9 Dateisystemüberwachung

Ich habe schon Anwendungen gesehen, die nur darauf warten, dass Dateien in einem bestimmten Verzeichnis gespeichert werden, um sie dann zu verarbeiten, z.B. eine Anwendung, die Daten aus einer Datei in eine Datenbank importiert. Datendateien können von einem Mainframe-System heruntergeladen oder auf andere Weise in ein Eingabeverzeichnis übertragen und anschließend von einer Anwendung in eine Datenbank importiert werden. Eine Anwendung muss aber nicht ständig das Verzeichnis nach neuen Dateien durchsuchen, sondern kann auf eine Meldung warten, die darüber informiert, dass eine neue Datei erstellt wurde. Programme mit dieser Funktionalität können zwar auch in Visual Basic 6 geschrieben werden, dies erfordert aber eine genaue Kenntnis der Win32-APIs. In Visual Basic .NET wird diese Aufgabe durch Verwendung der .NET-Frameworkklassen erheblich vereinfacht. Da die Implementierung eines solchen Programms in Microsoft .NET nach demselben Prinzip erfolgt wie alles andere in .NET, ist zudem nur ein geringer Lernaufwand erforderlich.

Das .NET Framework verfügt über eine vordefinierte Klasse namens *System.IO.FileSystemWatcher*, mit deren Hilfe Sie ein Programm zur Überwachung des Dateisystems erstellen können. Diese Klasse stellt Eigenschaften bereit, mit denen Sie den zu überwachenden Pfad angeben und festlegen können, ob Änderungen auf Datei- oder Unterverzeichnisebene überwacht werden sollen. Die *System.IO.FileSystemWatcher*-Klasse ermöglicht auch die Bestimmung der zu überwachenden Dateinamen und Typen. (Mit **.txt* werden beispielsweise Änderungen an sämtlichen Textdateien überwacht.) Sie können sogar angeben, welche Arten von Änderungen Sie verfolgen möchten – so z.B. neue Dateien, veränderte Dateiattribute oder geänderte Dateigrößen.

Nachdem Sie die Art der Dateien und Änderungen spezifiziert haben, können Sie Ereignishandler für die verschiedenen zu überwachenden Ereignisse scheiben. Bei den *FileSystemWatcher*-Klassenereignissen, die verfolgt werden können, handelt es sich um *Changed*, *Created*, *Deleted*, *Error* und *Renamed*. Für die Ereignisbehandlung schreiben Sie einen Ereignishandler, der dieselbe Deklaration aufweist wie der *FileSystemEventHandler*-Delegate. Anschließend fügen Sie diesen Ereignishandler der *FileSystemWatcher*-Klasse hinzu. (Das Programm, das wir im Lauf dieses Kapitels erstellen, veranschaulicht die Verwendung von Delegates.) In dieser auf Delegates basierenden Architektur können Sie mehrere Handler für dasselbe Ereignis oder einen Handler für mehrere Ereignisse verwenden. Das war in Visual Basic 6 nicht möglich.

Das File Sentinel-Programm

Anhand des File Sentinel-Programms werden Sie mehr über das .NET Framework erfahren. Darüber hinaus können Sie das Programm sofort einsetzen, wenn Sie im Bereich Netzwerkverwaltung tätig sind. Diese Anwendung wird im Hintergrund ausgeführt und überwacht Verzeichnisse und Dateien auf Veränderungen. Bei der Netzwerkverwaltung empfiehlt es sich, bestimmte Dateien im Auge zu behalten und über Dateiänderungen benachrichtigt zu werden. Sie können einen solchen »Wachtposten« auch auf einem Webserver einsetzen, damit Sie gewarnt werden, wenn sich jemand (sprich: ein Hacker) an den Dateien auf Ihrem Server zu schaffen macht. Sie könnten auch das Cookie-Verzeichnis überwachen, um nicht autorisierte Zugriffe auf Ihren Computer zu ermitteln.

Netzwerkmanager in größeren Organisationen erstellen häufig Pseudodateien mit verlockenden Namen wie *Lohnabrechnung.xls* oder *Kennwort.bin*, um festzustellen, ob jemand unberechtigterweise versucht, auf diese Dateien zuzugreifen. Sie stellen sozusagen einen »Honigtopf« ins Netzwerk. Ähnlich wie bei einer verdeckten Ermittlung zur Aufdeckung von Korruption soll der Honigtopf einen Anreiz für allzu neugierige Benutzer bieten. Mithilfe des File Sentinel-Programms können Sie den Honigtopf überwachen und feststellen, ob jemand versucht, darauf zuzugreifen. Wenn ein Benutzer versucht, den Honig zu stehlen, kann das File Sentinel-Programm Sie bei der Überführung unterstützen.

Dieses Programm können Sie zur Überwachung von Dateien auf einem lokalen Rechner oder in einem Netzwerk einsetzen. Während wir das Programm in diesem Kapitel schreiben, werden Sie Näheres über das Erstellen und Bearbeiten von Dateien sowie über Ereignisse und die neuen .NET-Delegates erfahren. In Visual Basic 6 waren Ereignisse vorgegebene Elemente, die nicht verändert oder erweitert werden konnten. Mithilfe der Delegates in Visual Basic .NET können Sie einem Programm selbst definierte Ereignisse hinzufügen.

ANMERKUNG: Das File Sentinel-Programm funktioniert nur unter Windows 2000 und Windows NT 4.0. Leider stellt das .NET Framework keine Möglichkeit bereit, diese Anwendung für Windows 9.*x* oder Windows Me zu schreiben. Die *FileSystemWatcher*-Klasse kann auch Disketten überwachen, solange diese nicht ausgewechselt oder entfernt werden. *FileSystemWatcher* löst keine Ereignisse für CDs und DVDs aus, da sich dort Zeitstempel und Eigenschaften nicht verändern. Damit Sie mit der Komponente auch Remotecomputer überwachen können, muss auf diesen ebenfalls entweder Windows 2000 oder Windows NT 4.0 installiert sein. Allerdings können Sie einen Windows NT 4.0-Computer nicht von einem Windows NT 4.0-Computer aus überwachen. Dies wird sich in zukünftigen .NET Framework-Versionen hoffentlich ändern.

So funktioniert das File Sentinel-Programm

Bevor wir mit dem Code beginnen, zunächst ein kurzer Blick auf die Funktionsweise des Programms. In dieser Anwendung kann ein Benutzer eine Datei oder einen Ordner zur Überwachung auswählen. Beachten Sie, dass in Abbildung 9.1 die Schaltfläche *Disable Sentinel* nicht zur Verfügung steht. Der Benutzer muss zunächst eine Datei oder einen Ordner zur Überwachung auswählen, bevor das Programm beginnen kann. Für den Fall, dass ein Benutzer etwas zu schnell auf *Enable Sentinel* klickt und zuvor keine Auswahl getroffen hat, legen wir als Standardverzeichnis das aktuelle Laufwerk fest. Wie immer schützen wir die Benutzer davor, versehentlich Fehler zu machen.

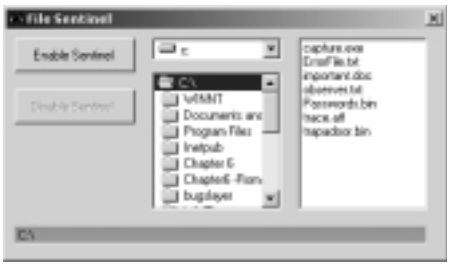

Abbildung 9.1: *Das File Sentinel-Programm: Die Anwendung startet erst nach der Auswahl einer Datei oder eines Ordners*

Zum Programm gehören auch einige QuickInfos, um die Benutzer bei der Verwendung zu unterstützen (siehe Abbildung 9.2). QuickInfos sind ausgesprochen einfach zu implementieren; sie erfordern nur eine einzige Codezeile und lassen eine Anwendung höchst professionell aussehen. In früheren Visual Basic-Versionen wurde in der *Tag*-Eigenschaft eines Steuerelements eine Zeichenfolge mit Hilfeinformationen zum Steuerelement gespeichert. Anschließend musste auf umständliche Weise mithilfe eines Timers und dem *Mouse_Move*-Ereignis Code hinzugefügt werden, um manuell eine QuickInfo anzuzeigen. In Visual Basic .NET gibt es nun eine viel einfachere Möglichkeit zur Anzeige von QuickInfos.

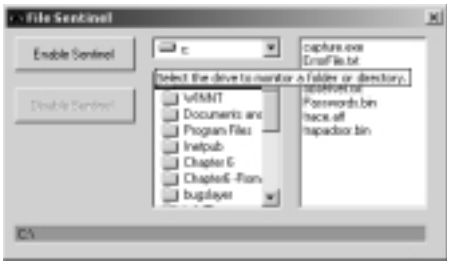

Abbildung 9.2: *Implementieren von QuickInfos als Hilfe für die Benutzer*

Die vom Programm erzeugte Ausgabe erfolgt in Form einer einfachen Textdatei. Natürlich können Sie das Programm mühelos so gestalten, dass es E-Mail-Nachrichten an Ihren Computer oder sogar eine Meldung an Ihren Pager schickt, sobald ein unberechtigter Zugriff auf die Serverdateien entdeckt wird. Für unsere Zwecke ist eine Textdatei aber völlig ausreichend.

Das Programm hält Datum und Uhrzeit des unberechtigten Zugriffs sowie die betroffenen Dateien oder Ordner und die Art des Zugriffs fest. Sie können die Ausgabedatei im Editor öffnen und so jeden überwachten Zugriff mit Datum und Uhrzeit ermitteln. Abbildung 9.3 zeigt, dass eine Datei mit dem Namen *trapdoor.bin* in *trash.doc* umbenannt und anschließend eingesehen wurde. Sie sehen also, wie ausgesprochen nützlich dieses Programm für Sie sein kann.

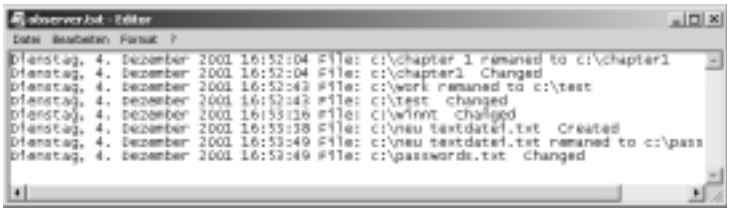

Abbildung 9.3: *Die File Sentinel-Ausgabe in einer Textdatei*

Das File Sentinel-Programm – Einstieg

Wie bereits in den vorherigen Kapiteln ausgeführt, besteht die Entwicklung einer Visual Basic .NET-Anwendung aus drei Schritten:

1. Erstellen der Schnittstelle
2. Setzen der Steuerelementeigenschaften
3. Schreiben des Codes

Im Gegensatz zu früheren Zeiten, als es noch keine visuellen Sprachen wie Visual Basic und Visual C++ gab und die Benutzerschnittstelle erst nachträglich implementiert wurde, sollte die Schnittstellenerstellung in Visual Basic .NET den ersten Schritt darstellen. Auch wenn das File Sentinel-Programm nur eine kleine Anwendung ist, können Sie dadurch die Gesamtgestaltung auf relativ einfache Weise festlegen. Die Benutzerschnittstelle sollte von Anfang an richtig geplant werden. Denken Sie daran: Die Benutzer setzen die Oberfläche mit dem Programm gleich.

Hinzufügen von Steuerelementen zur Toolbox

In der Toolbox werden drei Steuerelemente benötigt: *DirListBox*, *DriveListBox* und *FileListBox*. Diese Steuerelemente sind alte Bekannte aus Visual Basic, die für .NET etwas modernisiert wurden. Um die Steuerelemente hinzuzufügen, klicken Sie mit der rechten Maustaste auf die Toolbox, und wählen Sie *Toolbox anpassen*. Klicken Sie auf die Registerkarte *.NET Framework-Komponenten* (siehe Abbildung 9.4), wählen Sie die Steuerelemente aus, und klicken Sie anschließend auf *OK*.

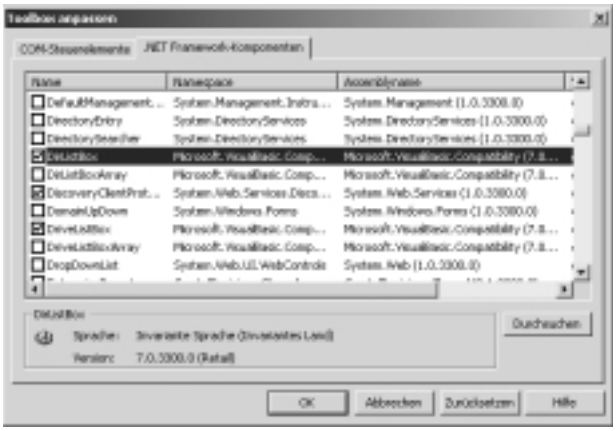

Abbildung 9.4: *Diese drei Steuerelemente werden in der Toolbox unter Windows Forms hinzugefügt*

Erstellen der Benutzerschnittstelle und Setzen der Eigenschaften

Erstellen Sie ein neues Visual Basic .NET-Windows-Projekt mit dem Namen *FileSentinel*, fügen Sie dem Standardformular die in Tabelle 9.1 aufgeführten Steuerelemente hinzu, und legen Sie die Werte für die zugehörigen Eigenschaften fest. Das Formular sollte im Entwurfsmodus jetzt etwa so aussehen wie das in Abbildung 9.5 gezeigte Formular.

Steuerelement	Eigenschaft	Wert
Button	Name	btnEnable
	Text	&Enable Sentinel
Button	Name	btnDisable

Steuerelement	Eigenschaft	Wert
	Text	&Disable Sentinel
Label	Name	lblWatching
	BackColor	Lime
	BorderStyle	Fixed3D
Form	FormBorderStyle	FixedDialog
	Icon	<Eigene Auswahl>
	Text	File Sentinel
	StartPosition	CenterScreen
	Locked	True
ToolTip	Name	ttTip
DriveListBox	Name	DriveListBox1
DirListBox	Name	DirListBox1
FileListBox	Name	FileListBox1

Tabelle 9.1: *Steuerelemente und Eigenschaften für das File Sentinel-Programm*

ANMERKUNG: Wie bereits in ▸Kapitel 2, »Objektorientierte Programmierung in Visual Basic .NET«, erwähnt, wird das QuickInfo-Steuerelement im Bereich unterhalb des Formulars platziert, da QuickInfos auf der Anwendungsoberfäche zunächst nicht sichtbar sind. In diesen Bereich werden alle nicht sichtbaren Steuerelemente wie QuickInfos, *Error Provider*-Objekte oder Zeitgeber platziert. Dieses Merkmal der Visual Basic .NET-IDE bietet den Vorteil, dass die in diesem Bereich abgelegten Steuerelemente bei der Benutzerschnittstellenentwicklung keinen Platz auf dem Formular beanspruchen.

