

Teil G

Entwickeln von SmartClient-Anwendungen

- 741 Programmieren mit Windows Forms**
 - 821 Im Motorraum von Formularen und Steuerelementen**
 - 863 GDI+ zum Zeichnen von Formular- und Steuerelementinhalten verwenden**
 - 893 Entwickeln von Steuerelementen**
 - 933 Mehreres zur gleichen Zeit erledigen – Threading in .NET**
 - 983 SQL Server 2005 und ADO.NET**
-

In den vergangenen Kapiteln haben Sie viele Techniken und Konzepte von Visual Basic und dem .NET-Framework kennen gelernt. Für das Entwickeln von Anwendungen unter Windows ist das Beherrschen dieser Techniken ein notwendiges Muss, aber sicherlich nicht alles. In diesem Teil geht es darum, vieles des Gelernten in einen sinnvollen Kontext zu setzen, um die Voraussetzung für das Entwickeln regelrechter Windows-Anwendungen bzw. – wie man sie heutzutage so gerne nennt – SmartClient-Anwendungen¹ zu schaffen. Das anschließende Kapitel – Programmieren von Windows Forms-Anwendungen – kümmert sich in erster Linie darum.

Fortgeschrittene Techniken, wie das Entwickeln von Steuerelementen, der Einsatz von GDI+, Threading und ADO.NET, runden diesen letzten Teil des Buches ab.

Übrigens: Für das Verstehen dieses Kapitels hilft es, wenn Sie ► Kapitel 3 durchgearbeitet haben, damit Ihnen der Aufbau von Formularen und der Einsatz der wichtigen Steuerelemente klar ist, so wie ► Kapitel 26, das das Anwendungsframework von Visual Basic 2005 für Windows-Anwendungen erklärt. Das Verständnis von Klassen und Schnittstellen, wie es der Klassenteil dieses Buches vermittelt, ist ebenfalls von großem Vorteil.

¹ Wobei »SmartClient« nicht nur impliziert, dass sie – im Gegensatz zu vielen Web-gestützten Anwendungen – über eine eigene Intelligenz verfügen – also »smart« sind – sondern auch mit Datenquellen verbunden werden können, wie beispielsweise Web Services oder SQL Server-Datenbanken, dabei aber auch offline, also unverbunden, ihren Dienst verrichten. Frei interpretiert kann man daher jede Windows Forms-Anwendung, die in irgendeiner Form Daten mit einem entfernten System austauscht, als SmartClient-Anwendung ansehen.

27 Programmieren mit Windows Forms

743	Strukturieren von Daten in Windows Forms-Anwendungen
754	Formulare zur Abfrage von Daten verwenden – Aufruf von Formularen aus Formularen
766	Überprüfung auf richtige Benutzereingaben in Formularen
771	Anwendungen über die Tastatur bedienbar machen
775	Über das »richtige« Schließen von Formularen
777	Grundsätzliches zum Darstellen von Daten aus Auflistungsklassen in Steuerelementen
777	Darstellen von Daten aus Auflistungen im ListView-Steuerelement
782	Verwalten von Daten aus Auflistungen mit dem DataGridView-Steuerelement
805	Entwickeln von MDI-Anwendungen
813	Vererben von Formularen

Zentraler Bestandteil einer jeden SmartClient-Anwendung sind Windows Formulare – im .NET-Jargon kurz *WinForms* genannt. Jedes Windows Formular dient als Träger für Steuerelemente, mit denen der Anwender eine SmartClient-Anwendung letzten Endes bedienen kann. Dabei gibt es im Grunde genommen nur drei Arten von Formularen, die in Ihren Programmen zum Einsatz kommen:

- Formulare, die Daten darstellen,
- Formulare, die der Datenerfassung dienen, und
- Formulare, mit der Sie Programmooptionen einstellen.

Und eigentlich sind die Formulare, die unter die letzte Kategorie fallen, auch nichts anderes als Formulare die der Datenerfassung dienen.

Typische SmartClient-Anwendungen unter Windows haben dabei immer den gleichen Aufbau: Sie enthalten ein Hauptanwendungsfenster, das den eigentlichen Arbeitsbereich der Anwendung darstellt. Und von diesem Formular aus verzweigt der Funktionsbereich der Anwendung mithilfe weiterer Formulare, die entweder dazu dienen, weitere Daten (vielleicht anderen Typs) zu erfassen oder die Parameter bestimmter Funktionen einzustellen, die auf die Hauptdaten der Anwendung in irgendeiner Form angewendet werden (doch auch das ist ja auch wieder eine Art von Datenerfassung).

Unterscheiden kann man schließlich noch zwei Grundformen der Datenaufbereitung und Form der Verarbeitung durch den Anwender: Es gibt Anwendungstypen, die ihre Daten in mehrere Dokumente aufteilen, und in so genannten MDI-Anwendungen (*Multi Dokument Interface* – etwa: Mehrdokumenten-Benutzerschnittstelle) für den Anwender zur Bearbeitung zur Verfügung halten. Verschiedene Datensammlungen gleicher Art (Textdokumente, Bilder, Adresskarten, etc.) können dabei in verschiedenen untergeordneten Fenstern zur gleichen Zeit dargestellt und zur Bearbeitung angeboten werden.

Im Gegensatz dazu gibt es die häufiger vorkommenden SDI-Anwendungen (*Single Document Interface*-Anwendungen), bei denen die Daten eben nicht über mehrere Dokumentenfenster verteilt werden, sondern lediglich ein einzelnes Fenster den durch die Anwendung vorgegebenen Datentyp zur Bearbeitung anbietet.

Für welche Art der Benutzeroberfläche sich ein Entwickler entscheidet, ist eigentlich mehr Geschmackssache als funktionelle Notwendigkeit. Viele Grafikprogramme sind als MDI-Anwendungen ausgelegt – der Anwender kann also »gleichzeitig« mehrere Grafiken als untergeordnete Fenster öffnen; kann er sie aber wirklich gleichzeitig bearbeiten? Moderne Computer ließen das, was die Performance anbelangt, sicherlich zu. Die »Schwachstelle«, wenn man das als solches überhaupt bezeichnen kann, ist dabei eher der Anwender. Ich jedenfalls würde mir es nicht zutrauen, zwei Texte wirklich gleichzeitig zu schreiben, und Microsoft Word ist wahrscheinlich aus ähnlichen Überlegungen bestes Beispiel dafür, wie aus einer MDI-Anwendung (bis Word 97) sinnvollerweise eine SDI-Anwendung (ab Word 2000) werden kann.

Doch ganz gleich welchen Aufbau einer SmartClient-Anwendung Sie auch wählen und welche Art von Daten Ihre Anwendung auch bearbeitet: Entscheiden Sie sich für den Einsatz von Windows Formularen unter .NET, stehen Sie immer wieder vor der Lösung derselben Detailprobleme. Um nur einige zu nennen:

- Wie strukturieren SmartClient-Anwendungen am besten die Daten, die sie verwalten und dem Anwender zur Bearbeitung anbieten?
- Wie kann man Formulare aufrufen, die Parameter für andere Formulare »einholen«?
- Wie kann die Dateneingabe für den Anwender benutzerfreundlich erfolgen?
- Welche Techniken kann man anwenden, um eine Benutzeroberfläche ergonomischer zu gestalten und sie – getreu dem Motto: »das Auge isst mit« – ohne großen Aufwand verschönern?
- Wie geht man vor, um eine Benutzeroberfläche auch in anderen Sprachen anbieten zu können.

Dieses Kapitel setzt sich exakt mit dieser Problematik auseinander, und Sie sehen, dass es einen anderen Ansatz verfolgt, als Sie ihn vielleicht aus vielen anderen Büchern kennen.

Natürlich kann ein Windows Forms-Kapitel hunderte von Seiten füllen, indem es beschreibt, wie das 76. Steuerelement im Detail funktioniert und über welche Eigenschaften, Methoden und Ereignisse es verfügt. Doch damit sind Sie noch keinen Schritt weiter, denn es geht ja schließlich darum, das Gelernte im Kontext der Anwendungsentwicklung umzusetzen. Und außerdem können Sie sich diese Art von Information auch aus der Online-Hilfe von Visual Studio beschaffen.

Viel interessanter ist es doch, Lösungen für verschiedene, häufig auftretende Problemstellungen zu bekommen, und genau diesen Ansatz möchte dieses Kapitel verfolgen. Technische Details, die natürlich für das Verständnis des einen oder anderen Ansatzes notwendig sind, kommen in den jeweiligen Erklärungen natürlich nicht zu kurz.

Strukturieren von Daten in Windows Forms-Anwendungen

Stellen Sie sich vor, Sie stehen vor der Aufgabe, eine kleine, einfache Adressenverwaltung zu entwickeln. Und um das Szenario wirklich einfach zu halten, gehen wir bei diesem Beispiel für den Moment davon aus, dass Sie die Daten nicht in einer Datenbank sondern in einer einfachen Datei speichern. Die Frage, die sich vielen als erstes stellt, lautet dann oftmals: Wie schaffe ich es, dass meine Anwendung von allen relevanten Stellen aus an die Daten herankommt? Und damit stehen viele bereits vor dem ersten wirklich schwerwiegenden Problem.

Das eigentliche Problem an dieser Stelle ist aber gar nicht die noch nicht bekannte Antwort. Im Grunde genommen ist es nämlich schon die Frage, die falsch gestellt wurde: Auch wenn sie die Klassenprogrammierung bereits aus dem Effeff kennen, fällt es vielen Entwicklern schwer, die OOP-Programmierung auch »im großen Rahmen« umzusetzen. Sie möchten nämlich, dass die Daten irgendwo zentral (und vor allen Dingen: global zugänglich) irgendwo in der Anwendung beherbergt werden. Und wenn, um beim Eingangsbeispiel zu bleiben, eine *Adressen-Bearbeiten*-Funktion aufgerufen wird, dann wird einfach ein neuer Dialog aufgerufen, der dann die Adressen entsprechend bearbeitet und dazu selbst auf den Gesamtdatenbestand der Anwendung zugreift. Soll eine neue Adresse erfasst werden, dann passiert der gleiche Datenzugriff eben aus dem »*Adresse-Neu-Anlegen*-Dialog« heraus. Jede Programmfunktion greift also nach Gutdünken auf den Datenbestand zu. Das Chaos ist dabei buchstäblich vorprogrammiert.

Der Ansatz sollte ein anderer sein. In einer gelungenen Windows Forms-Anwendung sollte eine bestimmte Komponente die wesentlichen Daten, die der Anwender bearbeiten können soll, kapseln. Und nur diese Komponente sollte in der Lage sein dürfen, bestimmte Daten zum Bearbeiten »herauszurücken« oder Änderungen bestimmter Daten wieder entgegenzunehmen. Andere Komponenten stellen die Datenstrukturen selbst dar. Und wieder andere Komponenten übernehmen die Kommunikation mit dem Anwender.

Aus diesem Grund ist es bei der OOP-Programmierung gängige Praxis, Anwendungen in verschiedene Ebenen bzw. Schichten – (englische Layer oder Tier) zu unterteilen.

- Eine Schicht stellt dabei die Datenschicht bereit. Das sind in der Regel Klassen, die die eigentlichen Daten beherbergen.
- Eine andere Schicht stellt die Anwendungslogik (auch *Geschäftslogik* oder neudeutsch: *Business Logic*) bereit. Das sind wiederum Klassen, die die Klassen, die von der Datenschicht zur Verfügung gestellt werden, einbinden und kapseln und die Funktionalität zur Verfügung stellen, um die Daten im Sinne des Anwenders aufzubereiten und zu verarbeiten.
- Eine letzte Schicht stellt die Schnittstelle zwischen dem Benutzer und der Anwendungslogik her. Sie hat nur Delegatenfunktion, was in diesem Zusammenhang bedeutet: Sie nimmt Anwenderanforderungen entgegen und leitet sie an die Schicht weiter, die die Anwendungslogik enthält.

Die Einteilung in solche Schichten bringt den Entwicklern dann gleich mehrere Vorteile, die sich vor allen Dingen längerfristig auswirken:

- Verschiedene Ihrer Anwendungen können die einzelnen Komponenten bzw. Schichten, aus denen eine mehrschichtige Anwendung besteht, ebenfalls verwenden.

- Umfangreiche Anwendungen bzw. Anwendungen, die komplexe Datenstrukturen zu verwalten haben, lassen sich einfacher in kleinere Portionen aufteilen.
- Ihr Anwendung kann relativ schnell einem Facelift unterzogen werden: Ohne dass die eigentliche Anwendungslogik wirklich ersetzt werden müsste, können Sie ihr ein neues Make-up verpassen und sie sieht damit in Null Komma nichts neuer und moderner aus.
- Verschiedene Schichten können einfacher im Team entwickelt werden. Durch die Aufteilung einer Anwendung in Schichten bzw. Komponenten lassen sich komplexe Anwendungen viel besser im Team entwickeln.

Wenn Sie große Teile dieses Buches bereits studiert haben, dann sind Sie vielleicht schon einem Ansatz einer kleinen Adressverwaltung hier und da begegnet. Ging es in diesen Kapiteln noch darum, bestimmte Details an plausiblen Beispielen zu erklären (► Kapitel 20, generische Auflistungen, ► Kapitel 22, XML-Serialisierung), so will das folgende Beispiel diese technischen Puzzleteile aufgreifen, neu strukturieren, wieder zusammensetzen und daraus eine kleine, unkomplizierte aber dennoch vollständige Musteranwendung schaffen, die nicht nur prinzipiell und theoretisch zeigt, wie eine saubere OOP-Entwicklung einer SmartClient-Anwendung aussehen sollte.

BEGLEITDATEIEN: Sie finden das Projekt für das folgende Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\Adresso01\`.

So Sie sich die ► Kapitel 20 und 22 schon zu Gemüte geführt haben, werden Sie bei diesem Projekt in Sachen technischen Gegebenheiten zunächst nichts Neues entdecken. Dennoch unterscheiden sich Projekt als auch Code deutlich von der letzten Version dieses Beispiels aus ► Kapitel 22, und das wird bereits beim Betrachten des Projektmappen-Explorers deutlich, denn:

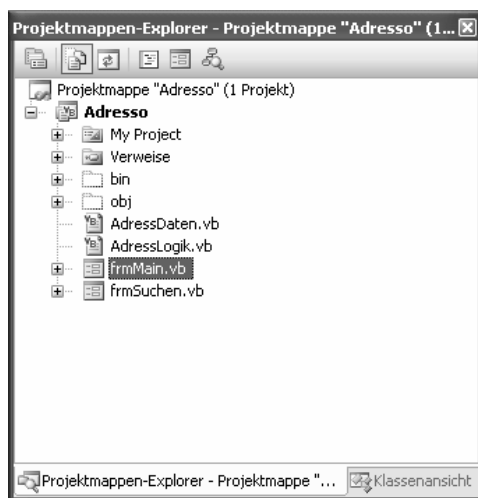


Abbildung 27.1: Schon im Projektmappen-Explorer wird deutlich, wie das Programm in drei Schichten eingeteilt ist

- Die Codedatei *AdressDaten.vb* enthält die Datenschicht der Anwendung – die Adresse-Klasse, die eine einzelne Adresse speichert.
- Die Codedatei *AdressLogik.vb* sorgt dafür, dass die einzelnen Adresse-Datensätze in einer Auflistung zusammengefasst und dort funktionell verwaltet werden. Und hier zeigt sich auch das erste Mal der Unterschied zwischen dieser Version von »Adresso« und der, die Sie noch aus ► Kapi-

tel 22 kennen. Die gesamte Funktionalität des Programms – beispielsweise zum Sortieren oder Suchen von Adressen – findet nun an dieser Stelle statt, und nicht mehr, wie in der »Vorgängerversion«, lustig verteilt mal bei der Benutzeroberflächenschicht in `frmMain` und mal, wie das Serialisieren, in der abgeleiteten Auflistung Adressen. Alle Aufgaben sind in dieser Version, so, wie es geplant war, sauber voneinander getrennt (wovon Sie sich im anschließenden Codelisting auch selber überzeugen können).

- In `frmMain.vb` schließlich, der dritten Schicht, finden nur noch Delegationsaufgaben statt. Funktionen, die der Anwender beispielsweise aus dem Pulldown-Menü abrufen, werden einfach an die Adresslogik weitergereicht. Das Komplizierteste, was diese Benutzeroberflächenschicht noch leisten muss, ist die Darstellung der durch die Adresslogik verwalteten Adresse-Objekte im `ListView`-Steuerelement.

Übrigens: Das Umgestalten des Codes vom Beispiel aus ► Kapitel 22 zu der Version, wie Sie sie hier sehen, ist ein Vorgang, den man unter dem Begriff *Refactoring* versteht.

Die Codedateien sehen in dieser Version des Beispiels nun folgendermaßen aus:

Datenschicht: `AdressDaten.vb`

`AdressDaten.vb` enthält nur noch die Datenstruktur zur Speicherung einer einzelnen Adresse in Form von Adresse-Klasse:

```
<Serializable(> _
Public Class Adresse

    'Membervariablen, die die Daten halten:
    Protected myMatchcode As String
    Protected myName As String
    Protected myVorname As String
    Protected myStraße As String
    Protected myPLZ As String
    Protected myOrt As String
    Protected myGeburtsdatum As Date

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' </summary>
    ''' <remarks>Ein parameterloser Konstruktor wird benötigt,
    ''' um XML-Serialisierung zu ermöglichen.</remarks>
    Sub New()
        MyBase.New()
    End Sub

    'Konstruktor - legt eine neue Instanz an
    Sub New(ByVal Matchcode As String, ByVal Name As String, ByVal Vorname As String, _
        ByVal Straße As String, ByVal Plz As String, ByVal Ort As String, _
        ByVal Geburtsdatum As Date)
        myMatchcode = Matchcode
        myName = Name
        myVorname = Vorname
        myStraße = Straße
        myPLZ = Plz
    End Sub
End Class
```

```

    myOrt = Ort
    myGeburtsdatum = Geburtsdatum
End Sub

Public Overridable Property Matchcode() As String
    Get
        Return myMatchcode
    End Get
    Set(ByVal Value As String)
        myMatchcode = Value
    End Set
End Property

Public Overridable Property Name() As String
    Get
        Return myName
    End Get
    Set(ByVal Value As String)
        myName = Value
    End Set
End Property

Public Overridable Property Vorname() As String
    Get
        Return myVorname
    End Get
    Set(ByVal Value As String)
        myVorname = Value
    End Set
End Property

Public Overridable Property Straße() As String
    Get
        Return myStraße
    End Get
    Set(ByVal value As String)
        myStraße = value
    End Set
End Property

Public Overridable Property PLZ() As String
    Get
        Return myPLZ
    End Get
    Set(ByVal Value As String)
        myPLZ = Value
    End Set
End Property

Public Overridable Property Ort() As String
    Get
        Return myOrt
    End Get

```

```

        Set(ByVal Value As String)
            myOrt = Value
        End Set
    End Property

    Public Overridable Property Geburtsdatum() As Date
        Get
            Return myGeburtsdatum
        End Get
        Set(ByVal Value As Date)
            myGeburtsdatum = Value
        End Set
    End Property

    Public Overrides Function ToString() As String
        Return Matchcode & ":" & Name & ", " & Vorname
    End Function
End Class

```

Sie werden übrigens feststellen, jedenfalls wenn Sie das Beispiel aus ► Kapitel 22 kennen, dass in dieser Version ein weiteres Adressdatenfeld hinzugekommen ist – auch wenn es nicht in der Adressenliste des Formulars zu sehen ist. Das Geburtsdatum ist nicht nur ein wesentlicher Bestandteil jeder noch so kleinen Adressverwaltung – dieses Feld eignet sich insbesondere auch zur Demonstration von Eingabeüberprüfungen bei neuen Adressen, mit dem sich der nächste Abschnitt beschäftigt.

TIPP: Wenn Sie wissen wollen, wie im Programm die dazugehörigen »Zufallsdatums« erzeugt werden, werfen Sie einen Blick in die Codedatei *AdressLogik.vb* und dort in die Funktion *ZufallSadressen*. Die Erklärung finden Sie in den Kommentaren am Ende dieser Prozedur.

Anwendungslogik: AdressLogik.vb

Alle Funktionen der Anwendung sind in dieser Codedatei zusammengefasst. Sie bildet damit den Code, der am intensivsten refaktoriert wurde. Die Änderungen im Vergleich zum Beispiel aus ► Kapitel 22 sind fett hervorgehoben.

```

<Serializable(> _
Public Class Adressen
    Inherits List(Of Adresse)

    Private mySortierenNach As AdressenSortierenNach

    ''' <summary>
    ''' Erstellt eine neue Instanz dieser Klasse.
    ''' </summary>
    ''' <remarks></remarks>
    Sub New()
        MyBase.New()
    End Sub

```

```

''' <summary>
''' Erstellt eine neue Instanz dieser Klasse
''' und ermöglicht das Übernehmen einer vorhandenen Auflistung in diese.
''' </summary>
''' <param name="collection">Die generische Adresse-Auflistung, deren Elemente
''' in diese Auflistungsinstanz übernommen werden sollen.</param>
''' <remarks></remarks>
Sub New(ByVal collection As ICollection(Of Adresse))
    MyBase.New(collection)
End Sub

''' <summary>
''' Gibt alle Adresse-Objekte als Instanz dieser Klasse zurück,
''' in denen der angegebene Suchbegriff vorkam.
''' </summary>
''' <param name="Text">Suchbegriff, nachdem diese Auflistung durchsucht werden soll.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function Suchen(ByVal Text As String) As Adressen

    'In dieser Adressen-Auflistung wird das Suchergebnis gesammelt.
    Dim locAdressen As New Adressen()

    For Each locAdresse As Adresse In Me
        If locAdresse.Name Like Text Then
            locAdressen.Add(locAdresse)
            'Direkt zum 'Next' springen.
            Continue For
        ElseIf locAdresse.Vorname Like Text Then
            locAdressen.Add(locAdresse)
            Continue For
        ElseIf locAdresse.Straße Like Text Then
            locAdressen.Add(locAdresse)
            Continue For
        ElseIf locAdresse.Ort Like Text Then
            locAdressen.Add(locAdresse)
            Continue For
        ElseIf locAdresse.PLZ.ToString Like Text Then
            locAdressen.Add(locAdresse)
        End If
    Next

    Return locAdressen
End Function

```

Die Suchenfunktion wurde im Gegensatz zum Beispiel aus ► Kapitel 22 auch funktionell abgeändert: Sie findet nun nicht nur die erste Adresse, auf die der Suchbegriff zutraf, sondern liefert eine Adressen-Auflistung zurück, die alle Adressen enthält, in denen der Suchbegriff vorkam. Darüber hinaus findet es den Suchbegriff nicht nur in einer bestimmten einstellbaren Datenspalte, sondern in der gesamten Adresszeile.

```

Public Sub Sortieren(ByVal SortierenNach As AdressenSortierenNach)
    'Das Array mithilfe eines Comparison-Delegaten sortieren
    mySortierenNach = SortierenNach
    Me.Sort(New Comparison(Of Adresse) (AddressOf AdressenVergleicher))
End Sub

'Der Comparison-Delegat zum Vergleichen zweier Elemente.
Private Function AdressenVergleicher(ByVal x As Adresse, ByVal y As Adresse) As Integer

    Try
        'Sortierung in Abhängigkeit zur Suchspalte durchführen
        Select Case mySortierenNach
            Case AdressenSortierenNach.Matchcode
                Return x.Matchcode.CompareTo(y.Matchcode)
            Case AdressenSortierenNach.Name
                Return x.Name.CompareTo(y.Name)
            Case AdressenSortierenNach.Ort
                Return x.Ort.CompareTo(y.Ort)
            Case AdressenSortierenNach.PLZ
                Return x.PLZ.CompareTo(y.PLZ)
            Case AdressenSortierenNach.Straße
                Return x.Straße.CompareTo(y.Straße)
            Case Else
                Return x.Vorname.CompareTo(y.Vorname)
        End Select
    Catch ex As Exception
        'Abfangen, dass einer der Suchstrings Nothing ist.
        'Der ist dann immer kleiner als alle anderen!
        Return -1
    End Try
End Function

''' <summary>
''' Serialisiert alle Elemente dieser Auflistung in eine XML-Datei.
''' </summary>
''' <param name="Dateiname">Der Dateiname der XML-Datei.</param>
''' <remarks></remarks>
Sub XMLSerialize(ByVal Dateiname As String)
    Dim locXmlWriter As New XmlSerializer(GetType(Adressen))
    Dim locXmlDatei As New StreamWriter(Dateiname)
    locXmlWriter.Serialize(locXmlDatei, Me)
    locXmlDatei.Flush()
    locXmlDatei.Close()
End Sub

''' <summary>
''' Generiert aus einer XML-Datei eine neue Instanz dieser Auflistungsklasse.
''' </summary>
''' <param name="Dateiname">Name der XML-Datei, aus der die Daten für
''' diese Auflistungsinstantz entnommen werden sollen.</param>
''' <returns></returns>
''' <remarks></remarks>

```

```

Public Shared Function XmlDeserialize(ByVal Dateiname As String) As Adressen
    Dim locXmlLeser As New XmlSerializer(GetType(Adressen))
    Dim locXmlDatei As New StreamReader(Dateiname)
    Return CType(locXmlLeser.Deserialize(locXmlDatei), Adressen)
End Function

''' <summary>
''' Liefert eine Instanz dieser Klasse mit Zufallsadressen zurück.
''' </summary>
''' <param name="Anzahl">Anzahl der Zufallsadressen, die erzeugt werden sollen.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function ZufallsAdressen(ByVal Anzahl As Integer) As List(Of Adresse)
    'Aus Platzgründen ausgelassen
End Function
End Class

```

```
Public Enum AdressenSortierenNach
```

```
Matchcode
```

```
Name
```

```
Vorname
```

```
Straße
```

```
PLZ
```

```
Ort
```

```
End Enum
```

Benutzeroberflächenschicht: frmMain.vb

Und schließlich gibt es die letzte Schicht der Anwendung, die, wie schon gesagt, nur noch Delegationsaufgaben erfüllt. Alle Funktionen, die das Formular durch entsprechende Ereignisse vom Benutzer entgegennimmt, leitet es direkt an die Klasse Adressen weiter, in der dann die eigentliche Problemlösung vollzogen wird.

Die Stellen, an denen es die Programmkontrolle zur eigentlichen Durchführung der »Geschäftslogik« an die Anwendungslogikschicht übergibt, sind im folgenden Listing fett markiert. Nicht relevante Codeauszüge sind aus Platzgründen ausgeblendet.

