

# 15 Ereignisse und Delegaten

---

415	<b>Konsumieren von Ereignissen mit WithEvents und Handles</b>
418	<b>Auslösen von Ereignissen</b>
420	<b>Zur-Verfügung-Stellen von Ereignisparametern</b>
423	<b>Dynamisches Anbinden von Ereignissen mit AddHandler</b>
432	<b>Delegaten</b>

---

Ereignisse kennen Sie wahrscheinlich in erster Linie aus der Windows-Programmierung, und zwar wenn es darum geht, auf bestimmte Benutzerereignisse wie das Klicken von Schaltflächen oder das Auswählen von Elementen in einer Liste zu reagieren. Doch das reine Konsumieren der Ereignisse stellt dabei nur die Spitze des Eisberges dar.

Klassen können auch eigene Ereignisse auslösen, die dann wiederum von anderen Klassen benutzt werden. Dabei sollten sich Ereignis auslösende Klassen an bestimmte Regeln halten, von denen später noch die Rede sein wird. Und: Ereignisse können auch nachträglich, also zur Laufzeit, verdrahtet und damit nur im Bedarfsfall konsumiert werden. Damit werden auch Szenarien möglich, die noch in Visual Basic 6.0 ohne die Hilfe von externen DLLs nicht möglich gewesen wären.

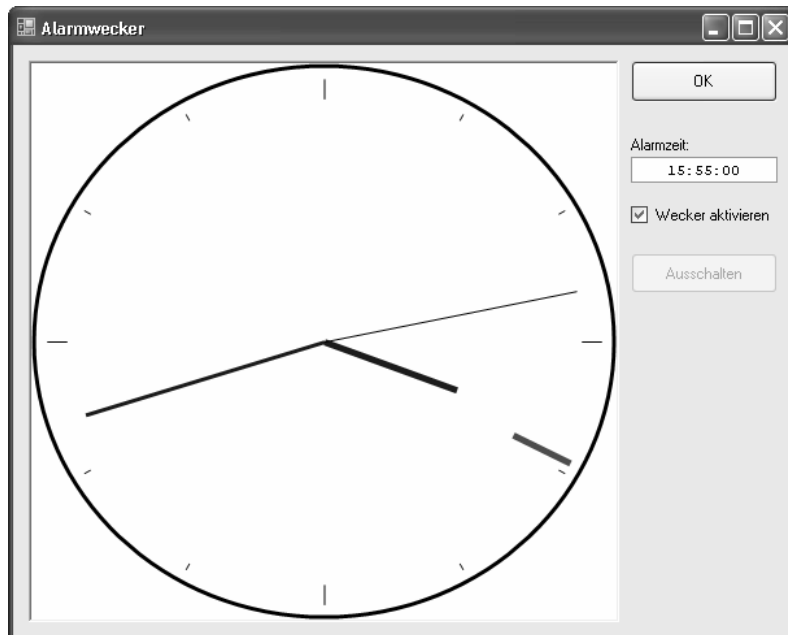
Interessant wird es schließlich mit Delegaten, die – vereinfacht ausgedrückt – so etwas wie Zeiger auf Funktionen bilden, wie man es früher nur von den C-Funktionszeigern kannte. Delegaten können sich dabei sogar polymorph verhalten, was eine enorme Flexibilität bei der Klassenprogrammierung ermöglicht.

---

**BEGLEITDATEIEN:** Der Beispielcode für die zunächst folgenden Abschnitt befindet sich in einem Projekt namens *Alarm.sln*, das Sie im Verzeichnis `\.VB 2005 - Entwicklerbuch\D - OOP\Kap15\Alarm01\` finden.

---

Dieses Beispielprogramm, das in der ersten Version einen einfachen Wecker imitiert, werden wir im Folgenden für die Ereigniserkundigungen verwenden. Wenn Sie dieses Beispielprojekt starten, sehen Sie ein Formular, wie Sie es auch in Abbildung 15.1 erkennen können.



**Abbildung 15.1:** Die Alarmwecker-Anwendung soll Ihnen anschauliche Beispiele für den Umgang mit Ereignissen geben

Ein paar Worte zur groben Funktionsweise des Programms:

Sie haben die Möglichkeit, im Feld *Alarmzeit* eine Uhrzeit im Format *HH:mm:ss*<sup>1</sup> anzugeben. Klicken Sie anschließend auf *Wecker aktivieren*, zeichnet das Programm eine kleine rote Markierung in das Ziffernblatt der Uhr ein, die die Alarmzeit markiert.

Ist diese Zeit erreicht, geht der Wecker los – in diesem Beispiel durch ein rotes Blinken des gesamten Zeichenbereichs der Uhr.

Wie die Uhr tatsächlich in das `PictureBox`-Steuerelement gezeichnet wird, ist in diesem Kapitel von untergeordnetem Interesse. Die Grundlagen über den Umgang mit Zeichenfunktionen des GDI+ lesen Sie in ► Kapitel 29.

Wichtig in diesem Zusammenhang ist eine Klasse, die das »Wecken« übernimmt. Diese Klasse funktioniert im Prinzip wie ein handelsüblicher Alarmwecker im wirklichen Leben. Mit einer bestimmten Eigenschaft stellen Sie die Weckzeit ein, und wenn diese Uhrzeit erreicht ist, löst die Klasse ein Ereignis aus.

Im Grunde genommen gibt es zwei Möglichkeiten, Ereignisse zu konsumieren:

---

<sup>1</sup> Kleiner Tipp am Rande: Die großgeschriebenen »H«s signalisieren, dass es sich um eine Darstellung im 24-Stunden-Format handelt – kleine »h«s würden das 12-Stunden-Format signalisieren. »H« ist dabei die Abkürzung des englischen »Hours« (Stunden), »m« für »Minutes« (Minuten) und »s« für »Seconds« (Sekunden). Für ein Datum wählt man die Formatabkürzungen »y« für »Years« (Jahre), »M« für »Months« (Monate) – großgeschrieben übrigens, um sie von den Minuten zu unterscheiden – sowie »d« für »Days« (Tage). Ein deutsches Datumsformat entspräche demnach *dd.MM.yyyy*, ein typisch amerikanisches *MM/dd/yyyy* (der Monat wird hier zuerst genannt!).

- Durch das Zuweisen einer beliebigen Prozedur mit einer Member-Variable einer Klasse, die mit `WithEvents` deklariert wurde.
- Durch das manuelle Hinzufügen eines Ereignisbehandlers zur Laufzeit mit `AddHandler`.

Eine schnellere Möglichkeit als das Konsumieren von Ereignissen gibt es, wenn in der Ableitung einer Klasse Ereignisauslöser mit `Onxxx`-Methoden implementiert werden. In diesem Fall ergibt es mehr Sinn, die Ereignis auslösenden Routinen zu überschreiben und die Ereignisbehandlung auf diese Weise zu implementieren. Bei Windows Forms-Anwendungen sollte das der bevorzugte Weg sein, um Formularereignisse zu verdrahten.

Und schließlich gibt es auch noch den Weg über so genannte Delegaten, über die im Abschnitt »Delegaten« ab Seite 432 die Rede sein wird.

---

**HINWEIS:** Wie Sie mithilfe von Editor- bzw. Designer Rumpfe von Ereignisbehandlungsroutinen in Ihren Code einfügen, hat ► Kapitel 2 (Abschnitt »Das Eigenschaftfenster«) bereits beschrieben. Dieses Kapitel geht den nächsten Schritt und zeigt, wie die Ereignisbehandlung im Detail funktioniert.

---

## Konsumieren von Ereignissen mit `WithEvents` und `Handles`

Wenn Sie in ein leeres Formular eine Schaltfläche einfügen und auf diese Schaltfläche doppelklicken, sorgt die Visual Studio-IDE dafür, dass der Editor geöffnet, der Formularcode angezeigt und ein Funktionsrumpf in diesem eingefügt wird, der später dann aufgerufen wird, wenn der Anwender zur Laufzeit die Schaltfläche betätigt.

Damit die Behandlung solcher Ereignisse möglich wird, bedarf es dreier Komponenten: Einerseits muss die Objektvariable, die das Ereignis anbietet, mit `WithEvents` deklariert worden sein. Andererseits muss die Signatur<sup>2</sup> der Prozedur, die das Ereignis verarbeiten soll, mit der Signatur des Ereignisses übereinstimmen, die das Objekt anbietet. Und zu guter Letzt muss die Prozedur mit dem Schlüsselwort `Handles` mit dem Objekt ereignis (oder auch mit anderen Ereignissen dieses Objektes bzw. mit den Ereignissen anderer Objekte, falls deren Signaturen dieselben sind) verdrahtet werden.

