Kapitel 13

Lebensdauer von Objekten

Wenn Sie bereits längere Zeit mit Objekten arbeiten, scheinen diese eine Art Eigenleben zu entwickeln. Objekte werden geboren, treiben sich eine Zeitlang auf Ihrem System herum und sterben schließlich wieder. Manchmal geraten sie auch ein wenig außer Kontrolle. Sie müssen sorgfältig darauf achten, die Objekte nach der Erledigung ihrer Arbeit wieder zu entfernen. Anderenfalls sammeln sich deren Überbleibsel in Ihrem System an und verbrauchen Ressourcen, auf die andere Objekte angewiesen sind. Meines Wissens gibt es keine bedeutendere Weltanschauung oder Religion, die ein ethisches Problem mit der Eliminierung von COM-Objekten hätte – die COM-Programmierung ist also eines der raren Felder, auf dem Sie sich nach Herzenslust und ohne Gewissensbisse als »Killer« betätigen können.

Die Beseitigung von Objekten kann mitunter in eine verzwickte Angelegenheit ausarten. Allein schon die Referenzierungen nachzuhalten und immer an deren Auflösung zu denken, ist nicht leicht. Die eigentliche Herausforderung liegt im Entwurf eines Objekt-Modells, daß die Bewältigung dieser Aufgabe auch tatsächlich ermöglicht. Ein Objekt-Modell, das Ihren Objekten gewissermaßen Unsterblichkeit verleiht (nur ein »Weltuntergang«, sprich: Beenden der Anwendung, verheißt dann noch Rettung), laßt sich dagegen ohne weiteres aus dem Ärmel schütteln.

13.1 Referenzierung

In Kapitel 4, »Das Component Object Model: Interfaces, Automation und Bindung«, führten wir das Konzept der Objekt-Variablen und der Funktionsweise ein. Halten wir uns dieses noch einmal kurz vor Augen.

Objekt-Variablen unterscheiden sich von anderen Variablen-Typen. Der Unterschied liegt darin, daß Objekt-Variablen selbst keine direkten Daten enthalten. Statt dessen enthalten sie Zeiger auf Interfaces von Objekten.

Vergegenwärtigen Sie sich den Vorgang der Deklaration zweier Variablen und der Zuweisung der einen zur anderen. In Abbildung 13.1 sehen Sie zwei `String`-Variablen, Variable `A` und Variable `B`. Zunächst wird Variable `A` der String `"Data in A"` zugewiesen und Variable B der String `"Data in B"`. Beide Variablen enthalten nun String-Daten. Weisen Sie nun per Anweisung `A = B` die Variable `B` der Variablen `A` zu, werden die Daten aus Variable `B` nach Variable `A` kopiert.

In Abbildung 13.2 sehen Sie dieselbe Operation mit Objekt-Variablen. Die zwei Variablen enthalten jeweils einen Zeiger auf ein Interface eines Objekts. Jedes Objekt hält nach, wie viele Variablen auf das Objekt zeigen. Zunächst wird auf jedes Objekt von einer Variablen verwiesen – die Referenzzähler der Objekte stehen damit jeweils auf 1.

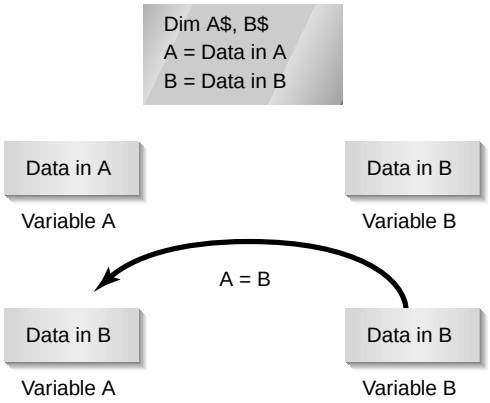
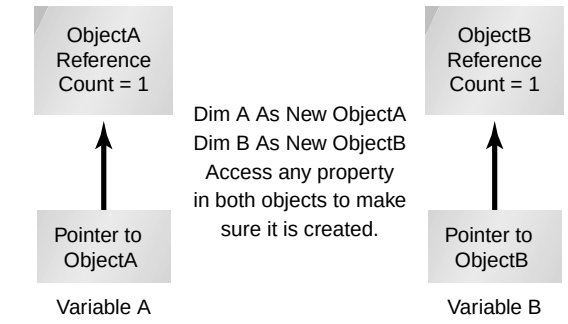


Abb. 13.1: Zuweisung von Daten-Variablen



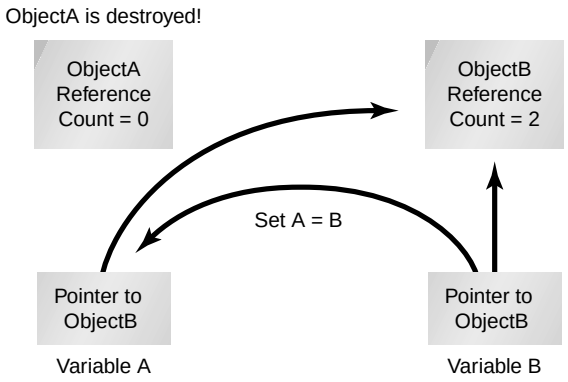


Abb. 13.2: Zuweisung von Objekt-Variablen

Bei einer Zuweisung nach dem Muster `Set A = B` steckt hinter dieser Operation mehr als nur eine schlichte Zuweisung. Zuerst muß Visual Basic eine `Release`-Operation (Freigabe) für das Interface von Objekt `A` durchführen. Dadurch wird der Referenzzähler von Objekt `A` um 1 erniedrigt und erreicht 0. Das Objekt wird freigegeben, oder wie man auch sagt, »zerstört«. Nimmt nun die Variable `A` einen Zeiger auf Objekt `B` auf, wird eine `AddRef`-Operation für das Interface von Objekt `B` ausgeführt. Dadurch wird der Referenzzähler von Objekt `B` auf 2 erhöht.

13.1.1 Was geschieht wann ...?

Schauen wir uns an, was tatsächlich geschieht, wenn die eine oder andere Objekt-Operation in Visual Basic stattfindet. Den Code des Beispiels dieses Abschnitts `Test1.vbp` finden Sie im Ordner zu Kapitel 13 auf der Buch-CD. Betrachten wir zunächst den Vorgang der Deklaration einer Variablen:

```
Private Sub cmdStart1_Click()
    Dim A As myobject

    Debug.Print "A is declared"
    If A Is Nothing Then
        Debug.Print "A is nothing"
    End If
End Sub
```

Das Objekt `myobject` ist ein einfaches Objekt, das `Debug.Print`-Anweisungen in den Ereignis-Prozeduren `Class_Initialize` und `Class_Terminate` enthält, damit Sie sehen können, wann das Objekt angelegt oder zerstört wird. Es verfügt über die Eigenschaft `Name` zur Identifikation der einzelnen Instanzen.

In diesem Fall wird die Variable `A` deklariert. Noch zeigt sie auf kein Objekt. Der Zeiger-Wert beträgt intern derzeit Null – eine ungültige Zeiger-Adresse. Diese ungültige Adresse wird jedoch noch nicht als Fehler wahrgenommen – damit wird Visual Basic und Ihrem Programm lediglich mitgeteilt, daß der Variablen noch kein Objekt zugewiesen worden ist. Dieser Nullzeiger trägt unter Visual Basic einen speziellen Namen: Nothing. Zu diesem Zeitpunkt läßt sich also die Aussage treffen: »Variable A ist gleich Nothing.« Sie können dies mit Hilfe der Operation `If A Is Nothing Then...` prüfen. Falls Sie nun versuchen sollten, auf eine Eigenschaft des Objekts, als dessen Typ die Variable `A` deklariert worden ist, zuzugreifen, etwa auf `A.Name`, würde dies mit einer Fehlermeldung quittiert – Sie können nicht auf eine Eigenschaft eines Objekts zugreifen, das (noch) nicht existiert.

Schauen wir uns den Beispiel-Code in `cmdStart2_Click` an:

```
Private Sub cmdStart2_Click()
    Dim A As New myobject

    Debug.Print "A is declared"
    MsgBox "Look at the immediate window"
```

```
    Debug.Print "Message box returns"
    If A Is Nothing Then
        Debug.Print "A is nothing"
    Else
        Debug.Print "A is valid"
    End If

End Sub
```

Der wesentliche Unterschied zwischen diesem Beispiel und dem vorangegangenen liegt in der Deklaration der Variablen `A` mit dem `New`-Operator. Wenn Sie dieses Beispiel starten, wird folgendes im Direkt-Fenster ausgegeben:

```
A is declared
Message box returns
Object is created
A is valid
Object is destroyed
```

An dieser Sequenz fällt auf, daß der `New`-Operator das Objekt nicht unmittelbar erzeugt. Er teilt Visual Basic lediglich mit, daß ein neues Objekt dann angelegt werden soll, wenn der erste Zugriff auf die Variable erfolgt. Und wo erfolgt in diesem Beispiel dieser erste Zugriff? Er erfolgt beim Vergleich mit `Nothing`. Daraus ergibt sich, daß eine mit New deklarierte Variable nie gleich `Nothing` sein wird – Visual Basic wird sofort beim ersten Versuch einer Referenzierung ein Objekt anlegen. Dies ist auch anhand des folgenden Codes zu erkennen:

```
Private Sub cmdStart3_Click()
    Dim A As New myobject

    Debug.Print "A is declared"
    If A Is Nothing Then
        Debug.Print "A is nothing"
    Else
        Debug.Print "A is valid"
    End If

    Set A = Nothing

    If A Is Nothing Then
        Debug.Print "A is nothing"
    Else
        Debug.Print "A is valid"
    End If

End Sub
```

Im Direkt-Fenster wird ausgegeben:

```
A is declared
Object is created
A is valid
Object is destroyed
Object is created
A is valid
Object is destroyed
```

Das erste Objekt wird bei der Ausführung der Operation `Set A = Nothing` zerstört. Doch sobald Sie die Objekt-Variable erneut prüfen wollen, wird von Visual Basic ein neues Objekt angelegt.

Erinnern Sie sich an meine Aussage, daß das Beseitigen von Objekten in Visual Basic eine Herausforderung sein kann? Wie Sie sehen, ist es anscheinend auch nicht so einfach, sich ihrer auf Dauer zu entledigen. In diesem Fall hier stellt das Wiedererscheinen kein besonderes Problem dar – zu einem solchen kann es sich aber dann auswachsen, wenn bei der Initialisierung oder Terminierung eines Objekts umfangreiche Operationen durchgeführt werden. So kann es unter Umständen vorkommen, daß Ihre Anwendung eine Menge Zeit damit verschwendet, gar nicht benötigte Objekte anzulegen und zu zerstören. Im Beispiel `cmdStart4_Click` sehen Sie einen Weg, dieses Problem zu umgehen.

```
Private Sub cmdStart4_Click()
    Dim A As myobject

    Debug.Print "A is declared"

    If A Is Nothing Then
        Debug.Print "A is nothing"
    Else
        Debug.Print "A is valid"
    End If

    Debug.Print "About to call Set A = New myobject"
    Set A = New myobject
    Debug.Print "Set A = New myobject called"

    If A Is Nothing Then
        Debug.Print "A is nothing"
    Else
        Debug.Print "A is valid"
    End If

    Set A = Nothing

    If A Is Nothing Then
```

```
        Debug.Print "A is nothing"
    Else
        Debug.Print "A is valid"
    End If

End Sub
```

Diese Prozedur führt zu folgenden Ausgaben:

```
A is declared
A is nothing
About to call Set A = New myobject
Object is created
Set A = New myobject called
A is valid
Object is destroyed
A is nothing
```

Bei der Deklaration der Variablen wird noch kein Objekt angelegt. Auch beim Zugriff auf die Variable wird nicht automatisch ein Objekt angelegt. Das geschieht erst durch die Operation `Set A = New myobject`. Da bei einem weiteren Zugriff nicht automatisch erneut ein Objekt angelegt wird, bleibt es auch weiterhin in der Versenkung verschwunden.

