Auszug aus

VB .NET

Objektorientiertes Programmieren in VB und Einstieg in die .NET-Klassenbibliothek

von Andreas Kühnel

Galileo Computing
960 S., 2002, geb., mit CD
49,90 Euro, ISBN 3-89842-129-5

Andreas Kühnel: VB .NET - Objektorientiertes Programmieren in VB und Einstieg in die .NET-Klassenbibliothek, 960 S., ISBN 3-89842-129-5, © Galileo Press 2002

7.2 Delegates

7.2.1 Problembeschreibung

Stellen Sie sich vor, Sie hätten den Auftrag bekommen eine Software zu entwickeln, die eine Pumpenanlage zum Befüllen des Schwimmbeckens eines Schwimmbades ansteuert. Es handelt sich bei dieser Anlage um Pumpen verschiedener Hersteller in Reihenschaltung. Grundsätzlich sollen alle Pumpen eingeschaltet werden, wenn das Becken gefüllt wird. Anzahl und Typ der Pumpen können dabei durchaus variieren, und Ihre Software soll dabei so flexibel sein, sich an diese Änderungen anpassen zu können.

Wie kann das Problem am besten gelöst werden?

Man kann davon ausgehen, dass die Ansteuerung jeder Pumpe auf eine andere Weise erfolgen muss. Daher bietet es sich an, für jede infrage kommende Pumpe eine eigene Klasse mit einer Methode zu entwickeln, aus der heraus die Pumpe gestartet wird. Wir wollen zunächst beispielhaft zwei Klassen entwickeln, *PumpeA* und *PumpeB*, deren Methoden *SwitchOnA* und *SwitchOnB* für die komplexen Einschaltvorgänge stehen sollen.

```
Public Class PumpeA
   Public Sub SwitchOnA()
      Console.WriteLine("Pumpe A wird eingeschaltet.")
   End Sub
End Class
```

```
Public Class PumpeB
   Public Sub SwitchOnB()
      Console.WriteLine("Pumpe B wird eingeschaltet.")
   End Sub
End Class
```

Belassen wir es bei diesen beiden Pumpen; sie werden in ihrer Einfachheit im späteren Gesamtkomplex dazu beitragen, die Thematik zu verstehen. Diese beiden Pumpen sollen nun der Reihe nach eingeschaltet werden. Dazu brauchen wir weiteren Programmcode, aus dem heraus die installierten Pumpen gestartet werden können. Es bietet sich hier vielleicht an, eine Klasse zu implementieren, die die Ansteuerung des Startvorgangs aller Pumpen übernimmt. Eine weitere Klasse ist als Komponente in der Benutzeranwendung implementiert und ruft eine Methode in der pumpensteuernden Klasse auf, mit der letztendlich der Startvorgang in Gang gesetzt wird.

Wenden wir uns zunächst der erstgenannten Klasse zu. Um alle Pumpen einzuschalten, muss die pumpenspezifische Methode aufgerufen werden, die diese

Funktionalität sicherstellt. Beachten Sie in diesem Zusammenhang, dass die Methoden in den oben definierten Klassen nicht gleichnamig sind, obwohl dies über die Implementierung einer Schnittstelle sehr einfach zu erreichen wäre.

Im einfachsten Fall könnte der Code wie folgt lauten:

```
Public Class ControlPumps
   Public Sub StartAllPumps()
      Dim p1 As New PumpeA()
      Dim p2 As New PumpeB()
      p1.SwitchOnA()
      p2.SwitchOnB()
   End Sub
End Class
```

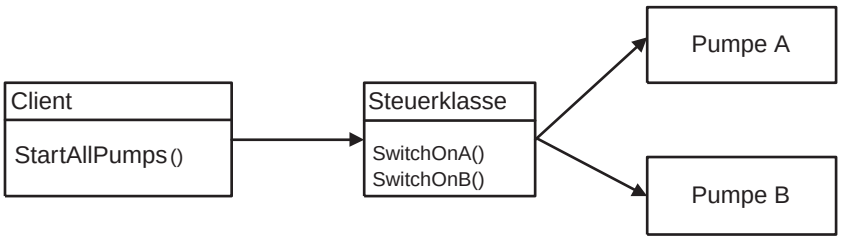


Abbildung 7.1 Das Starten der Pumpen über einen Client und einer steuernden Klasse

Ein Client könnte nun mit

```
Dim obj As New ControlPumps()
obj.StartAllPumps()
```

zwar das Füllen des Beckens in die Wege leiten, aber diesem Ansatz haftet ein ganz wesentlicher Nachteil an: Ihm fehlt die Flexibilität, der Steuerung eine oder auch mehrere Pumpen dynamisch hinzufügen zu können. Die Klasse *ControlPumps*, die den Kern der gesamten Anwendung darstellt, müsste mit jeder neu installierten Pumpe ausgetauscht werden. Das gilt selbstverständlich auch, wenn eine Pumpe deinstalliert wird. Das ist aber nicht das, was wir tatsächlich anstreben, und daher müssen wir uns also etwas anderes, besseres einfallen lassen.

7.2.2 Ein erster Lösungsansatz

Ein dynamisches Array, das von dem Client mit allen zur Verfügung stehenden Pumpen gefüllt wird, und dessen Elemente – also die Pumpenobjekte – von der steuernden Klasse durchlaufen werden, könnte die Lösung sein. In Kapitel 6.5 haben Sie bereits die prinzipiellen Vorteile und die einfache Handhabbarkeit spe-

zieller Objektauflistungen (Collections) kennen gelernt. Wir wollen uns diese Vorteile auch in unserer Klasse *ControlPumps* zunutze machen und wie im Beispiel der Aggregation die Klasse **System.Collections.ArrayList** aggregieren, um vom Benutzer aus die Auflistung mit allen aktuell vorhandenen Pumpen aufzufüllen. Der Code der überarbeiteten Klasse *ControlPumps* könnte dann wie folgt lauten:

```
Public Class ControlPumps
   Private col As New System.Collections.ArrayList()

   Public Sub AddPump(ByVal newPump As Object)
      col.Add(newPump)
   End Sub

   Public Sub StartAllPumps()
      Dim obj As Object
      For Each obj In col
         If TypeOf obj Is PumpeA Then
            obj.SwitchOnA()
         ElseIf TypeOf obj Is PumpeB Then
            obj.SwitchOnB()
         End If
      Next
   End Sub
End Class
```

Bei der Instanzierung eines Objekts vom Typ *ControlPumps* wird die private Variable *col* initialisiert, die auf ein Objekt vom Typ **ArrayList** verweist. Die Methode *AddPump* wird vom Benutzer für jede Pumpe aufgerufen, die eingeschaltet werden soll und hängt das beim Aufruf übergebene Objekt der aggregierten Auflistung an. Sind alle Pumpen in *ControlPumps* bekannt, kann die Methode *StartAllPumps* ausgeführt werden, die ihrerseits die gesamte Collection durchläuft und die spezifische Startmethode jedes Objekts ausführt.

Sehen wir uns die Schleife einmal etwas genauer an. Speziell für Objektauflistungen bietet sich die Variante `For Each...Next` an, deren allgemeine Syntax lautet:

```
For Each <Schleifenvariable> In <Auflistung>
   'Anweisungen
[Exit For]
   'Anweisungen
Next <Element>
```

In *ControlPumps* werden von der Auflistung *col* unterschiedliche Typen verwaltet. Da während der Schleifendurchläufe ausnahmslos jedes Objekt in der Auflistung erfasst wird, muss der Typ der Schleifenvariablen von einem Typ sein, in den jedes

verwaltete Objekt implizit konvertiert werden kann. Eine gemeinsame Basisklasse gibt es für *PumpeA* und *PumpeB* jedoch nicht, was dazu führt, dass die Schleifenvariable vom Typ **System.Object** deklariert werden muss. Die Schleife durchläuft jedes Element der angegebenen Auflistung und setzt dessen Referenz in die Schleifenvariable ein. Auf die Referenz lässt sich eine Methode des Objekts aufrufen.

Wenden wir uns nun dem Code des Clients zu, der wie folgt lautet:

```
Public Sub Main()
   Dim obj As New ControlPumps()
   With obj
      .AddPump(New PumpeA())
      .AddPump(New PumpeB())
      .AddPump(New PumpeA())
      .AddPump(New PumpeA())
      .StartAllPumps()
   End With
   Console.ReadLine()
End Sub
```

Zunächst wird ein Objekt des Typs *ControlPumps* erzeugt. Diesem wird durch Aufruf der *AddPump* ein Pumpenobjekt übergeben, dessen Instanzierung implizit erfolgt. Zum Schluss werden alle Pumpen mit der Methode *StartAllPumps* aktiviert. (Anmerkung: Sie finden den Code zu diesem Beispiel auf der Buch-CD unter *...\Beispielcode\Kapitel_7\ArrayList_Loesung*.)

Wir haben anscheinend unser Ziel erreicht – anscheinend, denn eine genauere Analyse des Codes zeigt, dass wir nicht alle Forderungen erfüllen können, um von einer flexiblen, anpassungsfähigen Steuerkomponente zu sprechen. Es ist nun zwar möglich, beliebig viele Pumpen anzusteuern – dazu ist die Steuerklasse fraglos in der Lage. Was ist aber, wenn ein dritter Pumpentyp – bezeichnen wir ihn als *PumpeC* – installiert wird, dessen Startmethode *SwitchOnC* lautet? Wir stehen nahezu wieder am Anfang und müssen nach einer anderen Lösung suchen. Das führt uns zu dem eigentlichen Thema dieses Abschnitts, denn mit **Delegates** wird eine anforderungsgerechte Implementierung möglich.