Auf einen funktionellen Unterschied zum Beispiel aus ► Kapitel 22 möchte ich in diesem Zusammenhang noch besonders hinweisen: Das Anklicken einer Adresszeile in der Liste bewirkt die Anzeige einiger Detaildaten in der Statuszeile des Formulars. Der entsprechende Code, der dieses Verhalten steuert, befindet sich ganz am Ende des dokumentierten Listings, das Sie im Folgenden finden.

```

Public Class frmMain

    Private myAdressen As Adressen          ' Alle Adressen

    'Dieses hier nicht als Ereignis einbinden - wäre von hinten, durch die Brust, ins Auge...
    'Besser überschreiben - frmListenForm ist schließlich von Form abgeleitet!
    Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
        'Basismethode aufrufen, damit das Formular machen kann,
        'was es machen muss, um alles einzurichten.
        MyBase.OnLoad(e)
    End Sub

```

```

'Instanzieren.
myAdressen = New Adressen

'Die ListView-Spalten einrichten.
Vorbereitungen()

'Die Adressenliste darstellen.
ElementeDarstellen()
End Sub

'Richtet lediglich die ListView ein.
Sub Vorbereitungen()
' Aus Platzgründen ausgelassen
End Sub

Sub ElementeDarstellen()

'Unterdrückt Neuzeichnen-Ereignisse bis zum
'nächsten EndUpdate; dadurch geht der Aufbau
'der Elemente schneller und wackelt nicht.
Me.lvwAdressen.BeginUpdate()

'Alle Elemente der ListView löschen.
Me.lvwAdressen.Items.Clear()

'Für jedes Element der Liste wird ElementInListe aufgerufen.
myAdressen.ForEach(New Action(Of Adresse)(AddressOf ElementInListe))

'So werden die Spaltenbreiten optimal angepasst.
For Each locCol As ColumnHeader In Me.lvwAdressen.Columns
    locCol.Width = -2
Next

'Aufbau der ListView ist beendet.
Me.lvwAdressen.EndUpdate()
End Sub

'Der Action-Delegat: für jedes Element der Liste wird diese Aktion durchgeführt.
Sub ElementInListe(ByVal Element As Adresse)
'Neues ListView-Element - Matchcode kommt zuerst.
Dim locLvwItem As New ListViewItem(Element.Matchcode)

'Die Untereinträge setzen
With locLvwItem.SubItems
    .Add(Element.Name)
    .Add(Element.Vorname)
    .Add(Element.Straße)
    .Add(Element.PLZ)
    .Add(Element.Ort)
End With

```

```

'Zum Wiederfinden Referenz in Tag
locLvwItem.Tag = Element

'Zur Listview hinzufügen
lvwAdressen.Items.Add(locLvwItem)
End Sub

'Wird aufgerufen, wenn eine der Spalten angeklickt wird.
Private Sub lvwAdressen_ColumnClick(ByVal sender As Object, ByVal e As
System.Windows.Forms.ColumnClickEventArgs) Handles lvwAdressen.ColumnClick

    'Spaltennummer, die in e.Column steht, in AdressenSortierenNach konvertieren
    Dim locNach As AdressenSortierenNach = CType(e.Column, AdressenSortierenNach)

'Sortieren
myAdressen.Sortieren(locNach)

    'Die Elemente neu sortiert darstellen
    ElementeDarstellen()
End Sub

Private Sub tsmAdresseSuchen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmAdresseSuchen.Click
    Dim locSuchFormular As New frmSuchen
    Dim locErsterGefundener As Boolean

    'Den Suchbegriff merken, damit der Predicate-Delegat darauf
    'zugreifen kann.
    Dim locSuchbegriff As String = locSuchFormular.Suchbegriff
    If String.IsNullOrEmpty(locSuchbegriff) Then
        Return
    End If

    'Alle gefundenen Elemente durchlaufen, und...
For Each locAdresse As Adresse In myAdressen.Suchen(locSuchbegriff)

        'jeweils alle ListView-Elemente durchsuchen und überprüfen, ob ...
        For Each locLvwItem As ListViewItem In Me.lvwAdressen.Items

            '... die Tag-Referenz der Referenz des gesuchten Objekts entspricht.
            If locLvwItem.Tag Is locAdresse Then

                'Gefunden! ListView-Element markieren,
                locLvwItem.Selected = True

                'und dafür sorgen, dass es der erste gefundene
                'Eintrag im sichtbaren Bereich liegt.
                If Not locErsterGefundener Then
                    locLvwItem.EnsureVisible()
                    locErsterGefundener = True
                End If
            End If
        End For
    End For
End Sub

```

```

        'Gefunden, wir müssen in der ListView
        'nicht weitersuchen!
        Exit For
    End If
Next
Next
End Sub

Private Sub tsmAdressenLöschen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmAdressenLöschen.Click

    If lvwAdressen.SelectedItems IsNot Nothing AndAlso lvwAdressen.SelectedItems.Count > 0 Then

        Dim locDr As DialogResult = MessageBox.Show(My.Resources.MB_Nachfrage_Löschen_Body, _
            My.Resources.MB_Nachfrage_Löschen_Titel, MessageBoxButtons.YesNo, _
            MessageBoxIcon.Question)
        If locDr = Windows.Forms.DialogResult.Yes Then
            For Each lvwItem As ListViewItem In lvwAdressen.SelectedItems
                myAdressen.Remove(DirectCast(lvwItem.Tag, Adresse))
            Next
            ElementeDarstellen()
        End If
    End If
End Sub

Private Sub tsmZufallsadressenAnfügen_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles tsmZufallsadressenAnfügen.Click
    ' Aus Platzgründen ausgelassen
End Sub

'Wird aufgerufen, wenn Anwender Datei/Adressliste speichern wählt.
Private Sub tsmAdresslisteSpeichern_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles tsmAdresslisteSpeichern.Click

    Dim dateiSpeichernDialog As New SaveFileDialog

    With dateiSpeichernDialog

        'Ermitteln des Dateinamens aus Platzgründen ausgelassen

        'Adressen serialisieren
        myAdressen.XMLSerialize(.FileName)
    End With
End Sub

'Wird ausgelöst, wenn der Anwender einen Eintrag oder mehrere Einträge der Liste selektiert
Private Sub lvwAdressen_SelectedIndexChanged(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles lvwAdressen.SelectedIndexChanged

    If lvwAdressen.SelectedIndices.Count = 0 Then

        'Kein Eintrag selektiert
        tsslAusgewählteAdresse.Text = "Keine Adresse selektiert"
    End If
End Sub

```

```

ElseIf lvwAdressen.SelectedIndices.Count > 1 Then
    'Mehrere Einträge selektiert
    tss1AusgewählteAdresse.Text = "Mehrere Adressen selektiert"
Else
    'Genau ein Eintrag selektiert, dann die Grunddaten und das Geburtsdatum darstellen
    Dim locAdresse As Adresse

    'Das ursprüngliche Adresse-Objekt aus der Liste holen.
    'Dieses wurde beim Erstellen der ListViewItem's in deren
    'Tag-Eigenschaften gespeichert, und auf diese Weise wird
    'die Relation zwischen Listeneintrag und eigentlichem Objekt hergestellt.
    locAdresse = DirectCast(lvwAdressen.SelectedItem(0).Tag, Adresse)

    'Text für die Darstellung in der Statuszeile zusammenbasteln:
    Dim locStatusText As String
    locStatusText = locAdresse.Matchcode & ": "
    locStatusText &= locAdresse.Name & ", " & locAdresse.Vorname & " - Geboren am "
    locStatusText &= locAdresse.Geburtsdatum.ToLongDateString

    'Text in der Statuszeile darstellen.
    tss1AusgewählteAdresse.Text = locStatusText
End If
End Sub

```

```
End Class
```

Wie Sie sehen, ist das Komplizierteste, was diese Schicht noch übernehmen muss, die Darstellung der Daten im ListView-Steuerelement. Für den vorhandenen Funktionsumfang ist der reine Formularcode aber nunmehr vergleichsweise kompakt. Genaueres über die Darstellung von Auflistungsklassen im ListView-Steuerelement erfahren Sie im ► Abschnitt »Darstellen von Daten aus Auflistungen im ListView-Steuerelement« ab Seite 777.

Formulare zur Abfrage von Daten verwenden – Aufruf von Formularen aus Formularen

Die seit Erscheinen von Visual Basic .NET wohl häufigste Frage zu Windows Forms-Anwendungen, der man in den verschiedensten Newsgroups begegnet, lautet: »Wie rufe ich aus einem Formular ein Formular auf – beispielsweise um Daten für die Hauptanwendung vom Anwender zu erfragen?«

Zweifelsohne gibt es auf diese Frage nicht *die eine* Antwort; es gibt natürlich mehrere Wege, Formulare zu realisieren, die zum Abfragen von bestimmten Parametern dienen und diese an das aufrufende Formular zurückliefern.

Die Technik, die ich Ihnen im Folgenden vorstellen möchte, ist in Absprache mit vielen Entwicklerkollegen aber wohl eine der gängigsten. Sie hat den Vorteil, dass sie ein Muster darstellt, das Sie in Anwendungen, die diese Technik des »Dateneinsammelns« erfordern, immer wieder einsetzen können.

Bei den Realisierungsüberlegungen und zum späteren besseren Verständnis des Codes, der diese Aufgabe übernimmt, sollte man sich die folgenden Punkte vor Augen führen:

Formulare, die zum Einlesen bestimmter Parameter dienen (im schon bekannten Beispielprojekt könnte das zum Beispiel das Erfassen einer neuen Adresse oder das Editieren einer vorhandenen sein), sollten ...

- ... als modale Dialoge dargestellt werden (siehe nächster Abschnitt).
- ... Daten, mit denen das Formular vorbelegt werden muss, vom aufrufenden Formular entgegennehmen können, und die durch den Anwender veränderten Daten wieder zurückliefern.
- ... möglichst als Funktion aufgerufen werden können, und selbst dafür sorgen, dass sie sich darstellen und schließlich wieder schließen.
- ... vor ihrem Schließen die Richtigkeit der durch den Anwender eingegebenen Daten überprüfen.

Der Umgang mit modalen Formularen

Bevor wir uns mit dem Erstellen eines grundsätzlichen Musters für das Erfragen von Daten mit modalen Formularen beschäftigen, sollten wir uns für ein besseres Verständnis zunächst mit modalen Dialogen an sich beschäftigen. Es gibt hier nämlich einiges zu entdecken, was nicht allgemein bekannt ist.

Modale Dialoge verhalten sich wie Meldungsfelder (MessageBox-Dialoge), was bedeutet, dass Sie die darunter liegenden Dialoge nicht erreichen können, solange der modale Dialog aktiv ist. Ein Anwender muss also den Dialog erst ausfüllen und bestätigen (oder alternativ abbrechen, wenn er den Dialog versehentlich aufgerufen hat), um mit der Anwendung weiterarbeiten zu können. Im Gegensatz dazu gibt es übrigens nicht-modale Dialoge, die sich auf gleicher Ebene mit anderen Fenstern befinden, also parallel zu anderen Fenstern bedient werden können.

Modale Formulare haben eine eigene Nachrichtenwarteschlange

In Windows selbst funktioniert die Kommunikation zwischen verschiedenen Komponenten auf Basis so genannter Nachrichtenwarteschlangen (das nächste Kapitel hält dazu mehr bereit, falls Sie sich für die Details interessieren). Wenn Sie eine Windows Forms-Anwendung mit einem Hauptformular starten, dann richtet das Anwendungsframework für diese Anwendung genau eine dieser Nachrichtenwarteschlangen ein. Sobald ein Ereignis eintritt, wird dieses Ereignis windows-intern dieser Nachrichtenschlange hinzugefügt und letzten Endes von entsprechenden .NET-Klassen des Anwendungsframework bzw. des Hauptformulars der Anwendung bearbeitet.

Das ändert sich in dem Moment, in dem Sie einen modalen Dialog ins Leben rufen. In diesem Fall wird eine neue Nachrichtenwarteschlange für den modal dazustellenden Dialog eingerichtet. Der Programmablauf stoppt dann an der Stelle, an der Sie den entsprechenden modalen Dialog aufrufen:

```
Dim locFrm As New IrgeneinFormular
locFrm.ShowDialog()
'Hier geht es erst weiter, wenn der Dialog beendet wurde.
```

Steuern von modalen Dialogen mit der DialogResult-Eigenschaft

Da sich modale Dialoge wie Meldungsfelder verhalten (oder besser: es sind), kommt einer ihrer Eigenschaften eine besondere Funktion zu: der Eigenschaft `DialogResult`. Das Formular bestimmt durch Setzen dieser Eigenschaft, wie das »Ergebnis« der Formularbedienung aussah. Doch das Setzen dieser Eigenschaft im Formular bewirkt noch mehr: Sobald Sie diese Eigenschaft im Rahmen eines Ereignisses auf einen anderen Wert als `None` setzen, wird der Dialog beendet!

Das bedeutet: Verfügt Ihr Dialog, den Sie modal aufrufen möchten, beispielsweise über eine `OK`-Schaltfläche, dann genügt eine einzige Zeile Code, nicht nur um das Dialogergebnis zu setzen, sondern auch, um den Dialog zu schließen – etwa auf die folgende Weise:

```
'Wird aufgerufen beim Auslösen von OK-Schaltfläche.  
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click  
  
    'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.  
    Me.DialogResult = Windows.Forms.DialogResult.OK  
  
End Sub
```

HINWEIS: Wichtig dabei ist es zu wissen, dass beim bloßen Setzen von `DialogResult` die Ressourcen des Dialoges nicht implizit wieder freigegeben werden. Wenn das Formular kritische Ressourcen blockiert, sollte es also in der Anwendung eine Instanz geben, die für das explizite Aufrufen von `Dispose` des Formulars sorgt. Eine gute Technik ist es, das Formular in einen `Using...End Using`-Block einzuschließen. Mehr zum Thema `Dispose` erfahren Sie in ► Kapitel 12. Der Umgang mit `Using` wird in ► Kapitel 6 beschrieben (Abschnitt: » Gezieltes Freigeben von Objekten mit `Using`«).

Ein Muster für modale Formulare implementieren, die sich selber verwalten können

Nachdem diese wichtigen Fakten bekannt sind, schreiten wir zur eigentlichen Aufgabe. Ziel ist es, ein Muster für einen Dialog zu schaffen, der, wie eingangs erwähnt, ...

- ... eine Funktion zur Verfügung stellt, die Parameter zur Bearbeitung entgegennimmt, den eigentlichen Dialog darstellt und die vom Anwender eingegebenen und geprüften Daten zurückliefert,
- ... `OK` und `Abbrechen` implementiert, mit denen der Dialog geschlossen wird,
- ... eine zentrale Routine anbietet, in der die Daten überprüft werden und
- ... dafür sorgt, dass die Ressourcen des Dialogs wieder freigegeben werden.

Beginnen wir mit dem ersten Teil, einer Funktion, die wir im Dialog als `Member-Methode` implementieren und aufrufen können, sobald uns eine Instanz des Dialogs vorliegt. Die Funktion innerhalb des Formularcodes sieht in etwa wie folgt aus:

```
Public Class frmFürDatenabfrage  
  
    'Datenmember, auf die FormClosing zugreifen und die eine Funktion zurückliefern kann  
    Private myDatenTyp As Datentyp  
  
    'Member-Funktion des Formulars, die das Bearbeiten von Daten regelt.  
    Public Function DatenBearbeiten(ByVal adr As Datentyp) As Datentyp  
        'Damit wird dafür gesorgt, dass die Ressourcen für das Formular  
        'nach Aufruf sofort wieder freigegeben werden!
```

```

Using Me
  'TODO: Daten in die Maske kopieren.

  'Dialog modal darstellen. Das Programm "hält" an dieser
  'Stelle quasi an, und nur noch Ereignisse innerhalb
  'dieses Formulars werden verarbeitet.
  Me.ShowDialog()

  'Hier geht's erst weiter, wenn OK oder Abbrechen geklickt wurde.
  Return myDatentyp
End Using
End Function
.
.
.
End Class

```

Diese Funktion deckt den ersten und den letzten Punkt unserer Todo-Liste bereits ab. Mit der Anweisung

```
Me.ShowDialog()
```

sorgt sie dafür, dass der Dialog als modaler Dialog dargestellt wird. Wenn die Funktion aufgerufen wird, nachdem eine Instanz des Formulars, das sie beherbergt, eingerichtet wurde, hält die Programmausführung an exakt dieser Stelle an und übergibt die Kontrolle der Nachrichtenwarteschlange des Formulars. Alle Ereignisse, die durch den Anwender bei der Bedienung des Dialogs auftreten können, werden von diesem Zeitpunkt an vom Formular verarbeitet, die entsprechenden Ereignisbehandlungsroutinen auch aufgerufen. Erst wenn der Dialog auf irgendeine Art und Weise wieder geschlossen wird, geht es an dieser Stelle hinter dem ShowDialog-Aufruf weiter, und das Formular kann schließlich die Daten, die die aufrufende Instanz angefordert hat (oder Nothing, wenn der Dialog abgebrochen wurde) zurückliefern.

Damit die Ressourcen des Formulars freigegeben werden, wenn die Funktion beendet ist, klammern wir den gesamten Funktionscode in Using mit Bezug auf das eigene Formular (Me) ein. Dieses Konstrukt sorgt dafür, dass jedes Objekt, das das Formular selbst als Member-Variable speichert, einerseits noch zurückgegeben werden kann, die Formularinstanz anschließend aber (und *erst* dann) noch »entsorgt« wird.

Implementieren müsste diese Formulkasse nun noch lediglich eine Methode, die die übergebenen Daten, die bearbeitet werden sollen, in die entsprechenden Steuerelemente des Formulars schreibt.

Die aufrufende Instanz würde diese Funktion nun wie folgt verwenden:

```

Dim locFrm As New frmFürDatenabfrage
Dim locDatentypZumbearbeiten as Datentyp = IrgendwasWasBearbeitetWerdenSoll
Dim locDatentypZurück As Datentyp = locFrm.DatenBearbeiten(locDatentypZumbearbeiten)

```

Auffällig ist die Objektvariable myDatentyp, die zu Beginn des Formulars als Klassen-Member deklariert wurde; sie global für das ganze Formular zu deklarieren, scheint auf den ersten Blick unnötig zu sein. Klarer wird ihr Gültigkeitsbereich, wenn wir den fehlenden Formularcode, der die Datenüberprüfung und das Schließen (bzw. Abbrechen) des Formulars regelt, im Kontext betrachten:

```

'Wird aufgerufen beim Auslösen von OK-Schaltfläche.
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click

    'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.
    Me.DialogResult = Windows.Forms.DialogResult.OK
End Sub
'Wird aufgerufen beim Auslösen von Abbrechen-Schaltfläche.
Private Sub btnAbbrechen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles btnAbbrechen.Click

    'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.
    Me.DialogResult = Windows.Forms.DialogResult.Cancel

End Sub

'Wird aufgerufen, wenn das Formular geschlossen werden soll.
Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)
    MyBase.OnClosing(e)
    'Überprüfung des Formulars nur, wenn OK geklickt wurde:
    If Me.DialogResult = Windows.Forms.DialogResult.OK Then

        myDatentyp = DatenDerMaskeÜberprüfenUndAlsObjektErmitteln

        'Wenn die ermittelten Daten Nothing und damit NICHT in Ordnung war, ...
        If myDatentyp Is Nothing Then
            '... bleibt der Dialog, der ja eigentlich geschlossen werden soll,
            'offen - und das erreichen wir durch Setzen von
            e.Cancel = True
        End If
    Else
        'Dieser Zeile kann nur erreicht werden, wenn Abbrechen
        'ausgelöst wurde. Dann wird auf jeden Fall Nothing als
        'Funktionsergebnis zurückgegeben.
        myDatentyp = Nothing
    End If
End Sub

```

Das Beenden des Dialogs wird eingeleitet, wenn eine der vorhandenen Schaltflächen *OK* oder *Abbrechen* des Formulars betätigt wird. Die jeweiligen Ereignisbehandlungsroutinen machen dazu nichts weiter, als die DialogResult-Eigenschaft des Formulars zu ändern, und damit das Schließen des Formulars anzustoßen.

Das Schließen durch *OK* darf allerdings nur dann erfolgen, wenn die Daten, die der Anwender in der Maske zuvor eingegeben hat, auch validiert werden konnten, anderenfalls muss nicht nur eine entsprechende Fehlermeldung ausgegeben werden oder eine Kennzeichnung der falsch eingegebenen Datenfelder erfolgen – das Formular darf vor allen Dingen nicht geschlossen werden.

Das Closing-Ereignis des Formulars verfügt über die Fähigkeit, das Schließen des Formulars zu verhindern (wenn es nicht gerade mit `Form.Dispose` sozusagen mit Brachialgewalt abgeschossen wurde, doch das sollte eh die Ausnahme bleiben). Also ergibt es Sinn, die Prüfung auf Richtigkeit der Anwendereingaben in exakt diese Ereignisbehandlungsroutine zu verlegen.

Übrigens: Denken Sie daran, dass wir uns in der OOP-Programmierung befinden, und dass Formularen im Grunde genommen auch nichts anderes als Klassen sind. Es wäre also von hinten durch die Brust ins Auge, wenn wir innerhalb eines Formulars ein Formularereignis einbänden. Stattdessen ist es guter Programmierstil, die Methode zu überschreiben, die das Ereignis auslöst – damit bekommt Ihr Formularcode das Ereignis natürlich schon ein paar Schritte vorher zu Gesicht. Im Codeauszug selbst passiert exakt das mit `Protected Overrides Sub OnClosing`.

Der Code dieser Methode ist nun auch von entscheidender Bedeutung: Nur wenn das Schließen des Formulars mit *OK* eingeleitet wurde (`DialogResult` war *OK*), muss eine Überprüfung (und am besten auch noch die gleichzeitige Erstellung der Instanz einer Datentypklasse aus den Formulardatenfeldern) erfolgen. Im Ereignisbehandlungscode wird das durch die Zeilen

```
If Me.DialogResult = Windows.Forms.DialogResult.OK Then
    myDatentyp = DatenDerMaskeÜberprüfenUndAlsObjektErmitteln
```

erledigt. Und hier zeigt sich auch, wieso das Vorhalten einer Objektvariablen für die Datenrückgabe als Klassen-*Member* so wichtig ist: `OnClosing` muss in diese Objektvariable das Ergebnis mit den vom Anwender eingegebenen Daten ablegen, und damit der Ausgangscode der Formularaufruffunktion `DatenBearbeiten` diese Objektvariable als Funktionsergebnis zurückliefern kann, muss auch ihr der Zugriff auf die Objektvariable gestattet sein.

Hat die Datenüberprüfungsfunktion Fehler in der Anwendereingabe gefunden, dann sollte sie zu diesem Zeitpunkt schon entsprechende Hinweismeldungen im Formular gezeigt haben. Darüber hinaus gibt sie `OnClosing` in diesem Fall `Nothing` zurück, und `OnClosing` weiß anhand dessen, dass ein Fehler aufgetreten ist. Es verwendet nun folgendes Verfahren, um das Schließen des Formulars zu verhindern:

```
If myDatentyp Is Nothing Then
    '... bleibt der Dialog, der ja eigentlich geschlossen werden soll,
    'offen - und das erreichen wir durch Setzen von
    e.Cancel = True
End If
```

Durch Setzen der `Cancel`-Eigenschaft im Ereignisparameter von `OnClosing` lässt sich das Schließen des Formulars an dieser Stelle verhindern.

TIPP: Falls Sie übrigens mehr zur Kommunikation zwischen Komponenten in .NET-Anwendungen in Form von Ereignissen erfahren wollen, werfen Sie einen Blick in ► Kapitel 15.

Implementierung des Musters in einer Anwendung

Theorie ist das eine, Praxis das andere. Wie sähe ein solcher Code nun aus, den wir in unsere schon bekannte Adresso-Anwendung implementieren wollten. Werfen wir dazu zunächst einen Blick auf die nächste Implementierungsstufe dieses Beispiels.

BEGLEITDATEIEN: Sie finden das Projekt für das folgende Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\Adresso02\`.

Sie entdecken nach dem Programmstart in diesem Beispielprojekt zwei neue Funktionen im Menü *Bearbeiten*: *Neue Adresse eingeben* und *Adresse bearbeiten* (Siehe Abbildung 27.2). Die letzte Funktion lässt sich natürlich nur dann aufrufen, wenn Sie zuvor eine Adresse in der Liste selektiert haben. Ein und dasselbe Formular in der Anwendung deckt letzten Endes beide Funktionen ab.

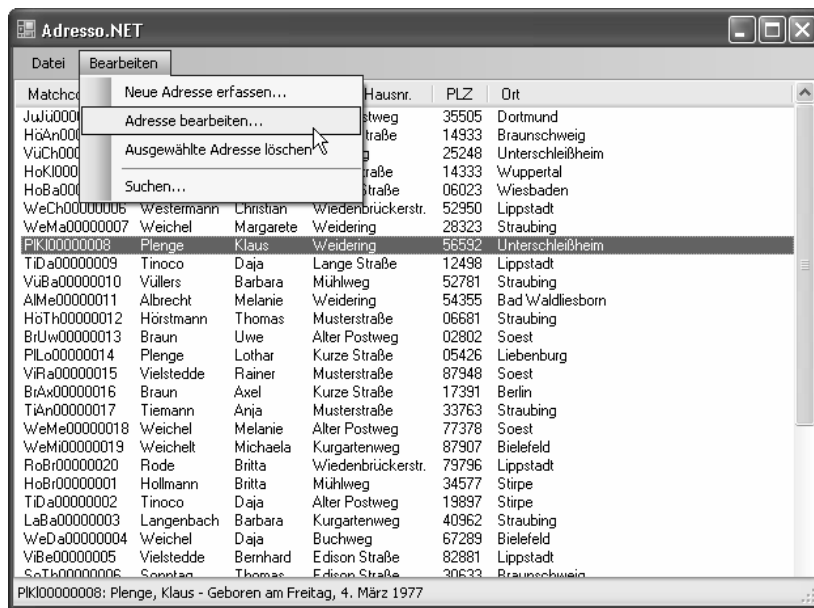


Abbildung 27.2: In dieser Ausbaustufe von Adresso können Sie die vorhandene Liste der Adressen bearbeiten und um neue ergänzen

Dennoch unterscheidet sich die Funktionsweise des Formulars bei beiden Funktionen marginal: Beim Bearbeiten einer vorhandenen Adresse lässt sich der Matchcode nicht ändern. Beim Neuanlegen muss er mit eingegeben werden. Der modale Dialog zum Ändern bzw. Erfassen einer Adresse sieht wie folgt aus:



Abbildung 27.3: Mit diesem modal dargestellten Dialog werden Adressdaten in Adresso bearbeitet oder neu angelegt

Damit das in ► Abschnitt »Strukturieren von Daten in Windows Forms-Anwendungen« (ab Seite 743) postulierte Prinzip der mehrschichtigen Programmentwicklung gewahrt bleibt, befinden sich die Aufrufe des Formulars nicht im Code des Hauptformulars sondern im Logikteil der Anwendung.

Der Aufruf des Formulars zum Bearbeiten gestaltet sich wie folgt:

```
''' <summary>
''' Ruft den Dialog zum Bearbeiten einer Adresse auf.
''' Liefert bei erfolgreicher Bearbeitung True zurück.
''' </summary>
''' <param name="adr">Adresse, die in dieser Instanz vorhanden sein muss
''' und bearbeitet werden soll.</param>
''' <returns></returns>
''' <remarks></remarks>
Public Function AdresseBearbeiten(ByVal adr As Adresse) As Boolean
    'Zu suchende Adresse merken, damit der Find-Predicate den
    'richtigen Index ausfindig machen kann.
    myAktuelleAdresse = adr

    'Index ermitteln
    Dim locIndex As Integer = Me.FindIndex(New Predicate(Of Adresse)(AddressOf IndexFinder))

    'Adresse ist nur bei Index > -1 in Auflistung vorhanden.
    'Die Nummer dieser Adresse merken wir uns in locIndex.
    If locIndex > -1 Then

        'Adresse bearbeiten.
        Dim locFrm As New frmAdresseNeuUndBearbeiten
        Dim locAdresse As Adresse = locFrm.AdresseBearbeiten(adr)

        'Die Adresse wurde geändert...
        If locAdresse IsNot Nothing Then
            '...und kann gegen die ursprüngliche ausgetauscht werden.
            Me(locIndex) = locAdresse

            'True zurückliefern, damit die aufrufende Instanz weiß, dass
            'es die Liste aktualisieren muss, um die Änderungen widerzuspiegeln.
            Return True
        End If
    End If
    'Keine Änderung
    Return False
End Function
```

Der relevante Codeteil ist hier fett hervorgehoben. Sie werden sehen, dass das Muster zum Aufruf des Dialogs prinzipiell dem entspricht, wie wir ihn im vorherigen Abschnitt (ab Seite 756) besprochen haben.

Übrigens: Um den Index des in der Liste markierten Objektes zu finden, bedient sich die Prozedur der FindIndex-Methode, die, angewendet auf die generische List(Of)-Auflistung, genauso wie deren Find-Methode zusammen mit einer Predicate-Instanz arbeitet. Genaueres zu diesem Thema finden Sie in ► Kapitel 20.

Ein wenig komplizierter ist das Neuanlegen einer Adresse, da sich hier ein besonderes Problem stellt: Wenn eine neue Adresse angelegt wird, darf der Matchcode, den der Anwender bestimmt, natürlich keinem schon vorhandenen Matchcode der Adressenliste entsprechen. Aus diesem Grund muss es eine Möglichkeit geben, dass der Dialog, in dem die Eingabe einer neuen Adresse erfolgt, die Adres-

sen auf einen bereits vorhandenen gleichen Matchcode kontrolliert. Nun entspricht es aber nicht der Aufteilung einer Anwendung in Schichten, wenn ein Erfassungsdialo Zugriff auf die komplette Datenschicht nehmen darf. Abermals helfen uns Delegationen (besprochen in ► Kapitel 15) aus dem Dilemma. Anstatt dem Dialog eine Referenz auf die kompletten Adressendaten zu geben, mit denen er anschließend eine Kontrolle auf einen vorhandenen Matchcode in der Liste durchführen könnte, übergeben wir ihm einfach die *Adresse einer Funktion*, die das für ihn erledigt. Zu gegebener Zeit ruft er über den ihm übergebenden Delegaten einfach eine Prozedur in der Anwendungslogikschicht auf, die diese Überprüfung für ihn vornimmt – das Prinzip der strikten Schichttrennung bleibt damit gewahrt.