Die Routinen, die im Code des Beispiels diese Art von Ereignisbehandlung durchführen, finden Sie im Folgenden:

```
'Ereignisbehandlungsroutine, die ausgelöst wird,
'wenn der Alarmgeber Alarm signalisiert, weil eine
'bestimmte Uhrzeit erreicht wurde.
Private Sub myAlarmgeber_Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs) _
    Handles myAlarmgeber.Alarm
    'Wecker schellt!
    myAlarmStatus = True
    'Und zwar so lange.
    myAlarmDownCounter = myAlarmDauer
    'Abschalten sollten wir das Schellen können.
```

<sup>2</sup> Zur Erinnerung: Die Signatur einer Prozedur ergibt sich aus der Abfolge der Typen, die eine Prozedur als Parameter entgegennimmt.

```

    btnAusschalten.Enabled = True
    'Und morgen um die gleiche Zeit, soll er wieder schellen.
    e.Neustellen = True
End Sub

'Ereignisbehandlungsroutine, die ausgelöst wird,
'wenn der Inhalt der PictureBox neu gezeichnet werden soll.
Private Sub picWecker_Paint(ByVal sender As Object, ByVal e As System.Windows.Forms.PaintEventArgs) _
    Handles picWecker.Paint
    If myAlarmgeber IsNot Nothing AndAlso myAlarmgeber.AlarmAktiviert Then
        DrawClock(e.Graphics, Date.Now, myAlarmgeber.Alarmzeit, myAktuelleFarbe)
    Else
        DrawClock(e.Graphics, Date.Now, myAktuelleFarbe)
    End If
End Sub
.
.
.
'Ereignisbehandlungsroutine, die ausgelöst wird,
'wenn der Timer abgelaufen ist. Dies passiert
'alle 500 Millisekunden, und wir zeichnen dann
'die komplette Uhr neu - berücksichtigen dabei
'auch das Hintergrundblinken, falls der "Wecker
'gerade schellt".
Private Sub myTimer_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles myTimer.Tick
    'Läuft der Alarm gerade?
    If myAlarmStatus Then
        'Ja, Farben alle 500 ms wechseln
        If myAktuelleFarbe = Color.White Then
            myAktuelleFarbe = Color.Red
        Else
            myAktuelleFarbe = Color.White
        End If
        'Alarmdauerzähler vermindern
        myAlarmDownCounter -= 1
        If myAlarmDownCounter = 0 Then
            'Alarm ausschalten, wenn dieser abgelaufen ist.
            AlarmAusschalten(True)
        End If
    End If

    'Ganze Uhr in jedem Fall neuzeichnen.
    picWecker.Invalidate()
End Sub

```

In diesem Fall sind es drei Objekte, die im Gültigkeitsbereich der Klasse deklariert wurden und damit Member-Variablen sind, deren Ereignisse von diesen Prozeduren behandelt werden:

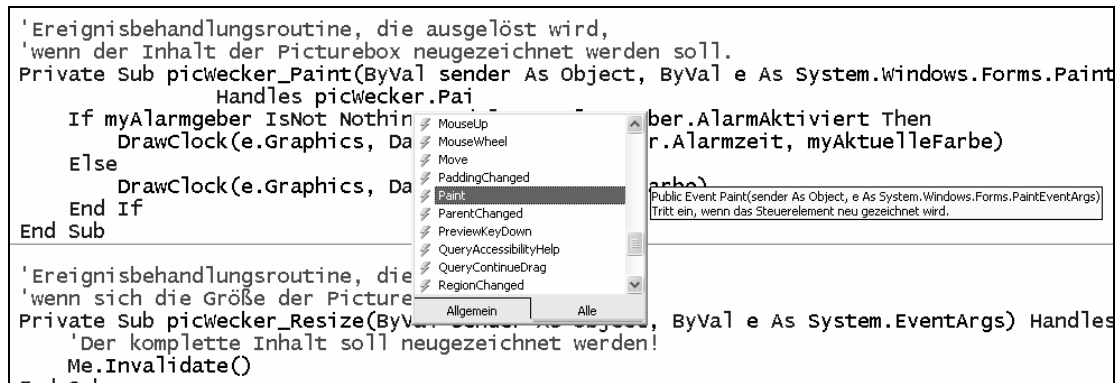
- myTimer,
- picWecker sowie
- myAlarmgeber

Und mit der Ausnahme der Objektvariablen `picWecker`, bei der es sich um eine Objektvariable handelt, die ein Steuerelement repräsentiert und die mithilfe des Designers ihren Weg ins Formular gefunden hat, sind dann auch alle Ereignis anbietenden Variablen mit dem `WithEvents`-Schlüsselwort deklariert worden.

```
Public Class frmMain
```

```
    Private WithEvents myTimer As Timer
    Private WithEvents myAlarmgeber As EinfacherAlarmgeber
```

Erst dann bietet IntelliSense entsprechende Ereignisse zum Objekt an, wenn die Verdrahtung einer Prozedur mit `Handles` erfolgen soll, wie Abbildung 15.2 zeigt.



**Abbildung 15.2:** IntelliSense hilft Ihnen auch beim Verdrahten von Ereignissen mit einer Prozedur – sowohl bei der Auswahl möglicher Ereignis anbietender Objekte als auch bei der Auswahl der Ereignisse

Übrigens: Auch Objektvariablen, die Steuerelemente referenzieren, werden auf die gleiche Weise in den Formularcode eingebunden – Sie können sie jedoch nur nicht auf Anhieb sehen. Der Designer, der ja auch den Formularcode generiert, nutzt nämlich die Möglichkeit von Visual Basic 2005, den Quellcode einer Klasse auf mehrere physische Dateien zu verteilen. Dadurch wird die eigentliche Codedatei eines Formulars aufgeräumter – einige Dinge, die hinter den Kulissen passieren, fehlen aber natürlich in dieser.

Um den Rest des Formularcodes sichtbar zu machen, klicken Sie im Projektmappen-Explorer auf das Symbol *Alle Dateien anzeigen* am oberen Rand dieses Toolfensters (der Tooltip hilft Ihnen beim Finden des richtigen Symbols). Sie werden sehen, dass sich anschließend vor jeder Formulardatei ein kleines Pluszeichen befindet, dessen dahinter stehenden Zweig Sie per Mausklick öffnen können. Sie werden des Weiteren feststellen, dass jedes Formular über eine Datei namens *form.designer.vb* verfügt, der diesen »Hinter-den-Kulissen-Code« beinhaltet. Und hier finden wir auch die Deklaration der verwendeten Steuerelemente – mit dem für die Ereignisverdrahtung notwendigen `WithEvents`:

```

.
. ' Am Ende der Datei frmMain.Designer.vb:
.
Friend WithEvents picWecker As System.Windows.Forms.PictureBox
Friend WithEvents Label1 As System.Windows.Forms.Label
Friend WithEvents mtbAlarmzeit As System.Windows.Forms.MaskedTextBox
Friend WithEvents chkAlarmAktivieren As System.Windows.Forms.CheckBox
Friend WithEvents btnOK As System.Windows.Forms.Button
Friend WithEvents btnAusschalten As System.Windows.Forms.Button

```

```
End Class
```

## Auslösen von Ereignissen

Sie sehen im Codeauszug der Ereignisbehandlungsroutinen, der ab Seite 415 beginnt, wie Ereignisse generell mit bestimmten Prozeduren verknüpft werden. Der fett hervorgehobene Teil dieses Listingausschnittes konsumiert ein Ereignis der Instanz einer Klasse, die nicht Bestandteil des Frameworks ist – diese Klasse ist in Heimarbeit entstanden, und sie soll demonstrieren, wie Klassen Ereignisse auslösen können.

