Kapitel 4

Das Component Object Model: Interfaces, Automation und Bindung

Als ich die Visual-Basic-Dokumentation las, stellte ich fest, daß Visual Basic die ActiveX-Funktionalität so gut gekapselt hat, daß ich wahrscheinlich das ganze Buch hätte schreiben können, ohne COM jemals zu erwähnen – so wie Microsoft es gemacht hat. Das Problem wäre dann allerdings, daß Sie als Leser über COM uninformiert blieben und keinen Schimmer davon hätten, ob das ein für Sie wichtiges Thema ist, oder was das Handbuch Ihnen eigentlich sagen will, wenn Microsoft weiterhin Begriffe wie *Automation*, *Interface* oder *QueryInterface* verwendet, ohne sie wirklich zu erklären. Meiner Meinung nach müssen Sie durchaus Bescheid wissen, was es mit Interfaces, Dispatch-IDs, früher und später Bindung, Prozeßräumen und allen diesen Dingen auf sich hat, die fast alle von Visual Basic handzuhaben sind, wenn Sie hochwertige ActiveX-Komponenten schreiben wollen. Sie werden einfach nicht umhin kommen, zumindest die Grundlagen des Component Object Models verstanden zu haben. Die Herausforderung besteht darin, diese Konzepte so deutlich darzustellen, daß auch Programmierneulinge sie verstehen können. Werde ich dort Erfolg haben, wo andere – wie soll ich sagen – gescheitert sind? Das Urteil überlasse ich Ihnen.

4.1 Merkmale von COM-Objekten

Kapitel 2, »ActiveX – eine historische (aber auch technische) Betrachtung« führte die Idee von COM ein. Ein COM-Objekt (hin und wieder auch »Windows-Objekt« genannt) zeichnet sich durch folgende Merkmale aus:

- Ein COM-Objekt kann Daten enthalten.
- Es verfügt über einen Satz oder mehrere Sätze von Funktionen, die *Interfaces* genannt werden und aufgerufen werden können, um Operationen mit den Daten oder anderer Art, wie etwa Darstellung oder Daten-Persistenz, auszuführen.
- Die Funktionen des Objekts sind in einer Anwendung oder einer Dynamic Link Library (DLL) enthalten. Sie sind in der Regel die einzigen Funktionen, die direkten Zugriff auf die Daten im Objekt haben.
- COM-Objekte können über einen GUID identifiziert werden, eine Kennung, die überall und für alle Zeiten eindeutig ist und bleiben wird.
- Der GUID kann vom System dazu verwendet werden, eine vorhandene Instanz eines Objekts mit der entsprechenden Anwendung oder DLL zu verknüpfen, die die Funktionen für dieses Objekt enthält (dieses Objekt also implementiert).

So weit, so gut.

Sie haben ebenfalls gesehen, daß für Visual-Basic-Programmierer Klassen die fundamentalen Mechanismen darstellen, eigene Objekte anzulegen. Und da VBA auf COM basiert, sind diese Klassen ebenfalls COM-Objekte. Die Eigenschaften und Methoden eines Objekts werden zum Interface für dieses Objekt – dem Satz der Funktionen, die das Objekt offenlegt.

An dieser Stelle würden wohl die C++-Programmierer, die COM wirklich verstanden haben, ihren E-Mail-Editor anwerfen und mir eine Message schreiben, mit etwa folgendem Inhalt: »Dan, wie können Sie sagen, daß die Eigenschaften und die Methoden des Objekts zum Interface des Objekts werden? Sie haben sich vorgenommen, COM zu erklären und haben statt dessen erfolgreich die Leser, für die das alles neu ist, in die Irre geführt, und die übrigen irritiert. Ihre Erklärung ist schlicht falsch.«

Mea culpa. Sie hätten recht.

Nun gut – vom Konzept her kann man sich Eigenschaften und Methoden eines Objekts als ein Interface des Objekts vorstellen. Doch wenn von COM-Objekten und ActiveX die Rede ist, hat der Begriff *Interface* eine spezielle Bedeutung, genauer gesagt, mehrere spezielle Bedeutungen. Um diese wird es im weiteren Verlauf dieses Kapitels gehen. Das Gute daran ist: Wenn Sie erst einmal verstanden haben, was Interfaces aus der COM-Perspektive darstellen, werden Sie auf dem richtigen Weg sein, ein Experte der ActiveX-Technologie zu werden. Die Konzepte dahinter bilden nämlich die Grundlagen für nahezu alles, was Sie dann mit ActiveX anstellen werden.

4.1.1 Inside COM

Sie wissen bereits, daß ein COM-Objekt einen Satz von Funktionen offenlegen soll. In Kapitel 2 habe ich davon gesprochen, daß ein COM-Objekt einen Satz von Funktionen offenlegen kann, über den es sich selbst in einem Container darstellen kann. Nicht alle COM-Objekte verfügen über eine visuelle Benutzeroberfläche, so daß natürlich nicht alle COM-Objekte auch über den speziellen Satz von Funktionen zur Darstellung verfügen. Das heißt also, daß nicht jedes COM-Objekt über ein Interface zum Betrachten des Objekts verfügt.