Der Code in der Anwendungslogik, der das Neuanlegen einer Adresse einleitet, sieht damit wie folgt aus:

```
''' <summary>
''' Delegat zum Überprüfen eines schon vorhandenen Matchcodes.
''' </summary>
''' <param name="Matchcode">Matchcode, der überprüft werden soll.</param>
''' <returns>True, wenn der Matchcode bereits existierte.</returns>
''' <remarks></remarks>
Public Delegate Function MatchcodeCheckenDelegate(ByVal Matchcode As String) As Boolean

''' <summary>
''' Ruft den Dialog zum Hinzufügen einer neuen Adresse auf.
''' Liefert bei erfolgreicher Eingabe True zurück.
''' </summary>
''' <returns>True, wenn die Adresse dieser Auflistung hinzugefügt wurde.</returns>
''' <remarks></remarks>
Public Function NeueAdresse() As Boolean
    'Formularinstanz erstellen
    Dim locFrm As New frmAdresseNeuUndBearbeiten

    'Neue Adresse ermitteln. Dazu die Routine als Delegat übergeben,
    'die überprüft, ob der Matchcode in dieser Auflistung bereits vorhanden ist.
    Dim locAdresse As Adresse = locFrm.NeueAdresse(AddressOf MatchcodeChecken)

    'Nur wenn eine gültige Adresse ermittelt wurde...
    If locAdresse IsNot Nothing Then

        '...diese dieser Auflistung hinzufügen.
        Me.Add(locAdresse)

        'True zurückliefern, damit die aufrufende Instanz weiß, dass
        'es die Liste aktualisieren muss, um die Änderungen widerzuspiegeln.
        Return True
    End If
    'Keine Änderung
    Return False
End Function
```

```

''' <summary>
''' Überprüft, ob ein vorhandener Matchcode bereits in der Auflistung vorhanden ist.
''' </summary>
''' <param name="Matchcode"></param>
''' <returns></returns>
''' <remarks></remarks>
Public Function MatchcodeChecken(ByVal Matchcode As String) As Boolean
    For Each locItem As Adresse In Me
        If locItem.Matchcode = Matchcode Then
            Return True
        End If
    Next
    Return False
End Function

```

Delegat und korrelierende Funktion entsprechen der ersten und letzten fett hervorgehobenen Listingzeile im oben zu sehenden Codeausschnitt. Der »mittlere Teil« entspricht fast wörtlich dem im vorherigen Abschnitt besprochenen Muster.

Mit dieser Kapselung der Funktionalität beschränkten sich die Änderungen, die an der Benutzeroberfläche in *frmMain.vb* nötig sind, auf folgende beiden Ereignisbehandlungsmethoden:

```

'Wird aufgerufen, wenn der Anwender im Menü Bearbeiten
'den Menüpunkt Neue Adresse erfassen anklickt.
Private Sub tsmNeueAdresseErfassen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmNeueAdresseErfassen.Click

    'Nur aktualisieren, wenn das Neuanlegen erfolgreich war.
If myAdressen.NeueAdresse() Then
        ElementeDarstellen()
    End If

End Sub

'Wird aufgerufen, wenn der Anwender im Menü Bearbeiten
'den Menüpunkt Adresse bearbeiten anklickt.
Private Sub tsmAdresseBearbeiten_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmAdresseBearbeiten.Click

    'Herausfinden, ob es überhaupt eine selektierte Adresse gibt
    If lvwAdressen.SelectedIndices.Count > 0 Then

        'Erste selektierte Adresse wird bearbeitet
        Dim locAdresse As Adresse = DirectCast(lvwAdressen.SelectedItems(0).Tag, Adresse)

        'Nur aktualisieren, wenn das Bearbeiten erfolgreich war.
If myAdressen.AdresseBearbeiten(locAdresse) Then
            ElementeDarstellen()
        End If
    End If

End Sub

```

Nun ist die eigentliche Realisierungsweise des Formularcodes zum Erfassen einer Adresse natürlich von besonderem Interesse. Abgesehen von der Überprüfung der Formulardaten, auf die der nächste Abschnitt eingehen will, entspricht dieser Code ebenfalls zu 99 % dem vorgestellten Muster:

```
Public Class frmAdresseNeuUndBearbeiten

    'Hier steht das Ergebnis der beiden Funktionen
    Private myAdresse As Adresse

    'Delegat, mit dessen Hilfe ein doppelter Matchcode überprüft wird.
    Private myMatchcodeCheckenProc As Adressen.MatchcodeCheckenDelegate

    ''' <summary>
    ''' Stellt diesen Dialog dar, und ermittelt das Objekt einer neuen Adresse.
    ''' </summary>
    ''' <param name="matchcodeChecken">Delegat, der auf eine Prozedur verweist, die doppelte
    ''' Matchcodes in der Auflistung überprüft.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Function NeueAdresse(ByVal matchcodeChecken As Adressen.MatchcodeCheckenDelegate) As Adresse

        'Damit wird dafür gesorgt, dass die Ressourcen für das Formular
        'nach Aufruf sofort wieder freigegeben werden!
        Using Me
            'Dialogtitel anpassen
            Me.Text = "Adresse bearbeiten"

            'Delegaten zuordnen - den brauchen wir später, um auf doppelte Matchcodes zu prüfen.
            myMatchcodeCheckenProc = matchcodeChecken

            'Dialog modal darstellen. Das Programm "hält" an dieser
            'Stelle quasi an, und nur noch Ereignisse innerhalb
            'dieses Formulars werden verarbeitet.
            Me.ShowDialog()

            'Hier geht's erst weiter, wenn OK oder Abbrechen geklickt wurde.
            Return myAdresse
        End Using
    End Function

    ''' <summary>
    ''' Stellt diesen Dialog dar, lässt den Anwender eine Adresse bearbeiten und liefert
    ''' ein neues Adresse-Objekt zurück, das die Änderungen widerspiegelt.
    ''' </summary>
    ''' <param name="adr">Adresse-Objekt, das bearbeitet wird.
    ''' ACHTUNG! Das Rückgabeobjekt stellt eine neuen Instanz dar!</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Function AdresseBearbeiten(ByVal adr As Adresse) As Adresse

        'Damit wird dafür gesorgt, dass die Ressourcen für das Formular
        'nach Aufruf sofort wieder freigegeben werden!
        Using Me
```

```

'Dialogtitel anpassen
Me.Text = "Adresse bearbeiten"

'Daten in die Maske kopieren.
DatenInMaske(adr)

'Matchcode darf nicht editiert werden:
txtMatchcode.ReadOnly = True

'Dialog modal darstellen. Das Programm "hält" an dieser
'Stelle quasi an, und nur noch Ereignisse innerhalb
'dieses Formulars werden verarbeitet.
Me.ShowDialog()

'Hier geht's erst weiter, wenn OK oder Abbrechen geklickt wurde.
Return myAdresse
End Using

End Function

'Wird aufgerufen beim Auslösen von OK-Schaltfläche.
Private Sub btnOK_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles btnOK.Click

'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.
Me.DialogResult = Windows.Forms.DialogResult.OK

End Sub

'Wird aufgerufen beim Auslösen von Abbrechen-Schaltfläche.
Private Sub btnAbbrechen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
Handles btnAbbrechen.Click

'Das Zuordnen eines Wertes für DialogResult beendet den Dialog.
Me.DialogResult = Windows.Forms.DialogResult.Cancel

End Sub

'Wird aufgerufen, wenn das Formular geschlossen werden soll.
Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)
MyBase.OnClosing(e)
'Überprüfung des Formulars nur, wenn OK geklickt wurde:
If Me.DialogResult = Windows.Forms.DialogResult.OK Then

'Daten aus Maske gibt ein Adresse-Objekt nur dann zurück,
'wenn die Daten in Ordnung waren.
myAdresse = DatenAusMaske()

'Wenn myAdresse also Nothing und damit NICHT in Ordnung war, ...
If myAdresse Is Nothing Then

'... bleibt der Dialog, der ja eigentlich geschlossen werden soll,
'offen - und das erreichen wir durch Setzen von
e.Cancel = True

```

```

        End If
    Else
        'Dieser Zeile kann nur erreicht werden, wenn Abbrechen
        'ausgelöst wurde. Dann wird auf jeden Fall Nothing als
        'Funktionsergebnis zurückgegeben.
        myAdresse = Nothing
    End If
End Sub

'Überprüft die Eingaben im Formular, und liefert
'im Erfolgsfall ein fix-und-fertiges Adresse-Objekt
'aus den Eingabefeldern zurück.
Private Function DatenAusMaske() As Adresse
    'Wird im nächsten Abschnitt beschrieben – daher hier ausgelassen.
End Function

'Kopiert ein vorhandenes Adresse-Objekt in die Maske
'für das weitere Bearbeiten.
Private Sub DatenInMaske(ByVal adr As Adresse)
    txtMatchcode.Text = adr.Matchcode
    txtVorname.Text = adr.Vorname
    txtNachname.Text = adr.Name
    txtStraße.Text = adr.Strasse
    mtbPlz.Text = adr.PLZ
    txtOrt.Text = adr.Ort
    mtbGeburtsdatum.Text = adr.Geburtsdatum.ToShortDateString
    lblWochentag.Text = adr.Geburtsdatum.ToString("dddd")
End Sub

Private Sub mtbGeburtsdatum_LostFocus(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles mtbGeburtsdatum.LostFocus
    Dim locGebDatum As Date
    If Date.TryParse(mtbGeburtsdatum.Text, locGebDatum) Then
        lblWochentag.Text = locGebDatum.ToString("dddd")
    End If
End Sub
End Class

```

Überprüfung auf richtige Benutzereingaben in Formularen

Die Qualität Ihres Arbeitsalltags fällt und wächst mit dem Aufwand, den Sie zu Zeiten der Softwareentwicklung in das Thema Eingabeprüfungen stecken. Wenn Sie professionell Softwareentwicklung betreiben, dann können Sie mir glauben, dass Sie, wenn Sie sich morgens an Ihren Entwicklungsrechner setzen, Sie kein Fax mit Inhalt dessen auf dem Schreibtisch liegen haben möchten, was Sie auch in Abbildung 27.4 sehen können.

Gerade bei der Datenerfassung, also wenn es um die Schnittstelle zwischen Ihrer Software und dem Anwender geht, können bei der Entwicklung viele Fehler passieren. Und Sie sollten nach Möglichkeit extrem kreativ werden, um Vorahnungen zu entwickeln, welche Kombinationen unglücklicher Umstände dazu führen können, dass Ihre Software jahrelang funktioniert, und plötzlich nicht mehr.

Anekdoten aus der Praxis

So wurde ich als Berater zur Analyse eines Phänomens einer Software bestellt – einer Erfassung von Mitarbeiterzeiten in Produktionsbetrieben – die jahrelang anstandslos ihren Dienst verrichtet hatte. Die Frage an den Kunden, ob in den vergangenen Tagen etwas Ungewöhnliches passiert sei, beantwortete er mit einem klaren Nein.

Nach einigem Recherchieren stellte sich allerdings heraus, dass doch etwas Außergewöhnliches vorgefallen war – wenn auch nicht im Betrieb vor Ort: Der Schwesterbetrieb dieses Berliner Unternehmens, das seinen Sitz in Magdeburg hatte, war nämlich tragischerweise direkt vom Elbe-Hochwasser 2002 betroffen.

Der Konzern bewies aber gute Führungsqualitäten in Form von schneller Reaktion und gutem Improvisationstalent, und so verlegte man kurzerhand die Produktion der wichtigsten Produkte von Magdeburg nach Berlin; die Berliner Mitarbeiter und der Betriebsrat erklärten sich sofort bereit, eine noch nie da gewesene 3. Schicht zu fahren, um den betroffenen Magdeburgern den Vortritt zu lassen.

Damit trat eine Konstellation ein, die der Software das erste Mal abverlangte, datumsübergreifende Buchungen durchführen zu müssen, und da das Tagesdatum nicht bei der Erfassung von Start- und Endzeit sondern nur als Buchungsdatum berücksichtigt wurde, sah es aus Sicht der Software so aus, als ob die Anfangszeit (20:00, 24.8.2002) zeitlich *hinter* der Endzeit (2:30, aber 25.8.2002) lag.

Die Software quittierte das mit einem Absturz bei den anschließenden Auswertungen; alle zuvor ermittelten Zeiten konnten aber glücklicherweise mit angepasstem Filter abermals aus den Zeiterfassungsterminals übernommen werden.

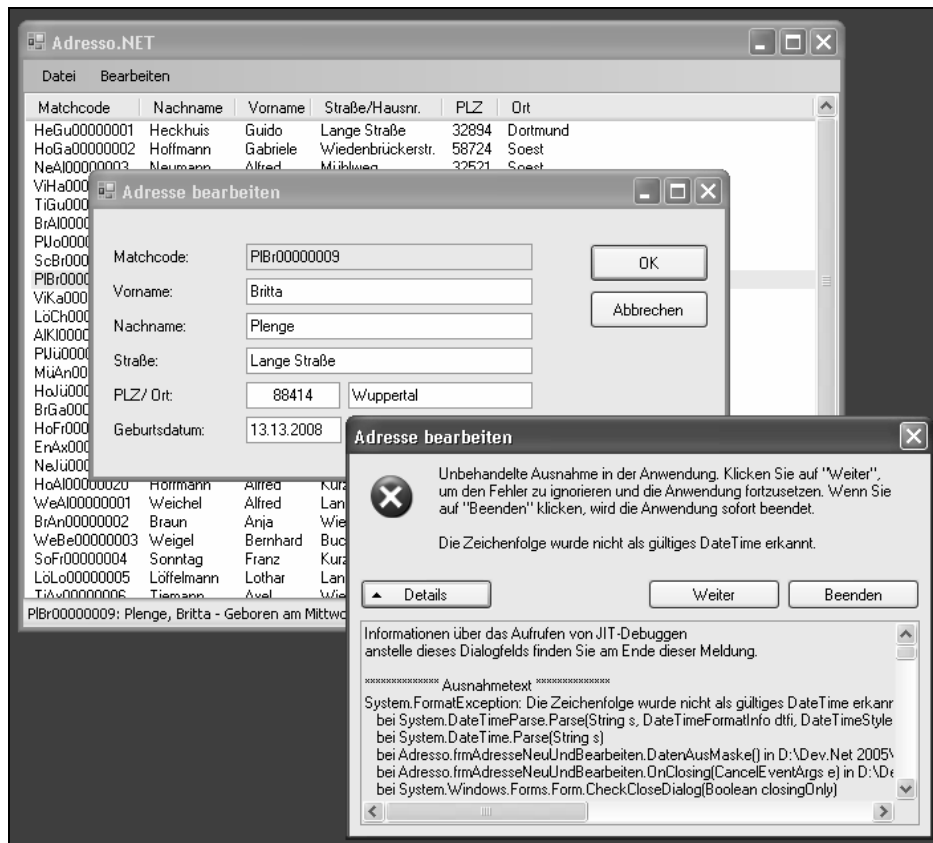


Abbildung 27.4: Wenn ein Anwender Ihrer Software einen solchen »Programmzustand« provozieren kann, liegt der Fehler leider in Ihrer Software ...

Es spielt keine Rolle, woher Ihre Software Ihre Daten bezieht – ob nun von Maschinen oder von Menschen. Sie sollten nicht zuletzt im eigenen Interesse (ruhiger schlafen, bessere Reputation) sehr großen Wert darauf legen, das größtmögliche Augenmerk auf das Sichern von Datenschnittstellen gegen Fahrlässigkeit und groben Unfug zu legen.

Was die manuelle Eingabe von Daten anbelangt, bietet das .NET-Framework eine Komponente an, die dem Anwender Falscheingaben auf eine – meiner Meinung nach – recht eigenwillige aber dennoch nicht minder plausible Weise vor Augen führt.

Abbildung 27.5 zeigt dieses Verfahren im Einsatz: Durch die ErrorProvider-Komponente lassen sich beim Auftreten von Fehlern in Eingabefeldern von Formularen diese betroffenen Felder mit einem roten Ausrufungszeichen markieren. Tritt der Fehler auf, blinken diese Ausrufungszeichen sogar für eine bestimmte (einstellbare) Weile.



Abbildung 27.5: Durch die *ErrorProvider*-Komponente können Sie Anwender auf Fehler in Eingabefeldern gezielt aufmerksam machen

Führt der Anwender anschließend mit dem Mauszeiger auf eines der Ausrufungszeichen, zeigt der *ErrorProvider* eine zuvor definierbare Fehlermeldung als Tooltip an.



Abbildung 27.6: Eine Komponente (ein zur Entwurfzeit nicht sichtbares Steuerelement) ziehen Sie bei Bedarf ebenfalls ins Formular; der Designer legt es dann im Komponentenfach ab

Um ein Formular mit einer *ErrorProvider*-Komponente auszustatten, brauchen Sie sie lediglich aus der Toolbox ins Formular zu ziehen. Der Visual Basic-Designer legt dann – so dies die erste Komponente war – ein Komponentenfach unterhalb des Formulars an und legt die Komponente dort ab.

Der *ErrorProvider* verfügt nämlich über keine eigene Benutzeroberfläche. Er erweitert einfach die Steuerelemente, die sich auf dem Container (dem Formular in diesem Fall) befinden, um die entsprechende Funktionalität.

Zum Anzeigen eines Eingabefelders neben einem Steuerelement des Formulars genügt anschließend ein einziger Methodenaufruf, etwa in dieser Form:

```
ErrorProvider.SetError(steuerelementName, "Fehlermeldungstext für den Tooltip!")
```

Wenn Sie möchten, dass alle Fehlermarkierungen wieder aufgehoben werden, verwenden Sie die folgende Methode:

```
ErrorProvider.Clear()
```

BEGLEITDATEIEN: Sie finden das Projekt zur Veranschaulichung dieses Abschnittes ebenfalls im Verzeichnis `.\\VB 2005 - Entwicklerbuch\\G - SmartClient\\Kap27\\Adresso02`.

Das Adresso-Beispiel verwendet diese Vorgehensweise, und der entsprechende Code, der von `OnFormClosing` (siehe Beispiele der vorherigen Abschnitte) aus aufgerufen wird, sieht folgendermaßen aus:

```
'Überprüft die Eingaben im Formular, und liefert
'im Erfolgsfall ein fix-und-fertiges Adresse-Objekt
'aus den Eingabefeldern zurück.
Private Function DatenAusMaske() As Adresse

    Dim locFehler As Boolean

    'Alle möglichen vorherigen Fehler zurücksetzen
    ErrProv.Clear()

    'Wenn keine Eingabe im Feld gemacht wurde,
    If String.IsNullOrEmpty(txtMatchcode.Text) Then

        'Fehlermeldung setzen.
        ErrProv.SetError(txtMatchcode, "Fehlende Eingabe!")
        locFehler = locFehler Or True

    Else

        'Der Delegat ist nur beim Neuanlegen einer Adresse vorhanden,
        'deswegen auf Vorhandensein überprüfen!
        If myMatchcodeCheckenProc IsNot Nothing Then
            'Feststellen, ob der Matchcode in der Auflistung schon existiert!
            'Das passiert in unserem Fall über einen Delegaten.
            If myMatchcodeCheckenProc.Invoke(txtMatchcode.Text) Then
                ErrProv.SetError(txtMatchcode, "Dieser Matchcode ist schon vorhanden!")
                locFehler = locFehler Or True
            End If
        End If

    End If

    'Ähnlich bei allen anderen vorgehen.
    If String.IsNullOrEmpty(txtVorname.Text) Then
        ErrProv.SetError(txtVorname, "Fehlende Eingabe!")
        locFehler = locFehler Or True
    End If

    If String.IsNullOrEmpty(txtNachname.Text) Then
        ErrProv.SetError(txtNachname, "Fehlende Eingabe!")
        locFehler = locFehler Or True
    End If
```

```

If String.IsNullOrEmpty(txtStraße.Text) Then
    ErrProv.SetError(txtStraße, "Fehlende Eingabe!")
    locFehler = locFehler Or True
End If

Dim locGebDate As Date
If Not Date.TryParse(mtbGeburtsdatum.Text, locGebDate) Then
    ErrProv.SetError(mtbGeburtsdatum, "Falsches Datumsformat!")
    locFehler = locFehler Or True
End If

If String.IsNullOrEmpty(txtOrt.Text) Then
    ErrProv.SetError(txtOrt, "Fehlende Eingabe!")
    locFehler = locFehler Or True
End If

'Fehler war vorhanden - Nothing zurückliefern
If locFehler Then Return Nothing

'Alles war OK, es gibt eine neue Adresse.
Return New Adresse(txtMatchcode.Text, txtNachname.Text, _
    txtVorname.Text, txtStraße.Text, _
    mtbPlz.Text, txtOrt.Text, locGebDate)
End Function

```

Anwendungen über die Tastatur bedienbar machen

Dieses Buch wäre schätzungsweise einen Monat später fertig geworden, könnte man die Funktionen von Microsoft Word ausschließlich über die Maus bedienen. Wenn ich selbst den typografischen Satz eines Buches vornehme, kann ich die Batterien aus meiner Funkmaus herausnehmen – ich brauche sie oft über mehrere Stunden gar nicht. Zwar ist Windows eine grafische Benutzeroberfläche, doch ist es vielen Softwareentwicklern gar nicht bewusst, wie wichtig es für einen Anwender ist, der täglich mit einer Software arbeiten muss, diese nur ausschließlich über die Maus bedienen zu können. Es kostet wahnsinnig viel Zeit, von der Tastatur, die man natürlich für Texteingaben benötigt, zur Maus umzugreifen, weil man nur so an eine bestimmte Programmfunktion herankommen kann.

Dabei bedarf es nicht viel, um Dialoge oder Menüs auch über die Tastatur bedienbar zu machen. Es gibt unter Windows definierbare Schnellzugriffstasten, deren Implementierung in die eigene Software im Handumdrehen erledigt ist.

Definition von Schnellzugriffstasten per »&«-Zeichen

Beim Erstellen von Menüs genügt ein Voranstellen des »&«-Zeichens, um dieses Menü mit dem dahinter stehenden Buchstaben über die Tastatur (in Verbindung mit **Alt**) aktivieren zu können (siehe Abbildung 27.7).

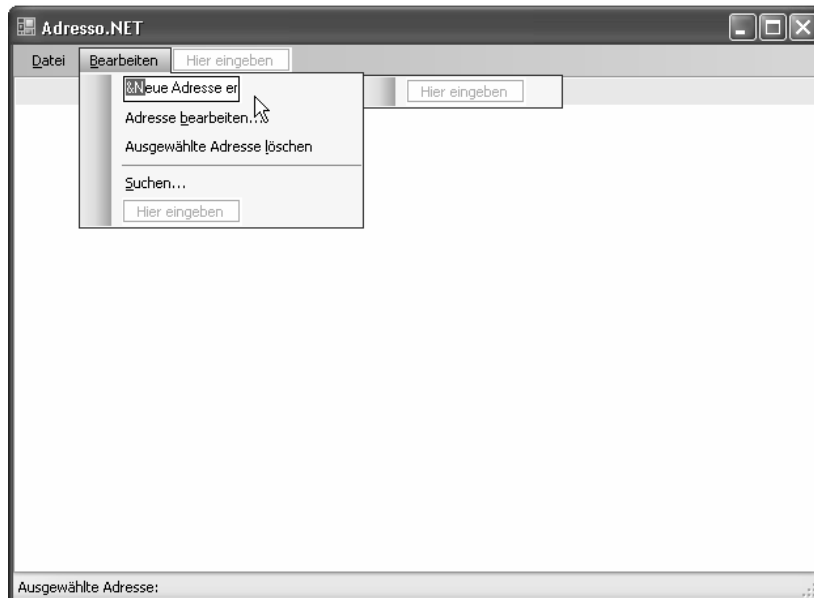


Abbildung 27.7: Mit dem Kaufmannsund (&&) definieren Sie Schnellzugriffstasten in Menüs ...

Für jeden Menüeintrag können Sie im Bedarfsfall zudem eine Funktionstaste definieren, mit der Sie den Menübefehl (und damit die dahinter stehende Funktion) direkt auslösen können. Dazu klicken Sie einfach auf den entsprechenden Menüeintrag und stellen im Eigenschaftenfenster seine `ShortcutKeyes`-Eigenschaft ein. Standardmäßig blendet dann der Menüeintrag neben dem eigentlichen Menütext einen Hinweis auf diese Tastenkombination für den Anwender ein. Sollte Ihnen diese Tastenkombinationsbeschreibung nicht passen, können Sie sie mit dem `ShortcutKeyDisplayString` nach Gutdünken anpassen.² Und falls Sie es überhaupt nicht wünschen, dass die Taste oder Tastenkombination neben dem Menütext angezeigt wird, setzen Sie die `ShowShortcutKeyes`-Eigenschaft des entsprechenden Eintrags auf `False`.

² Und in diesem Zusammenhang sei mir die bissige Bemerkung erlaubt: »Strg« steht auf deutschen Tastaturen nicht für »String«, nicht für »Strong«, nicht für »Struck« oder sonst einen Blödsinn. Es ist die vielleicht nicht ganz gelungene Abkürzung von »S t e u e r u n g«. Und falls Sie wissen, welche Geschichte sich hinter der Abkürzung »S-Abfr« auf älteren Tastaturen verbirgt (unterhalb der Druck-Taste) – für eine kurze E-Mail wäre ich dankbar ... ;-)

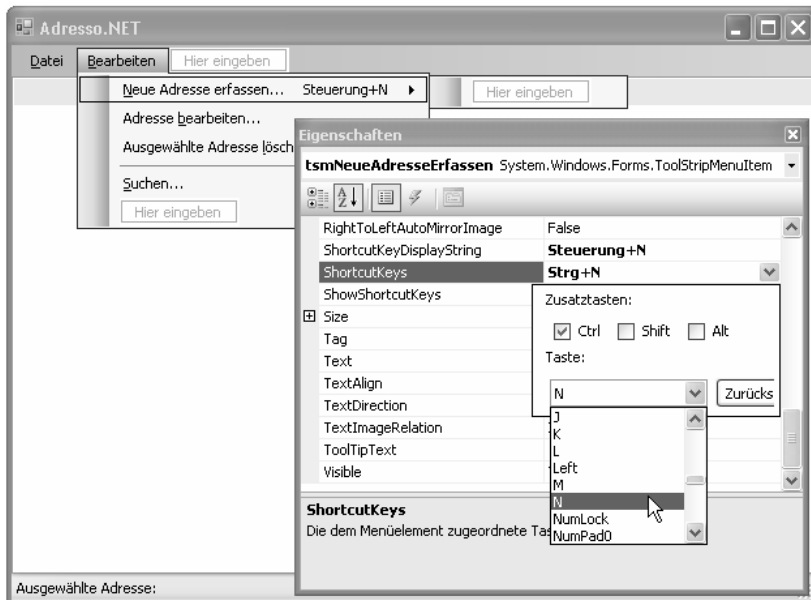


Abbildung 27.8: Neben den Schnellzugriffstasten lassen sich für jeden Menübefehl auch Funktionstasten (»Abkürzungstasten«) einrichten – einfach und schnell über das Eigenschaftenfenster ...

Bei Textfeldern in Formularen beispielsweise kommen zwei Faktoren ins Spiel, um das Eingabefeld per Tastatur zur fokussieren: Ein beschreibendes Label-Steurelement, das sich vor dem eigentlich zu aktivierendem Steuerelement befindet, und eine richtig eingestellte Aktivierungsreihenfolge.

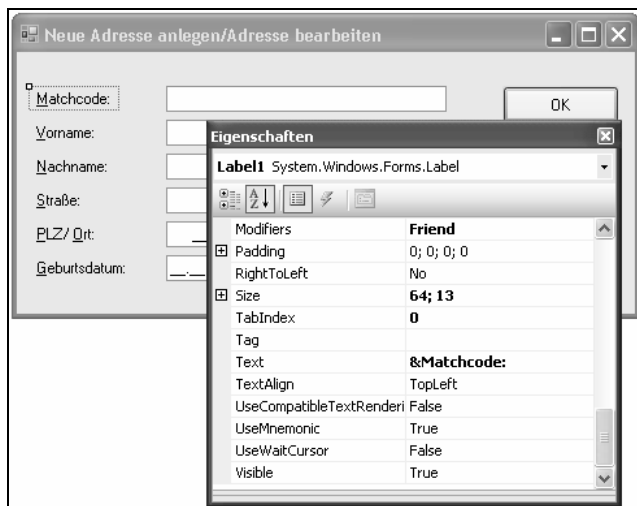


Abbildung 27.9: ... und mithilfe davor stehender Beschriftungen (*Label*) auch für Steuerelemente, die über keine selbst beschreibende Beschriftung verfügen – wie beispielsweise Textbox-Steurelemente

Da das Label vor dem Eingabesteuerelement (z.B. TextBox) nur eine beschreibende Funktion hat, selbst aber nicht aktiviert werden kann, wird beim Auslösen der Zugriffstaste das in der Aktivierungsreihenfolge hinter dem Label-Steuererelement stehende Steuerelement fokussiert.

Die Aktivierungsreihenfolge können Sie übrigens einstellen, indem Sie das Formular durch Anklicken im Designer selektieren, und anschließend aus dem Menü *Ansicht* den Menüpunkt *Aktivierungsreihenfolge* auswählen. Der Visual Studio-Designer schaltet anschließend in einen besonderen Bearbeitungsmodus, der das Bestimmen der Aktivierungsreihenfolge (englisch: *Tab Ordner* für *Tabulatorreihenfolge*) durch simples Nacheinander-Anklicken der Steuerelemente auf dem Formular erlaubt. Eine geschickte Auswahl der Aktivierungsreihenfolge ermöglicht, wie in Abbildung 27.10 zu sehen, dabei auch Konstellationen, in denen eine Beschriftung für zwei Eingabefelder gilt.



Abbildung 27.10: Mit der Funktion *Aktivierungsreihenfolge* bestimmen Sie die Zugriffsreihenfolge für das Fokussieren von Steuerelemente per **Tabulator**-Taste durch simples Nacheinander-Anklicken

Accept- und Cancel-Schaltflächen in Formularen

Viele modale Dialoge lassen sich mit den Tasten **Eingabe** und **Esc** beenden.

