12 Beerdigen von Objekten – Dispose, Finalize und der Garbage Collector

357 Der Garbage Collector – die Müllabfuhr in .NET
359 Finalize
364 Dispose

Für das nächste Thema möchte ich noch einmal auf ein Beispiel zu sprechen kommen, das ursprünglich der Demonstration eines ganz anderen Themas diente. Sie erinnern sich noch an die Klasse `DynamicList` im Abschnitt zur Polymorphie, die die Beispielanwendung nutzte, um die Artikeldatensätze (`ShopItem`) zu speichern? Neben der eigentlichen Fähigkeit, eine Methode zu implementieren, die eine dynamische Vergrößerung des benötigten Speichers demonstriert, zeigt dieses Beispiel noch was anderes – was allerdings mehr eine Fähigkeit des .NET-Framework selbst ist: den nicht benötigten Speicher nämlich wieder freizugeben.

Rufen Sie sich die `Add`-Methode dieser Klasse noch mal in Erinnerung:

```
Sub Add(ByVal Item As ShopItem)

    'Prüfen, ob aktuelle Arraygrenze erreicht wurde
    If myCurrentCounter = myCurrentArraySize - 1 Then
        'Neues Array mit mehr Speicher anlegen,
        'und Elemente hinüberkopieren. Dazu:

        'Neues Array wird größer:
        myCurrentArraySize += myStepIncreaser

        'Temporäres Array erstellen.
        Dim locTempArray(myCurrentArraySize - 1) As ShopItem

        'Elemente kopieren
        'Wichtig: Um das Kopieren müssen Sie sich,
        'anders als bei VB6, selber kümmern!
        For locCount As Integer = 0 To myCurrentCounter
            locTempArray(locCount) = myArray(locCount)
        Next

        'Temporäres Array dem Memberarray zuweisen.
        myArray = locTempArray
    End If
```

```
        'Element im Array speichern.
        myArray(myCurrentCounter) = Item

        'Zeiger auf nächstes Element erhöhen.
        myCurrentCounter += 1

    End Sub
```

Schauen Sie sich diesen Codeblock noch einmal an, aber dieses Mal unter einem anderen Aspekt. Dieses Mal stehen nicht der Speicherplatz im Vordergrund, der benötigt wird, und die Art und Weise, wie Arrays wachsen können, sondern der Speicher, der durch denselben Vorgang überflüssig wird.

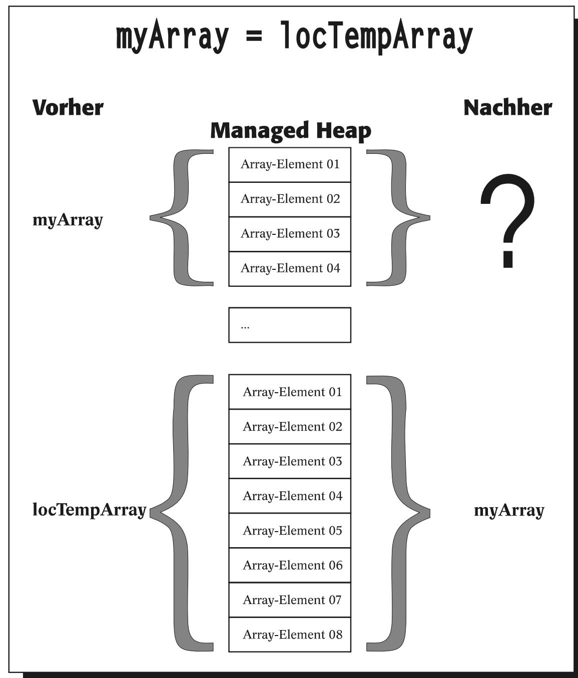

Abbildung 12.1: Was passiert mit den nach der Zuweisung im leeren Raum stehenden Array-Elementen?

Arrays in .NET sind keine Werte, sondern Verweistypen. Benötigter Speicher für alles, was von `System.Array` abgeleitet ist – und dazu zählen auch Arrays, die Sie durch `Dim` deklarieren – wird also auf dem Managed Heap reserviert. Im Codeauszug des Beispielprogramms gibt es zwei entscheidende Zeilen, die eigentlich ein »Speicherleck«, besser bekannt unter dem neudeutschen Begriff »Memory Leak«, verursachen würden, programmierten wir nicht im .NET-Framework. Abbildung 12.1 macht das Problem deutlich. Zunächst gibt es das Array und den auf dem Managed Heap dafür reservierten Speicherbereich, aber das Array ist nunmehr zu klein geworden. Also definiert die Prozedur ein neues Array und nimmt dafür die Variable `locTempArray` zu Hilfe. Nun passiert das Entscheidende: `locTempArray` wird `myArray` zugewiesen, der Zeiger auf den Speicherbereich der entsprechenden Elemente wird dabei quasi »verbogen«. `myArray` zeigt anschließend auf die Arrayelemente, auf die kurz zuvor noch `locTempArray` zeigte. Die Elemente, auf die von `myArray` verwiesen wurde, liegen nun unbrauchbar, da nicht mehr referenziert, irgendwo im Speicher – was passiert jetzt mit ihnen?

Erinnern wir uns, wie das bei COM geregelt war. Bei COM gab es für jedes Objekt einen Referenzzähler. Bei der ersten Zuweisung an eine Objektvariable wurde der Zähler auf Eins gesetzt. Mit jeder weiteren Zuweisung an eine Variable – also mit jeder weiteren Referenzierung – wurde dieser Zähler um eins erhöht. Nun trat der umgekehrte Fall ein: Einer Variablen, die zuvor das Objekt referenzierte, wurde ein anderes Objekt oder `Nothing` zugewiesen, oder das Programm verließ den Gültigkeitsbereich der Variablen, sodass sie aus diesem Grund das Objekt nicht mehr referenzieren konnte. In diesem Fall wurde der Referenzzähler um eins vermindert. Wurde er 0, dann konnte das Objekt entsorgt werden. Es wurde zu diesem Zeitpunkt nicht länger benötigt, da von keiner Stelle des Programms aus mehr referenziert – nur musste der Entwickler sich um die Entsorgung des Objektes selber kümmern.

Dieses Verfahren hatte allerdings zwei Nachteile: Zum einen kostete des Prinzip des Referenzzählers wertvolle Rechenzeit. Der andere Nachteil war das Problem der so genannten Zirkelverweise: Ein Objekt, das auf ein Objekt zeigte, was seinerseits wieder auf das Ausgangsobjekt zeigte, führte dazu, dass der Referenzzähler niemals null werden konnte. Selbst wenn es in diesem Fall keine Referenzierung mehr durch das eigentliche Programm gab, so referenzierten sich die Objekte dennoch selbst. Ein Speicherleck war oft genug die Folge.

Das .NET-Framework – oder, um genau zu sein – die Common Language Runtime, löst dieses Problem, indem sie ein komplett anderes Verfahren anwendet, Objekte zu entsorgen.