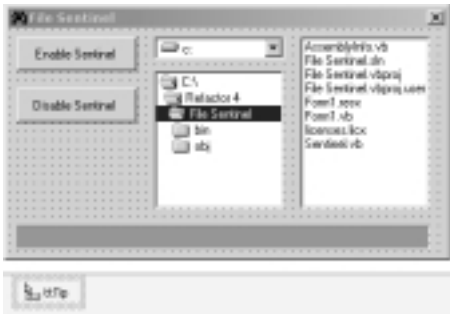

Abbildung 9.5: *Das Formular File Sentinel mit eingefügten Schnittstellensteuerelementen*

Nachdem wir nun die Steuerelemente erstellt und die Eigenschaften gesetzt haben, wird es Zeit, die Ärmel hochzukrempeln und ein bisschen Code zu schreiben. Wir werden die Funktionalität des File Sentinel-Programms in einer Klasse kapseln. Das soeben erstellte Formular stellt die Fassade dieses Programms nach außen dar.

Ein paar Worte zu ActiveX-Legacysteuerelementen

Sie haben sich sicherlich gewundert, warum bisher keines der altbekannten COM-ActiveX-Steuerelemente (.ocx) verwendet wurde. Die Common Language Runtime (CLR) verwaltet sämtlichen Code, der im .NET Framework ausgeführt wird. Unter der Kontrolle der CLR ausgeführter Code wird als *verwalteter* Code bezeichnet. Umgekehrt heißt Code, der außerhalb der CLR ausgeführt wird, *nicht verwalteter* Code. Bei COM-Komponenten, ActiveX-Schnittstellen und Win32-API-Funktionen handelt es sich um nicht verwaltete Codebestandteile.

Vermutlich haben Sie eigene COM-Steuerelemente erstellt oder einige teure Steuerelemente erworben, die für .NET noch nicht zur Verfügung stehen. In dem meisten Fällen ist es weder besonders praktisch noch überhaupt notwendig, eine COM-Komponente zu aktualisieren, nur um ihre Funktionalität in den verwalteten Code für eine Anwendung zu integrieren. Häufig ist es sinnvoller, mithilfe der von der CLR bereitgestellten Dienste auf vorhandene Funktionalität zurückzugreifen.

.NET Windows-Formulare können nur Steuerelemente der Klasse *System.Windows.Forms.Control* enthalten. Damit ein ActiveX-Legacysteuerelement in .NET funktioniert, muss es wie ein Windows Forms-Steuerelement aussehen. Ebenso erwartet das ActiveX-Steuerelement keine .NET-Umgebung, sondern einen ActiveX-Container. Hier hilft glücklicherweise die *System.Windows.Forms.AxHost*-Klasse. Diese Klasse verhält sich nach außen hin wie ein Windows Forms-Steuerelement, nach innen stellt es einen Container für ActiveX-Steuerelemente dar. Im Grunde erstellt die *AxHost*-Klasse eine *Wrapper*-Klasse, die die Eigenschaften, Methoden und Ereignisse offen legt. Das eine oder andere ActiveX-Steuerelement funktioniert vielleicht besser als andere Steuerelemente; wenn Sie also tatsächlich ein Legacysteuerelement benötigen, sollten Sie sich anhand des Windows Class Viewers (den ich in ▸ Kapitel 5, »Betrachtung des .NET-Klassenframeworks mithilfe von Dateien und Zeichenfolgen« vorgestellt habe) einen genauen Überblick verschaffen.

Hinzufügen der *Sentinel*-Klasse

Die .NET-Frameworkklasse *FileSystemWatcher* ist so praktisch, dass sie als Komponente in der Toolbox enthalten ist. Wir könnten in unser Formular zwar ein *FileSystemWatcher*-Steuerelement einbauen und dessen Eigenschaften festlegen, aber wir werden eine eigene Komponente in einer Klasse erstellen. Diese Vorgehensweise hat einen einfachen Grund: die Komponente soll von der vordefinierten Frameworkklasse erben, und anschließend soll ihr Funktionalität hinzugefügt werden, wie z.B. die Fähigkeit, in eine Datei zu schreiben oder einem Benutzer zu ermöglichen, Dateien oder Verzeichnisse zur Überwachung auszuwählen. Abbildung 9.6 zeigt die auf der Toolbox-Registerkarte *Komponenten* angezeigte *FileSystemWatcher*-Komponente.

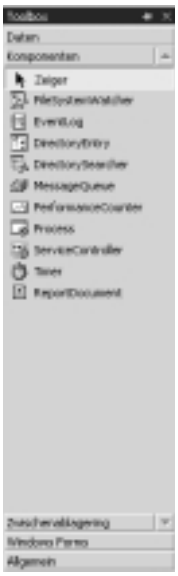

Abbildung 9.6: *Die Toolbox-Registerkarte Komponenten*

Hinzufügen einer Klasse zum Projekt

Die Klasse, die die *FileSystemWatcher*-Komponente implementiert, ist in unserem Programm für den größten Teil der Arbeit verantwortlich. Nach der Erstellung werden wir die Klasse in die Benutzerschnittstelle einbauen. Zunächst werden wir uns aber ansehen, wie die Klasse funktioniert.

1. Wählen Sie *Projekt | Klasse hinzufügen*.
2. Wählen Sie die Klassenvorlage, geben Sie der Klasse den Namen *Sentinel.vb*, und klicken Sie auf *Öffnen*. Löschen Sie den vorhandenen Code, und fügen Sie stattdessen folgenden Code ein:

```
Imports System
Imports System.Diagnostics
Imports System.IO
Imports System.Threading

Namespace SystemObserver

    Public Class sentinel

        Private m_Watcher As System.IO.FileSystemWatcher
        Private m_ObserveFileWrite As StreamWriter
        Private fiFileInfo As FileInfo

        Public Sub New(ByVal sToObserve As String)
            m_Watcher = New FileSystemWatcher()
            fiFileInfo = New FileInfo(sToObserve)

           If (fiFileInfo.Exists = False) Then

               If (Not sToObserve.EndsWith("\")) Then
                   sToObserve.Concat("\")
               End If
```

```
        With m_Watcher
            .Path = sToObserve
            .Filter = ""
            .IncludeSubdirectories = False
        End With
    Else
        With m_Watcher
            .Path = fiFileInfo.DirectoryName.ToString
            .Filter = fiFileInfo.Name.ToString
            .IncludeSubdirectories = False
        End With
    End If

    m_Watcher.NotifyFilter = _
        NotifyFilters.FileName Or _
        NotifyFilters.Attributes Or _
        NotifyFilters.LastAccess Or _
        NotifyFilters.LastWrite Or _
        NotifyFilters.Security Or _
        NotifyFilters.Size Or _
        NotifyFilters.CreationTime Or _
        NotifyFilters.DirectoryName

    AddHandler m_Watcher.Changed, AddressOf OnChanged
    AddHandler m_Watcher.Created, AddressOf OnChanged
    AddHandler m_Watcher.Deleted, AddressOf OnChanged
    AddHandler m_Watcher.Renamed, AddressOf OnRenamed
    AddHandler m_Watcher.Error, AddressOf onError

    m_Watcher.EnableRaisingEvents = True
    m_ObserveFileWrite = _
        New StreamWriter("C:\observer.txt", True)
End Sub

Private Sub OnChanged(ByVal source As Object, _
    ByVal e As FileSystemEventArgs)

    Dim sChange As String

    Select Case e.ChangeType
        Case WatcherChangeTypes.Changed : _
            sChange = "Changed"
        Case WatcherChangeTypes.Created : _
            sChange = "Created"
        Case WatcherChangeTypes.Deleted : _
             sChange = "Deleted"
     End Select

     If (Len(sChange) > 0) Then
         If (e.FullPath.IndexOf("observer.txt") > 0) _
             Then

             Exit Sub
         End If
     End If
```

```
            writeToFile("File: " & e.FullPath & _
                "  " & sChange)
        End Sub

        Private Sub OnRenamed(ByVal source As Object, _
            ByVal e As RenamedEventArgs)

            writeToFile("File: " & e.OldFullPath &  _
                " renamed to " & e.FullPath)
        End Sub

        Private Sub onError(ByVal source As Object, _
            ByVal errevent As ErrorEventArgs)

            writeToFile("ERROR: " & _
                errevent.GetException.Message())
        End Sub

        Private Sub writeToFile( _
            ByRef observeString As String)
            Dim sRightNow As String = _
                Date.Now.ToLongDateString() & _
                " " & Date.Now.ToLongTimeString()

            Try
                m_ObserveFileWrite.WriteLine(sRightNow & _
                    " " & observeString)
                m_ObserveFileWrite.Flush()
            Catch
            End Try
        End Sub

        Public Sub dispose()
            m_Watcher.EnableRaisingEvents = False
            m_Watcher = Nothing
            m_ObserveFileWrite.Close()
        End Sub

    End Class

End Namespace
```

So funktioniert der Code

Folgende vier Namespaces sollen importiert werden:

```
Imports System
Imports System.Diagnostics
Imports System.IO
Imports System.Threading
```

Anschließend wird die Klasse in den *SystemObserver*-Namespace eingebunden. Der Name der Klasse im Namespace lautet *sentinel*. Wir deklarieren drei private Klassenmembervariablen. Die erste Variable lautet *m_Watcher* und ist vom Typ *FileSystemWatcher*. Die Frameworkklasse *FileSystemWatcher* befindet sich im *System.IO*-Namespace.

Da die Ausgabe in eine Datei erfolgen soll, erstellen wir zusätzlich eine Membervariable *m_ObserveFileWrite* vom Typ *StreamWriter*. Die dritte Variable soll prüfen, ob eine Datei oder ein Ver-

zeichnis überwacht werden soll. Die Variable *fiFileInfo* vom Typ *FileInfo* stellt die erforderlichen Methoden bereit. Da diese Variablen am Anfang der Klasse eingefügt werden, sind sie für die gesamte Klasse sichtbar.

```
Namespace SystemObserver

    Public Class sentinel

        Private m_Watcher As System.IO.FileSystemWatcher
        Private m_ObserveFileWrite As StreamWriter
        Private fiFileInfo As FileInfo
```

Bei der Instanziierung eines *sentinel*-Objekts wird der Pfad der zu überwachenden Datei bzw. des Verzeichnisses als Zeichenfolge an den Klassenkonstruktor übergeben. Nun liegt zwar eine neue Instanz der *FileSystemWatcher*-Klasse vor, aber zu diesem Zeitpunkt ist noch nicht bekannt, ob die Zeichenfolgenvariable *sToObserve* eine Datei oder ein Verzeichnis enthält. Mithilfe der *FileInfo*-Klasse lässt sich feststellen, was überwacht werden soll.

```
Public Sub New(ByVal sToObserve As String)
    m_Watcher = New FileSystemWatcher()
    fiFileInfo = New FileInfo(sToObserve)
```

Konfigurieren der *FileSystemWatcher*-Klasse

Um das Verhalten der *FileSystemWatcher*-Klasse zu steuern, müssen verschiedene Eigenschaften festgelegt werden. Diese Eigenschaften bestimmen, welche Verzeichnisse und Unterverzeichnisse überwacht werden, und welche Vorkommnisse in diesen Verzeichnissen tatsächlich ein Ereignis auslösen sollen.

Die ersten beiden Eigenschaften zur Festlegung der von *FileSystemWatcher* zu überwachenden Verzeichnisse lauten *Path* und *IncludeSubdirectories*. Die Eigenschaft *Path* gibt den vollständigen Pfad des zu überwachenden Stammverzeichnisses an. Der Wert dieser Eigenschaft kann als Standardverzeichnispfad (*C:\Verzeichnis*) oder im UNC-Format (*\\Server\Verzeichnis*) angegeben werden. Die Eigenschaft *IncludeSubdirectories* bestimmt, ob auch die Unterverzeichnisse des Stammverzeichnisses in die Überwachung eingeschlossen werden. Wenn diese Eigenschaft auf *True* gesetzt wird, werden in den Unterverzeichnissen und im Hauptverzeichnis die gleichen Veränderungen verfolgt. Allerdings kann das Setzen von *IncludeSubdirctories* zur Folge haben, dass jedes überwachte Ereignis 10 bis 15 weitere Ereignisse erzeugt, die eigentlich gar nicht erwünscht sind. Das Windows-Betriebssystem erzeugt bei jeder Dateiänderung durch einen Benutzer Unmengen von Meldungen zu internen Dateien. Daher sollten Sie bei einer Überwachung des Stammverzeichnisses den Wert der *IncludeSubdirectories*-Eigenschaft besser auf *False* setzen.

Wenn der Benutzer den vollqualifizierten Pfadnamen einer Datei angibt, gibt die *fiFileInfo.Exists*-Methode den Wert *True* zurück. Handelt es sich um ein Verzeichnis, gibt die *Exists*-Methode den Wert *False* zurück. Beginnen wir mit der Überwachung für Verzeichnisse.

Die Angabe eines Verzeichnispfades wie z.B. »C:\« stellt uns vor keinerlei Probleme. Wenn die Eingabe jedoch z.B. »C:\Programme\Gemeinsame Dateien« lautet, muss ein umgekehrter Schrägstrich (\) als Trennzeichen eingefügt werden. Mithilfe von zwei Methoden des *String*-Objekts ist das ein Klacks. Wenn die Zeichenfolge *sToObserve* nicht auf einen umgekehrten Schrägstrich endet, wird automatisch einer eingefügt. So einfach ist das.

Wird ein Verzeichnis ausgewählt, setzen wir die Variable *sToObserve* auf diesen Pfad. Möchte der Benutzer alle Dateien im Verzeichnis »C:\Programme\Gemeinsame Dateien« überwachen, wird am Ende der Zeichenfolge einfach ein umgekehrter Schrägstrich eingefügt und die *Path*-Eigenschaft gesetzt.

Da in unserem Beispiel Änderungen an allen Dateien des ausgewählten Verzeichnisses verfolgt werden sollen, setzen wir die *Filter*-Eigenschaft auf eine leere Zeichenfolge (""). Zur Überwachung einer bestimmten Datei legen Sie den Wert der *Filter*-Eigenschaft auf den Dateinamen fest. Soll beispielsweise die Datei *Kennwort.bin* überwacht werden, setzen Sie die *Filter*-Eigenschaft auf »Kennwort.bin«. Es können auch bestimmte Dateitypen zur Überwachung ausgewählt werden. Wenn z.B Änderungen an allen Microsoft Word-Dateien verfolgt werden sollen, setzen Sie die *Filter*-Eigenschaft auf »*.doc«.