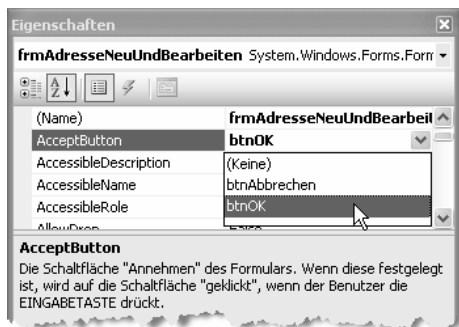


Abbildung 27.11: Die Einstellung der Schaltflächen für **Eingabe** und **Esc** nehmen Sie über *AcceptButton* und *CancelButton* am Formular vor

Für .NET-Formulare richten Sie diese Tasten für Schaltflächen (und ausschließlich für Schaltflächen) über die Formulareigenschaften *AcceptButton* (für Schaltflächen, die mit **Eingabe** auszulösen sind) und *CancelButton* (für Schaltflächen, die mit **Esc** auszulösen sind) ein.

HINWEIS: Dies ist eine für VB6-Entwickler unübliche Vorgehensweise, da sich diese Art der Formularsteuerung in VB6 über Eigenschaften der Schaltflächen und nicht über die des Formulars selbst einstellen ließen. Achten Sie also darauf, die beschriebenen Eigenschaften am Formular und nicht an den Schaltflächen im Eigenschaftenfenster zu suchen!

Im Normalfall entspricht `AcceptButton` der *OK*-Schaltfläche, `CancelButton` der *Abbrechen*-Schaltfläche eines Formulars. Sinn ergibt das Setzen dieser Eigenschaften übrigens nur in modal darzustellenden Formularen.

Über das »richtige« Schließen von Formularen

Es mag auf den ersten Blick nicht eines ganzen Abschnittes wert sein; besorgte Nachfragen im Usenet machen mich aber glauben, dass ein paar Sätze über das richtige Schließen eines Formulars nicht schaden können.

Um ein Formular programmtechnisch zu schließen, gibt es vier Möglichkeiten:

- `formInstance.Hide`
- `formInstance.Close`
- `formInstance.Dispose`
- `formInstance.DialogResult = [einWert<>DialogResult.None]` (nur in modalen Dialogen – lesen Sie dazu bitte den ► Abschnitt »Der Umgang mit modalen Formularen« ab Seite 755).

Unsichtbarmachen eines Formulars mit `Hide`

Die `Hide`-Methode macht nichts weiter, als die `Visible`-Eigenschaft eines Formulars auf `False` zu setzen. Damit gibt es die eigentliche Instanz eines Formulars zwar noch, das Formular befindet sich nur nicht mehr sichtbar auf dem Bildschirm.

Allerdings gibt es etwas zu beachten, wenn Sie Formulare modal darstellen:

Das nämlich bedeutet für das Anwenden der `Hide`-Methode (oder auch für das Setzen der `Visible`-Eigenschaft des Formulars auf `False`): Wenn die Darstellung des Formulars zuvor modal erfolgte, wird in diesem Moment die Warteschlange beendet und die Kontrolle an die aufrufende Instanz zurückgegeben, die Ressourcen des Formulars werden aber nicht sofort freigegeben.

Schließen des Formulars mit `Close`

Das Schließen des Formulars mit `Close`, »emuliert« sozusagen das Schließen des Formulars durch den Anwender. Dabei hat die das Formular einbindende Instanz (oder das Formular selbst) die Möglichkeit, den Vorgang des Schließens zu verhindern.

Entweder durch Überschreiben von `OnClosing` (soweit es das Formular selbst betrifft) oder durch Einbinden des `Closing`-Ereignisses (soweit es die das Formular einbindende Instanz betrifft) besteht die Möglichkeit, durch die `CancelEventArgs` des Ereignisses den kompletten Schließenvorgang abzubrechen:

```

Private Sub frmMain_Closing(ByVal sender As Object, ByVal e As System.ComponentModel.CancelEventArgs) _
    Handles MyBase.Closing
    'Formular soll nicht geschlossen werden
    e.Cancel = True
End Sub

```

Allerdings: Wenn nicht programmtechnisch interveniert wurde, das Formular zu schließen, ist es nicht nur unsichtbar geworden, sondern wird durch den Garbage Collector bei der nächsten Gelegenheit entsorgt. Die Close-Methode entspricht also quasi einem intervenierbaren Dispose (siehe nächster Abschnitt).

Was passiert bei Form.Close intern:

Für die Puristen unter Ihnen hier eine Ereignisabfolge, die beschreibt, in welcher Reihenfolge Dinge beim Schließen eines Formulars mit Close passieren (lesen Sie im Bedarfsfalls zunächst die entsprechenden Abschnitte im nachfolgenden Kapitel, um mehr Grundsätzliches über die interne Verwaltung und Verarbeitung von Nachrichten bei Windows-Anwendungen zu erfahren).

- Close wird aufgerufen, das Formular sendet windows-intern die Nachricht WM_Close mit seinem eigenen Window Handle an die Nachrichtenwarteschlange.
- Wenn die Nachrichtenschlange verarbeitet wird, schickt diese die Nachricht weiter an WndProc des Formulars.
- WndProc ruft die Formular-interne Methode WMClose auf.
- WMClose ruft die Dispose-Methode des Formulars auf, nachdem OnClosing und OnClose aufgerufen wurden, die ihrerseits die entsprechenden Ereignisse ausgelöst haben. Dispose wird aber nur dann aufgerufen, wenn OnClosing den Schließvorgang nicht abgebrochen hat.

TIPP: Wenn sich Ihr Formular selber schließen soll, es dies aber in einer Methode macht, die einen Rückgabewert an die aufrufende Instanz zurückliefern muss, der aus einem Member der Formularklasse hervorgeht, schließen Sie es mit Close, und wenden Sie dann Return an. Nur so ist gewährleistet, dass es den zurückzugebenden Member noch gibt, wenn Return ausgeführt wird. Dispose würde das Formular sofort zur Entsorgung freigeben; Sie könnten nicht sicher sein, dass es den zurückzugebenden Wert beim Return noch gibt.

Da diese Anforderungen in der Regel nur für modale Dialoge bestehen, verwenden Sie noch besser das Programmiermuster, über das der ► Abschnitt »Der Umgang mit modalen Formularen« ab Seite 755 genau Auskunft gibt.

Entsorgen des Formulars mit Dispose

Dispose macht mit einem Formular, was es auch mit jedem anderen Objekt macht: es zur Entsorgung durch den Garbage Collector freigeben. Das bedeutet: Dispose lässt ein Formular unaufhaltsam und für immer verschwinden. Das Closing-Ereignis wird *nicht* aufgerufen, und weder die das Formular einbindende Instanz noch das Formular selbst haben die Möglichkeit, etwas dagegen zu tun. Das gilt natürlich auch für alle Member des Formulars. Lesen Sie deswegen bitte auch die Ausführungen des vorherigen Abschnittes.

Grundsätzliches zum Darstellen von Daten aus Auflistungsklassen in Steuerelementen

Fast alle denkbaren Anwendungen verlangen es, dass Datenauflistungen in irgendeiner Form visualisiert werden, und das geschieht in der Regel in Form von Listen. Das `ListBox`-Steuerelement eignet sich dazu nur bedingt, da es im Prinzip nur eine Eigenschaft eines Datenelements darstellen kann. Wenn es darum geht, umfangreiche Mengen von Daten in SmartClient-Anwendungen übersichtlich darzustellen, bieten sich dazu zwei Steuerelemente an:

- Das `ListView`-Steuerelement.
- Das `DataGridView`-Steuerelement.

Das `ListView`-Steuerelement eignet sich hervorragend zur reinen Textdarstellung von tabellarischen Daten, wenn Sie seine detaillierte Listendarstellung verwenden. Sie kennen es aus dem Windows-Explorer und der Dateianzeige: Es ist ein unglaublich mächtiges Steuerelement, das die verschiedensten Darstellungsformen kennt. So können Sie sich auch im Windows-Explorer für eine Miniaturansicht von Daten, für die gekachelte Darstellung, die symbolische Darstellung, die einfache Listendarstellung in die detaillierte Listendarstellung entscheiden. In den anschließenden Beispielen werden wir uns auf diesen letzten Darstellungsstil beschränken.

Das `DataGridView`-Steuerelement ist ein komplett neues Steuerelement im .NET-Framework 2.0 und dann interessant, wenn nicht nur reine Textdaten, sondern komplexere Datenvisualisierungen in verschiedenen Spalten umgesetzt werden müssen. Im Gegensatz zum `ListView`-Steuerelement erlaubt es auch das Bearbeiten von Daten direkt in Tabellenform. Dafür ist der Programmieraufwand zur Verwaltung der Daten im `DataGridView`-Steuerelement auch größer, um Daten abzubilden und dem Anwender die Möglichkeit zu geben, diese zu bearbeiten.

Die folgenden Abschnitte beschreiben die grundsätzliche Vorgehensweise beim Abbilden von Auflistungsklassen in diesen Steuerelementen.

Darstellen von Daten aus Auflistungen im `ListView`-Steuerelement

Wenn Sie sich für die tabellarische Darstellung der Daten von Auflistungsklassen entscheiden, ist die detaillierte Listendarstellung des `ListView`-Steuerelements genau das Richtige. Um eine Instanz des `ListView`-Steuerelements für diese Form der Darstellung vorzubereiten, verwenden Sie die folgenden Eigenschaften (`lvwAdressen` sei dabei im Folgenden der Name eines `ListView`-Steuerelements im Formular):

<code>lvwAdressen.View = Details</code>	'Einstellung des Steuerelements auf detaillierte Listendarstellung
<code>lvwAdressen.FullRowSelect = True</code>	'Selektierung der gesamten Zeile, nicht nur des ersten Elements
<code>lvwAdressen.HideSelection = False</code>	'Beim Verlieren des Fokus bleibt die Selektierung erhalten
<code>lvwAdressen.GridLines = True</code>	'Zwischen den Zeilen werden kleine Abtrennungslinien eingezeichnet

Diese Einstellungen können Sie natürlich auch mit dem Eigenschaftfenster zur Entwurfzeit einstellen, so, wie es in den vergangenen und zukünftigen Beispielen dieses Kapitels auch der Fall ist.

Spaltenköpfe

Die Details-Ansicht des `ListView`-Steuerelements zeichnet sich dadurch aus, dass Daten in mehreren Spalten pro Zeile angeordnet werden können. Die Spalten selbst werden durch die `ColumnHeaderCollection`-Auflistung³ des Steuerelements eingerichtet und gesteuert. Um verschiedene Spalten dem Steuerelement hinzuzufügen, verfahren Sie beispielsweise wie folgt:

```
With Me.ListViewAdressen
  With .Columns
    'Alle Spaltenköpfe löschen.
    .Clear()
    'Spaltenüberschriften einrichten.
    .Add("Spalte1", -2, HorizontalAlignment.Left)      'Linksbündig Darstellung in der Spalte
    .Add("Spalte2", -2, HorizontalAlignment.Right)     'Rechtsbündig Darstellung
    .Add("Spalte3", -2, HorizontalAlignment.Center)    'Zentrierte Darstellung
    .Add("Spalte4", -2)                               'Ohne Angabe: Standard ist linksbündig
  End With
End With
```

Mit der `Columns`-Eigenschaft erhalten Sie Zugriff auf die `ColumnHeaderCollection`-Auflistung, durch die Sie dann die einzelnen Spaltenköpfe wie bei einer »normalen« Auflistung mit der `Add`-Methode hinzufügen können. Die einfachste Methode, das zu erreichen, wird hier demonstriert: Als ersten Parameter bestimmen Sie den Text eines Spaltenkopfes, als zweiten seine Breite. Wenn Sie möchten, dass die Spaltenbreite automatisch an die Breite des Textes angepasst wird, übergeben Sie als Spaltenbreite den Wert `-2`. Wichtig dabei ist zu wissen, dass Sie die Spaltenbreiten abermals auf `-2` setzen müssen (obwohl sich ja eigentlich dieser Wert bereits in der Eigenschaft befindet), nachdem Sie die Liste mit Daten gefüllt haben. Dann wird die Layout-Logik abermals für jede Spalte aufgerufen; dieses Mal werden die Breiten der Spalten dann jedoch aufgrund der Breite der Spaltenköpfe *und* aufgrund der Breite der einzelnen Listeneinträge neu berechnet und visuell angepasst.

Schließlich können Sie einen dritten Parameter übergeben, der die Ausrichtung der Texte in den Spalten bestimmt.

HINWEIS: Beachten Sie, dass die erste Spalte Texte grundsätzlich nur linksbündig darstellen kann, egal, welchen Parameter Sie hier angeben. Wenn Sie keinen Parameter übergeben, wird die linksbündige Darstellung der Texte in der entsprechenden Spalte angenommen.

Hinzufügen von Daten

Jede Zeile eines `ListView`-Steuerelements wird durch eine `ListViewItem`-Instanz repräsentiert. Ein `ListViewItem`-Objekt enthält also die Texte jeder Spalte einer Zeile, die im `ListView`-Steuerelement dargestellt werden sollen. Den Text, der in der ersten Spalte dargestellt wird, bestimmt die `Text`-Eigenschaft eines `ListViewItem`-Objekts.

³ Kleine Anmerkung am Rande: Die Benennung einer Auflistung mit dem endenden Wortteil »Collection« widerspricht übrigens Microsofts Richtlinien zur Namensvergabe von Auflistungen – Microsoft verstößt bei der Namensgebung von `ColumnHeaderCollection` also gegen seine eigenen Richtlinien. Am besten fahren Sie mit Begriffen, bei denen Sie die Datenklasse im Singular benennen (Adresse), eine entsprechende Auflistung, die diese Daten als Menge verwaltet, im Plural (Adressen).

Diesen Text können Sie direkt beim Instanzieren einer Instanz dieses Objektes angeben. Die Texte aller weiteren Spalten einer Zeile werden durch die SubItems-Elemente des ListViewItem-Objekts festgelegt. Da es sich bei SubItems wieder um eine Auflistung handelt, können Sie die Texte der einzelnen Spalten ebenfalls mit deren Add-Methode einrichten, etwa wie im Folgenden zu sehen:

```
Dim locLvwItem As New ListViewItem("Text, Spalte 1")

'Die Untereinträge setzen
With locLvwItem.SubItems
    .Add("Text, Spalte 2")
    .Add("Text, Spalte 3")
    .Add("Text, Spalte 4")
End With
```

Wenn Sie eine ListViewItem-Instanz auf diese Weise aufbereitet haben, stellen Sie die komplette Zeile im ListView-Steuerelement dar, indem Sie diese Instanz der Items-Auflistung des ListView-Steuerelements hinzufügen, etwa mit:

```
'Zur ListView hinzufügen
lvwAdressen.Items.Add(locLvwItem)
```

Beschleunigen des Hinzufügens von Elementen

Das Hinzufügen der Elemente auf die so beschriebene Weise erfolgt sichtbar, und das heißt: Sobald Sie ein Element hinzugefügt haben, wird der komplette Inhalt des ListView-Steuerelements neu gezeichnet. Bei hunderten von Elementen kann das sehr lange dauern – unnötigerweise.

Sie können mit zwei Methodenaufrufen – BeginUpdate sowie EndUpdate – festlegen, dass das Neuzeichnen der Elemente für die Dauer ihres Hinzufügens zur Liste ausgesetzt wird, und das funktioniert folgendermaßen:

```
'Unterdrückt Neuzeichnen-Ereignisse bis zum nächsten EndUpdate;
'dadurch geht der Aufbau der Elemente schneller und "wackelt" nicht.
Me.lvwAdressen.BeginUpdate()
'Hier fügen Sie alle Elemente der Liste hinzu:

'Aufbau der ListView ist beendet.
Me.lvwAdressen.EndUpdate()
```

Automatisches Anpassen der Spaltenbreiten nach dem Hinzufügen aller Elemente

Nachdem alle Elemente der Liste hinzugefügt wurden, können Sie mit einem kleinen Trick dafür sorgen, dass sich die Spaltenbreiten automatisch an die Breiten der Texte bzw. der Spaltenköpfe anpassen (die längsten Texte in den Köpfen und Datenzeilen einer jeden Spalte werden als Maß verwendet). Dazu setzen Sie die Breiten aller Spalten einfach komplett nochmals auf den Wert -2. In Form von Code sieht das so aus:

```
'So werden die Spaltenbreiten optimal angepasst.
For Each locCol As ColumnHeader In Me.lvwAdressen.Columns
    locCol.Width = -2
Next
```

Sie sollten diese Aktion auch zwischen `BeginUpdate` und `EndUpdate` durchführen, damit ein Neuzeichnen des gesamten Steuerelements erst anschließend stattfindet.

Zuordnen von `ListViewItem`s und Objekten, die sie repräsentieren

Anders als bei der einfachen `ListBox`, der Sie eine komplette Objektreferenz als Listeneintrag selbst übergeben können, deren Textdarstellung anschließend über die `ToString`-Funktion des Objektes geregelt wird, und bei der eine selektierte Zeile auch gleichzeitig einem selektierten Objekt entspricht, gibt es beim `ListView`-Steuerelement keine automatische Zuordnung zwischen einem Eintrag in der Liste und einem Objekt, das diese Zeile visuell darstellen soll. Die Frage lautet also: Wenn der Anwender eine Zeile angeklickt hat, wie kann ich dann herausfinden, welchem Objekt sie ursprünglich entsprach?

Um diesem Problem zu entgehen, verfügt jedes `ListViewItem`-Objekt, das eine Zeile der `ListView` darstellt, über eine `Tag`-Eigenschaft (sprich: »Tähg«, etwa: Kennzeichnung, Markierung). Diese Eigenschaft kann beliebige Objekte, sprich: »alles« aufnehmen. Der Trick besteht also darin, aus einem Objekt Texte für die Darstellung über eine `ListView`-Instanz zu generieren und die Objektreferenz zusätzlich noch in der `Tag`-Eigenschaft eines `ListViewItem` zu speichern.

Wenn Sie später eine bestimmte Zeile der `ListView` als `ListViewItem` über die `Items`-Auflistung abrufen, steht Ihnen so auch das Ursprungsobjekt, aus dem die Texte für die `ListViewItem`-Instanz entstanden sind, wieder zur Verfügung.

Eine Zuordnung eines beliebigen Objekts sieht in etwa wie folgt aus:

```
'Zum Wiederfinden: Referenz in Tag speichern
locLvwItem.Tag = locElement
```

Wenn Sie später wieder an das Ausgangsobjekt herankommen wollen, brauchen Sie es nur aus der `Tag`-Eigenschaft wieder auszulesen (und im Bedarfsfall in den Ursprungstyp zurückzucastern). Das folgende Codeschnipsel macht genau das (Voraussetzung dafür ist natürlich, dass mindestens ein Element in der `ListView` selektiert wurde):

```
'Das ursprüngliche Adresse-Objekt aus der Liste holen.
locElement = DirectCast(lvwAdressen.SelectedItems(0).Tag, ElementType)
```

Übrigens: Auch bei komplexen Objekten bedeutet diese Vorgehensweise natürlich kein Aasen mit Arbeitsspeicher, weil, wie Sie im Klassenteil dieses Buches erfahren konnten, natürlich nur eine *Referenz* auf das eigentliche Objekt gespeichert wird. Da es die `Tag`-Eigenschaft sowieso gibt (und der entsprechende Speicher für eine Referenz sowieso reserviert wird, auch wenn sie standardmäßig auf `Nothing` »zeigt«), kostet das Speichern jedes Objektes in der `Tag`-Eigenschaft jedes `ListViewItem` der Liste kein einziges Byte an zusätzlichen Speicher.

Feststellen, dass ein Element der Liste selektiert wurde

Wenn Sie eine `ListView` verwenden, wird der Anwender irgendwann einmal ein Element dieser Liste auswählen, um es beispielsweise bearbeiten zu können. In diesem Fall muss Ihre Anwendung natürlich nicht nur wissen, welches Element angeklickt wurde, sondern unter Umständen auch, wann dieses Ereignis auftrat, um zum Beispiel eine bestimmte Statusinformation (etwa in einer Statuszeile des Fensters) zu aktualisieren.

Mit dem `SelectedIndexChanged`-Ereignis informiert Sie ein `ListView`-Steuerelement, dass sich die Elementselektierungen in der Liste geändert haben. Das Ereignis wird immer dann ausgelöst, wenn ...

- ... ein Element selektiert wurde und zuvor keines selektiert war
- ... kein Element mehr selektiert ist, vorher aber mindestens ein Element selektiert war
- ... ein weiteres Element selektiert wurde, wenn das `ListView`-Steuerelement zuvor mithilfe der `MultiSelect`-Eigenschaft so eingestellt wurde (`True`), dass mehrere Elemente in der Liste selektiert werden konnten.

HINWEIS: Wenn Sie sehr zeitintensive Benutzeroberflächenaktualisierungen durchführen müssen, sobald sich die Selektierung in der Liste ändert, beachten Sie dabei, dass dieses Ereignis beim Wechsel einer Elementselektierung zweimal ausgelöst wird: Einmal für die Deselektierung des ersten Elements und ein anderes Mal für die Selektierung des neuen Elements.

Mit dem `ItemSelectionChanged`-Ereignis (neu im Framework 2.0) werden Sie darüber hinaus informiert, wenn sich der Auswahlzustand eines Elementes ändert.

Die Feststellung, welche Elemente im `ListView`-Steuerelement selektiert sind, kann mithilfe zweier Eigenschaften erfolgen:

- Die `SelectedItems`-Eigenschaft enthält eine Auflistung aller Elemente, die zurzeit im `ListView`-Steuerelement selektiert sind.
- Die `SelectedIndices`-Eigenschaft enthält eine Auflistung aller Indizes der Elemente, die zurzeit im `ListView`-Steuerelement selektiert sind.

Der folgende Code implementiert eine Routine zum Löschen aller selektierten Elemente im `ListView`-Steuerelement aus einer korrelierenden Auflistung, die die Ursprungsobjekte zur Darstellung im `ListView`-Steuerelement bereithält.

```
If lvwAdressen.SelectedItems IsNot Nothing AndAlso lvwAdressen.SelectedItems.Count > 0 Then
    For Each lvwItem As ListViewItem In lvwAdressen.SelectedItems
        myAdressen.Remove(DirectCast(lvwItem.Tag, Adresse))
    Next
    'Hier wird die Liste neu aufgebaut:
    ElementeDarstellen()
End If
End If
```

Selektieren von Elementen in einem ListView-Steuerelement

Wichtig für Anwendungen ist es nicht nur, herausfinden zu können, welche Elemente eines ListView-Steuerelements selektiert sind. Hier und da müssen auch Elemente programmtechnisch selektiert werden.

HINWEIS: Hierfür können Sie die `SelectedItems`-Eigenschaft *nicht* verwenden, da sie nur die selektierten Elemente eines ListView-Steuerelements *widerspiegelt*. Wenn Elemente eines ListView-Steuerelements selektiert werden sollen, erreichen Sie das ausschließlich über die `Selected`-Eigenschaft eines `ListViewItem`-Elements der Liste.

Der Code, um Elemente eines ListView-Steuerelements zu selektieren, in deren `Tag`-Eigenschaft sich Objekte einer korrelierenden Auflistung befinden, lautet folgendermaßen:

```
'Alle zu selektierenden Elemente durchlaufen, und...
For Each locAdresse As Adresse In zuSelektierendeAdressenObjekte

    'jeweils alle ListView-Elemente durchsuchen und überprüfen, ob ...
    For Each locLvwItem As ListViewItem In Me.lvwAdressen.Items

        '... die Tag-Referenz der Referenz des gesuchten Objekts entspricht.
        If locLvwItem.Tag Is locAdresse Then

            'Gefunden! ListView-Element markieren,
            locLvwItem.Selected = True

            'und wir müssen in der ListView
            'nicht weitersuchen!
            Exit For
        End If
    Next
Next
```

TIPP: Wenn Sie möchten, dass ein bestimmtes Element nicht nur selektiert wird, sondern es sich darüber hinaus auch auf jeden Fall im sichtbaren Bereich des ListView-Steuerelements befinden soll, verwenden Sie die `EnsureVisible`-Methode des entsprechenden `ListViewItem`-Objekts.

Verwalten von Daten aus Auflistungen mit dem DataGridView-Steuerelement

Wenn es um die reine Darstellung von Daten einer Auflistung geht, verrichtet das ListView-Steuerelement sicherlich gute Dienste, zumal es bei der Anzeige auch vieler Daten eine gute Performance an den Tag legt.

Flexibler für die Darstellung von Daten, wenn auch nicht so performant, ist ein Steuerelement, das neu mit dem Framework 2.0 eingeführt wurde: das DataGridView-Steuerelement. Der Name dieses Steuerelements ist vielleicht ein wenig irreführend, weil es durch die Silbe »View« (etwa: *Ansicht, Darstellung, Ausblick*) im Namen die Vorstellung erweckt, es könne Daten lediglich *anzeigen*. Doch das ist überhaupt nicht der Fall.

Ganz im Gegenteil: Mit dem DataGridView-Steuerelement legt Ihnen Microsoft eine visuelle Komponente in die Hände, die an Mächtigkeit in Sachen Datendarstellung und Datenerfassung eigentlich nicht mehr zu übertreffen ist.

Doch es ist wie bei allen Dingen des Lebens: Wo viel Licht ist, ist auch viel Schatten. Um bei dieser Analogie zu bleiben: Das DataGridView-Steuerelement ist ein 10.000-Watt-Suchscheinwerfer; deswegen gibt es auch eine Menge Falltüren, die es clever zu umschiffen gilt, wenn Sie für den Komfort des Anwenders das Letzte aus diesem Steuerelement herausholen wollen.

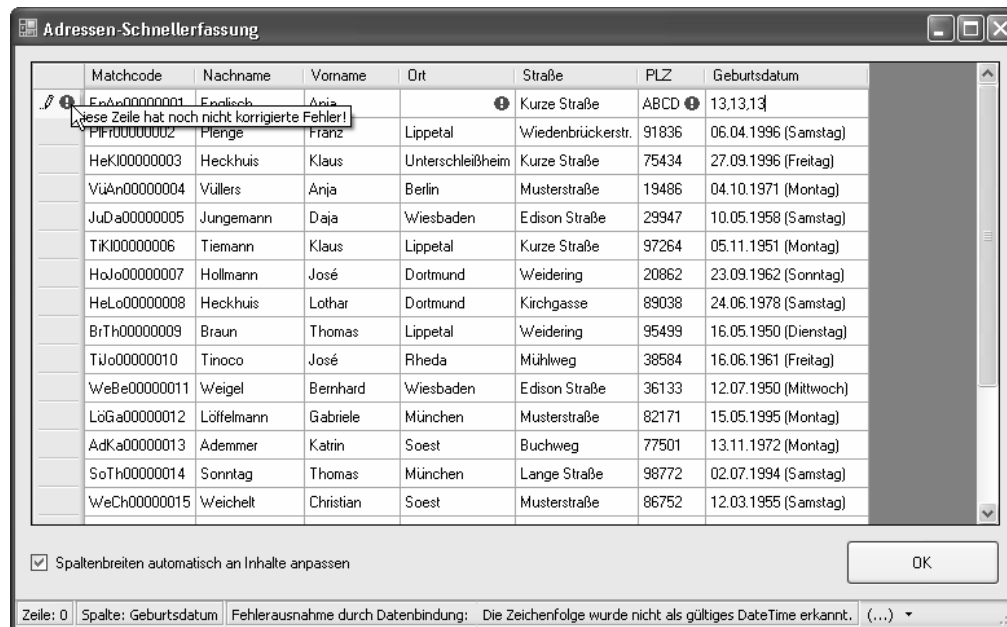


Abbildung 27.12: So komfortabel sollte eine Datenerfassungsanwendung mindestens sein, die Sie den Anwendern Ihrer Software über das *DataGridView*-Steuerelement zur Verfügung stellen

Um es vorweg zu sagen: Dem DataGridView-Steuerelement könnte man locker nicht nur ein eigenes Kapitel, sondern fast schon ein kleines Buch widmen, und wenn ich persönlich auch dieses Thema extrem spannend finde, so muss ich mich aus Platzgründen dennoch auf Wesentliches zu diesem Thema beschränken. Neben den grundlegendsten Dingen, die Sie für den Start benötigen, habe ich deswegen versucht, möglichst viele der Themen herauszuarbeiten, die Sie nicht ohnehin aus der Online-Hilfe erfahren könnten.

Und das impliziert meinen ersten Vorschlag: Die Lektüre der Online-Hilfe zum Thema DataGridView enthält wirklich umfangreiches und gar nicht so trocken geschriebenes Material. Wenn Sie sich viel mit Datenbindung und Datenaufbereitung in Ihren Anwendungen beschäftigen müssen, sollten Sie sich wirklich einen halben Tag Zeit nehmen und die Online-Hilfe von Visual Basic zu diesem Thema gründlich studieren.