Ich habe auch davon gesprochen, daß ein COM-Objekt einen Satz von Funktionen offenlegen kann, über den es sich in eine Datei oder in einen OLE-Stream bzw. einen OLE-Storage schreiben kann. Und wiederum gilt, daß nicht alle COM-Objekte ihren Zustand zu sichern brauchen, also auch nicht diesen Satz von Funktionen unterstützen müssen. Nicht alle COM-Objekte verfügen über ein Interface, über das sie sich selbst in Streams und Storages sichern können.

Wenn Sie die beiden vorangegangenen Absätze zusammennehmen, ergibt sich logischerweise daraus eines der Schlüssel-Konzepte von COM: COM-Objekte können mehr als nur ein Interface unterstützen! Ein COM-Objekt kann, so es denn will, ein Interface namens `IViewObject2` implementieren, über das es sich im Device-Kontext eines Containers darstellen kann. (Erlauben Sie mir, vorläufig das Thema der Namensgebung von Interfaces zu überspringen – als Erklärung mag hier vorläufig genügen, daß ein jedes Interface immer über einen GUID identifiziert werden kann, es aber der Bequemlichkeit halber für Menschen lesbare Namen zusätzlich gibt.) Ein COM-Objekt kann, so es denn will, ein Interface namens `IStream` implementieren, über das es seine Daten in einen Stream schrei-

ben kann. Der Autor eines Objekts kann darüber hinaus beliebig viele Interfaces für sein Objekt definieren. Wird allerdings ein Interface implementiert, müssen auch alle Funktionen, die der Funktionssatz des Interface umfaßt, implementiert werden.

Ich verstehe, daß das ein wenig verwirrend klingen mag. Aber erlauben Sie mir, zunächst noch ein wenig weiter auszuholen, bevor wir uns das Ganze noch einmal aus einer anderen Perspektive anschauen.

Jedes Objekt muß letztlich mindestens ein Interface implementieren – ein ganz spezielles Interface namens `IUnknown`. Dieses Interface umfaßt drei Funktionen:

- `AddRef`
- `Release`
- `QueryInterface`

Jedes COM-Objekt enthält einen internen Referenzzähler, der protokolliert, wie oft es referenziert worden ist. Die Funktion `AddRef` des `IUnknown`-Interfaces erhöht den Referenzzähler. Die Funktion `Release` erniedrigt ihn. Wenn Sie ein Objekt freigeben (»Release«) und der Referenzzähler springt auf Null, dann weiß das System, daß das Objekt entfernt werden kann.

Wir werden uns später noch weiter über das Zählen der Referenzen unterhalten. Hier jedoch erst einmal ein kleines Code-Fragment, daß den Teil von Visual Basic hinter den Kulissen illustriert:

```
Dim ob As Object      ' Definiert die Variable ob.
' Noch existiert kein Objekt.
Dim ob2 As Object     ' Definiert die Variable ob2.
                      ' Noch existiert kein Objekt.
Set ob = New myobject ' Legt das Objekt myobject an.
                      ' Ruft AddRef für das Objekt auf.
                      ' Der Referenzzähler beträgt nun 1
Set ob2 = ob          ' Setzt o2 als Referenz auf das Objekt.
                      ' Ruft AddRef für das Objekt auf.
                      ' Der Referenzzähler beträgt nun 2
Set ob2 = Nothing     ' Setzt ob2 auf Nothing.
                      ' Ruft Release für das Objekt auf.
                      ' Der Referenzzähler beträgt nun 1
Set ob = Nothing      ' Setzt ob auf Nothing.
                      ' Ruft Release für das Objekt auf.
                      ' Der Referenzzähler beträgt nun 0
                      ' Das Objekt wird freigegeben und seine
                      ' Terminierungs-Funktion wird
                      ' aufgerufen
```

Nachdem Sie nun einmal eine Referenzzählung gesehen haben, vergessen Sie das Thema vorläufig wieder. Wir werden in Kapitel 13, »Lebensdauer eines Objekts«, wieder darauf zurückkommen.

Uns interessiert nun die dritte Funktion des `IUnknown`-Interfaces, die Funktion `QueryInterface`, die nur einen Zweck hat: Über diese Funktion kann das Objekt gefragt werden, ob es ein bestimmtes Interface unterstützt. Ist dies der Fall, kann das Objekt einen Zeiger auf die Funktionen dieses Interface zurückgeben. Was aber ist ein Zeiger auf eine Funktion?

