Kapitel 12

Gruppen-Objekte

Dies ist ein Kapitel über Collections.

Nein, das stimmt nicht ganz. Es ist vielmehr ein Kapitel, in dem es um die Organisation und Verwaltung von Gruppen von Objekten geht. Ich denke, Visual-Basic-Programmierer lassen sich in dieser Hinsicht in vier Sorten einteilen:

- Da sind diejenigen, die »wissen«, daß man Gruppen von Objekten am besten mit Collections verwaltet.
- Dann gibt es diejenigen, die »wissen«, daß man bisher Gruppen von Objekten am besten in Collections verwaltet hat, das nun aber am besten in Dictionaries geht.
- Weiterhin gibt es diejenigen, die sich weder mit Collections noch mit Dictionaries auskennen, sondern Gruppen von Objekten in Arrays verwalten.
- Und schließlich gibt es diejenigen, die die Vor- und Nachteile von Collections, Dictionaries und Arrays kennen und sich jedesmal darüber den Kopf zerbrechen, welche Technik die bessere ist, wenn sie es mit mehr als einem Objekt in einem Projekt zu tun haben.

Ich hoffe, daß Sie sich zur vierten Gruppe rechnen werden, wenn wir dieses Kapitel hinter uns gebracht haben – auch wenn Ihnen das zunächst vielleicht noch etwas gegen den Strich gehen wird.

12.1 Die Merkmale von Collections und Dictionaries

Sie werden sich vielleicht darüber wundern, wieso ich nicht mit einer Einführung zu Arrays beginne, bevor ich mich dem etwas komplexeren Thema der Collections zuwende. Der Grund ist simpel: Arrays stellen ein derart fundamentales Sprachkonstrukt dar, so daß jeder Visual-Basic-Programmierer oberhalb der untersten Anfängerebene damit vertraut sein sollte. Wenn Sie sich also mit diesem Buch beschäftigen, sollten Sie bereits wissen, was ein Array ist und wie man Arrays verwendet. Sollte dies noch nicht der Fall sein, kann ich Ihnen nur empfehlen, eine Pause bei der Lektüre dieses Buches einzulegen und sich erst einmal mit den entsprechenden Kapiteln in der Visual-Basic-Dokumentation zu beschäftigen.

Nun, was genau ist eine Collection? Sie wissen bereits, daß ein `Variant` eine spezielle Variablen-Art ist, die nahezu jeden Datentyp aufnehmen kann. Eine Collection ist ein Objekt, das einen ganzen Haufen von Variants enthalten kann – so viele, wie Sie wollen. Uns interessiert hier nur die Fähigkeit eines `Collection`-Objekts, andere Objekte enthalten zu können. Sie sollten daher immer im Sinn behalten, daß das in diesem Kapitel Gesagte auch genausogut für andere Datentypen gilt. Bei der Komponenten-Entwicklung steht jedoch die Verwendung eines `Collection`-Objekts als Behälter für andere Objekte im Vordergrund.

Jedem Element in einer Collection sind zwei Werte zugeordnet. Zum einen ist dies die Position innerhalb der Collection (beginnend mit eins bis hin zur Anzahl der enthaltenen Elemente). Zum anderen ist es ein optionaler Schlüssel (`Key`) – ein String, der eindeutig ein Element kennzeichnet.

Eine Collection verfügt über folgende Eigenschaften und Methoden:

- Eigenschaft `Count`*:* Diese schreibgeschützte Eigenschaft liefert die Anzahl der in einer Collection enthaltenen Elemente.
- Methode `Item`*:* Diese Methode gibt einen `Variant` zurück, der das Element enthält, das durch einen der Methode übergebenen `Variant`-Parameter angegeben ist. Sie können sowohl eine Zahl übergeben, die die Position (den Index) des Elements in der Collection angibt, oder Sie können einen `String` übergeben, der den Schlüssel des gewünschten Elements darstellt. Im vorangegangenen Kapitel 10 im Abschnitt »Überladene Eigenschaften und Funktionen« haben wir gesehen, wie die tatsächliche Operation einer Methode von dem in einem `Variant`-Parameter übergebenen Datentyp abhängig gemacht werden kann.
- Methode `Add`*:* Diese Methode akzeptiert vier Parameter: `Item`, `Key`, `Before` und `After`. Alle bis auf den ersten sind optional. Im `Variant`-Parameter `Item` wird das einzufügende Element übergeben. In `Key` wird der Schlüssel für das Element übergeben, oder ein Leerstring, wenn kein Schlüssel verwendet werden soll. Die Parameter `Before` und `After` sind Variants, in denen entweder eine Zahl als Position oder ein `String` als Schlüssel eines bereits in der Collection enthaltenen Elements übergeben werden. Das hinzuzufügende Element wird daraufhin vor bzw. hinter dem in `After` bzw. `Before` angegebenen Element in die Collection eingefügt (abhängig davon, ob `After` *oder* `Before` angegeben wurde).
- Methode `Remove`*:* Diese Methode entfernt das Element aus der Collection, das im übergebenen `Variant`-Parameter über die Position oder den Schlüssel angegeben wurde.

Das `Collection`-Objekt wird ausführlich in der Visual Basic-Dokumentation besprochen. Dennoch werde ich ihm hier einigen Raum widmen. Warum verdient ein derart einfaches Objekt soviel Aufmerksamkeit? Zur Beantwortung dieser Frage werfen wir einmal einen Blick auf einige der Vorteile, die eine Collection Visual-Basic-Programmierern bietet:

- Sie ist einfach zu verwenden.
- Sie erledigt automatisch die notwendige Speicherverwaltung.
- Sie kann nahezu jeden Datentyp verarbeiten.
- Die Unterstützung von frei wählbaren Schlüsseln erleichtert das Ansprechen einzelner gewünschter Elemente.

- Wie jede andere Variable, die Objekte aufnehmen kann, verwaltet sie automatisch den Referenzzähler (so daß Sie nicht zu befürchten brauchen, daß in einer Collection enthaltene Objekte unbeabsichtigt verschwinden oder freigegeben werden).
- Collections erleichtern die Übergabe von großen Datenmengen von einem Objekt zu einem anderen.

Natürlich wartet eine Collection auch mit einigen Nachteilen auf:

- Sie ist so einfach zu verwenden, daß Sie manchmal unangebracht verwendet wird.
- Sie bringt einen Speicherverwaltungs-Overhead mit sich, der manchmal gar nicht notwendig ist.
- Wie jeder `Variant` ist sie im Vergleich zu spezifischen Datentypen weniger effizient und bringt Performance-Verluste mit sich.
- Objekte werden über Variants identifiziert, die entweder String-Schlüssel oder Index-Nummern enthalten. Bei der Verwendung von Index-Nummern entsteht interner Overhead bei der Prüfung des `Variant`-Datentyps. Bei der Verwendung von Schlüsseln entsteht interner Overhead bei der Bearbeitung der Strings.
- Da Collections Objekt-Referenzen enthalten können, wird unter Umständen ein Objekt nicht freigegeben, wenn eine Collection nicht ordentlich »aufgeräumt« wird.
- Über Collections können große Datenbestände einer Komponente offengelegt werden, die unter Umständen von Client-Anwendungen auf unerwünschte Weise manipuliert werden können.

Wie Sie sehen, gibt es zu jedem der Vorteile auch den entsprechenden Nachteil als Gegenpart. Wenn man von einem Objekt als einer zweischneidigen Klinge sprechen kann, dann ganz besonders vom Collection-Objekt. Nehmen wir uns also ein paar dieser Features vor und schauen uns an, wie sie bei der Komponenten-Entwicklung sinnvoll zum Zuge kommen können.

12.1.1 Entscheidungskriterien

Sie werden bei der Entwicklung von Komponenten in die Situation kommen, Gruppen von Objekten (oder anderen Datentypen) verwalten zu müssen. Visual Basic bietet selbst zwei Mechanismen dazu an: Collections und Arrays. Die VBScript-Laufzeitbasis bietet darüber hinaus noch ein neues `Dictionary`-Objekt an, auf das ich gleich näher eingehen werde.