Schauen wir uns dazu den relevanten Klassencode der Datei *Alarmgeber.vb* an:

```

Public Class EinfacherAlarmgeber

    Private WithEvents myTrigger As Timer
    Private myAlarmzeit As Date
    Private myAlarmAktiviert As Boolean
    Private mySchwellwert As Integer = 2

    ''' <summary>
    ''' Wird ausgelöst, wenn eine bestimmte Zeit erreicht wurde.
    ''' </summary>
    ''' <param name="Sender">Das Objekt, das dieses Ereignis ausgelöst hat.</param>
    ''' <param name="e">AlarmEventArgs, die näheres zum Objekt aussagen.</param>
    ''' <remarks></remarks>
    Public Event Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs)

    Sub New(ByVal Alarmzeit As Date)
        Me.Alarmzeit = Alarmzeit
    End Sub

    Sub New(ByVal Alarmzeit As Date, ByVal Aktiviert As Boolean)
        Me.Alarmzeit = Alarmzeit
        Me.AlarmAktiviert = Aktiviert
    End Sub

    .
    .
    .

    Private Sub myTrigger_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles myTrigger.Tick
        If myAlarmzeit < DateTime.Now Then
            Dim locWeckzeitEventArgs As New AlarmEventArgs(Alarmzeit)
            OnWecken(locWeckzeitEventArgs)

```

```

        If locWeckzeitEventArgs.Neustellen Then
            Alarmzeit = locWeckzeitEventArgs.Alarmzeit
        Else
            AlarmAktiviert = False
        End If
    End If
End Sub

''' <summary>
''' Löst das Wecken aus.
''' </summary>
''' <param name="e"></param>
''' <remarks></remarks>
Protected Overridable Sub OnWecken(ByVal e As AlarmEventArgs)
    RaiseEvent Alarm(Me, e)
End Sub
End Class

```

Zunächst einmal benötigt die Klasse selbst eine Hilfe, die für das Auslösen des Ereignisses zur rechten Zeit sorgt, denn sie muss in regelmäßigen Abständen getriggert werden, um herauszufinden, ob die gestellt Alarmzeit bereits erreicht wurde. Dazu verwendet sie ein `Timer`-Objekt namens `myTrigger`, bindet es mit `WithEvents` als Member-Variable ein und sorgt durch die Verdrahtung dessen `Tick`-Ereignisses mit der Prozedur `myTrigger_Tick`, dass diese beim Ablaufen des Timers aufgerufen wird.

Hier findet dann der Vergleich der aktuellen Uhrzeit mit der Weckzeit statt. Und wenn die Weckzeit erreicht wurde, dann wird es interessant:

`OnWecken` wird aufgerufen, und diese Routine sorgt nun dafür, dass das eigentliche Ereignis mit `RaiseEvent` ausgelöst wird. `RaiseEvent` kann alle Ereignisse auslösen, die zuvor auf Klassenebene durch das Event-Schlüsselwort definiert wurden – in unserem Beispiel ist das lediglich das Ereignis `Alarm`.

## Der Umweg über Onxxx

Falls Sie sich nun fragen, wieso `myTrigger_Tick` den Umweg über `OnWecken` nimmt: Denken Sie an OOP und die Polymorphie! Wenn Sie diese Klasse ableiten, und in der abgeleiteten Version dieser Klasse das Weckereignis mitbekommen wollen, dann überschreiben Sie einfach die Methode `OnWecken`. Wenn die Ereignisbehandlungsroutine des `Tick`-Ereignisses `myTrigger_Tick` die Routine `OnWecken` aufruft, ruft es dann nämlich nicht mehr die »Basisversion« sondern Ihre neue Implementierung auf – und Sie bekommen in Ihrer abgeleiteten Klasse das Ereignis auf direktem Wege mit.

Genau das ist das Prinzip bei allen Ereignis auslösenden Komponenten, die Sie im Framework durch Vererbung verwenden – und Formulare gehören übrigens auch dazu. Aus diesem Grund erklärt sich auch die Funktionsweise folgender Routine aus dem Ereignisbeispiel ganz wie von selbst, die die Formularereignisse nicht als Ereignis und mit `Handles` an bestimmte Prozeduren hängen, sondern einfach den Basiscode des Formulars überschreiben: `OnResize` löst nämlich seinerseits das `Resize`-Ereignis aus, das eintritt, wenn das Formular vergrößert oder verkleinert wird.

```

'Wird vom Basisklassenteile von System.Windows.Forms.Form angesprochen,
'wenn sich das Formular vergrößert oder verkleinert hat.
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
    'Wichtig: Basisfunktion aufrufen, sonst wird das
    'Resize-Ereignis nicht mehr ausgelöst!

```

```

MyBase.OnResize(e)
    'Inhalt der PictureBox neuzeichnen,
    'wenn sich die Größe des Formulats geändert hat.
    picWecker.Invalidate()
End Sub

```

---

**HINWEIS:** Dass überschreibbare Routinen, die Ereignisse auslösen, mit »On« beginnen, liegt einfach an der englischen Sprache: »Beim Feststellen der Notwendigkeit zum Auslösen des *Größenändern*-Ereignisses mache Folgendes« würde auf englisch in etwa »On the call of the *resize* event do the following« heißen, oder kurz: »*BeimGrößenändern*« bzw. »*OnResize*«.

---



---

**WICHTIG:** Denken Sie beim Überschreiben von Ereignis auslösenden Onxxx-Methoden in abgeleiteten Klassen unbedingt daran, die Basismethode mit `myBase.Onxxx` aufzurufen. Anderenfalls verhindern sie, dass für Konsumenten von Instanzen Ihrer Klasse Ereignisse ausgelöst werden, da `RaiseEvent` in diesem Fall nicht mehr stattfindet.

---

## Zur-Verfügung-Stellen von Ereignisparametern

Wenn Sie sich die verschiedenen Ereignisbehandlungsroutinen anschauen, werden Sie ein immer wiederkehrendes Schema in den Ereignisnaturen erkennen.

```

Private Sub myTrigger_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles myTrigger.Tick
    .
    .
    .
End Sub

```

- Es gibt einen Parameter – `sender` vom Typ `Object` – der über den Auslöser des Ereignisses Auskunft gibt.
- Es gibt einen Parameter – `e` vom Typ `EventArgs` (oder einer Ableitung von `EventArgs`) –, der entweder nähere Daten zum Ereignis liefert, oder mit dessen Hilfe sich die weitere Ereigniskette steuern lässt.

## Die Quelle des Ereignisses: Sender

`Sender` beinhaltet grundsätzlich die Quelle des Ereignisses als Objekt. Dies ist insbesondere dann wichtig, wenn eine Ereignisbehandlungsroutine gleich mehrere Ereignisse verschiedener Objekte behandeln soll – denn Sie müssen ja wissen, wer für das Ereignis verantwortlich war und dementsprechend reagieren.

---

**BEGLEITDATEIEN:** Falls Sie sich den Code für dieses Beispiel anschauen möchten, finden Sie ihn im Verzeichnis `.\VB 2005 - Entwicklerbuch\D - OOP\Kap15\Ereignistest` in der Projektmappe Ereignisse.

---

Wenn Sie dieses Projekt starten, sehen Sie wie in Abbildung 15.3 ein Formular mit mehreren Schaltflächen.



**Abbildung 15.3:** In diesem Beispiel gibt es eine Ereignisbehandlungsroutine für alle drei Schaltflächen

Soweit scheint es an diesem Beispiel auf den ersten Blick nichts Außergewöhnliches zu geben. Das Besondere ist aber: Alle drei Schaltflächen werden für den Fall des Anklickens vom Code berücksichtigt; es gibt aber lediglich eine einzige Ereignisbehandlungsroutine, die diese Behandlung übernimmt.

```
Public Class frmEreignisse
```

```
    Private Sub Button1Und2Ereignisse(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles Button1.Click, Button2.Click, Button3.Click
```

```
        'Eine MessageBox wird dargestellt, wenn der Anwender Button2 oder Button3 auslöst.
        MessageBox.Show(sender.ToString & "wurde gedrückt!")
```

```
    End Sub
```

```
End Class
```

Hier wird deutlich, wozu sender gut ist: Um überhaupt zu wissen, welche der Schaltflächen das Ereignis ausgelöst hat, müssen Sie sender unter die Lupe nehmen und entsprechende Fallunterscheidungen berücksichtigen, die dann das jeweils Richtige machen.

Übrigens: Da es sich bei sender um keine String-Variable mit beschreibenden Text, sondern um eine wirkliche Referenz auf das Ereignis auslösende Objekt handelt, können Sie sender auch wieder zurück in seinen Ausgangstyp casten.