Der Code eines Interface muß irgendwo im Arbeitsspeicher existieren. Das bedeutet, daß es zu jeder Funktion eine Speicheradresse gibt. Eine Variable, die eine Speicheradresse enthält, wird *Zeiger* genannt. Nun werden Sie sagen, daß Visual Basic doch gar keine Zeiger unterstützt – aber das stimmt nicht ganz. Sie mögen keine Zeiger-Variablen definieren können, aber eine Objekt-Variable ist tatsächlich nichts anderes als eine Zeiger-Variable. Es ist lediglich eine spezialisierte Art von Zeiger-Variablen, die einzig und allein auf OLE-Interfaces zeigen können – auf den Code einer Gruppe von Funktionen, die ein Interface implementieren.

Wenn Sie also einer Objekt-Variable ein Objekt zuweisen, haben Sie in Wahrheit hinter den Kulissen einen Zeiger auf ein Interface des Objekts erhalten und diesen der Objekt-Variablen zugewiesen, die in Wahrheit eine Zeiger-Variable darstellt.

Sie werden niemals explizit `QueryInterface` (oder `AddRef` oder `Release`) in Ihren Visual-Basic-Programmen aufrufen – implizit werden Sie das jedoch die ganze Zeit über tun.

Sie werden explizit nie ein Interface dieses Typs in Visual Basic anlegen, tatsächlich werden Sie eine spezielle Interface-Art anlegen, die *Automation-Interface* genannt wird.

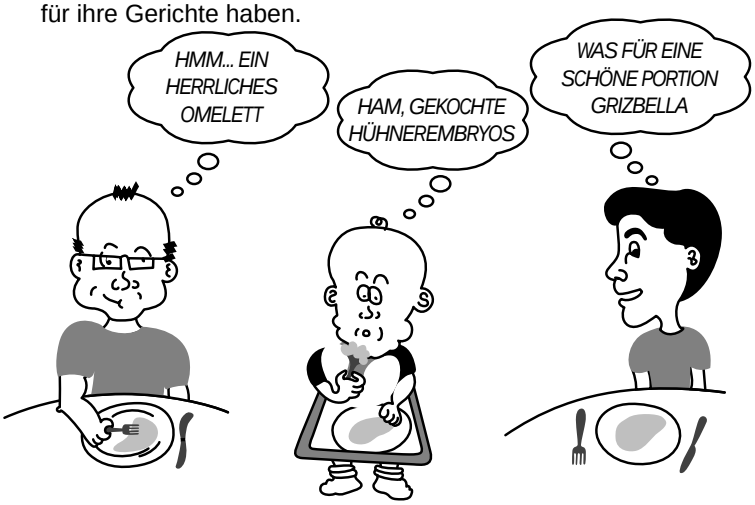
Das alles mag immer noch recht verwirrend klingen. Ich habe auch eine ganze Weile damit zu kämpfen gehabt. Ich habe aber gemerkt, daß es manchmal am leichtesten fällt, ein Konzept zu verstehen, wenn man aufhört, damit zu kämpfen und es locker anpackt. In diesem Sinne: Vergessen wir den technischen Jargon – blättern Sie um, und genießen Sie die Sonntags-COMics.

4.2 Der Sonntags-COMic

DER SONNTAGS-COMIC

GESCHRIEBEN VON DANIEL APPLEMAN
GEZEICHNET VON DAN SOHA

Stellen Sie sich eine Welt vor, in der es keine Restaurants gibt. Jeder ißt zuhause. Jeder kocht alles selbst. Niemand weiß, was andere Leute essen, so daß alle andere Namen für ihre Gerichte haben.



Ohne COM existieren Objekte nur im Kontext der eigenen Anwendung. Auch wenn sie ähnliche Funktionen haben, sind diese Funktionen und deren Parameter unterschiedlich benannt. Ein Objekt könnte über seine Funktion „Display“ dargestellt werden, ein anderes vielleicht über die Funktion „View“ – es gibt einfach keinen Standard.

Herr M. spricht bei einem Bankier vor, der skeptisch ist. Wie werden die Leute der Bedienung verständlich machen, was sie essen wollen?

Das Restaurant-Objekt hat zwei Interfaces, Frühstück und Mittagessen. Frühstück hat die ID Nr. 1, Mittagessen die ID Nr. 2. Das Frühstücks-Interface hat vier Funktionen: Die erste ist Puter, die zweite Hamburger usw.

Das Interface beschreibt nicht ausdrücklich, wie ein Gericht zu kochen ist – die Ausführung bleibt jedem Küchenchef selbst überlassen. Wenn Sie jedoch Waffeln bestellen, werden Sie mit Sicherheit in etwa das gleiche bekommen wie auch in einem anderen der Restaurants. Anders gesagt, wenn Sie eine Funktion eines Interface eines Objekts aufrufen, wird in etwa die gleiche Funktion ausgeführt, wenn Sie das gleiche Menü verwenden.