Wann sollten Sie Arrays und wann sollten Sie Collections oder Dictionaries verwenden? Das kann ich Ihnen nicht sagen. Bevor Sie nämlich entscheiden können,

welche Technik wann die bessere ist, müssen Sie erst einmal die zu lösende Aufgabe anhand einiger Kriterien durchleuchten:

- Wie kritisch ist der Performance-Aspekt? Wären Sie bereit, Zeit in die Entwicklung schnelleren Codes zu investieren?
- Wie kritisch ist der Aufwand der Entwicklung bzw. deren Dauer? Sind Sie bereit, Performance-Verluste hinzunehmen, um schneller mit Ihrem Programm fertig zu werden?
- Sollen die in einer Gruppe zusammengefaßten Objekte für andere Anwendungen offengelegt werden, oder wird die Gruppe nur innerhalb Ihrer eigenen Anwendung verwendet? In letzterem Fall können Sie eher auf die Entwicklung einer robusten externen Schnittstelle mit extensiver Typ- und Fehlerprüfung verzichten.

Auf diese Kriterien werde ich im Laufe dieses Kapitels noch näher eingehen, wenn es um verschiedene Ansätze der Implementierung von Collections geht. Schauen wir uns jetzt aber erst einmal den neuesten Typ eines Gruppenobjekts an.

12.1.2 Dictionaries – der Neuzugang

Sie werden wahrscheinlich schon von VBScript gehört haben, der Script-Sprache, von deren Verwendung in Webseiten anstelle von JavaScript Microsoft gerne jedermann überzeugen möchte. Diese Script-Engine enthält Objekte, die auch Visual-Basic-Programmierern über einen Verweis auf die Microsoft Scripting Runtime in jedem Projekt zur Verfügung stehen. Eines dieser Objekte ist das neue `Dictionary`-Objekt. Ein Dictionary ähnelt sehr stark einer Collection. Die wichtigsten Unterschiede sind:

- Jedem Element muß ein Schlüssel zugeordnet werden.
- Man kann den Schlüssel zu einem Element oder das Element zu einem Schlüssel ändern.
- Man kann eine Liste mit den Schlüsseln, den Elementen oder mit beidem erhalten.
- Man kann die Berücksichtigung der Groß-/Kleinschreibung beim Vergleich der Schlüssel kontrollieren.
- Man kann auf einfache Weise feststellen, ob ein Schlüssel bereits vorhanden ist.

Ich empfehle Ihnen, sich mit dem Verhalten von Dictionaries vertraut zu machen. Dictionaries können in manchen Fällen die bessere Wahl gegenüber Collections sein. Sie können gegebenenfalls eine höhere Performance bieten, da sich in manchen Fällen die manuelle Suche nach einem Element erübrigt. Sie begeben sich mit der Verwendung von Dictionaries jedoch in eine zusätzliche Abhängigkeit

beim Vertrieb Ihrer Anwendungen, da die Scripting-Runtime auf dem Zielsystem installiert sein muß. Dies ist jedoch nur bei älteren Systemen (wie etwa Windows 95 oder NT4 ohne die neueren Service-Packs) noch nicht der Fall.

Im Hinblick auf dieses Kapitel stellen Sie sich ein `Dictionary` einfach als Variante des `Collection`-Objekts vor. Alle Beispiele dieses Kapitels betreffen Collections, so daß Sie im Sinn behalten sollten, daß diese in den meisten Fällen auch durch Dictionaries ersetzt werden können, ohne daß die dahinterstehenden Grundgedanken verworfen werden müßten.

12.2 Die ~~drei~~ vier Ansätze, wie Komponenten Collections offenlegen

In der Visual-Basic-Dokumentation zu den Collections werden drei Ansätze für die Arbeit mit Collections und Objekten angeführt. Die dort verwendeten Beispiele handeln von der Verwaltung einer Gruppe von Angestellten über ein Formular. Diese drei Ansätze lauten wie folgt:

Ansatz Strohhütte: Hier wird eine öffentliche Collection zur Verwaltung der Angestellten-Objekte verwendet. Dieses Beispiel zeigt, wie ein anderer Teil des Programms darauf zugreifen und sogar ein ungültiges Objekt in die Collection einfügen kann.

Ansatz Blockhütte: Hier wird in dem Formular eine private Collection zur Verwaltung der Angestellten-Objekte verwendet. Dazu werden ein paar öffentliche Methoden und Eigenschaften geboten, über die andere Teile der Anwendung die Collection manipulieren können. Dies macht die Collection robuster, soweit es das übrige Programm betrifft, schützt aber die Collection immer noch nicht vor Bugs im Code des Formulars. Außerdem steht nicht mehr die Möglichkeit zur Verfügung, mit `For...Each` durch die Collection zu iterieren.

Ansatz Ziegelhaus: Hier wird eine separate Collection-Klasse zur Verwaltung der Angestellten angelegt.

Nun, dies ist eine schöne Beispielfolge dafür, was vermittelt werden soll, nämlich verschiedene Möglichkeiten, Objekte innerhalb einer Anwendung zu gruppieren. Der Fokus dieses Buches liegt jedoch auf der Entwicklung von Komponenten. Und da diese Beispiele auch Komponenten betreffen, fehlt ein wesentlicher Aspekt. Wie können – und sollen – Komponenten Gruppen von Objekten für Client-Anwendungen offenlegen?

Schauen wir uns dazu drei Beispiele an, die die gleichen Prinzipien auf Komponenten-Ebene demonstrieren.

12.2.1 Die Strohhütte

Als Züchter von Kaninchen haben Sie eine Anwendung zur Verwaltung ihres Zuchtprogramms entwickelt. Die Komponente `Rabbit1.vbp` bildet die Basis die-

ser Anwendung. Sie legt ein öffentliches Objekt namens `PetStore1` offen, von dem `Rabbit1`-Objekte angelegt werden. Das Modul `PetStore1` enthält folgenden Code:

```
' Guide to the Perplexed:
' Rabbit1 example
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

' Einkauf einer Collection von Kaninchen
Public Function BuyRabbits(ByVal RabbitCount _
 As Integer) As Collection
    Dim counter%
    Dim col As New Collection
    Dim obj As clsRabbit1
    ' Anlegen der gewünschten Anzahl von Kaninchen
    For counter = 1 To RabbitCount
        Set obj = New clsRabbit1
        col.Add obj
    Next counter
    ' Rückgabe einer Collection mit den Kaninchen
    Set BuyRabbits = col
End Function
```

Dies ist eine übliche Technik zum Anlegen und Abrufen einer Collection von Objekten.

Das Objekt `clsRabbit1` repräsentiert ein einzelnes Kaninchen und ist einfach. Jedes Kaninchen hat eine Farbe und eine Nummer. Die Nummern werden in der Reihenfolge der »Geburt« der Kaninchen vergeben. Das Klassen-Modul `clsRabbit1` sehen Sie im folgenden Listing.

```
' Guide to the Perplexed:
' Rabbit1 example
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

' Farbe des Kaninchens
Public Color As String

Private m_RabbitNumber As Long

Public Property Get Number() As Long
    Number = m_RabbitNumber
End Property
```

```
' Wir verwenden einen Zähler in einem Standard-Modul,
' um die Anzahl der angelegten Kaninchen nachzuhalten
Private Sub Class_Initialize()
    RabbitCounter = RabbitCounter + 1
    m_RabbitNumber = RabbitCounter
    ' Zufallsfarbe zuweisen
    Select Case Int(Rnd * 6)
        Case 0
            Color = "White"
        Case 1
            Color = "Pink"
        Case 2
            Color = "Grey"
        Case 3
            Color = "Blue"
        Case 4
            Color = "Brown"
        Case Else
            Color = "Black"
    End Select
    Debug.Print "Rabbit " & m_RabbitNumber & " born."
End Sub

Private Sub Class_Terminate()
    Debug.Print "Rabbit " & m_RabbitNumber & " died."
End Sub

' Kaninchen impfen
Public Sub Inoculate()
    ' In diesem Beispiel passiert hier noch nichts
End Sub
```

Die Eigenschaft `Color` ist ein `String`, der die Farbe des Kaninchens enthält. Sie wird im Ereignis `Class_Initialize` zufällig ausgewählt. In einer robusten Komponente sollten Sie diese Eigenschaft am besten als schreibgeschützte Eigenschaft mit einer privaten Variable anlegen, wie es hier auch bei der Eigenschaft `Number` mit der privaten Variablen `m_RabbitNumber` zu sehen ist.