Im allgemeinen ist dieser Ansatz der bessere gegenüber der Deklaration mit dem `New`-Operator. Er verleiht Ihnen eine bessere Kontrolle über die Lebenszeit eines Objekts.

Nebenbei werden Sie bemerkt haben, daß das Objekt jedesmal beim Verlassen der Prozedur zerstört wird. Dies ist so, weil die Variable A in der Funktion lokal deklariert ist und daher die Lebenszeit der Objekt-Variable auf den Funktions-Aufruf begrenzt ist. Wird die Funktion verlassen, wird die Variable freigegeben und damit auch das Objekt, auf das die Variable verweist.

Verifizieren wir nun die Operation aus Abbildung 13.2 anhand des Codes der Ereignis-Prozedur `cmdStart5_Click`:

```
Private Sub cmdStart5_Click()
    Dim A As myobject
    Dim B As myobject
    Set A = New myobject
    Set B = New myobject
    Debug.Print "A and B are set"
    A.Name = "Object A"
    B.Name = "Object B"
    Debug.Print "Name properties are set"
    Debug.Print "About to Set A=B"
    Set A = B
    Debug.Print "Set A=B done"
End Sub
```

Dieser Code führt zu folgenden Ausgaben im Direkt-Fenster:

```
Object is created
Object is created
A and B are set
Name properties are set
About to Set A=B
Object ObjectA is destroyed
Set A=B done
Object ObjectB is destroyed
```

Wie Sie sehen, führt die Operation `Set A = B` tatsächlich dazu, daß Objekt `A` zerstört wird.

Das Geheimnis, sicherzustellen, daß Objekte tatsächlich zerstört werden, liegt in dem Wissen, wann und wo sie referenziert werden. Aber das ist leicht gesagt ...

13.2 Zirkuläre Referenzen

Eine zirkuläre Referenz ist schnell demonstriert. Fügen Sie einfach folgende Zeile in den Allgemein-Teil der Klasse `myobject` ein (im Beispiel-Projekt `test1.vbp` ist dies bereits erledigt):

```
Public CircularReference As myobject
```

In der Funktion `cmdStart6_Click` wird diese Eigenschaft benutzt:

```
Private Sub cmdStart6_Click()
    Dim A As myobject
    Set A = New myobject
    Debug.Print "About to set circular reference"
    Set A.CircularReference = A
    Debug.Print "About to set A to Nothing"
    Set A = Nothing
    Debug.Print "A is Nothing"
End Sub
```

Im Direkt-Fenster wird folgendes ausgegeben, wenn diese Funktion ausgeführt wird:

```
Object is created
About to set circular reference
About to set A to Nothing
A is Nothing
```

Warum wird das Objekt nicht zerstört?

Nach der Ausführung der Operation `Set A.CircularReference = A` zeigt eine zweite Variable auf das Objekt. Dabei ist irrelevant, daß diese Variable Bestandteil des Objekts selbst ist. Wesentlich ist, daß der Referenzzähler des Objekts nach wie vor auf 2 steht.

Wird die Variable `A` gleich `Nothing` gesetzt, wird der Referenzzähler um 1 erniedrigt. Doch das reicht jetzt nicht mehr aus, damit das Objekt zerstört wird. Das Objekt lebt weiterhin, auch wenn auf es von keiner Stelle in Ihrem Programm mehr zugegriffen werden kann (alle Variablen, die das Objekt referenzieren, jedoch nicht Bestandteil des Objekts selbst sind, sind gelöscht worden). Das Objekt wird so lange existieren, bis Sie die Anwendung beenden. Schließen Sie das Formular `frmTest1` (benutzen Sie nicht den Visual-Basic-Stop-Befehl!) – Sie sehen, daß das Objekt mit dem Beenden der Anwendung zerstört wird.

Natürlich wird es wohl eher nur versehentlich vorkommen, daß ein Objekt sich selbst referenziert. Wann also betrifft uns das Problem der zirkulären Referenzen? Werfen wir noch einmal einen Blick auf unser Kaninchenzucht-Programm aus Kapitel 12.

13.2.1 Das Dilemma von Objekt-Modellen

Die Beispiele zur Kaninchenzucht dienten zur Demonstration der Gruppenbildung von Objekten. Wir haben uns dabei noch nicht um den Entwurf eines brauchbaren Objekt-Modells gekümmert. Zirkuläre Referenzen treten nicht zufällig auf – sie sind entwurfsbedingt. Betrachten wir nun unsere Kaninchen-Beispiele aus dem Blickwinkel eines Objekt-Modells. Zur Vereinfachung beginnen wir mit dem (Collection-basierten) Projekt `Rabbit3.vbp`, da die Technik der Gruppenbildung für dieses Beispiel ohne Bedeutung ist. Alle Dateien des neuen Beispiels tragen nun die Numerierung 5.

Die Klasse Kaninchen (nun `clsRabbit5` genannt) kann vorerst unverändert bleiben. Jedes Kaninchen hat eine eindeutige Nummer und eine Farbe. Die Anzahl der möglichen Farben ist niedrig gehalten, damit das Beispiel-Programm nicht zu kompliziert wird. Über die `Debug.Print`-Anweisungen können wir das Entstehen und den Verkauf von Kaninchen verfolgen.

Nebenbei gesagt: Wenn in diesen Beispielen vom »Verkauf« eines Kaninchens die Rede ist, so ist dies eine harmlose Umschreibung des Umstands, daß die Kaninchen dabei zerstört werden – vom »Umbringen« von Kaninchen zu sprechen erschien mir doch ein wenig zu heftig für ein familiengerechtes Programmierbuch ...

Die Klasse `RabbitCollection5` enthält einen Käfig (hutch) für Kaninchen – eine Gruppe von `clsRabbit5`-Objekten. Nun legen wir die erste Regel des Objekt-Modells fest: Jedes Kaninchen kann nur in einem einzigen `RabbitCollection5`-Objekt zur gleichen Zeit enthalten sein. Dies macht Sinn – ein Wunderkaninchen, das sich in mehreren Käfigen zugleich tummeln kann, hat die Welt noch nicht gesehen (Sie könnten sicher ein Vermögen damit verdienen ...). Bis jetzt existiert

jedoch noch kein Code, der die Einhaltung dieser Regel erzwingt – bis dato ist es lediglich eine Entwurfsforderung. Bei der Implementierung des Codes soll jede Kombination von Operationen als illegal (oder sogar als Bug) gelten, wenn sie dazu führen sollte, daß sich ein Kaninchen zugleich in zwei (oder mehreren) `RabbitCollection5`-Collections befindet.

Wir wollen eine Methode einführen, mit der Kaninchen von einer `RabbitCollection5`-Collection in eine andere transportiert werden können. Dazu muß das Kaninchen aus der einen `RabbitCollection5`-Collection herausgenommen werden, zum Verkauf oder zum Transport in eine andere Collection. Wir brauchen dazu ebenfalls einen Weg, eine Liste von Kaninchen eines gewünschten Typs zu erhalten. Diese Liste wird als Standard-Collection anstatt als `RabbitCollection5` geliefert. Warum? Wenn wir eine `RabbitCollection5` liefern wollten, müßten wir die obengenannte Regel verletzen, daß sich kein Kaninchen zur gleichen Zeit in mehreren `RabbitCollection5`-Collections befinden darf.

Listing 13.1 enthält die anfängliche Implementierung der `RabbitCollection5`-Klasse.

```
' Guide to the Perplexed - Rabbit Test
' Copyright (c) 1997 by Desaware Inc. all Rights Reserved

Option Explicit

'Lokale Variable für Collection
Private m_Hutch As Collection

' Vorhandenes Kaninchen einfügen oder neues anlegen
Public Sub Add(Optional obj As clsRabbit5)
    ' Wird kein Objekt übergeben, wird ein neues
    ' angelegt
    If obj Is Nothing Then Set obj = New clsRabbit5
    m_Hutch.Add obj
End Sub

Public Property Get Count() As Long
    Count = m_Hutch.Count
End Property

Public Property Get Item(IndexKey As Long) As clsRabbit3
    Set Item = m_Hutch(IndexKey)
End Property

Public Sub Remove(IndexKey As Long)
    m_Hutch.Remove IndexKey
End Sub
```

```
' Index eines Objekts in der Collection herausfinden
Public Function Find(findobj As clsRabbit5)
    Dim obj As clsRabbit5
    Dim counter&
    For counter = 1 To m_Hutch.Count
        If m_Hutch(counter) Is findobj Then
            Find = counter
            Exit Function
        End If
    Next counter
End Function

' For...Each ermöglichen
Public Property Get NewEnum() As IUnknown
    Set NewEnum = m_Hutch.[_NewEnum]
End Property

' Interne Collection initialisieren und zerstören

Private Sub Class_Initialize()
    Set m_Hutch = New Collection
End Sub

Private Sub Class_Terminate()
    Set m_Hutch = Nothing
End Sub

' Methode für neue Collection nur mit weißen Kaninchen
Public Function GetWhiteRabbits() As Collection
    Dim col As New Collection
    Dim obj As clsRabbit3
    For Each obj In m_Hutch
        If obj.Color = "White" Then
            col.Add obj
        End If
    Next
    Set GetWhiteRabbits = col
End Function
```

Listing 13.1.: Das Klassen-Modul RabbitCollection5

Es gibt nur wenige Unterschiede zur Implementierung der ursprünglichen Klasse `RabbitCollection3` aus Kapitel 12. Die `Add`-Methode erlaubt nun das Hinzufügen eines vorhandenen Kaninchens. Wir benötigen sie für den Transport eines Kaninchens von einer Collection in eine andere. Warum verwenden wir nicht wieder die Methode `AddExisting` wie in Kapitel 12? Es gibt keinen besonderen Grund dafür – es stellt nur eine weitere Möglichkeit dar.

Über die neue Methode `Find` kann der Index eines `clsRabbit5`-Objekts in der Collection festgestellt werden. Die Methode `GetWhiteRabbits` liefert eine Collection mit allen weißen Kaninchen in der `RabbitCollection5`.

In der Anwendung `RabbitTest5` (Projekt `RbtTest5.vbp`) wird die Verwendung dieser modifizierten Klasse demonstriert. Das Hauptformular der Anwendung sehen Sie in Abbildung 13.3 und den Code des Formulars in Listing 13.2.