QueryInterface fragt nach dem Frühstücks-Interface. Man hat dort eins, so daß Sie die Funktion Nr. 2 des Interface aufrufen können. Funktion Nr. 2 ist immer eine Waffel, überall in der Welt, so daß es überflüssig ist, sich tatsächlich den Namen einer Funktion zu merken.

Die Restaurant-Idee griff wie ein Lauffeuer um sich. Jetzt konnten die Leute überall auf der Welt Mahlzeiten nach ihrem Geschmack bestellen.

Einige Restaurants spezialisierten sich auf ein Menü.

Nicht jeder war zufrieden...

Frau C. beschließt, ihre eigene Restaurant-Kette zu eröffnen, die eine neue Art von Menü anbietet.

Das bedeutet einen mittelschweren Schock für die Leute, die es gewohnt waren, mit der Angabe von nur zwei Nummern bestellen zu können. Würde sich die Idee durchsetzen?

Es ist ein Erfolg! Das spezielle Dispatch-Menü ermöglicht es den Restaurants, ein Standard-Menü zu verwenden und dennoch viele verschiedene Gerichte anzubieten.

Ein Dispatch-Interface enthält Funktionen, die es dem Objekt ermöglichen, weitere Funktionen zu definieren. Sie können eine Liste der verfügbaren Funktionen samt deren Parametern erfragen.

Das Dispatch-Interface bietet für jedes Objekt einen Weg an, beliebig viele Funktionen und Eigenschaften offenzulegen. Sie können eine Funktion über seine Dispatch-ID-Nummer (Nummer eines Gerichts in diesem Beispiel) aufrufen oder das Dispatch-Interface nach der Dispatch-ID fragen, die mit einer Funktion oder einer Eigenschaft verknüpft ist.

Manchmal treten jedoch Probleme auf. Die Leute haben sich daran gewöhnt, über das Dispatch-Menü per Nummer zu bestellen, doch manchmal ändert ein Restaurant die Mahlzeiten zu einer Nummer.

Seltsam genug – genau diese Szene ist mir selbst einmal passiert. Da gibt es eine lokale Sandwich-Kette namens Togo's, die lange Zeit mein Favorit gewesen ist. Mein Lieblings-Sandwich war deren Nr. 13, das Schinken-Sandwich. Eines Tages bestellte ich wie gewohnt die Nr. 13 und bekam ein anderes Sandwich. Sie hatten die Nummern geändert. Ich war absolut nicht glücklich.

Wenn Sie ein Dispatch-Menü definieren, ist es durchaus in Ordnung, wenn Sie weitere Funktionen hinzufügen. Sie sollten jedoch lieber nicht die Funktionen zu einer vorhandenen Dispatch-ID ändern. Anderenfalls werden Programme, die dieses Objekt verwenden, fehlerhaft arbeiten, und die Leute werden enttäuscht sein. Sie können dieses Problem umgehen, indem Sie immer über den Namen „bestellen“. Dies ist der Unterschied zwischen früher und später Bindung – einThema, auf das wir in diesem Kapitel noch zu sprechen kommen.

Das Dispatch-Menü wurde ziemlich populär. Einige Restaurants boten sogar mehrere Dispatch-Menüs an, jedes mit einer eigenen Reihe von Mahlzeiten. Und was wurde aus Herrn M.?

Die Leute verwenden natürlich auch weiterhin die Standard-Menüs, die ja immerhin einen weltweiten Standard darstellen. Jedoch verwendeten mehr und mehr Restaurants Dispatch-Menüs, um speziell angepaßte Auswahlen zu ermöglichen. Es war leicht zu bewerkstelligen, und die neuen Restaurants mit all ihrer Vielfalt erregten mehr Aufmerksamkeit als die alten.

Ein Dispatch-Interface wird auch Automation-Interface genannt. Und ActiveX-Automation wird dazu verwendet, all die Funktionen und Eigenschaften zu implementieren, die Ihre Visual-Basic-Klassen und ActiveX-Objekte definieren.

4.2.1 Interface-Namen und die Natur des Vertrags

Als Visual-Basic-Programierer erstellen Sie Automation-Interfaces, die zuvor beschriebenen Dispatch-Interfaces. Die Funktionen und Methoden des Interface, das in der Dispatch-Tabelle enthalten ist, betreffen Sie. Bevor wir uns mit dem Thema Automation beschäftigen, schauen wir uns jedoch noch die Natur von Nicht-Dispatch-Interfaces (Standard-COM-Interfaces) an. Viele Prinzipien dort betreffen nämlich ebenfalls Automation-Interfaces.