7.2.3 Einfache Delegates

Wo liegt denn das ursächliche Problem, das zum Verwerfen der Lösung von oben geführt hat? Der entscheidende Punkt ist doch, dass die Startmethoden der verschiedenen Pumpentypen nicht gleichnamig sind. Unser Client hat Kenntnis über die eingebauten Methoden, und über diese Kenntnis muss das *ControlPumps*-Objekt auch verfügen – zumindest im Code von oben.

Um die geforderte Flexibilität zu erreichen, muss das *ControlPumps*-Objekt aber völlig im Unklaren darüber gelassen werden können, welche Typen überhaupt vertreten sind: Der Aufruf der Startmethoden muss Allgemeingültigkeit besitzen. Er muss sogar so allgemein gültig sein, dass die Klasse in jedem beliebigen Schwimmbad zur Steuerung der Pumpen eingesetzt werden kann.

Der Knackpunkt ist der Methodenaufruf. Er muss aus der Klasse *ControlPumps* herausgezogen und in den Verantwortungsbereich des Clients gelegt werden – schließlich ist es der Client der als Einziger weiß, welche Pumpen gestartet werden können bzw. gestartet werden sollen.

Da aber weiterhin der Aufruf der Startmethoden (*SwitchOnA*, *SwitchOnB*, ...) aus dem steuernden Objekt heraus erfolgen soll, muss man sich fragen, was dieses Objekt denn letzten Endes tatsächlich braucht, um eine Methode auszuführen. Die Antwort ist ganz einfach: Das Objekt vom Typ *ControlPumps* braucht die Adresse der aufzurufenden Methode.

Der Client muss also dem Steuerobjekt nur noch mitteilen, unter welcher Adresse eine Methode zu finden ist, die aufgerufen werden soll. Das klingt verdächtig nach der Zeigertechnik der Programmiersprache C/C++, genauer, nach Funktionszeigern. Und tatsächlich lehnt sich der von der .NET-Plattform bereitgestellte Lösungsansatz in Form von Delegates an dieser Technik an.

Fassen wir an dieser Stelle die bisherigen Überlegungen und die daraus resultierenden Konsequenzen in zwei Sätzen zusammen:

1. Der Client hat Kenntnis über die zur Verfügung stehenden Pumpen und kennt die Methoden, mit der jeder Pumpentyp gestartet werden kann.
2. Der Client übermittelt dem Objekt der steuernden Klasse einen Zeiger auf die aufzurufende Startmethode.

Zeigertechnik und .NET-Plattform – das passt eigentlich nicht zusammen. Die Zeigertechnik – so interessant sie auch sein mag – birgt einige Nachteile in sich: Sie ist schwierig zu lernen, sie ist sehr komplex, die Programme sind zu kompliziert. Ein falscher Einsatz führt nicht selten zu Speicherzugriffsfehlern und damit zum Absturz eines laufenden Programms. Nicht umsonst haben die Konstrukteure der .NET-Plattform (und auch die von Java) die Zeigertechnologie gemieden wie der Teufel das Weihwasser.

Dennoch gibt es Aufgabenstellungen, wie in unserem Fall, an denen kein Weg an Zeigern vorbeiführt. Zeiger gibt es tatsächlich auch unter der .NET-Plattform, allerdings in einer nicht sofort offensichtlichen Form. Wie Sie wissen, wird wirklich alles im .NET-Framework als ein Objekt angesehen – auch die Zeiger auf Methoden, nach denen unser Herz so sehr begehrt. Zeiger auf Methoden werden in ein Objekt verpackt und haben die Bezeichnung **Delegate**.

Ein **Delegate** ist ein Objekt und beinhaltet den Zeiger auf eine Objektmethode.

Ein Delegate kapselt einen Methodenaufruf einschließlich der Argumente und des Rückgabetyps. Schauen wir uns die Deklarationen eines Delegates an einem Beispiel an:

```
Public Delegate Sub MyDelegate(ByVal x As Int32)
```

Es wird eine Delegate mit dem Bezeichner *MyDelegate* deklariert sowie einer Parameterliste, die einen **Int32**-Wert beschreibt.

Ein Delegate kapselt einen Zeiger auf eine Methode, oder mit anderen Worten, ein Delegate steht für einen beliebigen Methodenaufruf. Ganz beliebig ist der Methodenaufruf jedoch nicht, denn jede Methode hat eine definierte Parameterliste, mit einer gewissen Anzahl von Parametern eines bestimmten Typs – natürlich auch dann, wenn die Methode überladen ist. Die Typen der Parameterliste einer Methode, auf die ein Delegate zeigt, müssen in der Parameterliste der Delegate-Deklaration angegeben werden.

Im Beispiel oben wird in der Parameterliste des Delegates *MyDelegate* ein **Int32**-Typ aufgeführt. Damit wäre ein Delegate dieses Typs in der Lage, jede x-beliebige Methode eines x-beliebigen Objekts aufzurufen – vorausgesetzt, diese Methode definiert eine Parameterliste, die genau ein **Int32**-Argument entgegennimmt.

Die Parameterliste einer Delegate-Deklaration entspricht der Parameterliste der Methode, deren Funktionszeiger ein Delegate repräsentiert.

Wollen Sie den Aufruf einer Methode kapseln, die als erstes Argument einen String und als zweites einen Boolean empfängt, müssten Sie wie folgt deklarieren:

```
Public Delegate Sub AnotherDelegate(ByVal s As String, ByVal b As
Boolean)
```

Delegates beschränken sich nicht nur auf Methoden ohne Rückgabewert. Verbirgt ein Delegate den Zeiger auf eine Methode mit Rückgabewert, könnte eine Deklaration beispielsweise wie folgt aussehen:

```
Public Delegate Function FuncWrapper(ByVal s As Int16) As Int32
```

Sie können eine Delegate-Deklaration wie eine Klassendefinition ansehen (wenn auch diese »Klassendefinition« ein wenig anders aussieht als die herkömmlicher

Klassen). Eine Klasse muss instanziiert werden, eine Delegate-Typ muss das ebenfalls. Sehen wir uns die Instanzierung und ihre Auswirkungen an einem Codefragment an:

```
Public Delegate Sub MyDelegate(ByVal x As Int32)

Module Module1
   Public Sub Main()
      Dim obj As New ClassA()
      Dim del As New MyDelegate(AddressOf obj.TestProc)
      'weitere Anweisungen
   End Sub
End Module

Public Class ClassA
   Public Sub TestProc(ByVal int As Int32)
      Console.WriteLine("In der Methode ClassA.TestProc")
   End Sub
End Class
```

Es ist der Typ *ClassA* mit der parametrisierten Methode *TestProc* definiert, die später durch einen Delegate-Typ ausgeführt werden soll, der im Allgemeinteil des Moduls deklariert ist. Der Delegate-Typ definiert eine Parameterliste mit einem **Int32**, was sich mit der Parameterliste der Methode *TestProc* der *ClassA* deckt: Typ und Anzahl der Parameter stimmen überein.

In der *Main*-Prozedur wird zuerst die Klasse *ClassA* instanziiert, im Anschluss daran der Delegate. Diese Anweisung müssen wir uns noch genauer ansehen:

```
Dim del As New MyDelegate(AddressOf obj.TestProc)
```

So wie alle anderen Objekte auch, muss eine Delegate-Objekt mit dem `New`-Operator erzeugt werden. Der Konstruktor aller Delegates erwartet als Argument die Übergabe der Adresse einer Methode. Da die einfache Angabe

```
obj.TestProc
```

syntaktisch für den Aufruf einer Methode steht, tritt ein neuer Operator ins Rampenlicht: `AddressOf`. Die Folge ist, dass mit dem Argument

```
AddressOf obj.TestProc
```

die Adresse, also ein Zeiger, auf die aufzurufende Methode des Objekts *obj* übergeben und nicht die Methode selbst sofort ausgeführt wird. Da durch ein Delegate nur ein Funktionszeiger gekapselt wird, dürfen auch keine Argumente übergeben werden.

Die Instanzierung eines Delegate-Typs führt noch nicht zu der sofortigen Ausführung der Methode, denn der Methodenaufruf muss explizit angestoßen werden. Der Code in unserem Beispiel ist noch nicht so weit entwickelt, dass er dazu in der Lage ist.

Die von einem Delegate beschriebene Methode wird erst durch den Aufruf der Methode `Invoke` auf das Delegate-Objekt ausgeführt. Dabei werden gleichzeitig die Argumente übergeben, die von der Parameterliste der eingeschlossenen Methode verlangt werden. `Invoke` ist die Standardmethode eines Delegates und muss deswegen nicht zwangsläufig angegeben werden – die einfache Angabe des Delegatebezeichners sowie der Parameterliste ist vollkommen ausreichend.

Ergänzen wir nun noch die *Main*-Prozedur unseres Beispiels um den Aufruf des Delegates:

```
Public Sub Main()
   Dim obj As New ClassA()
   Dim del As New MyDelegate(AddressOf obj.TestProc)
   del.Invoke(2)
   Console.ReadLine()
End Sub
```

An der Konsole wird daraufhin die in *TestProc* festgelegte Ausgabe erfolgen.

7.2.4 Delegates als flexible Lösung

Wenden wir uns wieder der Entwicklung der Software zu, mit der wir die Pumpen in einem Schwimmbad einschalten wollen. Mit den Erkenntnissen des vorhergehenden Abschnitts soll nun die Steuerung durch die Klasse so angepasst werden, dass sie universell einsetzbar ist.