Der Garbage Collector – die Müllabfuhr in .NET

Den vorhandenen Speicher des Managed Heap teilen sich alle Assemblies, die in einer so genannten *Application Domain* (»Anwendungsdomäne«, kurz »AppDomain«) laufen. Normale Windows-Anwendungen, die man parallel startet, werden durch verschiedene, streng voneinander abgeschottete Prozesse isoliert. Ein Prozess kann unter normalen Umständen nicht auf einen anderen Prozess zugreifen, auch die Daten verschiedener Prozesse sind integer und können bestenfalls durch so genannte Proxys (Stellvertreter) untereinander ausgetauscht werden.

AppDomains in .NET, die durch die Common Language Runtime verwaltet werden, erlauben das gleichzeitige Ausführen mehrere Anwendungen in *einem* Prozess. Die CLR garantiert dabei, dass die Anwendungen ebenso isoliert und ungestört laufen können, wie das bei einem Windows-Prozess der Fall wäre. Jede AppDomain verwaltet einen Speicherbereich, in dem die notwendigen Daten der Assemblies und Programme,[1] die innerhalb der AppDomain laufen, abgelegt werden. Dieser Speicherbereich wird Managed Heap genannt, mit dem wir uns schon einige Male beschäftigt haben. Wenn der Speicher im Managed Heap (und auch sonst schon einmal aus anderen Gründen) knapp wird, dann startet ein Prozess, der im wahrsten Sinne des Wortes der Müllabfuhr im echten Leben entspricht: Die Garbage Collection findet statt.

Der Garbage Collector läuft in einem eigenen Thread,[2] der die Objekte des Managed Heap dahingehend untersucht, ob sie noch in irgendeiner Form referenziert werden. Objekte, denen eine noch

---

[1] Streng genommen ist auch ein Programm eine Assembly.

[2] Grob erklärt: Ein Thread ist ein Programmteil, der die Eigenschaft hat, neben anderen Programmteilen quasi gleichzeitig laufen zu können. Ein Anwendungsprogramm besteht mindestens aus einem Thread. Bei einem Windows-Programm wird der Thread, der die Benutzereingaben überwacht und als Ereignisse weiterleitet, übrigens *UI-Thread* genannt (»UI« als Abkürzung von **U**ser **I**nterface = Benutzeroberfläche). Mehr zu Threads finden Sie in ▶ Kapitel 31.

gültige Referenzquelle zugeordnet werden kann, markiert der Garbage Collector. Im zweiten Teil sammelt der Garbage Collector alle markierten Objekte ein und ordnet sie im oberen Teil des Managed Heap an. Objekte, die nicht markiert waren, gibt der GC frei.

Der Algorithmus, mit dem der GC die verwendeten Objekte markiert, ist sehr hoch entwickelt. So erkennt der GC auch Objekte, die nur indirekt durch andere Objekte referenziert sind, und markiert sie. Bei dieser Vorgehensweise löst sich das COM-Problem der Zirkelverweise wie von selbst: Objekte, die von einer Anwendung der AppDomain aus nicht erreichbar sind, werden auch nicht markiert – selbst wenn sie sich untereinander referenzieren. Sie werden im zweiten Durchlauf des GC ebenfalls entsorgt.

Die Geschwindigkeit, mit der der GC seine Arbeit erledigt, ist erstaunlich hoch.[3]

Generationen

Die hohe Geschwindigkeit, mit der der GC arbeitet, bedingt sich insbesondere auch dadurch, dass der GC die zu testenden Objekte in Generationen klassifiziert. Bei der Entwicklung des GC-Algorithmus nahm man an, dass Objekte, die beim Start einer Applikation erzeugt werden, länger im Speicher verbleiben, als solche, die irgendwann zwischendurch oder lokal in Prozeduren generiert werden. Diese Annahme führt zu dem Schluss, dass es Sinn ergibt, bei der Objektentsorgung eine Klassifizierung der Objekte in eben diese Generationen zur Optimierung des GC-Algorithmus vorzunehmen. Der Garbage Collector markiert Objekte nicht nur für die weitere Instandhaltung, er stattet sie auch mit einem Zähler aus, der aussagt, wie oft ein Objekt für die Entsorgung durch den Garbage Collection getestet wurde. Je öfter der Garbage Collector das Objekt bereits »besucht« und nicht entsorgt hat, desto älter ist logischerweise das Objekt (und umso höher ist demzufolge auch seine Generationsnummer), aber desto unwahrscheinlicher ist es auch, dass das Objekt in einem erneuten GC-Lauf entsorgt werden wird.

Wenn der Speicherplatz knapp wird, reicht es deshalb in der Regel aus, Objekte älterer Generationen zunächst außen vor zu lassen, denn die Wahrscheinlichkeit, dass sie entsorgt werden können, ist, wie gesagt, eher gering. Der GC kümmert sich in der Regel also nur um Generation-0-Objekte und versucht diese zu entsorgen. Erst wenn diese Vorgehensweise nicht geholfen hat, für genügend neuen freien Speicher zu sorgen, schaut der GC-Prozess, ob nicht auch bei Objekten älterer Generationen etwas zu holen ist.

Leider wirft diese Vorgehensweise wieder ein Problem ganz anderer Art auf. Objekte können nicht wissen, wann sie entsorgt werden – denn nur die CLR entscheidet, wann ein GC-Durchlauf stattfindet (mit einer Ausnahme):

- Die Common Language Runtime fährt herunter. Das passiert in der Regel dann, wenn eine .NET-Applikation beendet wird.
- Der Speicher wird knapp, weil es zu viele Objekte gibt. Der Garbage Collector startet, um zu sehen, ob Generation-0-Objekte entsorgt werden können, und entsorgt sie im Bedarfsfall.

[3] Übrigens: Das Grundprinzip des Garbage Collectors ist gar nichts Neues. Schon das alte Commodore-Basic (C64, VC20 – meines Wissens auch das Apple-II-Basic) kannte den Garbage Collector für die Entsorgung nicht mehr benötigter Variablen.

- Der Garbage Collector ist in der AppDomain gezwungenermaßen durch die Anweisung *GC.Collect()* gestartet worden.

Hier hatte COM einen eindeutigen Vorteil. Wurde die letzte Referenz aufgelöst, und der Referenzzähler stand auf 0, dann trat das `Terminate`-Ereignis ein, und das Objekt konnte zur »richtigen« Zeit die notwendigen Schritte einleiten, um sich zu entsorgen.

Normalerweise ist es gar kein Problem, dass ein Objekt nicht weiß, dass es entsorgt wird. Wenn es weg ist, dann ist es eben weg. Wichtig, den Zeitpunkt seiner Entsorgung zu kennen, wird es für ein Objekt erst dann, wenn es Aufräumarbeiten erledigen muss, und zwar nicht hinsichtlich der eigenen Speicherverwaltung (denn wenn es andere Objekte referenziert, sorgt der Garbage Collector ja ebenfalls für deren Entsorgung), sondern hinsichtlich der Freigabe von Ressourcen, auf die der Garbage Collector keinen Zugriff hat.