So ist es im Beispiel ganz sicher, dass ein Typ-Casting nicht schief gehen kann, da ausschließlich Schaltflächen zum Einsatz kommen. Die folgenden Zeilen sind also durchaus denkbar, um von sender »zurück« zum Schaltflächenobjekt (Button-Objekt) zu gelangen und mit diesem Objekt anschließend Manipulationen durchzuführen, die über sender selbst nicht möglich gewesen wären:

```
Public Class frmEreignisse
```

```
    Private Sub Button1Und2Ereignisse(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles Button1.Click, Button2.Click, Button3.Click
```

```
        'Eine MessageBox wird dargestellt, wenn der Anwender Button2 oder Button3 auslöst.
        MessageBox.Show(sender.ToString & " wurde gedrückt!", "Ereignisbehandlung ergab:")
```

```
        'Schaltfläche, die gedrückt wurde, rot einfärben.
```

```
        'FEHLER! So geht es nicht, da Sender vom Typ Object ist:
```

```

    sender.BackColor = Color.Red

    'So geht es, denn es gibt nur Schaltflächen, also
    'ist es sicher in Button zu casten:
    Dim locGedrueckterButton As Button
    locGedrueckterButton = DirectCast(sender, Button)

    'Jetzt klappt es mit dem Roteinfärben
    locGedrueckterButton.BackColor = Color.Red
End Sub
End Class

```

## Nähere Informationen zum Ereignis: EventArgs

EventArgs ist eine Klasse, die ausschließlich dafür gedacht ist, Parameter an Ereignis empfangende Instanzen zu senden. Auch wenn ein Ereignis keine Parameter erforderlich macht, werden Ereignisparameter grundsätzlich mit einer EventArgs-Instanz übergeben – zu diesem Zweck gibt es übrigens eine statische Funktion namens EventArgs.Empty, die ein inhaltsloses aber dennoch instanziiertes EventArgs-Objekt generiert.

Wenn ein Ereignis bestimmte Parameter erforderlich macht, dann vererbt man EventArgs einfach in eine neue Klasse (deren Namen im Übrigen ebenfalls mit »EventArgs« enden sollte) und erweitert diese um die Eigenschaften und Konstruktoren, die erforderlich sind, um die Parameter für das Ereignis zu transportieren.

Eine Instanz von EventArgs sorgt in vielen Fällen aber nicht nur dafür, dass Parameter an die Ereignis einbindenden Instanzen übergeben werden. Eine Ereignis einbindende Instanz kann Parameter einer EventArgs-Instanz auch verändern, um dann ihrerseits zu signalisieren, dass die weitere Ereigniskette in irgendeiner Form gesteuert werden soll.

Zurück zu unserem Alarm-Beispiel, dass auch diese Komponente der Ereignisprogrammierung demonstriert. Wenn Sie sich die Ereignis auslösende Prozedur nochmals vor Augen führen,

```

Private Sub myTrigger_Tick(ByVal sender As Object, ByVal e As System.EventArgs) Handles myTrigger.Tick
    If myAlarmzeit < DateTime.Now Then
        Dim locWeckzeitEventArgs As New AlarmEventArgs(Alarmzeit)
        OnWecken(locWeckzeitEventArgs)
        If locWeckzeitEventArgs.NeuStellen Then
            Alarmzeit = locWeckzeitEventArgs.Alarmzeit
        Else
            AlarmAktiviert = False
        End If
    End If
End Sub

```

stellen Sie fest, dass auch sie von einer benutzerdefinierten EventArgs-Klasse abgeleitet wurde, um für das Ereignis die Parameter Alarmzeit und NeuStellen zur Verfügung zu stellen.

Diese AlarmEventArgs-Klasse hat im Übrigen keine andere Funktion, außer diese beiden Parameter zur Verfügung zu stellen, und Sie finden Sie im Folgenden:

```

Public Class AlarmEventArgs
    Inherits EventArgs

```

```

Private myAlarmzeit As Date
Private myNeuStellen As Boolean

Sub New(ByVal Alarmzeit As Date)
    myAlarmzeit = Alarmzeit
    myNeuStellen = True
End Sub

Sub New(ByVal Alarmzeit As Date, ByVal Neustellen As Boolean)
    myAlarmzeit = Alarmzeit
    myNeuStellen = Neustellen
End Sub

Public Property Alarmzeit() As Date
    Get
        Return myAlarmzeit
    End Get
    Set(ByVal value As Date)
        myAlarmzeit = value
    End Set
End Property

Public Property Neustellen() As Boolean
    Get
        Return myNeuStellen
    End Get
    Set(ByVal value As Boolean)
        myNeuStellen = value
    End Set
End Property
End Class

```

Und beide Codeausschnitte im Kontakt betrachtet zeigen nun, dass Ereignisparameter quasi in beide Richtungen fließen. Eine Prozedur, die das Ereignis verarbeitet, kann durch Setzen der Neustellen-Eigenschaft bestimmen, ob »der Wecker am nächsten Tag zur gleichen Zeit wieder schellen soll«. Wenn das Ereignis ausgelöst und eine Instanz der AlarmEventArgs-Klasse nach »oben hochgereicht« wird, dient ein Parameter (Alarmzeit) also der Ereignis einbindenden Klasse, der andere (Neustellen) der Ereignis auslösenden Klasse, den sie von der Ereignis einbindenden Klasse zurückerhält. Das Setzen dieses Parameters sehen Sie am Beispiel im Listingausschnitt, der auf Seite 418 beginnt (Private Sub myTrigger\_Tick).

## Dynamisches Anbinden von Ereignissen mit AddHandler

Das Einbinden von Ereignissen versagt bei WithEvents, wenn Objekte, die Ereignisse zur Verfügung stellen, nur auf Prozedurebene (also lokal) deklariert werden oder Bestandteil eines Arrays oder einer Auflistung sind.

Bei solchen Konstellationen besteht die Möglichkeit, Ereignisse dynamisch und zur Laufzeit mithilfe von AddHandler zu verdrahten.

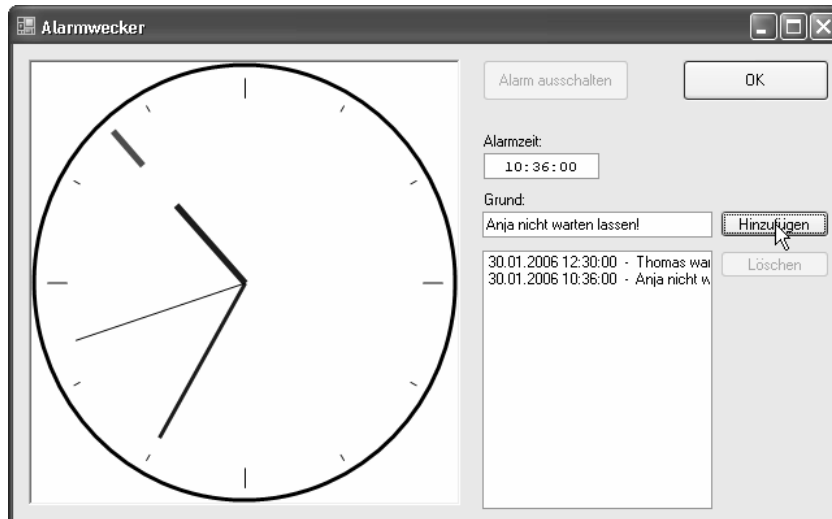
---

**BEGLEITDATEIEN:** Das folgende Beispiel, das Sie im Verzeichnis `.\\VB 2005 - Entwicklerbuch\\D - OOP\\Kap15\\Alarm02` finden, demonstriert ein solches Szenario.

---

Er erweitert die einfache »Alarmwecker«-Anwendung, indem es nicht nur eine einzige Weckzeit sondern gleich eine ganze Reihe von Terminen zu hinterlegen erlaubt.

Wenn Sie dieses Beispiel starten, sehen Sie einen Dialog, wie Sie ihn auch in Abbildung 15.4 sehen können.



**Abbildung 15.4:** Mit der modifizierten »Wecker«-Applikation können Sie gleich mehrere Weckzeiten erfassen ...

Mit dieser Anwendung erfassen Sie nicht nur eine Weckzeit – Sie können gleich mehrere angeben. Außerdem erlaubt diese Version des Beispiels auch einen Alarmtext einzugeben, der im Falle des Eintretens des Ereignisses im Ziffernblatt der Uhr eingeblendet wird – wie Abbildung 15.5 zeigt.

Intern führt diese Version dabei eine Klasse zur Verwaltung einer Liste mit Objektinstanzen ähnlich der `DynamicList`, die Sie schon aus anderen Beispielen vorheriger Kapitel kennen. Doch dieses Mal verwenden wir eine, die bereits fest im Framework eingebaut ist – eine generische Auflistung namens `Collection`.

Die Schwierigkeit hierbei ist, dass wir nun nicht mehr eine globale Variable innerhalb der abgeleiteten `Collection`-Klasse verwenden können, die dann mit `WithEvents` zu deklarieren wäre, denn es geht ja nunmehr um eine *Liste* mit `Alarmgeber`-Instanzen.