Hier sei nun die endgültige Implementierung der Klasse *ControlPumps* und des Clientcodes in der *Main*-Prozedur. Die Klassen *PumpeA* und *PumpeB* haben sich nicht verändert, wir setzen sie als gegeben voraus.

```
'-----------------------------------------------------------------
'Codebeispiel: ...\Beispielcode\Kapitel_7\SimpleDelegate
'-----------------------------------------------------------------
Public Delegate Sub PumpDelegate()

Module Module1
   Public Sub Main()
      Dim obj As New ControlPumps()
      'die erste Pumpe erstellen
      Dim P1 As New PumpeA()
```

```
      'Delegate erzeugen und die Methode SwitchOnA übergeben
      Dim del As New PumpDelegate(AddressOf P1.SwitchOnA)
      'Delegate der Auflistung der Delegates hinzufügen
      obj.AddPump(del)
      'analog die drei weiteren Pumpen erzeugen und die
      'typspezischen Methoden einem impliziten Delegate übergeben
      Dim P2 As New PumpeB()
      obj.AddPump(New PumpDelegate(AddressOf P2.SwitchOnB))
      Dim P3 As New PumpeA()
      obj.AddPump(New PumpDelegate(AddressOf P3.SwitchOnA))
      Dim P4 As New PumpeA()
      obj.AddPump(New PumpDelegate(AddressOf P4.SwitchOnA))
      'die Pumpen starten
      obj.StartAllPumps()
      Console.ReadLine()
   End Sub
End Module

Public Class ControlPumps
   Private colPumps As New System.Collections.ArrayList()

   Public Sub AddPump(ByVal newPump As PumpDelegate)
      colPumps.Add(newPump)
   End Sub

   Public Sub StartAllPumps()
      Dim delObj As PumpDelegate
      For Each delObj In colPumps
         delObj.Invoke()
      Next
   End Sub
End Class
```

Die einschneidenste Änderung im Vergleich zu unserer ersten Version ist der Typ, der von der Auflistung verwaltet wird. Waren es anfangs die Referenzen auf die Pumpen, so handelt es sich nun um Objekte vom Typ *PumpDelegate*. Diese werden in der *Main*-Prozedur zu jeder Pumpe erzeugt und kapseln den Funktionszeiger auf die Methode, mit der die zu dem Delegate gehörige Pumpe eingeschaltet wird. Nachdem der objektspezifische Delegate erzeugt ist, kann er unter Aufruf der Methode *AddPump* des *ControlPump*-Objekts zu einem von der Auflistung verwalteten Mitglied gemacht werden.

Betrachten wir den Ausschnitt aus der *Main*-Prozedur, mit dem das Delegate-Objekt des Zeigers auf die Startmethode einer *PumpeA* der Auflistung hinzugefügt wird:

```
Dim P1 As New PumpeA()
obj.AddPump(New PumpDelegate(AddressOf P1.SwitchOnA))
```

Diese beiden Codezeilen sind eine kürzere Variante von

```
Dim P1 As New PumpeA()
Dim del As New PumpDelegate(AddressOf P1.SwitchOnA)
obj.AddPump(del)
```

Das **Delegate**-Objekt, hier als *del* bezeichnet, wird aber im weiteren Verlauf des Programms nicht mehr aufrufen – uns interessiert nur die nackte Existenz, deren Verweis wir der Auflistung mitteilen müssen. Daher können wir den Konstruktor der **Delegate**-Klasse aufgerufen, ohne den Rückgabewert in Form der Referenz einer entsprechend typisierten Objektvariablen zuzuweisen.

Der Aufruf der *AddPump*-Methode landet im Objekt der Klasse *ControlPumps* und wird im Parameter *newPump* vom Typ *PumpDelegate* entgegengenommen. Das funktioniert tadellos, weil ein Delegate bekannterweise ein Objekt ist, und eine Auflistung der Klasse **System.Collections.ArrayList** grundsätzlich jeden Objekttyp verwalten kann.

```
Public Sub AddPump(ByVal newPump As PumpDelegate)
   colPumps.Add(newPump)
End Sub
```

Nachdem alle Pumpenobjekte erzeugt und das Objektarray *colPumps* mit den Delegates auf die Startmethoden gefüllt ist, lassen sich alle Pumpen durch den Aufruf der Methode *StartAllPumps* aktivieren. Doch statt alle Startmethoden in einer `For Each...Next`-Schleife direkt zur Ausführung zu bewegen, geschieht dies nun indirekt durch die in den Delegates eingeschlossenen Adressen der spezifischen Methoden.

Das Ergebnis ist perfekt, wir haben das Ziel erreicht. Die Klasse ist so flexibel implementiert, dass sie nicht nur die Belange eines Schwimmbads abdeckt, sondern überall dort eingesetzt werden könnte, wo Pumpen der Reihe nach eingeschaltet werden müssen. Eigentlich ist diese Aussage falsch, denn wir können sie sogar auf jedwede beliebige Komponente ausdehnen, unter der Voraussetzung, dass in der Komponente eine parameterlose Methode aufgerufen werden soll. Die Verhaltensweise, die von der Methode beschrieben wird, spielt dabei überhaupt keine Rolle – alles dank der Delegates.

7.2.5 Multicast-Delegates

.NET bietet die Möglichkeit, mehrere Delegates zu einem einzigen zusammenzufassen. Dadurch entsteht ein Delegateverbund der auch als **Multicast-Delegate** bezeichnet wird. Der Vorteil ist, dass durch den Aufruf eines Delegates mehrere Delegate zur Ausführung gebracht werden können.

Sehen Sie sich dazu noch einmal unser Beispiel *SimpleDelegate* des vorgehenden Kapitels an. Es wird darin von vier Pumpen ausgegangen, deren Startmethoden der Reihe nach in je ein Delegate-Objekt eingebunden werden. In der steuernden Klasse bedarf es eines Objektarrays, um alle Delegates zu verwalten.

Der Programmcode ist wesentlich einfacher und übersichtlicher, wenn ein Multicast-Delegate die Aufgabe übernimmt. Dies soll das folgende Beispiel zeigen, dass unter denselben Vorgaben wie das Beispiel *SimpleDelegate* entwickelt worden ist. Der Code dazu lautet:

```
'-----------------------------------------------------------------
'Codebeispiel: ...\Beispielcode\Kapitel_7\MulticastDelegate
'-----------------------------------------------------------------
Public Delegate Sub PumpDelegate()

Module Module1
   Public Sub Main()
      Dim obj As New ControlPumps()
      Dim P1 As New PumpeA()
      Dim P2 As New PumpeB()
      Dim P3 As New PumpeA()
      Dim P4 As New PumpeA()
      Dim del(3) As PumpDelegate
      del(0) = New PumpDelegate(AddressOf P1.SwitchOnA)
      del(1) = New PumpDelegate(AddressOf P2.SwitchOnB)
      del(2) = New PumpDelegate(AddressOf P3.SwitchOnA)
      del(3) = New PumpDelegate(AddressOf P4.SwitchOnA)
      Dim arrDel As PumpDelegate = System.Delegate.Combine(del)
      'Delegate an Objekt der Klasse ControlPumps übergeben
      obj.AddPump(arrDel)
      'Pumpen einschalten
      obj.StartAllPumps()
      Console.ReadLine()
   End Sub
End Module

'Klasse, die die Steuerung der Pumpen übernimmt
Public Class ControlPumps
   Private arrPump As PumpDelegate
```

```
    Public Sub AddPump(ByVal pumps As PumpDelegate)
        arrPump = pumps
    End Sub
    Public Sub StartAllPumps()
        arrPump.Invoke()
    End Sub
End Class
```

Die Klassendefinitionen der Pumpen haben sich natürlich auch jetzt nicht geändert. Werfen wir zuerst einen Blick auf den Code in der *Main*-Prozedur.

Es fällt als Erstes auf, dass die den Pumpenobjekten zugeordneten Delegates nun zu Elementen eines Array werden. Das hätten wir natürlich auch schon im Code des Beispiels *SimpleDelegate* so machen können. Nun steckt aber eine ganz bestimmte Idee dahinter, die in der darauf folgenden Codezeile deutlich wird:

```
Dim arrDel As PumpDelegate = System.Delegate.Combine(del)
```

Die Klasse **System.Delegate** stellt die statische Methode `Combine` bereit, um einen einfachen **Singlecast-Delegate** zu einem Multicast-Delegate zu erheben. Die Methode `Combine` ist wie folgt überladen:

```
Public Overloads Shared Function Combine(Delegate()) As Delegate
Public Overloads Shared Function Combine(Delegate, Delegate) _
                                                    As Delegate
```

Sie können als Argument also entweder ein Array vom Typ **Delegate** übergeben oder Sie haben die Möglichkeit, zwei explizit genannte Delegates miteinander zu verknüpfen. Der Rückgabewert ist in beiden Fällen vom Typ **Delegate** und kann, wie wir es in unserem Beispiel auch vollzogen haben, einer Objektvariablen desselben Typs zugewiesen werden.

In unserem Programm erscheint die Variante, die ein Array entgegennimmt, die geeignetere zu sein, denn sie erspart uns im Vergleich zu der anderen etwas Codierung.

Wir haben nun eine Objektvariable Namens *arrDel*, die einen Multicast-Delegate referenziert, der seinerseits wieder vier Singlecast-Delegates repräsentiert. Dem Objekt der steuernden Klasse müssen wir jetzt nur noch die Referenz *arrDel* übergeben und können uns daher in der Klassendefinition von *ControlPumps* das aggregierte **ArrayList**-Objekt ersparen. Das hat zur Folge, dass die Methoden *AddPump* und *StartAllPumps* an die neue Ausgangssituation angepasst werden müssen. Insgesamt reduziert sich aber der Code und wird dadurch deutlich einfacher.