Dies wurde übrigens früher oder wird heute noch bei anderen OOP-Sprachen mit einem Destruktor erledigt. Genau wie wir den Konstruktor kennen gelernt haben, als »Ereignisprozedur«, die ausgeführt wird, wenn ein Objekt erstellt wird, so wird der Destruktor ausgeführt, wenn das Objekt zerstört wird.

Das sind beispielsweise Fälle, in denen das Objekt ein Handle[4] auf eine bestimmte Geräte- oder Betriebssystemressource erhalten hat. Damit dieses Handle wieder freigegeben werden kann – eine geöffnete Datei beispielsweise sollte geschlossen werden – muss das Objekt die dafür erforderlichen Aktionen spätestens kurz bevor es vom Garbage Collector zerstört wird, durchführen.

Genau das geht nicht mehr in .NET. Objekte können nicht voraussehen oder den genauen Zeitpunkt erfahren, wann sie entsorgt werden. Es gibt allerdings die Möglichkeit, dass Objekte erfahren, *dass* sie entsorgt werden, und dann die notwendigen Schritte einleiten, um Ressourcen, die sie belegen, freizugeben.

Daher spricht man in .NET übrigens auch von »nicht deterministischen Destruktoren«, es ist also nicht voraussagbar (determiniert), wann der Destruktor läuft: Nämlich dann, wenn es dem GC passt und wir wissen nicht, wann es ihm passt.

Finalize

Wenn ein Objekt vom Garbage Collector zur Entsorgung markiert wurde, dann ruft der Garbage Collector in der Regel die `Finalize`-Methode des Objektes auf, bevor er den Speicher des Objektes endgültig freigibt. Schon die `Object`-Klasse hat `Finalize` implementiert, und da alle Klassen von `Object` abgeleitet sind, hat jedes Objekt in .NET eine `Finalize`-Methode.[5]

Die `Finalize`-Methode in ihrer Grundimplementierung von `Object` macht überhaupt nichts. Sie ist in erster Linie einfach nur vorhanden, und das bedeutet, dass ein Objekt die `Finalize`-Methode über-

[4] Eine vom Betriebssystem erteilte Kennung zur Nutzung einer bestimmten Ressource (Datei, Bildschirm, Schnittstellen, spezielle Windows-Betriebssystemobjekte, etc.).

[5] Wobei Finalize von `Object` streng genommen gar nicht im Rahmen des GCs aufgerufen wird, da es ohnedies nichts macht; der GC-Algorithmus findet heraus, ob ein »neues« `Finalize` implementiert wurde, und `Finalize` wird nur dann aufgerufen, wenn die `Finalize`-Methode überschrieben wurde.

schreiben muss, wenn es eine eigene Funktionalität für seine »Entsorgungsvorbereitung« implementieren will.

Der Finalizer ist also im Prinzip der Destruktor von .NET Klassen.

BEGLEITDATEIEN: Beachten Sie dazu das folgende Beispiel, das Sie im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap12\Finalize01* finden.

```
Module mdlMain

    Sub Main()
        Dim locTest As New Testklasse("Erste Testklasse")
        Dim locTest2 As New Testklasse("Zweite Testklasse")
        locTest = Nothing
        locTest2 = Nothing
        'GC.Collect()
        Console.WriteLine("Beide Objekte sind nun nicht mehr in Verwendung!")
    End Sub

End Module

Class Testklasse

    Private myName As String

    Sub New(ByVal Name As String)
        myName = Name
    End Sub

    Protected Overrides Sub Finalize()
        MyBase.Finalize()
        Console.WriteLine(Me.myName & " wurde entsorgt")
    End Sub
End Class
```

TIPP: Starten Sie dieses Programm mit der Tastenkombination **Strg+F5**, also ohne Debuggen, damit das Konsolenfenster nach dem Beenden des Programms nicht einfach wieder verschwindet. Sie würden das Ergebnis nicht sehen können.

Wenn Sie dieses Programm starten, dann stellen Sie fest, dass die Meldung »Beide Objekte sind nun nicht mehr in Verwendung!« zuerst ausgegeben wird. Erst anschließend erscheinen die Texte, die anzeigen, dass die beiden verwendeten Objekte finalisiert worden sind:

```
Beide Objekte sind nun nicht mehr in Verwendung!
Zweite Testklasse wurde entsorgt.
Erste Testklasse wurde entsorgt
```

Die Ursache dafür liegt schon fast auf der Hand: Der Garbage Collector arbeitet in diesem Programm nicht, während es läuft, und bei einem Speicheraufkommen von nur ein paar Bytes hat er dafür auch gar keinen Grund. Das Finalisieren der Objekte findet dennoch statt, und zwar beim Beenden der Anwendung. Zu diesem Zeitpunkt ist die letzte Zeile des eigentlichen Programms aber

längst verarbeitet worden; in diesem Fall war es die Codezeile, die den Meldungstext auf den Bildschirm ausgegeben hat.

Eine andere Ausgabe erscheint, wenn Sie das Kommentarzeichen vor der Zeile

```
'GC.Collect()
```

weglassen. Starten Sie das Programm anschließend, ändert sich die Ausgabe in:

```
Zweite Testklasse wurde entsorgt
Erste Testklasse wurde entsorgt
Beide Objekte sind nun nicht mehr in Verwendung!
```

WICHTIG: Im Beispielprogramm habe ich die `Console`-Klasse in der `Finalize`-Methode wie selbstverständlich verwendet. Machen Sie das nicht. Mal ganz davon abgesehen, dass Sie in der `Finalize`-Methode keine wie auch immer gearteten Bildschirmausgaben mehr machen sollten, um sie so schnell wie möglich hinter sich zu bringen, können Sie sich auch nicht sicher sein, ob Objekte, die Sie verwenden, zu diesem Zeitpunkt noch bestehen.

Wann Finalize nicht stattfindet

Unter Umständen verursachen Sie beim Verwenden bestimmter Objekte in `Finalize`, dass neue Objekte während des Vorgangs entstehen, die ihrerseits wiederum neue Objekte anlegen, usw. Diese Objekte müssen natürlich anschließend ebenfalls finalisiert werden. Im schlimmsten Fall lösen Sie damit eine solch enorme Kaskade von neuen Objekten aus, die alle niemals finalisiert werden könnten. Doch ihre Anwendung hängt sich deshalb nicht auf, denn das Framework hat zu diesem Zweck ein paar Sicherheitsmaßnahmen vorgesehen.

