus

Fügen Sie folgenden Code im Formular ein:

```
' Zu einer anderen Bestellung umschalten
    Private Sub cmdOrder1_Click(Index As Integer)
       Set CurrentOrder = Orders(Index)
       If CurrentOrder.AssociatedMenu Is Nothing Then
          ' Wenn die Bestellung das erste Mal gewählt wird,
          ' wird das aktuelle Menü verwendet
          Set CurrentOrder.AssociatedMenu = CurrentMenu
       End If
       Set CurrentMenu = CurrentOrder.AssociatedMenu
       LoadCommandButtons CurrentMenu
```

```
    UpdateAll
    lblOrder.Caption = "Current order: " & Index + 1
End Sub
```

Wird die Bestellung gewechselt, wird das aktuelle Menü abhängig von der Bestellung gesetzt. Zugleich werden die Schaltflächen dem verwendeten Menü angepaßt, und die `ListBox` und Summen werden entsprechend der aktuellen Bestellung aktualisiert. Das Markieren des jeweiligen Menüpunkts im Popup-Menü überlasse ich Ihnen als Fingerübung.

Und das war dann auch schon alles.

Der gesamte Code, der die Arbeit mit Bestellungen behandelt, betrifft das `clsOrder2`-Objekt, so daß für das Umschalten zwischen Bestellungen allein das Umschalten zwischen den Objekten ausreicht.

Damit die Anwendung für reale Anforderungen robuster und praktikabler wird, sollten Sie noch ein paar kleine Änderungen vornehmen. So könnten Sie die Menüs als Arrays anlegen und Menü-Objekte dynamisch aus einer Datenbank laden und anlegen. Die Objekte können in einem Array oder in einer Collection gehalten werden. Das Visual-Basic-Menü könnte als Menü-Control-Array angelegt werden, das die Objekte in der Collection widerspiegelt. Die vermeidet wieder die hart-codierten `clsMenu1`-Objekte.

Da nun das Objekt `clsOrder2` eine Referenz auf das Menü, das es verwendet, enthält, können Sie die Funktion `CalculateTotal` auch leicht als Methode der Klasse `clsOrder2` implementieren. Das überlasse ich Ihnen zur Übung.

Das Objekt `clsOrder2` kann dahingehend erweitert werden, daß es die Bezeichnungen der Bestellposten und deren Preise enthält, statt diese nur als Element des aktuellen Menüs zu referenzieren. So könnte der Preis eines einzelnen Postens geändert werden, anstatt ausschließlich die vorgegebenen Menüpreise zu verwenden. So kann etwa ein Hamburger für 1,99 und eine anderer für 1,12 verkauft werden (wobei es allerdings auf vertrauenswürdige Kassierer ankommt).

3.4 Wiederum Theorie der objektorientierten Programmierung

Objektorientierte Programmierung ist genausowenig nur ein Modewort wie auch nur ein Marketing-Begriff. Es steckt viel mehr dahinter – objektorientierte Programmierung ist eine praktische Technik.

Das haben wir an den Beispielen Retail2 und Retail3 gesehen. Beide Versionen unterscheiden sich fast gar nicht voneinander, außer in der Verwendung von Objekten – und das läßt Welten zwischen ihnen liegen. Bei der einen Version war eine größere Erweiterung recht schwierig zu implementieren, was bei der zweiten dann ein Kinderspiel war.

Warum wird die Implementierung bei objektorientierter Programmierung so viel einfacher? Nun, Software für heutige Anforderungen wird zunehmend komplexer. Der Prozeß der Entwicklung von Anwendungen besteht zum größten Teil darin, diese Komplexität in den Griff zu bekommen. Alles, was dazu beiträgt, große Probleme in kleine, handhabbare Einheiten aufzuteilen, kann daher nur von Vorteil sein. Und objektorientierte Programmierung ist *das* Werkzeug schlechthin, große Probleme in kleine Objekte zu zerlegen.

Beim effizienten Entwurf des `clsMenu1`-Objekts haben Sie einen Datentyp definiert und einen Satz an Funktionen angelegt, der mit den Daten arbeitet. Die Klassen-Funktionen `AddMenItem`, `ItemCount`, `ItemPrice` und `ItemName` sind alles, was Sie brauchen, um diese Menüobjekte zu verwalten. Sie können diese Funktionen als Interface dieses Objekts betrachten. (Sie werden später noch sehen, daß der Begriff *Interface* der treffende technische Terminus für einen Satz von Funktionen ist, die von einem Objekt offengelegt werden.)