Führen Sie das Programm aus, wird an der Konsole dieselbe Ausgabe erscheinen wie im Beispiel des Abschnitts 7.2.4:

```
Pumpe A wird eingeschaltet.
Pumpe B wird eingeschaltet.
Pumpe A wird eingeschaltet.
Pumpe A wird eingeschaltet.
```

7.2.6 Allgemeine Anmerkungen zu Delegates

Die Wirkungsweise und die von den Delegates eingebrachte Funktionalität ist nicht einfach zu verstehen. Haben Sie schon mit den Vorgängerversionen von VB .NET gearbeitet, wissen Sie, dass es kein Pendant bis einschließlich zur Version 6 dazu gibt. Einem Anfänger in der Programmierung wird dieser Bereich der Programmierung wahrscheinlich am Anfang sehr suspekt erscheinen. Ich habe Sie aus diesem Grund sehr langsam in die Thematik eingeführt und anhand eines Beispiels versucht zu erläutern, welcher Nutzen aus Delegates gezogen werden kann und welche Vorteile er bietet.

Wir sind aber noch nicht am Ende der Fahnenstange angekommen, es werden noch ein paar Aspekte hinzukommen, die bisher noch nicht erörtert worden sind. Dennoch soll in dieser Stelle ein kleines Resümee gezogen werden, einerseits um das Verständnis zu vertiefen, andererseits um auf ein paar weitere Möglichkeiten hinzuweisen.

Ein Delegate gilt als typsicherer Funktionszeiger, weil er nur auf Methoden mit einer bestimmten Parameterliste und einem bestimmten Rückgabewert verweisen kann. Stimmt beim Aufruf des Delegates entweder die Parameterliste oder der Rückgabewert nicht mit denen der Methode überein, die über den Delegate ausgeführt worden soll, erhalten Sie beim Kompilieren bereits eine Fehlermeldung. Es spielt jedoch keine Rolle, ob es sich um eine Instanzmethode oder um eine statische Methode handelt – ein Delegate unterscheidet in diesem Punkt nicht.

Delegates werden als Funktionszeiger verstanden, die die Adresse einer Methode in einem Objekt kapseln. Das erstaunliche daran ist, auf welche Weise ein Delegate-Objekt erzeugt wird:

```
Public Delegate Sub MyDelegate()
...
Dim del As New MyDelegate(AddressOf abc.xyz)
```

Hinter dem `New`-Operator wird per Sprachdefinition der Typ, also die Klasse, angegeben. An dieser Aussage ändern natürlich auch Delegates nichts. Allerdings haben wir an keiner Stelle im Code eine Klasse *MyDelegate* definiert und in der .NET-Klassenbibliothek werden wir natürlich auch nicht fündig. Wie ist das zu erklären?

Das Problem wird durch zwei Methoden verursacht: zum einen durch den Konstruktor und zum anderen durch die Aufrufmethode `Invoke`. Betrachten wir zuerst den Konstruktor.

Ein konkretes **Delegate**-Objekt ist erst dann als ein solches zu bezeichnen, wenn es seine Aufgabe erfüllen kann, einen Methodenzeiger zu kapseln. Daher wird auch nur ein Konstruktor unterstützt, der die erforderliche Adresse entgegennimmt. Im Beispiel oben ist es die Adresse der Methode *xyz* des Objekts *abc*. Wie der Konstruktor mit dieser Übergabe umgeht und sie intern verarbeitet, spielt für uns keine Rolle.

```
Public Sub New(ByVal pointer As Adresse)
   AdresseDerMethode = pointer
End Sub
```

Der Typ *Adresse* ist hier nur fiktiv angenommen. Jetzt kommt der alles entscheidende Punkt: Die Methode `Invoke`, die als Konsequenz des Benutzeraufrufs die Methode ausführt, welche durch die vom **Delegate**-Objekt gekapselte Speicheradresse beschrieben wird.

```
Public Sub Invoke(...)
   'Aufruf der Methode an der Speicheradresse AdresseDerMethode
End Sub
```

Eine Klasse muss vor der Instanzierung definiert sein. Rein theoretisch könnten Sie eine Methode wie `Invoke` natürlich selbst implementieren, wenn dagegen nicht ein Argument sprechen würde, das uns die Codierung unmöglich macht: .NET unterstützt die fundamentale Zeigertechnik nicht, die dazu notwendig wäre. Aus diesem Dilemma hilft uns die Deklaration eines Delegates, der dynamisch eine Klasse bereitstellt, die unseren Anforderungen genügt.

Wird die Deklaration kompiliert, wird tatsächlich eine neue Klasse generiert, die auf einer der beiden folgenden Klassen des .NET-Frameworks basiert: entweder auf **System.Delegate** oder auf **System.MulticastDelegate**.

Jedes **Delegate**-Objekt steht für eine Liste von Methodenaufrufen, die durchlaufen wird, sobald der Delegate ausgeführt wird. Im Falle eines Singlecast-Delegates enthält diese Liste nur ein Element, bei einem Multicast-Delegate können es

mehrere sein. Auf diese Aufrufliste können Sie mit der Methode `GetInvocationList` der Klasse **Delegate** bzw. **MulticastDelegate** zugreifen, der Rückgabewert ist ein Array vom Typ `Delegate`:

```
Overridable Public Function GetInvocationList() As Delegate()
```

Um eine Methode zu der Aufrufliste hinzuzufügen oder zu entfernen, definiert die **Delegate**-Klasse die beiden statischen Methoden `Combine` und `Remove`. Wir hatten in unserem Beispiel oben einen Multicast-Delegate erzeugt, indem wir vier Singlecast-Delegates in ein Array zusammengefasst und als Argument der `Combine`-Methode übergeben haben. Die zweite Version dieser überladenen Methode wollen wir uns zusammen mit der `Remove`-Methode anschauen:

```
Public Overloads Shared Function Combine(Delegate, Delegate) _
                                                  As Delegate
Public Shared Function Remove(Delegate, Delegate) As Delegate
```

Beide Parameterlisten sind identisch und erwarten sowohl im ersten als auch im zweiten Argument die Referenz auf ein **Delegate**-Objekt. Im ersten Parameter wird dabei die Referenz auf den Delegate erwartet, zu dessen Aufrufliste ein weiterer Delegate hinzugefügt bzw. im Fall der `Remove`-Methode entfernt werden soll. Der zweite Parameter beschreibt den hinzuzufügenden bzw. zu entfernenden Delegate. Dazu ein kleines Beispiel:

```
1:   Dim del_1 As New PumpDelegate(AddressOf P1.SwitchOnA)
2:   Dim del_2 As New PumpDelegate(AddressOf P2.SwitchOnB)
3:   del_2 = System.Delegate.Combine(del_2, del_1)
4:   ...
5:   del_2 = System.Delegate.Remove(del_2, del_1)
```

In der dritten Codezeile wird der Delegate zu einem Multicast-Delegate erhoben, in der fünften wird diese Zuordnung wieder aufgehoben. Es ist möglich, im zweiten Argument wieder einen Multicast-Delegate anzugeben, letztendlich verkleinert sich dadurch allerdings nicht der Programmcode. Bei einer Kombination mehrerer Delegates ist daher die Variante mit der Übergabe eines Arrays vorzuziehen.

Interessiert der Name der von einem Delegate gekapselten Methode, lässt sich das durch die schreibgeschützte Eigenschaft `Method` nebst vielen anderen Informationen in Erfahrung bringen. Der Aufruf von `Method` liefert als Rückgabewert die Referenz auf ein Objekt vom Typ **System.Reflection.MethodInfo**, das die unterschiedlichsten Informationen zu einer Methode bereitstellt, beispielsweise über die Instanzeigenschaft `Name` den Namen der von einem Delegate eingeschlossenen Methode.

```
Console.WriteLine(del_1.Method().Name)
```

Die ebenfalls schreibgeschützte Eigenschaft `Target` der **Delegate**-Klasse liefert einen **Object-Typ** an den Aufrufer. Diese Referenz beschreibt das eingeschlossene Objekt. Die Information könnte beispielsweise in der `CType`-Funktion zur expliziten Konvertierung benutzt werden, solange sich die zurückgelieferte Referenz in einer Vererbungsbeziehung mit dem im zweiten Argument genannten Typ befindet:

```
Dim newPump As PumpeA = CType(del_1.Target(), PumpeA)
```

Natürlich stehen nach einer solchen Konvertierung sämtliche Elementfunktionen zur Verfügung, sowohl die der Klasse **System.Object** als auch die der Klasse, in die konvertiert wurde.

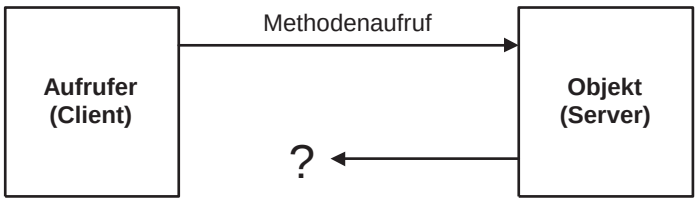
7.2.7 Delegates zur synchronen und asynchronen Benachrichtigung

Normalerweise ruft ein Client die Methode einer Klasse auf, um eine bestimmte Funktionalität zu nutzen. Der Client hat über die Objektvariable Kenntnis von der Existenz des Dienstanbieters, er kann Methoden ausführen, Eigenschaften setzen und auswerten – solange es die Definition der Entitäten zulässt. Das Objekt, das vom Client gehalten wird, weiß jedoch nichts von seinem Erzeuger, es kann nur auf die Aufrufe reagieren. Sie müssen sich das wie eine Einbahnstrasse vorstellen, bei der die vorgeschriebene Fahrtrichtung von A nach B führt, aber nicht zurück (wir wollen natürlich nicht diskutieren, dass sie sich verkehrswidrig verhalten könnten, uns interessiert nur das Reglement).