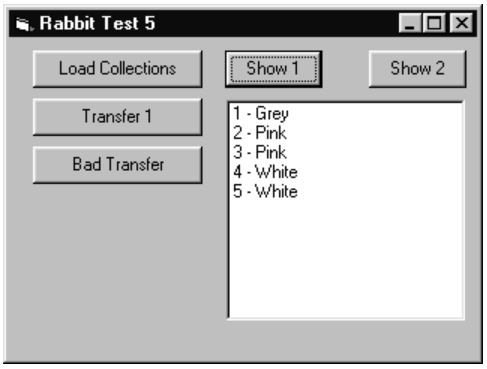


Abb. 13.3: Hauptformular von RabbitTest5 in Aktion

```
' Guide to the Perplexed - Rabbit5 example
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

' Zwei Käfige deklarieren
Dim Hutch1 As RabbitCollection5
Dim Hutch2 As RabbitCollection5
Dim PetStore As New PetStore5

' die zwei Hutch-Variablen füllen
Private Sub cmdLoad_Click()
    Set Hutch1 = PetStore.BuyRabbits(5)
    Set Hutch2 = PetStore.BuyRabbits(5)
End Sub

' Inhalt einer Hutch-Variablen anzeigen
Private Sub cmdShow_Click(Index As Integer)
    Dim obj As clsRabbit5
    Dim UseHutch As RabbitCollection5
    ' Den anzuzeigenden Käfig ermitteln
    If Index = 1 Then
        Set UseHutch = Hutch1
    Else
```

```
            Set UseHutch = Hutch2
        End If
        lstRabbits.Clear
        For Each obj In UseHutch
            lstRabbits.AddItem obj.Number & " - " _
             & obj.Color
        Next
    End Sub

    ' Erstes Kaninchen von Hutch1 nach Hutch2 transportieren
    Private Sub cmdTransfer1_Click()
        Dim obj As clsRabbit5
        Set obj = Hutch1(1)
        Hutch1.Remove 1
        Hutch2.Add obj
    End Sub
```

Listing 13.2: Code des Formulars frmRabbitTest5

Über die Schaltfläche LOAD COLLECTIONS werden die beiden Hutch-Variablen mit jeweils 5 Kaninchen gefüllt. Die Schaltflächen SHOW 1 und SHOW 2 zeigen den Inhalt der entsprechenden Hutch-Variablen in der Listbox. Die Schaltfläche TRANSFER1 verschiebt das erste Kaninchen aus `Hutch1` in die `Hutch2`-Collection. Sie können sich das Resultat dieses Codes über die SHOW-Schaltflächen ansehen.

Auch wenn dieser Code so funktioniert, ist jedoch der Widerspruch zum Objekt-Modell klar.

Wir haben die Regel vorgegeben, daß sich kein Kaninchen in mehr als einer `RabbitCollection5` zur gleichen Zeit befinden darf. Nichts in dem vorhandenen Objekt-Modell erzwingt jedoch die Einhaltung dieser Regel. Vergäßen Sie, ein Kaninchen aus der einen Hutch-Collection zu entfernen, bevor Sie es in eine andere einfügen, erschiene es in beiden Collections zur gleichen Zeit. Noch schlimmer ist, daß Sie das gleiche Kaninchen sogar mehrfach in ein und dieselbe Hutch-Collection einfügen könnten. Sie können eine derartige Operation in der Prozedur `cmdBadTransfer_Click` im `RabbitTest5`-Beispiel sehen. In der ersten Ausgabe dieses Buches habe ich noch angemerkt, daß eine solche Technik des Clonings auf softwaretechnischem Wege eine lustige Sache, aber noch Zukunftsmusik in biologischer Hinsicht wäre. Doch die verhältnismäßig kurze Zeit, die seitdem verstrichen ist, zeigt deutlich, daß nicht nur unsere Branche in einem rasanten Fortschritt begriffen ist. Nichtsdestotrotz bedeutet das Fehlen der erzwungenen Einhaltung von für ein Objekt-Modell aufgestellten Regeln einen schlechten Entwurf – auch wenn das so bei sorgfältiger Anwendung problemlos funktionieren kann.

13.2.2 Erzwungene Einhaltung von Objekt-Modell-Regeln

Wir fahren nun in der Demonstration mit der Projektgruppe `Rabbit6` fort, die das Server-Projekt `Rabbit6` und das Test-Programm `Rabbittest6` (Projekt `RbtTest6.vbp`) enthält.

Das Erzwingen der Einhaltung der Regel, daß ein `clsRabbit6`-Objekt nur in einer einzigen `RabbitCollection6`-Collection zur gleichen Zeit enthalten sein kann, erfordert, daß der Server alle existierenden `RabbitCollection6`-Collections kennt. Auf anderem Wege kann nicht festgestellt werden, welche Collection welche `clsRabbit6`-Objekte enthält. Wie können wir dies erreichen? Dafür gibt es zwei mögliche Ansätze:

- Nachhalten einer Liste aller `RabbitCollection6`-Objekte auf globaler oder höherer Ebene.
- Jedes `clsRabbit6`-Objekt merkt sich, in welcher `RabbitCollection6`-Collection es enthalten ist.

Wir lassen den globalen Ansatz vorläufig außer acht. Ein Beispiel für diesen Ansatz finden Sie in der StockMonitor-Anwendung in Kapitel 15 (die habe ich nicht vergessen!).

Zudem weist der Ansatz, daß jedes `clsRabbit6`-Objekt weiß, wo es sich befindet, eine gewisse logische Eleganz auf. Vom konzeptionellen Standpunkt aus gesehen, ist es sehr sinnvoll, daß ein Objekt weiß, wo (in welchem Container) es sich befindet, wie auch umgekehrt jeder Behälter (Container) wissen sollte, was sich in ihm befindet. Als Visual-Basic-Programmierer sollten Sie dieses Konzept längst kennen, auch wenn Sie sich dessen im Moment vielleicht nicht bewußt sind. Visual-Basic-Controls verfügen über die `Parent`-Eigenschaft, über die sie auf die Methoden und Eigenschaften des Formulars oder des Controls zugreifen können, in dem sie enthalten sind. Dies ist genau die gleiche Situation (Sie können 100%ig davon ausgehen, daß sich Visual Basic im Hintergrund alle Mühe gibt, die Seiteneffekte zu vermeiden, die wir noch sehen werden).

Den Container merken

Bei diesem Ansatz merkt sich jedes `clsRabbit6`-Objekt die `RabbitCollection6`-Collection, in der es sich befindet. Dies erfolgt über eine `private` Objekt-Variable in der `clsRabbit6`-Klasse und über die Offenlegung einer Eigenschaft (= zwei Property-Prozeduren):

```
Private m_Hutch As RabbitCollection6

' Jeder kann feststellen, in welchem Käfig sich das
' Kaninchen befindet
Public Property Get Hutch() As RabbitCollection6
    Set Hutch = m_Hutch
End Property
```

```
' Nur per Code innerhalb des Projekts kann dem Kaninchen
' ein Käfig zugewiesen werden
Friend Property Let Hutch(vNewValue _
 As RabbitCollection6)
    Set m_Hutch = vNewValue
End Property
```

Die `Property Let`-Prozedur ist als `Friend` deklariert. Clients, die unser Objekt-Modell verwenden, können jederzeit problemlos abfragen, in welchem Käfig sich ein Kaninchen befindet. Doch sollte kein Client die Zuweisung zu einem Käfig direkt, und ohne den Weg über eine entsprechende Methode des `RabbitCollection6`-Objekts zu gehen, vornehmen können. Auf diese Weise kann das `RabbitCollection6`-Objekt die Einhaltung der Objekt-Modell-Regel erzwingen. Die entsprechende Modifikation der Klasse `RabbitCollection6` sehen Sie in Listing 13.3.

```
' Guide to the Perplexed - Rabbit Test
' Copyright (c) 1997 by Desaware Inc. all Rights Reserved

Option Explicit

' Lokale Variable für Collection
Private m_Hutch As Collection

' Vorhandenes oder neues Kaninchen hinzufügen
Public Sub Add(Optional obj As clsRabbit6)
    Dim IndexToRemove As Long

    ' Wird kein Objekt übergeben, dann neues anlegen
    If obj Is Nothing Then
        Set obj = New clsRabbit6
    Else
        If Find(obj) > 0 Then
            RaiseError 0    ' Fehler: Objekt existiert
        End If
    End If

    m_Hutch.Add obj

    ' Käfig festlegen
    If Not obj.Hutch Is Nothing Then
        ' Wenn hinzugefügtes Objekt bereits in einem
        ' Käfig enthalten ist, wird es aus diesem
        ' entfernt
        IndexToRemove = obj.Hutch.Find(obj)
        obj.Hutch.Remove IndexToRemove
    End If
    ' Objekt verweist immer auf diesen Käfig
```

```
    obj.Hutch = Me
End Sub

Public Property Get Count() As Long
    Count = m_Hutch.Count
End Property

Public Property Get Item(IndexKey As Long) As clsRabbit6
    If IndexKey < 1 Or IndexKey > m_Hutch.Count Then
        RaiseError 1
    End If
    Set Item = m_Hutch(IndexKey)
End Property

' Objekt aus diesem Käfig entfernen
Public Sub Remove(IndexKey As Long)
    Dim obj As clsRabbit6
    If IndexKey < 1 Or IndexKey > m_Hutch.Count Then
        RaiseError 1
    End If
    Set obj = m_Hutch(IndexKey)
    ' Referenz auf diesen Käfig löschen
    obj.Hutch = Nothing
    m_Hutch.Remove IndexKey
End Sub

' Index eines Objekt in Collection ermitteln
Public Function Find(findobj As clsRabbit6)
    Dim obj As clsRabbit6
    Dim counter&
    For counter = 1 To m_Hutch.Count
        If m_Hutch(counter) Is findobj Then
            Find = counter
            Exit Function
        End If
    Next counter
End Function

' For...Each ermöglichen
Public Property Get NewEnum() As IUnknown
    Set NewEnum = m_Hutch.[_NewEnum]
End Property

' Initialisieren und Zerstören der internen Collection

Private Sub Class_Initialize()
    Set m_Hutch = New Collection
End Sub
```

```
Private Sub Class_Terminate()
    Set m_Hutch = Nothing
End Sub

' Neue Collection nur mit den weißen Kaninchen
Public Function GetWhiteRabbits() As Collection
    Dim col As New Collection
    Dim obj As clsRabbit6
    For Each obj In m_Hutch
        If obj.Color = "White" Then
            col.Add obj
        End If
    Next
    Set GetWhiteRabbits = col
End Function

' Zentralisierte Fehlerbehandlung
' 0 = Versuch, existierendes Objekt hinzuzufügen

Public Sub RaiseError(erroffset As Integer)
    Dim e$
    Select Case erroffset
        Case 0
            e$ = "Object already exists in collection"
        Case 1
            e$ = "Invalid collection index"
    End Select
    Err.Raise vbObjectError + 1000 + erroffset, _
     "RabbitCollection6", e$
End Sub
```

Listing 13.3: Das Klassen-Modul RabbitCollection6

Die umfangreichsten Änderungen betreffen die Methoden `Add` und `Remove`. Zusätzlich wurde eine Fehlerprüfung zur `Item`-Methode hinzugefügt.