Um Daten in einem DataGridView-Steuerelement anzeigen oder bearbeiten zu können, stehen Ihnen folgende Vorgehensweisen zur Verfügung:

- Sie sorgen komplett per Programmcode für das Einrichten der verschiedenen Spalten, für die Datentypen, die in den Zellen der einzelnen Spalten verarbeitet werden, für das Anzeigen der Daten in den Zellen und für die Aufbereitung und spätere Übernahme in den Datenklassen ihrer Anwendung.
- Sie binden das DataGridView-Steuerelement an eine Datenquelle, sorgen sich nicht um Tabelleneinrichtung, Datendarstellung und Datenüberprüfung und programmieren gar nicht.
- Sie finden den goldenen Mittelweg, verwenden die Datenbindung, aber greifen bei entscheidenden Prozessen per Programmcode in das Geschehen ein. Die folgenden Abschnitte und darin enthaltenen Beispiele gehen diesen Mittelweg, da er den meiner Meinung nach besten Kosten-/Nutzen-Faktor darstellt.

Über Datenbindung, Bindungsquellen, die BindungsSource-Komponente und warum Currency nicht unbedingt Währung heißt

Ganz simpel ausgedrückt: Bei der Datenbindung von Framework-Komponenten wird die Eigenschaft einer konsumierenden Datenquelle mit der Eigenschaft einer Daten vorgebenden Komponente verknüpft. Ändert sich der Wert der Eigenschaft der letzteren, ändert sich gleichzeitig der Wert der ersten – im Bedarfsfall auch umgekehrt.

Bei größeren Bindungsszenarien, die nicht nur auf Eigenschaften (also Datenfeldebene) beschränkt sind, bedeutet das beispielsweise für ein DataGridView-Steuerelement: Man kann auch eine ganze DataGridView-Instanz an eine Datenquelle binden. Die Datenquelle liefert Datenlisten, die DataGridView zeigt sie an, erlaubt das Bearbeiten, und die Änderungen laufen im Bedarfsfall wieder zurück in die Datenquelle. Eine Datenquelle kann dabei eine Komponente einer Datenbank sein. Beispielsweise eine Access-Datentabelle. Oder besser: Eine Tabelle oder ein Resultset⁴ einer SQL Server-Datenbank. Oder, und das ist das Tolle, auch etwas, was man im Microsoft-Jargon als *Business Object* (Geschäftsobjekt) bezeichnet, und dabei handelt es sich in der Regel um eine Auflistungsklasse, wie Sie sie beispielsweise in unserer ständig wachsenden Adresso-Anwendung kennen gelernt haben. Nicht nur einzelne Eigenschaften werden also dabei gebunden, sondern mehrere Objekte, die ihrerseits über mehrere Eigenschaften verfügen. Eine Tabellenzeile stellt also genau ein Objekt einer Auflistung dar (oder, im Falle von Datenbanken, einen Datensatz). Eine Tabellenzelle entspricht einer Eigenschaft eines Objektes der Auflistung (oder dem Inhalt eines Datenfeldes eines Datensatzes im Fall von Datenbanken). Die ganze DataGridView-Tabelle entspricht der gesamten Auflistung (oder – bei Datenbanken – einem kompletten Resultset). Beim Binden von Auflistungen spricht man übrigens nur von Datenbindungen, beim Binden von Datenbanktabellen von *komplexen* Datenbindungen.

Doch egal, welches Objekt an welches andere Objekt gebunden wird, sie benötigen immer eine vermittelnde Instanz. Im Fall von simplen Eigenschaftsverknüpfungen übernimmt diese Aufgabe das so genannte PropertyManager-Objekt; im Falle von Listen das CurrencyManager-Objekt.

⁴ ... was soviel wie *Abfrageergebnissatz* bedeutet und einer wie auch immer garteten Datentabelle entspricht.

HINWEIS: Lassen Sie sich dabei nicht von dem englischen Begriff *Currency* ins Bockshorn jagen, für den Sie vielleicht nur die deutsche Übersetzung *Währung* parat haben. *Currency* bedeutet nämlich auch *Zeitnähe* (von *current*, etwa: *aktuell*); ein *CurrencyManager* ist also eine überwachende Instanz, die dafür sorgt, dass Dinge *zeitnah* passieren – im .NET-Framework die zeitnahe Umsetzung der Datenbindung.

Bis zur Version 1.1 verlief das Binden einer Komponente an eine Datenquelle, die mehrere Listenelemente anbot, dubios und irgendwie im Hintergrund, ohne dass man so recht wusste, was genau passiert. Was wirklich passierte, war: Beim Zuweisen einer Datenquelle an eine Komponente – beispielsweise für eine Listendarstellung dieser Elemente – wurde implizit ein *CurrencyManager*-Objekt erstellt. Dieses *CurrencyManager*-Objekt sorgte dann dafür, dass – sehr vereinfacht ausgedrückt – beim Editieren einer Zeile auch das korrelierende Objekt in der Auflistung »getroffen« wurde.

Im .NET 2.0-Framework ist dieser Vorgang nicht nur offensichtlicher geworden – er wurde auch um Designer-Unterstützung ergänzt: Sie können mithilfe des Designers eine Datenquelle erstellen, oder besser: dem Designer mitteilen, welches Objekt (Datenbank, Auflistungsklasse) eine Datenquelle *sein soll*. Und dann können Sie interaktiv diese Datenquelle an eine Komponente wie beispielsweise eine *DataGridView*-Instanz binden. Der Designer erstellt dabei eine so genannte *BindingSource*-Komponente (auf deutsch etwa *Bindungsquelle*, eine Art *CurrencyManager XXL*), und legt diese Komponente im Komponentenfach ab.

Diese *BindingSource*-Komponente ist also im .NET-Framework 2.0 das Bindeglied zwischen der die Daten anbietenden und der die Daten konsumierenden Komponente. Damit das reibungslos funktioniert, stellt sie (ganz ähnlich wie früher *CurrencyManager*) Methoden wie *MoveNext* (*zum Nächsten bewegen*), *MovePrevious* (*zum Vorherigen bewegen*), *AddNew* (*Neuen anlegen*) und Ähnliches bereit. Und wozu? Ganz einfach:

Wenn Sie später beim Bearbeiten von Daten in Ihrer *DataGridView*-Instanz mit dem Cursor eine Zeile nach unten fahren, dann ruft die *DataGridView* die *MoveNext*-Methode der *BindingSource* auf. Die *BindingSource* sorgt dann dafür, dass ein interner Zeiger auf die einzelnen Elemente der Datenquelle ebenfalls weitergerückt wird. Die *DataGridView* kann sich also das nächste (*next*) Objekt aus der Datenquelle holen – im Falle einer Auflistung eben das nächste Objekt in der Liste.

Fahren Sie mit dem Cursor in der Tabelle eine Zeile nach oben, ruft die *DataGridView*-Instanz *MovePrevious* der *BindingSource* auf, und sie bekommt, um beim Beispiel zu bleiben, wieder über den Vermittler *BindingSource*, das vorherige (*previous*) Objekt der Auflistung.

Dieser Fall geht auch umgekehrt. Wenn Sie eine Objektauflistung als Datenquelle über eine *BindingSource*-Instanz an eine *DataGridView*-Instanz gebunden haben, können Sie auch selber *MoveNext* der *BindingSource* aufrufen, um einerseits den Zeiger zum nächsten Objekt der Datenauflistung, andererseits den Cursor der *DataGridView*-Instanz eine Zeile nach unten zu verschieben. Und ein Aufruf von *MovePrevious* macht das Gleiche bei beiden involvierten Komponenten, nur in die andere Richtung. Übrigens: Eine weitere im .NET-Framework 2.0 neue Komponente macht von dieser Methode regen Gebrauch, und sie nennt sich *BindingNavigator*. ► Kapitel 32 stellt sie im Rahmen von ADO.NET-Datenbindungen am einfachen Beispiel vor.

HINWEIS: Das ListView-Steuerelement sieht das Binden von Auflistungsdaten über eine BindingSource-Komponente übrigens nicht vor.

Wie das interaktive Erstellen einer solchen Korrelation in der Praxis aussieht, und wie einfach das vonstatten geht, zeigt der folgende Abschnitt.

Erstellen einer gebundenen DataGridView mit dem Visual Basic-Designer

Für das folgende Beispiel können Sie zwei Beispielprojekte der Begleitdateien verwenden – ein fix und fertiges und eines, mit dem Sie die folgenden Schritte nachvollziehen können.

BEGLEITDATEIEN: Sie finden letzteres im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\Adresso03 - zum Experimentieren\`. Das fertige Beispiel befindet sich im Verzeichnis `\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\Adresso03\`.

- Öffnen Sie das Projekt und rufen Sie das schon teilweise vorbereitete Formular `frmSchnellerfassung.vb` im Designer auf. Ihnen bietet sich anschließend ein Bild, wie Sie es in etwa auch in Abbildung 27.13 sehen können.

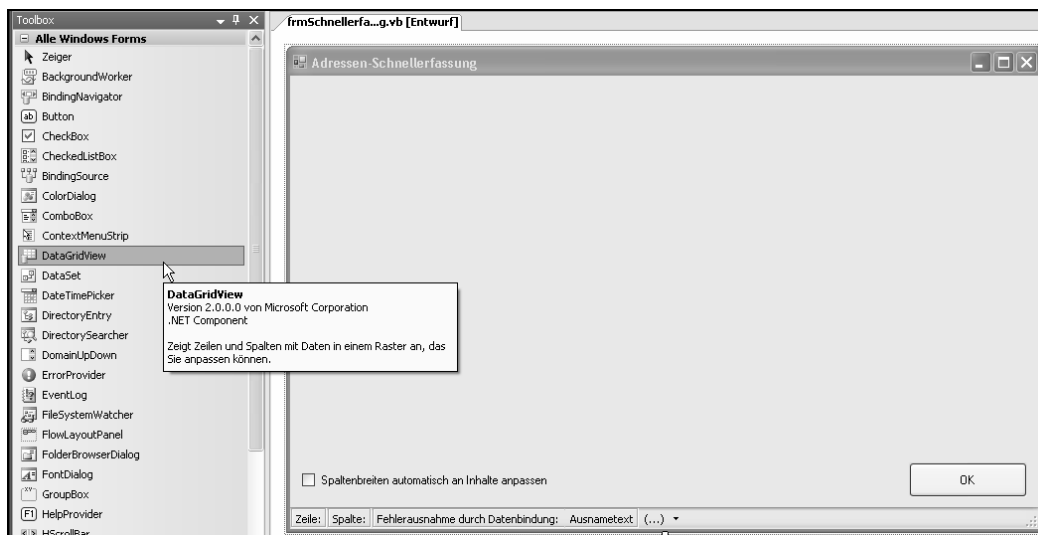


Abbildung 27.13: Suchen Sie das `DataGridview`-Steuerelement in der Toolbox, und ziehen Sie es ins Formular

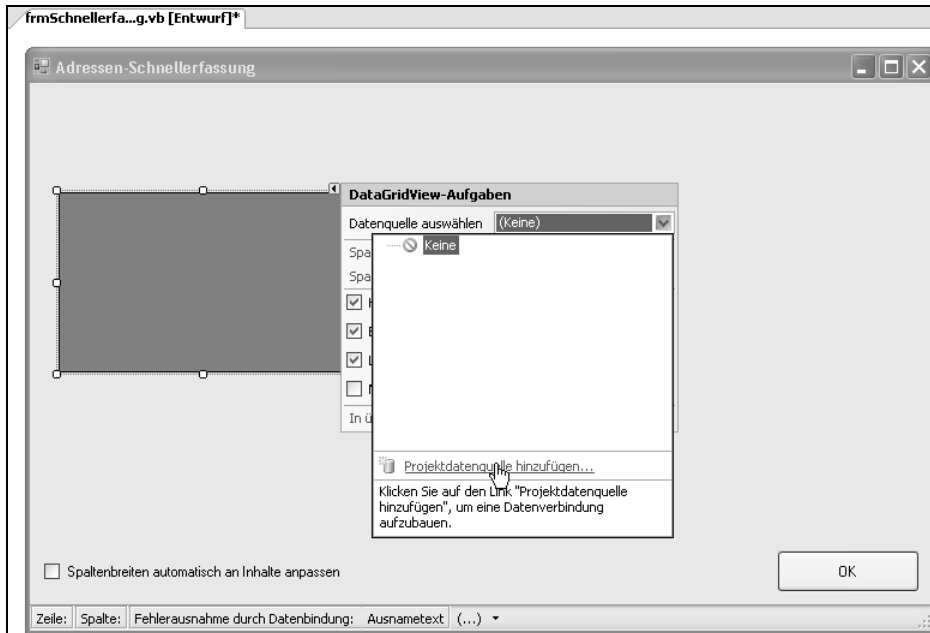


Abbildung 27.14: Wählen Sie aus der Aufgabenliste der *DataGridView* als Erstes *Datenquelle auswählen*, und klicken Sie – da es noch keine definierten Datenquellen gibt – auf den Link *Projektdatenquelle hinzufügen*

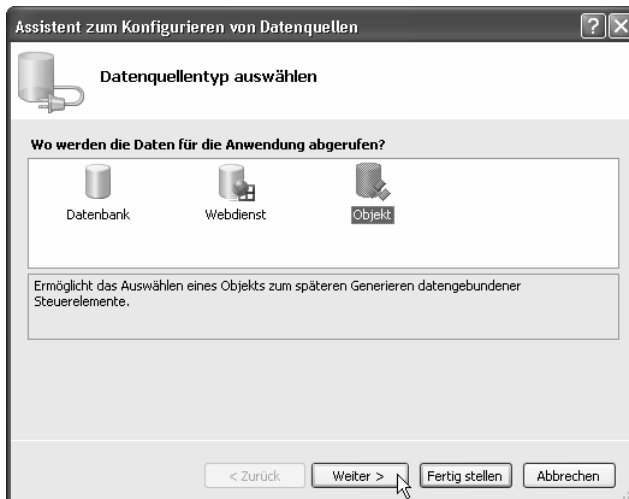


Abbildung 27.15: Der Assistent zum Konfigurieren von Datenquellen hilft Ihnen beim Erstellen einer neuen Datenquelle auf Auflistungsklassenbasis. Dazu wählen Sie im Assistentendialog das *Objekt*-Element.

- Suchen Sie in der Toolbox das *DataGridView*-Steuerelement. Ziehen Sie es anschließend ins Formular. Die Aufgabenliste des *DataGridView*-Steuerelements wird jetzt angezeigt.
- Klappen Sie die Liste *Datenquelle auswählen* auf.

- Da es noch keine Projektdatenquellen gibt, klicken Sie auf den Link *Projektdatenquelle hinzufügen*, um eine neue Datenquelle einzurichten. Ziel dabei wird es in den folgenden Schritten sein, die Adressen-Klasse, die die Auflistungsklasse für alle Adressen unserer kleinen Adressverwaltung darstellt, zur Datenquelle für das DataGridView-Steuerelement zu promovieren.
- Der Visual Basic-Designer zeigt nun einen Assistenten an, mit dem Sie Datenquellen konfigurieren können. Im ersten Assistentenschritt, etwa wie in Abbildung 27.15 zu sehen, wählen Sie als Datenquellentyp *Objekt*.

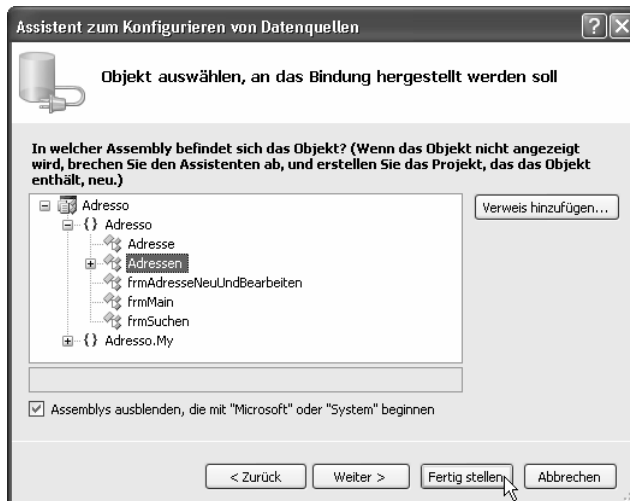


Abbildung 27.16: Bestimmen Sie in der TreeView, wo in der Objekt- und Namespace-Hierarchie Ihres Projektes die Klasse, die als Datenquelle fungieren soll, zu finden ist

- Bestätigen Sie diesen Schritt des Assistenten mit *Weiter*.
- Im nächsten und auch schon letzten Schritt (Abbildung 27.16) bestimmen Sie, wo innerhalb Ihrer Projektmappe sich das Objekt (sprich: die Auflistungsklasse) befindet, dessen Instanz als Datenquelle verwendet werden soll. Klappen Sie die entsprechenden Zweige in der Treeview auf, und selektieren Sie die *Adressen*-Auflistungsklasse.
- Klicken Sie anschließend auf *Fertigstellen*, um den Assistenten zu beenden.
- Wenn Sie das DataGridView-Steuerelement anschließend vergrößern, und es mehr oder weniger flächenfüllend auf dem Formular anordnen, erkennen Sie, dass ein Prozess stattgefunden haben muss, der die Adressenklasse analysiert und aus den verschiedenen Eigenschaften eines Elementes (Adresse) die Grobdefinition für die Tabelle erstellt hat (Abbildung 27.17).

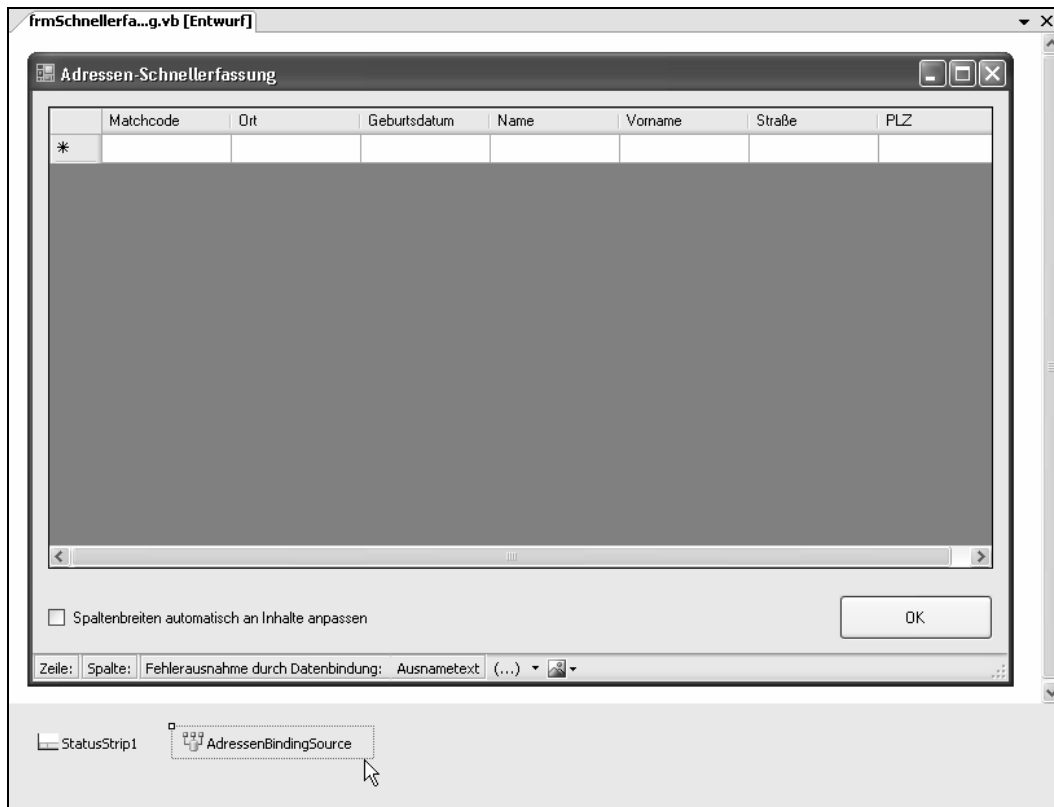


Abbildung 27.17: Nach dem Vergrößern des *DataGridView*-Steuerelements können Sie sehen, dass die Schemainformationen der *Adresse*-Klasse als Basis für das Einrichten der Tabellenspalten diente

In dieser Abbildung sehen Sie ebenfalls, dass der Designer die *BindingSource*-Komponente *AdressenBindingSource* erstellt und im Komponentenfach des Formulars untergebracht hat. Und diese stellt ab sofort das Bindeglied zwischen der Auflistungsklasse *Adressen* und dem *DataGridView*-Steuerelement dar.

Das Beste ist: Mit einer einzigen Zeile Code können Sie nun dafür sorgen, dass die Daten im *DataGridView*-Steuerelement angezeigt werden können.

- Dazu wechseln Sie zum Hauptformular des Projektes *frmMain.vb*, und öffnen Sie dieses im Designer.
- Klicken Sie im *MenuStrip*-Steuerelement des Formulars zunächst auf *Bearbeiten*, und doppelklicken Sie anschließend auf *Schnellerfassung von Adressen*.

Im Codeeditor können Sie nun die folgenden Zeilen eingeben, sodass sich für die Ereignisbehandlungsroutine Folgendes ergibt:

```

Private Sub tsmSchnellerfassung_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmSchnellerfassung.Click

    'Neue Instanz des Dialogs
    Dim locfrm As New frmSchnellerfassung

    'Eine Zeile reicht aus, um die Adressen darzustellen!
    Locfrm.AdressenBindingSource.DataSource = myAdressen

    'Dialog modal darstellen
    locfrm.ShowDialog()
End Sub

```

Wenn Sie diese Zeilen eingegeben haben und das Programm anschließend starten, wählen Sie zuerst aus dem Menü *Datei* den Menüpunkt *Zufallsadressen einfügen*. Wählen Sie anschließend aus dem Menü *Bearbeiten* den Menüpunkt *Schnellerfassung von Adressen*. Was Sie anschließend sehen, sollte nicht nur Abbildung 27.18 entsprechen, sondern Sie vor Begeisterung auch umhauen ...

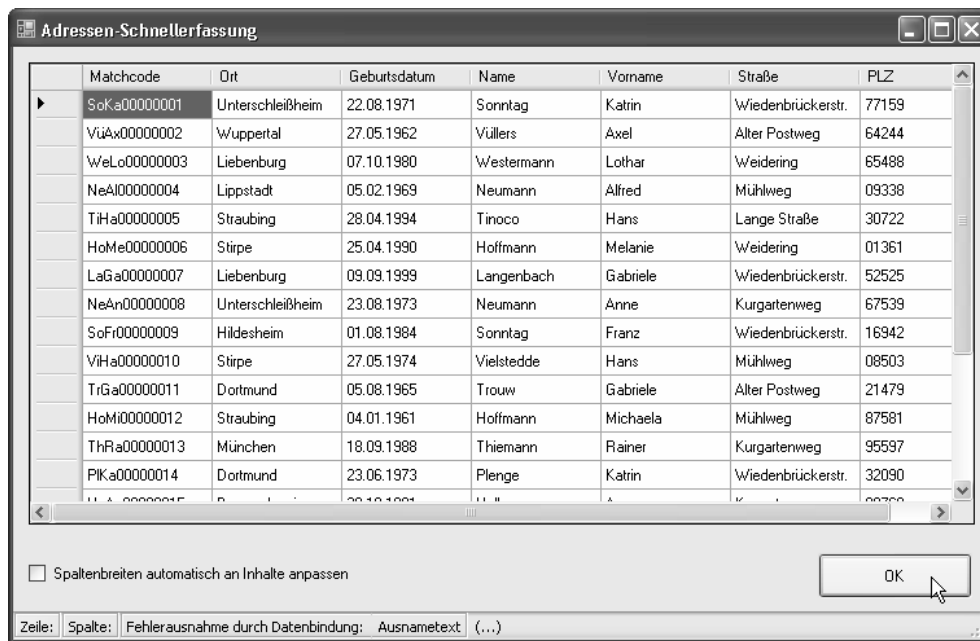


Abbildung 27.18: Mit einer einzigen Codezeile steht Ihnen ein komfortabler Editor zur Bearbeitung des Adress-Pools zur Verfügung

Doch der Teufel steckt wie immer im Detail, denn es gibt noch ein paar Punkte, die keinesfalls dem entsprechen, was man unter einer komfortablen Benutzeroberfläche verstehen könnte:

- Die Datenspalten sind willkürlich angeordnet; wir sollten versuchen, eine plausible Ordnung in die Datenspalten zu bekommen.
- Die Statuszeile sollte als Navigationshilfe die Position des Cursors im DataGridView-Steuerelement anzeigen.

- Sie können in jedem Feld jede beliebige Eingabe durchführen – Eingaben werden überhaupt nicht kontrolliert, und bei Feldern, die datentypbedingt ein bestimmtes Eingabeformat verlangen, kann man dabei auf die Nase fallen (siehe Abbildung 27.19).

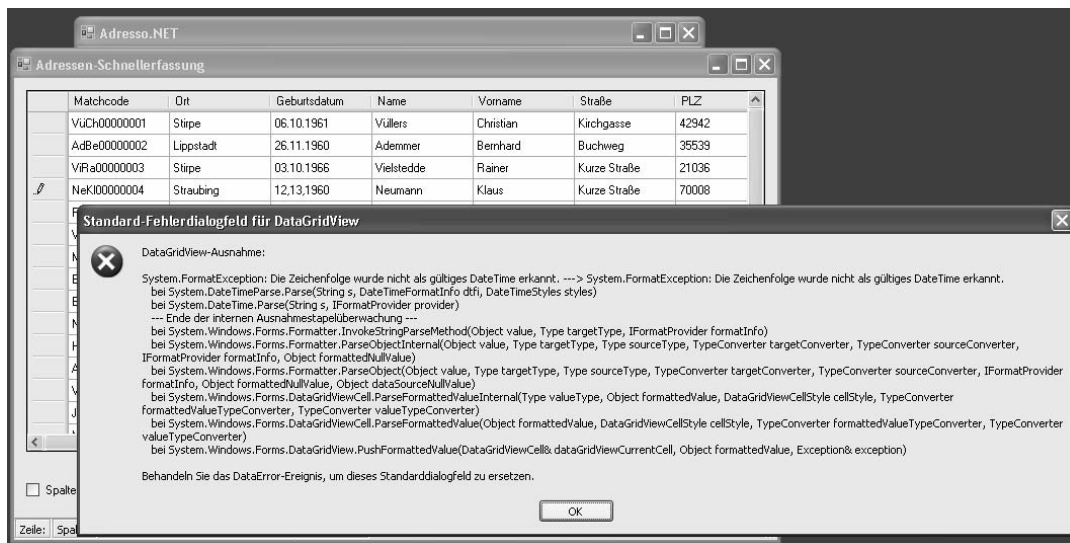


Abbildung 27.19: Solche Fehler sollten in der Release-Version Ihrer Anwendung nach Möglichkeit nicht mehr vorkommen

- Bei unserem Beispiel sollte der Matchcode nicht editierbar, auf jeden Fall aber neu eingebbar sein. Und: Auch hier dürfen doppelte Matchcodes nicht vorkommen und müssen im Rahmen einer Eingabeprüfung bei Neueingaben von Adressen ausgeschlossen werden.
- Und schließlich gibt es auch noch eine kleinen, aber gemeinen Störfaktor bei der Eingabe: Wenn Sie eine Zelle editiert haben und Ihre Änderungen mit **Eingabe** bestätigen, springt der Cursor nicht in das nächste Feld rechts des gerade bearbeiteten Feldes, sondern in das darunter. Damit ist der Anwender gar nicht in der Lage, Adressen eine nach der anderen (also zeilenweise) neu zu erfassen.

Die nächsten Abschnitte beschreiben, wie Sie mit gar nicht soviel Codierungsaufwand diese Ziele erreichen können.

Sortieren und Benennen der Spalten eines DataGridView-Steuerelements

Wenn Sie ein DataGridView-Steuerelement an eine Datenquelle gebunden haben, werden die Schema-informationen (welche Datenspalten mit welchen Datentypen gibt es) aus der Datenquelle ermittelt. Auf Basis dieser Schemainformationen werden die Spaltenköpfe eingerichtet. Programmtechnisch rufen Sie die Spaltendefinitionen einer Tabelle mithilfe der Columns-Eigenschaft (vom Typ DataGridViewColumnCollection) ab, die eine Auflistung mit DataGridViewColumn-Objekten enthält. Ein einzelnes DataGridViewColumn-Objekt dieser Auflistung bestimmt letzten Endes die Beschaffenheit einer Spalte, indem sie Spaltennamen, Beschriftung, darzustellenden Typ und Weiteres definiert.

Die Einrichtung bzw. Modifizierung nehmen Sie am einfachsten mit dem Designer vor. Die folgende Anleitung beschreibt, wie Sie ...

- ... definieren, welche Operationen (*Neuer Datensatz*, *Datensatz bearbeiten*, *Datensatz löschen*, *Umsortieren von Spalten*) mit dem DataGridView-Steuerelement durchgeführt werden dürfen,
- ... Spaltenreihenfolgen anordnen,
- ... die Beschriftung und Benennung von Spalten ändern,
- ... die Datentypen von Spalten einrichten bzw. verändern können.