BEGLEITDATEIEN: Das folgende Beispiel (im Verzeichnis unter *.\VB 2005 - Entwicklerbuch\D - OOP\Kap12\Finalize-NoNo01* zu finden) verdeutlicht, was Sie in echten Applikationen niemals machen sollten:

```
Module mdlMain

    Sub Main()
        Dim locTest As New Testklasse("Testklasse")
    End Sub

End Module

Class Testklasse

    Private myName As String

    Sub New(ByVal Name As String)
        myName = Name
    End Sub

    Sub WriteText()
        Console.Write(myName)
    End Sub
```

```
    Protected Overrides Sub Finalize()
        MyBase.Finalize()
        Dim locTemp As New Testklasse("locTemp")
        WriteText()
        Console.WriteLine(" wurde entsorgt")
    End Sub
End Class
```

Wenn Sie dieses Programm starten, wird eine Reihe von Meldungen ausgegeben. `Finalize` selbst legt dabei dummerweise eine neue Instanz von `Testklasse` an, um eine Meldung auszugeben. Diese Instanz muss natürlich ebenfalls finalisiert werden, und sie legt ihrerseits wieder eine neue Testklasse-Instanz an usw. Der GC erkennt nach einer Weile, dass der Finalisierungsprozess genau das Gegenteil von dem bewirkt, was er eigentlich bewirken sollte, es entstehen nämlich immer mehr Objekte, und der Speicherbedarf wächst und wächst. Er bricht das Finalisieren nach ein paar Sekunden schlicht und ergreifend ab.

Ein anderes schlechtes Beispiel ist das folgende (wenn Sie es unbedingt selbst probieren wollen: Sie finden es im Begleitdateienverzeichnis unter *\FinalizeNoNo02*). Es legt zwar keine Unmenge von neuem Speicher an, verbraucht für den Finalisierungsprozess aber einfach zu viel Zeit. Der GC wird nach vergleichsweise kurzer Zeit ungeduldig und bricht den gesamten Finalisierungsprozess ab. Das im Beispielprogramm zuerst deklarierte Objekt bekommt keine Chance mehr, finalisiert zu werden.

```
Module mdlMain

    Sub Main()
        '"Normales" Objekt, könnte problemlos finalisiert werden.
        Dim locTest1 As New Testklasse(False, "Erstes Testobjekt")
        'Der Störenfried, da Warteschleifenflag gesetzt.
        Dim locTest2 As New Testklasse(True, "Zweites Testobjekt")
    End Sub

End Module

Class Testklasse

    'Dieses Flag steuert den Einstieg in die Warteschleife.
    Private myWaitInFinalize As Boolean
    'Eine Eigenschaft zum Unterscheiden von Klasseninstanzen
    Private myName As String

    'Flag fürs Warten und den Namen definieren.
    Sub New(ByVal WaitInFinalize As Boolean, ByVal Name As String)
        myWaitInFinalize = WaitInFinalize
        myName = Name
    End Sub

    Protected Overrides Sub Finalize()
        MyBase.Finalize()

        'Nur wenn das Flag bei New gesetzt
        'wurde, in die Warteschleife springen.
        If myWaitInFinalize Then
```

```
            Dim locSecs As Integer
            Dim lastSec As Integer
            lastSec = Now.Second
            Do
                'Jede Sekunde eine Meldung ausgeben.
                If lastSec <> Now.Second Then
                    lastSec = Now.Second
                    locSecs += 1
                    Console.WriteLine("Warte bereits {0} Sekunden", locSecs)
                    'Nach 60 Sekunden wäre Schluss.
                    If locSecs = 60 Then Exit Do
                End If
            Loop

        End If
        'Erfolgreich finalisiert --> Meldung ausgeben
        Console.WriteLine("Objekt {0} wurde finalisiert!", myName)
    End Sub
End Class
```

Folgende Punkte sind also wichtig, wenn Sie eine eigene Finalisierungslogik in Ihren Klassen implementieren müssen:

- Achten Sie darauf, dass der Prozess so schnell wie möglich erledigt ist.
- Sorgen Sie dafür, dass Sie keine neuen Instanzen von irgendwelchen Objekten erstellen.

Wenn Sie diese Punkte beherzigen, tragen Sie enorm zum einwandfreien Funktionieren Ihrer Klassen bei.

Es gibt übrigens Objekte im Framework, deren Vorhandensein die CLR auch noch zum Finalisierungszeitpunkt garantiert. Da Sie Ausgaben bei der Finalisierung wahrscheinlich nur zu Testzwecken machen werden, verwenden Sie dafür besser die `Debug`-Klasse. Diese Klasse stellt ebenfalls eine `Write`- bzw. `WriteLine`-Methode zur Verfügung, hat aber gegenüber `Console` entscheidende Vorteile: Das Vorhandensein der Klasse zum Finalisierungszeitpunkt ist garantiert, und Ausgaben erfolgen darüber hinaus nur in einen so genannten Trace-Listener[6].

Soweit zur Finalisierung von Objekten durch das Framework. `Finalize` ist allerdings eine Methode, die ausschließlich durch das Framework aufgerufen werden darf. Was ist aber, wenn Sie Objekte erstellen wollen, die der Anwender selber – quasi – schließen oder freigeben will?

Das Framework bietet dazu ein Schnittstellenmuster über die so genannte `IDisposable`- Schnittstelle an. Der nächste Abschnitt verrät mehr über dieses Thema.

[6] Ein speicherresistentes Programm, das spezielle Debug-Ausgaben »abhört« und in eigenen Fenstern ausgibt oder sonst wie protokolliert. Falls Sie Programme in der Entwicklungsumgebung von Visual Studio laufen lassen, dann ist das Ausgabefenster der vorinstallierte Trace-Listener. Alle Ausgaben, die Sie mit `Debug.Write(Line)` durchführen, gelangen dann ins Ausgabefenster.

Dispose

Mit der `IDisposable`-Schnittstelle stellt das Framework eine Implementierungsvorschrift bereit, mit deren Hilfe Sie eine Methode implementieren können, die, anders als `Finalize`, auch aus Ihrem Code heraus aufgerufen werden darf, um das Objekt zu entsorgen. Wenn Sie die Schnittstelle per `Implements` in Ihrer Klasse implementieren, müssen Sie die `Dispose`-Methode einfügen, die dann für das notwendige Aufräumen die Verantwortung trägt.

Diese Aufräumarbeiten sind prinzipiell die gleichen Arbeiten, die auch eine `Finalize`-Methode durchführt. Doch `Dispose` hat ein wenig mehr Arbeit. Denn wenn Sie Aufräumarbeiten mit `Dispose` durchgeführt haben, dann müssen Sie dafür sorgen, dass der GC die Aufräumarbeiten innerhalb Ihres Objektes nicht noch einmal durch den Aufruf dieser Methode erledigt. Zu diesem Zweck gibt es die `SuppressFinalize`-Methode des Garbage Collector. Rufen Sie diese Methode auf, und übergeben Sie ihr Ihre Klasse als Argument, schließt der Garbage Collector sie für alle folgenden GC-Durchläufe von Aufräumarbeiten dieser Art aus.