Was ist aber nun, wenn eine aufgerufene Klasse dem Aufrufer eine Informationen zukommen lassen soll? Auf ein solches Problem trifft man in der Programmierung relativ häufig. Denken Sie nur an unser Beispiel mit den Pumpen im Schwimmbad zurück. Wir haben diese der Reihe nach eingeschaltet und gehen einfach davon aus, dass sie danach auch tatsächlich laufen. Was aber ist, wenn eine Pumpe ihren Dienst verweigert, aus welchen Gründen auch immer? Eine gute Lösung würde zumindest vorsehen, den Client von der erfolgreichen Einschaltung zu unterrichten. Dazu muss das Pumpenobjekt den aufrufenden Client benachrichtigen können – man sagt auch, es muss ihn zurückrufen.

Dies ist ein ganz wesentlicher Aspekt in der Programmierung und wird mit **Callbacks**, auch als **Rückrufmethoden** bezeichnet, gelöst.

"Normalfall": Der Client kennt das Objekt (Server), das Objekt andererseits jedoch nicht seinen Aufrufer (Client)



Callback: Der Aufrufer übergibt dem Objekt Informationen über sich, die das Objekt zu einem Rückruf nutzt

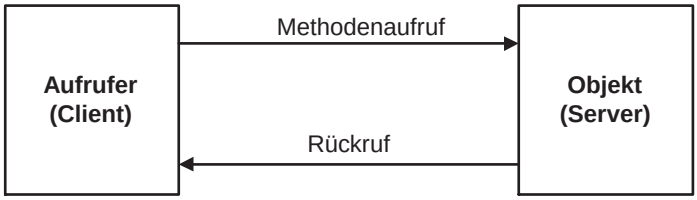


Abbildung 7.2 Callback eines aufgerufenen Servers

Callbacks spielen nicht nur in so einfachen Szenarien wie dem Beispiel unserer Pumpen eine Rolle, wo eine eingeschaltete Pumpe den Client vom Erfolg der Aktion unterrichten soll. Viel wichtiger sind sie bei der **asynchronen Bearbeitung** von Vorgängen. Nehmen wir zur Verdeutlichung das folgende Codefragment:

```
Public Class Client
   Public Sub TestProc()
      Dim obj As New Server()
      obj.LongTimeProc()
      'weitere Anweisungen
   End Sub
End Class
Public Class Server
   Public Sub LongTimeProc()
      'diese Ausführung kann "länger" dauern
   End Sub
End Class
```

Der Client erzeugt ein Objekt vom Typ *Server* und ruft darauf die Methode *LongTimeProc* auf, die, wie schon der Name verspricht, eine längere Zeitspanne zu ihrer Abarbeitung benötigt. Während dieser Zeit verharrt der Client im Stillstand, denn er muss solange warten, bis er nach dem Verlassen der Servermethode durch `End Sub` wieder die Kontrolle über die Laufzeit erlangt. Danach können die

Anweisungen, die dem *LongTimeProc*-Aufruf folgen, ausgeführt werden. Benötigt der Server 10 Minuten, dann wartet der Client auch 10 Minuten – eine grauenhafte Vorstellung, wenn man an den geplagten Anwender denkt, der seine Arbeitswut nicht mehr befriedigen kann und statt dessen in dieser Zeit nicht nur den Kaffee aufsetzen muss, sondern auch noch Zeit hat, ihn zu trinken. Diese Art der Bearbeitung von Vorgängen wird als **synchron** bezeichnet – die Methodenaufrufe werden der Reihenfolge nach ausgeführt.

Eine optimale Lösung wäre die gleichzeitige Bearbeitung der Vorgänge sowohl im Client als auch im Server. Der Client ruft eine bestimmte Methode im Server auf und gibt dabei eine Rückrufmethode bekannt, über die der Server den Client über die Beendigung seiner Arbeit informiert. Beide Komponenten, sowohl der Client als auch der Server, können dann parallel die ihnen zugewiesene Arbeit verrichten. Der Client kann sich also, während der Server mit sich selbst beschäftigt ist, anderen Aufgaben widmen, ohne auf die Beendigung des langwierigen Clientaufrufs warten zu müssen. Dies wird als **asynchrone Bearbeitung** bezeichnet.

Bei der synchronen Bearbeitung von Vorgängen werden die Anweisungen der Reihe nach ausgeführt, bei der asynchronen verlaufen zwei oder mehr Arbeitsvorgänge parallel.

Rückrufmethoden sind nicht unabdingbare Voraussetzung der asynchronen Bearbeitung, aber typisch. Meistens wird allerdings der Arbeitsablauf im Client durch die Information, dass der Server seine Ausführung beendet hat, maßgeblich bestimmt. Bei einer asynchronen Bearbeitung sind allerdings noch Multithreading-Aspekte zu berücksichtigen: Jeder Vorgang wird in einem eigenen Thread ausgeführt, bei einer Einprozessormaschine quasi-parallel, bei einer Mehrprozessormaschine möglicherweise tatsächlich gleichzeitig. Threads sind aber nicht Thema dieses Kapitels und werden erst in Kapitel 9 behandelt.

Rückrufe am Beispiel der Pumpenschaltung

Wenden wir uns wieder unserem Beispiel zu, anhand dessen wir mit den Delegates zu arbeiten gelernt haben. Ausgehend von dem Projekt *SimpleDelegate* entwickeln wir eine Anwendung, die es einer Pumpe ermöglicht, den Client über das erfolgreiche Anlaufen zu benachrichtigen.

Wir müssen einen Weg finden, um über die mittelnde Steuerklasse dem Pumpenobjekt einen Delegate bekannt zu geben, der nach dem Einschalten derselben zur Benachrichtigung des Benutzers dient. Zunächst sei der gesamte funktionsfähige Programmcode wiedergegeben, mit den Implementierungsdetails werden wir uns anschließend beschäftigen.

```
'---------------------------------------------------------------
'Codebeispiel: ...\Beispielcode\Kapitel_7\PumpCallback
'---------------------------------------------------------------
Public Delegate Sub PumpDelegate(ByVal info As InfoDelegate)
Public Delegate Sub InfoDelegate()

Public Class Client
   Public Shared Sub Main()
      Dim obj As New ControlPumps()
      Dim P1 As New PumpeA()
      Dim delPump As New PumpDelegate(AddressOf P1.SwitchOnA)
      Dim delInfo As New InfoDelegate(AddressOf PumpInfo)
      obj.AddPump(delPump)
      obj.StartAllPumps(delInfo)
      Console.ReadLine()
   End Sub

   Public Shared Sub PumpInfo()
      Console.WriteLine("Pumpe ist angelaufen.")
   End Sub
End Class

Public Class ControlPumps
   Private colPumps As New System.Collections.ArrayList()

   Public Sub AddPump(ByVal newPump As PumpDelegate)
      colPumps.Add(newPump)
   End Sub

   Public Sub StartAllPumps(ByVal callback As InfoDelegate)
      Dim delObj As PumpDelegate
      For Each delObj In colPumps
         delObj.Invoke(callback)
      Next
   End Sub
End Class

Public Class PumpeA
   Public Sub SwitchOnA(ByVal callClient As InfoDelegate)
      Console.WriteLine("Pumpe A wird eingeschaltet.")
      callClient.Invoke()
   End Sub
End Class
```

Der Einfachheit halber beschränkt sich der Code auf den Einsatz einer Pumpe vom Typ *PumpeA*. Um eine klarere Abgrenzung zu haben, wird auf die Verwendung eines Standardmoduls verzichtet und die Startmethode *Main* in einer Klasse

definiert. In den Projekteigenschaften muss daher auch das Startobjekt entsprechend eingestellt werden.

Den Kern der Anwendung bilden zwei Delegates. Der erste (*PumpDelegate*) dient weiterhin dazu, der steuernden Klasse einen Zeiger auf die Methode in der Pumpenklasse zu liefern, aus der heraus die Pumpe angeworfen wird. Der zweite Delegate (*InfoDelegate*) kapselt einen Zeiger auf die statische Methode *PumpInfo*:

```
Public Delegate Sub PumpDelegate(ByVal info As InfoDelegate)
Public Delegate Sub InfoDelegate()
```

Delegate *PumpInfo* verbirgt somit einen Funktionszeiger auf eine Methode, die ihrerseits wieder einen Delegate als Argument erwartet – es handelt sich hierbei also um einen verschachtelten Delegate.

Nach der Instanzierung mit

```
Dim delPump As New PumpDelegate(AddressOf P1.SwitchOnA)
Dim delInfo As New InfoDelegate(AddressOf PumpInfo)
```

verweist *delPump* auf die Adresse der Methode *SwitchOnA* des Objekts *PumpeA* und *delInfo* auf die Adresse der statischen Methode *PumpInfo* im Client, die später das Pumpenobjekt aufrufen soll. Mit

```
obj.AddPump(delPump)
```

wird die Pumpe der Auflistung *colPumps* im Objekt der Klasse *ControlPumps* hinzugefügt. Nach der Registrierung kann die Pumpe aktiviert werden. Der Client ruft dazu die Methode *StartAllPumps* auf und übergibt einen Delegate vom Typ *InfoDelegate*, mit anderen Worten, einen Zeiger auf die ihm eigene Methode *PumpInfo*:

```
obj.StartAllPumps(delInfo)
```

Dieser Delegate wird zunächst von *StartAllPumps* im Parameter *callback* entgegengenommen und beim Aufruf des Delegates, der im Pumpenobjekt das Einschalten bewirkt, weitergeleitet:

```
delObj.Invoke(callback)
```

Die mit

```
Public Sub SwitchOnA(ByVal callClient As InfoDelegate)
   Console.WriteLine("Pumpe A wird eingeschaltet.")
   callClient.Invoke()
End Sub
```

neu definierte Methode ist nun im Besitz eines Zeigers auf die parameterlose Methode *PumpInfo* und kann nun ihrerseits den Benutzer von der eigenen Aktivierung in Kenntnis setzen. Damit ist der Kreislauf geschlossen.