Wie kann einem jedem neuangelegten Objekt einer Klasse eine sequentielle Nummer zugewiesen werden? Dazu brauchen Sie einen Zähler, der global im Projekt ist. Die Variable `RabbitCounter` ist im Modul `mdRbt1.bas` definiert. Die Zähler-Variable muß in einem Standard-Modul gehalten werden, da sie global sein soll und statische Variablen innerhalb einer Klasse nur mit der jeweiligen Instanz der Klasse verbunden sind. Der Zähler wird beim Anlegen eines Objekts während des `Class_Initialize`-Ereignisses erhöht, und der aktuelle Wert wird der privaten Variablen `m_RabbitNumber` in der Klasse zugewiesen. Beachten Sie, daß diese Technik nicht gleichermaßen bei Multithread-Servern funktioniert – aber darum werden wir uns in Kapitel 14 kümmern.

In den Ereignissen `Class_Initialize` und `Class_Terminate` wird jeweils über die Anweisung `Debug.Print` das Anlegen und Freigeben der Objekte verfolgt. Die Kombination der Verwendung einer globalen Variablen zur Zuweisung einer eindeutigen Objekt-Kennung und der `Debug.Print`-Anweisung zur Verfolgung des Anlegens und Freigebens von Objekten ist eine gebräuchliche und hilfreiche Technik – im nächsten Kapitel wird das noch deutlicher werden.

Schauen wir uns nun das Test-Programm `RbtTest1.vbp` an. Dieses Projekt enthält ein Formular und ein einzelnes Klassen-Modul, das ein Fuchs-Objekt darstellt. Sie sehen das Formular `frmRabbitTest` in Abbildung 12.1 in Aktion.

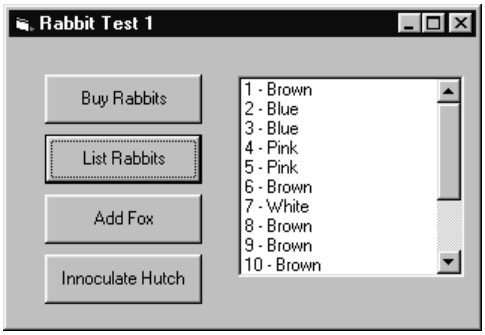

Abb. 12.1: Das Formular frmRabbitTest in Aktion

Dieses Formular enthält vier Schaltflächen und eine ListBox. Über die Ereignis-Prozedur `cmdBuy_Click` ruft ein Klick auf die Schaltfläche BUY Rabbits die Methode `BuyRabbits` der Klasse `PetStore1` auf. Dort wird der Variablen `Hutch` (Käfig) vom Typ `Collection` die von der `BuyRabbits`-Methode zurückgegebene Collection zugewiesen. Das folgende Listing zeigt dies und den übrigen Code des Formulars.

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

Option Explicit

' Die Tierhandlung für den Einkauf
Dim PetStore As New PetStore1

' Käfig für die eingekauften Kaninchen
Dim Hutch As Collection

' Einige Kaninchen kaufen
Private Sub cmdBuy_Click()
    Set Hutch = PetStore.BuyRabbits(15)
End Sub
```

```
Public Sub ListRabbits()
    Dim obj As clsRabbit1

    lstRabbits.Clear    ' Liste leeren
    For Each obj In Hutch
        lstRabbits.AddItem obj.Number & " - " _
         & obj.color
    Next
End Sub

' Alle im Käfig befindlichen Kaninchen auflisten
Private Sub cmdList_Click()
    ListRabbits
End Sub

' Die Prozedur imitiert einen Bug
Private Sub cmdAddFox_Click()
    Dim obj As New clsFox1
    Hutch.Add obj
End Sub

' Alle Kaninchen impfen
Private Sub cmdInoculate_Click()
    Dim obj As Object
    For Each obj In Hutch
        obj.Inoculate
    Next
End Sub
```

In der Ereignis-Prozedur `cmdList_Click` der Schaltfläche LIST RABBITS wird eine Liste der im Käfig befindlichen Kaninchen in die ListBox eingelesen.

Die Rückgabe einer Collection durch die Methode `BuyRabbits` der Klasse `PetStore1` weist Nachteile auf. Da eine Collection Elemente jeden Datentyps enthalten kann, kann versehentlich per Code ein Element einer falschen Klasse in die Collection eingefügt werden. Dies wird in der Ereignis-Prozedur `cmdAddFox` demonstriert. Hier wird ein Objekt der Klasse `clsFox1` (die zur Anwendung gehört) in die Käfig-Collection eingefügt.

Betätigen Sie zunächst einmal die Schaltfläche ADD FOX und versuchen es danach mit den Schaltflächen LIST RABBITS oder INNOCULATE HUTCH. Die Operation `ListRabbits` wird während der Iteration (mit `For...Each`) durch den Käfig (Hutch) fehlschlagen. Visual Basic wird auch beim Objekt der `clsFox1` versuchen, auf ein Interface `clsRabbit1` zuzugreifen, da die Enumerations-Variable als `clsRabbit1` deklariert ist. Das klappt natürlich nicht, da das `clsFox1`-Objekt kein Interface vom Typ `clsRabbit1` besitzt (Sie können das selbstver-

ständlich zum Laufen bringen, indem Sie das Interface `clsRabbit1` implementieren – doch aus welchem Grund sollte ein Fuchs ein Kaninchen implementieren??).

Die Ereignis-Prozedur `cmdInoculate` wird auf andere Weise fehlschlagen. Da hier das Enumerations-Objekt als `As Object` deklariert ist, kann sie auch auf ein `clsFox1`-Objekt verweisen. Diese Prozedur wird erst dann fehlschlagen, wenn Visual Basic den spät gebundenen Aufruf der Methode `Inoculate` versucht. Diese Methode ist nicht im `clsFox1`-Objekt vorhanden, so daß ein Fehler ausgelöst wird, wenn Visual Basic den Zugriff darauf versucht.

Diese kleinen Fehler-Szenarios sind schon schlimm genug. Aber was wäre, wenn statt der Rückgabe einer Collection, die den Käfig darstellen soll, die `Hutch`-Collection als öffentliche Eigenschaft der Klasse `PetStore1` implementiert würde (also die Komponente die Collection verwaltet anstelle des Formulars)? Damit würden Tür und Tor für andere Anwendungen geöffnet, unerwünschte (illegale) Daten in die Variablen der Komponente einschleusen zu können – das ist ein großes Problem.

Was können wir daraus schließen? Objekte in einer Komponente sollten niemals `Collection`-Objekte enthalten, die öffentlich sind. Es Client-Anwendungen zu erlauben, nach Gutdünken auf Collections Ihrer Komponente zuzugreifen, bedeutet geradezu eine Einladung zum Mißbrauch.

Die Rückgabe eines `Collection`-Objekts, das von einer Komponente angelegt wurde, ist nicht ganz so schlimm. Denn immerhin weiß dann Ihre Anwendung, daß sie eine gültige Collection von der Komponente erhält. Dann können falsche Daten nur noch auf der Client-Seite in die Collection gelangen und nur den Client beeinträchtigen, solange die Collection nicht wieder an die Komponente zurückgegeben wird. Gibt Ihre Komponente Collections zurück, die nur vorübergehend gebraucht werden, etwa zum Transport einer größeren Menge von Datenelementen, die die Anwendung recht bald verarbeitet, und es unwahrscheinlich ist, daß die Collection auf der Client-Seite noch verändert wird, dann ist die Rückgabe einer Collection durch eine Komponente eine verhältnismäßig sichere Angelegenheit und es lohnt sich nicht, nach einem anderen Ansatz zu suchen.

12.2.2 Die Blockhütte

Einen etwas robusteren Ansatz erhalten Sie, wenn Sie die `Hutch`-Collection in der `PetStore`-Klasse unterbringen. Allerdings sollte sie dann nicht über eine öffentliche Variable offengelegt werden (was dem vorangegangenen Beispiel entsprechen würde). Statt dessen ist sie als private Collection implementiert und es sind Methoden zur `PetStore`-Klasse hinzugefügt worden (siehe nächstes Listing). `PetStore2` ist Teil der Projekte `Rabbit2.vbp` und `RbtTest2.vbp`, die wiederum Teil der Projektgruppe `Rabbit2.vbg` sind.