Nun besteht die eigentliche Aufgabe der `Dispose`-Methode darin, zu unterscheiden, ob ein Aufruf durch das eigene Programm eben über `Dispose` (üblich ist bei bestimmten Klassen auch `Close`, das nichts anderes macht als `Dispose`, nur dass es einen anderen Namen trägt[7]) oder durch den Garbage-Collector über `Finalize` erfolgt ist. Ihr `Dispose` muss ebenfalls dafür sorgen, dass erkannt wird, ob ein Objekt schon entsorgt wurde, und im Bedarfsfall eine Ausnahme auslösen.

Ein Beispiel für die Implementierung einer vollständigen `Finalize/Dispose`-Lösung finden Sie in der folgenden Anwendung. Es stellt der Hauptanwendung eine Klasse namens `SoapSerializer` zur Verfügung. Mit Hilfe dieser Klasse können Sie beliebige Objekte im SOAP-Format in einer Datei speichern (mehr zum Serialisieren von Objekte erfahren Sie im ▶ Kapitel 22). Umgekehrt kann die Klasse aus einer Datei die ursprünglichen Objekte wieder automatisch herstellen.

BEGLEITDATEIEN: Sie finden die Projektdateien für dieses Beispiel im Verzeichnis *.\VB 2005 - Entwicklerbuch\D - OOP\Kap12\Dispose*.

Das Hauptprogramm, das Sie im Folgenden finden, ist dabei sehr einfach gehalten, selbst von eher untergeordnetem Interesse und auch wieder eine Konsolenanwendung. Es fragt den Anwender nach dem Programmstart, ob er eine SOAP-Datei laden und deren Daten anzeigen oder Daten erfassen und abspeichern möchte. Es bedient sich dabei zur Speicherung der Daten einer Klasse, die lediglich Namen und Vornamen einer Person aufnimmt:

```
Möchten Sie Daten erfassen und speichern (1)
oder Daten laden und anzeigen (2)?
Ihre Auswahl  :1
Wieviele Daten möchten Sie eingeben?  :4

----------------------------------
```

[7] Aber wie heißt es so schön: Ausnahmen bestätigen die Regel. Die `System.Windows.Forms`-Klasse gehört hierzu: Wenn Sie ein Formular mit `Close` schließen, haben Sie die Möglichkeit, das Schließen des Formulars im `FormClosing`-Ereignis zu verhindern (in dem Sie `e.Cancel` auf `True` setzen). »Schließen« Sie das Formular jedoch mit `Dispose`, was auch geht, dann »schießen« Sie es sozusagen ab. Es ist sofort zu, weg, verschwunden, entsorgt, und es löst auch keine Ereignisse mehr aus.

```
Eingabe der 1. Person
Nachname: Heckhuis
Vorname: Jürgen

-----------------------------------
Eingabe der 2. Person
Nachname: Thiemann
Vorname: Uwe

-----------------------------------
Eingabe der 3. Person
Nachname: Ademmer
Vorname: Ute

-----------------------------------
Eingabe der 4. Person
Nachname: Löffelmann
Vorname: Klaus
```

Wenn Sie den letzten Namen eingegeben haben, wird das Programm auch schon beendet. Die Daten befinden sich anschließend in der Datei »Test.xml« auf Laufwerk »C:«, und diese sieht folgendermaßen aus:


```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<xsd:int id="ref-1">
<m_value>4</m_value>
</xsd:int>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:Dataset id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Dispose/Dispose%2C%20Version%3D1.0.1431.30420%2C%20Cultur
e%3Dneutral%2C%20PublicKeyToken%3Dnull">
<myFirstName id="ref-3">Jürgen</myFirstName>
<myLastName id="ref-4">Heckhuis</myLastName>
</a1:Dataset>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
```

```
<a1:Dataset id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Dispose/Dispose%2C%20Version%3D1.0.1431.30420%2C%20Cultur
e%3Dneutral%2C%20PublicKeyToken%3Dnull">
<myFirstName id="ref-3">Uwe</myFirstName>
<myLastName id="ref-4">Thiemann</myLastName>
</a1:Dataset>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:Dataset id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Dispose/Dispose%2C%20Version%3D1.0.1431.30420%2C%20Cultur
e%3Dneutral%2C%20PublicKeyToken%3Dnull">
<myFirstName id="ref-3">Ute</myFirstName>
<myLastName id="ref-4">Ademmer</myLastName>
</a1:Dataset>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:Dataset id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Dispose/Dispose%2C%20Version%3D1.0.1431.30420%2C%20Cultur
e%3Dneutral%2C%20PublicKeyToken%3Dnull">
<myFirstName id="ref-3">Klaus</myFirstName>
<myLastName id="ref-4">Löffelmann</myLastName>
</a1:Dataset>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Man möchte sagen: Zwar ein ziemlicher SOAP-Overhead für diese paar Daten, aber was soll's: Es ist nur ein Demo und Speicher ist billig! Viel wichtiger ist, dass das Programm auch funktioniert, und das finden Sie heraus, indem Sie das Programm abermals starten und anschließend die Funktion zum Anzeigen der Daten auswählen:

```
Möchten Sie Daten erfassen und speichern (1)
oder Daten laden und anzeigen (2)?
Ihre Auswahl  :2
Heckhuis, Jürgen
Thiemann, Uwe
Ademmer, Ute
Löffelmann, Klaus
```

Wie arbeitet das Programm nun? Bevor ich zur Erklärung schreite, eine kleine Warnung vorweg: Die Funktionsweise des Programms ist recht wichtig für das spätere Verständnis von `Dispose` und `Finali-`

ze. Wundern Sie sich also bitte nicht, wenn ich zunächst auf den folgenden Seiten ein paar andere Themen aufgreife, bevor wir uns dann dem eigentlichen Gegenstand der Erklärung widmen.

Zurück zum Programm: Zunächst gibt es eine Klasse, die die Daten speichert. Wichtig dabei: Wenn Sie eine Klasse serialisieren, dann müssen folgende Voraussetzungen gegeben sein:

- Alle Datentypen, die die Klasse verwendet, müssen serialisierbar sein.
- Die Klasse selbst muss serialisierbar sein und dazu mit einem besonderen Attribut gekennzeichnet werden:

```
<Serializable()> _
Class Dataset

    Private myFirstName As String
    Private myLastName As String

    Sub New(ByVal FirstName As String, ByVal LastName As String)
        myFirstName = FirstName
        myLastName = LastName
    End Sub

    Overrides Function ToString() As String
        Return myLastName & ", " & myFirstName
    End Function

End Class
```

Das Hauptmodul ist dafür zuständig, die Daten zu erfassen und abzuspeichern bzw. zu laden und auf dem Bildschirm anzuzeigen. Sie werden überrascht sein, wie wenig Aufwand für das Sichern bzw. das Wiederherstellen erforderlich ist:

