28 Im Motorraum von Formularen und Steuerelementen

821 Über das Vererben von Form und die Geheimnisse des Designer-Codes

827 Ereignisbehandlung von Formularen und Steuerelementen

834 Wer oder was löst welche Formular- bzw. Steuerelementereignisse wann aus?

Nachdem Sie Ihre ersten Anwendungen mit Visual Basic erstellt haben, gehen Ihnen bestimmte Prinzipien beim Verdrahten von Benutzeroberflächen mit den eigentlichen Funktionalitäten Ihrer Programme in Fleisch und Blut über. So wissen Sie nach einer Weile einfach, wie Sie dafür zu sorgen haben, dass beim Klicken einer Schaltfläche eine entsprechende Ereignisbehandlungsmethode ausgeführt werden soll.

Doch was passiert eigentlich genau, wenn der Benutzer ein Element bedient? In welcher Reihenfolge treten welche Ereignisse auf, wenn ein Formular oder ein Steuerelement dargestellt wird? Und wie gliedern sich .NET-Framework-Windows-Anwendungen in das »alte« Betriebssystem (also vor Windows Vista) eigentlich ein? Dieses Kapitel möchte ein wenig Licht ins Dunkel bringen und auf diese Fragen Antworten geben.

BEGLEITDATEIEN: Das Beispielprojekt für die folgenden Abschnitt finden Sie im Verzeichnis *\VB 2005 - Entwicklerbuch\G - SmartClient\Kap28\PictureViewer*.

Über das Vererben von Form und die Geheimnisse des Designer-Codes

Immer wenn Sie Ihrem Projekt ein neues Windows-Formular hinzufügen, legt Visual Basic nicht nur eine einfache Klassendatei an, die den Formularaufbau enthält. Die Wahrheit ist: Wenn Sie ein neues Formular anlegen, und es im Codeeditor öffnen, dann sehen Sie zunächst außer dem Klassenrumpf erst einmal gar nichts. Im Beispielprojekt gibt es das Formular LeeresFormular.vb, das, wenn Sie es im Codeeditor öffnen, den folgenden Code bereits hält:

```
Public Class LeeresFormular

End Class
```

Öffnen Sie das Formular jedoch mit dem Designer, stellen Sie fest, dass es bereits eine ganze Reihe von Steuerelementen gibt. Nur: Wo ist denn nur die Information für diese Steuerelemente untergebracht?

Gerade für Visual Basic .NET-Umsteiger, die ihre ersten Gehversuche mit Visual Basic 2005 und Windows Forms-Anwendungen machen, sieht diese Tatsache auf den ersten Blick geradezu erschreckend aus, weil die für .NET-Programmiersprachen postulierte »OOP-Konsequenz« gebrochen zu sein scheint.

Es gibt weder einen Hinweis darauf, dass die Grundfunktionalität des Formulars sich aus irgendeiner im .NET-Framework schon vorhandenen Klasse ableitet, noch, an welcher Stelle die Platzierung der Steuerelemente im Formular zur Laufzeit erfolgt.

Wenn Sie das vorherige Kapitel aufmerksam studiert haben, wissen Sie es bereits:

Des Rätsels Lösung liegt darin, dass der spezielle Code zur eigentlichen Darstellung der Steuerelemente eines jeden Formulars in Visual Basic 2005 in einer besonderen Codedatei versteckt ist, 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 28.1).

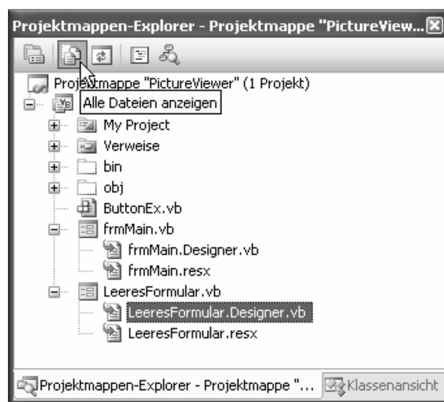

Abbildung 28.1: Nur mit Klick auf dieses Symbol können Sie alle Dateien eines Projektes anzeigen lassen, und damit auch Zugriff auf den Designer-Code eines Formulars nehmen. Bei mehreren Projekten in einer Projektmappe müssen Sie diese Einstellung für *jedes* Projekt einer Projektmappe wiederholen.

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, dass alle Steuerelemente des Formulars rechtzeitig instanziert und auf dem Formular dargestellt werden, befindet sich in der Datei mit der Endung *.Designer.vb*.

Und ohne dass der Entwickler im Formular *LeeresFormular.vb* nur eine einzige Zeile selbst programmiert hat, beträgt der Umfang Ihrer Formularklasse zu diesem Zeitpunkt schon weit über 100 Zeilen, denn:

In der Datei *LeeresFormular.Designer.vb* befinden sich die Codezeilen, die der Windows-Designer (eigentlich: *die* Designer, denn jedes Steuerelement verfügt streng genommen über seinen eigenen – siehe dazu den entsprechenden grauen Kasten in ▶ Kapitel 3) beim Anlegen des Projektes und bei jedem Hinzufügen der Komponenten und dem dazugehörigen Einstellen der Eigenschaften produziert hat.

Auch die Angabe der Klassenableitung mit `Inherits` steht in diesem Designer-Teil des Formularcodes – und mit dieser Technik wird zweierlei erreicht:

- Das OOP-Konzept für Framework-Klassen ist auch in Visual Basic 2005 nicht verletzt.
- Dennoch ist die Codedatei, in der Sie Ihren eigenen programmtechnischen Teil zum Funktionieren des Formulars beitragen, sauber aufgeräumt und bleibt stets übersichtlich. Der Designer-Code steht buchstäblich völlig außen vor, und sie bekommen ihn nur zu Gesicht, wenn Sie es ausdrücklich wollen.

Es ist aber dennoch wichtig, dass Sie verstehen, was beim Anzeigen eines Formulars zur Laufzeit passiert, denn nur dann haben Sie die Möglichkeit, ins Geschehen einzugreifen, wenn mal etwas nicht so funktioniert, wie Sie es sich gedacht haben, oder wenn Sie Erweiterungen implementieren müssen, die über die Standardimplementierung hinausgehen.

Die folgenden Abschnitte beschreiben daher die Funktionsweise des Designer-Codes am Beispiel.

Geburt und Tod eines Formulars – New und Dispose

Wenn Sie Ihrem Formular ein neues Formular hinzufügen, dann bekommt es automatisch vom Designer und eine `Dispose`-Methode verpasst.

Was in Visual Basic 2005 erstellten Formularen allerdings völlig fehlt, ist ein Konstruktor, und – nehmen wir die Tatsache vorweg, dass gerade der fehlende Konstruktor dafür zuständig ist, die Methode InitializeComponent aufzurufen, mit dem das eigentliche Einrichten und Platzieren der Steuerelemente des Formulars erfolgt – es stellt sich die Frage, wer oder was ruft InitializeComponent überhaupt auf?

Eine offensichtliche Antwort könnte lauten: Der Konstruktor der Basisklasse, denn der wird, wie Sie wissen, wenn Sie den Klassenteil dieses Buches aufmerksam studiert haben, in vererbten Klassen implizit aufgerufen. Doch das ist nicht der Fall.

Die Wahrheit ist: Der Visual Basic-Compiler kompiliert »virtuellen« Code beim Kompilieren in die Klasse hinein. Würden Sie das Kompilat des Beispielprogramms mit einem geeigneten Tool wieder zurück in seinen Visual Basic-Quelltext übersetzen, dann ergäbe sich nämlich Folgendes:

```
<DebuggerNonUserCode> _
Public Sub New()
      'Diese Zeile hilft beim schnellen Wiederauferstehen lassen des Formulars,
      'falls es doch nochmal benötigt werden sollte.
      frmMain.ENCList.Add(New WeakReference(Me))
      'Hier wird InitializeComponent aufgerufen
      Me.InitializeComponent
End Sub
```

Dieses Verhalten wird auch dadurch unterstrichen, dass etwas Außergewöhnliches passiert, wenn Sie in der eigentlichen Formularklasse einen parameterlosen Konstruktor einfügen. Der Editor zaubert nämlich aus der einfachen Eingabe von **Sub New** einen Coderumpf, wie Sie ihn auch in Abbildung 28.2 sehen können.

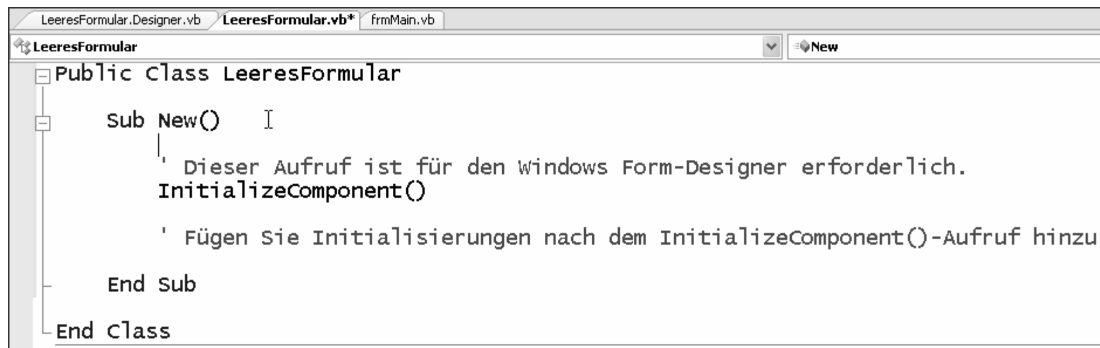

Abbildung 28.2: Wenn Sie einen Konstruktor durch die Eingabe von Sub New im Formularcode (nicht Designer-Code!) einfügen, ergänzt der Editor den Code um einen wesentlichen Aufruf

Sub New des Formulars

Egal, wo also die `Sub New` des Formulars also letzten Endes herkommt – sie gestaltet sich grundsätzlich folgendermaßen:

```
Public Sub New()
    MyBase.New()

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

    ' Initialisierungen nach dem Aufruf InitializeComponent() hinzufügen.
End Sub
```

Der Konstruktor – repräsentiert durch `Sub New` – ist existenziell für das saubere Funktionieren des Formulars. Der Formularkonstruktor wird nämlich nicht nur durch den Designer aufgerufen, wenn Sie das Formular zur Entwurfszeit bearbeiten, er wird natürlich auch vom Anwendungsframework benötigt, das das Formular instanziert und darstellt, wenn das Formular das Startformular der Anwendung ist.

InitializeComponent des Formulars

Der Konstruktor des Formulars ruft anschließend die Prozedur `InitializeComponent` auf. Und hier wird es interessant, denn `InitializeComponent` erledigt den Job, das Formular mit sinnvollen Inhalten zu füllen.

Um das zu tun, müssen die Komponenten und Steuerelemente, mit denen der Anwender später das Formular bedient, für den Gebrauch im Formular vorbereitet werden. Steuerelemente und Komponenten sind wie alle anderen Elemente in .NET Objekte. Und Objekte entstehen aus Klassen. Das heißt: Für jedes Steuerelement, das Sie zur Entwurfszeit auf das Formular gezogen haben, hat der Designer eine Objektvariable bereitgestellt. Die Deklaration dieser Objektvariablen finden Sie im Beispiel ganz am Ende des Designer-Klassencodes:

```
Friend WithEvents picViewArea As System.Windows.Forms.PictureBox
Friend WithEvents btnNextBitmap As PictureViewer.ButtonEx
Friend WithEvents btnOpenBitmap As PictureViewer.ButtonEx
Friend WithEvents btnQuitProgram As System.Windows.Forms.Button
```

```
Friend WithEvents btnSaveBitmap As System.Windows.Forms.Button
Friend WithEvents pnlPicture As System.Windows.Forms.Panel
```

Sie werden feststellen, dass die Namensgebung der Objektvariablen genau der entspricht, die Sie im Eigenschaftenfenster mit der `Name`-Eigenschaft vorgenommen haben. Sie stellen aber ebenfalls fest, dass die Komponentenvariablen mit dem Schlüsselwort `WithEvents` deklariert wurden. Diese Deklaration zeigt an, dass es über die jeweilige Objektvariable möglich ist, Ereignisse zu empfangen, die an eine zum Ereignis kompatible Prozedur delegiert werden kann. Mehr zu diesem Thema finden Sie übrigens in ▶ Kapitel 15.

Schauen wir uns als nächstes den ersten Teil von `InitializeComponent` genauer an:

```
'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.picViewArea = New System.Windows.Forms.PictureBox
    Me.btnNextBitmap = New PictureViewer.ButtonEx
    Me.btnOpenBitmap = New PictureViewer.ButtonEx
    Me.btnQuitProgram = New System.Windows.Forms.Button
    Me.btnSaveBitmap = New System.Windows.Forms.Button
    Me.pnlPicture = New System.Windows.Forms.Panel
```

Alle innerhalb des Formulars verwendeten Komponenten werden in diesem Teil instanziert. Das Attribut `DebuggerStepThrough`[1] gibt dem Debugger an, dass er nicht in diesen Teil des Programms schrittweise eintreten darf (auch, wenn Sie das Programm im Einzelschrittmodus debuggen und er dies eigentlich sollte).