```
' Guide to the Perplexed:
' Rabbit1 example
```

```
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

Private m_Hutch As Collection

' Kauf von Kaninchen
Public Sub BuyRabbits(ByVal RabbitCount As Integer)
    Dim counter%
    Dim col As New Collection
    Dim obj As clsRabbit2
    ' Anlegen der gewünschten Anzahl von
    ' Kaninchen-Objekten
    For counter = 1 To RabbitCount
        Set obj = New clsRabbit2
        col.Add obj
    Next counter
    ' Festhalten der neuen Kaninchen-Collection
    Set m_Hutch = col
End Sub

' Auf Elemente der Hutch-Collection zugreifen
Public Function Hutch(ByVal idx%) As clsRabbit2
    Set Hutch = m_Hutch(idx)
End Function

' Anzahl der Kaninchen abfragen
Public Function RabbitCount()
    RabbitCount = m_Hutch.Count
End Function
```

Da hier nicht mehr direkt auf die `Hutch`-Collection zugegriffen werden kann, ist die Implementierung einer separaten `RabbitCount`-Methode zur Ermittlung der Anzahl der Elemente in der Collection notwendig.

Die Klasse `clsRabbit2` bleibt gegenüber der Klasse `clsRabbit1` bis auf den neuen Klassen-Namen unverändert. Dieser neue Ansatz erfordert jedoch Änderungen im Formular `RabbitTest`.

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

Option Explicit

' Tierhandlung zum Kaninchen-Kauf
Dim PetStore As New PetStore2

' Kauf von Kaninchen
```

```
Private Sub cmdBuy_Click()
    PetStore.BuyRabbits 15
End Sub

Public Sub ListRabbits()
    Dim obj As clsRabbit2
    Dim counter%

    lstRabbits.Clear    ' Liste leeren
    For counter = 1 To PetStore.RabbitCount
        lstRabbits.AddItem _
         PetStore.Hutch(counter).Number & " - " _
         & PetStore.Hutch(counter).color
    Next
End Sub

' Die Prozedur imitiert einen Bug
Private Sub cmdAddFox_Click()
    MsgBox "Can't add a fox to a hutch"
End Sub

' Alle Kaninchen impfen
Private Sub CmdInoculate_Click()
    Dim obj As Object
    Dim counter%

    For counter = 1 To PetStore.RabbitCount
        PetStore.Hutch(counter).Inoculate
    Next
End Sub

' Alle im Käfig enthaltenen Kaninchen auflisten
Private Sub cmdList_Click()
    ListRabbits
End Sub
```

Der größte Vorteil dieses Ansatzes liegt darin, daß nun nicht mehr ungültige Objekte in die `Hutch`-Collection eingefügt werden können, da kein Zugriff mehr auf die Collection besteht.

Was ist an diesem Ansatz so abwegig, daß Microsoft ihn »nur« als Blockhütte betrachtet? Nun, erstens können Sie beispielsweise nicht mehr per `For...Each` durch die Elemente der Collection iterieren. Und zweitens können zwar nun nicht mehr ungültige Objekte von externen Clients eingefügt werden, aber das könnte immer noch in der Klasse `PetStore2` geschehen.

Ich denke, diese beiden Punkte sind nicht gerade weltbewegend. Denn wenn auch die Klasse `PetStore2` relativ komplex ist, sollten Sie eigentlich keine Probleme

mit dem Einfügen von ungültigen Objekten bekommen. Außerdem will mir nicht so recht einleuchten, welch schwerwiegenden Nachteil Sie davon haben sollten, keine `For...Each`-Schleife verwenden zu können. Es ist doch kein Problem, eine Zähler-abhängige Schleife zu verwenden. Vergleichen Sie einmal:

```
' Iterieren mit For...Each
Public Sub InnoculateAll1()
    Dim obj As clsRabbit2
    For Each obj In m_Hutch
        obj.Inoculate
    Next
End Sub

' Iterieren mit Zähler
Public Sub InnoculateAll2()
    Dim obj As clsRabbit2
    Dim counter As Integer
    For counter = 1 To m_Hutch.Count
        Set obj = m_Hutch.Item(counter)
        obj.Inoculate
    Next
End Sub
```

Der einzige Unterschied zwischen beiden Schleifen-Varianten tritt nur dann zutage, wenn ein Element während des Schleifendurchlaufs aus der Collection entfernt werden soll. In so einem Fall ist eine `For...Each`-Schleife etwas einfacher zu implementieren, da sie automatisch das jeweils richtige, nächste Objekt liefert. Bei einer Zähler-Schleife müssen Sie dagegen die Collection von hinten abwärts zählend durchlaufen, damit das Fehlen eines entfernten Objekts nicht die Zählung durcheinanderbringt.

Wenn Sie nicht vorhaben, den Code zur Verwaltung von Gruppen von Objekten (in diesem Fall Kaninchen) wiederzuverwenden, dann können Sie loslegen und diesen Ansatz wählen, wenn er Ihnen passend erscheint. Vor allem ersparen Sie sich damit die Mühe, die Funktionalität einer Collection selbst programmieren zu müssen.

12.2.3 Das Ziegelhaus

Die robusteste Lösung zur Verwaltung von Gruppen von Objekten dürfte das Erstellen einer eigenen Collection sein, die speziell auf den Umgang mit diesen Objekten abgestimmt ist. Die Projekte `Rabbit3.vbp` und `RbtTest3.vbp` der Projektgruppe `Rabbit3.vbg` illustrieren diese Technik.

Dieses Beispiel greift wieder den Ansatz des ersten Beispiels (Projektgruppe `Rabbit1`) auf, wo die `Hutch`-Collection in einer Formular-Variablen gespeichert und von der `BuyRabbits`-Methode der `PetStore`-Klasse zurückgegeben wurde. Der Unterschied besteht nun darin, daß keine generische Visual-Basic-Collection