Wird kein Verzeichnis, sondern eine Datei zur Überwachung ausgewählt, wissen wir, dass das *fiFileInfo*-Objekt alle erforderlichen Dateiinformationen enthält. Die *Path*-Eigenschaft des *FileSystemWatcher*-Objekts wird einfach auf *fiFileInfo.DirectoryName.ToString* gesetzt. Dieser Aufruf gibt den vollqualifizierten Pfad für die Datei zurück. Ebenso gibt die *Name*-Eigenschaft den Namen der zu überwachenden Datei zurück. Egal, ob Dateien oder Verzeichnisse überwacht werden sollen, die Eigenschaft *IncludeSubDirectories* wurde auf den Wert *False* gesetzt. Dadurch erhalten wir eine sauberere Protokolldatei, die nur Einträge für die fraglichen Dateien enthält.

```
If (fiFileInfo.Exists = False) Then

    If (Not sToObserve.EndsWith("\")) Then
        sToObserve.Concat("\")
    End If

    With m_Watcher
        .Path = sToObserve
        .Filter = ""
        .IncludeSubdirectories = False
    End With
Else
    With m_Watcher
        .Path = fiFileInfo.DirectoryName.ToString
        .Filter = fiFileInfo.Name.ToString
        .IncludeSubdirectories = False
    End With
End If
```

Nachdem wir die Eigenschaften *Path*, *Filter* und *IncludeSubdirectories* für das *FileSystemWatcher*-Objekt festgelegt haben, werden wir nun bestimmen, welche Arten von Datei- oder Verzeichnisänderungen verfolgt werden sollen. Dazu benötigen wir die Eigenschaft *NotifyFilter*.

```
m_Watcher.NotifyFilter = NotifyFilters.FileName Or _
    NotifyFilters.Attributes Or _
    NotifyFilters.LastAccess Or _
    NotifyFilters.LastWrite Or _
    NotifyFilters.Security Or _
    NotifyFilters.Size Or _
    NotifyFilters.CreationTime Or _
    NotifyFilters.DirectoryName
```

Im File Sentinel-Programm sollen alle Arten von Änderungen überwacht werden, daher nehmen wir mithilfe der *Or*-Anweisung eine Verkettung vor. Da es sich hier um eine Enumeration handelt (anders gesagt: die Werte werden hinter den Kulissen durch Zahlen dargestellt), werden die Werte einfach über *Or* verkettet. Tabelle 9.2 beschreibt die verschiedenen Werte für *NotifyFilters*.

Membername	Beschreibung
Attributes	Die Attribute einer Datei oder eines Ordners
CreationTime	Datum und Uhrzeit der Datei- oder Ordnererstellung
DirectoryName	Der Name des Verzeichnisses
FileName	Der Name der Datei
LastAccess	Das Datum des letzten Datei- oder Ordnerzugriffs
LastWrite	Das Datum der letzten Schreiboperation für Datei oder Ordner
Security	Die Sicherheitseinstellungen einer Datei oder eines Ordners
Size	Die Datei- oder Ordnergröße

Tabelle 9.2: *Werte für die NotifyFilters-Eigenschaft*

Damit sämtliche Änderungen verfolgt werden, wurden alle Werte in dieser Enumeration miteinander kombiniert. Sie können aber auch nur zwei oder drei *NotifyFilters*-Eigenschaften auswählen und diese per *Or*-Anweisung verketten. Sie könnten beispielsweise Größenänderungen für eine Datei oder einen Ordner oder veränderte Sicherheitseinstellungen verfolgen.

```
m_Watcher.NotifyFilter = NotifyFilters.FileName Or _
    NotifyFilters.Attributes Or NotifyFilters.LastAccess Or _
    notifyFilters.LastWrite Or NotifyFilters.Security Or _
    NotifyFilters.Size or NotifyFilters.CreationTime Or _
    NotifyFilters.DirectoryName
```

Sie fragen sich vielleicht gerade, wie die zu überwachenden Ereignisse mit den Ereignishandlern verknüpft werden. Hier kommt das Konzept der *Delegates* ins Spiel.

Delegates

Bei einem Ereignis handelt es sich im Grunde um nichts anderes als eine Nachricht, die von einer beliebigen Komponente gesendet wird, um einer beliebigen anderen Komponente etwas mitzuteilen. Sie werden es kaum glauben: in .NET heißt ein Objekt, das ein Ereignis auslöst, *Ereignissender*, und das empfangende Objekt *Ereignisempfänger*. Nun stellt sich das Problem, dass der Ereignisempfänger wissen muss, was er empfangen soll. Wenn keine empfangende Komponente vorhanden ist, nützt es auch nichts, Ereignisse zu senden.

Es ist zwar nicht erforderlich, dass das sendende und das empfangende Ereignis voneinander Kenntnis haben, dennoch müssen Sender und Empfänger miteinander verknüpft werden, um sicherzustellen, dass die Meldung übermittelt wurde. Zu diesem Zweck wird ein Delegate verwendet. Ein Delegate formalisiert den Prozess der Deklaration einer Prozedur, die auf ein Ereignis reagiert.

Ein Delegate wird zur Übermittlung der Nachricht (des ausgelösten Ereignisses) zwischen Quelle und überwachender Komponente eingesetzt. Ein Empfänger registriert den Delegate in einem Sender, damit der Sender weiss, dass der Empfänger auf ein gesendetes Ereignis reagiert. Ein weiteres leistungsfähiges Merkmal von Delegates ist das so genannte *Multicasting*, bei dem eine Nachricht eines einzelnen Senders an mehrere Empfänger gesandt werden kann. Sender und Empfänger bilden eine 1:n-Beziehung. Wir werden nun das Gegenteil davon implementieren, indem wir einen Delegate erstellen, der die Nachrichten mehrerer Sender an einen einzelnen Empfänger übermittelt. Auf diese Weise lässt sich die sendende Komponente problemlos bestimmen, der Code bleibt übersichtlich und gut lesbar.

Je nachdem, welche Änderungen in der überwachten Datei bzw. dem Ordner eintreten, kann das *FileSystemWatcher*-Objekt vier verschiedene Ereignisse auslösen. Dies sind folgende:

- *Created* Wird beim Erstellen einer Datei oder eines Verzeichnisses ausgelöst.
- *Deleted* Wird beim Löschen einer Datei oder eines Verzeichnisses ausgelöst.
- *Renamed* Wird beim Umbenennen einer Datei oder eines Verzeichnisses ausgelöst.
- *Changed* Wird ausgelöst, sobald Änderungen an Größe, Systemattributen, letztem Schreibvorgang, letztem Zugriff, oder den NTFS-Sicherheitsberechtigungen eines Verzeichnisses oder einer Datei vorgenommen werden. Natürlich können mithilfe der *NotifyFilter*-Eigenschaft die von *Changed* ausgelösten Ereignisse begrenzt werden.

Für jedes dieser vier *FileSystemWatcher*-Ereignisse werden Ereignishandler definiert, die bei Veränderungen eine Methode aufrufen. Jeder Ereignishandler enthält zwei Parameter zur Verarbeitung des Ereignisses: den Parameter *sender*, der einen Verweis auf das ereignisauslösende Objekt enthält, und den Parameter *e*, der ein Objekt mit Ereignisinformationen bereitstellt.

Wie Sie wissen, stellt ein Ereignis eine Nachricht dar, die von einem Objekt ausgegeben wird, sobald eine Aktion stattgefunden hat. Bei dieser Aktion kann es sich um eine Handlung des Benutzers handeln (z.B. ein Mausklick), die Aktion kann aber auch von einer Programmkomponente oder dem Betriebssystem ausgelöst werden. In unserem Falle soll ein Ereignis erzeugt werden, sobald ein Benutzer eine Aktion durchführt, also beispielsweise eine Datei umbenennt.

Hier ist die *FileSystemWatcher*-Komponente der Ereignissender. Das Objekt oder die Prozedur, die das Ereignis aufzeichnet und darauf reagiert, stellt den Ereignisempfänger dar. Die Ereignissenderklasse erhält jedoch keine Informationen darüber, welches Objekt oder welche Methode das ausgelöste Ereignis empfängt (verarbeitet). Aus diesem Grund wird zwischen Quelle und Empfänger eine vermittelnde Komponente benötigt. Im .NET Framework ist ein besonderer Typ, ein so genannter *Delegate* definiert, der diese Funktionalität bereitstellt.

Wir werden in unserem Beispiel einen Handler für die Ereignisse *Changed*, *Created*, *Deleted*, *Renamed* und *Error* hinzufügen, die vom *m_Watcher*-Objekt ausgelöst werden können. Dafür verwenden wir den *AddressOf*-Operator, mit dem wir einen Funktionsdelegate erstellen. Dieser Delegate zeigt auf die Funktion, die durch den Prozedurnamen des Operators bestimmt wird. Sobald das *m_Watcher*-Objekt ein *Changed*-Ereignis auslöst, sollte der Benutzer des Überwachungsprogramms benachrichtigt werden, damit er bei Bedarf darauf reagieren kann. Zu diesem Zweck fügen wir einen Ereignishandler *OnChanged* hinzu, der jedes *Changed*-Ereignis empfängt. Mit folgender Programmzeile wird dem *m_Watcher.Changed*-Ereignis ein Handler hinzugefügt, dessen Standort durch den *AddressOf*-Operator für die Prozedur *OnChanged* bestimmt wird.

```
AddHandler m_Watcher.Changed, AddressOf OnChanged
```

Der Delegate, der dem *m_Watcher*-Objekt mitteilt, wohin *Changed*-Ereignisse übermittelt werden sollen, ist hinzugefügt. Nun schreiben wir die *OnChanged*-Ereignisprozedur. Diese Prozeduren werden wir als privat deklarieren, damit sie nur innerhalb unserer Klasse offen gelegt werden.

```
Private Sub OnChanged(ByVal source As Object, _
    ByVal e As FileSystemEventArgs)

'Bei Datei- oder Verzeichnisänderungen Maßnahmen ergreifen

End Sub
```

Wie bereits erwähnt, erstellen wir mithilfe des Visual Basic .NET-Operators *AddressOf* einen Funktionsdelegate, der auf die Funktion zeigt, die durch *procedurename* bestimmt wird. In diesem Fall ist dies die Funktion *OnChanged*. Ich habe den Delegate in Kurzschrift erstellt; die beiden folgenden Zeilen führen jedoch zum gleichen Ergebnis:

```
AddHandler m_Watcher.Changed, AddressOf OnChanged
AddHandler m_Watcher.Changed, _
    New EventHandler(AddressOf OnChanged)
```

Mit diesem Code wird der Empfänger (*OnChanged*) beim Absender der Nachricht, *m_Watcher.Changed*, registriert. Jedes *Changed*-Ereignis, das durch das *m_Watcher*-Objekt ausgelöst wird, wird vom *OnChanged*-Ereignishandler aufgezeichnet.

Wir werden alle fünf möglichen Ereignisse verarbeiten, die vom *m_Watcher*-Objekt ausgelöst werden können. Die Ereignisse *Changed*, *Created* und *Deleted* werden jedoch alle vom *OnChanged*-Ereignishandler verarbeitet, stellen also eine n:1-Beziehung dar. Mit folgendem Code werden die Delegates definiert und registriert:

```
AddHandler m_Watcher.Changed, AddressOf OnChanged
AddHandler m_Watcher.Created, AddressOf OnChanged
AddHandler m_Watcher.Deleted, AddressOf OnChanged
AddHandler m_Watcher.Renamed, AddressOf OnRenamed
AddHandler m_Watcher.Error, AddressOf onError
```

Abschließend wird die Eigenschaft *EnableRaisingEvents* auf *True* gesetzt. Die Eigenschaften *Path* und *EnableRaisingEvents* müssen gesetzt sein, damit das Objekt seine Arbeit aufnimmt. (Da wir Schreibvorgänge in Dateien bereits in ▸ Kapitel 5 behandelt haben, sollte das *StreamWriter*-Objekt Ihnen bereits wie ein alter Freund erscheinen.)

```
m_Watcher.EnableRaisingEvents = True
m_ObserveFileWrite = _
    New StreamWriter("C:\observer.txt", True)
```

Nun ist das *m_Watcher*-Objekt in der Lage, eine Ereignisüberwachung durchzuführen und die entsprechenden Ereignisse in die Textdatei *C:\observer.txt* zu schreiben.

Verarbeiten der Ereignisse *Changed*, *Created* und *Deleted*

Wie oben bereits ausgeführt, werden die drei Ereignisse *Changed*, *Created* und *Deleted* allesamt durch den *OnChanged*-Ereignishandler verarbeitet. Mit dem *source*Parameter kann ermittelt werden, von welcher Quelle das Ereignis gesendet wurde. Im *FileSystemEventArgs*-Parameter sind Informationen über die jeweilige Nachricht enthalten. Die *ChangeType*-Eigenschaft von *FileSystemEventArgs* gibt Auskunft über die Art der eingetretenen Veränderung. Um herauszufinden, was genau passiert ist, prüfen wir also einfach den *FileSystemEventArgs*-Parameter.