```
Module mdlMain
    Sub Main()
        '"Menü" auf den Bildschirm zaubern:
        Console.WriteLine("Möchten Sie Daten erfassen und speichern (1)")
        Console.WriteLine("oder Daten laden und anzeigen (2)?  ")
        Console.Write("Ihr Auswahl  :")

        'Auswahl einlesen
        Dim locKey As String = Console.ReadLine()

        'Daten sollen erfasst werden.
        If locKey = "1" Then

            Dim locAnzPersonen As Integer
            Dim locName, locVorname As String
            Dim locSoapWriter As SoapSerializer

            Console.Write("Wieviele Daten möchten Sie eingeben?  :")
            locAnzPersonen = Integer.Parse(Console.ReadLine())
            If locAnzPersonen = 0 Then
                Exit Sub
            End If
```

```
        'Serializer vorbereiten.
        locSoapWriter = New SoapSerializer

        'Zum Schreiben öffnen.
        locSoapWriter.OpenForWriting("C:\Test.XML", True)

        'Anzahl der Datensätze abspeichern.
        locSoapWriter.SaveObject(locAnzPersonen)

        'Soviele Personen einlesen, wie zuvor eingegeben.
        For locCount As Integer = 1 To locAnzPersonen
            Console.WriteLine()
            Console.WriteLine("-----------------------------------")
            Console.WriteLine("Eingabe der {0}. Person", locCount)
            Console.Write("Nachname: ")
            locName = Console.ReadLine
            Console.Write("Vorname: ")
            locVorname = Console.ReadLine

            'In das Objekt übertragen und abspeichern.
            Dim locData As New Dataset(locVorname, locName)
            locSoapWriter.SaveObject(locData)
        Next

        'Das kann man schon mal vergessen!!!
        locSoapWriter.Close()

        'Der umgekehrte Weg: Die Daten werden geladen.
    Else

        Dim locAnzPersonen As Integer
        Dim locData As Dataset
        Dim locSoapReader As SoapSerializer

        'Deserializer vorbereiten.
        locSoapReader = New SoapSerializer

        'Zum Lesen öffnen.
        locSoapReader.OpenForReading("C:\Test.XML")

        'Anzahl der Datensätze lesen und
        'zurückboxen von Object zu Wertetyp Integer.
        locAnzPersonen = Convert.ToInt32(locSoapReader.LoadObject())

        'Soviele Personen-Datensätze lesen, wie ursprünglich erfasst.
        For locCount As Integer = 1 To locAnzPersonen
            'Deserialisieren und in den alten Objekttyp zurückwandeln.
            locData = DirectCast(locSoapReader.LoadObject(), Dataset)
            'Daten ausgeben.
            Console.WriteLine(locData.ToString)
        Next
```

```
            'So geht es auch:
            locSoapReader.Dispose()

        End If
    End Sub

End Module
```

Sie sehen: Womit das Programm am meisten zu tun hat, ist das Ausgeben der »Menü-Texte« auf dem Bildschirm. Die eigentliche Arbeit erledigt die `SoapSerializer`-Klasse, der wir uns als Nächstes und erklärungstechnisch ein wenig intensiver widmen wollen.

Die Klasse befindet sich in einer eigenen Datei im Projekt (namens *SoapSerializer.vb*). Doppelklicken Sie auf den Dateinamen im Projektmappen-Explorer, um sie im Codeeditor betrachten zu können.

Der Programmcode beginnt mit einer Reihe von `Imports`-Anweisungen, die verschiedene Namespaces einbinden.

```
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Soap
```

Da das Programm sowohl die `MemoryStream`-Klasse als auch die Serialisierung auf *Soap*-Basis verwendet, steht die `Imports`-Anweisung für die Namespaces, denen diese Klassen zugeordnet sind, an erster Stelle in der Codedatei. Zusätzlich gibt es eine Referenz auf die .NET-Framework-Assembly *System.Runtime.Serialization.Formatters.Soap*, die zuvor über den Projektmappen-Explorer mit *Verweis hinzufügen* eingebunden wurde.

```
Public Enum SoapSerializerMode
    Close
    OpenForWriting
    OpenForReading
End Enum
```

Die `Enum` benötigen wir lediglich, um das Programm leichter lesbar zu machen (mehr zum Thema *Enum* finden Sie in ▶ Kapitel 18).

```
Public Class SoapSerializer
    Implements IDisposable
```

Mit der `Implements`-Anweisung bindet die Klasse das `IDisposable`-Pattern ein. Zur Wiederholung: Bei einem Schnittstellen-Pattern wird eine Schnittstelle in erster Linie zur Standardisierung verwendet. Erst in zweiter Linie dient sie zur Realisierung von polymorphen Aufrufen von Methoden in abgeleiteten Klassen. Die `IDisposable`-Schnittstelle zwingt den Entwickler einer Klasse, die `Dispose`-Methode in einer bestimmten Form zu implementieren. Das Framework selbst ruft, wie ebenfalls bereits erwähnt, `Dispose` nie auf.

```
    Protected myFilename As String
    Protected myMemoryStream As MemoryStream
    Protected mySoapFormatter As SoapFormatter
    Protected mySerializerMode As SoapSerializerMode
    Protected myDisposed As Boolean
```

```
Sub New()
    mySoapFormatter = New SoapFormatter(Nothing, _
                    New StreamingContext(StreamingContextStates.File))
    mySerializerMode = SoapSerializerMode.Close
End Sub
```

Der SoapFormatter wird für die Serialisierung der Objekte verwendet. Er steuert quasi »das Aussehen« der Daten, wenn Objekte serialisiert werden. In diesem Zusammenhang möchte ich nicht näher darauf eingehen (im ▶ Kapitel 22 erfahren Sie mehr darüber).

```
Function OpenForWriting(ByVal Filename As String) As Boolean
    Return OpenForWriting(Filename, False)
End Function

Function OpenForWriting(ByVal Filename As String, ByVal OverwriteIfExist As Boolean) As Boolean

    Dim locFile As New FileInfo(Filename)

    If (Not OverwriteIfExist) And locFile.Exists Then
        Return True
    End If

    myFilename = Filename

    Try
        mySerializerMode = SoapSerializerMode.OpenForWriting
        myMemoryStream = New MemoryStream()
    Catch ex As Exception
        mySerializerMode = SoapSerializerMode.Close
    End Try

End Function

Function OpenForReading(ByVal Filename As String) As Boolean

    Dim locFile As New FileInfo(Filename)

    If Not locFile.Exists Then
        Return True
    End If

    Try
        mySerializerMode = SoapSerializerMode.OpenForReading
        myMemoryStream = New MemoryStream(My.Computer.FileSystem.ReadAllBytes(Filename))
    Catch ex As Exception
        mySerializerMode = SoapSerializerMode.Close
    End Try