Die Methode `Add` enthält nun einen Test zur Sicherstellung, daß Sie nicht versuchen, ein bereits in der Collection vorhandenes Objekt erneut hinzuzufügen. Über die Methode `Find` wird ermittelt, ob das hinzuzufügende Objekt bereits in der Liste enthalten ist. Wird es gefunden, wird ein Fehler ausgelöst. Die `Add`-Methode schaut die Eigenschaft `Hutch` des hinzuzufügenden Objekts nach, ob dort bereits auf ein `RabbitCollection6`-Objekt verwiesen wird. Ist dies der Fall, wird das Objekt aus der anderen Collection entfernt. Dann wird die `Hutch`-Eigenschaft auf die aktuelle, neue Collection gesetzt. Die `Remove`-Methode muß die `Hutch`-Eigenschaft beim zu entfernenden Objekt auf `Nothing` setzen, bevor dieses Objekt tatsächlich aus der betreffenden Collection entfernt wird.

Das Test-Projekt RabbitTest6 ähnelt dem Test-Projekt RabbitTest5, nur daß hier die beiden Transfer-Tests anders aussehen:

```
' Verschieben des ersten Kaninchens von hutch1 nach
' hutch2
Private Sub cmdTransfer1_Click()
    Dim obj As clsRabbit6
    Set obj = Hutch1(1)
    Hutch2.Add obj
End Sub

' Diesmal wird die Einhaltung der Regel erzwungen
Private Sub cmdBadTransfer_Click()
    Dim obj As clsRabbit6
    Set obj = Hutch1(1)
    Hutch1.Add obj
End Sub
```

Die Prozedur cmdTransfer1_Click funktioniert einwandfrei, auch wenn Sie das Objekt nicht ausdrücklich aus Hutch1 entfernen. Die Methode Hutch2.Add entfernt das Kaninchen automatisch aus der Hutch1-Collection. In der Prozedur cmdBadTransfer_Click wird dagegen ein Fehler ausgelöst, wenn Sie versuchen, ein Objekt ein zweites Mal der Collection hinzuzufügen. Scheint perfekt zu klappen, nicht wahr? Es gibt jedoch noch ein Problem.

Es liegt eine zirkuläre Referenz vor

Gehen Sie noch einmal zurück zur Projektgruppe Rabbit5. Klicken Sie auf die Schaltfläche LOAD COLLECTIONS und beobachten Sie anhand der Ausgaben im Direkt-Fenster, wie die 10 Kaninchen angelegt werden. Klicken Sie erneut auf die Schaltfläche LOAD COLLECTIONS. Sie werden sehen, wie die nächsten 10 Kaninchen angelegt werden, wobei die neuen Collections als Ersatz der alten angelegt werden. Sie werden auch sehen, wie die alten 10 clsRabbit5-Objekte zerstört werden, wenn die alten Collections zerstört werden.

Probieren Sie nun dasselbe in der Projektgruppe Rabbit6. Wenn Sie die neuen Collections laden, werden weder die Kaninchen noch die Collections zerstört! Betrachten wir einmal genauer, was hier vor sich geht.

In Abbildung 13.4 sehen Sie die Referenzen für eine der RabbitCollection5-Collections. Der Wert des Referenzzählers eines Objekts wird durch die Anzahl der auf ein Objekt zeigenden Pfeile dargestellt. Was passiert, wenn Sie die Hutch1-Variable auf Nothing oder auf eine andere Collection setzen? Der Referenzzähler des RabbitCollection5-Objekts wird auf 0 gesetzt und es terminiert. Da dieses Objekt terminiert, werden auch seine privaten Variablen zerstört, einschließlich der m_Hutch-Collection. Wenn diese Collection zerstört wird, hört sie damit auch auf, auf die in ihr enthaltenen Objekte zu verweisen. Nacheinander

werden die Referenzzähler der `clsRabbit5`-Objekte um eins erniedrigt und hier folglich auf 0 gesetzt. Die `clsRabbit5`-Objekte werden daher ebenfalls der Reihe nach zerstört.

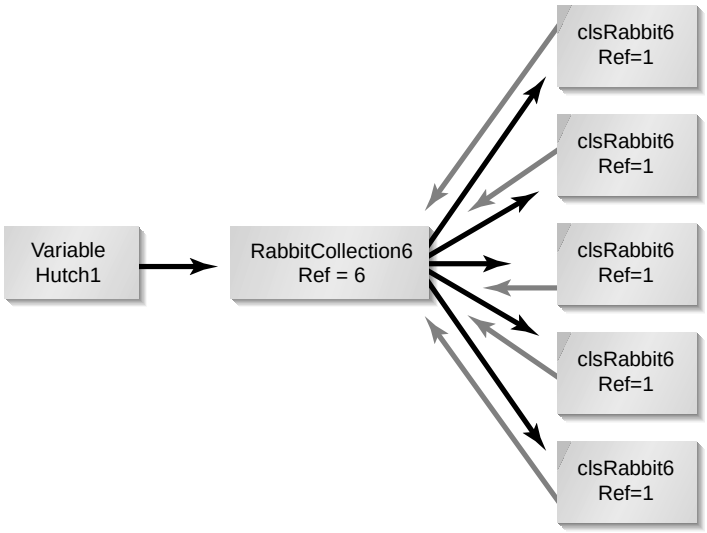


Abb. 13.4: Referenzierungen im Beispiel Rabbit5

In Abbildung 13.5 werden die Referenzierungen der `RabbitCollection6`-Collections dargestellt. Wie Sie sehen, enthält jedes `clsRabbit6`-Objekt in einer Collection eine Referenz zurück auf das Collection-Objekt. Der Referenzzähler des Collection-Objekts beträgt also 6. Wenn die `Hutch1`-Variable auf `Nothing` oder auf eine andere Collection gesetzt wird, wird der Referenzzähler des `RabbitCollection6`-Objekts um 1 erniedrigt – er beträgt also nun 5. Das bedeutet, daß das Objekt nicht zerstört wird, also auch nicht das `Terminate`-Ereignis ausgelöst wird. Und das bedeutet folglich auch, daß keines der vom `RabbitCollection6`-Projekt referenzierten Objekte zerstört wird, bevor die Anwendung beendet wird. Sie müssen erst zuvor alle `clsRabbit6`-Objekte aus der Collection entfernt haben!

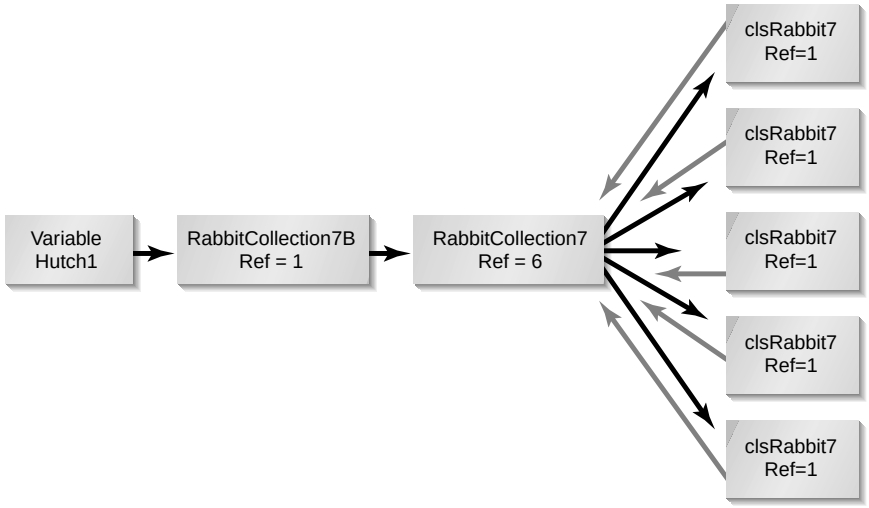


Abb. 13.5: Referenzierungen im Beispiel Rabbit6

13.2.3 Der Umgang mit zirkulären Referenzen

Genaugenommen gibt es nur drei Möglichkeiten, mit dem Problem zirkulärer Referenzen umzugehen:

- Sie können von den Clients fordern, Objekt-Referenzen explizit freizugeben.
- Sie können sie aus Ihrem Objekt-Modell verbannen.
- Sie können den Referenzen-Zirkel durchbrechen.

Explizite Freigabe von Objekt-Referenzen

Die erste Freigabe-Option ist offensichtlich. Der folgende Code wird nach dem Anklicken der Schaltfläche SAFE LOAD COLLECTION ausgeführt:

```
' Eine Möglichkeit, Hutch-Variablen zu laden, bei der die
' vorhergehende erst gelöscht wird
    Private Sub cmdSafe_Click()
        If Not Hutch1 Is Nothing Then
            Do While Hutch1.Count > 0
                Hutch1.Remove 1
            Loop
        End If
        Set Hutch1 = PetStore.BuyRabbits(5)
        If Not Hutch2 Is Nothing Then
            Do While Hutch2.Count > 0
                Hutch2.Remove 1
            Loop
```

```
    End If
    Set Hutch2 = PetStore.BuyRabbits(5)
End Sub
```

Indem vor der neuen Zuweisung alle `clsRabbit6`-Objekte aus der Collection entfernt werden, wird der Referenzzähler der Collection bis auf 1 erniedrigt, und schließlich auf 0, wenn der `Hutch1`-Variablen eine andere Collection zugewiesen wird.

Am Rande bemerkt: Denken Sie daran, daß in diesem Beispiel das Entfernen eines `clsRabbit6`-Objekt dieses zerstört, weil es hier nur von der Collection referenziert wird. Falls Sie noch an anderer Stelle einer anderen Variablen eine Referenz auf das Objekt zugewiesen haben sollten (so etwa in der Collection, die Sie mit dem Aufruf der `GetWhiteRabbits`-Methode erhalten haben könnten), wird das Objekt weiterhin von dort aus referenziert und nicht eher zerstört, bevor nicht auch diese Referenz freigegeben worden ist.

Diese erste Möglichkeit ist auch zugleich die einfachste. Sie ist nicht gerade sonderlich elegant, da sie die ausdrückliche Mitarbeit des Clients erfordert. Wahrscheinlich wird ihre erste Reaktion sein, sie für die schlechteste der Möglichkeiten zu halten.

Ganz allgemein ist dies auch die gesündeste Reaktion. Es entspricht tatsächlich einem schlechten Entwurf, Objekte offenzulegen, die gesonderte Operationen zum Terminieren erfordern. Bei einem öffentlichen Objekt wie `RabbitCollection6` ist es ein schlechter Stil. Innerhalb eines Projekts mag diese Technik jedoch angebracht sein. Wenn Sie eine Klasse innerhalb eines Projekts auf wohldefinierte Weise und diszipliniert verwenden, spricht nichts dagegen, diesen am einfachsten zu implementierenden Ansatz zu wählen.

Dieser Ansatz liegt auch dem folgenden zugrunde, bei dem wir ihn mit einem Neuentwurf des Objekt-Modells kombinieren, um die Probleme zumindest aus der Sicht eines Clients zu eliminieren.

Konzept des Objekt-Modells ändern

Die Projektgruppe `Rabbit7` demonstriert, wie Sie nach ein paar kleineren Änderungen zu einem Objekt-Modell gelangen, das das Problem der zirkulären Referenzen zumindest aus der Sicht von Client-Anwendungen löst. Die Numerierung der Dateien der Projektgruppe hat sich nunmehr von 6 auf 7 erhöht. Der Trick besteht in diesem Fall darin, die Klasse `RabbitCollection7` so zu modifizieren, daß sie lediglich intern verwendet wird, und darin, eine neue Klasse `RabbitCollection7B` zu definieren, die als öffentliches Objekt für Client-Anwendungen eine Collection von `clsRabbit7`-Objekt repräsentiert.