zurückgegeben wird, sondern eine neue Collection namens `RabbitCollection3` (siehe folgendes Listing). Das `RabbitCollection3`-Objekt kann nur `clsRabbit3`-Objekte aufnehmen. Es verwendet die interne `m_Hutch`-Collection zur Aufnahme der Objekte.

```
' 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

' An die Collection delegieren
Public Sub Add()
    ' Neues Objekt anlegen
    Dim obj As New clsRabbit3
    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

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

' Interne Collection initialisieren und freigeben
Private Sub Class_Initialize()
    Set m_Hutch = New Collection
End Sub

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

Dieser Ansatz bietet einige Vorteile. Zunächst erlaubt die `NewEnum`-Funktion wieder die Verwendung einer `For...Each`-Schleife zur Iteration durch die Collec-

tion. Dazu muß die Klasse über eine Eigenschaft (oder Funktion) mit dem Namen NewEnum verfügen, die ein IUnknown-Objekt zurückgibt (das allgemeine Interface eines jeden Objekts). In dieser Eigenschaft wird die [_NewEnum]-Eigenschaft des internen Collection-Objekt abgefragt (die eckigen Klammern um den Namen sind notwendig, da der Unterstrich als Markierung für eine verborgene Eigenschaft ein in Visual-Basic-Namen ungültiges Zeichen ist). Weiterhin müssen Sie im Prozedur-Attribute-Dialog als Prozedur-ID den Wert -4 angeben. Visual Basic weiß anhand dieser Prozedur-ID, daß es sich um einen Enumerator handelt. Sie sollten diese Eigenschaft ebenfalls verbergen. Versuchen Sie nicht, diese Eigenschaft als Friend zu deklarieren – damit das ganze funktioniert, muß sie Public sein.

Der nächste Vorteil besteht darin, daß trotz der internen Collection m_Hutch, die ja alle Datentypen aufnehmen kann, über die Add-Methode der RabbitCollection3-Klasse nur clsRabbit3-Objekte der Collection hinzugefügt werden können. Somit kann keine Client-Anwendung ungültige Objekte der Collection hinzufügen.

Weiterhin haben Sie wegen des Zugriffs auf die interne Collection m_Hutch über die von Ihnen implementierten Methoden und Eigenschaften die volle Kontrolle über die unterstützten Objekt-Typen. Sie können auch zusätzliche Daten- und Fehlerprüfungen nach Belieben einfügen, um die Robustheit der Collection zu erhöhen.

Dazu ist eine private Collection dieses Typs leicht wiederzuverwenden. Der Visual-Basic-Klassengenerator beschleunigt das Erstellen einer eigenen Collection-Klasse. Anders als bei den übrigen Fähigkeiten des Klassengenerators hilft er hier einmal wirklich, Zeit zu sparen.

Schließlich sind Sie bei der Implementierung einer eigenen Collection-Klasse nicht nur auf die Nachbildung der Standard-Methoden und -Eigenschaften einer Collection beschränkt. Sie können weitere Methoden und Eigenschaften für jeden Zweck nach Belieben hinzufügen.

Betrachten wir den letzten Punkt einmal näher. Sie könnten beispielsweise eine Methode hinzufügen, die ein neues RabbitCollection3-Objekt zurückgibt, das nur die weißen Kaninchen aus m_Hutch enthält. Dazu müßten Sie die folgenden Funktionen in die Klasse einfügen:

```
' Die Methode AddExisting sollte nicht
' extern verfügbar sein
Friend Sub AddExisting(ExistingRabbit As clsRabbit3)
    m_Hutch.Add ExistingRabbit
End Sub

' Methode, die eine neue Collection zurückgibt,
' die nur weiße Kaninchen enthält
Public Function GetWhiteRabbits() As RabbitCollection3
```

```
    Dim col As New RabbitCollection3
    Dim obj As clsRabbit3
    For Each obj In m_Hutch
        If obj.Color = "White" Then
            col.AddExisting obj
        End If
    Next
    Set GetWhiteRabbits = col
End Function
```

Die Methode `AddExisting` ist nur innerhalb des Projekts sichtbar und erlaubt es, ein bereits vorhandenes Kaninchen-Objekt in die Collection einzufügen (die `Add`-Methode legt dagegen immer ein neues Kaninchen-Objekt an). Dies können Sie mit folgendem Code im Formular testen:

```
' Liste der weißen Kaninchen abrufen
Private Sub cmdWhite_Click()
    Set Hutch = Hutch.GetWhiteRabbits()
    ListRabbits
End Sub
```

Auch dieser Ansatz wartet mit ein paar Nachteilen auf. Ein größerer Nachteil ist der höhere Code-Aufwand – etwas höher als beim »Blockhütten«-Ansatz und wesentlich höher, als wenn nur eine einfache Collection zurückgegeben würde. Ein kleinerer Nachteil ist die Verdoppelung der Objekt-Anzahl – jeweils ein Objekt der neuen Collection-Klasse und darin enthalten das interne `Collection`-Objekt.

Zusammenfassend läßt sich sagen: Wenn Sie eine Collection als öffentliche Eigenschaft implementieren wollen, definieren Sie eine eigene Collection (für deren Implementierung es auch noch bessere Möglichkeiten gibt, wie wir gleich sehen werden). Soll eine Client-Anwendung diese Collection erhalten und selbst bearbeiten (wie in diesem Beispiel), dann sollten Sie diesen Ansatz bevorzugen.

12.2.4 Das individuelle Eigenheim

Verzeihen Sie mir bitte die eigenmächtige Erweiterung der Analogie, doch waren bisher alle diese Haustypen sozusagen simple Fertighaustypen. Sie basierten auf dem generischen, Visual-Basic-eigenen Collection-Objekt, das – wie das eben bei Fertighäusern so ist – in der Regel den üblichen, durchschnittlichen Ansprüchen genügt. Ihren Dienst erfüllen sie in der Regel recht gut. Es ist nicht unbedingt der effizienteste Ansatz und er kann vielleicht auch nicht alle Komfortansprüche befriedigen, aber Sie können ganz gut damit leben.

Aber wenn Sie sich den (Zeit-)Aufwand leisten können, oder wenn Sie unbedingt Ihr Traumhaus wollen, dann führt kein Weg daran vorbei, dieses von einem Architekten maßschneidern zu lassen.

Soweit die Analogie.

Die Projektgruppe Rabbit4.vbg enthält zwei Projekte, Rabbit4.vbp und RbtTest4.vbp. Auch hier wieder hat sich die Numerierung in den Namen der Module erhöht, von 3 auf 4. Der DLL-Server des Projekts Rabbit4 ähnelt dem aus Projekt Rabbit3.vbp, nur daß diesmal zwei verschiedene Lösungen für Gruppen von clsRabbit4-Objekten vorhanden sind. Das Objekt RabbitCollection4 basiert wie die RabbitCollection3 auf einer generischen Visual-Basic-Collection. Es wurden lediglich drei Methoden geändert. GetWhiteRabbits gibt hier nun eine einfache Collection statt eines weiteren RabbitCollection4-Objekts zurück. Diese Änderung soll einem faireren Vergleich zum neuen Array-basierten Ansatz dienen.

```
' Rückgabe einer neuen Collection,
' die nur die weißen Kaninchen enthält
' Des fairen Vergleichs wegen verwenden wir hier eine
' generische Collection
Public Function GetWhiteRabbits() As Collection
    Dim col As New Collection
    Dim obj As clsRabbit4
    For Each obj In m_Hutch
        If obj.Color = "White" Then
            col.Add obj
        End If
    Next
    Set GetWhiteRabbits = col
End Function
```

Mit der neuen Methode SellRabbit kann ein Kaninchen aus der Collection entfernt werden. Als Parameter der Methode wird eine Objekt-Referenz auf ein clsRabbit4-Objekt übergeben, nach dem die Collection durchsucht wird.