BEGLEITDATEIEN: Um die Schritte der folgenden Unterabschnitte nachzuvollziehen, die zunächst noch den Visual Basic-Designer involvieren, verwenden Sie das bisher verwendete Beispiel im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\Adresso03 - zum Experimentieren\`. Dabei wird davon ausgegangen, dass Sie die Bindung der Datenquelle, wie im vorherigen Abschnitt beschrieben, bereits vorgenommen haben.

Öffnen der Liste der häufigen Aufgaben

Das DataGridView-Steuerelement lässt sich natürlich wie jedes andere Steuerelement über das Eigenschaftfenster des Visual Basic-Designers konfigurieren. Einfacher ist es allerdings, bestimmte Einstellungen über die Liste der häufigen Aufgaben auszuführen. Um diese Liste zu öffnen, verfahren Sie wie folgt:



Abbildung 27.20: Mit dem Smarttag des Steuerelements erreichen Sie die Liste der häufigen Aufgaben

- Klicken Sie auf das DataGridView-Steuerelement im Formular, um es zu selektieren.
- Klicken Sie auf den Smarttag des Steuerelements (der kleine nach rechts weisende Pfeil an der rechten oberen Ecke), um die Liste häufiger Aufgaben des Steuerelements zu öffnen.

Bestimmen, welche Aufgaben mit dem DataGridView-Steuerelement erledigt werden dürfen

- Öffnen Sie die Liste häufiger Aufgaben durch Klick auf das Smarttag des zuvor selektierten DataGridView-Steuerelements (siehe Abbildung 27.20).
- Wählen Sie im Dialogfeld durch An- oder Abwählen der entsprechenden Kontrollkästchen, welche Funktionen das DataGridView-Steuerelement zu Verfügung stellen soll.

HINWEIS: Auch wenn Sie hier das Hinzufügen von Datensätzen erlauben, wird diese Funktion zur Laufzeit nur dann zur Verfügung stehen, wenn die AllowNew-Eigenschaft der entsprechenden BindungsSource-Komponente (oder – bei programmtechnischer Verknüpfung – auch die CurrencyManager-Komponente) auf True gesetzt wurde.

Wenn Sie später zur Laufzeit feststellen, dass beide Eigenschaften auf True gesetzt wurden, Sie aber dennoch die Funktionalität zum Hinzufügen von Datensätzen vermissen, stellen Sie sicher, ob Sie die Daten einer Auflistung nicht versehentlich mithilfe ihrer DataSource-Eigenschaft *direkt* an das DataGridView-Steuerelement anstatt an die verknüpfte BindungsSource-Komponente zugewiesen haben.⁵

Anordnen der Spalten eines DataGridView-Steuerelements mit Designer-Unterstützung

- Öffnen Sie die Liste häufiger Aufgaben durch Klick auf das Smarttag des zuvor selektierten DataGridView-Steuerelements (siehe Abbildung 27.20).
- Klicken Sie auf Spalten bearbeiten. Sie sehen einen Dialog, etwa wie auch in Abbildung 27.21 zu sehen.

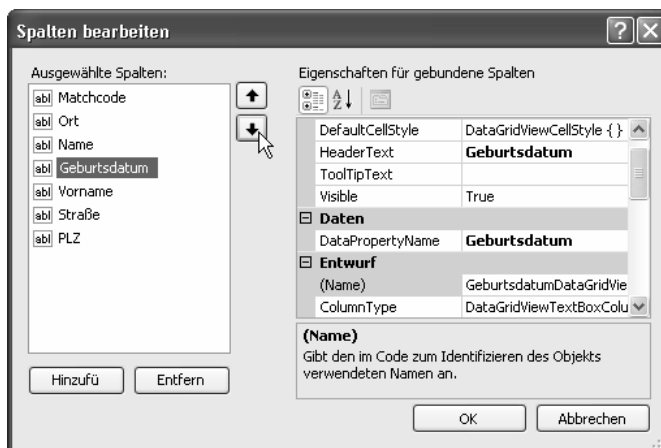


Abbildung 27.21: Mit den Pfeilschaltflächen (hier durch den Mauszeiger gekennzeichnet) ordnen Sie die Reihenfolge der Spalten neu an

- Mit den Pfeilschaltflächen ordnen Sie, wie in der Abbildung zu sehen, die Reihenfolge der Spalten des DataGridView-Steuerelements neu an. Der hier in der Liste ganz oben stehende Eintrag entspricht der ganz links im Steuerelement erscheinenden Spalte.

Die Beschriftung und Benennung von Spalten ändern

Insgesamt drei Eigenschaften eines DataGridViewColumn-Objekts, das für die Eigenschaften einer DataGridView-Spalte zuständig ist, haben etwas mit Benennungen bzw. Beschriftungen zu tun. Gerade wenn die Spaltendefinitionen aus Schemainformationen einer Datenquelle stammen, kann es anfangs verwirren, welche Eigenschaften für welche Aufgabe zuständig sind.

Um diese Einstellungen mit dem Designer durchzuführen, verfahren Sie wie folgt:

⁵ Und nun raten Sie mal, wieso dieses Kapitel einen halben Tag eher hätte fertig gestellt sein können...

- Öffnen Sie die Liste häufiger Aufgaben durch Klick auf das Smarttag des zuvor selektierten DataGridView-Steuerelements (siehe Abbildung 27.20).
- Klicken Sie auf *Spalten bearbeiten*. Sie sehen einen Dialog, etwa wie auch in Abbildung 27.21 zu sehen.
- In der Eigenschaftenliste bestimmen Sie folgende Eigenschaften:
 - **Headertext:** Legen Sie mit dieser Zeichenfolge fest, welche Überschrift die entsprechende Spalte im DataGridView-Steuerelement aufweisen soll.
 - **DataPropertyName:** Bestimmen Sie mit dieser Zeichenfolge, welcher Objekteigenschaft der Datenquelle diese Spalte zugeordnet sein soll. Sie sollten diese Eigenschaft nur dann ändern, wenn sich die Eigenschaft eines Objektes der als Datenquelle fungierenden Auflistung geändert hat. Belassen Sie anderenfalls diese Eigenschafteneinstellung so, wie sie ist.
 - **Name:** Bestimmen Sie mit dieser Zeichenfolge, mit welchem Textschlüssel Sie auf ein Element der DataGridViewColumnCollection oder der DataGridViewCellCollection (einer DataGridViewRow-Auflistung) zugreifen. Diese Eigenschafteneinstellung ist also dann wichtig, wenn Sie programmtechnisch entweder auf eine Spaltendefinition oder eine Zelle einer DataGridView-Instanz zugreifen wollen.

Zugriff auf ein Element der DataGridViewColumnCollection (eine Spaltendefinition):

'Index einer Spalte aus dem Spaltennamen ermitteln
 myGebDatColumnIndex = dgvAdressen.Columns("GeburtsdatumDataGridViewTextBoxColumn").Index

Zugriff auf ein Element der DataGridViewCellCollection (einer Zelle der Tabelle):

'Eine bestimmte Zelle aus Zeilennummer und Spaltennamen ermitteln
 Dim locRow As DataGridViewRow = dgvAdressen.Rows(Zeilenummer)
 Dim locCell As DataGridViewCell = locCurrentRow.Cells("MatchcodeDataGridViewTextBoxColumn")

Bestimmen der Spaltentypen einer DataGridView-Instanz

Mit der ColumnType-Eigenschaft des *Spalten bearbeiten*-Dialogs bestimmen Sie, mit welchem Steuerelementtyp die Spalte ausgestattet werden soll.

HINWEIS: Die ColumnType-Eigenschaft bestimmt dadurch *nicht* notwendigerweise, welcher Datentyp in einer Spalte verarbeitet wird. So wählen Sie für numerische Werte, Datumswerte oder Texte als ColumnType-Eigenschaft wahrscheinlich immer DataGridViewTextBoxColumn (DataGridViewComboBoxColumn würden Sie dabei wählen, um aus einer vorgegebenen Liste mit Werten nur bestimmte zur Auswahl zuzulassen, oder wenn Sie eine Verknüpfung mit einer anderen Datentabelle herstellen wollen).

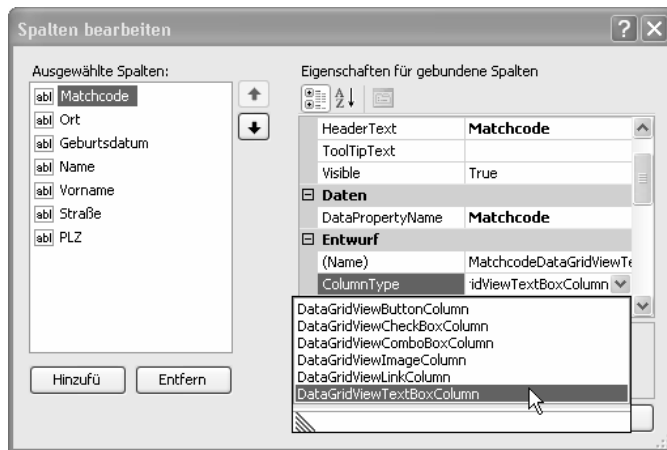


Abbildung 27.22: Mit der *ColumnType*-Eigenschaft bestimmen Sie den Steuerelementtyp (nicht Datentyp!), den eine *DataGridView*-Spalte verarbeiten soll

Insgesamt 6 verschiedene Spaltentypen stehen standardmäßig zur Verfügung:

ColumnType	Aufgabe
DataGridViewButtonColumn	Implementiert Schaltflächen in den Datenzeilen für die entsprechende Spalte
DataGridViewCheckBoxColumn	Implementiert Kontrollkästchen in den Datenzeilen für die entsprechende Spalte
DataGridViewComboBoxColumn	Implementiert Aufklapplisten in den Datenzeilen für die entsprechende Spalte
DataGridViewImageColumn	Implementiert Bilddarstellungen in den Datenzeilen für die entsprechende Spalte
DataGridViewLinkColumn	Implementiert einen Web-Link in den Datenzeilen für die entsprechende Spalte
DataGridViewTextBoxColumn	Implementiert ein Texteingabefeld in den Datenzeilen für die entsprechende Spalte

Tabelle 27.1: Die standardmäßig vorhandenen Steuerelemente, die als Darstellungs- und Anzeigeeinstanzen innerhalb von *DataGridView*-Tabellen fungieren können

HINWEIS: Sie werden die *ColumnType*-Eigenschaft vergeblich im *DataGridViewColumn*-Objekt suchen – es handelt sich nämlich um eine nennen wir sie mal: »virtuelle« Designereigenschaft, die sich nur auf die Erstellung des Designer-Codes (das nächste Kapitel erklärt mehr dazu) bezieht. Wenn Sie für ein gebundenes *DataGridView*-Steuerelement eine Spaltentypumstellung vornehmen wollen, platzieren Sie den entsprechenden Code dazu am besten direkt unterhalb von *InitializeComponent* im Standardkonstruktor des Formulars, das die *DataGridView* beinhaltet.

Programmtechnisch legen Sie das zu verwendende Steuerelement für die entsprechende Spalte mit der *CellTemplate*-Eigenschaft fest. Das Umdefinieren einer Spalte führen Sie am besten im Standardkonstruktor des Formulars durch, etwa wie folgt:

```
Public Class frmSchnellerfassung
    Sub New()

        ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.
        InitializeComponent()

        ' Fügen Sie Initialisierungen nach dem InitializeComponent()-Aufruf hinzu.
```

```

'Eines der möglichen Steuerelemente für die Spalte manuell zuordnen.
dgvAdressen.Columns("NameDataGridViewTextBoxColumn").CellTemplate = New DataGridViewTextBoxColumn()
End Sub
.
.
.
End Class

```

TIPP: Wenn Sie eigene Steuerelemente für Tabellenzellen mit individuellen Verhaltensweisen implementieren wollen, leiten Sie entweder eines der vorhandenen Steuerelemente der oben stehenden Tabelle ab, erweitern es und binden es auf die hier beschriebene Weise ein. Oder Sie leiten es von der Basis aller Steuerelemente ab, die in Tabellenzeilen einer DataGridView zur Anwendung kommen können. Verwenden Sie dazu die abstrakte Basisklasse `DataGridViewCell`. Wichtig: Achten Sie dabei darauf, die `Clone`-Methode zu überschreiben und neu zu implementieren, da die Definition für alle Zellen einer Spalte aus der ersten Instanz erstellt wird, die man über die `CellTemplate`-Eigenschaft definiert. Dies gilt nur dann, wenn Sie einem vorhandenen Steuerelement weitere Eigenschaften hinzufügen, deren Inhalte natürlich zusätzlich zu den bereits vorhandenen kopiert werden müssen. Dies gilt aber auf jeden Fall, wenn Sie das Steuerelement auf Basis von `DataGridViewCell` komplett neu implementieren.

Programmtechnisches Abrufen der aktuellen Zeile und aktuellen Spalte

Die aktuelle Zeile und die aktuelle Spalte, also die Position innerhalb der DataGridView-Tabelle, in der sich der Cursor gerade befindet, rufen Sie mit der Eigenschaft `CurrentCell` ab. Diese Eigenschaft liefert ein Objekt auf Basis der `DataGridViewCell`-Klasse zurück.

Um die aktuelle Zeile zu ermitteln, verwenden Sie die `RowIndex`-Eigenschaft dieses Objektes. Die aktuelle Spalte ermitteln Sie mit `ColumnIndex`.

Feststellen, dass sich die aktuelle Zelle geändert hat

Wenn Sie informiert werden möchten, wenn sich die aktuelle Position des Cursors in der Tabelle geändert hat, binden Sie das `CurrentCellChanged`-Ereignis des DataGridView-Steuerelements ein. Das folgende Beispiel demonstriert, wie Sie Zeile und Spalte in einer Statuszeile anzeigen lassen können; diese Anzeige wird aktualisiert, sobald sich die Cursorposition im DataGridView-Steuerelement verändert.

BEGLEITDATEIEN: Sie finden diese Prozedur als Teil des Projektes im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\Adresso03` in der Codedatei `frmSchnellerfassung.vb`.

```

'Wird aufgerufen, wenn eine neue Zelle zur aktuellen Zelle wird.
Private Sub dgvAdressen_CurrentCellChanged(ByVal sender As Object, ByVal e As System.EventArgs) _
    Handles dgvAdressen.CurrentCellChanged
    With dgvAdressen
        ts1Zeile.Text = "Zeile: " & .CurrentCell.RowIndex.ToString
        ts1Spalte.Text = "Spalte: " & .Columns(.CurrentCell.ColumnIndex).HeaderText
    End With
End Sub

```

Formatieren von Zellen

Damit Daten in einem `DataGridView`-Steuerelement übersichtlich aufbereitet werden können, lassen sie sich durch eine Formatzeichenfolge formatieren (mehr zum generellen Umgang mit Formatzeichenfolgen lesen Sie in ► Kapitel 17). Möchten Sie zum Beispiel für eine Spalte, die Datumswerte anzeigt, bestimmen, dass diese im Format »ddd, dd. MMM yyyy« dargestellt werden (was für den 24.2. des Jahres 2006 die Zeichenfolge »Fr, 24 Feb 2006« ergäbe), dann stellen Sie dieses Format global für alle Zellen der entsprechenden Spalte wie folgt ein:

```
dgvAdressen.Columns("GeburtsdatumDataGridViewTextBoxColumn").DefaultCellStyle.Format = "ddd, dd. MMM yyyy"
```

Wenn Ihr `DataGridView`-Steuerelement Zellen lediglich darstellen aber nicht bearbeiten muss, dann genügt diese Art der Formatierungsfestlegung für die entsprechende(n) Spalten(n) im Regelfall.

Wenn sich die Formatierung dabei nicht über eine Formatzeichenfolge realisieren lässt, können Sie das `CellFormatting`-Ereignis des `DataGridView`-Steuerelements behandeln, um die Formatierung dort vorzunehmen. Zwei Möglichkeiten stehen Ihnen nun zur Formatierung zur Verfügung:

- Bestimmen Sie über die Eigenschaftskombination `CellStyle.Format` des Ereignisparameters `e` die individuell anzuwendende Formatzeichenfolge. Oder:
- Lesen Sie den zu formatierenden Wert über die `Value`-Eigenschaft des Ereignisparameters `e` aus, konvertieren Sie ihn im Bedarfsfall in den entsprechenden Datentyp, formatieren Sie ihn (in der Regel in Form einer Zeichenkette) und schreiben Sie ihn anschließend zurück in die `Value`-Eigenschaft. Setzen Sie in diesem Fall die `FormattingApplied`-Eigenschaft des Ereignisparameters `e` auf `True`, um dem `DataGridView`-Steuerelement anzuzeigen, dass Sie die Formatierung der Zelle komplett selbst übernommen haben.

HINWEIS: Denken Sie bitte daran, dass das `CellFormatting`-Ereignis für jede darzustellende Zelle ausgelöst wird. Damit Sie die richtige zu formatierende Zelle »treffen«, testen Sie mit dem Ereignisparameter `ColumnIndex`, ob sich das gerade ausgelöste Ereignis auf die entsprechende Spalte bezieht. Ein Beispiel dafür finden Sie im nächsten Abschnitt.

Wenn Ihre individuell formatierte Zelle auch die Bearbeitung von Werten zulässt, lesen Sie bitte auch den nächsten Abschnitt.

Problemlösung für das Parsen eines Zellenwerts mit einem komplexen Anzeigeformat

Bei der Datenerfassung von anderen Typen als reine Zeichenketten stellt sich immer das Problem der Datenkonvertierung. Für die wichtigsten Datentypen implementiert das `DataGridView`-Steuerelement so genannte `Parsing`-Algorithmen, die den meisten Konvertierungsansprüchen Genüge tun.

Das Problem bei der Verarbeitung anderer Typen als reine Zeichenketten ist, dass der Datentyp für die Darstellung in eine Zeichenkette umgewandelt werden muss. Bearbeitet der Anwender eine Zelle, in der ein anderer Datentyp als eine Zeichenkette erfasst werden soll, editiert er natürlich zunächst einmal ebenfalls eine Zeichenkette, die anschließend wieder zurück in den eigentlichen Datentyp konvertiert werden muss.

Solange wie Sie ein sehr einfaches Standardformat für die Darstellung des Datentyps verwenden, treten dabei keine Probleme auf. Handelt es sich aber um eine sehr informative Aufbereitung eines Datentyps, dann wird der Standardparser des DataGridView-Steuerelements dieses Formats nicht mehr hergeben. Oder, um ein Beispiel zu bemühen:

Sie stellen in einer Spalte, in der Datumswerte angezeigt und bearbeitet werden sollen, die Werte im Format »dd.MM.yyyy« dar. Die Darstellung ist zwar sehr simpel, aber der Parser hat mit dieser Darstellung keine Probleme – er kann also einen in diesem Format editierten Wert ohne Probleme wieder zurück in den eigentlichen Datentyp umwandeln.

Entscheiden Sie sich allerdings für die Darstellung von »dd.MM.yyyy (dddd)« – der Wochentag wird hier also in Klammern neben dem eigentlichen Datum angezeigt –, und Sie lassen den Anwender die Zelle auch in diesem Format editieren, wird der Parser mit dieser komplexen Darstellung des Datums nicht mehr zurechtkommen.

Nun gibt es zwei generelle Möglichkeiten, dieses Problem zu umgehen: Sie implementieren für die entsprechende Datenspalte eine Ereignisbehandlungsroutine für das CellParsing-Ereignis und sorgen dort selbst dafür, dass der Parsing-Vorgang auch mit dem komplex formatierten Datentyp nicht fehlschlägt. Das kann aber unter Umständen eine ganze Menge Programmieraufwand bedeuten.

Oder, und das wäre meine empfohlene Vorgehensweise, sie formatieren zum Editieren die entsprechende Zelle anders als für die reine Anzeige. Und das funktioniert, weil sowohl beim Anzeigen als auch kurz vor dem Editieren das CellFormatting-Ereignis ausgelöst wird. Das folgende Beispiel zeigt, wie es funktioniert:

BEGLEITDATEIEN: Sie finden diese Prozedur als Teil des Projektes im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\Adresso03` \ in der Codedatei `frmSchnellerfassung.vb`.

```
'Wird für jede Zelle aufgerufen, wenn der Inhalt dieser formatiert wird.
Private Sub dgvAdressen_CellFormatting(ByVal sender As Object, _
ByVal e As System.Windows.Forms.DataGridViewCellFormattingEventArgs) Handles dgvAdressen.CellFormatting

    'Nur die Geburtsdatums-Spalte wird besonders formatiert.
    If e.ColumnIndex = myGebDatColumnIndex Then
        'Die Zelle ermitteln. Aus Geschwindigkeitsgründen sollte das in einem
        'Rutsch erfolgen; hier getrennt, damit es deutlicher wird.
        Dim locCurrentRow As DataGridViewRow = dgvAdressen.Rows(e.RowIndex)
        Dim locCurrentCell As DataGridViewCell = locCurrentRow.Cells(e.ColumnIndex)

        'Wenn ein Geburtsdatum zum Bearbeiten formatiert wird,
        'dann ein "parse"-bares Format darstellen.
        If locCurrentCell.IsInEditMode Then
            e.CellStyle.Format = "dd.MM.yyyy"
        Else
            'Nur für die Anzeige ein "mehr informatives Format" darstellen.
            e.CellStyle.Format = "dd.MM.yyyy (dddd)"
        End If
    End If
End Sub
```

Das Ergebnis dieses Aufwands können Sie betrachten, wenn Sie das Beispielprojekt starten, ein paar Zufallsadressen erzeugen und diese zum Schnellbearbeiten darstellen lassen. Wenn Sie anschließend das Geburtsdatum eines Datensatzes bearbeiten, sehen Sie, dass sich das Format zur Eingabe ändert.

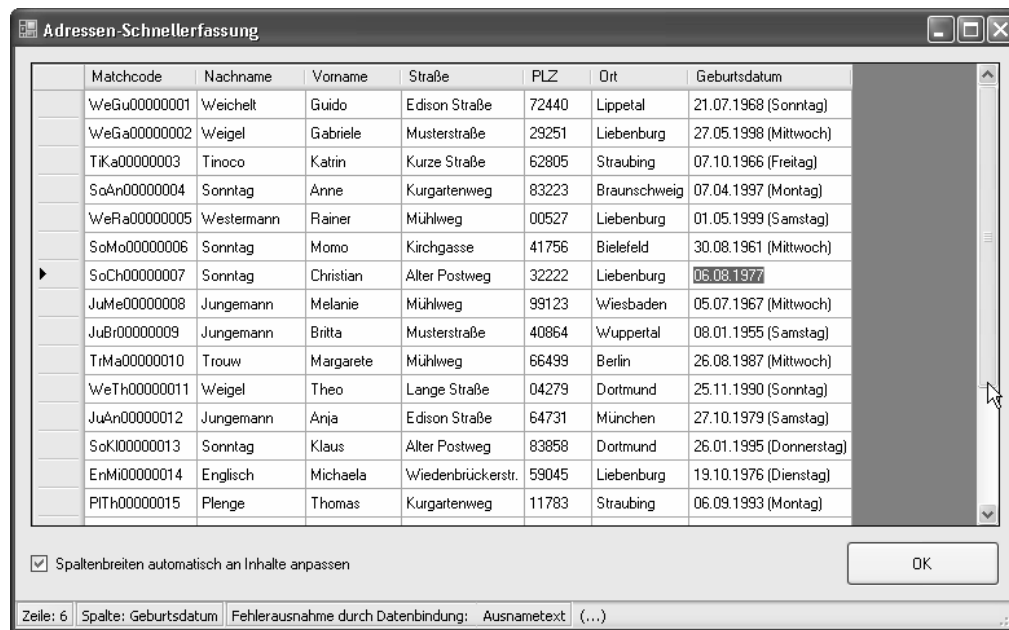


Abbildung 27.23: Durch geschickte Behandlung des *CellFormatting*-Ereignisses wird für die Bearbeitung ein anderes Format als für die Anzeige verwendet

HINWEIS: Da das *CellFormatting*-Ereignis für *jede* einzelne darzustellende Zelle im Anzeigebereich des *DataGridView*-Steuerelements aufgerufen wird, sollten Sie sehr auf eine effiziente Programmierung achten und Code formulieren, der so wenig Zeit wie möglich benötigt.

Verhindern des Editierens bestimmter Spalten aber Gestatten ihrer Neueingabe

Falls Sie das Adresso-Szenario bis hierher verfolgt haben, dann wissen Sie, dass das Beispiel das Erfassen eines Matchcodes beim Neuerfassen einer Adresse erlaubt, es aber verbietet, den Matchcode einer vorhandenen Adresse zu editieren.

Mit dem Setzen simpler Eigenschaften können Sie das *DataGridView*-Steuerelement nicht dazu bringen, solche Teilreglementierungen umzusetzen.

Zwar ist es möglich, beispielsweise eine Spalte mit der Anweisung

```
dgvAdressen.Columns("MatchcodeDataGridViewTextBoxColumn").ReadOnly = True
```

für die Bearbeitung ganz zu sperren. Doch das entspricht nicht dem Sinne des Erfinders, jedenfalls nicht bei unserem Beispiel.

Mit dem `CellBeginEdit`-Ereignis können Sie die Kontrolle über das Editieren einer Zelle in die eigenen Hände nehmen. Innerhalb dieses Ereignisses brauchen Sie nämlich nur festzustellen, ob eine neue Zeile eingegeben oder eine vorhandene editiert wird, und um welche Datenspalte es sich dabei handelt.

```
'Wird aufgerufen, bevor das Bearbeiten einer Zelle startet.
Private Sub dgvAdressen_CellBeginEdit(ByVal sender As Object, ByVal e As _
    System.Windows.Forms.DataGridViewCellCancelEventArgs) Handles dgvAdressen.CellBeginEdit

    'Verhindern, dass der Matchcode bearbeitet werden kann.
    If e.ColumnIndex = dgvAdressen.Columns("MatchcodeDataGridViewTextBoxColumn").Index Then

        'Neue Matchcodes können zwar bearbeitet werden, aber keine vorhandenen.
        'IsNewRow zeigt ein, ob es sich um die "Neue-DataGridView-Zeile" handelt.
        If Not dgvAdressen.CurrentRow.IsNewRow Then
            'Wie gesagt: NICHT neue Zeile, dann Bearbeitung nicht zulassen.
            e.Cancel = True
        End If
    End If
End Sub
```

Schlüssel bei diesem Beispiel sind zwei Komponenten: Zum einen die Eigenschaft `IsNewRow` des `CurrentRow`-Objektes des `DataGridView`-Steuerelements, die Auskunft darüber gibt, ob eine vorhandene Zeile bearbeitet oder eine neue eingegeben wird. Zum anderen die `Cancel`-Eigenschaft des Ereignisparameters `e` des `CellBeginEdit`-Ereignisses, mit dem Sie durch Setzen auf `True` verhindern können, dass die Zeile editiert wird.

Überprüfen der Richtigkeit von Eingaben auf Zellen- und Zeilenebene

Ein Blick auf Abbildung 27.12 offenbart, dass es beim `DataGridView`-Steuerelement so etwas wie `ErrorProvider` auf Zellen- und Zeilenebene gibt (falls Sie diese `ErrorProvider`-Klasse nicht kennen, der ► Abschnitt »Überprüfung auf richtige Benutzereingaben in Formularen« ab Seite 766 weiß mehr dazu).

Und in der Tat: Idealerweise erlauben Sie Falscheingaben zunächst zwar auf Zellenbasis, machen den Anwender aber im Moment des Verlassens einer »falsch« bearbeiteten Zelle auf diesen Missstand aufmerksam. Sie erlauben aber nicht das Verlassen einer Zeile, solange diese noch nicht korrigierte Fehler enthält.

Wenn der Anwender eine Eingabe durchgeführt hat, muss die Richtigkeit seiner Eingabe überprüft werden. Diesen Vorgang nennt man Validieren. Dementsprechend nennen sich die Ereignisse des `DataGridView`-Steuerelements, die ausgelöst werden, wenn Eingabeüberprüfungen auf Zellen- bzw. Zeilenebene anstehen `CellValidating` und `RowValidating`.

BEGLEITDATEIEN: Sie finden diese Prozedur als Teil des Projektes im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\Adresso03` \ in der Codedatei `frmSchnellerfassung.vb`.

Die folgenden beiden Ereignisbehandlungsroutinen implementieren die Eingabereglementierungen für unsere Adresso-Anwendung auf bekannte Weise:

- Es dürfen keine doppelten Matchcodes vorhanden sein.
- Jedes Feld muss ausgefüllt werden.
- Postleitzahlen müssen mindestens 4-stellig und dürfen höchstens 5-stellig sein, und sie müssen aus Ziffern bestehen.
- Nur gültige Datumsformate dürfen eingegeben werden.