Das Klassen-Modul `RabbitCollection7B` (siehe Listing 13.4) enthält ein einzelnes `RabbitCollection7`-Objekt und bildet dessen Methoden und Eigenschaften per Delegation nach außen hin ab. Denken Sie daran, in den Prozedur-Attributen

die Eigenschaft `Item` als Standard-Eigenschaft (Voreinstellung) festzulegen und die Prozedur-ID (Dispatch-ID) der Eigenschaft `NewEnum` auf -4 zu setzen und diese zu verbergen.

```
' Guide to the Perplexed:
' Rabbit7 example
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

Private m_Internal As RabbitCollection7

' Alle Methoden an interne Collection delegieren
Public Sub Add(Optional obj As clsRabbit7)
    m_Internal.Add obj
End Sub

Public Property Get Count() As Long
    Count = m_Internal.Count
End Property

Public Property Get Item(IndexKey As Long) As clsRabbit7
    Set Item = m_Internal.Item(IndexKey)
End Property

Public Sub Remove(IndexKey As Long)
    m_Internal.Remove IndexKey
End Sub

Public Function Find(findobj As clsRabbit7)
    Find = m_Internal.Find(findobj)
End Function

Public Property Get NewEnum() As IUnknown
    Set NewEnum = m_Internal.NewEnum
End Property

Public Function GetWhiteRabbits() As Collection
    Set GetWhiteRabbits = m_Internal.GetWhiteRabbits
End Function

Private Sub Class_Initialize()
    Set m_Internal = New RabbitCollection7
End Sub

' Beim Terminieren dieses Objekts wird das enthaltene
' Objekt ebenfalls sauber terminiert
```

```
Private Sub Class_Terminate()
    Do While m_Internal.Count > 0
        m_Internal.Remove 1
    Loop
End Sub
```

Listing 13.4: Das Klassen-Modul RabbitCollection7B

Alle Referenzen auf Rabbit-Collection-Objekte im `PetStore`-Modul und im `RabbitTest7`-Projekt werden geändert und verweisen nun auf `RabbitCollection7B`. Im `clsRabbit7`-Klassen-Modul verweisen jedoch die Referenzen nach wie vor auf das `RabbitCollection7`-Objekt. Das bedeutet, daß die `Hutch`-Eigenschaft nicht länger öffentlich sein kann. Wäre Sie öffentlich, würde das `RabbitCollection7`-Objekt offengelegt, das wir nun jedoch nur intern verwenden wollen. Die Deklaration der `Hutch`-Eigenschaft wurde also in `Friend` geändert.

Die Hauptarbeit wird in den Ereignissen `Class_Initialize` und `Class_Terminate` der Klasse `RabbitCollection7B` erledigt. Im `Initialize`-Ereignis wird das interne Objekt angelegt. Im Terminate-Ereignis werden alle `clsRabbit7`-Objekte aus der eingebetteten Collection entfernt, damit diese sauber terminieren können.

Woher können wir nun wissen, daß das `RabbitCollection7B`-Objekt ebenfalls sauber terminiert? In Abbildung 13.6 sehen Sie die Referenzierungen für dieses Objekt. Während das Objekt `RabbitCollection7` weiterhin mehrfach referenziert wird, unterbleibt dies beim Objekt `RabbitCollection7B`. Wenn die Variable `Hutch1` erneut zugewiesen wird, wird das Objekt `RabbitCollection7B` terminieren. Dabei wird das eingebaute `RabbitCollection7`-Objekt ordnungsgemäß aufgeräumt.

Dieser Ansatz erfordert noch eine kleinere Änderung im `RabbitCollection7`-Objekt. Die `Count`-Eigenschaft wurde wie folgt modifiziert:

```
Public Property Get Count() As Long
    If Not m_Hutch Is Nothing Then
        Count = m_Hutch.Count
    End If
End Property
```

Dies ist notwendig, weil Sie nicht wissen können, in welcher Reihenfolge die Objekte beim Terminieren der Anwendung selbst zerstört werden. Was geschieht, wenn das eingebettete `RabbitCollection7`-Objekt bereits vor dem `RabbitCollection7B`-Objekt, in dem es ja enthalten ist, terminiert hat? Dort wird im `Terminate`-Ereignis die Variable `m_Hutch` auf `Nothing` gesetzt. Wird das `Terminate`-Ereignis des `RabbitCollection7B`-Objekts ausgeführt, wird die `Count`-Eigenschaft des `RabbitCollection7`-Objekts ausgeführt, die wiederum auf die längst nicht mehr gültige Variable `m_Hutch` zuzugreifen versucht. Dies

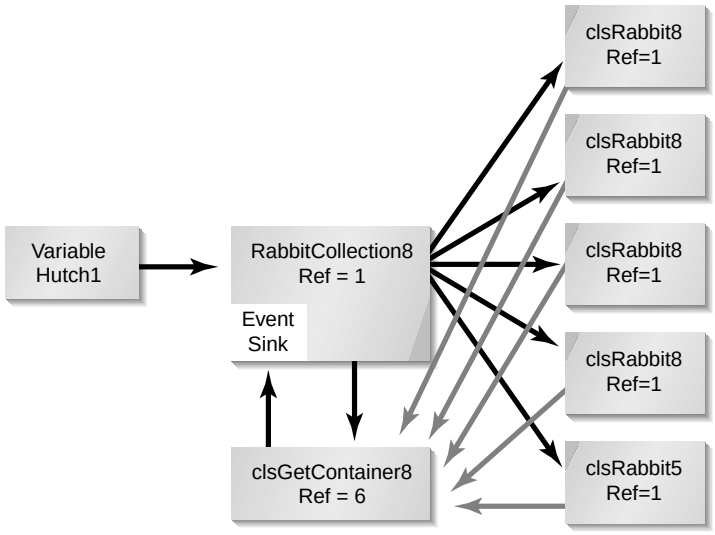


Abb. 13.6: Referenzierungen im Beispiel Rabbit7

wirft die Frage auf, wie denn ein Objekt Methoden eines anderen Objekts aufrufen kann, dessen `Terminate`-Ereignis bereits ausgeführt worden ist.

Die automatische Terminierung von Objekten beim Beenden einer Anwendung ist ein spezieller Fall, bei dem die Reihenfolge der Terminierungen unbestimmt ist. Visual Basic versucht, alle Objekte aufzuräumen, die übriggeblieben sind. Dieses Problem würde nicht auftreten, wenn die Container-Anwendung so nett wäre, ihre `Hutch1`- und `Hutch2`-Variablen freizugeben, wenn das Formular terminiert:

```
Private Sub Form_Terminate()
    Set Hutch1 = Nothing
    Set Hutch2 = Nothing
End Sub
```

Sie können jedoch nicht voraussetzen, daß die Clients Ihrer Anwendung selbst ordentlich bei sich aufräumen. Sie sollten Ihre Anwendungen also auch in Situationen testen, in denen beim Beenden noch Objekte von einer Client-Anwendung referenziert werden.

Den Referenzen-Zirkel durchbrechen

Seit Visual Basic 5 steht eine interessante neue Technik zur Verfügung, zirkuläre Referenzen zu durchbrechen. Zum Durchbrechen von zirkulären Referenzen können Sie die Fähigkeit von Visual Basic nutzen, Ereignisse auszulösen. Dies machen wir in der Projektgruppe `Rabbit8` – hier tragen alle Dateien die Nummer 8.

Wir definieren eine neue Klasse namens `clsGetContainer8`, die als Empfangsobjekt dient und auf die die `clsRabbit`-Objekte anstelle des `RabbitCollection8`-Objekts verweisen können. Die `clsRabbit8`-Objekte mit einem anderen Objekt zu verbinden, ist eine offensichtliche und leicht zu implementierende Lösung. Die Frage lautet nun: Wie kann dieses Objekt Referenzen auf den Container erhalten? Die Antwort lautet, ein Ereignis zu verwenden. Das Objekt `clsGetContainer8` enthält folgenden Code:

```
' Guide to the Perplexed - Rabbit Test
' Copyright (c) 1997 by Desaware Inc. all Rights Reserved

Option Explicit

Public Event ContainerRequest(ContainerObject As Object)

' Diese Methode löst ein Ereignis im Container aus, das
' eine Objekt-Referenz auf den Container erfragt
Public Function GetContainer() As Object
    Dim obj As Object
    RaiseEvent ContainerRequest(obj)
    Set GetContainer = obj
End Function
```

Jedes `RabbitCollection8`-Objekt enthält ein eigenes `clsGetContainer8`-Objekt und kann auf das `ContainerRequest`-Ereignis reagieren. Wenn ein `clsGetContainer8`-Objekt eine Objekt-Referenz zu seinem Container benötigt, löst es dieses Ereignis aus. Der Code in der Ereignis-Prozedur setzt dann den Parameter Container-Objekt auf sich selbst.

Das Unterobjekt `clsGetContainer8` ist wie folgt deklariert:

```
Private WithEvents ContainerSubObject As clsGetContainer8
```

Das Objekt wird im Ereignis `Class_Initialize` des `RabbitCollection8`-Objekts angelegt:

```
Set ContainerSubObject = New clsGetContainer8
```

Der Ereignis-Code sieht so aus:

```
Private Sub ContainerSubObject_ContainerRequest( _
 ContainerObject As Object)
    Set ContainerObject = Me
End Sub
```

Ein `clsRabbit8`-Objekt benötigt nun nur noch eine Verbindung zu diesem Unterobjekt anstatt zum Container selbst. Über die folgende Funktion erhält es diese Verbindung:

```
Friend Function GetContainerSubObject() _
 As clsGetContainer8
    Set GetContainerSubObject = ContainerSubObject
End Function
```

Im clsRabbit8-Objekt ist die Hutch-Eigenschaft wie folgt deklariert:

```
' In welchem Käfig ist das Kaninchen enthalten?
Private m_Hutch As clsGetContainer8

' Jeder kann ermitteln, in welchem Käfig sich das
' Kaninchen befindet
Public Property Get Hutch() As RabbitCollection8
    If m_Hutch Is Nothing Then Exit Property
    Set Hutch = m_Hutch.GetContainer()
End Property

' Nur per Code innerhalb des Projekts kann der Käfig für
' ein Kaninchen festgelegt werden
Friend Property Let Hutch(vNewValue _
 As RabbitCollection8)
    If vNewValue Is Nothing Then
        Set m_Hutch = Nothing
        Exit Property
    End If
    Set m_Hutch = vNewValue.GetContainerSubObject()
End Property
```

Gehen wir das Ganze einmal im Detail durch.

Wenn ein clsRabbit8-Objekt angelegt wird, wird es der RabbitCollection8-Collection hinzugefügt.

Das clsRabbit8-Objekt benötigt eine Referenz auf die Collection. Enthielte es eine Referenz auf die Collection selbst, ergäbe sich eine zirkuläre Referenz. Statt dessen wird beim Setzen der Hutch-Eigenschaft die Methode GetContainerSubobject verwendet, um eine Referenz auf ein clsGetContainer8-Objekt zu erhalten, das dem RabbitCollection8-Objekt gehört.