An der Konsole wird nach dem Start der Anwendung die korrekte Ausgabe erscheinen:

```
Pumpe A wird eingeschaltet.
Pumpe ist angelaufen.
```

7.2.8 Delegates und Events

Ruft ein Objekt die Methode eines anderes Objekt auf, hat das aufgerufene Objekt keine Kenntnis von der Existenz des Aufrufers. Normalerweise interessiert das auch nicht besonders. Das aufgerufene Objekt verrichtet brav seine Arbeit, der Aufrufer zieht seinen Nutzen daraus und die Geschichte ist beendet.

Ein Callback geht einen Schritt weiter. Der Aufrufer teilt dem Objekt seine Existenz in Form einer Methodenadresse mit, das aufgerufene Objekt kann, wenn es die Situation erfordert, mit diesen Informationen zumindest teilweise die Steuerung des Aufrufers übernehmen – für einen Moment sind die Positionen vertauscht. Wir haben die Anforderungen an einen Rückruf mit Delegates gelöst, aber vielleicht erinnern Sie sich noch an die Ereignisse, über die Sie in Kapitel 5 eine erste Einführung erfahren haben. Ein Ereignis macht auch nichts anderes, als unter bestimmten Umständen eine Methode im Client aufzurufen. Daraus lässt sich sofort die Artverwandtschaft zu einem Delegate schlussfolgern.

Schauen Sie sich zunächst das folgende Beispiel an. Dieser Code beinhaltet keine Besonderheiten und sollte aus den Aussagen des Kapitels 5 heraus verständlich sein. Es wird ein Ereignis in der Klasse *ClassA* definiert, das von einem Objekt des Typs der Klasse *ClassB* in einem Ereignishandler empfangen wird.

```
Module Module1
    Sub Main()
      Dim newClassB As New ClassB()
      newClassB.MyProc()
      Console.ReadLine()
    End Sub
End Module

Public Class ClassA
   'Deklaration des klassebspezifischen Events
   Public Event OnHallo()

   Public Sub TestProc()
```

```
      'Auslösen des Ereignisses im Client
      RaiseEvent OnHallo()
   End Sub
End Class

Public Class ClassB
   'Deklaration mit WithEvents bewirkt die Berücksichtigung
   'der vom Objekt ausgelösten Ereignisse
   Dim WithEvents obj As ClassA

   Public Sub MyProc()
      'Objekt initialisieren
      obj = New ClassA()
      obj.TestProc()
   End Sub

   'Ereignisprozedur des Objekts obj
   Private Sub obj_OnHallo() Handles obj.OnHallo
      Console.WriteLine("Hallo hier bin ich.")
   End Sub
End Class
```

Analysieren Sie diesen Code, kommen Sie sicherlich zu der Einsicht, dass die Benachrichtigung des Clients an den Ereignishandler fest gebunden ist – eine Bindung, die sich zur Kompilierzeit durch das **Handles**-Statement fixiert. Das ist wenig flexibel, denn es könnte durchaus wünschenswert sein, einen Ereignishandler zur Laufzeit dynamisch und in Abhängigkeit gewisser Begleitumstände an eine andere Routine zu binden.

Der Zusammenhang zwischen einem Delegate und einem Ereignis ist offensichtlich. So wie bei einem Rückruf mit einem Delegate, hat ein Ereignissender Informationen über seinen Ereignisempfänger, kennt also das Ziel seiner Benachrichtigung – der Sender bezieht diese Informationen offensichtlich »irgendwoher«. Insbesondere erinnert ein Event sehr an eine Rückrufmethode, einen Callback: Ein Client hält das Objekt einer Klasse, und das Objekt ist in der Lage, eine Nachricht an den Client zu senden. Das ist doch genau die Funktionsweise, die wir schon vorher in diesem Abschnitt beschrieben haben.

Tatsächlich sind diese Gemeinsamkeiten nicht zufällig, denn ein Event basiert auf einem Delegate. Traditionell wird zur Deklaration eines Ereignisses das Schlüsselwort **Event** benutzt, beispielsweise

```
Public Event OnHallo()
```

Gleichwertig könnte man auch die folgende syntaktische Variante wählen. Dabei wird zuerst ein Delegate deklariert, der später einem Event zugeordnet wird:

```
Public Delegate Sub OnHalloHandler()
...
Public Event OnHallo As OnHalloHandler
```

Zuerst wird hinter dem Schlüsselwort `Event` der Bezeichner des Ereignisses festgelegt und danach der Typ des Delegates. Ausschlaggebend ist auch hier wieder die Parameterliste, die es durchaus ermöglicht, einem Delegate mehreren Events zuzuordnen. Die traditionelle Syntax macht nichts anderes, nur dass sich alles im Hintergrund abspielt, ohne Einsichtnahme und daher verborgen für den Entwickler.

> Die Parameterliste eines Ereignisses darf keine optionale Parameter enthalten. Folglich ist auch die Angabe eines **ParamArrays** nicht erlaubt.

Im Programmcode des Ereignisauslösers wird weiterhin mit `RaiseEvent` bekannt gegeben, wann der Ereignisempfänger eine Benachrichtigung erfahren soll.

Die Deklaration eines Events

Wir wollen im Folgenden ein wenig am Beispiel der Pumpen basteln und unsere Erkenntnisse dort einfließen lassen. Dabei werden wir den Code jedoch deutlich vereinfachen, um das wirklich Wichtige herauszukristallisieren. Wir verzichten daher auch auf die Klasse *ControlPumps*, und die Startmethoden *SwitchOnA* bzw. *SwitchOnB* werden direkt aus dem Benutzer heraus aufgerufen.

Definieren wir zuerst eine Pumpenklasse, die den Benutzer von der Inbetriebnahme durch ein Ereignis informiert.

```
Public Delegate Sub OnStartHandler()

Public Class PumpeA
   Public Event OnStart As OnStartHandler

   Public Sub SwitchOnA()
      Console.WriteLine("Pumpe A wird eingeschaltet.")
      RaiseEvent OnStart()
   End Sub
End Class
```

Diese Klasse wird nun von einem Benutzercode getestet:

```
Module Module1
   Dim WithEvents obj As PumpeA

    Sub Main()
      obj = New PumpeA()
```

```
      obj.SwitchOnA()
      Console.ReadLine()
    End Sub

   Public Sub obj_OnStart() Handles obj.OnStart
      Console.WriteLine("Pumpe A ist eingeschaltet.")
   End Sub
End Module
```

Das Ergebnis ist erwartungsgemäß, es werden die beiden Informationen an der Console ausgegeben. Beachten Sie bitte, dass durch das `Handles`-Statement immer noch eine feste Bindung zwischen dem ausgelöstem Event und der aufgerufenen Methode vorliegt.

Einen Event mit einem Ereignishandler verbinden

Im Benutzer wurde bisher immer mit der `Handles`-Klausel ein Ereignis an eine Routine gebunden. Diese Bindung ist statisch, weil sie zur Laufzeit nicht mehr verändert werden kann – ihr mangelt es an Flexibilität, die in manchen Situationen durchaus nützlich sein kann.

Es gibt noch einen weiteren Weg, der uns unter VB .NET die Anbindung eines Ereignisses an einen Handler zur Laufzeit ermöglicht. Ermöglicht wird dies durch die Anweisung `AddHandler`. Der Code, der sich im Benutzer durch den Einsatz von `AddHandler` ändert, zeigt das folgende Codefragment:

```
Module Module1
   Dim obj As New PumpeA

   Sub Main()
      AddHandler obj.OnStart, AddressOf PumpSwitchedOn
      obj.SwitchOnA()
      Console.ReadLine()
   End Sub

   Public Sub PumpSwitchedOn()
      Console.WriteLine("Pumpe A ist eingeschaltet.")
   End Sub
End Module
```

Es sind mehrere Positionen, die sich von unserem ersten Benutzer unterscheiden. Am Auffälligsten ist die Erweiterung der *Main*-Prozedur durch

```
AddHandler obj.OnStart, AddressOf PumpSwitchedOn
```

`AddHandler` bindet das Ereignis eines Objekt an eine bestimmte Prozedur. Im ersten Argument des Statements wird dabei das Objekt und ein Ereignis desselben aufgeführt, im zweiten die Adresse der aufzurufenden Methode, wenn das im ers-

ten Argument bekannt gegebene Ereignis im Objekt ausgelöst wird. Die allgemeine Syntax lautet demnach:

```
AddHandler <Objekt>.<Ereignis>, AddressOf <Objekt>.<Methodenname>
```

Sehen Sie sich den Code oben noch einmal an. Interessant ist die Tatsache, dass es nun keine **Handles**-Klausel mehr gibt, die das Ereignis statisch an die Methode bindet, sondern dass die Bindung nun im Programmcode erfolgt und daher auch vom Anwender gesteuert werden kann. Dies wird später durch die Erweiterung im Clientcode noch gezeigt.

Eine weitere Änderung fällt uns auf: Es kommt im Programmcode kein **WithEvents**-Statement mehr vor.