End Function
```

Diese Funktionen erstellen, je nach Anforderung, einen so genannten MemoryStream, in den eine Objektserialisierung oder Deserialisierung im Speicher vorgenommen werden kann. Die Vorgehensweise beim Serialisieren in eine Datei ist einfach:

- `MemoryStream` erstellen,
- Objekt in diesen Speicher »hineinserialisieren«,
- MemoryStream nach Serialisierung aller Objekte in eine Datei schreiben.

Analog funktioniert das Deserialisieren, bei dem Daten einer Datei, die in einem bestimmten Format vorliegen, zunächst in einen MemoryStream geladen werden, aus dem dann wiederum die verschiedenen Objekte »gewonnen« werden können.

Zum nächsten Punkt:

```
Sub SaveObject(ByVal Data As Object)

    'Serialisierung geht nur, wenn SoapSerializer
    'zum Schreiben geöffnet wurde.
    If mySerializerMode <> SoapSerializerMode.OpenForWriting Then
        Dim Up As New IOException("SoapSerializer nicht zum Schreiben geöffnet!")
        Throw Up
    End If

    mySoapFormatter.Serialize(myFileStream, Data)

End Sub

Function LoadObject() As Object

    'Deserialisierung geht nur, wenn SoapSerializer
    'zum Schreiben geöffnet wurde.
    If mySerializerMode <> SoapSerializerMode.OpenForReading Then
        Dim Up As New IOException("SoapSerializer nicht zum Lesen geöffnet!")
        Throw Up
    End If
    Return mySoapFormatter.Deserialize(myFileStream)

End Function
```

Sie sehen, wie einfach das Serialisieren und Deserialisieren von Objekten im Grunde genommen ist. Mit jeweils einem einzigen Befehl können Sie aus einem Objekt einen Datenstrom oder aus einem Datenstrom wieder ein Objekt machen. Beim Deserialisieren lässt die CLR das Objekt in seiner ursprünglichen Gestalt »wieder auferstehen«. Am einfachsten zu vergleichen ist das mit dem Vorgang des Beamens in StarTrek. Wenn Sie eine Person an einen anderen Platz beamen, wird sie zunächst serialisiert, dann in einem Strom (Strahl, o.k.) ans Ziel geschickt und dort wieder deserialisiert. Der Unterschied: Im Framework funktioniert das »zum Leben erwecken« von Objekten tatsächlich …

```
'Nur eine andere Form von Disposed
Sub Close()
    Debug.WriteLine("Close() wurde aufgerufen!")
    Dispose()
End Sub
```

```
    'Hier wird das Entsorgen delegiert
    Public Sub Dispose() Implements IDisposable.Dispose
        'Wir kümmern uns um die Entsorgung,
        'der Garbage Collector wird informiert,
        'dass er nichts mehr damit zu tun hat
        Debug.WriteLine("Dispose() wurde aufgerufen!")
        GC.SuppressFinalize(Me)
        Dispose(True)
    End Sub

    Public Sub Dispose(ByVal Disposing As Boolean)

        Debug.WriteLine("Dispose(Disposing) wurde aufgerufen...")
        'Falls der Aufruf nicht durch die GC kam
        If Disposing Then
            Debug.WriteLine("...kam von der Applikation")
            'prüfen, ob nicht schon entsorgt
            If myDisposed Then
                'Übergründlich geht nicht, dann --> Exception
                Dim up As New ObjectDisposedException("SoapSerializer")
                Throw up
            End If
        Else
            Debug.WriteLine("...kam vom Garbage Collector")
        End If

        'An dieser Stelle werden die Daten in die Datei geschrieben
        'So kann verhindert werden, dass die Datei die ganze Zeit
        'geöffnet bleibt.
        If mySerializerMode = SoapSerializerMode.OpenForWriting Then
            My.Computer.FileSystem.WriteAllBytes(myFilename, myMemoryStream.ToArray, False)
            Debug.WriteLine("Daten aus Memorystream geschrieben - Datei ist sicher!")
            myMemoryStream.Dispose()
        End If
        Debug.WriteLine("MemoryStream disposed")
        mySerializerMode = SoapSerializerMode.Close
    End Sub

    Protected Overrides Sub Finalize()
        'Falls myMemoryStream schon durch den GC entsorgt wurde
        'könnte ein Fehler auftreten, den es abzufangen gilt
        Try
            Debug.WriteLine("Me.Finalize ruft Me.Dispose auf!")
            Dispose(False)
        Finally
            Debug.WriteLine("Base.Finalize aufrufen!")
            MyBase.Finalize()
        End Try
    End Sub