Aussetzen der Layout-Logik – Suspend Layout und ResumeLayout

Um die nächsten beiden Codezeilen ranken sich in der Entwicklergemeinde mehr Mythen als Wahrheiten. Es geht um die Methode `SuspendLayout`, und Sie finden sie im besprochenen Beispielcode in den folgenden Codezeilen:

```
    Me.pnlPicture.SuspendLayout()
    Me.SuspendLayout()
```

Viele Entwickler glauben, dass sie mit `SuspendLayout` verhindern können, dass ein Steuerelement, das einem Container (das kann ein Formular oder ein Steuerelement sein, das andere Steuerelemente enthält, wie beispielsweise das GroupBox-Steuerelement) zugeordnet ist, gezeichnet wird, während es sich im *Suspend*-Zustand befindet. Das ist falsch. `SuspendLayout`, angewendet auf einen Container, der andere Steuerelemente enthält, sorgt lediglich dafür, dass das `Layout`-Ereignis[2] des Steuerelements nicht ausgelöst wird. Das `Layout`-Ereignis tritt dann ein, wenn einem Steuerelement, das als Container

---

[1] Sinngemäß übersetzt: »Debugger, überspring dies«.

[2] Mehr Informationen zu Ereignissen erfahren Sie zu einem späteren Zeitpunkt in diesem Kapitel. Fürs Erste reicht es zu wissen, dass Ereignisse dazu da sind, automatisch bestimmte Prozeduren auszuführen, wenn ein bestimmter Zustand eintritt. Die dem *Click*-Ereignis zugewiesene Prozedur beispielsweise wird also dann automatisch aufgerufen, wenn der Zustand »Anwender hat Schaltfläche angeklickt« eintritt.

fungiert, weitere Steuerelemente hinzugefügt werden, wenn Steuerelemente aus ihm entfernt werden oder wenn sich die Begrenzungen eines Steuerelements ändern.

Wieso ist es aber so wichtig, das `Layout`-Ereignis zu unterdrücken? Aus zweierlei Gründen. Zum einen ergibt es gerade beim Initialisieren eines Steuerelements aus Geschwindigkeitsgründen keinen Sinn, auf jede Eigenschaft zu reagieren, die das Layout verändert. Es reicht, wenn sich ein Steuerelement an das neue Layout anpasst, wenn alle seine Eigenschaften vollständig gesetzt sind. Zum anderen kann es gerade beim Initialisieren zum Dilemma kommen, wenn bestimmte Eigenschaften sich gegenseitig beeinflussen, und eine das Layout beeinflussende Eigenschaft, die weiter hinten im Programmcode ausgeführt wird, durch das *Layout*-Ereignis indirekt eine Eigenschaft verändert, die bereits gesetzt wurde und so Zirkelaufrufe enstehen könnten.

Die restlichen Zeilen im `InitializeComponent`-Code sind übrigens lediglich dafür verantwortlich, die geänderten Eigenschaften für die Steuerelemente so zu setzen, wie Sie sie im Eigenschaftenfenster zur Entwurfszeit definiert haben. Teilweise holpert der Code dabei ein wenig, wie beispielsweise beim Setzen der *Anchor*-Eigenschaft, dessen umständliche Formulierung

```
Me.pnlPicture.Anchor = CType((((System.Windows.Forms.AnchorStyles.Top Or _
                                System.Windows.Forms.AnchorStyles.Bottom) _
                                Or System.Windows.Forms.AnchorStyles.Left) _
                                Or System.Windows.Forms.AnchorStyles.Right), 
System.Windows.Forms.AnchorStyles)
```

auch einfach nur mit der Zeile

```
Me.pnlPicture.Anchor = AnchorStyles.Top Or AnchorStyles.Left Or AnchorStyles.Right Or AnchorStyles.Top
```

funktionieren würde; aber natürlich wurde diese Zeile nicht von Menschenhand, sondern durch eine Maschine erzeugt. Und vermutlich, um allgemein gültige Algorithmen bei der Codegenerierung einsetzen zu können, gibt es hier und da schon einmal Typ-Castings, wo keine nötig wären, aber offensichtlich schadet es auch nicht.

Steuerelemente auf dem Formular mit der ControlCollection sichtbar machen

Ungleich interessanter sind die letzten Zeilen von `InitializeComponent`, die dafür sorgen, dass die zu dieser Zeit bereits instanzierten Steuerelemente auch auf dem Formular sichtbar werden. Jedes von der `Control`-Klasse abgeleite Objekt – und dazu gehört auch `Form` – verfügt über eine so genannte `Controls`-*Auflistung*, die die Steuerelemente enthält, die dieses als Container fungierende Steuerelement enthält. Ein instanziertes Formular ist nichts weiter als ein erweitertes `Control`-Objekt, also gilt das für ein Formular gleichermaßen. In dem Moment, in dem ein Objekt mit einer instanzierten `Control`-Klasse (oder einer von `Control` abgeleiteten) der `Controls`-*Auflistung* einem `Control`-Objekt mit `Add` hinzugefügt wurde, wird das Steuerelement auch im Container (im Beispiel also dem Formular) sichtbar. Voraussetzung dafür ist natürlich, dass es sich um eine sichtbare Komponente handelt und dessen `Visible`-Eigenschaft auf `True` gesetzt wurde. Mal abgesehen von den nicht in diesen Zusammenhang passenden Zeilen

```
        Me.AutoScaleDimensions = New System.Drawing.SizeF(6.0!, 13.0!)
        Me.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font
        Me.ClientSize = New System.Drawing.Size(582, 431)
```

die kurz vorher noch die Größeneinstellungen des Formulars vornehmen, dienen die folgenden Codezeilen schließlich dazu, den Komponenten die Möglichkeit zu geben, im Formular sichtbar zu werden.

```
Me.Controls.Add(Me.btnNextBitmap)
Me.Controls.Add(Me.btnOpenBitmap)
Me.Controls.Add(Me.btnQuitProgram)
Me.Controls.Add(Me.btnSaveBitmap)
Me.Controls.Add(Me.pnlPicture)
Me.Name = "LeeresFormular"
Me.Text = "LeeresFormular"
Me.pnlPicture.ResumeLayout(False)
Me.ResumeLayout(False)
```

Die beiden letzten Zeilen schließlich sorgen dafür, dass die `Layout`-Ereignisse für `Panel` und das gesamte Formular wieder in der gewohnten Weise stattfinden können.

Um das Verhalten zu verdeutlichen: Wenn Sie der `Controls`-Auflistung eines Formulars eine Instanz einer `TextBox`-Komponente mit `Add` hinzufügen, ist die `TextBox` (entsprechend eingestellte Eigenschaften vorausgesetzt) direkt nach dem Ausführen der `Add`-Methode im Formular sichtbar und einsatzbereit.

Ereignisbehandlung von Formularen und Steuerelementen

Ereignisse haben gerade bei der Programmierung von Windows-Anwendungen eine ganz zentrale Bedeutung. Wann immer der Anwender Ihres Programms im weitesten Sinne „Irgendetwas" macht, löst er damit ein Ereignis aus, auf das Sie reagieren können.

Abbildung 28.3: Mit dem PictureViewer können Sie beliebige Grafiken eines Verzeichnisses betrachten

Das wohl einfachste Beispiel ist der Klick auf eine Schaltfläche, und es mag Ihnen als erfahrenem Entwickler ein wenig überflüssig vorkommen, etwas über die Funktionsweise des *Click*-Ereignisses zu erfahren, doch wissen Sie genau, was unter der Haube passiert?

Dazu werfen wir an dieser Stelle einen ersten Blick auf das Beispielprogramm selbst, indem wir es starten. Nach dem Start des Programms sehen Sie einen Dialog auf dem Bildschirm, wie er auch in Abbildung 28.3 zu erkennen ist.

Auf den ersten Blick scheint es überflüssig zu sein, die Funktionen dieses Beispiels zu erklären, schließlich wirkt jede Funktion so offensichtlich!

Doch eine Besonderheit hat das Programm, von der Sie Notiz nehmen können, wenn Sie mit dem Mauszeiger über eine der Schaltflächen *Grafik öffnen* oder *Nächste Grafik* fahren. Achten Sie dazu auf das, was im Ausgabefenster von Visual Studio angezeigt wird.

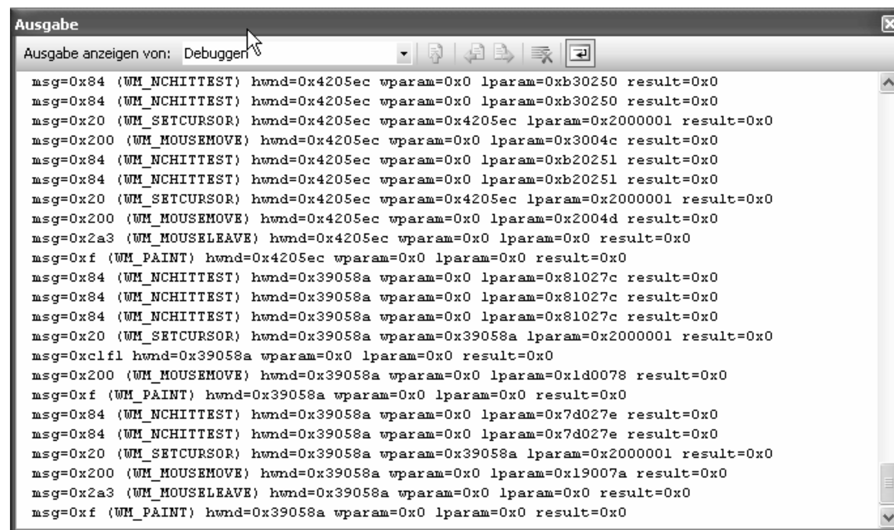

Abbildung 28.4: Wenn Sie eine der beiden Schaltflächen Grafik öffnen oder Nächste Grafik mit der Maus berühren, über sie hinwegfahren und diese betätigen, zeigt das Ausgabefenster die nativen Nachrichten der Windows-Nachrichtenschlange an

Um zu verstehen, was genau passiert, und dieses Verständnis für die funktionelle Erweiterung von Steuerelementen zu nutzen, ist allerdings ein detailliertes Verständnis für die internen Abläufe bei auf Windows-Nachrichten basierenden Ereignissen von Steuerelementen erforderlich. Der folgende Abschnitt liefert diese Grundlagen, doch nehmen Sie sich für seine Lektüre ein wenig Zeit: Er hat es nämlich »in sich«!

Vom Programmstart mithilfe des Anwendungsframeworks über eine Benutzeraktion zur Ereignisauslösung

Solange Sie nichts anderes sagen, verwenden Windows Forms-Anwendungen das Anwendungsframework, um ein Windows-Programm ans Laufen zu bekommen. Ganz vereinfacht ausgedrückt startet dabei zunächst eigentlich nicht ihr eigenes Programm, sondern ein völlig anderer Prozess, der mit

Ihrem Programm noch nichts zu tun hat. Das ist notwendig, damit eine automatisierte Aktualisierung einer SmartClient-Anwendung (über das Thema ClickOnce-Deployment, die hier den Rahmen weit sprengen würde, hält die Online-Hilfe Genaueres bereit) überhaupt funktionieren kann – denn Programmdateien können nicht ausgetauscht werden, wenn das Programm, dessen Dateien ausgetauscht werden müssen, bereits läuft.

Dieser Prozess ist übrigens auch dafür zuständig, einen eventuell darzustellenden Begrüßungsdialog anzuzeigen (den Sie ganz einfach in den Projekteinstellungen definieren können) und schließlich auf den eigentlichen Prozess Ihrer Anwendung zu initiieren und eine so genannte Windows-Nachrichtenwarteschlange für das Startformular Ihrer Anwendung einzurichten.

Diese Warteschlange ist, stark vereinfacht ausgedrückt, eine Endlosschleife, die zunächst nichts weiter macht, als zu überprüfen, ob es irgendwelche neue Meldungen vom Windows-System gibt, die an den Thread gerichtet sind, in dem sie ausgeführt werden.

Das Hauptformular spielt in der Warteschlange dabei eigentlich gar keine besondere Rolle – jedenfalls was die Verarbeitung der Warteschlange an sich anbelangt. Es wird mit seiner `Dispose`-Methode nur an den so genannten *Application Context* (etwa: Anwendungskontext) »gebunden«, und das bewirkt, dass die Warteschlangenroutine des laufenden Prozesses (und damit letzten Endes auch die Anwendung) beendet wird, wenn das Formular geschlossen wird.

Der Schlüssel der ganzen Ereignissteuerung für Windows-Nachrichten liegt eigentlich in dem Zusammenspiel zwischen der `Control`-Klasse und einer intern verwendeten Klasse, die sich `NativeWindow` nennt – und damit sind wir mitten im Thema, nämlich was passiert, wenn der Anwender beispielsweise eine Schaltfläche (die ja eine wenn auch abgeleitete `Control`-Klasse darstellt) betätigt.

Wichtig ist es zunächst einmal zu wissen, dass jedes noch so kleine Element im Windows-Betriebssystem (Schaltfläche, PictureBox – sogar ein angezeigter Tooltip) ein Window darstellt, genauso wie ein Fenster, dass Sie selbst erst als Window bezeichnen würden. Der Ereignisverlauf von einem richtigen »Windows« ist also prinzipiell kein anderer als der einer Schaltfläche, die ja schließlich auch ein Window im Windows-Betriebssystemsinne darstellt.