Wir machen jetzt einen kleinen Umweg und suchen mithilfe unseres alten Bekannten, dem WinCV-Tool, nach *FileSytemEventArgs*. Wir wollen anhand der *ChangeType*-Eigenschaft herausfinden, welche Änderungen eingetreten sind, und mit *FullPath* ermitteln wir, welche Datei betroffen ist. Um es noch einmal zu wiederholen: Sie sollten sich näher mit dem WinCV-Tool beschäftigen – nicht nur, um Übung im Lesen der .NET-Klassen zu erhalten, sondern auch um die genaue Funktionsweise der Klassen zu ermitteln.

```
public class System.IO.FileSystemEventArgs :
    EventArgs
{

    // Fields

    // Constructors
    public FileSystemEventArgs(
        System.IO.WatcherChangeTypes changeType,
        string directory, string name);
```

```
    // Properties
    public WatcherChangeTypes ChangeType { get; }
    public string FullPath { get; }
    public string Name { get; }

    // Methods
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
} // end of System.IO.FileSystemEventArgs
```

Wenn das *m_Watcher*-Objekt eines der drei Ereignisse *Changed*, *Created* oder *Deleted* auslöst, wird der *OnChanged*-Ereignishandler aufgerufen. Wir können ermitteln, welches dieser Ereignisse eingetreten ist und ein entsprechendes Zeichenfolgenliteral in die lokale Zeichenfolgenvariable *sChange* schreiben.

```
Private Sub OnChanged(ByVal source As Object,  _
    ByVal e As FileSystemEventArgs)

    Dim sChange As String

    Select Case e.ChangeType
        Case WatcherChangeTypes.Changed : _
            sChange = "Changed"
        Case WatcherChangeTypes.Created : _
            sChange = "Created"
        Case WatcherChangeTypes.Deleted : _
            sChange = "Deleted"
    End Select
```

Nun wissen wir, welches Ereignis ausgelöst wurde. Die Protokollierung der Ereignisse in der Textdatei *observer.txt* löst wiederum ein Ereignis aus. Diese Änderungen sollen natürlich nicht aufgezeichnet werden.

```
If (Len(sChange) > 0) Then
    If (e.FullPath.IndexOf("observer.txt") > 0) _
        Then

        Exit Sub
    End If
End If
```

Mithilfe der Routine *writeToFile* können die Änderungen in die Datei geschrieben werden. Indem die *FullPath*-Eigenschaft der Datei sowie die Art der Änderung übergeben wird, lässt sich genau ermitteln, welches Ereignis eingetreten ist.

```
writeToFile("File: " & e.FullPath & "  " & sChange)
```

Verarbeiten der Ereignisse *Renamed* und *Error*

Diese Ereignishandler funktionieren genauso wie die im vorigen Abschnitt beschriebenen Handler. Prüfen Sie zu Übungszwecken mithilfe des WinCV-Tools die Klasse *RenamedEventArgs*. Sie können die Eigenschaften *FullPath*, *OldFullPath* und *OldName* anzeigen und so den neuen Dateinamen ermitteln.

```
public class System.IO.RenamedEventArgs :
    System.IO.FileSystemEventArgs
{
```

```
    // Fields

    // Constructors
    public RenamedEventArgs(
        System.IO.WatcherChangeTypes changeType,
        string directory, string name, string oldName);

    // Properties
    public WatcherChangeTypes ChangeType { get; }
    public string FullPath { get; }
    public string Name { get; }
    public string OldFullPath { get; }
    public string OldName { get; }

    // Methods
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public Type GetType();
    public virtual string ToString();
} // end of System.IO.RenamedEventArgs
```

Durch eine Abfrage des *RenamedEventArgs*-Parameters erfahren wir alle Details, die wir zu der umbenannten Datei benötigen. Ebenso kann anhand von *ErrorEventArgs* ermittelt werden, um welchen Fehlertyp es sich handelt. Beide Ereignishandler erzeugen eine Zeichenfolge, die zur Protokollierung der Umbenennungs- und Fehlerereignisse an die *writeToFile*-Routine übergeben wird.

```
Private Sub OnRenamed(ByVal source As Object, _
    ByVal e As RenamedEventArgs)

    writeToFile("File: " & e.OldFullPath &  _
        " renamed to " & e.FullPath)
End Sub

Private Sub onError(ByVal source As Object, _
    ByVal errevent As ErrorEventArgs)

    writeToFile("ERROR: " & errevent.GetException.Message())
End Sub
```

Schreiben in die Protokolldatei

Es kann sich als nützlich erweisen, Änderungen mit einem Zeitstempel zu versehen. Wir erstellen also ein Zeichenfolgenobjekt und rufen das aktuelle Datum und die Uhrzeit ab. Der *Try…Catch*-Block wurde in ▸Kapitel 7, »Fehlerbehandlung und Debugging«, bereits beschrieben und sollte Ihnen vertraut sein. Da beim Schreiben in Dateien unter Umständen Probleme auftreten können, wird der Dateizugriffscode in den geschützten *Try*-Block platziert. *Catch* ist leer und dient dazu, eventuelle Fehler abzufangen, damit das Programm nicht abstürzt, wenn der Schreibvorgang fehlschlägt. Anschließend verwenden wir die *WriteLine*-Methode des *StreamWriter*-Objekts, das in der privaten Membervariable *m_ObserveFileWrite* enthalten ist. Durch Aufrufen der *Flush*-Methode kann sichergestellt werden, dass der protokollierte Eintrag sofort auf Festplatte gespeichert wird.

```
Private Sub writeToFile(ByRef observeString As String)
    Dim sRightNow As String = _
        Date.Now.ToLongDateString() & _
        " " & Date.Now.ToLongTimeString()
```

```
            Try
                m_ObserveFileWrite.WriteLine(sRightNow & _
                    " " & observeString)
                m_ObserveFileWrite.Flush()
            Catch
            End Try

End Sub
```

Nach Freigabe des Objekts wird die *dispose*-Methode der Klasse aufgerufen. Die Ereignisaufzeichnung wird beendet, indem wir der *EnableRaisingEvents*-Eigenschaft den Wert *False* zuweisen, das Objekt auf *Nothing* setzen und die Datei schließen.

ANMERKUNG: Denken Sie daran, dass *Nothing* das Objekt nur zum Löschen kennzeichnet, Speicher und Ressourcen aber nicht wie in Visual Basic 6 sofort freigegeben werden. Die endgültige Freigabe erfolgt erst beim nächsten Durchlauf des Garbage Collectors. Der Garbage Collector stellt fest, dass das Objekt zum Löschen markiert ist und entfernt es, allerdings eventuell erst einige Minuten nach der Markierung.

```
Public Sub dispose()
    m_Watcher.EnableRaisingEvents = False
    m_Watcher = Nothing
    m_ObserveFileWrite.Close()
End Sub
```

Und schon ist die Klasse zur Überwachung wichtiger Ereignisse in Dateien und Verzeichnissen fertig. Nun geht es daran, die Klasse mit unserer Benutzerschnittstelle zu verknüpfen.

Einbinden der Benutzerschnittstelle

Haben Sie Lust, ein bisschen Code zu schreiben? Wechseln Sie in der Visual Basic .NET-IDE auf die Registerkarte *Form1.vb*, damit Sie den Code schreiben können, den wir für das Dateiüberwachungsprogramm benötigen. Wie üblich muss für eine solche Anwendung kein besonders großes Programm geschrieben werden. Dass wir auch ohne Unmengen von Code eine umfangreiche Funktionalität erhalten, verdeutlicht erneut die Leistungsstärke des .NET Frameworks. Fügen Sie auf *Form1.vb* folgenden Code ein:

```
Imports FileSentinel.SystemObserver.sentinel

Public Class Form1
    Inherits System.Windows.Forms.Form

    Private m_sFilesToScan As String
    Private m_fsSentinel As _
        FileSentinel.SystemObserver.sentinel

    Public Sub New()
        MyBase.New()

        ' This call is required by the Windows Form Designer.
        InitializeComponent()

        ' Add any initialization after the
        ' InitializeComponent() call

        lblWatching Text = DriveListBox1.Drive.ToUpper & "\"
```

```
    '-- Aufruf der privaten Routine zur Initialisierung der GUI
    InitializeGUI()

End Sub

Private Sub InitializeGUI()

'-- Deaktivieren der Optionen bis zur Auswahl von
' gültigen Verzeichnissen/Dateien
    btnEnable.Enabled = True
    btnDisable.Enabled = False

    ttTip.SetToolTip(btnEnable, _
        "Enable the File Sentinel to monitor a folder " & _
        "or directory. ")
    ttTip.SetToolTip(btnDisable, _
        "Stop monitoring a folder or directory.")
    ttTip.SetToolTip(DriveListBox1, _
        "Select the drive to monitor a folder or " & _
        "directory.")
End Sub

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

    m_fsSentinel.dispose()
    btnEnable.Enabled = True
    btnDisable.Enabled = False
    DriveListBox1.Enabled = True
    DirListBox1.Enabled = True
    FileListBox1.Enabled = True
End Sub

Private Sub DriveListBox1_SelectedIndexChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles DriveListBox1.SelectedIndexChanged

    Try
        DirListBox1.Path = DriveListBox1.Drive
        lblWatching.Text = DriveListBox1.Drive
    Catch
        DriveListBox1.Drive = DirListBox1.Path
    End Try

End Sub

Private Sub DirListBox1_SelectedIndexChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles DirListBox1.SelectedIndexChanged
```

```
        Try
            FileListBox1.Path = DirListBox1.Path
            lblWatching.Text = DirListBox1.Path
        Catch
        End Try
    End Sub

    Private Sub FileListBox1_SelectedIndexChanged( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles FileListBox1.SelectedIndexChanged

        If FileListBox1.Path.EndsWith("\") Then
            lblWatching.Text = FileListBox1.Path & _
                FileListBox1.FileName
        Else
            lblWatching.Text = FileListBox1.Path & _
                "\" & FileListBox1.FileName
        End If

    End Sub

    Protected Sub startWatching()

        '-- Initialisieren der QuickInfos
        '-- Erstellen einer neuen Instanz von File Sentinel
        m_fsSentinel = _
            New FileSentinel.SystemObserver.sentinel( _
                m_sFilesToScan)
        '-- Aktualisieren der Benutzerschnittstelle --
        btnEnable.Enabled = False
        btnDisable.Enabled = True
        DriveListBox1.Enabled = False
        DirListBox1.Enabled = False
        FileListBox1.Enabled = False
    End Sub

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

        startWatching()
    End Sub

    Private Sub lblWatching_TextChanged( _
        ByVal sender As Object, _
        ByVal e As System.EventArgs) _
        Handles lblWatching.TextChanged

        m_sFilesToScan = lblWatching.Text
    End Sub

    ' Other Windows Form Designer generated code omitted.

End Class
```

So funktioniert der Code für die Schnittstelle

Zunächst muss die *SystemObserver*-Klasse importiert werden. Anschließend können wir das Objekt in der Benutzerschnittstelle referenzieren.

```
Imports FileSentinel.SystemObserver.sentinel
```

Ganz am Anfang der *Form1*-Klasse fügen wir zwei private Variablen ein. Die erste, *m_sFilesToScan*, enthält die zu überwachende Datei bzw. das zu überwachde Verzeichnis. Die zweite Variable, *m_fsSentinel*, enthält natürlich einen Verweis auf eine Instanz der *sentinel*-Klasse.

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    Private m_sFilesToScan As String
    Private m_fsSentinel As _
        File_Sentinel.SystemObserver.sentinel
```

Im Formularkonstruktor soll ein Wert initialisiert werden, der anschließend im Label *lblWatching* angezeigt wird. Wenn der Benutzer unmittelbar nach dem Start des Programms auf die Schaltfläche *Enable Sentinel* klickt, wird zumindest ein Standardwert verarbeitet, und das Programm stürzt nicht ab. Anschließend rufen wir die vordefinierte *InitializeGUI*-Routine auf, die die Benutzerschnittstelle implementiert. Beachten Sie, dass wir das Label erst nach dem Aufruf der *InitializeComponent*-Routine setzen. So wird sichergestellt, dass sich alle Steuerelemente am richtigen Platz befinden und das Formular vollständig aufgebaut und angezeigt wird. Nun können wir in das Formular oder ein beliebiges Steuerelement schreiben, ohne dass eine Fehlermeldung erzeugt wird. Stellen Sie sicher, dass jeglicher Code, der ein sichtbares Formularelement bearbeitet, erst nach der *InitializeComponent*-Routine ausgeführt wird.

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

    ' This call is required by the Windows Form Designer.
    InitializeComponent()

    ' Add any initialization after the
    ' InitializeComponent() call
    lblWatching.Text = DriveListBox1.Drive.ToUpper & "\"

    '-- Aufruf der privaten Routine zur Initialisierung der GUI
    InitializeGUI()

End Sub
```

Das QuickInfo-Steuerelement

Nachdem das Formular geladen, aufgebaut und angezeigt wurde, wird die *InitializeGUI*-Routine aufgerufen. Mit dieser Routine aktivieren wir die *Enable Sentinel*-Schaltfläche und erzeugen die QuickInfos. Wir fügen den Schaltflächen *Enable Sentinel* und *Disable Sentinel* sowie dem Steuerelement *DriveListBox* jeweils ein QuickInfo-Steuerelement hinzu. QuickInfos lassen sich mit folgender Syntax problemlos einrichten:

```
ToolTipControl.SetToolTip(controlToAssociate, _
    "Message to display")
```

Sie können diese QuickInfos unterschiedlich gestalten, indem Sie ein paar Eigenschaften des Steuerelements festlegen. Beispielsweise können Sie verschiedene Werte für die Anzeigeverzögerung einrichten. Diese Eigenschaften werden in Millisekunden angegeben. Die Eigenschaft *InitialDelay* bestimmt,

wie lange ein Benutzer auf das entsprechende Steuerelement zeigen muss, um die QuickInfo anzuzeigen. Die Eigenschaft *ReshowDelay* gibt an, wieviel Zeit (in Millisekunden) verstreichen soll, bevor die nächste QuickInfo angezeigt wird, wenn der Mauszeiger von einem QuickInfo-Steuerelement zum nächsten verschoben wird. Die Eigenschaft *AutoPopDelay* legt fest, wie lange QuickInfos angezeigt werden. Sie können diese Werte einzeln angeben oder global festlegen, indem Sie den Wert der Eigenschaft *AutomaticDelay* setzen. Die weiteren Verzögerungswerte werden dann automatisch in einem bestimmten Verhältnis zum *AutomaticDelay*-Wert festgelegt. (Wenn *AutomaticDelay* auf den Wert *N* gesetzt wird, erhält *InitialDelay* den Wert *N*, *ReshowDelay* den Wert *N*/5 und *AutoPopDelay* den Wert 5*N*.) Wir werden die Standardwerte verwenden, weil sie gut in unsere Anwendung passen.

```
Private Sub InitializeGUI()

    '-- Deaktivieren der Optionen bis zur Auswahl eines
    ' gültigen Verzeichnisses bzw. einer Datei
    btnEnable.Enabled = True
    btnDisable.Enabled = False

    '-- Initialisieren der QuickInfos
    ttTip.SetToolTip(btnEnable, _
        "Enable the File Sentinel to monitor a folder " & _
        "or directory. ")
    ttTip.SetToolTip(btnDisable, _
        "Stop monitoring a folder or directory.")
    ttTip.SetToolTip(DriveListBox1, _
        "Select the drive to monitor a folder " & _
        "or directory.")
    End Sub
```

Beim Öffnen des File Sentinel-Programms ist die Schaltfläche *Enable Sentinel* aktiviert, die Schaltfläche *Disable Sentinel* wird abgeblendet dargestellt. Damit der Benutzer eine Auswahl treffen kann, werden auch die Listenfelder für Laufwerk, Verzeichnis und Dateien aktiviert.