Jedes Mal, wenn ein clsRabbit8-Objekt eine Referenz auf seinen Container benötigt, wird die GetContainer-Methode des vom clsRabbit8-Objekt referenzierten clsGetContainer8-Objekts aufgerufen. Diese Methode löst ein Ereignis im RabbitCollection8-Objekt aus. Über dieses Ereignis gibt das RabbitCollection8-Objekt eine Referenz auf sich selbst an das clsGetContainer8-Objekt zurück. Dieses wiederum reicht die Referenz an das clsRabbit8-Objekt weiter.

Wenn die Hutch1-Variable auf Nothing gesetzt oder ihr ein anderes Objekt zugewiesen wird, wird das RabbitCollection8-Objekt zerstört. Dies funktioniert, da

ein per `WithEvents` deklarierter Ereignis-Empfänger den Referenzzähler des Client-Objekts (des Ereignis-Senders) nicht erhöht. Im `Terminate`-Ereignis des `Collection`-Objekts wird die interne Collection zerstört und damit werden auch alle darin enthaltenen `clsRabbit8`-Objekte freigegeben. Sobald diese Objekte und auch das `Collection`-Objekt zerstört sind, verfügt auch das `clsGetContainer8`-Objekt über keinerlei Referenzen mehr und wird ebenfalls zerstört.

Wieso funktioniert das? Die Antwort lautet: Ein Ereignis-Server hält eine Referenz auf ein internes Ereignis-Empfänger-Objekt anstelle des Client-Objekts, das die Ereignisse empfängt. Dieses interne Empfänger-Objekt muß zwar auch seinen Besitzer, also den eigentlichen Ereignis-Client kennen – doch er kennt ihn über einen COM-internen Mechanismus, der den Referenzzähler des Clients eben nicht erhöht. Ereignisse sind genau aus diesem Grund so gestaltet, damit sie die Art von zirkulären Referenzen vermeiden, über die wir hier gerade sprechen. Dieser Ansatz macht sich also die Vorteile des Ereignis-Mechanismus auf eine verallgemeinerte Weise zunutze, um zirkuläre Referenzen zu vermeiden. Dieses Konzept sehen Sie noch einmal in Abbildung 13.7 dargestellt.

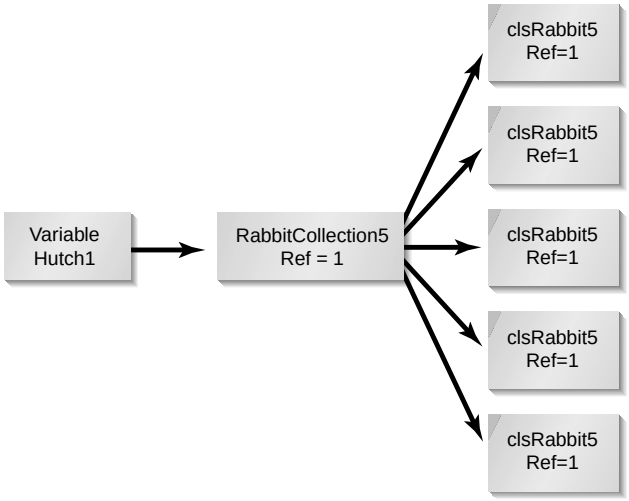


Abb. 13.7: Referenzierungen im Beispiel Rabbit8

13.3 Verschiedenes

Wir müssen noch ein paar Punkte anschneiden, bevor wir die Diskussion der Lebenszeit eines Objekts abschließen können.

13.3.1 Initialisierungs-Ereignisse

Alle Klassen und klassenähnlichen Module (Formular, MDI-Formular, `UserControl`, `UserDocument` und `PropertyPage`) in Visual Basic bieten Initialisierungs- und Terminierungs-Ereignisse.

Verwenden Sie das Initialize-Ereignis zur Initialisierung von objekteigenen Daten. Dazu gehört gewöhnlich auch das Anlegen von Unterobjekten, wie wir es in den verschiedenen Rabbit-Beispielen gesehen haben. Denken Sie daran, daß das Initialize-Ereignis für jede einzelne, neu angelegte Instanz ausgelöst wird.

Wie erreicht man es, eine klassenspezifische Initialisierung zu erhalten, die nur dann genau einmal ausgeführt wird, wenn die erste Instanz dieser Klasse initialisiert wird? Eine Möglichkeit hierfür wäre bei den Rabbit-Beispielen, die Numerierung der Kaninchen mit einer anderen Zahl als 1 beginnen zu lassen. Die globale Variable RabbitCounter im Modul modRabbit8 müßte genau und nur dann initialisiert werden, wenn das erste clsRabbit8-Objekt im RabbitCollection8-Objekt angelegt wird.

Sie deklarieren dazu in dem globalen Modul eine Variable vom Datentyp Boolean, die anzeigt, ob die Initialisierung bereits stattgefunden hat oder nicht. Prüfen Sie diese Variable jedes Mal während des Initialize-Ereignisses des betreffenden Objekts (hier der clsRabbit8-Klasse). Ist der Wert der Variablen False, führen sie die Erstinitialisierung durch und setzen dann die Variable auf True, um den weiteren Objekten anzuzeigen, daß die Erstinitialisierung bereits stattgefunden hat.

Vermeiden Sie die Code-Menge in den Initialize-Ereignissen. Dies gilt insbesondere bei EXE-Servern, bei denen ein zu lang andauernder Initialisierungsvorgang einen OLE-Time-Out im Client auslösen kann.

13.3.2 Terminierungs-Ereignisse

Setzen Sie im Terminate-Ereignis Variablen, die Objekt-Referenzen enthalten, auf Nothing.

Testen Sie das Beenden Ihrer Server-Anwendung, indem Sie die Client-Anwendung schließen und prüfen, ob die Objekte Ihrer Anwendung tatsächlich zerstört werden und die Client-Anwendung auch terminiert.

Denken Sie daran, daß während des Beendens einer Anwendung die Reihenfolge der Objekt-Zerstörung nicht vorherbestimmbar ist.

Prüfen Sie die Terminierung auch in der kompilierten Version Ihrer Komponente. Das Verhalten zur Laufzeit kann sich vom Verhalten in der Entwicklungsumgebung unterscheiden. So werden beispielsweise in der Entwicklungsumgebung DLL-Server tatsächlich nie aus dem Prozeßraum von Visual Basic entladen.

Vermeiden Sie das Auslösen von Fehlern im Terminate-Ereignis. Der Client könnte sich in einem Zustand befinden, in dem er den Fehler nicht behandelt, was zu einem Ausnahmefehler im Client führen könnte.

13.3.3 Nicht beendbare Anwendungen

Die Rabbit-Beispiele in diesem Kapitel haben einen entscheidenden Vorteil gegenüber EXE-Servern. Alle Objekte laufen In-Process. Wenn Sie letztlich die Haupt-Anwendung schließen, sorgt Windows von selbst dafür, daß alle Objekte zerstört werden, auch wenn sich einige davon oder alle in zirkulären Referenzen verfangen haben sollten.

Dies ist bei EXE-Servern anders. Enthält etwa ein EXE-Server namens `ServerA` eine Referenz auf einen anderen EXE-Server namens `ServerB`, und enthält umgekehrt `ServerB` eine Referenz auf `ServerA`, wird keiner der beiden Server jemals terminieren. Das liegt daran, daß ein EXE-Server nur dann terminieren kann, wenn alle seine Objekte zerstört worden sind. Achten Sie also darauf, gerade bei EXE-Server-Komponenten peinlich genau mit Ihren Objekten aufzuräumen.

13.3.4 Private Objekte

Übergeben Sie niemals eine Referenz auf ein privates Objekt an einen Client oder an ein Client-Objekt, der bzw. das selbst nicht Bestandteil der Komponente ist. In den meisten Fällen wird es zwar trotzdem funktionieren. Jedoch können private Objekte nicht verhindern, daß ein Server entladen wird. Dies bedeutet, daß ein Client über eine Objekt-Referenz verfügen könnte, zu der das Objekt nicht mehr existiert, was unweigerlich zu einem Speicherausnahmefehler oder zu anderen Speicherfehlern in Ihrer Anwendung führen dürfte.

Private Objekte sind nicht dasselbe wie Objekte, deren `Instancing`-Eigenschaft auf `PublicNotCreatable` gesetzt ist. Letztere sind öffentlich und dazu vorgesehen, bei Bedarf an andere Clients und Komponenten weitergegeben zu werden. Nur können sie nicht von einem Client angelegt werden – die Referenz kann nur über ein anderes Objekt innerhalb der Komponente erzeugt werden.

13.3.5 Verfolgen Sie Ihre Objekte

Die Rabbit-Beispiele demonstrieren, wie Sie mit Hilfe eines in einem globalen Standard-Modul untergebrachten Objekt-Zählers und `Debug.Print`-Anweisungen das Anlegen und Terminieren von Objekten verfolgen können. Visual Basic sichert den Inhalt des Direkt-Fensters, so daß Sie diesen nach einem Stopp der Anwendung analysieren können. So können Sie sehr leicht feststellen, ob Ihre Objekte ordnungsgemäß terminieren.

13.3.6 Beenden Sie Ihre Anwendung nicht per Stop

In der Visual Basic-Entwicklungsumgebung gibt es zwei Möglichkeiten, eine Anwendung zu beenden. Zum einen kann dies über das reguläre Schließen des letzten bzw. Haupt-Formulars der Anwendung erfolgen – so wie auch eine kompilierte Anwendung beendet werden muß. Zum anderen kann es über den BEENDEN-Befehl im AUSFÜHREN-Menü bzw. der entsprechenden Schaltfläche in der Symbolleiste der Visual Basic-Entwicklungsumgebung erfolgen.

Eine laufende Anwendung kann auch auf zwei ähnliche Weisen beendet werden. Zum einen kann dies auch hier wieder über das reguläre Schließen des letzten bzw. Haupt-Formulars erfolgen, zum anderen über die `End`-Anweisung im Code.

Generell sollten Sie Ihre Anwendungen in der Visual Basic-Entwicklungsumgebung regulär beenden und nicht den BEENDEN-Befehl der Entwicklungsumgebung verwenden.

Für kompilierte Anwendungen gilt gleichermaßen, niemals im Code die Anweisung `End` zum Beenden einer Anwendung oder einer Komponente zu verwenden. Die einzige einzuräumende Ausnahme wäre der Fall, daß eine Fehlerbehandlung innerhalb der Anwendung auf ein derart schweres Problem stößt, daß es gefährlich werden könnte, auch nur eine einzige weitere Zeile Code auszuführen. Ich kann mich jedoch nicht daran erinnern, die `End`-Anweisung selbst jemals aus einem anderen Grund als zur Demonstration ihrer Unbrauchbarkeit zu verwenden.

Denn wenn Sie den BEENDEN-Befehl der Entwicklungsumgebung oder die `End`-Anweisung im Code verwenden, weisen Sie Visual Basic damit an, auf der Stelle mit der Ausführung jeglichen Codes aufzuhören. Visual Basic wird zwar noch versuchen, den Speicher so gut es geht aufzuräumen. Jedoch wird kein Terminierungs-Code mehr aufgerufen.