Schon vor .NET gab es OOP und wenn Sie sich einmal die MFC (Microsoft Foundation Class) unter C++ anschauen (Windows als Betriebssystem wurde fast vollständig in C und C++ geschrieben), sehen Sie, dass ein Button wirklich ein Fenster IST – es ist nämlich eine abgeleitete Klasse, die sie auch wieder zu einem Window downcasten können.

Sobald die `Visible`-Eigenschaft eines `Control`-Objekts oder eines von `Control` abgeleiteten Objekts auf `True` gesetzt wird und damit das erste Mal die Notwendigkeit besteht, ein Window im Sinne vom Windows-Betriebssystem ins Leben zu rufen, legt die `Control`-Klasse eine Member-Variable auf Basis der Klasse `NativeWindow` (etwa: »grundlegendes Windows«) an. `NativeWindow` wird dabei die eigene Instanz von `Control` übergeben – es gibt damit einen Zirkelverweis zwischen der `Control`-Instanz (unserer Schaltfläche `Grafik öffnen`, um beim Beispiel zu bleiben) und ihrem `NativeWindow`-Member. Zirkelverweis in diesem Zusammenhang bedeutet: Das `Control`-Objekt kann nicht nur auf die `NativeWindow`-Instanz zugreifen, sondern die `NativeWindow`-Instanz weiß auch, zu welchem `Control`-Objekt sie gehört.

`NativeWindow` ist deswegen so wichtig für `Control`, weil es die Schnittstelle zur Warteschlange bildet und zwar auf eine ganz raffinierte Art und Weise: `NativeWindow` selbst erstellt bei seiner Instanzierung ein Objekt der so genannten `WindowClass`-Klasse. Diese Klasse können Sie selbst nicht verwenden, sie steht nur dem Framework zur Verfügung und bildet eine Art Verwalter zwischen dem Windows-Subsystem und dem darüber liegenden .NET-Framework. Wenn eine `WindowClass`-Klasse instanziert

wird, dann zu dem Zweck, ein Window im Windows-Betriebssystem-Sinne zu erstellen. Dazu legt sie zunächst durch spezielle Betriebssystemaufrufe ein so genanntes *Window Handle*[3] an (jede Schaltfläche, jedes Fenster, jede Liste – ja sogar jeder Auswahlbereich *unterhalb* einer aufklappbaren Liste unter Windows ist, wie schon gesagt, ein Window im Betriebssystem-Sinne). Anschließend trägt sie sowohl dieses Window Handle als auch das `NativeWindow`-Objekt (das sie kennt, da es ihr als Konstruktorparameter übergeben wurde) in eine Tabelle ein. Auf diese Tabelle kann die Warteschlange Zugriff nehmen, die vom Anwendungsframework (oder, für Puristen, mit der Methode `Application.Run`) beim Start der Applikation eingerichtet worden ist.

Irgendwann bekommt die Warteschlange eine für sie bestimmte Nachricht, beispielsweise wenn der Anwender unseres Beispielprogramms auf die Schaltfläche `Grafik öffnen` klickt. Sie prüft nun, welches spezielle Window Handle in der Nachricht als Kennung gespeichert ist und durchforstet die Tabelle mit der Zuordnung Window Handle/`NativeWindow`-Instanz nach diesem. Auf diese Weise findet sie das entsprechende `NativeWindow`-Objekt, ruft dessen Callback[4]-Funktion auf und übergibt der Funktion dabei die Nachricht. Callback macht seinerseits wiederum ein so genanntes `Message`-Objekt daraus, das für die Nachrichtenverarbeitung in .NET verwendet wird, und ruft die `WndProc`-Routine seiner Instanz auf.

Nun wird es Zeit für eine weitere Information, die ich Ihnen bisher verschwiegen habe, um die bisherigen Zusammenhänge nicht zu undurchschaubar werden zu lassen. `Control` arbeitet nämlich eigentlich gar nicht mit einer Instanz von `NativeWindow`, sondern mit einer davon abgeleiteten Klasse namens `ControlNativeWindow`. `ControlNativeWindow` überschreibt die `WndProc`-Routine seiner Basisklasse `NativeWindow` und leitet damit den Aufruf zur `OnMessage`-Funktion um (das Programm befindet sich zu diesem Zeitpunkt immer noch in der `ControlNativWindow`-Instanz). `OnMessage` greift anschließend auf die zuvor gespeicherte Instanz des `Controls` zurück und ruft schließlich die `WndProc`-Routine von `Control` auf – die Nachricht ist jetzt bei der richtigen Komponente angekommen.

Die Aufgabe von `Control` ist jetzt nur noch, die Nachricht zu filtern, und das daraus resultierende Ereignis auszulösen. Das bedeutet aber auch, dass `WndProc` die unterste Basis jeder `Control`-Instanz (und damit auch jedes Formulars) darstellt, sich in die Ereigniskette hineinzuhängen.

In unserem Beispiel war das `Control` die Schaltfläche `Grafik öffnen`. Dessen `Click`-Ereignis haben wir in unserem Programm eingebunden, mit einer entsprechenden Auswertungslogik versehen, und dass das Ergebnis tadellos funktioniert, können Sie sehen, wenn Sie das Programm starten und auf die Schaltfläche klicken.

Doch lassen Sie uns an dieser Stelle herausfinden, was es mit der Anzeige der Nachrichten im Ausgabefenster auf sich hat. Dazu werfen Sie einen Blick auf das Hauptformular des Beispielprojekts, und betrachten die *Grafik öffnen*-Schaltfläche ein wenig genauer im Designer (siehe Abbildung 28.5).

---

[3] Eine eindeutige ID, die jedes Windows-Element unter Windows zugewiesen bekommt, wenn es ins Leben gerufen wird.

[4] Es ist nicht nur eine Callback-Funktion im klassischen C-Sinne, die Funktion heißt tatsächlich so. Kleine Randnotiz: Wenn sich das .NET-Programm im Debug-Modus befindet, ruft sie eine andere Funktion namens *DebuggableCallback* auf.

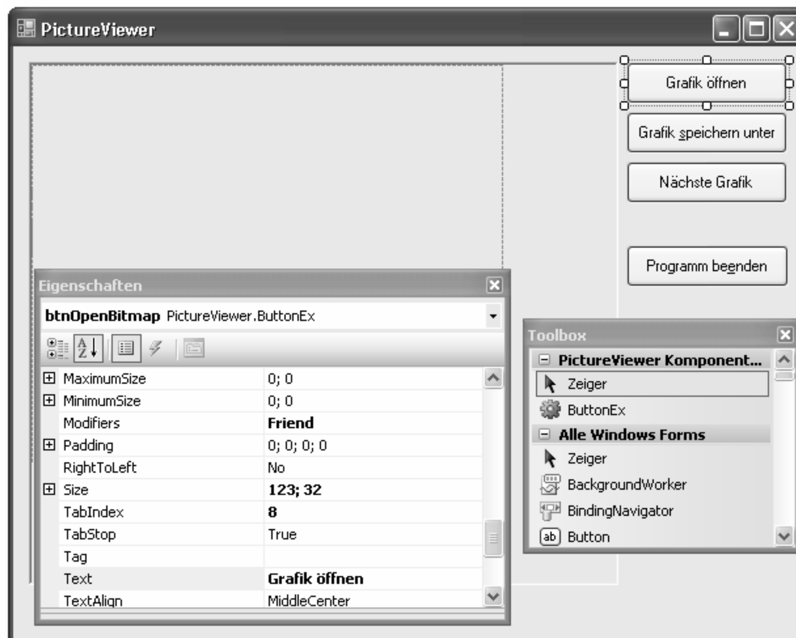

Abbildung 28.5: Bei genauer Betrachtung wird deutlich, dass es sich bei der *Grafik öffnen*-Schaltfläche nicht um ein *Button*-, sondern um ein *ButtonEx*-Steuerelement handelt

Sie sehen, dass es sich bei der Schaltfläche im Formular gar nicht um eine »herkömmliche« Schaltfläche, sondern um eine Erweiterung handelt. Eine zusätzliche Codedatei namens ButtonEx.vb klärt, wieso sowohl Toolbox als auch Formular über eine eigentlich unbekannte Schaltfläche verfügen können:

```
Public Class ButtonEx
    Inherits Button

    Private Const WM_RBUTTONDOWN As Integer = &H204
    Private Const WM_RBUTTONUP As Integer = &H205
    Private myDownFlag As Boolean

    Public Event RightClick(ByVal Sender As Object, ByVal e As EventArgs)

    Protected Overrides Sub WndProc(ByRef m As System.Windows.Forms.Message)
        Debug.Print(m.ToString)
        If m.Msg = WM_RBUTTONDOWN Then
            myDownFlag = True
        End If
        If m.Msg = WM_RBUTTONUP And myDownFlag Then
            myDownFlag = False
            OnRightClick(Me, EventArgs.Empty)
        End If
        MyBase.WndProc(m)
    End Sub
```

```
    Protected Overridable Sub OnRightClick(ByVal Sender As Object, ByVal e As EventArgs)
        RaiseEvent RightClick(Sender, e)
    End Sub
End Class
```

Das Erstellen einer Klassendatei mit diesem Namen und das Einfügen dieses Codes haben ausgereicht, um nach dem ersten Kompilieren das »neue« Steuerelement in der Toolbox anzeigen zu lassen als auch es sofort anschließend im Formular verwenden zu können.

Für ein weiteres Experiment setzen Sie nun in der im oben stehenden Klassencode in der fett ausgezeichneten Zeile mit der Funktionstaste **F9** einen Haltepunkt. Starten Sie das Programm anschließend und klicken Sie mit der *rechten* Maustaste auf die Schaltfläche *Grafik öffnen*.

Sie können im Ausgabefenster nun in aller Ruhe die Nachrichtenhistorie der Schaltfläche betrachten (siehe Abbildung 28.5). *hwnd* im Fenster bezeichnet übrigens das Window Handle, von dem in den vorherigen Erklärungen die Rede war und das die Warteschlangenroutine verwendet hat, um die .NET-Schaltfläche wieder zu finden. Der Debugger bietet Ihnen die Möglichkeit, die Aufruf-Hierarchie der aktuellen Funktion aufzulisten, in der das Programm gerade »steckt«. Um die Aufrufliste darzustellen, drücken Sie einfach die Tastenkombination **Strg+Alt+C** (alternativ wählen Sie aus dem Menü *Debuggen*, den Menüpunkt *Fenster* und weiter den Untermenüpunkt *Aufrufliste*).

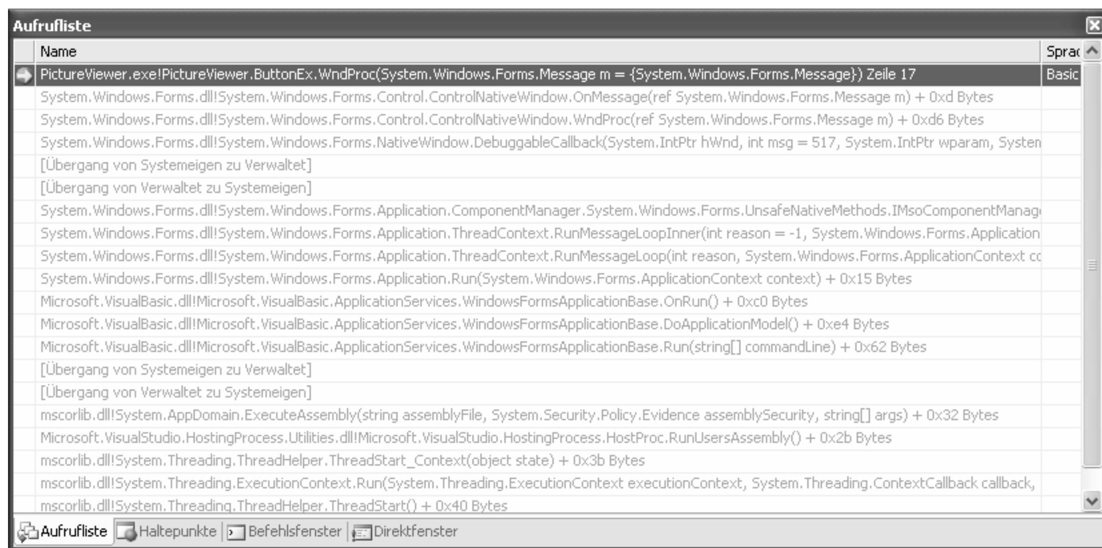

Abbildung 28.6: Die Aufrufliste zeigt die Quelle des Funktionsaufrufs

Die Aufrufliste verifiziert die vorhin geschilderte Erklärung. Fast lückenlos sind die verschiedenen Funktionsaufrufe in der Kette zu verfolgen, die letzten Endes zum Ausführen der Routine *WndProc* des jeweiligen Steuerelements oder Formulars führen.

HINWEIS: In der Standardeinstellung von Visual Studio werden Aufrufe, die nicht von Ihrer Anwendung stammen, nicht im Ausgabefenster angezeigt. In diesem Fall öffnen Sie das Kontextmenü der Aufrufliste mit der rechten Maustaste und wählen den Eintrag *Nicht-benutzerseitigen Code anzeigen* aus.