Jedes COM-Objekt legt mindestens ein Interface offen, das Interface `IUnknown`. Doch was bedeutet `IUnknown`? Entsprechend der Konvention erhalten Interfaces Namen, die mit dem Großbuchstaben »I« (für »Interface«) beginnen. Aber das ist einfach nur eine Konvention, um Interfaces in einer für Menschen lesbaren Form zu bezeichnen. Windows verwendet eigentlich Interfaces über deren GUID. Dies ist beispielsweise die GUID des `IUnknown`-Interfaces:

```
{00000000-0000-0000-C000-000000000046}
```

GUIDs sind, wie Sie sich erinnern werden, global eindeutig. Nachdem eine GUID einmal einem Interface zugewiesen worden ist, können Sie sich mit nahezu absoluter Sicherheit darauf verlassen, daß Sie immer das gleiche gewünschte Interface erhalten werden, wenn Sie über eine GUID nach einem Interface fragen, unabhängig von dem Objekt, mit dem Sie gerade arbeiten. OLE (bzw. ActiveX) definiert viele Standard-Interfaces, die zur Implementierung von OLE-Funktionalitäten von Datenaustausch bis Objekt-Einbettung reichen. Wenn Microsoft die Funktionalität von ActiveX erweitern möchte, braucht man dort lediglich ein neues Interface zu definieren. Die Liste der auf einem System verfügbaren Interfaces ist in der System-Registrierung und dem Schlüssel `HKEY_CLASSES_ROOT\Interface` zu finden (siehe Abbildung 4.1).

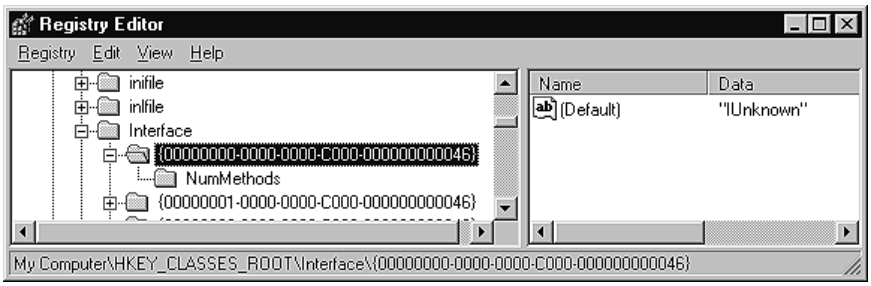


Abb. 4.1: Standard-Interfaces in der Registrierung ausfindig machen

Wir haben festgestellt, daß sich ein einmal definiertes Interface über seine GUID eindeutig im gesamten Universum identifizieren läßt. Es gibt aber noch ein weiteres wichtiges Kriterium, damit COM funktioniert: Sie müssen wissen, daß ein Interface eine Art Vertrag darstellt.

Ein *Vertrag* ist eine Übereinkunft. Was bedeutet es also, wenn wir sagen, daß ein Objekt ein Interface unterstützt? Es bedeutet, daß das Objekt alle Funktionen des Interface auf eine standardisierte Weise implementiert. Wenn wir sagen, daß alle COM-Objekte `IUnknown` implementieren, sagen wir tatsächlich, daß bei der Inanspruchnahme des `IUnknown`-Interface alle drei Funktionen aufgerufen werden können: `AddRef`, `Release` und `QueryInterface`. Diese Funktionen akzeptieren immer die gleichen Parameter und geben die gleichen Werte zurück, unabhängig davon, ob das Objekt etwa ein Bild, eine Animation oder ein Klang-Objekt ist. Die Funktionen folgen immer in der gleichen Reihenfolge in der Interface-Deklaration aufeinander und führen immer exakt die gleichen Operationen aus.

Jedoch wird von COM nicht vorgeschrieben, wie diese Funktionen ihre Aufgabe erfüllen – wie also die tatsächliche Implementierung im Code auszusehen hat. Ein Interface ist ein Vertrag und eine Spezifikation, aber kein Implentierungsdiktat.

Dies zieht signifikante Auswirkungen nach sich. Es bedeutet, daß Objekte in verschiedenen Programmiersprachen erstellt werden können, sogar für verschiedene Prozessorplattformen. So lange die Objekte Interfaces (die aus Zeigern auf Funktionen bestehen) auf standardisierte Weise offenlegen und die Funktionen für diese Interfaces die Parameter lesen und Werte zurückgeben können, wie es der COM-Standard vorschreibt, werden diese Objekte ihre Arbeit verrichten. Daher können mit Visual Basic erstellte Objekte von C++ oder von Access-Programmierern verwendet werden. Die Sprache und die Implementierung ist ohne Belang, nur das Interface und der COM-Standard zählen.

Daher kann eine Textverarbeitung, die ein Objekt darstellen möchte, das `IViewObject2`-Interface über die QueryInterface-Funktion anfordern. Wenn das Objekt dieses Interface unterstützt, kann die Textverarbeitung das Objekt darstellen. Sie braucht den Typ des Objekts nicht zu kennen – sie muß lediglich wissen, daß das Objekt das gewünschte Interface unterstützt. Das setzt natürlich voraus, daß ein Objekt, das den Anspruch erhebt, ein Interface zu unterstützen, dies auch vollständig und korrekt tut. Diesbezügliche Versäumnisse resultieren in Problemen von Funktionsfehlern bis hin zu System-Ausnahmen.