```
' Ein bestimmtes Kaninchen verkaufen
Public Function SellRabbit(rabbit As clsRabbit4) As Long
    Dim counter&
    Dim RabbitCount As Long
    ' Beachten Sie die leichte Optimierung, indem _
    ' m_Hutch.Count aus der Schleife herausgenommen wird
    RabbitCount = m_Hutch.Count
    For counter = 1 To RabbitCount
        If m_Hutch(counter) Is rabbit Then
            m_Hutch.Remove counter
            Exit Function
        End If
    Next counter
    SellRabbit = -1 ' Fehlermeldung im API-Stil
End Function
```

Einen vollkommen anderen Ansatz für eine Gruppe von `clsRabbit4`-Objekten wird in der Klasse `RabbitArray4` (`RbtArry4.cls`) im folgenden Listing gezeigt. In dieser Klasse werden die `clsRabbit4`-Objekte in einem Array statt einer Collection gehalten. Wegen der Verwendung eines Arrays kann diese Klasse nicht die Vorteile einer eingebetteten Collection bieten, wie Schlüssel, Unterstützung der `For...Each`-Syntax und Unterstützung jedes Datentyps. Allerdings werden diese Features in einem Anwendungsfall wie diesem auch nicht benötigt – es werden sowieso nur `clsRabbit4`-Objekte bearbeitet. Aber könnten Sie diese Features gegebenenfalls trotzdem nachrüsten? Ja, das könnten Sie. Sie könnten ein `Variant`-Array zur Unterstützung beliebiger Datentypen verwenden. Sie könnten ein separates Array von Strings, Longs oder Variants zur Unterstützung von Schlüsseln verwenden. Und Sie könnten mit Hilfe eines Zusatzprodukts, wie etwa mit SpyWorks von Desaware, die `For...Each`-Unterstützung einbauen.

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

Option Explicit

' Lokale Variable für Array von Kaninchen-Objekten
Private m_Hutch() As clsRabbit4
Private m_LastValidEntry As Long
Private m_HutchSize As Long

Public Sub Add()
    ' Neues Objekt anlegen
    Dim obj As New clsRabbit4
    ' Sicherstellen, daß das Array groß genug ist
    On Error GoTo AddResizeError
    If m_HutchSize = m_LastValidEntry Then
        ' Beliebige Auflösung beim Einfügen
        m_HutchSize = m_HutchSize + 4
        ReDim Preserve m_Hutch(m_HutchSize)
    End If
    m_LastValidEntry = m_LastValidEntry + 1
    Set m_Hutch(m_LastValidEntry) = obj
    Exit Sub
AddResizeError:
    ' Memory-Allocation-Fehler hier auslösen
End Sub

Public Property Get Count() As Long
    Count = m_LastValidEntry
End Property

Public Property Get Item(IndexKey As Long) As clsRabbit4
    Set Item = m_Hutch(IndexKey)
```

```
End Property

' Kaninchen von angegebener Position entfernen
Public Sub Remove(IndexKey As Long)
    Dim counter&
    If IndexKey < 0 Or IndexKey > m_LastValidEntry Then
        ' Eventuell hier Fehler auslösen
        Exit Sub
    End If
    For counter = IndexKey To m_LastValidEntry - 1
        Set m_Hutch(counter) = m_Hutch(counter + 1)
    Next counter
    Set m_Hutch(m_LastValidEntry) = Nothing
    m_LastValidEntry = m_LastValidEntry - 1
    ' Array verkleinern
    If m_LastValidEntry + 4 < m_HutchSize Then
        ReDim Preserve m_Hutch(m_LastValidEntry + 4)
        m_HutchSize = m_LastValidEntry + 4
    End If
End Sub

' Initialisierung und Freigabe


' Alle Objekte im Array freigeben
Private Sub Class_Terminate()
    ReDim m_Hutch(0)
End Sub

' Rückgabe einer neuen Collection,
' die nur die weißen Kaninchen enthält
' Des fairen Vergleichs wegen verwenden wir hier eine
' generische Collection
Public Function GetWhiteRabbits() As Collection
    Dim col As New Collection
    Dim counter&
    For counter = 1 To m_LastValidEntry
        ' Array selbst ist per Definition früh gebunden
        If m_Hutch(counter).Color = "White" Then
            col.Add m_Hutch(counter)
        End If
    Next counter

    Set GetWhiteRabbits = col
End Function

' Angegebenes Kaninchen verkaufen
```

```
Public Function SellRabbit(rabbit As clsRabbit4) As Long
    Dim counter&
    Dim RabbitCount As Long
    For counter = 1 To m_LastValidEntry
        If m_Hutch(counter) Is rabbit Then
            ' Entfernen für klasseneigene Methode
            Remove counter
            Exit Function
        End If
    Next counter
End Function
```

In der Methode `Add` wird zunächst das Array `m_Hutch` vergrößert. Die tatsächliche Anzahl der in das Array aufgenommenen Objekte wird unabhängig von der Größe des Arrays verwaltet. Wenn ein neues Objekt in die Collection aufgenommen werden soll, wird zuerst geprüft, ob noch genügend Platz im Array vorhanden ist, indem die Variablen `m_HutchSize` und `m_LastValidEntry` miteinander verglichen werden. Wird weiterer Platz benötigt, wird das Array mit der Anweisung `ReDim` und der `Preserve`-Option (damit die im Array bereits vorhandenen Werte erhalten bleiben) vergrößert. Es wird mehr Platz geschaffen als eigentlich aktuell benötigt, da davon ausgegangen wird, daß voraussichtlich gleich mehrere Elemente hintereinander eingefügt werden. Die Vergrößerung um 4 in einem Schritt reduziert die Zahl der notwendigen `ReDim`-Operation um den Faktor 4. Dahinter steht eine Abwägung zwischen Speicherbedarf und Performance – mögliche Speicherverschwendung gegen Geschwindigkeitssteigerung. Die meisten Array-basierten Collections verwenden diese Technik. Wieviel Platz in einem Schritt geschaffen wird, bleibt Ihnen überlassen. Eine größere Zahl erhöht die Wahrscheinlichkeit von Speicherverschwendung, resultiert aber in höherer Geschwindigkeit bei den darauffolgenden Einfügungen.

Die Eigenschaften `Count` und `Item` entsprechen denen des Collection-basierten Ansatzes.

Die Methode `Remove` ist etwas komplexer. Da hier die Arbeit nicht an eine eingebettete Collection delegiert werden kann, müssen Sie das betreffende Element selbst aus dem Array entfernen. So wie das hier implementiert ist, sind Leerräume im Array nicht erlaubt. Sobald das zu entfernende Element ausfindig gemacht worden ist, müssen alle nachfolgenden Elemente aufgerückt werden.

Die Methode `GetWhiteRabbits` ist im Prinzip identisch zu der im `RabbitCollection4`-Objekt. Ein Unterschied beruht auf dem etwas anderen internen Zugriff auf die Collection bzw. das Array. Bei einer internen Collection müssen Sie das Objekt, mit dem Sie arbeiten, einer Variablen des Objekt-Typs `clsRabbit4` zuweisen. In der `RabbitCollection4` ist das mit einer `For...Each`-Schleife so gelöst:

```
Dim obj As clsRabbit4
For Each obj In m_Hutch
'...
```

Wenn Sie dies nicht machen, erfolgt ein spät gebundener Zugriff auf das Objekt, was einen bedeutenden Einfluß auf die Performance nach sich zieht. Beim Array-Ansatz ist dies nicht notwendig. Alle Zugriffe auf das Array können früh gebunden erfolgen, da das Array selbst bereits als vom Typ `clsRabbits4` deklariert ist. Daher erfolgt der Zugriff auch hier früh gebunden:

```
If m_Hutch(counter).Color = "White" Then
```

Die Methode `SellRabbit` sucht im Array nach dem als Parameter übergebenen Objekt und entfernt es, wenn es gefunden wird.

Beide Objekte, `RabbitCollection4` und `RabbitArray4`, bieten exakt die gleiche Funktionalität. Welches der Objekte erfüllt seine Aufgabe besser?

Performance-Tests

Benchmarks sind immer eine heikle Angelegenheit. Das Test-Projekt `RbtTest4.vbp` bemüht sich um einen fairen Vergleich des Collection- und des Array-basierten Ansatzes. Den Code sehen Sie im folgenden Listing:

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

Option Explicit

' Die Tierhandlung für den Einkauf
Dim PetStore As New PetStore4

' Wieder eine Variable für die Collection
Dim Hutch1 As RabbitCollection4
' Diesmal auch eine für das Array
Dim Hutch2 As RabbitArray4

' Kaninchen kaufen
Private Sub cmdBuy_Click()
    Dim tempdouble As Double
    Dim repetitions As Long
    repetitions = 1
    Dim counter As Long

    ' Ein clsElapsedTime-Objekt zur Zeitmessung
    Dim time1 As New clsElapsedTime
    Dim time2 As New clsElapsedTime
    tempdouble = Rnd(-1)    ' Zufällige Zahlenfolge erzeugen
    time1.StartTheClock
```

```
    For counter = 1 To repetitions
        Set Hutch1 = PetStore.BuyRabbits(10000)
    Next counter
    time1.StopTheClock
    tempdouble = Rnd(-1) ' Rücksetzen der zufälligen
                         ' Zahlenfolge
    time2.StartTheClock
    For counter = 1 To repetitions
        Set Hutch2 = PetStore.BuyRabbitArray(10000)
    Next counter
    time2.StopTheClock
    lstRabbits.AddItem "Collection Adds: " _
     & time1.Elapsed(repetitions) & " ms/10000"
    lstRabbits.AddItem "Array Adds: " _
     & time2.Elapsed(repetitions) & " ms/10000"
End Sub