Implementieren neuer Steuerelementereignisse auf Basis von Warteschlangenauswertungen

Nun wissen Sie, wie Nachrichten des Windows-Betriebssystems in .NET-WinForms-Anwendungen verarbeitet werden. Dieses Wissen können Sie sich zunutze machen, um vorhandene Steuerelemente um eigene Ereignisse zu erweitern.

So könnte es für den Entwickler beispielsweise von Vorteil sein, ein `RightClick`-Ereignis zu erhalten, wenn der Anwender über der Schaltfläche die rechte Maustaste drückt.

Wenn ein Mausklickereignis ein bestimmtes Objekt erreichen soll, dann reicht es nicht aus, die Nachrichtenwarteschlange von Windows daraufhin abzufragen. Eine Klick-Nachricht gibt es nämlich in dieser Form überhaupt nicht. Ein Klick besteht aus einer direkten Folge aus Nachrichten vom Nachrichtentyp *WM_LMOUSEDOWN*[5] und *WM_LMOUSEUP* – jedenfalls soweit das die linke Maustaste betrifft.

In der Routine *WndProc*, die die Windows-Nachrichten schon gefiltert für die eigene Instanz der Klasse (*TestButton* im Beispiel) bekommt, sollte es deswegen eine Member-Variable geben, die sich merkt, ob die rechte Maustaste bereits gedrückt wurde. Dann kann der Auswertungsabschnitt für die Nachricht des Loslassens der rechten Maustaste beide Aktionen als »Mausklick rechts« interpretieren und das *RightClick*-Ereignis auslösen.

HINWEIS: Mir ist klar, dass es mehrere Möglichkeiten gibt, ein solches Ereignis zur Verfügung zu stellen. Alternativ zum Abfangen der Windows-Nachrichten in *WndProc* ließe sich auch *PreProcessMessage* überschreiben – eine Funktion, die den Vorteil hat, bereits fix und fertig aufbereitete Windows-Nachrichten im .NET-üblichen *Message*-Format zu empfangen. Auch das Überschreiben von *OnMouseDown* und *OnMouseUp* wäre denkbar – aber: Es geht an dieser Stelle mehr um die Demonstration der grundlegenden Verfahren als um die beste Form der Implementierung. Die schnellste ist die *WndProc*-Methode allemal, denn alle anderen Funktionen werden mehr oder weniger direkt aus *WndProc* heraus aufgerufen.

Der Beispielcode auf Seite 831 zeigt die komplette Implementierung, die letzten Endes zum Ausführen des Ereignisses führt. Mehr über die Grundlagen von Ereignissen und zum Auslösen von Ereignissen erfahren Sie übrigens in ▶ Kapitel 15.

Entfernen Sie nun den Haltepunkt, den Sie im vorherigen Abschnitt gesetzt haben, und starten Sie die Beispielanwendung erneut. Und nun beobachten Sie, was passiert, wenn Sie die Schaltflächen *Grafik öffnen* und *Nächste Grafik* mit der rechten Maustaste bedienen.

- Wenn Sie die Schaltfläche *Grafik öffnen* normal betätigen, wird standardmäßig jeweils das Verzeichnis in der Dateiauswahl angezeigt, das sie als Letztes verwendet haben. Klicken Sie auf diese Schaltfläche jedoch mit der *rechten* Maustaste, sehen Sie grundsätzlich die Verzeichnisauswahl Ihres *Eigene-Bilder*-Verzeichnisses.

---

[5] Die Bezeichnung von Ereignissen in Form von Konstanten hat eine lange Geschichte und reicht zurück bis zur Programmierung von Windows 2.11 unter C. WM_ steht dabei natürlich für Windows Message. In .NET sind diese Konstanten leider nicht vordefiniert, aber spätere Beispiele zeigen, wie Sie an die entsprechenden Definitionen gelangen.

- Wenn Sie die Schaltfläche *Nächste Grafik* normal betätigen, wird automatisch das nächste Bild im Verzeichnis dargestellt. Betätigen Sie jedoch die *rechte* Maustaste, dann wird die erste Bilddatei des Verzeichnisses dargestellt.

Wer oder was löst welche Formular- bzw. Steuerelementereignisse wann aus?

Es gibt eine Vielzahl von Ereignissen, die durch Benutzeraktionen bei Formularen (und den Schaltflächen, die sie beherbergen) ausgelöst werden können. Auch, wenn Sie sich schon eine Weile mit diesem Thema beschäftigt haben, bleibt es immer noch aufwändig herauszufinden, welche Aktion des Benutzers welches Ereignis wann auslöst.

Ich habe mir lange Gedanken darüber gemacht, was Ihnen beim Finden des richtigen Ereignisses und beim Verstehen der richtigen Zusammenhänge bei Ereignissen am besten helfen kann. Das Ergebnis ist das folgende Programm, das die Geheimnisse um Ereignisse sowohl für Formulare als auch für Komponenten auf seine Weise löst.

BEGLEITDATEIEN: Sie finden das Programm zu diesem Beispiel im Verzeichnis.*\VB 2005 - Entwicklerbuch\G - SmartClient\Kap28\Ereignistester*.

Dieses Programm soll gleich zweierlei Zwecke erfüllen. Auf der einen Seite soll es Ihnen als eine Art Nachschlagewerk dienen. Sämtliche Methoden, die es überschreibt, sind dokumentiert, und sie beschreiben sozusagen direkt vor Ort, welche Überschreibung welchen Zweck erfüllt. Wenn Sie dennoch nicht sicher sind, in welchem Zusammenhang die verschiedenen Ereignisse stehen, können Sie es auf der anderen Seite auch als Testprogramm verwenden, um mit den Ereignissen zu experimentieren. Die wichtigsten Ereignisse sind nämlich überschrieben, und wenn Sie das Programm starten, generiert es ein Testformular, das auch eine Testkomponente enthält. Durch Verschieben, Vergrößern, Verkleinern, Darüberfahren mit der Maus, Daraufklicken und das Ausführen anderer Aktionen sehen Sie im Ausgabefenster, welches Ereignis zu welcher Zeit aufgerufen wird.

Wenn Sie dieses Programm starten, sehen Sie zunächst einen Dialog, etwa wie in Abbildung 28.7 auf dem Bildschirm:

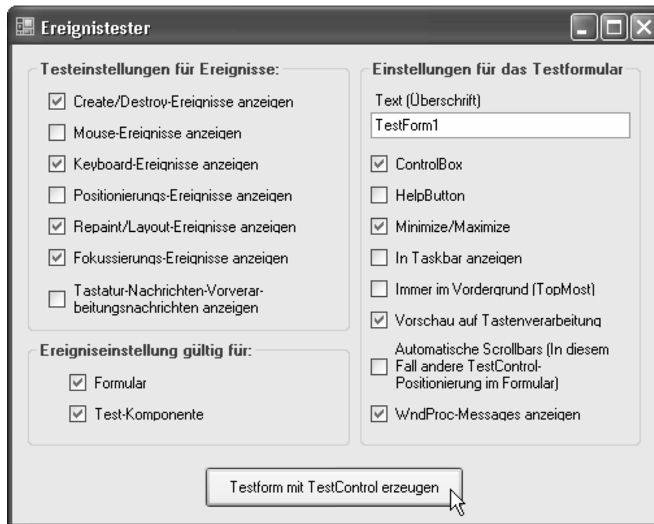

Abbildung 28.7: Im Hauptdialog der Anwendung nehmen Sie die Einstellungen für die Elemente und Ereignisse vor, die Sie nachverfolgen bzw. testen möchten

Dieser Dialog erlaubt Ihnen zu bestimmen, welche Ereigniskategorie im Ausgabefenster protokolliert werden soll. Die verschiedenen Ereignisse sind dabei in Gruppen eingeteilt. Durch die Kontrollkästchen im Bereich *Testeinstellungen für Ereignisse* wählen Sie die zu protokollierenden aus. Im darunter liegenden Bereich bestimmen Sie, ob die Ereignisse nur für das Formular, nur für die Testkomponente oder für beide ausgegeben werden sollen.

In der Rubrik *Einstellungen für das Testformular* bestimmen Sie, mit welchen Attributen Sie das Testformular ausstatten möchten. Diese Optionen repräsentieren die wichtigsten Eigenschaften, die Sie auch zur Entwurfszeit für ein Formular einstellen können.

Wählen Sie nun bitte die Einstellungen so aus, wie Sie sie auch in Abbildung 28.7 sehen können. Klicken Sie anschließend auf die Schaltfläche *Testform mit TestControl* erzeugen.

Sie sehen anschließend ein Formular mit einer wunderschönen, selbst gestrickten Komponente, wie Sie auch in Abbildung 28.8 zu sehen ist.

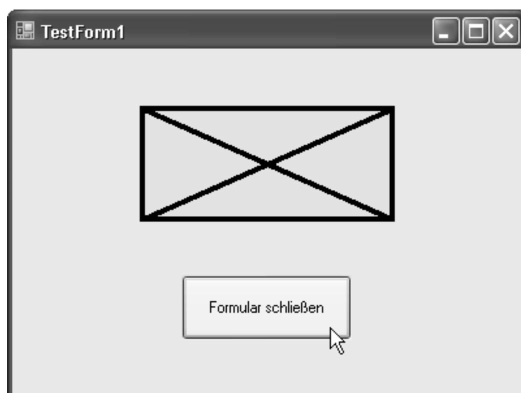

Abbildung 28.8: Mit diesem Testformular und seiner benutzerdefinierten Komponente können Sie Ihre Experimente durchführen

Und jetzt toben Sie sich aus! Bewegen Sie das Formular über den Bildschirm. Fahren Sie mit dem Mauszeiger über Formular und Komponenten. Vergrößern und verkleinern Sie das Formular. Wechseln Sie mit der Tabulator-Taste den Fokus der beiden Komponenten. Klicken Sie in das Formular. Halten Sie den Mausknopf über dem Formular gedrückt, und bewegen Sie dabei die Maus.

Wann immer Sie Aktionen durchführen, sehen Sie im Ausgabefenster eine entsprechende Kommentierung dazu, wie etwa in folgendem Protokollauszug zu sehen:

```
'FormEventChain.exe': 'c:\windows\assembly\gac\system.xml\1.0.5000.0__b77a5c561934e089\system.xml.dll'
geladen, keine Symbole geladen.
FormEventChain.frmTest, Text: frmTest: OnLayout: AffectedControl=FormEventChain.frmTest, Text: frmTest;
AffectedProperty=Bounds
FormEventChain.frmTest, Text: frmTest: OnLayout: AffectedControl=FormEventChain.frmTest, Text: frmTest;
AffectedProperty=Bounds
FormEventChain.frmTest, Text: TestForm1: OnLayout: AffectedControl=; AffectedProperty=
FormEventChain.TestControl: OnLayout: AffectedControl=FormEventChain.TestControl; AffectedProperty=Bounds
FormEventChain.TestControl: OnLayout: AffectedControl=FormEventChain.TestControl; AffectedProperty=Bounds
FormEventChain.frmTest, Text: TestForm1: OnLayout: AffectedControl=FormEventChain.TestControl;
AffectedProperty=Parent
FormEventChain.frmTest, Text: TestForm1: OnLayout: AffectedControl=System.Windows.Forms.Button, Text:
Formular &schließen; AffectedProperty=Parent
FormEventChain.frmTest, Text: TestForm1: OnHandleCreated
FormEventChain.frmTest, Text: TestForm1: OnActivated
FormEventChain.frmTest, Text: TestForm1: OnInvalidated: InvalidRect={X=0,Y=0,Width=390,Height=236}
FormEventChain.TestControl: OnHandleCreated
FormEventChain.TestControl: OnCreateControl
FormEventChain.frmTest, Text: TestForm1: OnLoad
FormEventChain.frmTest, Text: TestForm1: OnCreateControl
FormEventChain.frmTest, Text: TestForm1: OnLayout: AffectedControl=; AffectedProperty=
FormEventChain.frmTest, Text: TestForm1: OnPaintBackground:{X=0,Y=0,Width=390,Height=236}
FormEventChain.TestControl: OnInvalidated: InvalidRect={X=0,Y=0,Width=195,Height=79}
FormEventChain.frmTest, Text: TestForm1: SetVisibleCore: value=True
FormEventChain.frmTest, Text: TestForm1: OnPaint:{X=0,Y=0,Width=390,Height=236}
FormEventChain.TestControl: OnPaintBackground:{X=0,Y=0,Width=195,Height=79}
FormEventChain.TestControl: OnPaint:{X=0,Y=0,Width=195,Height=79}
FormEventChain.TestControl: OnInvalidated: InvalidRect={X=0,Y=0,Width=195,Height=79}
FormEventChain.TestControl: OnPaintBackground:{X=0,Y=0,Width=195,Height=79}
FormEventChain.TestControl: OnPaint:{X=0,Y=0,Width=195,Height=79}
'FormEventChain.exe':
'c:\windows\assembly\gac\microsoft.visualbasic\7.0.5000.0__b03f5f7f11d50a3a\microsoft.visualbasic.dll'
geladen, keine Symbole geladen.
FormEventChain.frmTest, Text: TestForm1: OnDeactivate
FormEventChain.frmTest, Text: TestForm1: OnHandleDestroyed
FormEventChain.TestControl: OnHandleDestroyed
FormEventChain.TestControl: Dispose
FormEventChain.frmTest, Text: : Dispose
```

Es ist interessant zu sehen, wie die einzelnen Ereignisse sich gegenseitig bedingen, finden Sie nicht?