Der Interface-Vertrag schreibt folgendes fest:

- Eine GUID (die ID, die das Interface eindeutig identifiziert)
- Die Reihenfolge der Funktionen im Interface
- Die Parameter jeder Funktion
- Die Rückgabewerte jeder Funktion

Folgendes wird nicht festgelegt:

- Die Implementierung der Funktionen
- Die Sprache, in der die Funktionen implementiert werden

Aber was ist, wenn Sie ein Interface ändern müssen?

Sie werden dies tunlichst unterlassen.

Denn wenn Sie eine Funktion weglassen oder seine Parameter ändern, wird jede Anwendung, die dieses Interface verwendet, nicht mehr richtig mit Objekten umgehen können, die die neuere Definition des Interface verwenden. Wenn Sie eine Funktion hinzufügen, werden Anwendungen, die sich auf das neuere Interface verlassen, mit Objekten, die auf der Basis der ältere Definition entstanden sind, nicht mehr umgehen können. Wie herum auch immer – es würden schwerwiegende Probleme auftreten.

Die Lösung lautet, ein neues Interface mit einem neuen Namen anzulegen. So entstehen Interfaces wie etwa `IViewObject2`. Es ähnelt dem Interface `IViewObject`, außer daß es eine neue Funktion enthält. Dies löst das Problem in beiden Richtungen. Neuere Anwendungen und Objekte können gegebenenfalls die Vorteile des neueren Interface in Anspruch nehmen und trotzdem absolut darauf vertrauen, daß das Original-Interface nach wie vor reibungslos seine Dienste verrichtet.

Und was ist, wenn Sie ein Objekt haben, das ein Interface implementiert, Sie aber das Objekt neu codieren möchten, um beispielsweise dessen Performance zu erhöhen?

Dem steht nichts im Wege. Der Interface-Vertrag schreibt die Implementierung nicht vor. Solange Sie weder die Funktionsdeklarationen, die Reihenfolge der Funktionen im Interface noch die Arbeitsweise der Funktionen ändern, können Sie den Code jederzeit nach Belieben modifizieren. Und schließlich profitiert jede Anwendung, die das Objekt verwendet, unmittelbar und auf der Stelle von den Verbesserungen, die Sie vorgenommen haben.

4.3 Automation (Dispatch) Interfaces und Bindung

Das `IDispatch`-Interface umfaßt die folgenden Funktionen:

- `GetTypeInfoCount` – wird zur Feststellung verwendet, ob Type-Informationen für ein Interface zur Verfügung stehen.
- `GetTypeInfo` – dient zur Ermittlung der Type-Informationen eines Interface.
- `GetIDsOfNames` – dient zur Ermittlung der Dispatch-ID (`DispID`) einer Methode oder einer Eigenschaft.
- `Invoke` – darüber wird eine Methode aufgerufen oder eine Eigenschaft gesetzt bzw. ausgelesen.

Natürlich werden Sie diese Funktionen nie direkt aus Visual Basic aufrufen (zumindest nicht ohne Hilfe von Zusatz-Produkten oder -Komponenten). Hier geht es lediglich darum, daß Sie ein Gefühl dafür bekommen, was hinter den Kulissen vor sich geht, damit Sie besser verstehen, wie Ihre eigenen Komponenten funktionieren.

Von den angeführten Funktionen sind die letzten beiden die wichtigsten. Die Funktionen `GetTypeInfoCount` und `GetTypeInfo` werden benötigt, um die Liste der Methoden und Eigenschaften eines Interface durchgehen zu können. Type-Informationen enthalten nicht nur die Funktionsnamen, sondern auch detaillierte Informationen über Parameter, Datentypen der Parameter und Rückgabewerte. Ein Dispatch-Interface muß nicht unbedingt Type-Informationen liefern. Doch Sie brauchen sich keinesfalls darum zu kümmern, da Ihre Visual-Basic-Objekte diese Funktionen automatisch implementieren.

Jedes Dispatch-Interface kann beliebig viele Funktionen anbieten. Es gibt drei Typen von Funktionen:

- Aufruf einer Funktion (im engl. »Invoke«; kann aus Visual-Basic-Sicht sowohl eine Prozedur ohne Rückgabewert als auch eine Funktion mit Rückgabewert sein)
- Setzen einer Eigenschaft
- Lesen einer Eigenschaft

Nur auf diesem Wege kann ein Objekt bearbeitet werden, das ein Dispatch-Interface verwendet. Jede Methode oder Eigenschaft des Interface trägt eine *Dispatch-ID* – eine Kennziffer, die die Methode oder Eigenschaft identifiziert. Wenn auch jede Methode ihre eigene Dispatch-ID trägt, teilen sich die Funktionen zum Setzen und zum Lesen einer Eigenschaft eine gemeinsame Dispatch-ID.