**Abbildung 15.5:** ... und ein zusätzlicher Meldungstext wird im Alarmfall zusätzlich im Ziffernblatt eingeblendet

Wir müssen also einen anderen Weg gehen, und der sieht folgendermaßen aus: Wann immer der Anwender einen neuen Wecktermin eingibt, und somit eine neue Alarmgeber-Instanz der Liste mit Add hinzugefügt wird, müssen wir das in der Collection-Ableitung abfangen und einen Ereignisauslöser, den wir in dieser Klasse neu definieren, zur Laufzeit mit dem Alarm-Ereignis der Alarmgeber-Instanz verdrahten.

Umgekehrt müssen wir dafür sorgen, dass beim Löschen eines Alarmgebers aus der Liste zuvor diese Ereignisverknüpfung wieder aufgehoben wird. Und alle Ereignisverknüpfungen der vorhandenen Alarmgeber-Instanzen in der Liste müssen wir dann löschen, wenn sie mit Clear komplett gelöscht wird.

Und das sieht dann codemäßig wie folgt aus:

```
Imports System.Collections.ObjectModel

''' <summary>
''' Verwaltet eine Liste mit Alarmgeber-Objekten, und löst ein Ereignis aus
''' wenn eines der Alarmgeber-Objekte
''' </summary>
''' <remarks></remarks>
Public Class Terminliste
    Inherits Collection(Of Alarmgeber)

    ''' <summary>
    ''' Wird ausgelöst, wenn eines der dieser Instanz hinzugefügten
    ''' Alarmgeber-Objekte seinerseits ein Alarm-Ereignis auslöst.
    ''' </summary>
    ''' <param name="sender">Referenz, auf das Alarmgeber-Objekt, das das Ereignis ausgelöst hat.</param>
    ''' <param name="e">AlarmEventArgs-Instanz, die Parameter zum Ereignis enthalten.</param>
    ''' <remarks></remarks>
    Public Event Alarm(ByVal sender As Object, ByVal e As AlarmEventArgs)
```

```

'Enthält Nothing oder das Datum des als nächstes anstehenden Termin.
Private myNächsterTermin As Nullable(Of Date)

'Wird beispielsweise durch Add oder Insert der Collection-Klasse ausgelöst.
'Überschrieben, da das Alarm-Ereignis des Objektes mit der
'AlarmHandler-Prozedur verknüpft werden muss.
Protected Overrides Sub InsertItem(ByVal index As Integer, ByVal item As Alarmgeber)

    'Das Alarm-Ereignis des Objektes dynamisch mit der Prozedur AlarmHandler verbinden.
    AddHandler item.Alarm, AddressOf AlarmHandler

    'Die Basisprozedur machen lassen, was sie machen muss
    '(nämlich das Element an die richtige Stelle setzen).
    MyBase.InsertItem(index, item)

    'Liste hat sich geändert - der nächste Termin könnte ein anderer werden!
    AktualisiereNächsteTerminEigenschaft()
End Sub

'Wird durch Zuweisung eines Elementes über die Item-Eigenschaft aufgerufen.
'Überschrieben, da das Alarm-Ereignis des Objektes mit der
'AlarmHandler-Prozedur verknüpft werden muss.
Protected Overrides Sub SetItem(ByVal index As Integer, ByVal item As Alarmgeber)

    'Das alte Element an dieser Stelle lösen:
    RemoveHandler Me(index).Alarm, AddressOf AlarmHandler

    'Das neue Element verknüpfen
    AddHandler item.Alarm, AddressOf AlarmHandler

    'Die Basisprozedur machen lassen, was sie machen muss
    '(nämlich das Element an die richtige Stelle setzen).
    MyBase.SetItem(index, item)
End Sub

'Wird beispielsweise durch Remove oder RemoveAt der Collection-Klasse aufgerufen.
'Überschrieben, um das verknüpfte Ereignis wieder mit AlarmHandler zu lösen.
Protected Overrides Sub RemoveItem(ByVal index As Integer)

    'Das Alarm-Ereignis des Objektes dynamisch von der Prozedur AlarmHandler lösen.
    RemoveHandler Me(index).Alarm, AddressOf AlarmHandler

    'Die Basisprozedur machen lassen, was sie machen muss
    '(nämlich das Element aus der Liste löschen).
    MyBase.RemoveItem(index)

    'Liste hat sich geändert - der nächste Termin könnte ein anderer werden!
    AktualisiereNächsteTerminEigenschaft()
End Sub

'Beim Löschen aller Elemente werden die Ereignisse aller Objekte gelöst.
Protected Overrides Sub ClearItems()

```

```

'Alle Ereignisse lösen.
For Each locItem As Alarmgeber In Me
    RemoveHandler locItem.Alarm, AddressOf AlarmHandler
Next

'Basisroutine aufrufen.
MyBase.ClearItems()

'Gibt keinen "nächsten Termin" mehr.
myNächsterTermin = Nothing
End Sub

''' <summary>
''' Löst ein Ereignis aus, sobald diese Routine ihrerseits als
''' Ereignisbehandlungsroutine eines der Elemente in dieser Collection in Aktion tritt.
''' </summary>
''' <param name="sender"></param>
''' <param name="e"></param>
''' <remarks></remarks>
Private Sub AlarmHandler(ByVal sender As Object, ByVal e As AlarmEventArgs)
    RaiseEvent Alarm(sender, e)
End Sub

''' <summary>
''' Sucht den als nächstes anliegenden Termin in der Elementeliste.
''' </summary>
''' <remarks></remarks>
Private Sub AktualisiereNächsteTerminEigenschaft()

    'Keine Elemente vorhanden...
    If Me.Count = 0 Then

        '...dann gibt es keinen nächsten Termin
        myNächsterTermin = Nothing
    Else

        'Alle Elemente durchsuchen, welches Element "jünger"
        'als alle anderen ist.
        myNächsterTermin = Me(0).Alarmzeit
        For Each locItem As Alarmgeber In Me
            If locItem.Alarmzeit < myNächsterTermin.Value Then
                myNächsterTermin = locItem.Alarmzeit
            End If
        Next
    End If
End Sub

''' <summary>
''' Liefert den nächsten anstehenden Termin oder Nothing zurück.
''' </summary>
''' <value></value>
''' <returns></returns>
''' <remarks></remarks>

```

```

Public ReadOnly Property NächsterTermin() As Nullable(Of Date)
    Get

        'Nur Wert zurückliefern. Die Suche nach dem jüngsten
        'Datum wurde schon bei jeder Listenänderung durchgeführt.
        Return myNächsterTermin.Value

    End Get
End Property
End Class

```

Lassen Sie mich zum besseren Verständnis zunächst ein paar Worte zur generischen `Collection` verlieren, die übrigens über den Namespace `System.Collections.ObjectModel` und nicht – wie man vielleicht annehmen könnte – über `System.Collections.Generic` erreichbar ist.<sup>3</sup> Sie tritt an die Stelle unserer bislang verwendeten `DynamicList`. Durch das Vererben dieser Klasse in eine neue Klasse `Terminliste` und das anschließende Überschreiben der Methoden `InsertItem`, `RemoveItem` und `ClearItems` können wir auf das Hinzufügen, das Löschen eines Elementes oder das Löschen aller Elemente dennoch genau so Einfluss nehmen, als hätte wir sie selber entwickelt. Dabei spielt es keine Rolle, ob beispielsweise das Einfügen neuer Elemente etwa durch `Add` oder `Insert` erfolgt – `Collection` sorgt dafür, dass in diesen Fällen `InsertItem` aufgerufen wird und wir so über die Vorgänge der Listenveränderung informiert werden. Mehr zum Thema Auflistungen (*Collections*) erfahren Sie übrigens in ► Kapitel 19.

Wenn nun ein neues Alarmgeber-Objekt der Liste hinzugefügt wurde, ist es wichtig, dass wir dessen Ereignis mitbekommen, damit wir, wenn der Ereignisfall dieses Alarmgeber-Objektes eintritt, wiederum ein Ereignis auslösen können. `WithEvents` scheidet aber, wie schon anfangs erwähnt, in diesem Fall aus, da wir es mit einer lokalen Objektvariable zu tun haben, die wir erst dann »kennen lernen«, wenn sie uns durch `InsertItem` zur Verfügung gestellt wird.

Aus diesem Grund verknüpfen wir das Ereignis dynamisch mit `AddHandler` zur Laufzeit – der erste in Fettschrift gesetzte Block des vorherigen Listingauszugs zeigt, wie das funktioniert. `AddHandler` nimmt zwei Parameter: zum einen das Objekt und dessen Ereignis (in diesem Fall `item.Alarm`), zum anderen einen so genannten Delegaten (dazu später mehr), bei dem es sich in diesem Fall vereinfacht ausgedrückt um die Adresse einer Prozedur handelt, die der Signatur des Ereignisses entspricht, das es zu verknüpfen gilt. Wird später das Alarm-Ereignis für das entsprechende Alarmgeber-Objekt ausgelöst, das wir gerade im Begriff sind zur Liste hinzuzufügen, dann behandelt die Prozedur dieses Ereignis, die im zweiten Parameter von `AddHandler` mit `AddressOf` angegeben wurde.