Noch interessanter ist, welche Ereignisse mit welchen Prozeduren abgefangen werden können. Prinzipiell spielt es natürlich keine Rolle, ob Sie ein Ereignis als Event im Sinne eines .NET-Framework-Events behandeln oder, wenn es im Kontext möglich ist, eine entsprechende Prozedur durch Überschreiben verwenden. Zu diesem Zweck sollen Ihnen die folgenden Seiten dienen, die das kommentierte Listing enthalten. Es ist gemäß den Kategorien der Ereignisse in verschiedene Sektionen

unterteilt, die auch mit entsprechenden Überschriften versehen sind. Dadurch können Sie dieses Listing auch als Nachschlagewerk verwenden, wenn Sie später, beim Entwickeln Ihrer eigenen Komponenten und Anwendungen, schnell eine geeignete Ereignisroutine finden müssen.

HINWEIS: Die Definitionszeilen der einzelnen Prozeduren sind fett formatiert, damit Sie sie leichter im Listing erkennen können. Darüber hinaus sind die Prozeduren – soweit möglich – nach der Reihenfolge ihres Auftretens innerhalb der einzelnen Kategorien sortiert.

Auch wenn das Abdrucken beider Teile – der des Formulars und der der `Control`-Klasse – auf den ersten Blick doppelt und damit überflüssig erscheint: Formulare und Steuerelemente lösen in gleichen Situationen oft verschiedene Ereignisse aus – das ist der Grund.

Die Kommentare sind zu Gunsten der leichteren Lesbarkeit im folgenden Listung in normalen Fließtext umgewandelt worden. Im Code selbst finden Sie die Kommentare in gleichem Wortlaut als Kommentarzeilen.

Kategorie Erstellen und Zerstören des Formulars

```
'********************************************************
'Erstellen, Aktivieren, Deaktivieren und Zerstören
'********************************************************
```

OnHandleCreated: Wird aufgerufen, nachdem das *Window-Handle* für die Formular-Instanz erstellt wurde. Ab diesem Zeitpunkt ist das Formular von der Nachrichtenwarteschlange erkennbar.

```
Protected Overrides Sub OnHandleCreated(ByVal e As System.EventArgs)
    MyBase.OnHandleCreated(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnHandleCreated")
    End If
End Sub
```

OnLoad: Tritt ein, kurz bevor die Formular-Instanz das erste Mal sichtbar wird. Sie haben hier die Möglichkeit, Initialisierungen für das Formular vorzunehmen. Beachten Sie, dass das Fokussieren von Steuerelementen zu dieser Zeit noch nicht funktioniert und eine Ausnahme auslösen würde.

```
Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
    MyBase.OnLoad(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnLoad")
    End If
End Sub
```

OnCreateControl: Tritt ein, nachdem die Framework-seitigen Ressourcen für das Formular erstellt wurden. Die Basisfunktion muss in den Framework-Versionen 1.0 und 1.1 nicht notwendigerweise aufgerufen werden; aus Aufwärtskompatibilitätsgründen sollte es aber dennoch passieren.

```
Protected Overrides Sub OnCreateControl()
    MyBase.OnCreateControl()
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnCreateControl")
    End If
End Sub
```

OnActivated: Wird aufgerufen, wenn das Formular aktiviert wurde. Bei einem Formular wird diese Funktion aufgerufen, wenn es zum zuoberst liegenden wird – entweder durch Benutzerklick auf das Formular, oder, da es das Hauptfenster der Anwendung ist, dadurch, dass die Anwendung gestartet oder aktiviert wurde.

```
Protected Overrides Sub OnActivated(ByVal e As System.EventArgs)
    MyBase.OnActivated(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnActivated")
    End If
End Sub
```

SetVisibleCore: Wird aufgerufen, um einer ableitenden Klasse beim Initialisierungsvorgang die Möglichkeit zu geben, die Sichtbarkeit (durch die `Visible`-Eigenschaft gesteuert) zu ändern. Eigentlich ist diese Routine kein richtiges Ereignis, sondern nur die ausführende Unterfunktion einer Eigenschaft.

```
Protected Overrides Sub SetVisibleCore(ByVal value As Boolean)
    MyBase.SetVisibleCore(value)
    If myShowCreationDestroy Then
        Debug.WriteLine(String.Format(Me.ToString + ": SetVisibleCore: value={0}", value))
    End If
End Sub
```

OnClosing: Wird aufgerufen, wenn der Schließen-Vorgang des Formulars beginnt. Sie können das Schließen verhindern, indem Sie die `Cancel`-Eigenschaft von `e` auf `True` setzen.

```
Protected Overrides Sub OnClosing(ByVal e As System.ComponentModel.CancelEventArgs)
    MyBase.OnClosing(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnClosing")
    End If
End Sub
```

OnClosed: Wird aufgerufen, wenn das Formular geschlossen wurde.

```
Protected Overrides Sub OnClosed(ByVal e As System.EventArgs)
    MyBase.OnClosed(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnClosed")
    End If
End Sub
```

OnDeactivate: Wird aufgerufen, wenn das Formular deaktiviert wurde.

```
Protected Overrides Sub OnDeactivate(ByVal e As System.EventArgs)
    MyBase.OnDeactivate(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnDeactivate")
    End If
End Sub
```

OnHandleDestroyed: Wird aufgerufen, wenn das *Window-Handle* des Formulars zerstört wurde.

```
Protected Overrides Sub OnHandleDestroyed(ByVal e As System.EventArgs)
    MyBase.OnHandleDestroyed(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnHandleDestroyed")
    End If
End Sub
```

Dispose: Das Formular überschreibt den Löschvorgang der Basisklasse, um Komponenten zu bereinigen. Diese Routine wird in der Regel durch den Formular-Designer implementiert.

```
Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
    If disposing Then
        If Not (components Is Nothing) Then
            components.Dispose()
        End If
    End If
    MyBase.Dispose(disposing)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": Dispose")
    End If
End Sub
```

Kategorie Mausereignisse des Formulars

```
'*******************************************************
'Mausereignisse
'*******************************************************
```

OnMouseDown: Wird aufgerufen, wenn ein Mausbutton gedrückt wird und sich die Maus über einem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) befindet.

```
Protected Overrides Sub OnMouseDown(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseDown(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format( _
                Me.ToString + ": OnMouseDown: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
                                , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnClick: Wird aufgerufen, wenn ein Mausklick mit der linken Maustaste über einem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) durchgeführt wird.

```
Protected Overrides Sub OnClick(ByVal e As System.EventArgs)
    MyBase.OnClick(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnClick")
    End If
End Sub
```

OnDoubleClick: Wird aufgerufen, wenn ein Doppelklick mit der linken Maustaste über einem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) durchgeführt wird.

```
Protected Overrides Sub OnDoubleClick(ByVal e As System.EventArgs)
    MyBase.OnDoubleClick(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnDoubleClick")
    End If
End Sub
```

OnMouseUp: Wird aufgerufen, wenn ein Mausbutton losgelassen wird und sich die Maus über einem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) befindet.

```
Protected Overrides Sub OnMouseUp(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseUp(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format( _
                Me.ToString + ": OnMouseUp: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
                                    , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnMouseEnter: Wird aufgerufen, wenn der Mauszeiger den Bereich des Formulars, aber nicht einen *ChildWindow*-Bereich (andere Komponente) betritt.

```
Protected Overrides Sub OnMouseEnter(ByVal e As System.EventArgs)
    MyBase.OnMouseEnter(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseEnter:" + e.ToString)
    End If
End Sub
```

OnMouseHover: Wird aufgerufen, wenn der Mauszeiger das erste Mal nach dem Betreten des Formularbereichs zur Ruhe gekommen ist.

```
Protected Overrides Sub OnMouseHover(ByVal e As System.EventArgs)
    MyBase.OnMouseHover(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseHover:" + e.ToString)
    End If
End Sub
```

OnMouseMove: Wird aufgerufen, wenn der Mauszeiger über dem Bereich des Formulars, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) bewegt wird.

```
Protected Overrides Sub OnMouseMove(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseMove(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format( _
                Me.ToString + ": OnMouseMove: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
                            , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnMouseLeave: Wird aufgerufen, wenn der Mauszeiger den Bereich des Formulars verlässt.

```
Protected Overrides Sub OnMouseLeave(ByVal e As System.EventArgs)
    MyBase.OnMouseLeave(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseLeave:" + e.ToString)
    End If
End Sub
```

OnMouseWheel: Wird aufgerufen, wenn das Mausrad über dem Bereich des Formulars bewegt wird. Dieses Ereignis wird für alle untergeordneten Komponenten (*ChildWindows*) ebenfalls ausgelöst!

```
Protected Overrides Sub OnMouseWheel(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseWheel(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format( _
                Me.ToString + ": OnMouseWheel: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
                                , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

Kategorie Tastaturereignisse des Formulars

```
'*******************************************************
'Tastatur
'*******************************************************
```

OnKeyDown: Wird aufgerufen, wenn eine Taste gedrückt wird. Wird allerdings nicht aufgerufen, wenn es eine weitere, fokussierte Komponente im Formular gibt und die `KeyPreview`-Eigenschaft auf `False` gesetzt wurde bzw. die überschriebene `ProcessKeyPreview`-Methode (s.u.) das Ereignis schon verarbeitet hat.

```
Protected Overrides Sub OnKeyDown(ByVal e As System.Windows.Forms.KeyEventArgs)
    MyBase.OnKeyDown(e)
    If myShowKeyboard Then
        Debug.WriteLine(String.Format( _
            Me.ToString + ": OnkeyDown: KeyCode={0}; KeyData={1}; KeyValue={2}; Modifiers={3}", _
                e.KeyCode, e.KeyData, e.KeyValue, e.Modifiers))
    End If
End Sub
```

OnKeyPress: Wird aufgerufen, wenn eine Taste gedrückt wird; wird nicht aufgerufen, wenn eine Steuerungstaste (wie **Strg** oder **Shift**) alleine oder in Kombination mit einer anderen gedrückt wird. Diese Prozedur wird auch dann nicht aufgerufen, wenn es eine weitere, fokussierte Komponente im Formular gibt und die `KeyPreview`-Eigenschaft auf `False` gesetzt wurde bzw. die überschriebene `ProcessKeyPreview`-Methode (s.u.) das Ereignis schon verarbeitet hat.

```
Protected Overrides Sub OnKeyPress(ByVal e As System.Windows.Forms.KeyPressEventArgs)
    MyBase.OnKeyPress(e)
    If myShowKeyboard Then
        Debug.WriteLine(String.Format( _
                    Me.ToString + ": OnKeyPress: KeyChar={0}", _
                    e.KeyChar))
    End If
End Sub
```

OnKeyUp: Wird aufgerufen, wenn eine Taste wieder losgelassen wird. Diese Prozedur wird nicht aufgerufen, wenn es eine weitere, fokussierte Komponente im Formular gibt und die `KeyPreview`-Eigenschaft auf `False` gesetzt wurde bzw. die überschriebene `ProcessKeyPreview`-Methode (s.u) das Ereignis schon verarbeitet hat.

```
Protected Overrides Sub OnKeyUp(ByVal e As System.Windows.Forms.KeyEventArgs)
    MyBase.OnKeyUp(e)
    If myShowKeyboard Then
        Debug.WriteLine(String.Format( _
            Me.ToString + ": OnKeyUp: KeyCode={0}; KeyData={1}; KeyValue={2}; Modifiers={3}", _
                e.KeyCode, e.KeyData, e.KeyValue, e.Modifiers))
    End If
End Sub
```

Kategorie Position und Größe des Formulars

```
'*******************************************************
'Größe und Position
'*******************************************************
```

OnMove: Wird aufgerufen, wenn die Formularposition verändert wird. Diese Methode wird kontinuierlich aufgerufen, während der Anwender das Formular verschiebt und die Anzeigeneinstellung so vorgenommen wurde, dass der Fensterinhalt beim Ziehen mit verschoben wird.

```
Protected Overrides Sub OnMove(ByVal e As System.EventArgs)
    MyBase.OnMove(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnMove:" + e.ToString)
    End If
End Sub
```

OnLocationChanged: Wird aufgerufen, wenn sich die Position des Formulars verändert hat. Diese Methode wird leider ebenfalls kontinuierlich aufgerufen, wenn die Anzeigeneinstellung so vorgenommen wurde, dass der Fensterinhalt beim Ziehen mit verschoben wird, sodass ein Ende der Verschiebeaktion hiermit nicht festgestellt werden kann. Um das zu erreichen, müssten Sie `WndProc` überschreiben und die empfangene Nachricht dort auf `WM_EXITSIZEMOVE` überprüfen. Ein Beispiel dazu finden Sie in ▶ Kapitel 29.

```
Protected Overrides Sub OnLocationChanged(ByVal e As System.EventArgs)
    MyBase.OnLocationChanged(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnLocationChanged:" + e.ToString)
    End If
End Sub
```