Das Objekt selbst enthält Datenstrukturen (hier ein Array des benutzerdefinierten Datentyps `MenuItem`) und Code zur Verwaltung dieser Datenstrukturen. Ein Objekt kann auch eigene interne Funktionen zur Be- und Verarbeitung der Daten des Objekts enthalten. Das Objekt `clsMenu1` nutzt diese Möglichkeit allerdings nicht.

Die Quintessenz der objektorientierten Programmierung lautet: Nachdem Sie ein Objekt implementiert haben, also seine Datenstrukturen angelegt und den internen Code zur Bearbeitung dieser Datenstrukturen geschrieben (und getestet) haben, können Sie sich die Freiheit nehmen, nicht mehr daran zu denken. Sie brauchen sich keine Gedanken mehr über den Code und die Datenstrukturen innerhalb der Klasse zu machen. Sie brauchen sich nur noch das Interface zu merken – die Funktionen, die die Klasse offenlegt – und können sich auf wichtigere Dinge konzentrieren.

Sie brauchen sich auch keine Sorgen mehr darüber zu machen, daß Änderungen in einem Teil Ihres Programms in Konflikt mit der Funktionsweise eines Objekts geraten, oder daß andere Funktionen oder Objekte zufällig die Datenstrukturen innerhalb des Objekts beeinträchtigen könnten. Auf die Daten des Objekts kann nur über die Interface-Funktionen zugegriffen werden. Diese Trennung zwischen Objekten kann zur langfristigen Zuverlässigkeit einer Anwendung beitragen, vor allem, wenn mehrere Entwickler daran mitschreiben. Auch das Hinzufügen von neuen Features zu einer Anwendung wird vereinfacht. Es genügt, die Interfaces einiger Klassen zu erweitern, anstatt den gesamten Code der Anwendung zu überarbeiten oder neu zu schreiben.

Die Fähigkeit, Daten und Funktionalität hinter einem Satz von Methoden und Eigenschaften zu verbergen, ist ein Merkmal objektorientierter Programmierung, die *Kapselung* genannt wird. Ein zweites Merkmal wird anhand der Antwort auf folgende Frage offensichtlich: Was tut die `Clear`-Anweisung in Visual Basic?

Nun, bei der `ListBox` wird die `Clear`-Anweisung zum Leeren der `ListBox` verwendet. Beim Objekt `clsOrder1` dient sie zum Löschen der Bestellung. Sie können tatsächlich beliebig viele `Clear`-Anweisungen antreffen und verwenden. Jedes Objekt wird mit seinen internen Daten genau das anstellen, was als Operation hinter der Anweisung (Methode) z.B. `Clear` für ein Objekt definiert wird. Die Fähigkeit, daß Objekte den gleichen Namen für eine Methode verwenden können, wird *Polymorphie* genannt.

Das sei noch einmal ausdrücklich betont: Der Vorteil von Polymorphie liegt im Beitrag zur Reduktion von Komplexität. Andernfalls müßten in Visual Basic für jedes Objekt andere Befehle definiert werden – z.B. `ClearListBox` bei einer `ListBox` oder eben `ClearClsOrder1`, um eine Bestellung zu löschen. Wenn 20 verschiedene Objekte eine `Clear`-Methode hätten, müßten Sie 20 verschiedene Funktionen definieren. Das hieße 20 Funktionen merken und gegebenenfalls in der Dokumentation nachschlagen. Eine einzige `Clear`-Methode ist einfacher zu behalten.

Natürlich werden Puristen an dieser Stelle anmerken, daß sich Polymorphie eigentlich darauf bezieht, daß in einer Programmiersprache die korrekten Funktionen aufgerufen werden können, wenn Objekte über eine generische Objekt-Variable referenziert werden. Wenn Sie etwa eine Variable vom Typ `Object` angelegt haben, und diese Variable eines aus Dutzenden von Objekten referenzieren kann, die alle über eine `Clear`-Methode verfügen, wird immer korrekt die `Clear`-Methode des gerade von der Variablen referenzierten Objekts aufgerufen. Aus der Sicht eines Visual-Basic-Programmierers ist die exakte Definition von Polymorphie irrelevant – Sie brauchen lediglich zu wissen, daß Methode- und Eigenschaftennamen problemlos von mehreren Objekten verwendet werden können.

Ein drittes Merkmal objektorientierter Programmierung wird *Vererbung* genannt. Vererbung im klassischen Sinne ist in Visual Basic nicht implementiert. Daher entspricht Visual Basic nicht der klassischen Definition einer objektorientierten Sprache. Seien Sie nicht traurig darüber. Visual Basic ermöglicht Ihnen die Lösung vieler Aufgaben, die gewöhnlich über Vererbung gelöst werden. Darüber werden Sie mehr in Kapitel 5, »Aggregation und Polymorphie«, lesen.