' Verkauf des ersten vorhandenen Elements
Private Sub cmdSell_Click()
    Dim time1 As New clsElapsedTime
    Dim time2 As New clsElapsedTime
    Dim col1 As Collection
    Dim col2 As Collection
    Dim obj As clsRabbit4
    Set col1 = Hutch1.GetWhiteRabbits
    Set col2 = Hutch2.GetWhiteRabbits
    If col1.Count = 0 Then Exit Sub
    time1.StartTheClock
    Call Hutch1.SellRabbit(col1(1))
    time1.StopTheClock
    time2.StartTheClock
    Call Hutch2.SellRabbit(col2(1))
    time2.StopTheClock
    lstRabbits.AddItem "Sell White Col: " & time1.Elapsed() _
     & " ms"
    lstRabbits.AddItem "Sell White Array: " _
     & time2.Elapsed()& " ms"
End Sub

' Verkauf des letzten vorhandenen Elements
Private Sub cmdSell2_Click()
    Dim time1 As New clsElapsedTime
    Dim time2 As New clsElapsedTime
    Dim col1 As Collection
    Dim col2 As Collection
    Dim obj As clsRabbit4
    Set col1 = Hutch1.GetWhiteRabbits
```

```
    Set col2 = Hutch2.GetWhiteRabbits

    time1.StartTheClock
    Call Hutch1.SellRabbit(col1(col1.Count))
    time1.StopTheClock
    time2.StartTheClock
    Call Hutch2.SellRabbit(col2(col2.Count))
    time2.StopTheClock
    lstRabbits.AddItem "Sell White Col: " & time1.Elapsed() _
     & " ms"
    lstRabbits.AddItem "Sell White Array: " _
     & time2.Elapsed() & " ms"
End Sub

' Dauer des Auslesens der weißen Kaninchen
Private Sub cmdWhite_Click()
    Dim time1 As New clsElapsedTime
    Dim time2 As New clsElapsedTime
    Dim col1 As Collection
    Dim col2 As Collection
    Dim repetitions As Long
    Dim counter As Long
    repetitions = 5
    time1.StartTheClock
    For counter = 1 To repetitions
        Set col1 = Hutch1.GetWhiteRabbits
    Next counter
    time1.StopTheClock
    time2.StartTheClock
    For counter = 1 To repetitions
        Set col2 = Hutch2.GetWhiteRabbits
    Next counter
    time2.StopTheClock
    lstRabbits.AddItem "Find White Col: " _
     & time1.Elapsed(repetitions) & " ms/" & col1.Count
    lstRabbits.AddItem "Find White Array: " _
     & time2.Elapsed(repetitions) & " ms/" & col2.Count
End Sub

Private Sub Form_Unload(Cancel As Integer)
    Set Hutch1 = Nothing
    Set Hutch2 = Nothing
End Sub
```

Dieses Test-Programm enthält zwei Objekt-Variablen auf Modul-Ebene: `Hutch1` für das `RabbitCollection4`-Objekt und `Hutch2` für das `RabbitArray4`-Objekt. Diese Objekte werden in der Ereignis-Prozedur `cmdBuy_Click` mit jeweils

10.000 `clsRabbit4`-Objekten gefüllt. Da zur Festlegung der Kaninchen-Farbe Zufallszahlen verwendet werden, und weil die Position und Zahl der Kaninchen-Farbe Einfluß auf die späteren Tests hat, wird mit `Rnd(-1)` der Zufallszahlengenerator von Visual Basic zurückgesetzt, damit sowohl `Hutch1` als auch `Hutch2` exakt die gleiche Folge von Kaninchen enthalten.

Mit Hilfe von zwei `clsElapsedTime`-Objekten wird die Zeit zum Laden der beiden Collections gemessen. Diese Klasse beruht auf Code zur Messung verstrichener Zeit, der in früheren Beispielen dieses Buchs verwendet worden ist. Da ich diesen jedoch häufiger gebrauchen kann, habe ich mich dazu entschlossen, daraus eine wiederverwendbare Klasse zu machen. Diese Klasse sehen Sie im folgenden Listing. Die Startzeit wird über die Methode `StartTheClock` gesetzt, die Endzeit über die Methode `StopTheClock`. Die verstrichene Zeit wird in Millisekunden als String von der Methode `Elapsed` zurückgegeben.

```
' Elapsed time class
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

Private Declare Function GetTickCount& Lib "kernel32" ()

Private m_CreationTime As Long
Private m_StopTime As Long

' Aktualisieren der Startzeit. Der Aufruf sollte immer
' erfolgen, da die Initialisierung der Klasse nicht exakt
' kontrollierbar ist.
Public Sub StartTheClock()
    m_CreationTime = GetTickCount()
End Sub

' Markieren der Stopp-Zeit. Wird automatisch aufgerufen,
' wenn das erste Mal die verstrichene Zeit abgefragt wird
Public Sub StopTheClock()
    m_StopTime = GetTickCount()
End Sub

' Formatierter String mit der Zeit Mikrosekunden
Public Function Elapsed(Optional ByVal repetitions _
 As Long = 1) As String
    Dim timeval As Long
    If m_StopTime = 0 Then StopTheClock
    timeval = m_StopTime - m_CreationTime
    ' timeval < 0 zeigt an, daß StartTheClock
    ' noch nie aufgerufen wurde
    ' Gegebenenfalls hier Fehler auslösen
```