> Um ein Ereignis statisch mit der **Handles**-Klausel an eine Prozedur zu binden, ist die Deklaration einer Objektvariablen mit dem **WithEvents**-Statement Voraussetzung, bei der dynamischen Bindung mit **AddHandler** kann darauf verzichtet werden.

Ähnlich wie wir ein Ereignis mit einer Methode verknüpft haben, kann die Bindung auch wieder gelöst werden. Dazu dient der Befehl **RemoveHandler**, dessen Syntax identisch der von **AddHandler** ist:

```
RemoveHandler <Objekt>.<Ereignis>, AddressOf <Objekt>.<Methodenname>
```

Wenden wir uns nun wieder unserem Beispiel zu und verbinden das Ereignis zur Laufzeit dynamisch entweder mit einer Routine, die eine deutschsprachige Ausgabe an der Konsole anzeigt, oder mit einer englischsprachigen Version.

```
'-------------------------------------------------------------
'Codebeispiel:...\Beispielcode\Kapitel_7\AddHandler
'-------------------------------------------------------------
Option Compare Text
Imports System.Console

Module Module1
   Dim obj As New PumpeA()

   Sub Main()
      WriteLine("Wählen Sie die Sprachversion:")
      WriteLine("E - englisch")
      WriteLine("D - deutsch")
      WriteLine("----------------------")
      'Anwendereingabe der Sprachversion
```

```
      'E = Englisch
      'D = Deutsch
      Write("Ihre Wahl = ")
      Dim str As String = ReadLine()
      'den Handler entsprechend der Wahl binden
      If str = "E" Then
         AddHandler obj.OnStart, AddressOf PumpSwitchedOnE
      ElseIf str = "D" Then
         AddHandler obj.OnStart, AddressOf PumpSwitchedOnD
      Else
         Exit Sub
      End If
      'Pumpen starten
      obj.SwitchOnA()
      ReadLine()
   End Sub

   'deutschsprachige Ausgabezeichenfolge
   Public Sub PumpSwitchedOnD()
      WriteLine("Pumpe A ist eingeschaltet.")
   End Sub

   'englischsprachige Ausgabezeichenfolge
   Public Sub PumpSwitchedOnE()
      WriteLine("Pumpe A is activated.")
   End Sub
End Module

Public Class PumpeA
   Public Delegate Sub OnStartHandler()
   Public Event OnStart As OnStartHandler

   Public Sub SwitchOnA()
      Console.WriteLine("Pumpe A wird eingeschaltet.")
      RaiseEvent OnStart()
   End Sub
End Class
```

Nach der Wahl der englischsprachigen Variante wird an der Konsole

```
Pumpe A wird eingeschaltet.Pumpe A is activated.
```

ausgegeben. Dass nun bei der Wahl der englischsprachigen Variante gleichzeitig eine deutsch- und englischsprachige Ausgabe an der Konsole erscheint, nehmen wir mit einem großzügigen und wohl wollendem Schmunzeln zur Kenntnis – erinnert es uns doch an so manche Meldung des Windows-Betriebssystems oder seiner Microsoft-Anwendungskumpane ...

7.2.9 Der Sonderfall eines Events in einer Struktur

Eine Struktur ist bekanntermaßen mit Fähigkeiten ausgestattet, die denen einer Klasse ähnlich sind, ohne dabei allzu viel Overhead zu verschwenden. Strukturen können Konstruktoren definieren, Methoden, Eigenschaften und Ereignisse.

Der Zugriff auf die Strukturmitglieder erfolgt in derselben Weise wie bei einer Klasse – nur das Auffangen der von einer Struktur ausgelösten Ereignisse unterliegt strengeren Richtlinien: es darf nicht mit `WithEvents`, sondern nur mit `AddHandler` einfangen werden.

Im folgenden Codefragment wird in der Struktur *EventTest* das Ereignis *SayHallo* definiert, das beim Aufruf der Methode *DoEvent* ausgelöst wird:

```
Public Structure EventTest
   Public Event SayHallo()

   Public Sub DoEvent()
      RaiseEvent SayHallo()
   End Sub
End Structure
```

Wäre *EventTest* eine Klasse, könnten wir unter anderem mit

```
Dim WithEvents obj As EventTest
```

auf die ausgelösten Ereignisse reagieren. Weil *EventTest* aber als Struktur vorliegt, erzeugt diese Art der Deklaration einen Fehler und nötigt uns, den Weg über `AddHandler` zu beschreiten, beispielsweise

```
Sub Main()
   Dim obj As EventTest
   AddHandler obj.SayHallo, AddressOf MyEventProc
   obj.DoEvent()
   Console.ReadLine()
End Sub

Public Sub MyEventProc()
   Dim str As String
   str = "Ich wurde von einem EventTest-Objekt ausgelöst"
   Console.WriteLine(str)
End Sub
```

7.2.10 EventArgs und EventHandler

Dieses Buch versucht, Ihnen die Grundlagensyntax von Visual Basic .NET zu vermitteln. Das ist auch der Grund, weshalb wir – abgesehen von einem kleinen Beispiel im Kapitel 6 – die Belange von grafischen Benutzeroberflächen bisher völlig

außer Acht gelassen haben. Thematisch bietet es sich an dieser Stelle an, einen kurzen Blick auf die Programmierung grafischer Oberflächen zu werfen, die sich syntaktisch nicht von der Programmierung an der Eingabekonsole unterscheidet.

Neben der Visualisierung gibt es aber ein weiteres, insbesondere aus Sicht eines Entwicklers wesentliches Unterscheidungsmerkmal hinsichtlich des Programmablaufs: Während eine Konsolenanwendung meist auf den Aufruf von Methoden reagiert, sind es Ereignisse, die den Ablauf eines Windowsprogramms beeinflussen.

Die Ereignisse visualisierter Komponenten können auf unterschiedliche Art und Weise ausgelöst werden: Klickt ein Anwender zum Beispiel auf eine Schaltfläche, wird ein **Click**-Event ausgelöst; wird die Maus über eine Komponente bewegt, hat das die Auslösung des **MouseMove**-Ereignisses der Komponente zur Folge. Ereignisse werden ausgelöst, sobald ein Anwender eine Taste der Tastatur betätigt, wenn eine Komponente sich selbst neu zeichnet, wenn eine Komponente den Fokus erhält oder ihn wieder verliert usw. Die Liste ist sehr, sehr lang und abhängig vom Typ der Komponente.

In den Komponenten einer grafischen Benutzeroberfläche sind alle infrage kommenden Events bereits vordefiniert. Ob der Client das Ereignis programmiert oder nicht, bleibt ihm allerdings selbst überlassen. Weil die Events eine zentrale Stellung bei Windows-Anwendungen einnehmen, wird auch von der »ereignisgesteuerten Programmierung« gesprochen.

Wir wollen uns in diesem Abschnitt die Definition dieser Ereignisse ansehen. Denn um diese zu verstehen, gibt es keine bessere Ausgangsposition als die, in der wir uns in diesem Moment befinden. Natürlich können wir uns nicht mit allen möglichen Ereignissen grafischer Komponenten auseinander setzen. Erstens gibt es zu viele davon, und zweitens reichen auch schon zwei repräsentative Beispiele, um die Systematik zu verstehen.

Dazu betrachten wir nur die beiden schon oben erwähnten Ereignisse **Click** und **MouseMove**:

```
Private Sub Form1_Click(ByVal sender As Object, _
   ByVal e As System.EventArgs) Handles MyBase.Click

Private Sub Form1_MouseMove(ByVal sender As Object, _
   ByVal e As System.Windows.Forms.MouseEventArgs) _
   Handles MyBase.MouseMove
```

Beide Ereignisdefinitionen werden von einem Objekt namens *Form1* ausgelöst, was Sie an der `Handles`-Klausel erkennen können: Das **Click**-Ereignis wird ausgelöst, wenn der Anwender mit der Maus auf die Oberfläche eines Fensters klickt,

das **MouseMove**-Ereignis, wenn er den Mauszeiger über die Form bewegt. Ich möchte Ihre Aufmerksamkeit nun auf die den beiden Ereignissen eigene Parameterliste lenken.

Im ersten Parameter wird jeweils die Referenz auf ein Objekt vom Typ **Object** übergeben, im zweiten taucht in beiden Fällen zumindest der Ausdruck **EventArgs** auf. Wozu sind aber diese beiden Parameter notwendig, welchem Zweck dienen sie?

Ganz allgemein sei hier festgestellt, dass Eventhandler konventionsgemäß immer mit einer solchen Parameterliste ausgestattet werden sollten, weil ihr unter Umständen wesentliche Informationen zu entnehmen sind.

Widmen wir uns zunächst dem ersten Parameter – **sender**. Der Bezeichner sagt eigentlich schon aus, welcher Art die Referenz ist: Hier gibt sich der Ereignisauslöser selbst bekannt. Im ersten Moment mag die Frage auftauchen, warum dieser Parameter notwendig ist, da die `Handles`-Klausel doch schon eindeutig Auskunft darüber gibt. Aber erinnern Sie sich an die Ausführungen des letzten Abschnitts, man kann auf die `Handles`-Klausel verzichten und mit `AddHandler` die Prozedur an das Ereignis binden. Die Folge ist, dass mehrere Events dieselbe Routine aufrufen können oder diese sogar dynamisch zugeteilt wird.

```
AddHandler Form1.Click, AddressOf Form1_Click
AddHandler obj.MyOwnEvent, AddressOf Form1_Click
...
Private Sub Form1_Click(ByVal sender As Object, _
                        ByVal e As System.EventArgs)
```

Der ursprüngliche Event ist zu einer einfachen Prozedur mit dem Namen *Form1_Click* degradiert und kann, wenn Sie sich das Codefragment anschauen, sowohl bei der Auslösung des **Click**-Ereignisses des Objekts mit dem Namen *Form1* als auch bei der Auslösung des Ereignisses *MyOwnEvent* des über *obj* referenzierten Objekts aufgerufen werden. Sollte es notwendig sein, den Typ des Aufrufers von `Form1_Click` zu ermitteln, müssen Sie nur den ersten Parameter auswerten. Damit wäre die Existenzberechtigung dieses Parameters erklärt.