OnResize: Wird aufgerufen, wenn die Formulargröße verändert wird. Diese Methode wird kontinuierlich aufgerufen, während der Anwender das Formular vergrößert oder verkleinert und die Anzeigeneinstellung so vorgenommen wurde, dass der Fensterinhalt beim Ziehen mit verschoben wird.

```
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
    MyBase.OnResize(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnResize:" + e.ToString)
    End If
End Sub
```

OnSizeChanged: Wird aufgerufen, wenn sich die Größe des Formulars verändert hat. Diese Methode wird leider ebenfalls kontinuierlich aufgerufen, wenn die Anzeigeneinstellung so vorgenommen wurde, dass der Fensterinhalt beim Ziehen mit verschoben wird, sodass ein Abschluss der Größenänderung hiermit nicht festgestellt werden kann. Um das zu erreichen, müssten Sie `WndProc` überschreiben und die empfangene Nachricht dort auf den Wert `WM_EXITSIZEMOVE` überprüfen. Ein Beispiel dazu finden Sie in ▶ Kapitel 29.

```
Protected Overrides Sub OnSizeChanged(ByVal e As System.EventArgs)
    MyBase.OnSizeChanged(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnSizeChanged:" + e.ToString)
    End If
End Sub
```

Kategorie Anordnen der Komponenten und Neuzeichnen des Formulars

```
'******************************************************
'Anordnen und Neuzeichnen
'******************************************************
```

OnInvalidated: Wird aufgerufen, wenn eine Entität das Neuzeichnen des Formularinhalts mit `Invalidate` anfordert. `Invalidate` sollte in der Regel von `OnResize` aufgerufen werden, wenn der Inhalt des Fensters in Abhängigkeit von der Fenstergröße komplett neu gezeichnet werden muss. Ausgenommen sind Änderungen am Verhalten durch `SetStyle` (▶ Kapitel 29).

```
Protected Overrides Sub OnInvalidated(ByVal e As System.Windows.Forms.InvalidateEventArgs)
    MyBase.OnInvalidated(e)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(String.Format( _
                    Me.ToString + ": OnInvalidated: InvalidRect={0}", _
                    e.InvalidRect))
    End If
End Sub
```

OnLayout: Wird aufgerufen, wenn das Formular anzeigt, dass seine beinhaltenden Steuerelemente aus irgendwelchen Gründen neu angeordnet werden müssen.

HINWEIS: Für Änderungen an einem Steuerelement, z. B. Größenänderungen, Ein- oder Ausblenden sowie Hinzufügen oder Entfernen untergeordneter Steuerelemente ist es notwendig, dass das Layout der untergeordneten Steuerelemente vom Steuerelement festgelegt wird. Der diesem Ereignis mitgegebene Parameter `LayoutEventArgs` gibt das geänderte untergeordnete Steuerelement und die davon betroffene Eigenschaft an. Wenn z. B. ein Steuerelement seit dem letzten Layoutvorgang sichtbar gemacht wurde, ist davon die `Visible`-Eigenschaft betroffen. Die `AffectedControl`- und `AffectedProperty`-Eigenschaften werden auf `Nothing` festgelegt, wenn beim Aufruf der `PerformLayout`-Methode keine Werte bereitgestellt wurden. Dieses Ereignis erfolgt nicht, wenn das Formular das Layout-Ereignis mit `SuspendLayout` außer Kraft gesetzt hat.

```
Protected Overrides Sub OnLayout(ByVal levent As System.Windows.Forms.LayoutEventArgs)
    MyBase.OnLayout(levent)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(String.Format( _
                    Me.ToString + ": OnLayout: AffectedControl={0}; AffectedProperty={1}", _
                    levent.AffectedControl, levent.AffectedProperty))
    End If
End Sub
```

OnPaintBackground: Wird aufgerufen, wenn der Hintergrund des Formulars neu gezeichnet werden muss. Das `Graphics`-Objekt, das mit dem Parameter vom Typ `PaintEventArgs` dem Ereignis übergeben wird, ist ausschließlich auf den Bereich geclipped, der neu gezeichnet werden muss. Wenn durch die Vergrößerung des Fensters der Fensterinhalt komplett neu gezeichnet werden muss, dann sollte `OnResize` bzw. das `Resize`-Ereignis die Methode `Invalidate` aufrufen, damit den Paint-Ereignissen ein ungeclippter Bereich für das Neuzeichnen des kompletten Inhalts übergeben wird. Beispiele dafür gibt es auch in ▶ Kapitel 29.

```
Protected Overrides Sub OnPaintBackground(ByVal pevent As System.Windows.Forms.PaintEventArgs)
    MyBase.OnPaintBackground(pevent)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(Me.ToString + ": OnPaintBackground:" + pevent.ClipRectangle.ToString)
    End If
End Sub
```

OnPaint: Wird aufgerufen, wenn der Fensterinhalt neu gezeichnet werden muss. Das `Graphics`-Objekt, das mit dem Parameter vom Typ `PaintEventArgs` dem Ereignis übergeben wird, ist ausschließlich auf den Bereich `geclipped`, der neu gezeichnet werden muss. Wenn durch die Vergrößerung des Fensters der Fensterinhalt komplett neu gezeichnet werden muss, dann sollte `OnResize` bzw. das `Resize`-Ereignis die Methode `Invalidate` aufrufen, damit den `Paint`-Ereignissen ein ungeclippter Bereich für das Neuzeichnen des kompletten Inhalts übergeben wird. Beispiele dafür gibt es auch in ▶ Kapitel 29.

```
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    MyBase.OnPaint(e)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(Me.ToString + ": OnPaint:" + e.ClipRectangle.ToString)
    End If
End Sub
```

Kategorie Fokussierung des Formulars

```
'*******************************************************
'Fokussierung
'*******************************************************
```

OnEnter: Wird aufgerufen, wenn das Formular aktiviert wird, aber nur, wenn es mindestens eine weitere Komponente beinhaltet, die den Fokus beim Aktivieren bekommen kann.

```
Protected Overrides Sub OnEnter(ByVal e As System.EventArgs)
    MyBase.OnEnter(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnEnter:" + e.ToString)
    End If
End Sub
```

OnGotFocus: Wird aufgerufen, wenn das Formular aktiviert wird, aber nur, wenn es kein weiteres `Steuerelement` beinhaltet, das den Fokus beim Aktivieren bekommen könnte.

```
Protected Overrides Sub OnGotFocus(ByVal e As System.EventArgs)
    MyBase.OnLostFocus(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnGotFocus:" + e.ToString)
    End If
End Sub
```

OnLostFocus: Wird aufgerufen, wenn das Formular deaktiviert wird (zum Beispiel, weil ein anderes Fenster in den Vordergrund geklickt wurde), aber nur, wenn es keine weitere Komponente beinhaltet, die den Fokus beim Deaktivieren verlieren könnte.

```
Protected Overrides Sub OnLostFocus(ByVal e As System.EventArgs)
    MyBase.OnLostFocus(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnLostFocus:" + e.ToString)
    End If
End Sub
```

OnLeave: Wird aufgerufen, wenn das Formular deaktiviert wird, aber nur, wenn es mindestens eine weitere Komponente beinhaltet, die den Fokus beim Deaktivieren verlieren kann.

```
Protected Overrides Sub OnLeave(ByVal e As System.EventArgs)
    MyBase.OnLeave(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnLeave:" + e.ToString)
    End If
End Sub
```

Kategorie Tastaturvorverarbeitungsnachrichten des Formulars

```
'******************************************************
'Nachrichtenverarbeitung
'******************************************************
```

ProcessCmdKey: Wird ausgelöst, wenn eine Befehlstaste (z.B. **ALT**+**Anfangsbuchstabe**) gedrückt wurde.

```
Protected Overrides Function ProcessCmdKey(ByRef msg As System.Windows.Forms.Message, _
                            ByVal keyData As System.Windows.Forms.Keys) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessCmdKey:" + _
                msg.ToString + ": KeyData: " + keyData.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funtionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessCmdKey(msg, keyData)
End Function
```

ProcessDialogChar: Wird ausgelöst, wenn eine Dialogtaste (auch Steuerungstaste) gedrückt wurde.

```
Protected Overrides Function ProcessDialogChar(ByVal charCode As Char) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessDialogChar: " + charCode)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funtionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessDialogChar(charCode)
End Function
```

ProcessDialogKey: Wird ausgelöst, wenn eine Dialog-Taste (aber keine Steuerungstaste) gedrückt wurde.

```
Protected Overrides Function ProcessDialogKey(ByVal keyData As System.Windows.Forms.Keys) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessDialogKey:" + _
                ": KeyData: " + keyData.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funtionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessDialogKey(keyData)
End Function
```

ProcessKeyPreview: Wird bei jedem Tastenereignis des Formulars ausgelöst und regelt bei Formularen, ob in Abhängigkeit der `KeyPreview`-Eigenschaft Tastatur-Ereignis-Prozeduren aufgerufen werden.

```
Protected Overrides Function ProcessKeyPreview(ByRef m As System.Windows.Forms.Message) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessKeyPreview:" + m.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funtionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde
    Return False
End Function
```

WndProc: Wird bei jeder Nachricht aufgerufen, die das Fenster in irgendeiner Form betrifft. Um erweiterte Ereignisse selbst auszulösen, überschreiben Sie diese Prozedur. Rufen Sie die Basisfunktion im Anschluss nur dann auf, wenn Sie möchten, dass die Nachrichten, die Sie bereits verarbeitet haben, von der Basisklasse auch verarbeitet werden sollen.

```
Protected Overrides Sub WndProc(ByRef m As System.Windows.Forms.Message)
    If myShowWndProcMessages Then
        Console.WriteLine(m)
    End If
    MyBase.WndProc(m)
End Sub
```

Kategorie Erstellen/Zerstören des Controls (des Steuerelements)

OnHandleCreated: Wird aufgerufen, nachdem das *Window-Handle* für die Steuerelement-Instanz erstellt wurde.

```
Protected Overrides Sub OnHandleCreated(ByVal e As System.EventArgs)
    MyBase.OnHandleCreated(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnHandleCreated")
    End If
End Sub
```

OnCreateControl: Tritt ein, nachdem die Framework-seitigen Ressourcen für das Steuerelement erstellt wurden. Die Basisfunktion muss in den Framework-Versionen 1.0 und 1.1 nicht notwendigerweise aufgerufen werden; aus Aufwärtskompatibilitätsgründen sollte das aber dennoch passieren.

```
Protected Overrides Sub OnCreateControl()
    MyBase.OnCreateControl()
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnCreateControl")
    End If
End Sub
```

OnHandleDestroyed: Wird aufgerufen, wenn das *Window-Handle* zerstört wurde.

```
Protected Overrides Sub OnHandleDestroyed(ByVal e As System.EventArgs)
    MyBase.OnHandleDestroyed(e)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": OnHandleDestroyed")
    End If
End Sub
```

Dispose: Wird aufgerufen, wenn das Steuerelement entweder durch den Garbage Collector oder durch `Dispose` des einbindenden *Controls*/Formulars entsorgt wird.

```
Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
    MyBase.Dispose(disposing)
    If myShowCreationDestroy Then
        Debug.WriteLine(Me.ToString + ": Dispose")
    End If
End Sub
```

Kategorie Mausereignisse des Controls

```
'******************************************************
'Mausereignisse
'******************************************************
```

OnMouseDown: Wird aufgerufen, wenn ein Mausbutton gedrückt wird und sich die Maus über einem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) befindet.

```
Protected Overrides Sub OnMouseDown(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseDown(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format( _
                Me.ToString + ": OnMouseDown: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
                                                , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnClick: Wird aufgerufen, wenn ein Mausklick mit der linken Maustaste über einem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) durchgeführt wird.

```
Protected Overrides Sub OnClick(ByVal e As System.EventArgs)
    MyBase.OnClick(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnClick")
    End If
End Sub
```

OnDoubleClick: Wird aufgerufen, wenn ein Doppelklick mit der linken Maustaste über einem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) durchgeführt wird.

```
Protected Overrides Sub OnDoubleClick(ByVal e As System.EventArgs)
    MyBase.OnDoubleClick(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnDoubleClick")
    End If
End Sub
```

OnMouseUp: Wird aufgerufen, wenn ein Mausbutton losgelassen wird und sich die Maus über einem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) befindet.

```
Protected Overrides Sub OnMouseUp(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseUp(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format( _
                Me.ToString + ": OnMouseUp: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
                                                , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnMouseEnter: Wird aufgerufen, wenn der Mauszeiger den Bereich des *Controls*, aber nicht einen *ChildWindow*-Bereich (andere Komponente) betritt.

```
Protected Overrides Sub OnMouseEnter(ByVal e As System.EventArgs)
    MyBase.OnMouseEnter(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseEnter:" + e.ToString)
    End If
End Sub
```

OnMouseHover: Wird aufgerufen, wenn der Mauszeiger das erste Mal nach dem Betreten des Steuerelement-Bereichs zur Ruhe gekommen ist.

```
Protected Overrides Sub OnMouseHover(ByVal e As System.EventArgs)
    MyBase.OnMouseHover(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseHover:" + e.ToString)
    End If
End Sub
```

OnMouseMove: Wird aufgerufen, wenn der Mauszeiger über dem Bereich des *Controls*, aber nicht über einem *ChildWindow*-Bereich (andere Komponente) bewegt wird.

```
Protected Overrides Sub OnMouseMove(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseMove(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format( _
                Me.ToString + ": OnMouseMove: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
                                        , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

OnMouseLeave: Wird aufgerufen, wenn der Mauszeiger den Bereich des *Controls* verlässt.