```
    If timeval < 0 Then timeval = 0
    ' timeval ist die Differenz in Millisekunden
    Elapsed = Format$(CDbl(timeval) / repetitions, "0.###")
End Function
```

Das Formular enthält 4 Schaltflächen (siehe Abbildung 12.2). Jede entspricht einem Benchmark-Test. Sie sollten die Schaltfläche BUY MANY RABBITS betätigen, bevor Sie eine der anderen Schaltflächen betätigen, damit die Objekte `Hutch1` und `Hutch2` geladen und gefüllt werden. Die Schaltfläche FIND WHITE mißt die Zeit, die zum Durchsuchen der Liste und zum Anlegen einer Collection der `clsRabbit4`-Objekte mit weißer Farbe benötigt wird. Damit können Sie sowohl die Zeit zum Durchsuchen einer Liste als auch zum Vergleich einer Eigenschaft messen.

Über zwei Schaltflächen können `clsRabbit4`-Objekte aus den Collections entfernt werden. Die Schaltfläche SELL FIRST WHITE RABBIT entfernt das erste gefundene weiße Kaninchen. Die Schaltfläche SELL LAST WHITE RABBIT entfernt das letzte Kaninchen. Wie Sie bald sehen werden, gibt es einen erheblichen Unterschied zwischen beiden Varianten.

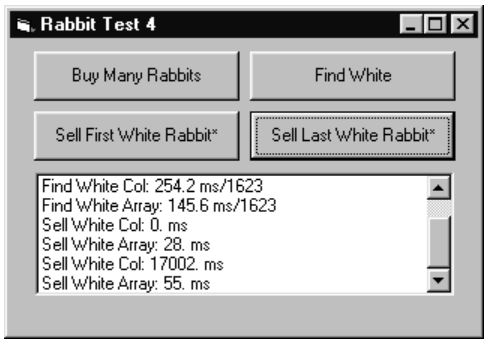

Abb. 12.2: Das Programm RabbitTest4 in Aktion

Ergebnisse

Bevor Sie die Tests ausführen, sollten Sie die DLL und die Test-Anwendung als Native-Code kompilieren. Dies gewährleistet den fairsten Test von beiden Ansätzen. Hier handelt es sich nämlich um Low-Level-Operationen, die am ehesten von nativem Code profitieren.

Tabelle 12.1 zeigt die Ergebnisse der Tests auf meinem System (bei Ihnen können sich andere Werte ergeben). Wie bei allen Benchmarks sind diese Ergebnisse nur mit Vorbehalt zu interpretieren.

Befehl	**Collection-basiert**	**Array-basiert**
Buy Many Rabbits	3816	2774
Find White Rabbits	248	126
Sell First White Rabbit	0	30
Sell Last White Rabbit	16554	30

Tab. 12.1: *Test-Ergebnisse von RabbitTest4*

Die Operation BUY MANY RABBITS ist beim Array-basierten Ansatz ungefähr 10 Prozent schneller. Bedeutet dies, daß der Array-Ansatz allgemein nur unwesentlich schneller ist als der Ansatz mit einer eingebetteten Collection? Nein. Bedenken Sie, daß die Messung den Overhead der Funktionen `BuyRabbit` und `BuyRabbitArray` im `PetStore`-Objekt einschließt. Hinzu kommt auch noch der Overhead des Anlegens der `clsRabbit4`-Objekte, wobei eine `String`-Zuweisung während der Festlegung der Farbe erfolgt. Dieser Overhead macht einen erheblichen Teil der Zeit aus. Es ist anzunehmen, daß bei einer Messung einer reinen Einfügen-Operation bei beiden Techniken der Array-Ansatz deutlich schneller sein dürfte als der Collection-Ansatz.

Der Array-Ansatz ist etwa 50 Prozent schneller als der Collection-Ansatz, wenn es um das Durchsuchen der Objekte und das Herausziehen des gewünschten Objekts geht. Die Operation wird fünfmal ausgeführt und anschließend wird der Durchschnittswert ermittelt, um etwas sauberere Ergebnisse zu erhalten.

Beim Entfernen eines Elements ergeben sich erhebliche Unterschiede zwischen dem Entfernen des ersten und des letzten Elements. Die Ergebnisse legen die Vermutung nahe, daß Collections äußerst schnell Objekte entfernen können, die sich am Anfang der Collection befinden. Der Array-Ansatz ist dagegen weniger schnell, wenn ein Element am Anfang des Arrays entfernt werden soll, da alle übrigen Elemente aufgerückt werden müssen, um den entstandenen Leerraum wieder zu füllen.

Schockierend ist hingegen, wie stark die Performance des Collection-Ansatzes einbricht, wenn ein Objekt am Ende der Collection entfernt wird. Da uns die interne Implementierung des `Collection`-Objekts verborgen bleibt, läßt sich nicht genau sagen, woran das liegt. Doch die Ergebnisse belegen eindeutig, daß in diesem Beispiel der Collection-Ansatz etwa 250mal langsamer als der Array-Ansatz ist.

Zusammenfassung

Bedeuten diese Ergebnisse nun, daß Sie künftig alle Ihre Collections um des Performance-Gewinns willen als Arrays anlegen sollten? Nein. So hat etwa ein Leser der vorigen Ausgabe dieses Buches mich mit Recht darauf hingewiesen, daß die Performance des Collection-basierten Ansatzes deutlich erhöht werden kann, wenn Schlüssel zur Identifizierung der Elemente verwendet werden. Ein Großteil der Zeit beim Entfernen eines Elements geht zu Lasten des Durchsuchens der Collection in der `For...Each`-Schleife. Bei einer Visual-Basic-Collection kann dagegen das zu entfernende Element äußerst schnell über den Schlüssel angesprochen werden. Läßt dies den von mir vorgenommenen Vergleich ungültig werden? Eigentlich nicht – denn die Array-basierte Collection könnte durch die Verwendung einer Hash-Tabelle und der entsprechenden Technik ebenfalls noch stark beschleunigt werden.

Die hier verwendeten Algorithmen mögen vielleicht tatsächlich etwas langsam sein, wenn es in der Praxis darum geht, Tausende von Objekten zu verwalten. Bei kleineren Anzahlen können Sie jedoch recht brauchbar sein. Der Array-basierte Ansatz erfordert jedoch zusätzlichen Code- und Test-Aufwand. Und dieser Aufwand wächst um so mehr, je mehr Features Sie der Collection hinzufügen.

Mit Ausnahme der `For...Each`-Unterstützung können Sie mit Visual Basic das `Collection`-Objekt exakt nachbilden. Unter Zuhilfenahme eines Zusatzprodukts wie SpyWorks von Desaware können Sie auch die `For...Each`-Unterstützung nachrüsten. Sie erhalten dadurch sogar eine größere Flexibilität und Kontrolle über die Enumerierungsfolge und das Einfügen und Entfernen von Elementen.

Wenn Sie aber tatsächlich eine exakte Nachbildung der Visual-Basic-Collection erstellen, werden Sie vermutlich herausfinden, daß die Performance nicht besser als die des Visual-Basic-Objekts sein wird. Der Vorteil des Array-Ansatzes liegt allein darin, daß nicht immer alle Features der Original-Collection benötigt werden und daher implementiert werden müssen.

Natürlich ist hier zwischen Performance und Entwicklungsaufwand abzuwägen. Aber das müssen Sie selbst anhand der Anforderungen Ihrer Anwendung entscheiden.

Der wahre Vorteil des Array-Ansatzes liegt darin, daß Sie ihn unendlich flexibel modifizieren können. Sie können auf traditionelle Algorithmen aus der Informatik zurückgreifen, wie etwa verkettete Listen, binäre Suche, oder Hash-Tabellen zur Optimierung der Suche, des Einfügens und des Entfernens, und sich so von Microsofts Implementierung des Collection-Objekts unabhängig machen.

Sie können ebenso Ihr eigenes Schlüsselschema entwickeln und sogar mehrfache Schlüssel vorsehen. Sie können anstelle der langsamen String-Schlüssel auch viel schnellere numerische Schlüssel verwenden, wenn Sie eine eigene Array-basierte Collection erstellen.

Denken Sie jedoch immer daran, daß eine gut konzipierte eigene Collection-Klasse wiederverwendbar sein sollte, so daß sich der höhere Aufwand auf lange Sicht bezahlt machen kann.

Zu guter Letzt verdeutlicht dieses Beispiel wieder einmal die überwältigenden Vorteile der objekt-orientierten Programmierung. Ist Ihnen aufgefallen, daß der Code, der die Objekte `Hutch1` und `Hutch2` bearbeitet, nahezu identisch ist (abgesehen von der Stelle, an der sie angelegt werden)? Dies bedeutet, daß Sie durchaus zunächst eine einfache Collection-basierte eigene Collection-Klasse anlegen und diese später durch eine interne Umstellung auf ein Array optimieren können!

Wie Sie sich sicher erinnern werden, brauchen Sie bei einem COM-Objekt lediglich das Interface unberührt beibehalten – die interne Implementierung können Sie beliebig modifizieren.

Wenn Sie sich nicht sicher sind, ob Sie Performance-kritischen Code schreiben, machen Sie erst einmal weiter, greifen Sie auf den Collection-basierten Ansatz zurück und vermeiden Sie die Verwendung von `For...Each`-Schleifen. Sie können sich später immer noch an die Optimierung der Collection-Klasse begeben, ohne den Code außerhalb der Klasse anzurühren. Befindet sich diese Klasse obendrein in einer separaten, kompilierten Code-Komponente, brauchen Sie noch nicht einmal die Client-Anwendungen rekompilieren.

Ihnen wird aufgefallen sein, daß das Test-Programm beim Beenden eine halbe Ewigkeit braucht. Das liegt daran, daß alle angelegten Objekte gelöscht werden müssen – das sind immerhin in beiden Collections zusammen 20.000 Objekte.

Da wir gerade vom Löschen sprechen – in dem vorliegenden RabbitTest-Beispiel kann ein Kaninchen nur über den Aufruf der `Sell`-Methode eines der Hutch-Objekte verkauft, also gelöscht werden. Sie könnten nun auf die Idee kommen, die Kaninchen-Klasse selbst mit einer `Sell`-Methode zu versehen. Das können Sie natürlich machen. Aber dann müßte das Kaninchen-Objekt selbst korrekt nachschauen, in welcher Collection es enthalten ist (schließlich kann sich ein Kaninchen nur schlecht in zwei verschiedenen Käfigen zugleich aufhalten). Die Idee mag einfach klingen. Doch wie Sie im nächsten Kapitel sehen werden, öffnen Sie damit die Tür zu einem der wichtigsten, manchmal verwirrendsten und häufig frustrierendsten Themen der ActiveX-Komponenten-Entwicklung: Objekt-Referenzen.