Dies bringt Sie nicht nur um die Möglichkeit, Ihren Terminierungs-Code zu testen, sondern Sie könnten auch feststellen, daß einige Objekte nicht sauber aufgeräumt werden. Falls Sie beispielsweise API-Funktionen zum Anlegen von System-Objekten verwendet haben, sind Visual Basic und Windows unter Umständen nicht mehr dazu in der Lage, diese aufzuräumen, wenn Ihre Anwendung auf diese Weise beendet wird. Visual Basic wird diese Objekte definitiv erst dann schließen oder freigeben, wenn Sie die Anwendung erneut zu starten versuchen. So kann es dann beispielsweise vorkommen, daß der Zugriff auf eine beim ersten Lauf der Anwendung geöffnete Datei nun fehlschlägt, da Sie vielleicht noch gesperrt ist.

Gewöhnen Sie sich also daran, Ihre Anwendungen über das SYSTEM-Menü des Haupt-Formulars zu schließen. Sie können sich auch dazu zwingen, indem Sie die entsprechenden Menü-Punkte und Symbolleisten-Schaltflächen entfernen.

13.4 Lebensdauer von Objekten unter MTS

Bis noch vor kurzem brauchte sich ein Visual-Basic-Programmierer in bezug auf die Lebenszeit von Objekten nur darum zu kümmern, daß Objekte ordentlich zerstört werden, wenn Sie nicht mehr benötigt werden. Zirkuläre Referenzen und ähnliche Probleme können zu einer stetigen Zunahme von überflüssigen, aber »untoten« Objekten führen, die mehr und mehr System-Ressourcen binden, bis schließlich dem System der Speicher ausgeht.

Eine solche Situation ist schon schlimm genug, wenn sie auf einem Client-System eintritt. Doch wie steht es darum, wenn diese freischwebenden Objekte Fern-Referenzen auf Objekte auf einem Netz-Server über DCOM oder Remote Automation halten sollten? Oder genauso bedeutsam, wenn die Objekte Teil einer Komponente sind, die auf einem Netz-Server läuft, der Hunderte oder gar Tausende Client-Maschinen zu versorgen hat? Ein kleiner oder unmerklicher Speicherfehler mag auf einer Workstation nicht mehr als ein Ärgernis darstellen – auf einem Netz-Server kann sich das jedoch fatal auswachsen.

Natürlich, werden Sie sagen, werden Sie alles Notwendige unternehmen, damit Ihre Objekte ordentlich zerstört werden. Wenn Sie nun aber eine Server-Anwendung erstellen wollen, die Hunderte oder Tausende Clients versorgen soll? Dann werden auch Hunderte oder Tausende Objekte in Ihrer Server-Anwendung angelegt. Auch wenn der Server brav seinen Dienst tut, wird eine fürchterlich große Zahl an Objekten herumliegen.

Nun, wenn einer der Clients »seine« Objekte ständig braucht, ist das unabänderlich. Doch wenn die Clients, die Objekte auf Ihrem Server anlegen, sich die meiste Zeit im Leerlauf befinden und beispielsweise auf Anwender-Eingaben warten oder andere Aufgaben ausführen, sieht das schon anders aus. Sie könnten jeden Client dazu auffordern, netterweise alle nicht gerade benötigten Objekte freizugeben. Vielleicht werden nicht viele Clients dieser Aufforderung nachkommen. Wenn aber doch, werden Sie feststellen, daß das dann einsetzende ständige Anlegen und Freigeben samt Auf- und Abbau der Verbindungen erhebliche Ressourcen auf der Server-Seite binden wird. Der sich so ergebende Overhead könnte nur allzuschnell die sich durch die Reduzierung der Objekte ergebende Ersparnis an Ressourcen wieder auffressen.

Wenn Sie sich dazu noch vor Augen halten, daß jeder der Clients Remote-Objekte in separaten Prozessen oder Threads auf Ihrer Server-Maschine anlegt, werden Sie feststellen, daß es gar nicht so vieler Clients bedarf, Ihren Server in die Knie zu zwingen. Was können Sie als Programmierer dagegen unternehmen?

13.4.1 Die Größe macht's

Wenn es nur um eine kleinere oder mittlere Geschäftsanwendung geht und die Anzahl der gleichzeitig zu bedienenden Clients relativ gering bleibt, fällt die Antwort kurz aus: Kaufen Sie einen leistungsfähigeren Rechner und ein paar Megabyte RAM mehr dazu. Wenn es sich aber um eine große Web-Site handelt, auf der Active Server Pages oder IIS-Anwendungen laufen (die fast dasselbe sind), oder wenn eine große Zahl an Anwendern einer Server-Komponente gleichzeitig über DCOM oder Remote-Automation bedient werden soll, werden Sie wohl früher oder später nach Möglichkeiten Ausschau halten, die Zahl der Objekte auf dem Server zu verringern oder eine Reihe von Objekten von mehreren Clients gemeinsam nutzen zu lassen.

Microsofts Antwort auf dieses Problem ist der Microsoft Transaction Server (MTS). Nun, MTS schmeckt ein wenig nach OLE – es ist eine Kiste voller (OLE-)Technologien, die zusammen das Problem bekämpfen sollen. Es schmeckt aber auch deswegen nur ein wenig nach OLE, weil Microsoft sich noch nicht ganz schlüssig ist, wie das Kind denn eigentlich zu taufen sei. Ihnen ist vielleicht bereits der Begriff »COM+« untergekommen (ausgesprochen heißt das COM-Plus – wen wundert's?). Tatsächlich soll COM+ das gute alte COM plus MTS sein. MTS ist jedoch ein Thema, mit dem sich locker ein eigenes Buch füllen ließe, so daß es mir hier nun nicht darum gehen wird, Ihnen alles über MTS beizubringen. Ich hoffe jedoch, Ihnen den Bezug dieser Technologie zur Geschichte mit der Lebensdauer von Objekten vermitteln zu können. MTS umfaßt auch ein Konzept, das sich »Transactioning« nennt und dabei helfen soll, daß eine Gruppe von Komponenten unter sich ausmachen kann, wann sie aktiv bleiben soll und wann sie freigegeben werden kann, und die sicherstellen soll, daß ein zusammenhängender Satz von Anweisungen entweder komplett korrekt ausgeführt wird oder eben gar nicht. Doch dies ist ein Thema, das den Rahmen dieses Buches sprengen würde.

Die Wirkung von MTS auf ein Objekt sehen Sie anhand der `gtpActivity`-Komponente im Ordner zu diesem Kapitel 13 auf der Buch-CD. Was macht diese Komponente zu einer MTS-Komponente? Zunächst einmal enthält sie einen Verweis auf die Microsoft-Transaction-Server-Typbibliothek. Damit kann die Komponente auf die MTS-Komponenten zugreifen. Weiterhin implementiert sie das `ObjectControl`-Interface. Eine MTS-geeignete Komponente braucht dieses Interface nicht unbedingt zu implementieren. Doch wenn das Objekt benachrichtigt werden soll, wenn es deaktiviert oder gemeinsam benutzt wird, ist dieses Interface unabdinglich. Schließlich legt das Objekt fest, ob es im Transaktionmodus laufen soll, indem die Eigenschaft `MTSTransactionMode` auf einen der Transaktionmodi gesetzt wird. Diese Eigenschaft wirkt sich nur auf Objekte aus, die über den MTS laufen. Im Beispiel ist die Eigenschaft eingestellt, Transaktionen zu erfordern.

In Listing 13.5 sehen Sie den Code der `Demo`-Klasse der `gtpActivity`-Komponente.

```
' Einfaches MTS-Beispiel
' Copyright © 1998 by Desaware Inc. All Rights Reserved

Option Explicit

Implements ObjectControl

Public MyClient As Object
Private objectnum As Long

Public Property Get MyID() As Long
   MyID = objectnum
End Property
```

```
Public Sub PerformOperation(ByVal objectnumber As Long, _
 ByVal iscomplete As Boolean)
   Dim obj As ObjectContext
   objectnum = objectnumber
   MyClient.message "Operation Performed: " & objectnum
   Set obj = GetObjectContext()

   If iscomplete Then obj.SetComplete
End Sub

Private Sub Class_Terminate()
   If Not MyClient Is Nothing Then
      MyClient.message "Object terminating: " & objectnum
   End If
End Sub

Private Sub ObjectControl_Activate()
   If Not MyClient Is Nothing Then
MyClient.message "Object activated: " & objectnum
   End If
End Sub

Private Function ObjectControl_CanBePooled() As Boolean
   ObjectControl_CanBePooled = True
End Function

Private Sub ObjectControl_Deactivate()
   If Not MyClient Is Nothing Then
      MyClient.message "Object deactivated: " & objectnum
   End If
End Sub
```

Listing 13.5: Die Demo-Klasse der Komponente gtpActivity

Die Objekt-Variable `MyClient` enthält eine Referenz auf den Client. Sie sollten diese Technik nicht in einer realen Anwendung verwenden, doch in diesem Beispiel können wir so auf einfache Weise den Zustand der MTS-Komponente ermitteln. Die Komponente wird die `Message`-Methode des `MyClient`-Objekts aufrufen (falls es existiert), um den Client davon zu benachrichtigen, daß etwas geschehen ist. Dies entspricht voll und ganz der OLE-Rückruf-Technik, die wir in Kapitel 11, »Ereignisse«, dargestellt haben. Wir kümmern uns nicht um die Tatsache, daß der Client nicht vom Server freigegeben wird, da unser Client selbst sorgfältig genug vorgeht und alle Server-Objekte freigibt. Denken Sie daran, daß dies nur eine »billige« Lösung zur Erlangung von Debug-Informationen ist und kein empfehlenswerter Ansatz für eine reale Anwendung, insbesondere in Verbindung mit dem MTS, der eigene Anforderungen an gemeinsam genutzte Objekt-Referenzen stellt.

Jedem Objekt wird vom Client eine Nummer zugewiesen, damit Sie die Instanzen verfolgen können. Diese Nummer wird in der `PerformOperation`-Methode zugewiesen. Dieser Methode wird vom Client ein Parameter übergeben, der festlegt, ob die `ObjectContext.SetComplete`-Methode aufgerufen wird. Dies werde ich gleich näher erklären. Dazu wird der Client versuchen, Nachrichten an die Komponente zu senden, wenn das Objekt aktiviert, deaktiviert oder terminiert wird.

Zum Testen dieser Komponente müssen Sie ein Paket erstellen und es als MTS-Komponente registrieren. Ich gehe davon aus, daß Sie sich mit der Installation eines MTS-Pakets auskennen, wenn Sie dieses Beispiel zu testen wünschen. Wenn Sie sich mit MTS noch nicht auskennen, brauchen Sie das Beispiel nicht unbedingt laufen zu lassen, um weiterlesen und das weitere verstehen zu können.

13.4.2 Just-in-Time-Aktivierung und Deaktivierung

Denken Sie einmal kurz zurück an die Marshaling-Diskussion in Kapitel 6, »Das Leben und die Lebensdauer einer ActiveX-Komponente«. Damit ein Objekt in einem separaten Prozeß oder auf einer Remote-Maschine laufen kann, benötigt die lokale Maschine ein Proxy-Objekt, das über dieselben Interfaces wie das Remote-Objekt verfügt. Methoden-Aufrufe beim Proxy-Objekt werden über die Prozeßgrenzen hinweg zum Remote-Server übertragen (»marshaled«). Dann werden die Methoden des »realen« Objekts mit den gleichen Parametern aufgerufen, die der Client beim Aufruf des Proxy-Objekts verwendet hat. Überlegen Sie einmal: Ab welchem Zeitpunkt ist es wirklich notwendig, daß das Objekt tatsächlich existiert? Wenn das Client-Programm ein Remote-Objekt anlegt, weiß es nicht, ob es eine Referenz auf ein reales oder auf ein Proxy-Objekt erhält – die Hauptsache ist, es erhält eine brauchbare Referenz auf ein Objekt. Das Remote-System braucht nicht unbedingt ein reales Objekt anzulegen, damit dem Client eine Referenz auf ein Proxy-Objekt zur Verfügung gestellt werden kann. Es kann so lange mit dem Anlegen des realen Objekts warten, bis der erste Methoden-Aufruf über das Proxy-Objekt und das Marshaling-System eintrifft. Dieses verzögerte Anlegen eines Objekts nennt man »Just-in-Time«-Aktivierung und wird vom MTS angewendet, um mit dem Anlegen eines Objekts warten zu können, bis der Client das Objekt tatsächlich benötigt.