Der zweite Parameter dient dazu, ereignisspezifische Daten an den Eventhandler weiterzureichen. Bei einem **Click**-Ereignis spielt so etwas keine Rolle, entweder wird dieser Event ausgelöst oder nicht. Anders verhält sich die Sachlage bei dem oben erwähnten **MouseMove**-Ereignis, das permanent vom Objekt ausgelöst wird, wenn sich der Mauszeiger über die Oberfläche einer visuellen Komponente bewegt: Bei jeder Ereignisauslösung werden dem Eventhandler im zweiten Parameter die augenblicklichen x- und y-Koordinaten mitgeteilt.

Beachten Sie dazu, dass der zweite Parameter des **Click**-Ereignis von einem anderen Typ (**EventArgs**) ist als der zweite Parameter des **MouseMove**-Events (**MouseEventArgs**).

Entwickeln eines konventionsgerechten Events

Um ein tieferes Verständnis der Events zu bekommen, werden wir nun schrittweise eine Klasse entwickeln, die einen Event nach den Designrichtlinien, die im letzten Abschnitt erläutert worden sind, bereitstellt.

Die Klasse soll *TestEventClass* heißen und das Ereignis *OnMousePress*. Der Bezeichner des Events klingt verheißungsvoll, suggeriert er doch sofort die Auslösung für den Fall, dass der Anwender eine Taste der Maus drückt. So leistungsfähig wird unser Ereignis am Ende nicht sein, dafür müssten wir noch ein paar Zeilen Code mehr einbeziehen. Aber Sie werden nach einem genauen Studium des Beispiels eine sehr gute Grundlage haben, um später mit der scheinbaren Eigenwilligkeit der Ereignisdefinitionen in Windows-Anwendungen geradezu spielerisch umzugehen.

Am Anfang steht die Idee, das Ereignis *OnMousePress* in der Klasse *TestEventClass* bereitzustellen. Das Ereignis soll mit der spezifischen Leistungsfähigkeit ausgestattet sein, die X- und Y-Koordinaten beim Drücken der Maustaste dem Eventhandler mitzuteilen.

Ein Ereignis ist immer vom Typ **Delegate**. Die Designrichtlinien besagen, dass dieser Delegate einen Zeiger auf eine Prozedur kapseln soll, die zwei Parameter bereitstellt:

- Der erste Parameter liefert die Referenz auf ein Objekt. Da der Typ des Objekts nicht bekannt ist, wird dieser Parameter `As Object` deklariert und gilt insofern für alle Klassen des .NET-Frameworks.
- Im zweiten Parameter wird ebenfalls ein Objekt übergeben. Der Typ dieses Objekts ist allerdings etwas spezieller, es ist vom Typ **EventArgs**.

Der Typ **EventArgs** ist in der Klassenbibliothek im Namespace **System** definiert und dient als Basisklasse aller Klassen, die Ereignisdaten bereitstellen. Konsequenterweise bedeutet das auch, dass wir eine Klasse definieren müssen, die aus der Klasse **EventArgs** abgeleitet ist und die gewünschten Daten über die öffentliche Schnittstelle bereitstellt, die wir dem späteren Ereignis mit auf dem Weg geben werden. Diese Klasse soll *ClassForEventHandler* lauten, die Implementierung sieht wie folgt aus:

```
Public Class ClassForEventHandler : Inherits EventArgs
   Private intX As Int32
   Private intY As Int32

   Public Sub New(ByVal x As Int32, ByVal y As Int32)
      intX = x
      intY = y
   End Sub

   Public ReadOnly Property X_Value() As Int32
      Get
         Return intX
      End Get
   End Property

   Public ReadOnly Property Y_Value() As Int32
      Get
         Return intY
      End Get
   End Property
End Class
```

Diese Klasse veröffentlicht die Eigenschaften *X* und *Y*, die später beim Aufruf des Ereignisses angezeigt werden sollen. Dem Konstruktor werden beim Aufruf die beiden Werte genügen. Diese einfache Implementierung soll für unser Beispiel vollkommen ausreichend sein.

Nun wollen wir die Klasse *EventTestClass* entwickeln, die in der Lage ist, das Ereignis *OnMousePress* auszulösen.

```
Public Class EventTestClass
   Public Delegate Sub MyEventHandler(ByVal sender As Object, _
                          ByVal e As ClassForEventHandler)
   Public Event OnMousePress As MyEventHandler
   Public Sub TestProc()
      RaiseEvent OnMousePress(Me.ToString(), _
                           New ClassForEventHandler(4, 3))
   End Sub
End Class
```

Zunächst ist ein Delegate deklariert, *MyEventHandler*, der sich an den Designrichtlinien orientiert: Der erste Parameter nimmt die Referenz auf ein Objekt des .NET-Frameworks entgegen, der zweite ist ein von **EventArgs** abgeleiteter Typ, nämlich vom Typ der von uns im ersten Schritt entwickelten Klasse *ClassForEventHandler*. Im zweiten Schritt wird das Ereignis *OnMousePress* deklariert.

Die Klasse enthält nur die Methode *TestProc*, bei deren Aufruf der klassenspezifische Event ausgelöst wird. Beachten Sie, dass hinter **RaiseEvent** nicht nur der Name des Events, sondern auch die erforderlichen Argumente übergeben werden: Das erste Argument liefert als **String** den Namen der Klasse, der zweite Parameter eine implizite Referenz auf ein Objekt vom Typ der Klasse, die für uns die eventspezifischen Daten bereitstellt – *ClassForEventHandler*.

Was wir bisher entwickelt haben, gleicht dem, womit Sie bei Windows-Anwendungen permanent in Kontakt kommen: Es liegt eine Klasse vor, die zwar keine visuelle Oberfläche hat wie eine WinForm oder ein beliebiges Steuerelement, dafür veröffentlicht unsere Klasse einen Event, der sogar in der Lage ist, spezifische Daten an den Nutzer der Klasse zu übermitteln. Dass diese Daten in unserem Beispiel statisch sind, sollte nicht weiter stören – es kommt darauf an, das Prinzip zu verstehen.

Nun müssen wir uns noch vom Erfolg unserer Bemühungen überzeugen und einen Testclient entwickeln. Dazu dient der folgende Code:

```
'-------------------------------------------------------------
'Codebeispiel: ...\Beispielcode\Kapitel_7\UserDefinedEvent
'-------------------------------------------------------------
Module Module1
   Dim WithEvents obj As New EventTestClass()

   Sub Main()
      obj.TestProc()
   End Sub

   Public Sub obj_OnMousePress(ByVal sender As System.Object, _
                  ByVal e As ClassForEventHandler) _
                  Handles obj.OnMousePress
      Console.WriteLine("X-Wert = {0}", e.X)
      Console.WriteLine("Y-Wert = {0}", e.Y)
      Console.ReadLine()
   End Sub
End Module
```

Tatsächlich werden Sie an der Konsole die beiden Werte der Koordinaten X und Y angezeigt bekommen, die der Event an den Ereignisempfänger geschickt hat.

7.2.11 Zusammenfassung

- Ein Delegate ist ein Objekt und kapselt den Zeiger auf eine Objektmethode.
- Bei der Delegate-Deklaration ist eine Parameterliste zu definieren, die der Parameterliste der Methode entspricht, dessen Adresse der Delegate repräsentiert.
- Ein Delegate-Objekt wird mit dem Operator `New` erzeugt. Als Argument des Konstruktoraufrufs muss dabei mit `AddressOf <Objekt>.<Methode>` die Adresse der gewünschten Methode übergeben werden. Aufgerufen wird die gekapselte Methode mit `Invoke` unter Übergabe der gegebenenfalls von der Methode eingeforderten Argumente.
- Alle Delegates leiten sich aus der Klasse **System.Delegate** ab, die neben der Methode `Invoke` auch andere Methoden an einen benutzerdefinierten Delegate vererbt.
- Mehrere Delegate-Objekte können mit der überladenen Methode `Combine` zu einem Multicast-Delegate zusammengefasst werden. Der Aufruf der `Invoke`-Methode auf ein Multicast-Delegate-Objekt hat die Ausführung aller enthaltenen Delegates zur Folge. Multicast-Delegates leiten sich aus der Klasse **System.MulticastDelegate** des .NET-Frameworks ab.
- Aufrufe werden als synchron bezeichnet, wenn die Methoden der Reihe nach abgearbeitet werden. Man bezeichnet Methodenaufrufe als asynchron, wenn die Aufrufe parallel (oder quasi-parallel) bearbeitet werden.
- Das `WithEvents`-Statement legt zusammen mit der `Handles`-Klausel den Eventhandler fest. Die Bindung des Eventhandlers an das Ereignis ist statisch.
- Die Schlüsselwörter `AddHandler` und `RemoveHandler` binden ein Ereignis an eine Prozedur bzw. lösen diese Verbindung. Damit ist diese Bindung dynamisch.
- Die Parameterliste eines Ereignisses darf keine optionalen Parameter enthalten. Daher ist auch die Angabe eines **ParamArrays** nicht erlaubt.

Inhalt