Im umgekehrten Fall müssen wir dafür sorgen, das Ereignis wieder von dieser Prozedur zu lösen, wenn das Alarmgeber-Element aus der Liste entfernt wird. Dazu verwenden wir das Schlüsselwort `RemoveHandler`, das äquivalent zu `AddHandler` funktioniert. Und die gleiche »Umpolerei« passiert, wenn ein Element der Terminliste durch Zuweisung etwa wie

```
Terminlisteninstanz.Item(x)=NeuesAlarmgeberElement
```

---

<sup>3</sup> Wieso das so ist, können Sie übrigens im Blog von Krzysztof Cwalina nachlesen, den Sie unter dem IntelliLink *D1501* finden. Sein Vorschlag anstelle von `Collection(Of type)` lieber `List(Of Type)` zu verwenden, können wir übrigens nicht in die Tat umsetzen, da `List(Of Type)` uns keinen Einfluss auf das Einfügen oder Entfernen von Elementen nehmen lassen könnte – `List(Of Type)` stellt keine überschreibbaren Methoden zur Verfügung, in die wir uns einklinken könnten.

geändert wird. In diesem Fall trennen wir das Ereignis des bisherigen Elements mit `RemoveHandler` und verknüpfen das neu zugewiesene mit `AddHandler`.

Im Ergebnis erreichen wir damit, dass egal welches `Alarmgeber`-Objekt der Terminliste ein `Alarm`-Ereignis auslöst, immer die Prozedur `AlarmHandler` aufgerufen wird, die ihrerseits dann ein Ereignis auslöst, das anschließend durch das Formular `frmMain` verarbeitet werden kann.

Damit das Hauptprogramm, das ja eine Instanz vom Typ `Terminliste` zur Speicherung der Termine einbindet, es mit dem Einzeichnen des jeweils nächsten Termins möglichst einfach hat, existiert eine Prozedur, die den jeweils nächsten Termin findet (`AktualisiereNächsteTerminEigenschaft`), in einer Member-Variablen (`myNächsterTermin`) ablegt und über eine Eigenschaft (`NächsterTermin`) dem Hauptprogramm zur Verfügung stellt. Das Aktualisieren dieser Variablen findet immer dann statt, wenn sich irgendetwas an der Terminliste geändert hat.

Für diese Member-Variable kommt dabei übrigens der generische Wertetyp `Nullable(Of )` zum Einsatz. Dieser erlaubt es, aus jedem Wertetyp einen »null-baren« Typ zu machen, der nicht nur seinen eigentlichen Wert, sondern auch `Nothing` speichern kann (was Wertetypen, wie Sie wissen, standardmäßig nicht können). Man kann dann mit der Eigenschaft `HasValue` prüfen, ob die `Nullable`-Instanz einen Wert hat (oder eben `Nothing` ist) und diesen mit dessen `Value`-Eigenschaft ermitteln. Da unsere Terminliste natürlich auch leer sein kann, kommen uns die Eigenschaften der generischen `Nullable`-Klasse sehr gelegen, da wir für den jeweils nächsten Termin lediglich ein Datum oder eben `Nothing` speichern müssen, falls die Liste leer ist. Und genau das wäre mit einer Objektvariablen nur vom Typ `Date` eben nicht möglich. Mehr zum Thema `Nullable` erfahren Sie übrigens in ► Kapitel 20.

Die Verarbeitung der Terminliste innerhalb des Hauptprogramms, also in `frmMain`, sieht dann folgendermaßen aus:

```
Public Class frmMain

    Private WithEvents myTimer As Timer
    Private WithEvents myTerminliste As Terminliste

    'Die Hintergrundfarbe der Uhr, die bei
    'anhaltendem Alarm wechselt.
    Private myAktuelleFarbe As Color

    'Alarmdauer in 500ms-Schritten (=25 Sekunden).
    Private myAlarmDauer As Integer = 50

    'Zähler für die Dauer des Restalarms.
    Private myAlarmDownCounter As Integer

    'True: Alarm ist gerade aktiv --> die Uhr blinkt.
    Private myAlarmStatus As Boolean

    ' Ist dieser String nicht Nothing wird eine Textmeldung in der Uhr
    'angezeigt, ansonsten nicht.
    Private myLetzteAlarmmeldung As String

    Sub New()
```

```

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

'Diesen Time benötigen wir zum Darstellen der Uhr
'und zum Blinken der Uhr bei anhaltendem Alarm
myTimer = New Timer()
myTimer.Interval = 500
myTimer.Start()

'Standardhintergrundfarbe der Uhr ist weiß.
myAktuelleFarbe = Color.White

'Terminliste ist zunächst noch leer.
myTerminliste = New Terminliste
End Sub

```

Bis zu diesem Abschnitt findet das Vorgeplänkel statt. Benötigte Member-Variablen werden deklariert, und Sub New sorgt für die korrekte Initialisierung aller Member-Variablen (sowie mit dem Aufruf von InitializeComponent auch für die Ausstattung des Formulars mit allen notwendigen Steuerelementen). Besonders wichtig ist hier natürlich das Timer-Objekt myTimer, das sich um die regelmäßige Aktualisierung der Uhrdarstellung kümmert. Dieser Timer läuft alle 500 ms ab und leitet dann in der myTimer\_Tick-Ereignisbehandlung mit Invalidate ein Neuzeichnen des PictureBox-Inhaltes ein.

Im Unterschied zur vorherigen Version des Programms verwenden wir an dieser Stelle nicht nur ein einfaches Alarmgeber-Objekt sondern die neu implementierte Liste myTerminliste, die mit WithEvents deklariert wurde, damit uns das Alarm-Ereignis erreicht, sobald eines der in ihr enthaltenen Alarm-Objekte seinerseits ein Alarm-Ereignis auslöst. Dieses Ereignis wird im folgenden Codeteil behandelt.

```

'Ereignisbehandlungsroutine, die ausgelöst wird,
'wenn der Alarmgeber Alarm signalisiert, weil eine
'bestimmte Uhrzeit erreicht wurde.
Private Sub myAlarmgeber_Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs) _
    Handles myTerminliste.Alarm
    'Wecker schellt!
    myAlarmStatus = True
    'Und zwar so lange.
    myAlarmDownCounter = myAlarmDauer
    'Abschalten sollten wir das Schellen können.
    btnAlarmAusschalten.Enabled = True

    'Hochgereichten Meldungstext setzen
    myLetzteAlarmmeldung = e.AlarmText

    'Aus Liste löschen
    lstTermine.Items.Remove(Sender)

    'Aus Terminliste löschen
    myTerminliste.Remove(DirectCast(Sender, Alarmgeber))
End Sub

```

Die AlarmEventArgs – Sie sehen es an dieser Stelle – haben ebenfalls eine kleine Überarbeitung erfahren: Sie liefern den Meldungstext direkt mit, sodass dieser dann anschließend in der Mitte des Ziffernblattes angezeigt werden kann.

Wenn nun eines der Alarmgeber-Elemente ein Alarm-Ereignis ausgelöst hat, erfahren wir es über den Umweg Terminliste an dieser Stelle. Wir sorgen dann dafür, dass die »Weckphase« in der Darstellung eingeleitet wird und im Übrigen auch dafür, dass man den zum Termin gehörigen Meldungstext im Ziffernblatt der Uhr sieht (durch Setzen von `myLetzteAlarmmeldung = e.AlarmText` – dieser gesetzte Meldungstext wird dann beim nächsten Neuzeichnen der Uhr berücksichtigt).

```
.
.
.
End Sub

'Wird aufgerufen, wenn der Anwender die Hinzufügen-Schaltfläche betätigt hat
Private Sub btnHinzufügen_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnHinzufügen.Click

    Dim locAlarmzeit As Date

    'Zeit und Termingrund aus den TextBox-Steuerelementen holen.
    If Date.TryParse(mtbAlarmzeit.Text, locAlarmzeit) And _
        Not String.IsNullOrEmpty(txtGrund.Text) Then

        'Neues Alarmgeber-Objekt instanzieren, das wir anschließend
        'direkt zur ListBox...
        Dim locAlarm As New Alarmgeber(locAlarmzeit, txtGrund.Text, True)
        lstTermine.Items.Add(locAlarm)

        '...sowie zur Terminliste hinzufügen.
        myTerminliste.Add(locAlarm)

        'Uhr Neuzeichnen, damit die Weckzeit des nächsten Termins als
        'roter Strich ins Ziffernblatt kommt!
        picWecker.Invalidate()
    Else

        'Ups! Solch eine Uhrzeit gibt es nicht - TryParse
        'ist fehlgeschlagen.
        MessageBox.Show("Bitte überprüfen Sie die Eingabe auf Fehler!")
    End If
End Sub
```

Neu damit erwähnenswert ist die oben stehende Routine, die dafür sorgt, dass ein neues Alarmgeber-Objekt erstellt und der Liste hinzugefügt wird, wenn der Anwender die Daten für einen neuen Termin erfasst und anschließend auf *Hinzufügen* klickt.