Die prinzipielle Arbeitsweise beider Routinen ist wie folgt:

- Als erstes wird das `CellValidating`Ereignis aufgerufen, wenn der Cursor das Eingabefeld verlässt. **Wichtig:** Dieses Ereignis wird auch dann aufgerufen, wenn keine Eingabe oder Änderung in einem Feld erfolgte, also auch dann, wenn der Cursor nur über das Feld hinweg bewegt wurde. Die Validierung wird in der unten stehenden Ereignisbehandlungsroutine deswegen nur dann ausgeführt, wenn die Zelle mit ihrer `IsInEditMode`-Eigenschaft anzeigt, dass sie tatsächlich bearbeitet wurde (oder besser: *wird* – denn zum Zeitpunkt der Validierung läuft die Bearbeitung noch und das Beenden der Bearbeitung könnte auch mit `e.Cancel = True` verhindert werden!).
- Wurde ein Fehler in einem der Eingabefelder festgestellt, wird die Zelle durch Zuweisen eines Textes an ihre `ErrorText`-Eigenschaft als fehlerhaft markiert. Das bewirkt, dass ein kleines rotes Ausrufungszeichen an das Ende der Zelle gesetzt wird. Den Fehlertext bekommt der Anwender als Tooltip angezeigt, wenn er mit dem Mauszeiger auf das Ausrufungszeichen fährt. War das Eingabefeld okay, wird die `ErrorText`-Eigenschaft auf `Nothing` gesetzt. Ein etwaiger vorhandener Fehlertext aus einer vorherigen Falscheingabe wird auf diese Weise zurückgesetzt. An dieser Stelle wird das Verlassen einer Zelle nach rechts oder links allerdings noch nicht verhindert.
- Versucht der Anwender eine neue Zeile »zu betreten«, tritt das `RowValidating`-Ereignis ein. Jetzt hat das Programm die Möglichkeit, die Richtigkeit der kompletten Zeile zu überprüfen. Das ist dann der Fall, wenn keine einzelne Zelle der betroffenen Zeile mehr über eine gesetzte `ErrorText`-Eigenschaft verfügt, und auf genau das überprüft der Code in dieser Ereignisbehandlungsroutine. Sollte er allerdings dabei auf noch nicht korrigierte Fehler stoßen, verhindert er einerseits das Verlassen der Zeile durch Setzen von `e.Cancel = True` und setzt ebenfalls einen Fehlerhinweis (dieses Mal auf Zeilenebene), dass es in der Zeile noch nicht korrigierte Fehler gibt.

'Wird aufgerufen, wenn der Inhalt einer Zelle überprüft werden soll.

```
Private Sub dgvAdressen_CellValidating(ByVal sender As System.Object, ByVal e As _  
    System.Windows.Forms.DataGridViewCellValidatingEventArgs) Handles dgvAdressen.CellValidating
```

```
    'Aktuell zu validierende Zeile und Zelle ermitteln  
    Dim locCurrentRow As DataGridViewRow = dgvAdressen.Rows(e.RowIndex)  
    Dim locCurrentCell As DataGridViewCell = locCurrentRow.Cells(e.ColumnIndex)
```

```
    'Zellen werden nur validiert, wenn sie bearbeitet wurden.  
    '(Und beim Validieren *werden* sie noch bearbeitet)
```

```
    If locCurrentCell.IsInEditMode Then
```

```
        'Postleitzahleninhalt überprüfen  
        If e.ColumnIndex = dgvAdressen.Columns("PLZDataGridViewTextBoxColumn").Index Then
```

```
            'Postleitzahlen dürfen nicht mehr als 5-stellig sein,
```

```

'und nur aus Ziffern bestehen.
If (e.FormattedValue.ToString.Length < 6 And _
    e.FormattedValue.ToString.Length > 4 And _
    IsNumeric(e.FormattedValue.ToString)) Then
    locCurrentCell.ErrorText = Nothing
Else
    locCurrentCell.ErrorText = "Die Postleitzahl hat das falsche Format!"
End If

ElseIf e.ColumnIndex = dgvAdressen.Columns("GeburtsdatumDataGridViewTextBoxColumn").Index Then

    'Geburtsdatumsformat überprüfen
    Dim locDate As Date
    If Not Date.TryParse(e.FormattedValue.ToString, locDate) Then
        locCurrentCell.ErrorText = "Das eingegebene Datum hat das falsche Format!"
    Else
        'Zurücksetzen
        locCurrentCell.ErrorText = Nothing
    End If
Else
    'Alle anderen Felder nur auf nicht vorhandene Eingabe überprüfen
    If String.IsNullOrEmpty(e.FormattedValue.ToString) Then
        locCurrentCell.ErrorText = "Fehlende Eingabe!"
    Else
        locCurrentCell.ErrorText = Nothing
    End If
End If
End If
End Sub

'Wird aufgerufen, wenn beim Wechseln der Zeile
'die Richtigkeit der Inhalte *aller* Zellen überprüft werden sollen.
Private Sub dgvAdressen_RowValidating(ByVal sender As Object, ByVal e As _
    System.Windows.Forms.DataGridViewCellCancelEventArgs) Handles dgvAdressen.RowValidating

    'Alle Zellen der betroffenen Zeile durchlaufen
    For Each locCell As DataGridViewCell In dgvAdressen.Rows(e.RowIndex).Cells

        'Schauen, ob noch irgendwo ein Fehlertext vermerkt ist
        If Not String.IsNullOrEmpty(locCell.ErrorText) Then
            'Falls ja, dann auch einen Fehler auf Zeilenbasis eintragen
            dgvAdressen.Rows(e.RowIndex).ErrorText = "Diese Zeile hat noch nicht korrigierte Fehler!"
            e.Cancel = True
            Return
        End If
    Next

    'Keine der Zellen wies einen Fehler auf --> Zeilenfehler zurücksetzen.
    dgvAdressen.Rows(e.RowIndex).ErrorText = Nothing
End Sub

```

Ändern des Standardpositionierungsverhaltens des Zellencursors nach dem Editieren einer Zelle

Das DataGridView-Steuerelement hat ein etwas seltsames Verhalten, das bei Anwendern, die viele Eingaben mit seiner Hilfe vornehmen müssen, auf wenig Gegenliebe stößt. Nachdem Sie eine Zelle im Steuerelement geändert haben, springt der Cursor nicht in die rechts daneben stehende Zelle sondern springt eine Zelle nach unten.

Da es Anwender nun einmal gewohnt sind, eine Eingabe auch mit der Taste **Eingabe** und nicht mit der Cursor-nach-rechts-Taste zu beenden, macht sich hier schnell Unmut breit.

Dummerweise verfügt das DataGridView-Steuerelement zwar über die meisten Einstellmöglichkeiten aller Steuerelemente überhaupt, aber über keine, um dieses Verhalten direkt zu ändern.

Die Klassenvererbung ist die einzige Möglichkeit, diesem Problem zu entgehen. Das bedeutet: Sie vererben das DataGridView-Steuerelement, das im Prinzip ja nichts anderes ist als eine Klasse, in eine neue Klasse und kommen auf diese Weise an bestimmte Methoden und Eigenschaften heran, die Ihnen bei der direkten Verwendung eines DataGridView-Objektes nicht zugänglich sind. Und dabei handelt es sich um genau die Elemente, die als `Protected` so gekennzeichnet sind, dass auf sie nur von der Klasse selbst und von jeder geerbten Klasse darauf zugegriffen werden kann.

Wenn Sie ein Steuerelement in Visual Basic 2005 vererben, erstellen Sie automatisch ein neues Steuerelement. Damit hat dieser Abschnitt in diesem Kapitel eigentlich gar nichts verloren, denn erst ► Kapitel 30 beschäftigt sich mit diesem Thema detaillierter. Mir schien es aber sinnvoll, gerade dieses herbe Manko des DataGridView-Steuerelements im Kontext des Themas *DataGridView* zu erklären. Wenn Sie genauer interessiert, wie Sie eigene Steuerelemente mit Visual Basic 2005 für die Vereinfachung von Programmierungen Ihrer eigenen Anwendungen erstellen, lesen Sie eben Kapitel 30. Die Erklärung des Programms an dieser Stelle erfolgt deswegen auch nur in aller Kürze.

BEGLEITDATEIEN: Sie finden das folgende Beispielprojekt im Verzeichnis `.\\VB 2005 - Entwicklerbuch\\G - Smart-Client\\Kap27\\Adresso04 \\`.

Zentrale Änderung zum vorherigen Beispielprojekt ist eine neue Klassendatei namens *DataGridViewEx.vb*, die die Erweiterung des DataGridView-Steuerelements in Form eines neuen Steuerelements namens *DataGridViewEx* implementiert.

Um dieses erweiterte Steuerelement mit der »richtig« funktionierenden **Eingabe**-Taste in Ihren zukünftigen Projekten zu verwenden, kopieren Sie diese Klassencoddatei einfach in Ihr Projektverzeichnis, lassen anschließend mit Klick auf das entsprechende Symbol im Projektmappenexplorer alle Dateien anzeigen, wählen die nun sichtbar gewordene Klassendatei aus und rufen aus dem Kontextmenü des Projektmappenexplorers den Befehl *Zum Projekt hinzufügen* auf. Nach dem ersten Kompilieren des Projekts erscheint das neue Steuerelement automatisch in der Toolbox, und Sie können es wie eine normale DataGridView verwenden.

Möchten Sie, dass dieses Steuerelement anstelle bestehender DataGridView-Steuerelemente verwendet wird, dann schließen Sie zunächst ein möglicherweise im Designer geöffnetes Formular, das die »betroffene« DataGridView beinhaltet, und öffnen Sie den Designer-Code dieses Formulars. Den Designer-Code erreichen Sie, indem Sie im Projektmappen-Explorer mit dem entsprechenden Symbol *alle Dateien anzeigen* lassen und anschließend den Zweig vor dem Formular erweitern, das das DataGridView-Steuerelement erhält. Öffnen Sie nun den Code der unter der eigentlichen Formular-

klasse stehenden Codedatei mit der Endung *.designer.vb*. Suchen Sie hier alle `System.Windows.Forms.DataGridView`-Referenzen (per `DataGridView` dürften es nicht mehr als 2 sein) und ersetzen Sie sie durch `DataGridViewEx`. Kompilieren Sie das Projekt anschließend neu.

Den dokumentierten Code des erweiterten Steuerelements finden Sie im Folgenden:

```
Public Class DataGridViewEx
    Inherits DataGridView

    'Verarbeitet Tasten, z. B. TAB, ESC, die EINGABETASTE und Pfeiltasten,
    'die zum Steuern von Dialogfeldern verwendet werden.
    Protected Overrides Function ProcessDialogKey(ByVal keyData As Keys) As Boolean

        'Nur die Keycodes interessieren, Rest wird ausmaskiert.
        Dim key As Keys = (keyData And Keys.KeyCode)

        'Eingabe-Taste gedrückt?
        If key = Keys.Enter Then

            'Ja, dann in Cursor-rechts umleiten, wo
            'Eingabe-Taste nochmal gesondert behandelt wird.
            Return Me.ProcessRightKey(keyData)
        End If

        'Andere Keyes bleiben nicht betroffen
        Return MyBase.ProcessDialogKey(keyData)
    End Function

    'Die alte Cursor-Rechts-Funktion wird überschattet, damit die
    'Polymorphie dieser Funktion an dieser Stelle unterbrochen wird.
    Public Shadows Function ProcessRightKey(ByVal keyData As Keys) As Boolean

        'Nur die Keycodes interessieren, Rest wird ausmaskiert.
        Dim key As Keys = (keyData And Keys.KeyCode)

        'Wenn es sich um die umgeleitete Eingabe-Taste handelte...
        If key = Keys.Enter Then

            'Feststellen, ob sich der Cursor am Ende der letzten Spalte befand, ...
            If MyBase.CurrentCell.ColumnIndex = (MyBase.ColumnCount - 1) Then
                '...dann den Cursor nach vorn und eine Zeile nach unten verschieben.
                MyBase.CurrentCell = MyBase.Rows(MyBase.CurrentCell.RowIndex + 1).Cells(0)
                Return True
            End If
        End If

        'Bei allen anderen Position reicht es aus,
        'das Standardverhalten von Cursor-rechts statt Eingabe zu verwenden.
        Return MyBase.ProcessRightKey(keyData)
    End Function
End Class
```

```

'Verarbeitet Tasten, die zum Navigieren in der DataGridView verwendet werden.
Protected Overrides Function ProcessDataGridViewKey(ByVal e As KeyEventArgs) As Boolean
    If e.KeyCode = Keys.Enter Then
        Return Me.ProcessRightKey(e.KeyData)
    End If
    Return MyBase.ProcessDataGridViewKey(e)
End Function
End Class

```

Entwickeln von MDI-Anwendungen

MDI ist die Abkürzung für *Multi Document Interface*, und das bedeutet etwas freier übersetzt: Benutzerschnittstelle für mehrere Dokumente. Was dieser Ausdruck meint: MDI-Anwendungen verfügen über ein Hauptformular (oder, um meinen Fachlektor zufrieden zu stellen, über ein *Hauptfenster*), das die generische Grundfunktionalität der gesamten Anwendung anbietet. Es bindet aber gleichzeitig für jedes eigentliche Dokument, das Sie mit ihm bearbeiten, mehrere Unterfenster (die aus dem Englischen übernommene Bezeichnung müsste eigentlich Kindfenster – von Child Window – heißen, hat sich aber nicht durchgesetzt) ein, die jeweils das eigentliche Dokument darstellen.

Was dabei genau ein »Dokument« darstellt, bestimmt die Anwendung. Das können Bitmaps, Zeichnungen, Kalkulationstabellen oder auch Textdokumente sein, etwa wie in der folgenden Abbildung zu sehen:

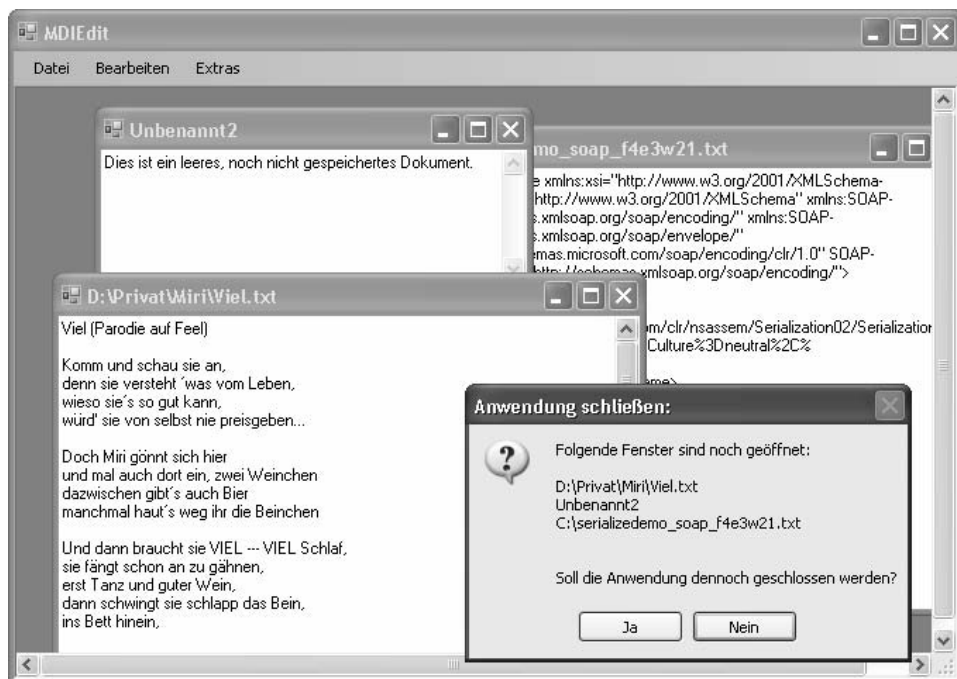


Abbildung 27.24: Eine typische MDI-Anwendung – hier zum Bearbeiten von Textdokumenten

Das eigentliche Darstellen eines untergeordneten Fensters in einem übergeordneten (eigentlich: *Elternfenster*, da von *Parent Window* abgeleitet), ist noch das kleinere Problem bei der Entwicklung bzw. Konzeption von MDI-Anwendungen. Mit insgesamt zwei Eigenschaften der Formalklasse sorgen Sie nämlich dafür, ein Formular einerseits zum Container für untergeordnete Fenster zu machen (`FormInstanz.IsMdiContainer = True`), andererseits dafür, dass eine neue Instanz eines Formulars zum untergeordneten Formular eines MDI-Hauptformulars wird (`UntergeordneteForm.MdiParent = Elternform`). Ein Aufruf der `Show`-Methode des untergeordneten Formulars genügt anschließend, um das Formular als Kindfenster einer MDI-Anwendung darzustellen.

Größere Probleme bilden bei der Entwicklung von MDI-Anwendungen die folgenden Punkte:

- Welche Funktionen werden von der Hauptanwendung, welche von den untergeordneten Formalklassen idealerweise übernommen?
- Wie lassen sich die Funktionsaufrufmöglichkeiten der beiden Instanzen visuell vereinen?

Lassen Sie uns beim Beispiel aus Abbildung 27.24 bleiben, um diese Problemstellungen zu erläutern.

Beim Entwickeln einer MDI-Texteditor-Anwendung wird recht schnell klar, dass globale Funktionen wie das Öffnen einer Datei, das Zur-Verfügung-Stellen eines globalen Optionsdialogs oder das Beenden des Programms in der Klasse des Hauptformulars implementiert werden müssen. Alle weiteren Funktionen wie das Speichern einer Datei, das Drucken, Suchen und Ersetzen im Text müssen von der jeweiligen Dokumentenklasse bereitgestellt werden, denn diese zuletzt genannten Funktionen beziehen sich immer auf die verschiedenen Dokumentinstanzen.

BEGLEITDATEIEN: Sie finden das Beispielprojekt für diesen Abschnitt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\MDIEdit\`.

Um das Beispiel einfach zu halten, beschränken wir uns auf folgende Funktionalität.

Für das globale Hauptformular:

- Öffnen einer Datei
- Erstellen einer neuen Datei
- Beenden des Programms (und Schließen aller noch geöffneten Dokumente mit Sicherheitsabfrage und Abbruchmöglichkeit)
- Aufruf eines Options-Dialogs.

Für jede Dokumentenklasse:

- Speichern der Datei
- Aufruf einer Suchfunktion.

In dieser Beispielanwendung gibt es nur eine einzige Dokumentenklasse. Denkbar wären aber auch zwei Dokumentenklassen mit unterschiedlicher Funktionalität – eine beispielsweise für das Bearbeiten reiner Textdateien, eine andere für das Bearbeiten von Richt-Text-Dokumenten, also solchen, die mit Formatierungen ausgestattet werden können.

Und hier ergibt sich nun das nächste Problem, nämlich das der unterschiedlichen Benutzeroberfläche. Eine Dokumentenklasse, die Richt-Text-Dokumente bearbeitet, muss beispielsweise in der Hauptanwendung ein *Format*-Menü bereitstellen; eine einfache Textdokumentenklasse benötigt dieses nicht.

Elegant wäre es, wenn jede Dokumentenklasse ihre eigene Benutzeroberfläche in Form von Menü und Toolbar mitbrächte, die dann mit der der Hauptanwendung verschmelzen würde, sobald man eine ihrer Instanz aktivierte.

Das Ergebnis sähe dann so aus, wie in den folgenden beiden Abbildungen zu sehen:



Abbildung 27.25: Solange es keine geöffneten Dokumente gibt (und damit keine instanziierten Dokumentklassen), steht nur der Grundpool an Funktionen zur Verfügung, der über die Menüstruktur zu erreichen ist. Dieser wird ...

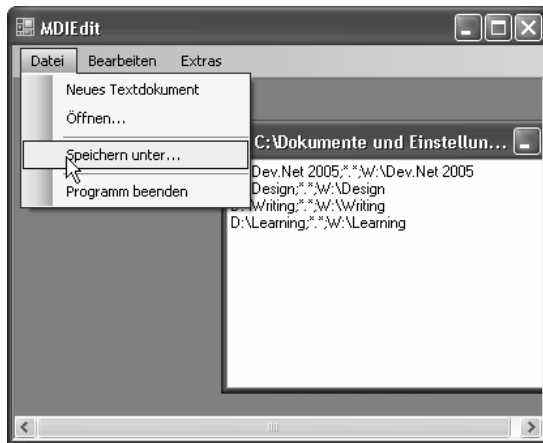


Abbildung 27.26: ... aber durch die jeweils aktive Dokumentklasseninstanz erweitert – die beiden Funktionssätze verschmelzen sozusagen.

Um diese Technik zu erreichen, bieten sowohl das `MenuStrip`-Steuerelement (mit dem Sie eine Menüstruktur im Fenster darstellen lassen können) als auch das `ToolStrip`-Steuerelement (mit dem Sie Toolbars anzeigen lassen können) besondere Eigenschaften an: Menüs (und auch Toolbars) von untergeordneten Formularen können dynamisch und nur durch Aktivierung zur Laufzeit mit denen von übergeordneten Formularen zusammengeführt werden.

Wichtig zu wissen sind dabei die folgenden Punkte:

- Sie erstellen ein Menü oder eine Toolbar für das Hauptfenster und müssen nur darauf achten, dass die standardmäßig gesetzte `AllowMerge`-Eigenschaft des jeweiligen Steuerelements (`MenuStrip`, `ToolStrip`) auch wirklich gesetzt ist.

- Sie erstellen ein Menü oder eine Toolbar für jedes untergeordnete Fenster und »tun« zunächst so, als wäre das Menü oder die Toolbar ausschließlich für dieses Fenster gedacht.
- Für untergeordnete Fenster bestimmen Sie anschließend über die MergeAction und MergeIndex-Eigenschaften, auf welche Weise und an welcher Stelle im übergeordneten Fenster die Elemente vereinigt werden sollen. Bei Elementen mit Unterelementen ergibt sich dabei eine besonders knifflige Problematik, wenn Elemente untergeordneter Ebenen zusammengeführt werden sollen. In diesem Fall müssen Sie die MergeIndex-Eigenschaft der Elemente der übergeordneten Ebene auf die gleiche Indexnummer des entsprechenden Elements im übergeordneten Fenster setzen, und als MergeAction den Wert MergeOnly einstellen. Möchten Sie aus dem untergeordneten Formular bestimmte Elemente übernehmen, legen Sie für diese als MergeIndex die Positionen fest, die der Position im Hauptfenster entsprechen, und für MergeAction bestimmen Sie den Wert Insert.
- Wenn das untergeordnete Formular zur Laufzeit seine sämtlichen Elemente (Menüs, Toolbars) mit dem übergeordneten verschmelzen lassen soll, dann legen Sie schon zur Entwurfszeit die Visible-Eigenschaft des Steuerelements auf False fest.

ACHTUNG: Aber aufgepasst dabei: Wenn Sie die Visible-Eigenschaft eines MenüStrip- oder eines ToolStrip-Steuerelements zur Entwurfszeit auf False setzen, dann verschwindet es auch im Designer. Das ist ungewöhnlich, aber logisch: Denn Ihnen muss beim Gestalten des Formulars ja schließlich der Platz für das unsichtbare Steuerelement »gutgeschrieben« werden. Das Problem: Wie verändern Sie die Elemente eines unsichtbaren Steuerelements, denn Sie können es ja nicht mehr anklicken!? Abbildung 27.27 hat die Lösung: Sie können es für den Augenblick der Bearbeitung wieder sichtbar machen, indem Sie es mithilfe des Eigenschaftenfensters (obere Aufklappliste) selektieren. Sobald das Steuerelement anschließend seinen Fokus wieder verliert, weil Sie ein anderes angeklickt haben, ist es auch schon wieder verschwunden.

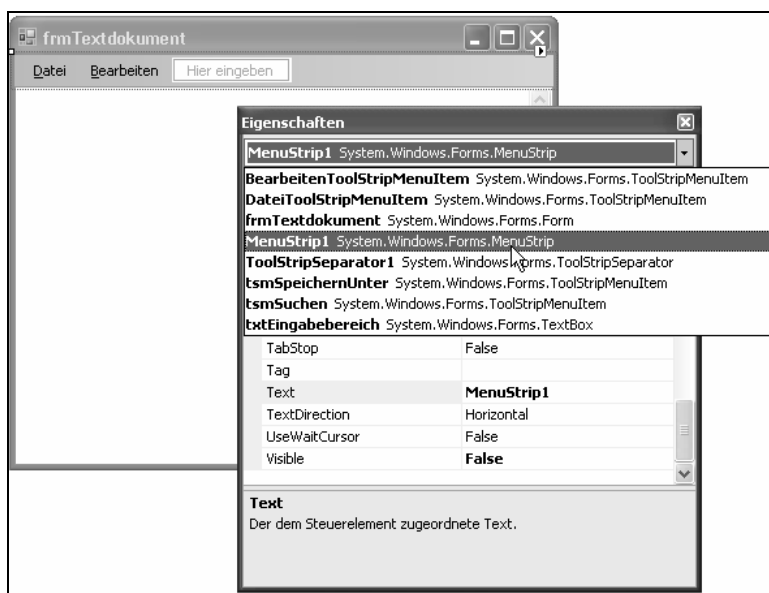


Abbildung 27.27: Ein *MenuStrip*- oder *ToolStrip*-Steuerelement, das Sie mit seiner *Visible*-Eigenschaft unsichtbar machen mussten, erreichen Sie zum Selektieren über das Eigenschaftenfenster

- Die Einstellungen von `MergeIndex` und `MergeAction` für das jeweilige Steuerelement im übergeordneten Formular bleiben völlig unberührt.

HINWEIS: Viele Entwickler sind der Meinung, dass korrelierende Indexnummern der `MergeIndex`-Eigenschaften der Steuerelemente im untergeordneten und übergeordneten Fenster die Korrelation bestimmen. Das ist falsch. Die `MergeIndex`-Nummer des Steuerelements des untergeordneten Formulars bezieht sich immer nur auf die Nummer schon *vorhandener* Elemente von Steuerelementen des übergeordneten Formulars, und nur auf *gleicher Ebene*!

Und das bedeutet: Wenn das Menü des untergeordneten Steuerelements die Einträge:

- *Datei*
- *Datei / Trennlinie*
- *Datei / Speichern*
- *Bearbeiten*
- *Bearbeiten / Suche*

aufweist, und das übergeordnete Formular bereits die Menüeinträge

- *Datei* (0)
- *Datei / Neues Dokument* (0) (0)
- *Datei / Dokument öffnen* (0) (1)
- *Datei / Trennlinie* (0) (2)
- *Datei / Programm beenden* (0) (3)
- *Extra* (1)
- *Extras / Optionen* (1) (0) enthält,

so ist die `MergeAction`-Eigenschaft für *Datei* des untergeordneten Steuerelements auf `MergeOnly` und die `MergeIndex`-Eigenschaft auf 0 zu setzen. Der erste Menüeintrag auf dieser Ebene (*Datei*) im untergeordneten Formular soll nämlich nur mit dem ersten Menüeintrag gleicher Ebene im übergeordneten Formular verschmolzen werden.

Für *Datei/Trennlinie* und *Datei/Speichern* des untergeordneten Formulars setzen Sie die `MergeAction`-Eigenschaft auf `Insert`, weil diese Menüpunkte im Menü des übergeordneten Formulars eingefügt werden sollen, und zwar hinter *Datei/Dokument öffnen* und vor *Datei/Trennlinie*. Da bei `Insert` immer *vor* einem Element eingefügt wird, und oben angegebene Nummerierung im übergeordneten Formular gilt, bestimmen Sie für *Datei/Trennlinie* den Wert 2 als `MergeIndex` für *Datei speichern*.

Für *Bearbeiten* bestimmen Sie aus gleichen Gründen 1 für `MergeIndex` und `Insert` für `MergeAction`, weil das komplette *Bearbeiten*-Menü dann vor dem *Extras*-Menü eingefügt wird.

Und damit ist die Einrichtung der Benutzeroberfläche auch schon vollbracht! Sie können sich nun um die programmtechnische Umsetzung der eigentlichen Funktionen kümmern. Die ist in unserem Beispiel vergleichsweise simpel, wie die beiden folgenden Listings des Hauptformulars und das des Formulars zeigen, das die Dokumentenklasse in der Beispielanwendung ausmacht.