Angenommen, Sie hätten ein Objekt, das eine einfache Operation wie die Addition zweier Zahlen durchführt und das Ergebnis zurückliefert. Ebenfalls angenommen, das Objekt enthält keine internen modulweit gültige Variablen – eben nur die Methode zur Addition zweier Zahlen. Was würde geschehen, wenn Sie das Objekt zwischen zwei Methoden-Aufrufen freigeben würden? Würde es trotzdem funktionieren? Aber sicher doch – der Client ist lediglich daran interessiert, daß die Methode beim Aufruf ihren Dienst erfüllt. Da in dem Objekt keine internen Daten gespeichert werden, braucht es den Client nicht zu interessieren, ob das Objekt für jeden Methoden-Aufruf extra wieder angelegt wird. Ein Objekt, das keine internen Daten enthält, wird »zustandslos« (»stateless«) genannt.

Stellen Sie sich nun weiterhin vor, Sie hätten tausend Clients, die dieses zustandslose Objekt benötigen, um zwei Zahlen zu addieren. Sie wissen bereits, daß der MTS mit dem Anlegen des Objekts so lange wartet, bis der erste Methoden-Aufruf erfolgt. Ist es aber unbedingt notwendig, tausend Instanzen des Objekts anzulegen, für jeden Client eine eigene? Das Objekt ist zustandslos. Somit könnten Sie es durchaus allen Clients gestatten, ein und dieselbe Instanz gemeinsam zu nutzen. Sie könnten auch diese Instanz nach jedem Gebrauch wieder zerstören und sie erneut anlegen, wenn ein Client darauf zuzugreifen versucht. Das Client-seitige Proxy-Objekt bleibt bestehen, so daß der Client nicht das geringste merken würde. Im MTS beschreiben die Begriffe »Activate« und »Deactivate« diese Situation. Legt ein Client eine Referenz auf ein MTS-Objekt an, wird nicht zugleich das reale Objekt angelegt. Es verbleibt deaktiviert. Ruft der Client eine Methode des Objekts auf, wird das reale Objekt abgelegt, also aktiviert. Woher weiß der MTS, wann ein Objekt sicher deaktiviert werden kann? Das Objekt muß über eine Möglichkeit verfügen, dem MTS mitzuteilen, daß es gefahrlos deaktiviert werden kann, d.h. daß keine internen Daten für dieses Objekt bewahrt zu werden brauchen. Das Objekt kann dazu folgendes tun:

- Es kann aktiviert bleiben und die internen Daten bewahren.
- Es kann dem MTS mitteilen, daß es seine Arbeit erledigt hat (Transaktion) und auch keine internen Daten bewahren will. Dies geschieht über die Methode `SetComplete` des Objekts `ObjectContext`.
- Es kann dem MTS mitteilen, daß es gegebenenfalls aktiviert werden kann und daß es auf den Aufruf der `Deactivate`-Methode des `ObjectControl`-Interfaces (dort ist diese Methode implementiert) wartet, über den die Deaktivierung angefordert wird. Trifft diese Anforderung ein, kann das Objekt seine internen Daten – seinen Status – über einen beliebigen Mechanismus sichern. Die Erledigung wird über die Methode `EnableCommit` des `ObjectContext`-Objekts gemeldet.

Einen sorgfältigen Entwurf vorausgesetzt, kann der MTS die Anzahl der jeweils auf dem Server aktiven Objekte spürbar deutlich verringern. Der Nachteil ist natürlich, daß es schon ein tiefergehendes Wissen zum Erstellen von Objekten erfordert, die sicher und korrekt unter dem MTS laufen können. Dazu ist auch noch spezielles Wissen zur Konfiguration und Verteilung solcher Komponenten erforderlich. Das ist ein umfangreiches Thema, das den Rahmen dieses Buches sprengen würde. Doch nun wissen Sie, wohin die Reise gehen wird, wenn Sie diese Ebene der Skalierbarkeit zu erklimmen versuchen wollen.

13.4.3 Testen des Demo-Objekts

Das Beispiel-Programm `gtpActTest` aus Listing 13.6 verwendet das `gtpActivity.Demo`-Objekt zur Veranschaulichung der Lebenszeit eines Objekts unter MTS.

```
' gtpActivity Test
' Copyright © 1998 by Desaware Inc. All Rights Reserved

Option Explicit

Dim Demos As New Collection

Dim counter As Long

Private Sub cmdCreate_Click()
   Dim ac As Demo
   Dim AllowDeactivate As Boolean
   Set ac = CreateObject("gtpActivity.Demo")
   Set ac.MyClient = Me
   counter = counter + 1
   If chkDeactivate.Value <> 0 Then AllowDeactivate = True
   ac.PerformOperation counter, AllowDeactivate
   Demos.Add ac
End Sub

Public Sub message(str As String)
   Debug.Print str
End Sub

Private Sub cmdDelete_Click()
   If Demos.Count >= 1 Then
      Demos.Remove 1
   End If
End Sub

Private Sub cmdCycle_Click()
   Dim ac As Demo
   Dim c As Long
   For Each ac In Demos
      c = c + 1
      Debug.Print c & ": Object " & ac.MyID & _
       " in collection"
   Next
   If Demos.Count = 0 Then
      Debug.Print "No objects in list"
   End If
End Sub
```

Listing 13.6: Das Projekt gtpActTest.vbp

Das Formular enthält drei Schaltflächen. Die Schaltfläche `cmdCreate` sorgt dafür, daß ein `gtpActivity.Demo`-Objekt angelegt wird. Die Eigenschaft `MyClient`

wird auf das Formular gesetzt, die über eine `Message`-Methode verfügt, die von dem Objekt zur Benachrichtigung beim Auftreten verschiedener Ereignisse aufgerufen werden kann. Die Nachrichten werden im Direkt-Fenster ausgegeben. Jedes Mal wenn ein Objekt angelegt wird, wird seine Methode `PerformOperation` aufgerufen, die die Objekt-Nummer setzt und festlegt, ob eine Deaktivierung erlaubt sein soll. Wenn Sie das Flag `Deactivation` auf `True` setzen, wird die Komponente die Methode `SetComplete` des `ObjectContext`-Objekts aufrufen, bevor die Funktion zurückkehrt. Dies informiert den MTS darüber, daß die Transaktion abgeschlossen ist und daß das Objekt deaktiviert werden kann. Schließlich wird das Objekt in die Collection `Demos` aufgenommen. Die Schaltfläche `cmdDelete` entfernt ein Objekt aus dieser Collection. Über die Schaltfläche `cmdCycle` wird die Objekt-Nummer jedes `Demo`-Objekts angezeigt, das sich in der `Demos`-Collection befindet.

Starten Sie das Programm und klicken Sie dreimal auf die Schaltfläche CREATE OBJECT, einmal auf die Schaltfläche CYCLE und dreimal auf die Schaltfläche DELETE OBJECT. Im Direkt-Fenster wird folgendes ausgegeben:

```
Click Create Object 3 times:

Operation Performed: 1
Operation Performed: 2
Operation Performed: 3

Cycle pressed here:

1: Object 1 in collection
2: Object 2 in collection
3: Object 3 in collection

Click Delete Object 3 times:

Object deactivated: 1
Object terminating: 1
Object deactivated: 2
Object terminating: 2
Object deactivated: 3
Object terminating: 3
```

Wie Sie sehen können, Wird das Objekt nicht gelöscht, bevor Sie nicht auf die Schaltfläche DELETE OBJECT klicken. Versuchen Sie es noch einmal, setzen Sie jedoch zuvor die Option ALLOW DEACTIVATION. Die Ergebnisse im Direkt-Fenster lauten diesmal:

```
Click Create Object 3 times:

Operation Performed: 1
Object deactivated: 1
Object terminating: 1
```

```
Operation Performed: 2
Object deactivated: 2
Object terminating: 2
Operation Performed: 3
Object deactivated: 3
Object terminating: 3

Cycle pressed here:

1: Object 0 in collection
2: Object 0 in collection
3: Object 0 in collection

Click Delete Object 3 times:
```

Was geht hier vor? Die Demo-Komponente ruft die Methode `SetComplete` des `ObjectContext`-Objekts auf, bevor die Methode `PerformOperation` verlassen wird. Dies teilt dem MTS mit, daß er das Objekt ohne weiteres löschen kann. Dadurch werden die Ereignisse `Deactivate` und `Terminate` des Objekts ausgelöst – das reale Objekt wurde zerstört. Doch der Client geht nach wie vor davon aus, daß sein Objekt weiterhin existiert – er hält nach wie vor eine Referenz auf das Proxy-Objekt des `Demo`-Objekts. Wenn Sie auf die Schaltfläche CYCLE klikken, werden die drei Objekte angezeigt. Aber warum tragen alle die Nummer Null? Und warum sehen Sie keine `Activation`-Methode? Nun, wenn der MTS die Abfrage der `MyID`-Eigenschaft empfängt, legt er das reale Demo-Objekt an – Just-in-Time-Aktivierung. Jedoch sind die zuvor in dem Objekt gespeicherten Daten, die Objekt-Nummer und die `MyClient`-Referenz längst zerstört worden! Und schließlich hat dieses Objekt keine besonderen Maßnahmen zur Bewahrung dieser Daten unternommen. Somit gibt jedes Objekt den Initialisierungswert der Objekt-Nummer, nämlich Null zurück. Da auch keine Referenz auf den Client mehr existiert, kann die Komponente auch keine Nachrichten über die Aktivierung und Deaktivierung übermitteln.

Interessanterweise wird das nicht wieder deaktiviert. Denn die Komponente hat nach dem Lesen der `MyID`-Eigenschaft nicht die `SetComplete`-Methode aufgerufen. Das Objekt wird erst vom Client selbst wieder zerstört.

Sie sehen, die Verwaltung der Lebensdauer eines Objekts mit MTS-Komponenten fügt der Entwicklung von COM-Komponenten eine weitere Facette hinzu. Aber es ist eine, die sich für die Entwicklung von umfangreichen Unternehmenslösungen als sehr wertvoll erweisen kann.

Da wir gerade von nützlichen Features für umfangreichere Systeme sprechen – vielleicht haben Sie schon gehört, daß Visual Basic Multithreading unterstützt und sind ganz begierig darauf, sich damit beschäftigen zu können. Na, im nächsten Kapitel werden Sie sehen, daß es recht spaßig werden kann, damit herumzuspielen, aber auch, daß es sehr nützlich sein kann – aber nur dann, wenn mit Bedacht und Vernunft angewendet.