Dies betrifft Ihre Visual-Basic-Objekte auf merkwürdige Weise, der Sie sich vielleicht nicht bewußt sind. Ich wurde des öfteren gefragt, ob es einen Performance-Unterschied dazwischen gibt, wenn eine Eigenschaft als öffentliche Variable eines Klassen-Moduls oder über die Property Get/Let-Funktionen implementiert wird. Die Antwort lautet: Es macht keinen Unterschied. Die Möglichkeit, eine Eigenschaft als öffentliche Variable zu deklarieren, ist lediglich eine von Visual Basic angebotene Sprach-Konvention. Intern wird auf die Eigenschaft in jedem Fall über separate Property Let- und Property Get-Funktionen zugegriffen. Dies ist der einzige Mechanismus, den eine Dispatch-Tabelle für den Zugriff auf Eigenschaften eines Objekts anbieten kann.

Die Funktion `GetIDsOfNames` des `IDispatch`-Interface ermöglicht es, die Dispatch-ID einer Methode oder Eigenschaft zu ermitteln, wenn deren Name vorliegt. `GetIDsOfNames` kann auch Kennungen der Parameter einer Methode oder Eigenschaft ermitteln. Somit können über diese Funktion sowohl die Dispatch-ID als auch die Parametertypen jeder Methode oder jeder Eigenschaft ermittelt werden.

Microsoft definiert eine Reihe von Standard-Dispatch-IDs, alle mit negativen Werten. So lautet beispielsweise die Dispatch-ID für die `Hwnd`-Eigenschaft eines ActiveX-Controls -515. Die Verwendung von Standard-Dispatch-IDs ermöglicht es ActiveX-Containern, verschiedene Methoden oder Eigenschaften unabhängig vom betreffenden Objekt einheitlich zu behandeln.

Die Funktion `Invoke` des `IDispatch`-Interface trägt die Hauptlast der Arbeit. Ihr werden eine bestimmte Dispatch-ID und ein Array von Varianten und Strukturen, die Datentypen-Parameter enthalten, übergeben, und sie ruft dann die eigentliche Funktion im Objekt auf.

Sie mögen sich an dieser Stelle über diese umständliche Abwicklung eines Funktionsaufrufs wundern. Denn selbst beim simpelsten Aufruf einer Methode, etwa einer Methode namens `MeineMethode`, muß folgendes Prozedere abgearbeitet werden:

1. Der Aufruf von `GetIDsOfNames`, um die Dispatch-ID zum Namen `MeineMethode` zu ermitteln.
2. Vorbereitung eines Variant-Arrays zur Aufnahme der Parameter der Methode.
3. Aufruf der `Invoke`-Funktion, um die Methode auszuführen.

Das ist tatsächlich recht aufwendig, hat aber einen großen Vorteil. Die Anwendung, die ein Objekt verwendet, benötigt vor der tatsächlichen Benutzung des Projekts keine näheren Informationen über das Interface. Sie kann die notwendigen Informationen zur Laufzeit herausfinden, nachdem das Objekt bereits existiert. Diese Vorgehensweise wird *Späte Bindung* (»Late Binding«) genannt.

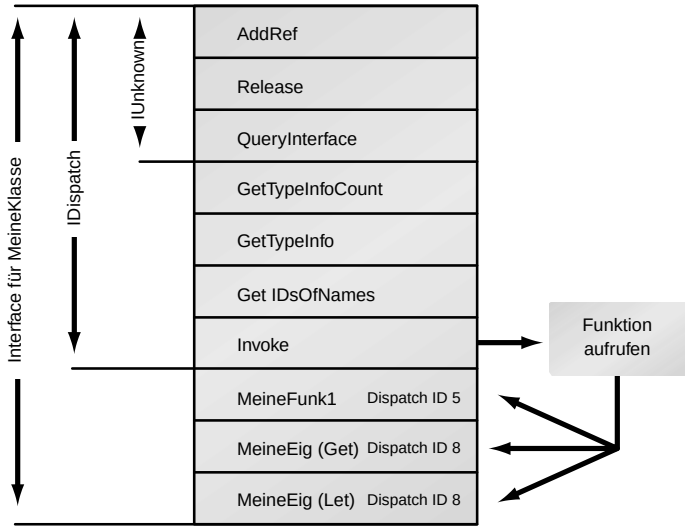
Eine späte Bindung erfolgt in Visual Basic immer dann, wenn Sie eine Objekt-Variable als `As Object` deklarieren, oder wenn Sie auf eine Methode oder Eigenschaft über die `CallByName`-Anweisung zugreifen. Eine als `As Object` deklarierte Variable kann jeden Objekttyp aufnehmen. Somit müssen auch alle Methoden und Eigenschaften eines Objekts unterstützt werden, ohne daß Visual Basic vorher wissen könnte, welche Methoden und Eigenschaften das zur Laufzeit sein werden. Ohne die späte Bindung und ohne das `IDispatch`-Interface wäre die Deklaration `As Object` nicht möglich. Die Anweisung `CallByName` ermöglicht den Aufruf einer Methode oder Eigenschaft über einen Text-String, der den Namen der gewünschten Methode oder Eigenschaft enthält – das ist exakt die Funktionalität der `Invoke`-Funktion des `IDispatch`-Interface. Tatsächlich macht `CallByName` kaum mehr, als die `Invoke`-Methode in `IDispatch` aufzurufen.