Listing von frmMain.vb (übergeordnetes Hauptformular)

```
Public Class frmMain

    'Der "Unbenannt"-Dateinamenzähler.
    Private myUnbekanntesDokumentZähler As Integer

    Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
        MyBase.OnLoad(e)

        'Den "Unbenannt"-Dateinamenzähler beim Laden des Formulars initialisieren.
        myUnbekanntesDokumentZähler = 1
    End Sub

    'Wird aufgerufen, wenn der Anwender aus dem Menü Datei den Menüpunkt Neues Dokument auswählt.
    Private Sub NeuesDokument_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        tsmNeuesDokument.Click
        'Neue Instanz des MDI-Dokuments erstellen.
        Dim locFrmTextdokument As New frmTextdokument

        'Dieser Instanz mitteilen, dass es ein MDI-untergeordnetes Fenster werden soll.
        locFrmTextdokument.MdiParent = Me

        'Die Funktion zum Anzeigen des neuen untergeordneten Fensters aufrufen.
        locFrmTextdokument.NeuesTextdokument(myUnbekanntesDokumentZähler)

        'Den "Unbenannt"-Dateinamenzähler um eins erhöhen.
        myUnbekanntesDokumentZähler += 1
    End Sub

    'Wird aufgerufen, wenn der Anwender aus dem Menü Datei den Menüpunkt Öffnen auswählt.
    Private Sub tsmÖffnen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        tsmÖffnen.Click

        Dim dateiÖffnenDialog As New OpenFileDialog
        With dateiÖffnenDialog
            .CheckFileExists = True
            .CheckPathExists = True
            .DefaultExt = "*.txt"
            .Filter = "Textdateien" & " (" & "*.txt" & ")|" & "*.txt" & "|Alle Dateien (*.*)|*.*"

            'Dateinamen der zu öffnenden Textdatei ermitteln.
            Dim dialogErgebnis As DialogResult = .ShowDialog
            If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
                Exit Sub
            End If

            'Neue Instanz des MDI-Dokuments erstellen.
            Dim locFrmTextdokument As New frmTextdokument

            'Dieser Instanz mitteilen, dass es ein MDI-untergeordnetes Fenster werden soll.
            locFrmTextdokument.MdiParent = Me
        End With
    End Sub
End Class
```

```

        'Die Funktion zum Laden des Textes und Anzeigen des untergeordneten Fensters aufrufen.
        locFrmTextdokument.TextdokumentÖffnen(.FileName)
    End With
End Sub

'Wird aufgerufen, wenn der Anwender aus dem Menü Extras den Menüpunkt Optionen auswählt.
Private Sub tsmOptionen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmOptionen.Click
    MessageBox.Show("Hier für die Optionen-Funktion implementiert werden")
End Sub

'Wird aufgerufen, wenn der Anwender aus dem Menü Datei den Menüpunkt Programm beenden auswählt.
Private Sub tsmProgrammBeenden_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmProgrammBeenden.Click
    Me.Close()
End Sub

'Wird beim Schließen des Formulars aufgerufen
Protected Overrides Sub OnFormClosing(ByVal e As System.Windows.Forms.FormClosingEventArgs)
    MyBase.OnFormClosing(e)

    'Gibt es untergeordnete Fenster?
    If Me.MdiChildren IsNot Nothing AndAlso Me.MdiChildren.Length > 0 Then
        Dim locFensterTitel As String = ""

        'Die Fenstertitel der untergeordneten Fenster ermitteln
        For Each locForm As Form In Me.MdiChildren
            locFensterTitel &= locForm.Text & vbCrLf
        Next

        'Warnhinweis mit der Liste der noch offenen Fenster.
        Dim locDr As DialogResult = MessageBox.Show("Folgende Fenster sind noch geöffnet:" & _
            vbCrLf & vbCrLf & locFensterTitel & _
            vbCrLf & vbCrLf & _
            "Soll die Anwendung dennoch geschlossen werden?", _
            "Anwendung schließen:", MessageBoxButtons.YesNo, _
            MessageBoxIcon.Question, MessageBoxDefaultButton.Button2)

        If locDr = Windows.Forms.DialogResult.No Then
            'Der Anwender hat Nein gesagt!
            'Das Fenster bleibt offen.
            e.Cancel = True
        End If
    End If
End Sub
End Class

```

Listung von frmTextdokument.vb (untergeordnete Dokumentenklasse)

```
Public Class frmTextdokument
    ''' <summary>
    ''' Zeigt dieses Formular mit einem leeren Textfeld an
    ''' </summary>
    ''' <remarks></remarks>
    Public Sub NeuesTextdokument(ByVal DokumentNr As Integer)
        txtEingabebereich.Text = Nothing

        'Dokumentnamen "Unbenannt_X" im Titel anzeigen.
        Me.Text = "Unbenannt" & DokumentNr

        'Formular nicht-modal anzeigen.
        Me.Show()
    End Sub

    ''' <summary>
    ''' Zeigt dieses Formular an, und lädt eine Textdatei in den Eingabebereich.
    ''' </summary>
    ''' <param name="Dateiname"></param>
    ''' <remarks></remarks>
    Public Sub TextdokumentÖffnen(ByVal Dateiname As String)

        'Textdatei in den Eingabebereich laden.
        txtEingabebereich.Text = My.Computer.FileSystem.ReadAllText(Dateiname)

        'Dateinamen als Dokumentnamen im Titel anzeigen.
        Me.Text = Dateiname

        'Formular nicht-modal anzeigen.
        Me.Show()
    End Sub

    'Wird aufgerufen, wenn der Anwender aus dem Menü Datei den Menüpunkt Speichern unter auswählt.
    Private Sub tsmSpeichernUnter_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles tsmSpeichernUnter.Click
        Dim dateiSpeichernDialog As New SaveFileDialog
        With dateiSpeichernDialog

            'Pfad muss vorhanden sein!
            .CheckPathExists = True

            'Warnen, wenn die Datei schon existiert!
            .OverwritePrompt = True

            'Dateinamenerweiterungen einrichten:
            .DefaultExt = "*.txt"
            .Filter = "Textdateien" & " (" & "*.txt" & ")|" & "*.txt" & "|Alle Dateien (*.*)|*.*"
            Dim dialogErgebnis As DialogResult = .ShowDialog
            If dialogErgebnis = Windows.Forms.DialogResult.Cancel Then
                Exit Sub
            End If
        End With
    End Sub
End Class
```

```

        'Textdatei speichern
        My.Computer.FileSystem.WriteAllText(.FileName, txtEingabebereich.Text, False)

        'Fenstertitel ist Dateiname
        Me.Text = .FileName
    End With
End Sub

'Wird aufgerufen, wenn der Anwender im Menü Bearbeiten auf Suchen klickt.
Private Sub tsmSuchen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles tsmSuchen.Click
    MessageBox.Show("Hier würde die Suchenfunktion implementiert werden!")
End Sub
End Class

```

Vererben von Formularen

Da Formulare nichts anderes sind als Klassen, können Sie sie natürlich auch vererben. Im Prinzip machen Sie das automatisch, sobald Sie eine neue Windows Forms-Anwendung anlegen: Das Formular, das Ihrer Anwendung automatisch hinzugefügt wird, erbt aus der Basisklasse *Form*.

Allerdings gibt es eine besondere Möglichkeit, Formulare zu vererben; in Visual Studio .NET spricht man dabei von der visuellen Vererbung von Formularen. Die IDE bezeichnet solche Formulare schlicht als *geerbte Formulare*.

BEGLEITDATEIEN: Wenn Sie, anstatt die folgenden Schritte selbst durchzuführen, lieber auf das fertige Projekt zurückgreifen wollen: Sie finden Sie es unter `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap27\FormVererbung\`.

Die Vorgehensweise dazu ist denkbar einfach:

- Erstellen Sie ein neues *Windows Forms*-Projekt, beispielsweise unter dem Namen *FormVererbung*.
- Platzieren Sie ein paar *TextBox*-Steuerelemente auf dem Formular, etwa wie in Abbildung 27.28 zu sehen.

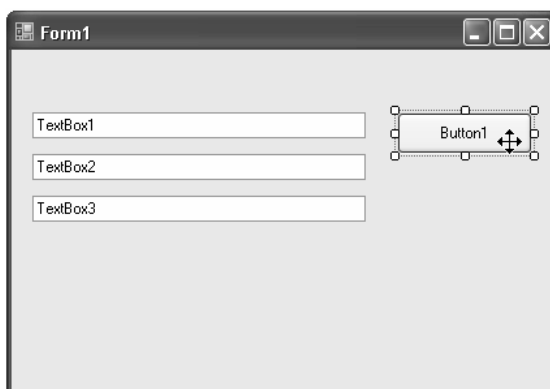


Abbildung 27.28: Dieses Formular dient als Basis für die visuelle Vererbung

- Doppelklicken Sie auf die Schaltfläche *OK*, um den Codeeditor zu öffnen, und fügen Sie den folgenden Programmcode ein:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
    Handles Button1.Click

    Dim locString As String

    For Each locControl As Control In Me.Controls
        If TypeOf locControl Is TextBox Then
            locString += "Inhalt von " + locControl.Name + ": "
            locString += DirectCast(locControl, TextBox).Text + vbNewLine
        End If
    Next
    locString = "Inhalt der TextBox-Komponenten im Formular:" + _
        vbNewLine + vbNewLine + locString
    MessageBox.Show(locString, "Hinweis:")
End Sub
```

- Starten Sie das Programm anschließend, geben Sie ein paar Zeichen in das *TextBox*-Steuerelement ein, und beobachten Sie, welche Aktion ein Mausklick auf *OK* anschließend auslöst (siehe Abbildung 27.29).

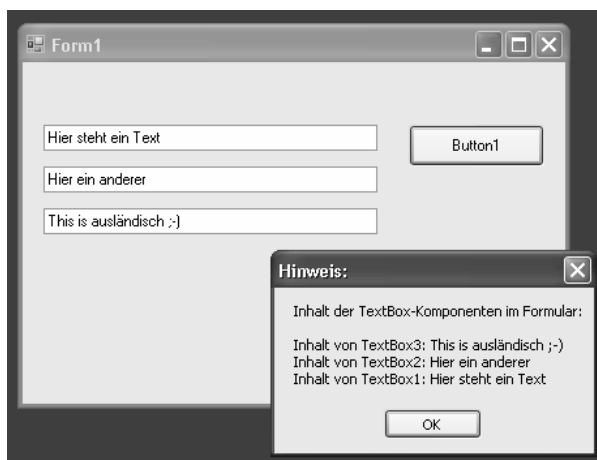


Abbildung 27.29: Formulare kommen übrigens auch ohne Steuerelemente-Arrays aus. Das hat zwar nichts mit dem Thema dieses Abschnitts zu tun, ergibt sich aber durch das Demo für die visuelle Vererbung so ganz nebenbei.

ControlCollection vs. Steuerelemente-Array aus VB6

Auch wenn es nichts mit dem eigentlichen Thema dieses Abschnitts zu tun hat, so zeigt dieses Beispielprogramm dennoch auf einfache Weise, wie unnötig Steuerelemente-Arrays in .NET sind und wieso es sie aus diesem Grund auch nicht mehr gibt. Denn auch die *Controls*-Auflistung, die vordergründig dazu dient, das Formular mit Steuerelementen auszustatten, lässt sich wie jede andere Auflistung, die über einen Enumerator verfügt, mit *For/Each* durchlaufen. Anhand des Typen, den Sie wie im Beispiel mit *Type Of* ermitteln können, haben Sie die Möglichkeit, durch eine entsprechende Typkonvertierung auf die gewünschte Eigenschaft des Steuerelements zuzugreifen. ▶

Kleiner Tipp am Rande: Falls Sie weitere, spezifische Informationen in einem Steuerelement abspeichern möchten, die nur der Identifizierung dienen, verwenden Sie die Tag⁶-Eigenschaft, die jedes auf Control basierende Steuerelement zur Verfügung stellt. Anders als in Visual Basic 6.0 können Sie in der Tag-Eigenschaft nicht nur Strings, sondern beliebige Objekte speichern.

Für die eigentliche visuelle Vererbung dieses Formulars beenden Sie bitte das Programm, indem Sie auf die Schließschaltfläche des Formulars klicken. Um ein geerbtes Formular dem Projekt hinzuzufügen, verfahren Sie folgendermaßen:

- Fahren Sie mit der Maus in den Projektmappen-Explorer, klicken Sie mit der rechten Maustaste über dem Projektnamen, und wählen Sie aus dem Kontextmenü, das sich jetzt öffnet, den Eintrag *Hinzufügen und geerbtes Formular hinzufügen*.
- Geben Sie im Dialog, der jetzt erscheint, *GeerbtesFormular* als neuen Formularnamen ein.
- Klicken Sie anschließend auf *Öffnen*.

Visual Studio zeigt Ihnen anschließend einen Dialog, wie Sie ihn auch in Abbildung 27.30 sehen können. In diesem Dialog werden alle Formulare der Projektmappe angezeigt, die sie visuell vererben können. Wählen Sie für dieses Beispiel den einzig vorhandenen Eintrag aus, und klicken Sie abschließend auf *OK*.

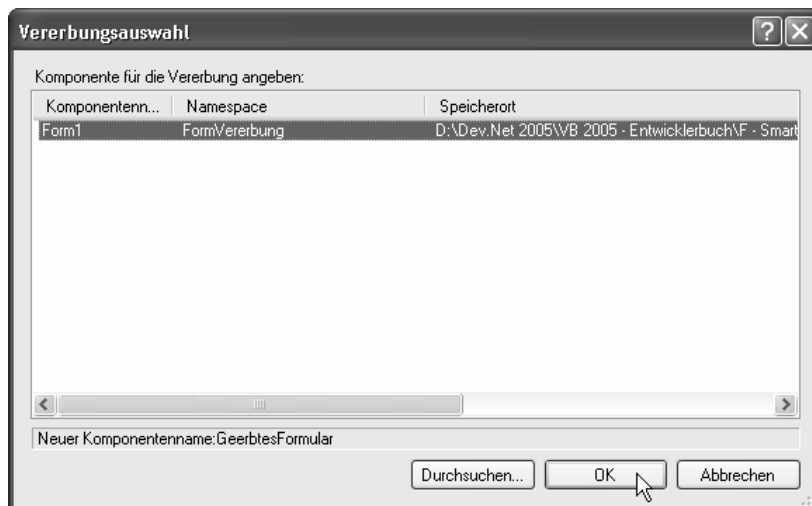


Abbildung 27.30: In dieser Liste, die alle Formulare der Projektmappe zeigt, wählen Sie das Formular, aus dem das neue Formular abgeleitet werden soll

Wenn Sie diesen Vorgang ausgeführt haben, gibt es anschließend einen zweiten Dialog in Ihrer Projektmappe mit dem Namen *GeerbtesFormular.vb*. Klicken Sie ihn doppelt an, um ihn darzustellen und sich davon zu überzeugen, dass er seine äußerliche Ähnlichkeit vom Vater-Dialog geerbt hat (siehe Abbildung 27.31).

⁶ Engl. Tag (ausgesprochen: »Tähg«), auf deutsch etwa: *Markierung, Kennzeichnung*.

Wenn Sie Ihre ersten Experimente mit dem vererbten Steuerelement unternehmen, wird Ihnen eines gleich auffallen: Zwar sind alle Steuerelemente im Formular vorhanden, doch lassen sie sich nicht verändern. Weder können Sie ihre Eigenschaften editieren, noch lassen sie sich umpositionieren oder in ihrer Größe verändern (was letzten Endes auch nichts anderes ist als ein Verändern der Eigenschaften).

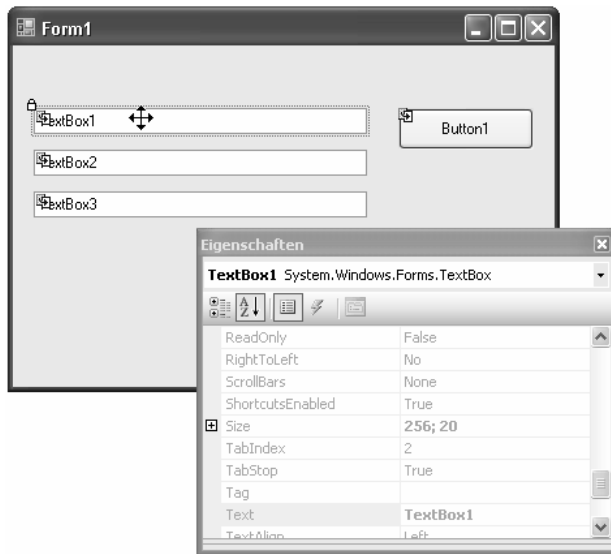


Abbildung 27.31: Die Steuerelemente des Basisdialogs sind zwar alle vorhanden, aber ihre Eigenschaften lassen sich in diesem Zustand nicht verändern, da sie standardmäßig als *Friend* deklariert, damit nicht überschreibbar und somit aus dem vererbten Formular heraus nicht manipulierbar sind

Fürs Erste behalten Sie diese Erkenntnis einfach nur im Hinterkopf – ich werde später darauf zurückkommen.

Lassen Sie uns vorerst herausfinden, wie wir aus dem Vererben des Formulars nun einen Nutzen ziehen können. Zu diesem Zweck fügen Sie unter den vorhandenen TextBox-Steuerelementen zwei weitere ein – denn das ist durchaus möglich!

Öffnen Sie anschließend die Projekteigenschaften (rechte Maustaste für das Kontextmenü im Projektmappen-Explorer über dem Projektnamen), und wählen Sie als Startobjekt das neue, vererbte Formular *GeerbtesFormular* aus. Starten Sie das Programm anschließend.

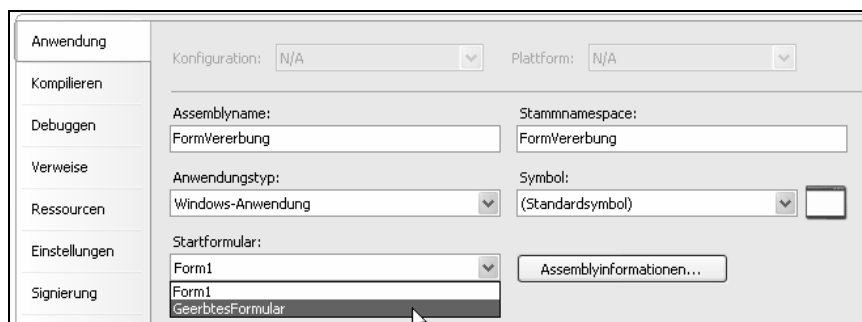


Abbildung 27.32: Legen Sie hier das geerbte Formular als neues Startformular fest

Geben Sie in alle Eingabefelder einen beliebigen Text ein, und beobachten Sie, was geschieht, wenn Sie auf die Schaltfläche klicken (siehe Abbildung 27.33).

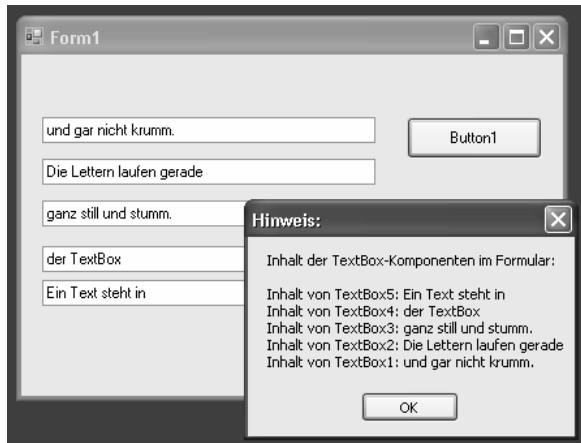


Abbildung 27.33: Keine Zeile Code ist hinzugekommen, und dennoch funktioniert das Programm automatisch auch mit den neu hinzugefügten Komponenten im abgeleiteten Formular!

Überrascht? Sie haben nicht eine einzige Zeile am Code geändert und dennoch – dank Polymorphie – hat der ursprünglich in *Form1* implementierte Code auch in dieser Ableitung seine volle Gültigkeit.

Ein erster Blick auf den Designer-Code eines Formulars

Vielleicht sind Sie sogar ein weiteres Mal überrascht, wenn Sie sich jetzt, nachdem Sie das Programm beendet haben, auf die Suche nach dem Code in der abgeleiteten Klasse begeben. Wechseln Sie mit dem Projektmappen-Explorer in die Codeansicht von *GeerbtesFormular.vb*, und bestaunen Sie, was der Visual Studio-Designer aus dem Code gemacht hat:

```
Public Class GeerbtesFormular
```

```
End Class
```

Nichts, absolut gar nichts an zusätzlichem Code ist hier zu sehen. Im Übrigen nicht einmal der Hinweis auf eine Vererbung?!

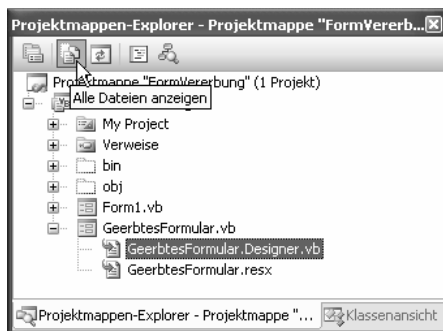


Abbildung 27.34: Nur mit Klick auf dieses Symbol können Sie alle Dateien eines Projektes anzeigen lassen. Bei mehreren Projekten in einer Projektmappe müssen Sie diese Einstellung für *jedes* Projekt einer Projektmappe wiederholen.

Des Rätsels Lösung liegt darin, dass der Designer den Designercode in Visual Basic 2005 in einer besonderen Codedatei versteckt, die Sie normalerweise gar nicht zu Gesicht bekommen.

Um Sie dennoch zu sehen, klicken Sie im Projektmappenexplorer auf das Symbol *Alle Dateien anzeigen* (siehe Abbildung 27.34). Erst dann können Sie den Zweig vor jedem Formular aufklappen, und Sie werden feststellen, dass jedes in Visual Basic 2005 erstellte Formular eigentlich mindestens aus zwei Dateien besteht. Der Designer-Code, also der Code, der zum einen vom Designer erstellt wurde und der zum anderen dafür zuständig ist, das alle Steuerelemente des Formulars rechtzeitig instanziiert und auf dem Formular dargestellt werden, befindet sich in der Datei mit der Endung *.Designer.vb*.

```
<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Partial Class GeerbtesFormular
    Inherits FormVererbung.Form1

    'Das Formular überschreibt den Löschvorgang, um die Komponentenliste zu bereinigen.
    <System.Diagnostics.DebuggerNonUserCode()> _
    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing AndAlso components IsNot Nothing Then
            components.Dispose()
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Wird vom Windows Form-Designer benötigt.
    Private components As System.ComponentModel.IContainer

    'Hinweis: Die folgende Prozedur ist für den Windows Form-Designer erforderlich.
    'Das Bearbeiten ist mit dem Windows Form-Designer möglich.
    'Das Bearbeiten mit dem Codeeditor ist nicht möglich.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        Me.TextBox4 = New System.Windows.Forms.TextBox
        Me.TextBox5 = New System.Windows.Forms.TextBox
        Me.SuspendLayout()
        '
        'TextBox4
        '
        Me.TextBox4.Location = New System.Drawing.Point(16, 147)
        Me.TextBox4.Name = "TextBox4"
        Me.TextBox4.Size = New System.Drawing.Size(256, 20)
        Me.TextBox4.TabIndex = 5
        '
        'TextBox5
        '
        Me.TextBox5.Location = New System.Drawing.Point(16, 173)
        Me.TextBox5.Name = "TextBox5"
        Me.TextBox5.Size = New System.Drawing.Size(256, 20)
        Me.TextBox5.TabIndex = 6
        '
        'GeerbtesFormular
        '
        Me.ClientSize = New System.Drawing.Size(416, 262)
    End Sub
End Class
```

```

Me.Controls.Add(Me.TextBox5)
Me.Controls.Add(Me.TextBox4)
Me.Name = "GeerbtesFormular"
Me.Controls.SetChildIndex(Me.TextBox4, 0)
Me.Controls.SetChildIndex(Me.TextBox5, 0)
Me.ResumeLayout(False)
Me.PerformLayout()

End Sub
Friend WithEvents TextBox4 As System.Windows.Forms.TextBox
Friend WithEvents TextBox5 As System.Windows.Forms.TextBox

```

End Class

Mithilfe des Konzepts der partiellen Klassen (mehr zum Schlüsselwort `Partial` finden Sie in ► Kapitel 6) kann der Klassencode den manchmal störenden Designer Code in einer eigene Codedatei ableiten. Die eigentliche Vererbungsanweisung befindet sich, wie Sie sich anhand der ersten fett hervorgehobenen Zeilen dieses Listings selbst überzeugen können, ebenfalls in dieser Codedatei.

Ein Formular ist ja nichts weiter als eine ganz normale Klasse, und wenn Sie sie aus einer Basisklasse ableiten, erbt sie alle Eigenschaften dieser – und damit natürlich auch die Ereignisbehandlungsmethode des `Click`-Ereignisses.

Das abgeleitete Formular verfügt über einen Unterschied zur Basisklasse, den allerdings nicht wir, sondern der Designer als Code in das Listing eingefügt hat:

Es sind dies, wie Sie hier sehen können, die zusätzlichen Definitionen der beiden `TextBox`-Steuerelemente. Was passiert nun genau, wenn Sie das Programm starten und im vererbten Formular auf die Schaltfläche klicken? Die nachstehende Abfolgenbeschreibung macht das deutlich:

- Wenn Sie das Programm starten, wird der Konstruktor der abgeleiteten Klasse aufgerufen. Dieser ruft zunächst den Konstruktor der Basisklasse auf, der seinerseits `InitializeComponent` der Basisklasse aufruft. Damit sind die ursprünglichen drei `TextBox`-Steuerelemente im Formular vorhanden.
- Anschließend ruft der Konstruktor `InitializeComponent` seiner eigenen Klasse auf. Da es für jedes Formular nur eine `Controls`-Auflistung gibt (sie enthält die Steuerelemente eines Formulars und ist aus der Basisklasse entstanden), werden die beiden neuen `TextBox`-Steuerelemente zu den bereits vorhandenen in der Auflistung hinzugefügt.
- Klickt der Anwender zur Laufzeit auf die Schaltfläche, werden alle Texte aus den in der `Controls`-Auflistung vorhandenen `TextBox`-Steuerelementen ausgelesen – dazu gehören jetzt auch die beiden neuen `TextBox`-Steuerelemente.

Das Ergebnis: Die Texte aller fünf Steuerelemente werden im Nachrichtenfeld dargestellt – Polymorphie ist eine feine Sache, finden Sie nicht?

Modifizierer von Steuerelementen in geerbten Formularen

Wenn Sie sich das zuvor abgedruckte Listing des Designer Codes genau anschauen, dann werden Sie feststellen, dass die Variablen, die die TextBox-Steuerelemente instanzieren, mit dem Friend-Modifizierer gekennzeichnet sind. Nun eignet sich der Friend-Modifizierer genau wie Protected, Public oder Protected Friend gleichermaßen, einer geerbten Klasse in der gleichen Assembly (in der gleichen DLL oder der gleichen EXE-Datei) Zugriff auf derart deklarierte Member der Basisklasse zu verschaffen. Dennoch sieht es so aus, als wenn dem Designer der Zugriff verwehrt wäre – denn Sie können Steuerelemente, die als Friend deklariert sind, offensichtlich nicht in der abgeleiteten Klasse verändern. Das liegt daran, dass der Designer, der die Klasse zur Darstellung auf seine eigene Weise verwendet, sich in einer anderen Assembly befindet als das Formular, das Sie ableiten: Er instanziiert es nicht nur, sondern leitet es auf seiner Basis neu ab – durch den Friend-Zugriffsmoifizierer kann er dann natürlich nicht mehr auf die Eigenschaften der Steuerelemente (die nunmehr versteckte Member der Ursprungsklasse sind) zugreifen. Anders ist das, wenn Sie die Modifiers-Eigenschaft eines Steuerelements in Protected, Protected Friend oder Public ändern. Jetzt kann der Designer auch auf die so definierten Steuerelemente des abgeleiteten Formulars zugreifen, und Sie können die Eigenschaften dieser Steuerelemente verändern und sie damit auch beispielsweise positionstechnisch verändern.

Probieren Sie es aus:

- Klicken Sie auf die Registerkarte *Form1.vb [Entwurf]*, um das Basisformular in der Entwurfsansicht anzeigen zu lassen.
- Klicken Sie die erste TextBox an, und ändern Sie im Eigenschaftenfenster ihre Modifiers-Eigenschaft auf Protected. Sobald Sie diese Änderung vorgenommen haben, zeigt Ihnen die Aufgabenliste einen Hinweis an, etwa wie in Abbildung 27.35 zu sehen.

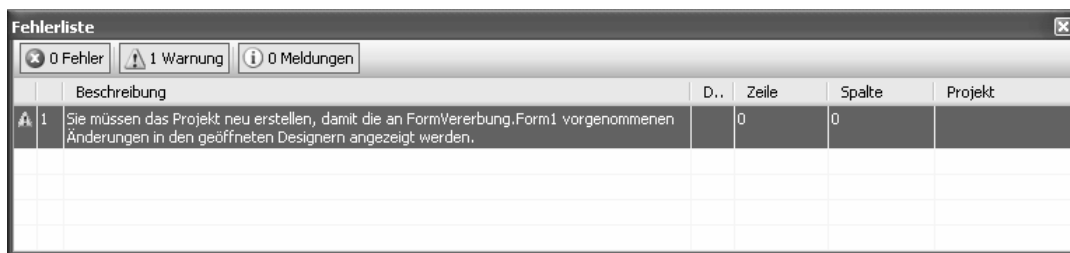


Abbildung 27.35: Wenn Sie Änderungen am Basisformular vorgenommen haben, müssen Sie die Anwendung neu erstellen, damit sich die Änderungen auf die Ableitungen auswirken

- Erstellen Sie die Anwendung neu, indem Sie aus dem Menü *Erstellen* den Menüpunkt *Projektmappe neu erstellen* wählen.
- Lassen Sie anschließend den Designer zum vererbten Form *GeerbtesForm.vb [Entwurf]* anzeigen.

Sie sehen, dass Sie die erste TextBox jetzt nach Belieben verändern können. Sowohl die Position lässt sich beliebig anpassen als auch andere Eigenschaften der TextBox. Der Name des Steuerelements ist natürlich nach wie vor unveränderlich.