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

    m_fsSentinel.dispose()
    btnEnable.Enabled = True
    btnDisable.Enabled = False
    DriveListBox1.Enabled = True
    DirListBox1.Enabled = True
    FileListBox1.Enabled = True
End Sub
```

Der Benutzer soll aus allen lokalen oder Remotelaufwerken auswählen können, auf die der Computer zugreifen kann. In Visual Basic 6 wurde üblicherweise das *DriveListBox*-Steuerelement dazu verwendet, in einem Dialogfeld zum Öffnen oder Speichern von Dateien Laufwerke auszuwählen oder zu ändern. Leider gibt es in Visual Basic .NET kein entsprechendes Steuerelement. Wenn Sie ältere Visual Basic-Anwendungen auf Visual Basic .NET aktualisieren, werden vorhandene *DriveListBox*-Steuerelemente in ein *VB6.DriveListBox*-Steuerelement konvertiert, das in der Kompatibilitätsbibliothek bereitgestellt wird (*Microsoft.VisualBasic.Compatibility*). Genau dieses Steuerelement werden wir verwenden, da es in .NET konvertiert wurde und als verwalteter Code gilt. Nach dem Hinzufügen des Steuerelements wird dieses im Dialogfeld *Toolbox anpassen* auf der Registerkarte *.NET Framework-Komponenten* angezeigt.

Wenn der Benutzer ein Laufwerk auswählt, soll dies auch im Listenfeld der Verzeichnisse angezeigt werden. Falls jedoch beispielsweise Laufwerk *A:* gewählt wird und sich im Laufwerk keine Diskette befindet, wird ein Fehler ausgegeben. Durch Einfügen der folgenden Zeile in einen geschützten *Try*-Block können solche Fehler verarbeitet werden:

```
DirListBox1.Path = DriveListBox1.Drive
```

Wenn das gewählte Laufwerk gültig ist, wird das Verzeichnislistenfeld auf dieses Laufwerk gesetzt und im Label angezeigt. Sollte jedoch die Wahl eines ungültigen Laufwerks zu einem Fehler führen, wird der *Catch*-Block ausgeführt, und das Listenfeld wird auf das vorherige gültige Laufwerk zurückgesetzt. Durch dieses einfache Verfahren können wir Laufzeitfehler verhindern.

```
Private Sub DriveListBox1_SelectedIndexChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles DriveListBox1.SelectedIndexChanged

    Try
        DirListBox1.Path = DriveListBox1.Drive
        lblWatching.Text = DriveListBox1.Drive
    Catch
        DriveListBox1.Drive = DirListBox1.Path
    End Try
End Sub
```

Ist das gewählte Laufwerk verfügbar, wird der Pfad im Verzeichnislistenfeld auf den neuen Laufwerkbuchstaben gesetzt. Erzeugt diese Änderung keinen Fehler, wird das neue Verzeichnis im Label *lblWatching* angezeigt. Für das *DirListBox*-Steuerelement gilt das gleiche wie für *DriveListBox*: Es gibt in Visual Basic .NET keine entsprechende Version.

```
Private Sub DirListBox1_SelectedIndexChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles DirListBox1.SelectedIndexChanged

    Try
        FileListBox1.Path = DirListBox1.Path
        lblWatching.Text = DirListBox1.Path
    Catch
    End Try
End Sub
```

Sofern die Änderung des Verzeichnislistenfeldes keinen Fehler erzeugt, wird das Dateilistenfeld aktualisiert. Wie bereits erwähnt, muss der Zeichenfolge ein abschließender umgekehrter Schrägstrich hinzugefügt werden, wenn dieser nicht vom Benutzer eingegeben wird.

```
Private Sub FileListBox1_SelectedIndexChanged( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) _
    Handles FileListBox1.SelectedIndexChanged

    If FileListBox1.Path.EndsWith("\") Then
        lblWatching.Text = FileListBox1.Path & _
            FileListBox1.FileName
    Else
        lblWatching.Text = FileListBox1.Path & "\" & _
            FileListBox1.FileName
    End If

End Sub
```

Bei Auswahl eines gültigen Laufwerks, Verzeichnisses oder Ordners wird das Label *lblWatching* aktualisiert. Diese Aktualisierung löst das *TextChanged*-Ereignis des Labels aus. Wir setzen die private Klassenvariable *m_sFilesToScan* auf den Wert der *Text*-Eigenschaft des Labels.

```
Private Sub lblWatching_TextChanged( _
    ByVal sender As Object, _
    ByVal e As System.EventArgs) _
    Handles lblWatching.TextChanged

    m_sFilesToScan = lblWatching.Text
End Sub
```

Wenn der Benutzer die zu überwachende Datei bzw. das Verzeichnis ausgewählt hat, klickt er auf die Schaltfläche *Enable Sentinel*, die wir mit dem Namen *btnEnable* erstellt haben. Dieses Ereignis ruft die Prozedur *startWatching* auf.

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

    startWatching()
End Sub
```

Sobald das Programm aktiviert wurde, wird eine neue Instanz der *sentinel*-Klasse erstellt. Damit der Benutzer während der Überwachung nicht auf Schaltflächen oder Laufwerke klicken kann, werden die Schaltfläche *Enable Sentinel* sowie sämtliche Listenfelder deaktiviert. Wir unterstützen die Benutzer also bei der Auswahl der Optionen, die während der Ausführung des Programms zur Verfügung stehen.

```
Protected Sub startWatching()

    '-- Erstellen einer neuen Instanz von File Sentinel
    m_fsSentinel = _
        New File_Sentinel.SystemObserver.sentinel( _
            m_sFilesToScan)

    '-- Aktualisieren der Benutzerschnittstelle --
    btnEnable.Enabled = False
    btnDisable.Enabled = True
    DriveListBox1.Enabled = False
    DirListBox1.Enabled = False
    FileListBox1.Enabled = False
End Sub
```

Da wir den gesamten Code in der Klasse gekapselt haben – wie es sich bei einem guten Programmdesign gehört –, sind in der Benutzerschnittstelle nur die Methoden *dispose* und *GetType* zu sehen. Der Rest ist verborgen, wie Sie in Abbildung 9.7 sehen können.

Abbildung 9.7: *In der Schnittstelle sind nur die Methoden dispose und GetType zu sehen.*

Mögliche Erweiterungen des File Sentinel-Programms

Wenn das File Sentinel-Programm in einer Arbeitsumgebung verwendet werden soll, benötigen die Benutzer u.U. mehr Auswahlmöglichkeiten für die Überwachung. Um dies zu ermöglichen, fügen Sie drei *WriteOnly*-Eigenschaften hinzu: *WatchAttributes*, *WatchFileSize* und *WatchLastAccess*. Das Pro-

gramm überwacht diese Eigenschaften zwar standardmäßig, Sie können dem Benutzer aber diese Optionen zur Verfügung stellen, indem Sie sie der *sentinel*-Klasse hinzufügen. Darüber hinaus können Sie auch einzelne Standardfilter hinzufügen (wir haben in unserem Programm alle Filter eingebaut), beispielsweise *LastWrite*, *Security*, *CreationTime* und *DirectoryName*.

```
m_Watcher.NotifyFilter = NotifyFilters.FileName Or _
    NotifyFilters.LastWrite Or _
    NotifyFilters.Security Or _
    NotifyFilters.CreationTime Or _
    NotifyFilters.DirectoryName
```

Anschließend fügen Sie der Klasse drei benutzerdefinierte *WriteOnly*-Eigenschaften hinzu. Wenn ein Benutzer weitere Ereignisse wie z.B. Attribute, Dateigröße oder den letzten Dateizugriff überwachen möchte, werden diese Eigenschaften in der Klasse auf *True* gesetzt. Da es sich bei der *NotifyFilters*-Eigenschaft um eine Enumeration handelt, können Sie eine beliebiges Attribut hinzufügen, indem Sie den Aufzählungsdatentyp der *NotifyFilters*-Eigenschaft der Klasse hinzufügen.

Um verschiedene Änderungen zu verfolgen, können Sie die Werte der *NotifyFilter*-Aufzählung bitweise miteinander verknüpfen. Sie könnten beispielsweise Änderungen an der Größe einer Datei oder eines Ordners oder veränderte Sicherheitseinstellungen verfolgen. Sobald eine solche Änderung erfolgt, wird ein Ereignis ausgelöst. In Tabelle 9.3 werden die Member von *NotifyFilters* aufgeführt.

Membername	**Beschreibung**
Attributes	Die Attribute einer Datei oder eines Ordners
CreationTime	Datum und Uhrzeit der Datei- oder Ordnererstellung
DirectoryName	Der Name des Verzeichnisses
FileName	Der Name der Datei
LastAccess	Das Datum des letzten Datei- oder Ordnerzugriffs
LastWrite	Das Datum der letzten Schreiboperation für Datei oder Ordner
Security	Die Sicherheitseinstellungen einer Datei oder eines Ordners
Size	Die Datei- oder Ordnergröße

Tabelle 9.3: *NotifyFilters-Member*

Sie können der Klasse folgende drei benutzerdefinierte *WriteOnly*-Eigenschaften hinzufügen, um Veränderungen an Dateiattributen, -größe oder -zugriff zu verfolgen:

```
WriteOnly Property WatchAttributes() As Boolean
    Set(ByVal Value As Boolean)
        If (value = True) Then
            m_Watcher.NotifyFilter += _
                IO.NotifyFilters.Attributes
        End If
    End Set
End Property

WriteOnly Property WatchFileSize() As Boolean
    Set
        If (value = True) Then
            m_Watcher.NotifyFilter += _
                IO.NotifyFilters.Size
```

```
    End Set
End Property

WriteOnly Property WatchLastAccess() As Boolean
    Set
        If (value = True) Then
            m_Watcher.NotifyFilter += _
                IO.NotifyFilters.LastAccess
        End If
    End Set
End Property
```

Anschließend fügen Sie der Benutzerschnittstelle drei Kontrollkästchen hinzu. Wenn ein Benutzer eine oder mehrere Optionen aktiviert, wird einfach die entsprechende Klasseneigenschaft auf *True* gesetzt.

```
If chkAttributes.Checked = True Then _
    m_fsSentinel.WatchAttributes = True
If chkSize.Checked = True Then _
    m_fsSentinel.WatchFileSize = True
If chkAccess.Checked = True Then _
    m_fsSentinel.WatchLastAccess = True
```

Damit ist unsere *File Sentinel*-Klasse auch schon fertig. Sie fragen sich vielleicht, warum wir keine weitere Funktionalität für die Benutzerschnittstelle (wie beispielsweise die Anzeige von Benachrichtigungen in Dialogfeldern) in der Klasse implementiert haben. Das liegt ganz einfach daran, dass wir unsere Klasse in einen Windows-Dienst konvertieren werden, und diese Dienste verfügen nun mal nicht über eine Benutzerschnittstelle.

ANMERKUNG: Die Legacysteuerelemente *DriveListBox*, *DirListBox* und *FileListBox* verhalten sich in der .NET-Plattform etwas unberechenbar – während der Ausführung des File Sentinel-Programms muss eventuell mehrfach darauf geklickt werden, damit die richtigen Elemente angezeigt werden.

Einführung in die Windows-Dienste

Mithilfe der Microsoft Windows-Dienste (die früheren NT-Dienste) können Sie langlebige Anwendungen erstellen, die in eigenen Windows-Sitzungen ausgeführt werden. Dienstanwendungen können so eingerichtet werden, dass sie beim Hochfahren des Computers starten, und sie können angehalten und neu gestartet werden.

Diese Dienste besitzen keine Benutzerschnittstelle, wodurch die Konvertierung unserer Klasse in einen Dienst zur Ausführung auf einem Server (oder an einer anderen Stelle, an der Sie eine stabile Anwendung benötigen, die anderen Benutzern auf demselben Computer nicht in die Quere kommt) problemlos zu bewältigen ist. Sie können Dienste im Sicherheitskontext eines anderen Benutzerkontos als des angemeldeten Benutzers oder Standardcomputers ausführen.

Um einen Dienst zu erstellen, schreiben Sie einfach eine Anwendung und installieren diese als Dienst. Wir können unsere *sentinel*-Klasse als Dienst zur Hintergrundausführung auf einem Server konvertieren. Da wir die Klasse nach dem Prinzip der objektorientierten Programmierung entworfen haben, können wir sie so verwenden, wie sie ist – und hierin liegt die wahre Bedeutung des Begriffs der Codewiederverwendbarkeit.

Der Lebenszyklus eines Dienstes

Ein Dienst durchläuft in seinem Lebenszyklus verschiedene Phasen. Zunächst wird er auf dem System installiert, auf dem er ausgeführt werden soll, beispielsweise auf einem Webserver. Dabei werden die Installationskomponenten für das Dienstprojekt ausgeführt und der Dienst in den Windows-Dienststeuerelement-Manager (Service Control Manager, SCM) dieses Computers geladen. Der SCM ist das zentrale Windows-Dienstprogramm zur Verwaltung von Diensten. Er kann zur automatischen Ausführung während des Computerstarts konfiguriert oder manuell gestartet und beendet werden. Ein Dienst wird solange ausgeführt, bis er beendet oder angehalten oder der Computer heruntergefahren wird. Folgende drei Zustände sind für einen Dienst möglich: ausgeführt, angehalten oder beendet.

Anders als bei anderen Arten von Projekten müssen für Dienste Komponenten für die Installation der Dienstanwendung erstellt werden, Diese Komponenten installieren und registrieren den Dienst auf dem Server und erstellen mithilfe des Dienststeuerelement-Managers einen Eintrag für den Dienst. Zum Glück ist diese Aufgabe in .NET problemlos zu bewältigen; ich werde die erforderlichen Schritte im Einzelnen erläutern.

Windows-Dienstanwendungen werden in einer anderen Windows-Station ausgeführt als andere Anwendungen (die Windows-Station ist ein sicheres Objekt, das eine Zwischenablage, einen Satz globaler Elemente und eine Gruppe von Desktopobjekten enthält). Aus diesem Grund werden Dialogfelder einer Windows-Dienstanwendung nicht angezeigt, was u.U. dazu führen kann, dass das Programm nicht mehr reagiert. Sie sollten also unbedingt darauf achten, dass alle Meldungen (auch Fehlermeldungen) in eine Textdatei geschrieben und nicht in Form von Meldungsfeldern auf der Benutzeroberfläche angezeigt werden.

So wird aus dem File Sentinel-Programm ein Windows-Dienst

Öffnen Sie ein neues Visual Basic-Projekt mit einer Vorlage für Windows-Dienste, wie in Abbildung 9.8 gezeigt, und geben Sie dem Projekt den Namen *vbFileMonitorService*. Im Textfeld *Speicherort* wird in dem aufgeführten Verzeichnis ein neues Verzeichnis mit diesem Namen angelegt.