```
Protected Overrides Sub OnMouseLeave(ByVal e As System.EventArgs)
    MyBase.OnMouseLeave(e)
    If myShowMouse Then
        Debug.WriteLine(Me.ToString + ": OnMouseLeave:" + e.ToString)
    End If
End Sub
```

OnMouseWheel: Wird aufgerufen, wenn das Mausrad bewegt wird. Wichtig: Alle Komponenten des Formulars empfangen dieses Ereignis!

```
Protected Overrides Sub OnMouseWheel(ByVal e As System.Windows.Forms.MouseEventArgs)
    MyBase.OnMouseWheel(e)
    If myShowMouse Then
        Debug.WriteLine(String.Format( _
            Me.ToString + ": OnMouseWheel: x={0}; y={1}; delta={2}; button={3}; clicks={4}" _
                                , e.X, e.Y, e.Delta, e.Button, e.Clicks))
    End If
End Sub
```

Kategorie Tastaturereignisse des Controls

```
'*******************************************************
'Tastatur
'*******************************************************
```

OnKeyDown: Wird aufgerufen, wenn eine Taste gedrückt wird und das Steuerelement den Fokus hat. Fungiert das Steuerelement als Container (von `Scrollable`- oder `ContainerControl` abgeleitet), verwenden Sie die Tastaturvorverarbeitungsnachrichten-Ereignisse, um die Tastaturereignisse auszuwerten, da sie selbst in diesem Fall nicht ausgelöst werden.

```
Protected Overrides Sub OnKeyDown(ByVal e As System.Windows.Forms.KeyEventArgs)
    MyBase.OnKeyDown(e)
    If myShowKeyboard Then
        Debug.WriteLine(String.Format( _
            Me.ToString + ": OnkeyDown: KeyCode={0}; KeyData={1}; KeyValue={2}; Modifiers={3}", _
                            e.KeyCode, e.KeyData, e.KeyValue, e.Modifiers))
    End If
End Sub
```

OnKeyPress: Wird aufgerufen, wenn eine Taste gedrückt wird und das Steuerelement fokussiert ist; wird nicht aufgerufen, wenn eine Steuerungstaste (wie **Strg** oder **Shift**) alleine oder in Kombination mit einer anderen gedrückt wird. Fungiert das Steuerelement als Container (von `Scrollable`- oder `ContainerControl` abgeleitet), verwenden Sie die Tastaturvorverarbeitungsnachrichten-Ereignisse, um die Tastaturereignisse auszuwerten, da sie selbst in diesem Fall nicht ausgelöst werden.

```
Protected Overrides Sub OnKeyPress(ByVal e As System.Windows.Forms.KeyPressEventArgs)
    MyBase.OnKeyPress(e)
    If myShowKeyboard Then
        Debug.WriteLine(String.Format( _
                    Me.ToString + ": OnKeyPress: KeyChar={0}", _
                    e.KeyChar))
    End If
End Sub
```

OnKeyUp: Wird ausgerufen, wenn eine Taste wieder losgelassen wird und das Steuerelement den Fokus hat. Fungiert das Steuerelement als Container (von `Scrollable`- oder `ContainerControl` abgeleitet), verwenden Sie die Tastaturvorverarbeitungsnachrichten-Ereignisse, um die Tastaturereignisse auszuwerten, da sie selbst in diesem Fall nicht ausgelöst werden.

```
Protected Overrides Sub OnKeyUp(ByVal e As System.Windows.Forms.KeyEventArgs)
    MyBase.OnKeyUp(e)
    If myShowKeyboard Then
        Debug.WriteLine(String.Format( _
                Me.ToString + ": OnKeyUp: KeyCode={0}; KeyData={1}; KeyValue={2}; Modifiers={3}", _
                e.KeyCode, e.KeyData, e.KeyValue, e.Modifiers))
    End If
End Sub
```

Kategorie Größe und Position des Controls

```
'*******************************************************
'Größe und Position
'*******************************************************
```

OnMove: Wird aufgerufen, wenn die Steuerelement-Position verändert wird.

```
Protected Overrides Sub OnMove(ByVal e As System.EventArgs)
    MyBase.OnMove(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnMove:" + e.ToString)
    End If
End Sub
```

OnLocationChanged: Wird aufgerufen, wenn sich die Position des *Controls* verändert hat.

```
Protected Overrides Sub OnLocationChanged(ByVal e As System.EventArgs)
    MyBase.OnLocationChanged(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnLocationChanged:" + e.ToString)
    End If
End Sub
```

OnResize: Wird aufgerufen, wenn sich die Ausmaße des *Controls* ändern. Wenn das Steuerelement seinen Inhalt in Abhängigkeit seiner Größe verändert, sollte an dieser Stelle ein Aufruf an `Invalidate` erfolgen, damit das parallel automatisch ausgelöste `Paint`-Ereignis (nur beim Vergrößern) verhindert wird und stattdessen ein neues `Paint`-Ereignis ausgelöst wird, das dann aber in der Lage ist, den Neuaufbau des *gesamten* Client-Bereichs durchzuführen. Hintergrund: Das Standard-`Paint`-Ereignis kann nur die neu zu zeichnenden Bereiche verarbeiten und wird gar nicht ausgelöst, wenn das Steuerelement nur verkleinert wird (siehe auch ▶ Kapitel 29 und Kapitel 30).

```
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
    MyBase.OnResize(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnResize:" + e.ToString)
    End If
    Invalidate()
End Sub
```

OnSizeChanged: Wird aufgerufen, wenn sich die Ausmaße eines *Controls* geändert haben (siehe auch ▶ Kapitel 29 und Kapitel 30).

```
Protected Overrides Sub OnSizeChanged(ByVal e As System.EventArgs)
    MyBase.OnSizeChanged(e)
    If myShowPositioning Then
        Debug.WriteLine(Me.ToString + ": OnSizeChanged:" + e.ToString)
    End If
End Sub
```

SetBoundsCore: Diese Prozedur dient zweierlei Dingen: Zum einen wird sie von der Basisklasse bei jedem Ereignis aufgerufen, das durch das Ändern der Größe oder der Position des *Controls* aufgerufen wird. Klassen, die diese Routine überschreiben, können die Position und Ausmaße des *Controls* durch das Verändern der Parameter auf der anderen Seite reglementieren. Wenn Sie also beispielsweise nicht wollen, dass die Ausmaße des *Controls* eine bestimmte Größe überschreiten, definieren Sie in dieser Funktion für den entsprechenden Parameter einen neuen Wert, bevor Sie die Basisfunktion mit den geänderten Werten aufrufen. Der Parameter `BoundsSpecified` informiert Sie darüber, welcher Parameter durch ein Ereignis geändert wurde. Ein richtiges Beispiel dafür finden Sie in ▶ Kapitel 30.

```
Protected Overrides Sub SetBoundsCore(ByVal x As Integer, ByVal y As Integer, ByVal width As Integer, _
ByVal height As Integer, ByVal specified As System.Windows.Forms.BoundsSpecified)
    MyBase.SetBoundsCore(x, y, width, height, specified)
    If myShowPositioning Then
        Debug.WriteLine(String.Format( _
            Me.ToString + ": SetBoundsCore: X={0}; y={1}; width={2}; height={3}; specified={4}" _
                        , x, y, width, height, specified))
    End If
End Sub
```

SetClientSize: Wird aufgerufen, wenn sich die Größe des *Controls* durch das Setzen der `ClientSize`-Eigenschaft ändern soll.

```
Protected Overrides Sub SetClientSizeCore(ByVal x As Integer, ByVal y As Integer)
    MyBase.SetClientSizeCore(x, y)
    If myShowPositioning Then
        Debug.WriteLine(String.Format(Me.ToString + ": SetClientSizeCore: X={0}; y={1}", x, y))
    End If
End Sub
```

Kategorie Neuzeichnen des Controls und Anordnen untergeordneter Komponenten

```
'*******************************************************
'Anordnen und Neuzeichnen
'*******************************************************
```

OnInvalidated: Wird aufgerufen, wenn eine Entität das Neuzeichnen des Steuerelement-Inhalts durch `Invalidate` anfordert. Hier im Beispielprogramm geschieht das durch `Resize`, `GotFocus` und `LostFocus`.

```
Protected Overrides Sub OnInvalidated(ByVal e As System.Windows.Forms.InvalidateEventArgs)
    MyBase.OnInvalidated(e)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(String.Format( _
                    Me.ToString + ": OnInvalidated: InvalidRect={0}", _
                    e.InvalidRect))
    End If
End Sub
```

OnLayout: Wird aufgerufen, wenn das Steuerelement anzeigt, dass seine beinhaltenden Steuerelemente aus irgendwelchen Gründen neu angeordnet werden müssen.

```
Protected Overrides Sub OnLayout(ByVal levent As System.Windows.Forms.LayoutEventArgs)
    MyBase.OnLayout(levent)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(String.Format( _
                    Me.ToString + ": OnLayout: AffectedControl={0}; AffectedProperty={1}", _
                    levent.AffectedControl, levent.AffectedProperty))
    End If
End Sub
```

OnPaintBackground: Wird aufgerufen, wenn der Hintergrund des *Controls* neu gezeichnet werden muss.

```
Protected Overrides Sub OnPaintBackground(ByVal pevent As System.Windows.Forms.PaintEventArgs)
    MyBase.OnPaintBackground(pevent)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(Me.ToString + ": OnPaintBackground:" + pevent.ClipRectangle.ToString)
    End If
    DrawControlBackground(pevent.Graphics)
End Sub
```

OnPaint: Wird aufgerufen, wenn der Fensterinhalt neu gezeichnet werden muss.

```
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    MyBase.OnPaint(e)
    If myShowRepaintAndLayout Then
        Debug.WriteLine(Me.ToString + ": OnPaint:" + e.ClipRectangle.ToString)
    End If
    DrawControl(e.Graphics)
End Sub
```

Kategorie Fokussierung des Controls

```
'*******************************************************
'Fokussierung
'*******************************************************
```

OnEnter: Wird aufgerufen, wenn das Steuerelement dabei ist, den Fokus zu erhalten.

```
Protected Overrides Sub OnEnter(ByVal e As System.EventArgs)
    MyBase.OnEnter(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnEnter:" + e.ToString)
    End If
End Sub
```

OnGotFocus: Wird aufgerufen, wenn das Steuerelement fokussiert wird, aber nicht, wenn es ein `ContainerControl` ist, das weitere Komponenten enthält! (In diesem Fall verwenden Sie `OnEnter`, um das Ereignis zu empfangen.)

```
Protected Overrides Sub OnGotFocus(ByVal e As System.EventArgs)
    MyBase.OnLostFocus(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnGotFocus:" + e.ToString)
    End If
    'Das fokussierte Control sieht anders aus als das nicht-fokussierte;
    'deswegen: alles NeuZeichnen
    Invalidate()
End Sub
```

OnLeave: Wird aufgerufen, wenn das Steuerelement dabei ist, den Fokus zu verlieren.

```
Protected Overrides Sub OnLeave(ByVal e As System.EventArgs)
    MyBase.OnLeave(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnLeave:" + e.ToString)
    End If
End Sub
```

OnLostFocus: Wird aufgerufen, wenn das Steuerelement den Fokus verloren hat, aber nicht, wenn es ein `ContainerControl` ist, das weitere Komponenten enthält! (In diesem Fall verwenden Sie `OnLeave`, um das Ereignis zu empfangen.)

```
Protected Overrides Sub OnLostFocus(ByVal e As System.EventArgs)
    MyBase.OnLostFocus(e)
    If myShowFocussing Then
        Debug.WriteLine(Me.ToString + ": OnLostFocus:" + e.ToString)
    End If
    'Das fokussierte Control sieht anders aus als das nicht fokussierte;
    'deswegen: alles neu zeichnen.
    Invalidate()
End Sub
```

Kategorie Tastaturnachrichtenvorverarbeitung des Controls

```
'*******************************************************
'Tastatur-Vorverarbeitung
'*******************************************************
```

ProcessCmdKey: Wird ausgelöst, wenn eine Befehlstaste (z.B. **ALT+Anfangsbuchstabe**) gedrückt wurde.

```
Protected Overrides Function ProcessCmdKey(ByRef msg As System.Windows.Forms.Message, _
                            ByVal keyData As System.Windows.Forms.Keys) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessCmdKey:" + _
                msg.ToString + ": KeyData: " + keyData.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funtionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessCmdKey(msg, keyData)
End Function
```

ProcessDialogChar: Wird ausgelöst, wenn eine Dialogtaste (auch Steuerungstaste) gedrückt wurde.

```
Protected Overrides Function ProcessDialogChar(ByVal charCode As Char) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessDialogChar: " + charCode)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funtionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessDialogChar(charCode)
End Function
```

ProcessDialogKey: Wird ausgelöst, wenn eine Dialogtaste (aber keine Steuerungstaste) gedrückt wurde.