Hier wird übrigens deutlich, dass das Programm mit einem kleinen Manko zu kämpfen hat, denn die Referenzen auf die eigentlichen Termine – die Alarmgeber-Objekte – werden im Grunde genommen in zwei Listen gespeichert: in der `Collection(Of Alarmgeber)`-Auflistung und in der internen Liste des `ListBox`-Steuerelements `lstTermine`. Die `ListBox`-Liste benötigen wir aber einerseits für die Listendarstellung der Termine; die `Collection(Of Alarmgeber)`-Auflistung dafür, dass wir »mitbekommen«,

wenn eines der in ihr enthaltenen Alarmgeber-Objekte ein Ereignis ausgelöst hat. Dementsprechend müssen wir beide Listen pflegen, wenn der Anwender einen neuen Termin erfasst oder einen bereits erfassten Termin wieder aus der Liste löscht.

Ein Ansatz, dieses Manko zu umgehen, wäre es, die komplette `AddHandler`-Logik in das Hauptprogramm zu verlegen. Das würde allerdings dem OOP-Anspruch, wieder verwendbare Komponenten zu schaffen, nicht gerade entsprechen.

Eine andere Möglichkeit stellen so genannte Delegaten dar, von denen im folgenden Abschnitt die Rede ist.

## Delegaten

Im vorherigen Abschnitt haben Sie den Einsatz des `AddressOf`-Operators im Zusammenhang mit `AddHandler` und `RemoveHandler` kennen gelernt. Der `AddressOf`-Operator ermittelt sozusagen die Speicheradresse einer Prozedur, an der diese später nach dem Kompilieren bzw. nach der Verarbeitung durch den JITter im Arbeitsspeicher steht. Nur so wird die Verdrahtung eines Ereignisses mit einer Prozedur zur Laufzeit möglich.

Denn das, was `AddressOf Prozedurname` eigentlich ermittelt, ist etwas, was man als eine Delegaten-Konstante bezeichnen könnte. Ein Delegat speichert die Position einer Prozedur im Programm und erlaubt auch deren Zuweisung an eine Variable. Damit wird es nicht nur möglich, zur Laufzeit zu entscheiden, welche Prozeduren zu einem bestimmten Zeitpunkt aufgerufen werden sollen, man kann sogar dafür sorgen, dass Prozeduren aufgerufen werden, die Sie zur Entwurfszeit noch gar nicht kennen.

Zugegeben: Ohne Beispiel ist diese Erklärung reichlich abstrakt. Werfen Sie am besten einmal einen Blick auf den folgenden kleinen Formularcode, der in Abhängigkeit des Klickens auf eine von zwei vorhandenen Schaltflächen unterschiedliche Prozeduren innerhalb des Formular-Codes aufruft – was im Grunde genommen noch nichts Besonderes wäre, würde dieses Beispiel das Problem nicht mit der Hilfe von Delegaten lösen:

---

**BEGLEITDATEIEN:** Den Code für dieses Beispiel finden Sie im Projekt namens *Delegate.sln* im Verzeichnis `VB 2005 - Entwicklerbuch\D - OOP\Kap15\Delegate`.

---

```
Public Class Form1
```

```
    Delegate Sub TestDelegate(ByVal Text As String)
```

```
    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _  
        Handles Button1.Click, Button2.Click
```

```
        Dim locDel As TestDelegate
```

```
        If sender Is Button1 Then
```

```
            locDel = AddressOf BehandlungsroutineButton1
```

```
        Else
```

```
            locDel = AddressOf BehandlungsroutineButton2
```

```
        End If
```

```
        locDel.Invoke("Dies kam über einen Delegaten!")
```

```

End Sub

Private Sub BehandlungsroutineButton1(ByVal Text As String)
    MessageBox.Show("Behandlungsroutine für Button1 sagt: " & Text)
End Sub

Private Sub BehandlungsroutineButton2(ByVal Text As String)
    MessageBox.Show("Behandlungsroutine für Button2 sagt: " & Text)
End Sub
End Class

```

Interessant für dieses Beispiel sind zunächst nur die beiden ersten Schaltflächen. Wenn Sie das Programm starten, werden in Abhängigkeit von der ausgelösten Schaltfläche entweder die Methoden `BehandlungsroutineButton1` bzw. `BehandlungsroutineButton2` aufgerufen. Soweit ist das noch nichts Besonderes. Doch nun schauen Sie sich an, wie diese Behandlungsroutinen aufgerufen werden: Innerhalb der `Click`-Ereignisbehandlungsroutinen für beide Schaltflächen wird eine spezielle Variable deklariert (`locDel`), die in der Lage ist, quasi die Prozeduren an sich zu speichern, die aufgerufen werden sollen. Im Grunde genommen ist diese Erklärung aber Unsinn, denn Variablen können natürlich keine Prozeduren speichern. Was Variablen allerdings speichern können, sind die Startadressen der Prozeduren, an denen ihr Code später im Arbeitsspeicher beginnt. Diese spezielle Variable `locDel` ist im Beispiel vom Typ `TestDelegat`. Und wenn Sie nun an den Anfang des Klassencodes schauen, sehen Sie die Definition dieses Typen: `TestDelegat` wird einerseits als Sub (also als Methode ohne Rückgabewert) und andererseits mit einer bestimmten Signatur definiert (sie erwartet einen einzigen Parameter vom Typ `String`). Diese Signatur entspricht exakt den beiden Behandlungsroutinen, deren Adresse wir später in einer Variable von dem Typ `TestDelegat` speichern, den wir an dieser Stelle mit der `Delegate`-Anweisung definieren.

Möchten wir nun, nachdem die Delegatvariable `locDel` deklariert und ihr anschließend die Adresse einer signaturgleichen Prozedur zugewiesen wurde, diese zugewiesene Prozedur auch tatsächlich aufrufen, genügt dazu ein simpler `Invoke`-Methodenaufruf.

In diesem Beispiel, das lediglich den grundsätzlichen Umgang mit Delegaten verdeutlichen soll, ergibt diese Vorgehensweise natürlich noch nicht wirklich Sinn – die beiden Behandlungsroutinen hätten natürlich auch direkt aus dem Programm heraus ohne das Zur-Hilfe-Nehmen eines Delegaten aufgerufen werden können.

Doch Sie ahnen sicherlich bereits, was der Einsatz von Delegaten ermöglicht: Mit ihrer Hilfe haben Sie nämlich die Möglichkeit, dafür zu sorgen, dass Sie Methoden aufrufen, die Sie zur Entwurfszeit Ihrer Klasse noch gar nicht kennen. Und genau das eröffnet eine Alternative zu Ereignissen.

Sie könnten beispielsweise eine Eigenschaft in Ihrer Klasse zur Verfügung stellen, die vom Typ eines Delegaten ist. Eine Klasse, die Ihre Klasse einsetzt, könnte diese Eigenschaft dann mit einem Wert belegen, der eine Prozedur dieser Ihre Klasse einbindenden Klasse darstellt. Innerhalb Ihrer Klasse könnten Sie dann zur gegebenen Zeit dafür sorgen, dass – sofern die Delegaten-Eigenschaft definiert wurde (also nicht `Nothing` ist) – die Prozedur per `Invoke` aufgerufen wird. Welche Prozedur dabei genau dahinter steckt, wissen Sie zum Zeitpunkt, zu dem Sie Ihre Klasse entwickeln, noch gar nicht – genau so, wie es bei Ereignissen der Fall ist (Sie wissen bei der Implementierung eines Ereignisses ebenfalls nicht, »wer« dieses Ereignis Ihrer Klasse später konsumieren wird).