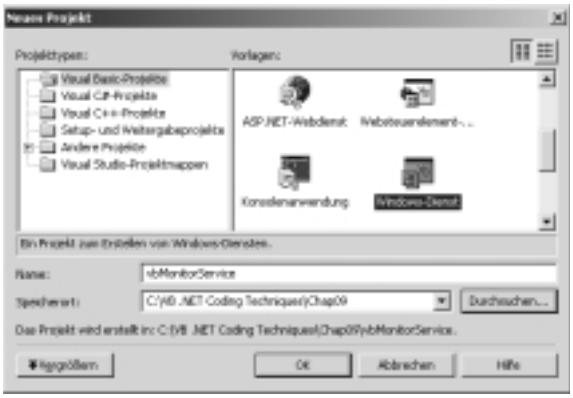

Abbildung 9.8: *Erstellen einer Windows-Dienstanwendung mithilfe des Projektsymbols Windows-Dienst*

Der Entwurfsbereich ist nach dem Erstellen des Projekts zunächst leer. Klicken Sie auf den Hyperlink *klicken Sie hier, um zur Codeansicht zu wechseln* (siehe Abbildung 9.9). Es wird automatisch eine Dienstvorlage geöffnet, der wir einige Zeilen hinzufügen müssen, um unsere *sentinel*-Klasse einzubauen und in eine Ereignisprotokolldatei schreiben zu können.

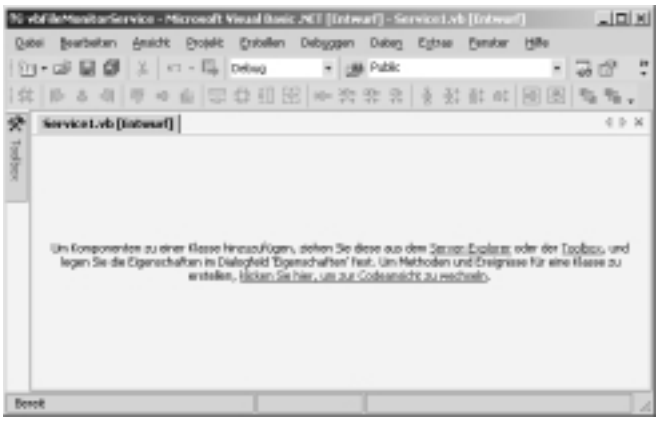

Abbildung 9.9: *Mithilfe des Hyperlinks zur Codeansicht wechseln*

Hinzufügen der *Sentinel*-Klasse zum Dienst

Wir werden nun diesem Projekt die im vorliegenden Kapitel erstellte *sentinel*-Klasse hinzufügen und somit die Wiederverwendbarkeit von objektorientierten Klassen demonstrieren. Wir werden die Klasse auf die leichtestmögliche Weise hinzufügen – indem wir im Windows Explorer die Datei *Sentinel.vb* aus dem Verzeichnis unseres letzten Projekts in das Verzeichnis *vbFileMonitorService* kopieren. Bei *Sentinel.vb* handelt es sich um die Textdatei, die den Code mit dem Namespace *SystemObserver* und der öffentlichen Klasse *sentinel* enthält.

Nach dem Kopieren der Datei in das Verzeichnis *vbFileMonitorService* wird sie dem aktuellen Projekt anhand folgender Schritte hinzugefügt:

1. Wählen Sie in der Visual Studio .Net-IDE den Befehl *Projekt* | *Vorhandenes Element hinzufügen*.
2. Wählen Sie im Dialogfeld *Vorhandenes Element hinzufügen* die Datei *Sentinel.vb*, um sie dem Projekt *vbFileMonitorService* hinzuzufügen. Nun befindet sich die Klassendatei im aktuellen Projektverzeichnis und wurde der Projektmappe hinzugefügt.
3. Klicken Sie auf die Registerkarte *Sentinel.vb*, um den Code für die *sentinel*-Klasse anzuzeigen. Kommentieren Sie die Zeile *EnableRaisingEvent* aus. Wir werden zur Aktivierung/Deaktivierung dieses Ereignisses zwei weitere Eigenschaften hinzufügen. Ändern Sie den Namen der Protokolldatei in *vbFMS.txt*, sodass Sie beide Dateien auf einem Datenträger speichern und zu unterschiedlichen Zwecken einsetzen können.

```
AddHandler m_Watcher.Changed, AddressOf OnChanged
AddHandler m_Watcher.Created, AddressOf OnChanged
AddHandler m_Watcher.Deleted, AddressOf OnChanged
AddHandler m_Watcher.Renamed, AddressOf OnRenamed
AddHandler m_Watcher.Error, AddressOf onError

' m_Watcher.EnableRaisingEvents = True

m_ObserveFileWrite = _
    New StreamWriter("C:\vbFMS.txt", True)
```

Sie erinnern sich: der Ereignishandler *OnChanged* prüft die Protokolldatei auf Änderungen. Wenn dies der Fall ist, ignorieren wir das Ereignis. Fügen Sie der Eigenschaft *IndexOf* den Namen der neuen Protokolldatei in Kleinbuchstaben hinzu: *vbfms.txt*. Dadurch wird verhindert, dass jeder einzelne Schreibvorgang in dieser Datei im Protokoll aufgezeichnet wird.

```
If (Len(sChange) > 0) Then
    If (e.FullPath.IndexOf("vbfms.txt") > 0) Then
        Exit Sub
    End If
End If
```

Abschließend fügen Sie der Klasse die folgenden zwei öffentlichen Eigenschaften hinzu. Mithilfe dieser Eigenschaften wird die *sentinel*-Klasse so konfiguriert, dass sie mit der Protokollierung von Dateiänderungen beginnt, sobald der Dienst gestartet wird und bei dessen Beendigung die Überwachung beendet.

```
Public Sub StartLogging()
    m_Watcher.EnableRaisingEvents = True
End Sub

Public Sub StopLogging()
    m_Watcher.EnableRaisingEvents = False
End Sub
```

Nachdem wir diese kleinen Änderungen eingefügt haben, können wir die *sentinel*-Klasse importieren und so problemlos eine Windows-Anwendung in einen Windows-Dienst umwandeln.

Aktualisieren der Datei *Service1.vb*

Die Visual Studio .Net-Vorlage *Windows-Dienst* erledigt, ebenso wie andere Projektvorlagen, den größten Teil der Arbeit für uns. Sie referenziert die entsprechenden Klassen und Namespaces, richtet die Vererbung von der Basisklasse für Dienste ein und überschreibt die Methoden, die überschrieben werden sollen.

Nun müssen Sie nur noch folgende Schritte ausführen, um einen funktionstüchtigen Dienst einzurichten:

- Setzen Sie die Eigenschaft *ServiceName*.
- Erstellen Sie die erforderlichen Installationskomponenten für die Dienstanwendung.
- Überschreiben Sie die Methoden *OnStart* und *OnStop* mit eigenen, um das Verhalten des Dienstes benutzerdefiniert anzupassen.

Klicken Sie auf die Registerkarte *Service1.vb*, um zu dem Code für die Dienstvorlage zu wechseln, der von der IDE erzeugt wurde. Um Platz zu sparen, werden in folgendem Listing die Zeilen hervorgehoben, die Sie hinzufügen müssen. Da wir auf das File Sentinel-Programm verweisen, muss dieses dem Projekt zunächst hinzugefügt werden.

```
Imports System.ServiceProcess
Imports vbFileMonitorService.SystemObserver.sentinel


Public Class Service1
    Inherits System.ServiceProcess.ServiceBase

    Dim vbFMS As vbFileMonitorService.SystemObserver.sentinel

#Region " Component Designer generated code "

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

```
        ' This call is required by the Component Designer.
        InitializeComponent()

        ' Add any initialization after the
        ' InitializeComponent() call
        EventLog.EnableRaisingEvents = True
        Me.AutoLog = True
        Me.CanStop = True

        vbFMS = New _
            vbFileMonitorService.SystemObserver.sentinel("C:\")

    End Sub

    ' The main entry point for the process
    Shared Sub Main()
        Dim ServicesToRun() As _
            System.ServiceProcess.ServiceBase

        ' More than one NT Service may run within the same
        ' process.To add another service to this process,
        ' change the following line to create a second
        ' service object.For example,
        '
        '   ServicesToRun = New _
        '       System.ServiceProcess.ServiceBase ()
        '      {New Service1, New MySecondUserService}
        '

        ServicesToRun = New _
            System.ServiceProcess.ServiceBase () _
            {New Service1}

        System.ServiceProcess.ServiceBase.Run(ServicesToRun)
    End Sub

    ' Required by the Component Designer
    Private components As System.ComponentModel.Container

    ' NOTE: The following procedure is required by the
    ' Component Designer.
    ' It can be modified using the Component Designer.
    ' Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        components = New System.ComponentModel.Container()
        Me.ServiceName = "vbFileMonitorService"
    End Sub

#End Region

    Protected Overrides Sub OnStart(ByVal args() As String)
        ' Add code here to start your service. This method
        ' should set things in motion so your service can
        ' do its work.
        vbFMS.StartLogging()
```

```
        EventLog.WriteEntry("Started logging files.")
    End Sub

    Protected Overrides Sub OnStop()
        ' Add code here to perform any tear-down
        ' necessary to stop your service.
        vbFMS.StopLogging()
        EventLog.WriteEntry("Stopped logging files.")
    End Sub

End Class
```

So funktioniert der Dienst

Die *Main*-Methode der Dienstanwendung benötigt den Befehl *Run* zur Ausführung der Dienste im Projekt. Die Methode *Run* lädt den Dienst in den Dienststeuerelement-Manager auf dem entsprechenden Server. Da wir den Dienst mit der Projektvorlage *Windows-Dienst* erstellt haben, ist die *Run*-Methode bereits vorhanden. Denken Sie daran: Das Laden eines Dienstes ist nicht gleichbedeutend mit dem Start.

Zunächst wird die Klasse importiert, die die Dateiüberwachung enthält. Da wir die Klasse in das *vbFileMonitorService*-Projekt implementieren, fügen wir folgende *Imports*-Anweisung ein:

```
Imports vbFileMonitorService.SystemObserver.sentinel
```

Anschließend fügen wir auf Klassenebene eine Variable hinzu, die einen Verweis auf eine Instanz der *sentinel*-Klasse enthält.

```
Dim vbFMS As vbFileMonitorService.SystemObserver.sentinel
```

Protokollieren der Ereignisse in der Ereignisanzeige

Die Ereignisprotokollierung in Windows bietet ein zentrales Standardverfahren für Anwendungen zur Aufzeichnung wichtiger Hard- und Softwareereignisse. Wenn ein Fehler auftritt, muss der Systemadministrator herausfinden können, wodurch dieser Fehler enstanden ist. Für diesen Fall ist es ausgesprochen hilfreich, wenn wichtige Ereignisse wie Speicherüberlastung oder fehlgeschlagene Dateizugriffe von Anwendungen, dem Betriebssystem oder anderen Systemdiensten protokolliert werden. Mithilfe des Ereignisprotokolls kann der Systemadministrator Fehlerursache und -kontext ermitteln.

In Windows steht eine Standardbenutzerschnittstelle zur Anzeige der Ereignisprotokolle (siehe Abbildung 9.10) sowie eine Programmierschnittstelle zur Untersuchung der Protokolleinträge bereit. In Visual Basic 6 waren für einige Ereignisprotokolle begrenzte Schreibvorgänge möglich, die meisten Protokolle waren aber nur schwer zugänglich.

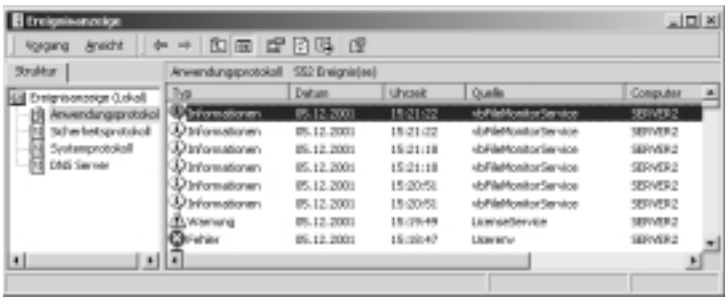

Abbildung 9.10: *Die Windows-Ereignisanzeige*

In .NET dagegen verwenden Sie einfach die *EventLog*-Komponente, um Ereignisprotokolle auf lokalen oder Remotecomputern anzuzeigen und Einträge zu protokollieren. Sie können auch Einträge vorhandener Protokolle einlesen sowie benutzerdefinierte Ereignisprotokolle erstellen. In unserer Klasse werden wir der Einfachheit halber in das Standardanwendungsprotokoll schreiben. Der Dienst soll alle Ereignisse im Anwendungsprotokoll aufzeichnen, die wir für wichtig halten. Standardmäßig lautet der Ereignistyp *Information*; Sie können aber auch einen anderen Typ angeben. Sie legen den Ereignistyp fest, indem Sie einen Parameter einer überladenen *WriteEntry*-Methode des *EventLog*-Objekts angeben.

Wenn Sie in der Ereignisanzeige auf eines der Ereignisse doppelklicken, wird ein Dialogfeld mit den Eigenschaften dieses Ereignisses geöffnet. In Abbildung 9.11 sehen Sie eine benutzerdefinierte Meldung, die im Anwendungsprotokoll aufgezeichnet wurde.

Abbildung 9.11: *Das Eigenschaftendialogfeld für ein Ereignis*

Die *EnableRaisingEvents*-Eigenschaft legt fest, ob das *EventLog*-Objekt ein Ereignis auslöst, wenn Einträge protokolliert werden. Wenn diese Eigenschaft auf *True* gesetzt ist, werden alle Komponenten, die das *EventWritten*-Ereignis empfangen, bei neuen Einträgen in das Protokoll benachrichtigt. Wird die *EnableRaisingEvents*-Eigenschaft auf *False* gesetzt, werden in diesen Fällen keine Ereignisse ausgelöst. In unserem Beispiel wird *EnableRaisingEvents* auf *True* gesetzt, das Ereignis wird jedoch nicht untersucht. Ich habe mich für diese einfache Variante entschieden, weil ich Ihnen das Konzept und die Syntax veranschaulichen möchte.