```
Protected Overrides Function ProcessDialogKey(ByVal keyData As System.Windows.Forms.Keys) As Boolean
    If myShowPreProcessing Then
        Debug.WriteLine(Me.ToString + ": ProcessDialogKey:" + _
                ": KeyData: " + keyData.ToString)
    End If
    'Wenn Ihre Instanz eine Nachricht verarbeitet hat, dann geben Sie
    'True als Funtionsergebnis zurück, sonst False. Die Basis rufen Sie nur
    'auf (dann aber auf jeden Fall!), wenn die Nachricht NICHT verarbeitet wurde.
    Return MyBase.ProcessDialogKey(keyData)
End Function
```

Die Steuerungsroutinen des Beispielprogramms

Der Vollständigkeit halber finden Sie an dieser Stelle auch die Listings der Programmabschnitte, die das Testformular ins Leben rufen, den Code des Testformulars selbst und auch den Code der `TestControl`-Klasse, deren Instanz die Anwendung zur Laufzeit erstellt.

Soviel an Information vorweg: Sowohl `TestControl` als auch das Testformular haben zwei zusätzliche Konstruktoren, denen eine Sammlung mit Flags übergeben wird. Diese Flags steuern, welche der Kategorien im Ausgabefenster von Visual Studio protokolliert werden.

Programmcode von frmTest:

```
Public Class frmTest
    Inherits System.Windows.Forms.Form

    Private myShowCreationDestroy As Boolean
    Private myShowMouse As Boolean
    Private myShowKeyboard As Boolean
    Private myShowPositioning As Boolean
    Private myShowRepaintAndLayout As Boolean
    Private myShowFocussing As Boolean
    Private myShowPreProcessing As Boolean
    Private myShowWndProcMessages As Boolean

    '<Vom Windows Form Designer generierter Code (ausgeblendet)>

    Public Sub New(ByVal ShowCreationDestroy As Boolean, _
```

```
                ByVal ShowMouse As Boolean, _
                ByVal ShowKeyboard As Boolean, _
                ByVal ShowPositioning As Boolean, _
                ByVal ShowRepaintAndLayout As Boolean, _
                ByVal ShowFocussing As Boolean, _
                ByVal ShowPreProcessing As Boolean, _
                ByVal ShowWndProcMessages As Boolean)
        MyBase.New()
        myShowCreationDestroy = ShowCreationDestroy
        myShowMouse = ShowMouse
        myShowKeyboard = ShowKeyboard
        myShowPositioning = ShowPositioning
        myShowRepaintAndLayout = ShowRepaintAndLayout
        myShowFocussing = ShowFocussing
        myShowPreProcessing = ShowPreProcessing
        myShowWndProcMessages = ShowWndProcMessages
        InitializeComponent()
    End Sub
```

Steuerungscode von TestControl:

```
Public Class TestControl
    Inherits Control

    Private myShowCreationDestroy As Boolean
    Private myShowMouse As Boolean
    Private myShowKeyboard As Boolean
    Private myShowPositioning As Boolean
    Private myShowRepaintAndLayout As Boolean
    Private myShowFocussing As Boolean
    Private myShowPreProcessing As Boolean

    'Standardkonstruktor: Basiskonstruktor aufrufen
    Public Sub New()
        MyBase.new()
    End Sub

    'Erweiterter Konstruktor: Parameter für die Debug-Ausgaben setzen.
    Public Sub New(ByVal ShowCreationDestroy As Boolean, _
                ByVal ShowMouse As Boolean, _
                ByVal ShowKeyboard As Boolean, _
                ByVal ShowPositioning As Boolean, _
                ByVal ShowRepaintAndLayout As Boolean, _
                ByVal ShowFocussing As Boolean, _
                ByVal ShowPreProcessing As Boolean)
        MyBase.New()
        myShowCreationDestroy = ShowCreationDestroy
        myShowMouse = ShowMouse
        myShowKeyboard = ShowKeyboard
        myShowPositioning = ShowPositioning
        myShowRepaintAndLayout = ShowRepaintAndLayout
        myShowFocussing = ShowFocussing
        myShowPreProcessing = ShowPreProcessing
    End Sub
```

```
'Wird von OnBackgroundPaint aufgerufen, damit der Hintergrund
'des Controls gelöscht wird. Zeichnet hier im Beispiel
'einen gelben Hintergrund.
Protected Overridable Sub DrawControlBackground(ByVal g As Graphics)
    Dim locBrush As New SolidBrush(Color.Yellow)
    g.SetClip(Me.ClientRectangle, Drawing2D.CombineMode.Replace)
    g.FillRectangle(locBrush, Me.ClientRectangle)
End Sub

'Wird von OnPaint aufgerufen, damit das TestControl einen sichtbaren Inhalt hat.
'Zeichnet hier im Beispiel ein umrandetes Kreuz mit einer bestimmten Stiftdicke,
'die von der Fokussierung der Komponente abhängig ist.
Protected Overridable Sub DrawControl(ByVal g As Graphics)
    Dim locPenWidth As Integer
    Dim locClientRecPenWidthIncluded As Rectangle

    'Wenn das Control fokussiert ist,
    If Me.Focused Then
        locPenWidth = 4
    Else
        locPenWidth = 2
    End If

    'Die Dicke des Pens bei den Koordinaten berücksichtigen!
    locClientRecPenWidthIncluded = New Rectangle( _
        Me.ClientRectangle.X + locPenWidth \ 2, _
        Me.ClientRectangle.Y + locPenWidth \ 2, _
        Me.ClientRectangle.Width - locPenWidth, _
        Me.ClientRectangle.Height - locPenWidth)

    'Pen zum Malen.
    Dim locPen As New Pen(Color.Black, locPenWidth)

    'Rahmen zeichnen.
    g.DrawRectangle(locPen, locClientRecPenWidthIncluded)

    'Kreuz malen.
    g.DrawLine(locPen, locClientRecPenWidthIncluded.X, locClientRecPenWidthIncluded.Y, _
            locClientRecPenWidthIncluded.Right, locClientRecPenWidthIncluded.Bottom)

    g.DrawLine(locPen, locClientRecPenWidthIncluded.Right, locClientRecPenWidthIncluded.Y, _
            locClientRecPenWidthIncluded.X, locClientRecPenWidthIncluded.Bottom)

End Sub
```

Programmcode von frmMain:

```
Public Class frmMain
    Inherits System.Windows.Forms.Form

    '<Vom Windows Form Designer generierter Code (ausgeblendet)>

    Private Sub btnCreateWithTestControl_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
                                            Handles btnCreateWithTestControl.Click
        Dim locfrmTest As frmTest
        Dim locTestControl As TestControl
        Dim locButton As Button

        'Einstellungen gelten für das Formular...
        If chkFormular.Checked Then
            locfrmTest = New frmTest( _
                        chkCreateDestroy.Checked, chkMouse.Checked, chkKeyboard.Checked, _
                     chkPositioning.Checked, chkRepaintLayout.Checked, chkFocussing.Checked, _
                     chkPreProcessing.Checked, chkWndProcMessages.Checked)
        Else
            locfrmTest = New frmTest
        End If

        '...und/oder für die TestButton-Komponente.
        If chkSchaltfläche.Checked Then
            locTestControl = New TestControl( _
                        chkCreateDestroy.Checked, chkMouse.Checked, chkKeyboard.Checked, _
                     chkPositioning.Checked, chkRepaintLayout.Checked, chkFocussing.Checked, _
                     chkPreProcessing.Checked)
        Else
            locTestControl = New TestControl
        End If

        'Einstellungen für das Formular durchführen.

        'Das Formular hat ein Viertel der Bildschirmgröße
        'und soll in der Bildschirmmitte des primären Bildschirms erscheinen.
        Dim locBounds As Rectangle = Screen.PrimaryScreen.Bounds
        locfrmTest.Width = locBounds.Width \ 4
        locfrmTest.Height = locBounds.Height \ 4
        locfrmTest.StartPosition = FormStartPosition.CenterScreen
        locfrmTest.Text = txtFormText.Text

        'Einstellungen entsprechend der CheckBox-Controls im Formular
        locfrmTest.ControlBox = chkControlBox.Checked
        locfrmTest.MinimizeBox = chkMinMax.Checked
        locfrmTest.MaximizeBox = chkMinMax.Checked
        locfrmTest.HelpButton = chkHelpButton.Checked
        locfrmTest.ShowInTaskbar = chkShowInTaskbar.Checked
        locfrmTest.TopMost = chkTopMost.Checked
        locfrmTest.KeyPreview = chkKeyPreview.Checked
        locfrmTest.AutoScroll = chkScrollBars.Checked
```

```
    'TestControl mittig und im obeneren Drittel Formular platzieren
    '- nicht zu klein oder zu groß.
    locTestControl.Width = CInt(locfrmTest.ClientSize.Width / 2)
    locTestControl.Height = CInt(locfrmTest.ClientSize.Height / 3)
    locTestControl.Location = _
            New Point(CInt(locfrmTest.ClientSize.Width / 2 - locTestControl.Width / 2), _
                      CInt(locfrmTest.ClientSize.Height / 3 - locTestControl.Height / 2))

    'TestControl verankern, wenn die AutoScroll-Funktion des Formulars nicht gewünscht wird
    If Not chkScrollBars.Checked Then
        locTestControl.Anchor = AnchorStyles.Bottom Or AnchorStyles.Top Or _
                                AnchorStyles.Left Or AnchorStyles.Right
    End If

    'Schließschaltfläche im unteren Drittel positionieren.
    locButton = New Button
    locButton.Width = CInt(locfrmTest.ClientSize.Width / 3)
    locButton.Height = CInt(locfrmTest.Height / 6)
    locButton.Location = _
            New Point(CInt(locfrmTest.ClientSize.Width / 2 - locButton.Width / 2), _
                      CInt(locfrmTest.ClientSize.Height / 4 * 3 - locButton.Height / 2))

    locButton.Text = "Formular &schließen"
    'Schließschaltfläche verankern, wenn die AutoScroll-Funktion des Formulars nicht gewünscht wird.
    If Not chkScrollBars.Checked Then
        locButton.Anchor = AnchorStyles.Bottom Or _
                                AnchorStyles.Left Or AnchorStyles.Right
    End If

    'Return und Escape lösen Click-Ereignis des Buttons aus.
    Me.AcceptButton = locButton
    Me.CancelButton = locButton

    'Zur Laufzeit einstellen, dass das Click-Ereignis der Schließschaltfläche
    'von TestButton-Click behandelt wird.
    AddHandler locButton.Click, AddressOf TestButton_Click

    'Beide Controls der Formular-ControlCollection hinzufügen;
    'damit werden die beiden Komponenten windowstechnisch angelegt und dargestellt.
    locfrmTest.Controls.Add(locTestControl)
    locfrmTest.Controls.Add(locButton)

    locfrmTest.Show()
End Sub

'Ereignis-Routine des Buttons; wird zur Laufzeit eingebunden (s.o.).
Sub TestButton_Click(ByVal Sender As Object, ByVal e As EventArgs)

    Dim locButton As Button

    'Könnte schief gehen, wenn Sender nicht die Schaltfläche ist,
    'deswegen sichergehen durch Try/Catch.
    Try
```

```
            'Das sendende Objekt herausfinden
            locButton = DirectCast(Sender, Button)
        Catch ex As Exception
            Return
        End Try
        'Sendendes Objekt war der Button selbst, dann dessen Parent (das Formular) entsorgen.
        'Damit wird das Formular, das den Button enthält, geschlossen.
        locButton.Parent.Dispose()
    End Sub
End Class
```

Anmerkungen zum Beispielprogramm

Dem einen oder anderen unter Ihnen könnte es sich nicht auf den ersten Blick erschließen, wie der Inhalt von `TestControl` letzten Endes auf den Bildschirm gelangt. Die Zeichenroutinen sind ja offensichtlich die, die sich in `OnPaint` befinden. Das würde ja bedeuten, dass das Steuerelement sich jedes Mal neu zeichnen muss, wenn es beispielsweise durch ein anderes Fenster zuvor verdeckt wurde – und man könnte meinen, dass das sehr lange dauerte. Die Frage, die sich aus diesem Grund vielleicht stellt: Gibt es keinen Weg, den Inhalt von `TestControl` nur einmal zeichnen zu müssen, und der bleibt dann bestehen?

Genau das ist aber die Vorgehensweise, wenn Sie benutzerdefinierte Inhalte in Formularen oder sichtbaren Komponenten (Steuerelementen) anzeigen lassen wollen. Wenn Sie – wie in einem vorherigen Beispiel zu sehen war – beispielsweise die `Image`-Eigenschaft des `PictureBox`-Steuerelements bestimmen, brauchen Sie sich um das ständige Neuzeichnen nicht selbst zu kümmern. Dafür muss das `PictureBox` in der `OnPaint`-Prozedur selbst erledigen. Es gibt unter Windows keine festen Inhalte in *der* Form. Wenn ein Fenster von einem anderen überdeckt wird und dann wieder sichtbar wird, muss es seinen Inhalt neu zeichnen – diese Vorgehensweise muss jedes Windows-Programm anwenden, egal in welcher Sprache es entwickelt wurde. Selbst wenn es so aussieht, als sei der Inhalt »fest«, gibt es dennoch irgendwo eine Routine, die dafür sorgt, dass er nur »fest wirkt«. Das nächste Kapitel und ▶ Kapitel 30 verraten Ihnen weitere Geheimnisse zu diesem Thema.