Mit dem Wissen können wir das schon bekannte Weckerbeispiel auf Delegaten umstellen. Sie sorgen in unserem Fall sogar dafür, dass das Programm ein wenig kompakter wird, denn: Die Notwendig-

keit, eine zweite Klasse zu entwerfen, die eine Liste mit Alarmgeber-Objekten enthält, deren Ereignisse dann wieder ein Ereignis auslösen und so »weiter nach oben delegiert« werden, fällt weg.

Stattdessen genügt es, die Alarmgeber-Klasse mit einem Delegaten auszustatten. Das Hauptprogramm kann dann die verschiedenen Alarmgeber-Instanzen direkt in der ListBox halten und braucht keine weitere Liste, die die Ereignisse nach oben hinauf reicht. Sie übergibt jeder Alarmgeber-Instanz stattdessen mit AddressOf die Prozedur, die im Fall des Erreichens der Weckzeit *direkt* aufgerufen werden soll. Und anstatt ein Ereignis auszulösen, überprüft die Alarmgeber-Instanz, ob der Delegat, den sie zur Verfügung stellt, definiert wurde, ruft im positiven Fall Invoke des Delegaten und damit direkt die Prozedur des Hauptprogramms auf, die dann schließlich dafür sorgt, dass der Alarm visualisiert wird.

---

**BEGLEITDATEIEN:** Das auf Delegatengebrauch umgestellte Beispiel finden Sie im Verzeichnis `.\\VB 2005 - Entwicklerbuch\\D - OOP\\Kap15\\Alarm03`, und es demonstriert ein solches Szenario.

---

Betrachten wir zunächst die relevanten Codeteile, mit denen die Alarmgeber-Klasse um den Einsatz mit Delegaten erweitert wird (nicht relevante Codeteile und Kommentare sind hier aus Platzgründen ausgelassen):

```
Public Class Alarmgeber
```

```
    Public Delegate Sub AlarmDelegate(ByVal sender As Object, ByVal e As AlarmEventArgs)
```

```
    Private WithEvents myTrigger As Timer
    Private myAlarmText As String
    Private myAlarmzeit As Date
    Private myAlarmAktiviert As Boolean
    Private mySchwellwert As Integer = 2
    Private myAlarmDelegate As AlarmDelegate
```

In dieser Klasse wird der Delegatentyp als Erstes definiert, damit einer einbindenden Klasse vorgeschrieben wird, wie die Signatur einer Prozedur auszusehen hat, die ein Alarmgeber-Objekt im »Alarmfall« mithilfe des Delegaten aufrufen soll. Anschließend definiert die Klasse den eigentlichen Delegaten als solchen Delegatentyp – dieser speichert später die eigentliche Rückrufadresse der Prozedur.

```
    ''' <summary>
    ''' Wird ausgelöst, wenn eine bestimmte Zeit erreicht wurde.
    ''' </summary>
    ''' <param name="Sender">Das Objekt, das dieses Ereignis ausgelöst hat.</param>
    ''' <param name="e">AlarmEventArgs, die näheres zum Objekt aussagen.</param>
    ''' <remarks></remarks>
    Public Event Alarm(ByVal Sender As Object, ByVal e As AlarmEventArgs)

    Sub New(ByVal Alarmzeit As Date, ByVal AlarmText As String, ByVal AlarmRückrufroutine As AlarmDelegate)
        Me.Alarmzeit = Alarmzeit
        Me.AlarmText = AlarmText
        myAlarmDelegate = AlarmRückrufroutine
    End Sub
```

```

Sub New(ByVal Alarmzeit As Date, ByVal AlarmText As String, ByVal Aktiviert As Boolean, _
        ByVal AlarmRückrufoutine As AlarmDelegate)
    Me.Alarmzeit = Alarmzeit
    Me.AlarmAktiviert = Aktiviert
    Me.AlarmText = AlarmText
    myAlarmDelegate = AlarmRückrufoutine
End Sub
.
.
.

```

Eine Instanz der Alarmgeber-Klasse erhält die Rückrufprozedur ausschließlich über den Konstruktor. Unser Hauptprogramm muss also, wenn es eine Alarmgeber-Klasse instanziiert, mit dem AddressOf-Operator festlegen, welche seiner Prozeduren im Alarmfall aufgerufen werden soll.

```

Protected Sub OnWecken(ByVal e As AlarmEventArgs)
    If myAlarmDelegate IsNot Nothing Then
        myAlarmDelegate.Invoke(Me, e)
    End If
    RaiseEvent Alarm(Me, e)
End Sub
.
.
.
End Class

```

Der Rest des Klassencodes bleibt, wie er ist – mit der Ausnahme der Methode OnWecken, die bislang lediglich das Alarm-Ereignis auslöste. Diese Methode kümmert sich in der neuen Version um eine weitere Sache: Sie ruft Invoke des Delegates auf und damit die zuvor mit AddressOf übergebene Prozedur.

Das Anlegen einer Instanz der Alarmgeber-Klasse im Hauptprogramm wiederum passiert dann, wenn der Anwender einen neuen Termin erfasst. Der entsprechende Code des Hauptprogramms sieht mit den notwendigen Änderungen dann folgendermaßen aus:

```

'Wird aufgerufen, wenn der Anwender die Hinzufügen-Schaltfläche betätigt hat
Private Sub btnHinzufügen_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnHinzufügen.Click

    Dim locAlarmzeit As Date

    'Zeit und Termingrund aus den TextBox-Steuer-elementen holen.
    If Date.TryParse(mtbAlarmzeit.Text, locAlarmzeit) And _
        Not String.IsNullOrEmpty(txtGrund.Text) Then

        'Neues Alarmgeber-Objekt instanzieren, das wir anschließend
        'direkt zur ListBox hinzufügen. Hierbei wird auch gleichzeitig
        'die Rückruf-Prozedur angegeben, die angesprungen wird,
        'wenn der Alarm ausgelöst wurde.
        Dim locAlarm As New Alarmgeber(locAlarmzeit, txtGrund.Text, True, AddressOf myAlarmgeber_Alarm)
        lstTermine.Items.Add(locAlarm)
    End If
End Sub

```

```

'Den nächsten Termin ermitteln, damit er eingezeichnet wird
'(oder eben nicht, wenn es keinen mehr gibt!)
FindeNächstenTermin()

'Uhr Neuzeichnen, damit die Weckzeit des nächsten Termins als
'roter Strich ins Ziffernblatt kommt!
picWecker.Invaliddate()
Else

'Ups! Solch eine Uhrzeit gibt es nicht - TryParse
'ist fehlgeschlagen.
MessageBox.Show("Bitte überprüfen Sie die Eingabe auf Fehler!")
End If
End Sub

```

Sie sehen hier zweierlei: zum einen, dass die zusätzliche Terminliste zur Speicherung der Alarmgeber-Instanzen (der Termine) nun nicht mehr benötigt wird. Alle Termine werden als Alarmgeber-Instanzen direkt im `ListBox`-Steuerelement `lstTermine` gespeichert. Zum anderen wird dem erweiterten Konstruktor nun mit `AddressOf` die Adresse der vormaligen Ereignisbehandlungsroutine des Alarm-Ereignisses übergeben. Diese Prozedur wird durch `Invoke` des Delegaten (siehe vorheriges Listing) indirekt aufgerufen, wenn das Alarmgeber-Objekt den Alarm auslöst.

Durch den Einsatz von Delegaten erreichen wir in diesem Fall einen wesentlich kompakteren Code: Die komplette Implementierung der `Collection(Of Alarmgeber)`-Liste `Terminliste` ist weggefallen und damit natürlich auch die Anforderung, zwei Listen homogen zu pflegen. Zusätzlich ist der Einsatz von Delegaten auch schneller – zwar nur für den Bruchteil von Millisekunden auf modernen Rechnern und damit für dieses Beispiel nicht sonderlich relevant – aber wenn Sie zeitkritische und häufige Ereignisaufrufe durchführen müssen, kann das schon ein ernst zu nehmender Entscheidungsfaktor für den Einsatz von Delegaten werden.

Ein weiteres Beispiel für Delegaten finden Sie übrigens in ► Kapitel 21, das einen Formelparser zum Berechnen beliebiger mathematischer Ausdrücke vorstellt, den Sie mithilfe von Delegaten um eigene Funktionen erweitern können. Und ► Kapitel 20 zeigt im Rahmen von generischen Auflistungen ebenfalls einige besondere Methoden, die sich Delegaten bedienen.

---

**HINWEIS:** In Delegaten gespeicherte Prozeduren lassen sich auf asynchron aufrufen, was bedeutet, dass sie im Sinne des Multitasking als neuer Thread initiiert werden. Da diese Vorgehensweise zum Thema Threading gehört, finden Sie es im entsprechenden ► Kapitel 31 beschrieben.

---