```
EventLog.EnableRaisingEvents = True
```

Sie sollten sehr umsichtig entscheiden, was in die Protokolldatei geschrieben werden soll, da sie ansonsten schnell überfüllt ist und wichtige Meldungen möglicherweise übersehen werden können. Beim Einsatz in Produktionsumgebungen sollten Ressourcenprobleme protokolliert werden. Wenn für die Anwendung beispielsweise zu wenig Speicherplatz zur Verfügung steht (aufgrund eines Programmfehlers oder weil der Speicher zu klein ist) und dadurch die Leistung beeinträchtigt wird, können Sie anhand des Protokolls mögliche Fehler ermitteln, wenn bei fehlgeschlagener Speicherplatzreservierung ein Ereignis protokolliert wird. Sie können auch Informationsereignisse protokollieren. Für eine serverbasierte Anwendung (z.B einen Datenbankserver) kann es von Nutzen sein, Ereignisse wie Benutzeranmeldungen, das Öffnen von Datenbanken oder die Übertragung von Dateien zu aufzuzeichnen. Der Server kann auch Fehlerereignisse (fehlgeschlagene Dateizugriffe, getrennte Hostver-

bindungen und Ähnliches) sowie Datenbankfehler oder erfolgreiche oder fehlgeschlagene Dateiübertragungen protokollieren.

Standardmäßig können alle Windows-Dienstprojekte durch die Protokollierung von Informationen und Ausnahmen mit dem Anwendungsereignisprotokoll interagieren. Mithilfe der *AutoLog*-Eigenschaft können Sie angeben, ob diese vordefinierte Funktionalität in Ihre Anwendung integriert werden soll. Die Protokollierung ist für alle Dienste, die Sie mithilfe der Projektvorlage für Windows-Dienste erstellen, standardmäßig aktiviert. In unserem Projekt werden wir eine statische Variante der *EventLog*-Klasse einsetzen, um Dienstinformationen in ein Protokoll zu schreiben. Es ist nicht erforderlich, eine Instanz der *EventLog*-Komponente zu erstellen oder manuell eine Quelle zu registrieren.

Wenn Sie das standardmäßige Anwendungsprotokoll nicht verwenden möchten, setzen Sie *AutoLog* auf *False*, erstellen im Code für den Dienst ein benutzerdefiniertes Ereignisprotokoll und registrieren den Dienst als gültige Quelle für Einträge in diesem Protokoll.

```
Me.AutoLog = True
```

In einigen Fällen ist es nicht erforderlich oder sogar hinderlich, Dateiänderungen zu protokollieren. Für Wartungszwecke oder bei normalem Dateizugriff kann der Dienst bei Bedarf einfach deaktiviert werden. Wenn für einen Dienst der Befehl *Stop* aufgerufen wird, prüft der Dienststeuerelement-Manager anhand des Wertes von *CanStop*, ob dieser Befehl für diesen Dienst gültig ist. Bei den meisten Diensten lautet der *CanStop*-Wert *True*, einige Betriebssystemdienste erlauben jedoch keine Dienstbeendigung durch einen Benutzer.

Wenn der *CanStop*-Wert *True* lautet, wird der Beendigungsbefehl an den Dienst übergeben und die *OnStop*-Methode aufgerufen. Dies ist auch bei unserem Dienst der Fall. Wenn keine *OnStop*-Methode definiert wurde, verarbeitet der Dienststeuerelement-Manager den Beendigungsbefehl über die leere Basisklassenmethode *ServiceBase.OnStop*.

```
Me.CanStop = True
```

Die nächste Zeile sollte Ihnen vertraut vorkommen. Mit dieser Zeile instanziieren wir eine neue Instanz unserer *sentinel*-Klasse. Das Stammverzeichnis des Laufwerks *C:* wird als Parameter an den Konstruktor übergeben. Da die Windows-Dienste keine Benutzerschnittstelle besitzen, muss der Wert manuell übergeben werden, er kann nicht in einer Benutzerschnittstelle erstellt werden. Es empfiehlt sich, in diesem Parameter beispielsweise das Stammverzeichnis für Ihre Webseiten oder einen anderen wichtigen Speicherort anzugeben. Im Moment möchten wir aber alle Dateien im Stammverzeichnis überwachen.

```
vbFMS = New _
    vbFileMonitorService.SystemObserver.sentinel("C:\")
```

In der nächsten Zeile nach der *InitializeComponent*-Prozedur ändern wir den Namen des Dienstes in einen für unser Programm aussagekräftigeren Namen.

```
Me.ServiceName = "vbFileMonitorService"
```

Dann fügen wir unserem File Sentinel-Programm zwei einzeilige Methoden hinzu. Diese Methoden, *StartLogging* und *StopLogging*, aktivieren bzw. deaktivieren die *EnableRaisingEvents*-Eigenschaft des *FileSystemMonitor*-Objekts. Beim Starten des Dienstes wird die Eigenschaft auf *True*, bei Beendigung auf *False* gesetzt.

Bei jedem Aufruf der Klasse zur Aktivierung bzw. Deaktivierung der Ereignisse wird ein Eintrag im Ereignisprotokoll erstellt, um verfolgen zu können, wann der Dienst gestartet und beendet wird. Dazu verwenden wir die *WriteEntry*-Methode der statischen *EventLog*-Klasse. Diese versieht den Eintrag automatisch mit einem Zeitstempel.

```
Protected Overrides Sub OnStart(ByVal args() As String)
    ' Add code here to start your service. This method
    ' should set things in motion so your service can
```

```
    ' do its work.
    vbFMS.StartLogging()
    EventLog.WriteEntry("Started logging files.")
End Sub

Protected Overrides Sub OnStop()
    ' Add code here to perform any tear-down
    ' necessary to stop your service
    vbFMS.StopLogging()
    EventLog.WriteEntry("Stopped logging files.")
End Sub
```

Die *Main*-Methode für die Dienstanwendung *vbFileSystemMonitor* löst den Befehl *Run* zur Ausführung der Dienste im Projekt aus. Auch diese Methode wird von der Windows-Dienstprojektvorlage bereitgestellt. Sobald der Dienst geladen ist, kann er mithilfe des Dienststeuerelement-Managers gestartet und beendet werden.

```
Shared Sub Main()
    Dim ServicesToRun() As System.ServiceProcess.ServiceBase

    ServicesToRun = New _
        System.ServiceProcess.ServiceBase () _
            {New Service1}
    System.ServiceProcess.ServiceBase.Run(ServicesToRun)
End Sub
```

Hinzufügen eines Installers zum Windows-Dienst

Für einen Windows-Dienst muss eine Installationskomponente implementiert werden, was zur Installation eines normalen Windows-Programms nicht erforderlich ist. Diese Komponente nimmt uns den Großteil der Arbeit ab, die zur Registrierung des Dienstes mit dem Dienststeuerelement-Manager erforderlich ist. Die *ServiceInstaller*-Klasse erledigt spezifische Aufgaben für den Dienst, mit dem sie verknüpft ist. Diese Klasse wird von dem Installationsdienstprogramm verwendet, das dem Dienst hinzugefügt wird, um Registrierungswerte für den Dienst in einen Unterschlüssel des Registrierungschlüssels *HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services* zu schreiben.

Der Dienst wird in diesem Unterschlüssel durch den Namen *ServiceName* identifiziert. Der Unterschlüssel enthält ebenfalls den Namen der ausführbaren Datei oder der DLL, zu der der Dienst gehört. In der *Regedit.exe* können Sie sich nach der Installation den Registrierungseintrag des Dienstes anschauen. Die Registrierung enthält im Schlüssel *ImagePath* den vollständigen Namen des Dienstes (siehe Abbildung 9.12).

Abbildung 9.12: *Der registrierte Dienst im Registrierungs-Editor*

Hinzufügen der Installationskomponente *ServiceInstaller*

Das Einfügen einer *ServiceInstaller*-Komponente in unseren Windows-Dienst lässt sich problemlos bewältigen. Wechseln Sie auf die Registerkarte *Service1.vb [Entwurf]*, und klicken Sie mit der rechten Maustaste darauf. Klicken Sie im Popupmenü auf die Option *Installer hinzufügen* (siehe Abbildung 9.13).

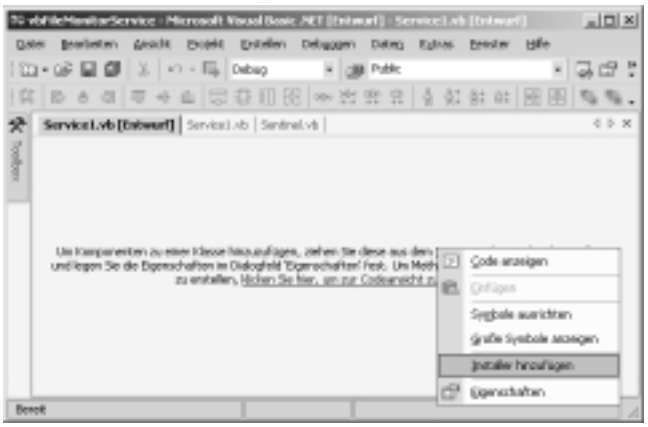

Abbildung 9.13: *Hinzufügen der Installationskomponente ServiceInstaller*

Wie Sie in Abbildung 9.14 sehen können, werden zwei Komponenten hinzugefügt: *ServiceProcessInstaller* und *ServiceInstaller*. Die IDE fügt bereits den größten Teil des benötigten Codes ein. Das Installationsprogramm *InstallUtil.exe* liest diesen Code und installiert *vbService* als Windows-Dienst.

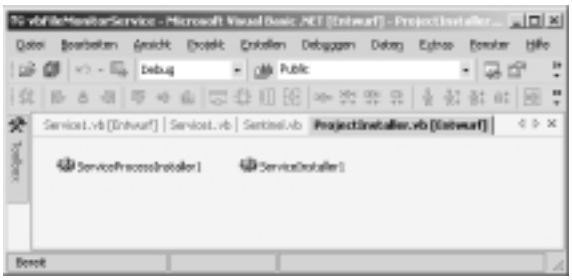

Abbildung 9.14: *Die IDE fügt einen ServiceProcessInstaller und einen ServiceInstaller hinzu*

Doppelklicken Sie in der IDE auf der Registerkarte *ProjectInstaller.vb [Entwurf]* nacheinander auf die Symbole *ServiceProcessInstaller1* und *ServiceInstaller1*. In der Vorlage ist bereits der Löwenanteil des erforderlichen Codes vorhanden, wir müssen nur noch ein paar Zeilen hinzufügen, damit der Installer unser Programm ordnungsgemäß installieren kann. Auch hier sind aus Platzgründen nur die Zeilen hervorgehoben, die Sie einfügen müssen.

```
Imports System.ComponentModel
Imports System.Configuration.Install

<RunInstaller(True)> Public Class ProjectInstaller
    Inherits System.Configuration.Install.Installer

#Region " Component Designer generated code "

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

```
        ' This call is required by the Component Designer.
        InitializeComponent()

        ' Add any initialization after the
        ' InitializeComponent() call

    End Sub
    Friend WithEvents ServiceProcessInstaller1 As _
        System.ServiceProcess.ServiceProcessInstaller
    Friend WithEvents ServiceInstaller1 As _
        System.ServiceProcess.ServiceInstaller

    ' Required by the Component Designer
    Private components As System.ComponentModel.Container

    ' NOTE: The following procedure is required by the
    ' Component Designer
    ' It can be modified using the Component Designer.
    ' Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        Me.ServiceProcessInstaller1 = New _
            System.ServiceProcess.ServiceProcessInstaller()
        Me.ServiceInstaller1 = New _
            System.ServiceProcess.ServiceInstaller()
        '
        ' ServiceProcessInstaller1
        '
        Me.ServiceProcessInstaller1.Account = _
            System.ServiceProcess.ServiceAccount.LocalSystem
        Me.ServiceProcessInstaller1.Password = Nothing
        Me.ServiceProcessInstaller1.Username = Nothing
        '
        ' ServiceInstaller1
        '
        Me.ServiceInstaller1.ServiceName = _
            "vbFileMonitorService"
        '
        ' ProjectInstaller
        '
        Me.Installers.AddRange(New _
             System.Configuration.Install.Installer() _
             {Me.ServiceProcessInstaller1, Me.ServiceInstaller1})

    End Sub

#End Region

    Private Sub ServiceInstaller1_AfterInstall( _
        ByVal sender As System.Object, _
        ByVal e As _
            System.Configuration.Install.InstallEventArgs) _
        Handles ServiceInstaller1.AfterInstall

    End Sub
```

```
    Private Sub ServiceProcessInstaller1_AfterInstall( _
        ByVal sender As System.Object, _
        ByVal e As _
            System.Configuration.Install.InstallEventArgs)  _
        Handles ServiceProcessInstaller1.AfterInstall

    End Sub
End Class
```

So funktioniert der Installationscode

Dieser Code teilt der *ServiceProcessInstaller*-Komponente mit, unter welchem lokalen Konto bzw. Prozessraum der Dienst ausgeführt werden soll. Standardmäßig wird der Dienst manuell gestartet. Sie können ihn aber auch so konfigurieren, dass er beim Hochfahren des Computers automatisch gestartet wird. Diese Einstellung können Sie auch nach der Installation noch ändern. Wir werden den Dienst für den manuellen Aufruf einrichten.

```
Me.ServiceProcessInstaller1.Account = _
    System.ServiceProcess.ServiceAccount.LocalSystem
```

Wie gehabt erhält der Dienst den Namen, der im Dienstfenster angezeigt werden soll.

```
Me.ServiceInstaller1.ServiceName = _
    "vbFileMonitorService"
```

Wenn die Protokollierung aktiviert ist, registriert der Installer den Dienst im Anwendungsprotokoll des Computers, auf dem der Dienst installiert wird, als gültige Quelle für Ereignisse. Jedes Mal, wenn der Dienst gestartet, beendet, angehalten, fortgesetzt, installiert oder deinstalliert wird, erfolgt ein Eintrag in das Protokoll. Eventuell auftretende Fehler werden ebenfalls protokolliert. Sie müssen für das Standardverhalten des Dienstes keinerlei Code zur Ereignisprotokollierung schreiben. Diese ganze Arbeit nimmt der Dienst Ihnen ab. Ziemlich cool, oder?

Installieren des Dienstes

So, jetzt können wir unseren Dienst installieren. Wählen Sie in der IDE den Befehl *Erstellen* | *Erstellen*. Dadurch wird die Datei *vbFileMonitorService* im Verzeichnis *\bin* erstellt.

Wenn Sie die Anwendung erstellt haben, installieren Sie den Dienst mithilfe des Befehlszeilendienstprogramms *InstallUtil.exe* und übergeben den Pfad an die *.exe*-Datei des Dienstes. Das geht am einfachsten, wenn Sie im Windows-Explorer zur Datei *InstallUtil.exe* wechseln. Kopieren Sie diese Datei in das Verzeichnis, in dem sich die *.exe*- Datei Ihres Dienstes befindet. Auf meinem Rechner ist diese Datei unter C:\Chapter 9\vbFileMonitorService\bin gespeichert. Wenn Sie die Beispieldateien der beiliegenden CD in das Standardverzeichnis kopiert haben, lautet der Pfad *C:\Coding Techniques for Visual Basic .NET\Chap09\vbFileMonitorService\bin*.