End Class
```

Diese letzten Zeilen der Klasse haben es in sich, und nach dem ganzen Vorgeplänkel sind wir leider erst jetzt beim eigentlichen Thema.

Klären wir zunächst die Frage: Wozu braucht diese Klasse überhaupt ein `Finalize` und ein `Dispose`? Die Klasse verwendet ein `MemoryStream`-Objekt. Und: Die Klasse schreibt zunächst in diesen »Dateistrom«, aber erst bei der Ausführung von `Close` den erstellten Datenstrom in eine Datei. Wenn Sie Daten in einen Datenstrom hineinschreiben, dann müssen Sie sicherstellen, dass Daten, die sich dort befinden, sich später auch bis aufs letzte Byte in der angegebenen Datei befinden. Das gewährleistet während des `Close`- bzw. `Dispose`-Vorgangs die Zeile, die Sie im oben stehenden Listing in fetter Schrift sehen.

Nun öffnet unsere Klasse ein `MemoryStream`-Objekt automatisch, wenn der Entwickler, der die Klasse verwendet, eine der beiden Methoden `OpenForWriting` oder `OpenForReading` verwendet. Er kann nun mit `SaveObject` bzw. `LoadObject` den Datenstrom verwenden. Er muss anschließend aber auch – und jetzt kommt der `IDisposable`-Pattern ins Spiel – dafür sorgen, dass alles wieder geschlossen wird, nur dann wird – und das ist wichtig im Falle des Schreibens – der zunächst im Speicher angelegte Datenstrom mit den Objektdaten tatsächlich in die Datei geschrieben. Macht er es nicht, dann sollte unsere Klasse intelligent genug sein, um zu retten, was zu retten ist. Die Klasse sorgt also für das Speichern des Speicherdatenstroms, wenn …

- … der Entwickler die `Dispose`-Methode (oder die `Close`-Methode – das ist in diesem Fall dasselbe), so wie es sein sollte, selbst aufruft, oder
- … der Entwickler es vergessen hat, aber der Garbage Collector uns durch den Aufruf von `Finalize` anzeigt, dass die Klasse zur Entsorgung ansteht, und spätestens jetzt alle verwendeten Ressourcen möglichst schnell aufgeräumt und freigegeben werden sollten.

Nun könnte man meinen, es reiche aus, `Finalize` einfach `Dispose` oder umgekehrt aufrufen zu lassen und die Implementierung zum korrekten Freigeben der Ressourcen einfach in einer der beiden Routinen zu verstecken. Das geht aber leider nicht, denn es gibt die drei folgenden Einschränkungen:

- `Finalize` darf nur vom Garbage Collector aufgerufen werden; `Finalize` der Basisklasse muss dabei obendrein grundsätzlich, immer und um jeden Preis aufgerufen werden.
- `Dispose` darf maximal einmal aufgerufen werden, denn ein einmal entsorgtes Objekt kann nicht noch einmal entsorgt werden. Wird `Dispose` ein zweites Mal aufgerufen, sollte die Klasse eine `ObjectDisposedException` ausgeben.
- Der Garbage Collector darf `Finalize` nicht aufrufen, wenn das Objekt bereits quasi »durch sich selbst« (also durch `Dispose`) entsorgt wurde.

Genau diese drei Fälle werden im Code berücksichtigt:

Sobald der Anwender die Klasse schließen will, ruft er entweder die Methode `Close` oder `Dispose` auf. Ruft er `Close` auf, wird er an `Dispose` (ohne Parameter) weitergeleitet. Diese Version von `Dispose` sorgt als erstes mit `GC.SuppressFinalize(Me)` dafür, dass der Garbage Collector, falls eine Garbage Collection ansteht, `Finalize` für dieses (`Me`) Objekt nicht mehr aufrufen wird – `Dispose` ist ja schließlich gerade dabei, die Finalisierung durchzuführen, und einmal reicht!

HINWEIS: Aufgepasst dabei: `GC.SuppressFinalize(Me)` bedeutet nicht, dass der Garbage Collector das Objekt nicht mehr entsorgen wird – er wird während der Entsorgung lediglich nicht die `Finalize`-Methode des Objektes aufrufen; die Entsorgung findet immer statt; ein Objekt kann sich nicht dagegen wehren und die Entsorgung auch nicht verzögern oder die Reihenfolge in irgendeiner Form beeinflussen. Wenn es seine Lebensberechtigung verloren hat, ist es fällig, so oder so.

`Dispose` ruft nun seinerseits die `Dispose`-Überladung (mit Parameter) auf und übergibt ihr `True` als Argument und zum Zeichen, dass der Aufruf nicht durch den `Finalize`-Prozess bedingt war. Die überladene `Dispose`-Methode kann nun anhand der booleschen Variable feststellen, ob sie von `Finalize` oder manuell durch das Programm ins Leben gerufen wurde.

Falls das Programm der Grund für den Aufruf war, stellt `Dispose` sicher, dass das Objekt nicht schon entsorgt wurde – zu diesem Zweck gibt es die Member-Variable `myDisposed`, die quasi Buch darüber führt. Sollte dies jedoch der Fall gewesen sein, bringt `Dispose` die geforderte `ObjectDisposedException`-Ausnahme.

Anschließend erfolgen die eigentlichen Aufräumarbeiten. Der Speicherdatenstrom wird – sofern es sich um einen Schreibvorgang handelte – in eine Datei mit dem zuvor bestimmten Dateinamen geschrieben. Zu guter Letzt wird noch das Flag gesetzt, das das Objekt nunmehr als entsorgt kennzeichnet, damit ein zweites, versehentliches `Dispose` nicht mehr stattfinden kann. Und das ist auch wichtig, denn das `MemoryStream`-Objekt gibt es bei einem möglichen zweiten `Dispose`-Aufruf nicht mehr. Eine Ausnahme wäre die unvermeidliche Folge.

Sollte der Entwickler vergessen, `Dispose` oder `Close` aufzurufen, dann erfolgt kein `GC.SuppressFinalize(Me)` und der Garbage Collector ruft beim Entsorgen des Objektes die `Finalize`-Methode auf. Damit beim Finalisierungsendspurt keine Fehler zu einer Ausnahme führen, wird mit der Konstruktion `Try/Finally` dafür gesorgt, dass der Aufräumprozess soweit wie möglich gelingt, die Basismethode aber – ganz gleich ob Ausnahme oder nicht – noch in jedem Fall aufgerufen wird. `Finalize` des Objektes selbst ruft `Dispose` auf, jetzt aber mit `Disposing = False`. Für dieses Beispiel macht das keinen großen Unterschied; der Test auf ein bereits stattgefundenes `Dispose` findet lediglich nicht statt. Andere Objekte müssen aber möglicherweise diesen Unterschied kennen, um entsprechend auf die verschiedenen Auslöser (manuelles `Dispose` oder `Finalize`) reagieren zu können.

Sie können das Verhalten dieser Klasse übrigens sehr einfach nachvollziehen, indem Sie mit dem Programm experimentieren und das Ausgabefenster (nicht das Konsolenfenster) dabei beobachten. Es informiert Sie stets über die gerade durchgeführte Aufgabe, und durch das Verändern einiger Eigenschaften bzw. Aufrufe der `SoapSerializer`-Klasse im Hauptprogramm können Sie die unterschiedlichen Reaktionen des Finalisierungsprozesses testen.

Unterstützung durch den Visual Basic-Editor beim Einfügen eines Disposable-Patterns

Das Muster, auf das Sie beim Implementieren einer »disposebaren« Klasse achten müssen, ist nicht nur etwas komplexer, als bei der Implementierung anderer Funktionen, die Sie durch die Einbindung einer oder mehrerer Schnittstellen implementieren müssen.

Sie müssen auch darauf achten, dass ein internes Muster, so, wie Sie es im vorherigen Beispiel kennen gelernt haben, strikt einzuhalten ist – und das geht natürlich weit über das bloße Vorhandensein einer entsprechenden `Dispose`-Methode hinaus.

Aus diesem Grund gibt es eine besondere Editor-Unterstützung für die `IDisposable`-Schnittstelle. Sobald sie diese am Klassenkopf einfügen, und nach `Implements IDisposable` **Eingabe** betätigen, fügt der Editor nicht nur den Funktionsrumpf, sondern ein etwas spezifischeres Funktionsgerüst als Code in Ihre Klasse ein, wie im Folgenden zu sehen:

```
    Private disposedValue As Boolean = False        ' So ermitteln Sie überflüssige Aufrufe

    ' IDisposable
    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        If Not Me.disposedValue Then
            If disposing Then
                ' TODO: Nicht verwaltete Ressourcen freigeben, wenn sie explizit aufgerufen werden
            End If

            ' TODO: Gemeinsam genutzte nicht verwaltete Ressourcen freigeben
        End If
        Me.disposedValue = True
    End Sub

#Region " IDisposable Support "
    ' Dieser Code wird von Visual Basic hinzugefügt, um das Dispose-Muster richtig zu implementieren.
    Public Sub Dispose() Implements IDisposable.Dispose
' Ändern Sie diesen Code nicht. Fügen Sie oben in Dispose(ByVal disposing As Boolean) Bereinigungscode ein.
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
#End Region
```

Diese Aktion bildet eine, wie ich finde, perfekte Unterstützung für die Implementierung von `IDisposable`. Sie müssen Modifizierungen nur unterhalb der mit `'TODO` gekennzeichneten Codeteile vornehmen – ansonsten können Sie die Implementierung so belassen, wie sie ist.