Öffnen Sie eine MS-DOS-Eingabeaufforderung, führen Sie *installutil,* aus, und übergeben Sie den vollständigen Namen des Dienstes. In der Eingabeaufforderung werden verschiedene Meldungen zum Installationsvorgang angezeigt, die etwa so aussehen müssten:

```
C:\Chapter 9\vbFileMonitorService\bin>installutil
 vbfilemonitorservice.exe
Microsoft (R) .NET Framework Installation utility
Copyright (C) Microsoft Corp 2001. All rights reserved.

Running a transacted installation.

Beginning the Install phase of the installation.
See the contents of the log file for the C:\Chapter
 8\vbFileMonitorService\bin\vbfilemonitorservice.exe
```

```
 assembly's progress.
The file is located at C:\Chapter
 9\vbFileMonitorService\bin\vbfilemonitorservice.InstallLog.
Call Installing. on the C:\Chapter
 9\vbFileMonitorService\bin\vbfilemonitorservice.exe assembly.
Affected parameters are:
   assemblypath = C:\Chapter
 9\vbFileMonitorService\bin\vbfilemonitorservice.exe

   logfile = C:\Chapter
 9\vbFileMonitorService\bin\vbfilemonitorservice.InstallLog
Installing service vbFileMonitorService...
Service vbFileMonitorService has been successfully installed.
Creating EventLog source vbFileMonitorService in log
 Application...

The Install phase completed successfully, and the Commit
 phase is beginning.
See the contents of the log file for the C:\Chapter
 9\vbFileMonitorService\bin\vbfilemonitorservice.exe
 assembly's progress.
The file is located at C:\Chapter
 9\vbFileMonitorService\bin\vbfilemonitorservice.InstallLog.
Call Committing. on the C:\Chapter
 9\vbFileMonitorService\bin\vbfilemonitorservice.exe assembly.
Affected parameters are:
   assemblypath = C:\Chapter
 9\vbFileMonitorService\bin\vbfilemonitorservice.exe

   logfile = C:\Chapter
 9\vbFileMonitorService\bin\vbfilemonitorservice.InstallLog

The Commit phase completed successfully.

The transacted install has completed.
```

ANMERKUNG: Zur Deinstallation des Dienstes muss zunächst die Dienstverwaltungskonsole geschlossen werden. Wenn die Konsole geöffnet ist, müssen Sie sie schließen und erneut öffnen, um den Deinstallationsvorgang zu beenden. Dazu verwenden Sie die Syntax *installutil /u vbFileMonitorService.exe*.

Anzeigen des Dienstes *vbMonitorService* im Fenster *Dienste*

Nachdem wir den Dienst erfolgreich installiert haben, wollen wir ihn auch einsetzen.

1. Öffnen Sie das *Dienste*-Fenster, indem Sie auf *Start* | *Programme* | *Verwaltung* | *Dienste* klicken. (In Windows 2000 Professional öffnen Sie die *Systemsteuerung*, und doppelklicken Sie nacheinander auf *Verwaltung* und *Dienste*.) Im Dialogfeld *Dienste* (siehe Abbildung 9.15) wird der neue Dienst neben den bereits vorhandenen Systemdiensten angezeigt.

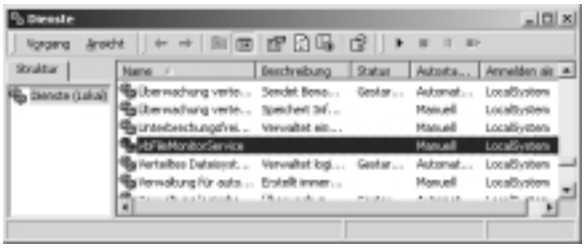

Abbildung 9.15: *Unser neuer Dienst im Fenster Dienste*

2. Doppelklicken Sie auf *vbMonitorService*, um das Eigenschaftendialogfeld anzuzeigen (siehe Abbildung 9.16), und geben Sie in das Textfeld *Beschreibung* den Namen *File Sentinel* ein. Dieser Name wird in der Spalte *Beschreibung* des Dienstfensters angezeigt. Beachten Sie, dass der Beschreibungstext unter Windows XP nicht geändert werden kann.

Abbildung 9.16: *Das Eigenschaftendialogfeld für vbFileMonitorService*

3. Klicken Sie auf *Übernehmen*, um dem Dienst diese Beschreibung hinzuzufügen.
4. Nun ist es soweit: unser neuer Dienst kann starten und seine Arbeit aufnehmen. Klicken Sie auf die Schaltfläche *Start*. Während der Dienst gestartet wird, wird für einige Sekunden ein Dialogfeld mit einer Fortschrittsanzeige geöffnet (siehe Abbildung 9.17).

Abbildung 9.17: *Die Fortschrittsanzeige während des Startens*

Das *Dienste*-Fenster zeigt nun auch an, dass der Dienst ausgeführt wird (siehe Abbildung 9.18). Solange der Dienst aktiv ist, werden alle Änderungen an Dateien im Stammverzeichnis von Laufwerk *C:* überwacht.

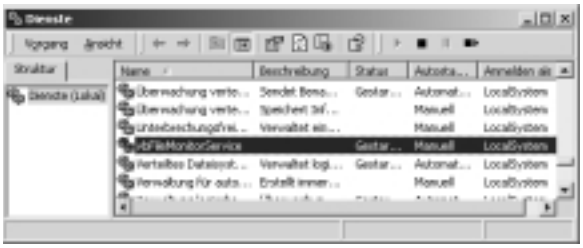

Abbildung 9.18: *Der Dienst wurde gestartet*

In der Textdatei *vbFMS.txt* können Sie sehen, ob verdächtige Dateizugriffe stattgefunden haben. Werfen wir einen Blick auf die Datei:

```
Monday, July 30, 2001 9:33:13 PM File: C:\passwords.exe
 renamed to C:\trash.exe
Monday, July 30, 2001 9:33:13 PM File: C:\trash.exe  Changed
Monday, July 30, 2001 9:33:18 PM File: C:\trash.exe  Changed
Monday, July 30, 2001 9:33:39 PM File: C:\trash.exe  Deleted
Monday, July 30, 2001 9:34:08 PM File: C:\salaries.xls  Changed
Monday, July 30, 2001 9:34:08 PM File: C:\salaries.xls  Changed
Monday, July 30, 2001 9:34:18 PM File: C:\passwords.txt  Changed
```

Zum Beenden des Dienstes doppelklicken Sie einfach im *Dienste*-Fenster auf *vbFileMonitorService* und klicken anschließend auf *Stop*. In einem Dialogfeld wird der Fortschritt des Vorgangs anzeigt (siehe Abbildung 9.19). Nun ist klar, warum wir der *sentinel*-Klasse die beiden Methoden *StartLogging* und *StopLogging* hinzugefügt haben. Mithilfe dieser Methoden ist das Starten und Beenden eines Dienstes überhaupt kein Problem.

Abbildung 9.19: *Der Fortschritt des Beendigungsvorgangs*

Debuggen eines Windows-Dienstes

Nachdem der Dienst geschrieben ist, muss die kompilierte *.exe*-Datei auf dem Zielcomputer installiert werden, um sinnvoll arbeiten zu können. Darüber hinaus kann eine Dienstanwendung in der Entwicklungsumgebung nicht durch Drücken der Tasten *F5* oder *F11* gedebuggt bzw. ausgeführt werden. Bedingt durch Funktionsweise und Aufgabe eines Dienstes kann er nicht direkt ausgeführt oder auf den Code zugegriffen werden.

Da ein Dienst im Kontext eines Dienststeuerelement-Managers ausgeführt werden muss, ist hierbei das Debuggen nicht ganz so einfach wie bei anderen Visual Studio .Net-Anwendungstypen. Um einen Dienst zu debuggen, müssen Sie ihn zuerst starten und anschließend dem Prozess, in dem der Dienst ausgeführt wird, einen Debugger hinzufügen. So stehen für das Debuggen des Dienstes die Standarddebuggingfunktionen der Visual Studio .Net-IDE zur Verfügung.

ANMERKUNG: Sie sollten einem Prozess erst dann einen Debugger hinzufügen, wenn Sie den Prozess genau kennen, denn der Prozess könnte möglicherweise zerstört werden. Wenn Sie dem Windows-Anmeldungsprozess einen Debugger hinzufügen und den Debuggingvorgang anschließend unterbrechen, wird das System angehalten, da es ohne Windows-Anmeldung nicht betriebsfähig ist.

Ein Debugger kann nur an einen ausgeführten Dienst angehängt werden. Erwartungsgemäß wird durch das Anhängen die Funktionsweise des Dienstes unterbrochen, nicht aber die Prozessausführung beendet oder angehalten – wenn der Dienst also zu Beginn des Debuggingvorgangs gestartet war, befindet er sich technisch gesehen noch immer im Modus *Gestartet*, arbeitet aber nicht mehr.

Durch das Anhängen eines Debuggers an den Prozess können Sie den größten Teil des Codes, jedoch nicht den gesamten Code debuggen. Da der Dienst schon gestartet war, kann der Code in der *OnStart*-Methode des Dienstes und der zum Laden erforderlichen *Main*-Methode nicht gedebuggt werden. Beide Prozeduren wurden bereits ausgeführt, um den Dienst zu laden.

Eine Möglichkeit, diese Beschränkung zu umgehen, besteht darin, innerhalb des Dienstes einen zweiten Dienst zu erstellen, der einzig dem Zweck dient, das vollständige Debuggen zu ermöglichen. Installieren Sie beide Dienste, und starten Sie zum Laden des Prozesses den Dummydienst. Nachdem der temporäre Dienst den Prozess gestartet hat, können Sie mithilfe des Menüs *Debugger* in Visual Studio .Net einen Debugger an den Prozess anhängen.

Anschließend können Sie Haltepunkte setzen und diese zum Debuggen des Codes einsetzen. Sobald das Dialogfeld geschlossen wird, in dem der Debugger angefügt wird, ist der Debugmodus aktiv. Im Dienststeuerelement-Manager können Sie den Dienst starten, beenden, anhalten oder fortsetzen und so mit den gesetzten Haltepunkten arbeiten. Entfernen Sie den Dummydienst nach dem erfolgreichen Debuggen.

Wenn Sie alles für das Debuggen vorbereitet haben, starten Sie den Dienst im Dienststeuerelement-Manager. Wenn der Dienst ausgeführt wird, können Sie mit dem Debuggen beginnen.

1. Wählen Sie in der IDE die Option *Debuggen* | *Prozesse*, um das Dialogfeld *Prozesse* anzuzeigen. Da es sich bei unserem Prozess um einen Systemprozess handelt, muss das Kontrollkästchen *Systemprozesse anzeigen* aktiviert werden, das standardmäßig deaktiviert ist. Natürlich besitzt unser Dienst keine Benutzerschnittstelle, also wird in der Spalte *Titel* auch kein Titel angezeigt (siehe Abbildung 9.20). -

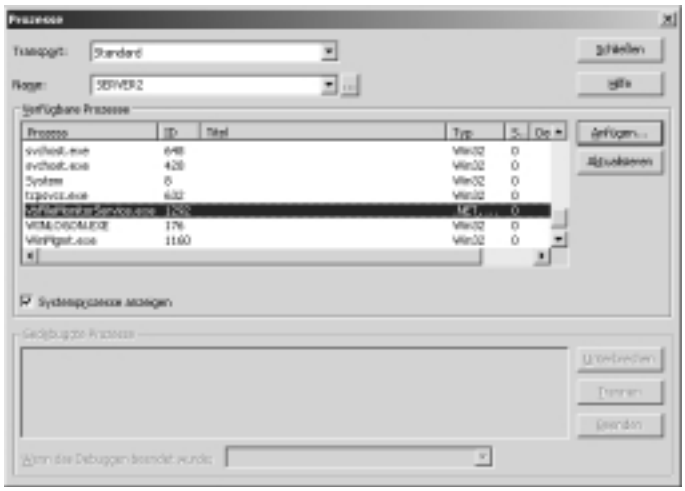

Abbildung 9.20: *Anhängen eines Debuggers an einen Prozess*

2. Doppelklicken Sie auf den Prozess *vbFileMonitorService*, um das Dialogfeld *An den Prozess anhängen* zu öffnen (siehe Abbildung 9.21), aktivieren Sie die Option *Common Language Runtime*, und klicken Sie anschließend auf *OK*, um den Prozess zu debuggen.

Abbildung 9.21: *Das Dialogfeld An den Prozess anhängen*

3. Wie Sie in Abbildung 9.22 sehen können, wurde dem *vbFileMonitorService* ein Debugger angehängt. Klicken Sie auf *Schließen*, um das Dialogfeld *Prozesse* zu schließen.

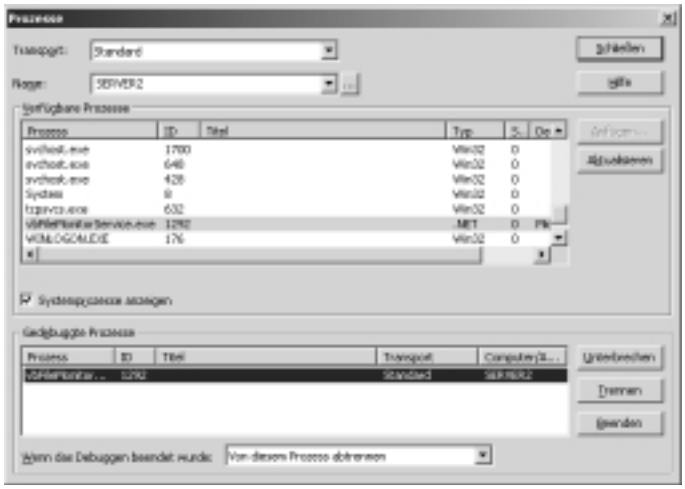

Abbildung 9.22: *Geschafft – dem Dienstprozess wurde ein Debugger angehängt*

Sie befinden sich nun im Debugmodus. Setzen Sie Haltepunkte, die im Code verwendet werden sollen. Im Dienststeuerelement-Manager können Sie den Dienst starten, beenden, anhalten oder fortsetzen und mit den gesetzten Haltepunkten arbeiten.

Fazit

Dieses Kapitel war ziemlich wichtig. Sie haben Einiges über das .NET Framework und das Erstellen von Klassen in Visual Basic .NET erfahren. Dabei haben Sie Delegates kennen gelernt und mithilfe von *StreamWriter*-Objekten eine Ereignisprotokollierung durchgeführt. Darüber hinaus wurde die Klasse in einen Windows-Dienst ohne Benutzerschnittstelle konvertiert.

In den ersten neun Kapiteln dieses Buches habe ich die Verfahren vorgestellt, die zur Arbeit in Visual Basic .NET und zur Verwendung der Frameworkklassen in beliebigen Namespaces erforderlich sind. Mit diesem Wissen ausgerüstet können wir uns nun mit ADO.NET beschäfigen und unsere Kenntnisse auch gleich bei den .NET-Methoden für den Datenbankzugriff einsetzen.