Wenn Sie nun aber den Typ des Objekts, das Sie referenzieren wollen, bereits vorher kennen? Gibt es einen Weg, die Performance-Bremse durch `IDispatch` zu umgehen?

Es gibt diesen Weg. Der Trick besteht darin: Wenn ein `IDispatch`-Interface implementiert wird, muß irgendwo eine Tabelle mit den Funktionszeigern auf die von dem Interface unterstützten Methoden und Eigenschaften vorhanden sein. Wenn nun ein Programm, das ein Objekt verwendet, frühzeitig genug die Positionen der Funktionen und von deren Parametern herausfinden kann, bräuchten Sie zur Laufzeit lediglich einen Zeiger auf diese Tabelle und könnten die Funktionen direkt aufrufen. Aber wie kann der Zeiger auf diese Tabelle zur Laufzeit angeboten werden? Ganz einfach – das ist genau das, was ein Standard-COM-Interface

macht. Und ein Interface, das sowohl ein `IDispatch`-Interface als auch ein Standard-Interface anbietet, wird *Duales Interface* (»Dual Interface«) genannt.

Ein duales Interface überläßt der Anwendung die Wahl. Nutzt eine Anwendung das direkte Interface eines Objekts, wird das als *Frühe Bindung* (»Early Binding«) bezeichnet. In Abbildung 4.2 sehen Sie, wie das intern funktioniert.




```
Klasse MeineKlasse

    Public Funktion MeineFunk1() As Integer
    Public Property Get MeineEig() As Variant
    Public Property Let MeineEig(vNewValue As Variant)
```

Abb. 4.2: Interne Funktionsweise eines Visual-Basic-Objekts

Wenn Visual Basic auf die Methoden oder Eigenschaften eines Objekts zugreifen will, ist zunächst eine Referenz auf `IUnknown` vorhanden. Ist die Variable als `As Object` deklariert, verwendet Visual Basic die Funktion `QueryInterface`, um eine Referenz auf das `IDispatch`-Interface zu erhalten (diese kann gegebenenfalls, muß aber nicht unbedingt der gleiche Wert sein). Dann kann Visual Basic die Funktion `Invoke` aufrufen, um beispielsweise die Methode `MeineMethode` oder aber die Eigenschaft `MeineEigenschaft` aufzurufen, die als Funktionspaar angelegt ist: eine Funktion zum Setzen der Eigenschaft, eine zweite Funktion zum Auslesen der Eigenschaft.

Ist die Variable als `As MeineKlasse` deklariert worden, wird die frühe Bindung verwendet. Visual Basic kann `QueryInterface` aufrufen, um eine Referenz auf das `MeineKlasse`-Interface zu erhalten – einem dualen ActiveX-Automation-Interface. Dieses Interface umfaßt sowohl die Interfaces `IUnknown` als auch `IDispatch`.

Visual Basic kann nun die Methode `MeineMethode` oder die Eigenschaft `MeineEigenschaft` direkt über dieses Interface aufrufen, ohne über den Umweg des `Invoke`-Aufrufs gehen zu müssen.

In Visual Basic erhalten Sie eine frühe Bindung, indem Sie einen Verweis auf ein Objekt anlegen (über den Verweise-Dialog), und dann eine Variable als spezifischen Objekttyp deklarieren, statt nur allgemein als `As Object`. Auf diese Weise wird der Zeitaufwand des Zugriffs auf Methoden und Eigenschaften merklich verringert.

Haben Sie sich schon darüber gewundert, warum den meisten Visual-Basic-Variablen Werte direkt zugewiesen werden können, bei Objekt-Referenzen jedoch die `Set`-Anweisung verwendet werden muß? Das liegt daran, daß dahinter ein vollkommen anderer Vorgang steht.

Wenn Sie eine Variable einer anderen zuweisen, z. B.

```
Dim A As string, B As string
A = B
```

dann kopiert Visual Basic lediglich den Wert der einen Variablen in die andere. Visual Basic liest den Inhalt von Variable `B` und schreibt eine Kopie dieser Daten in Variable `A`.

Wenn Sie jedoch Objekte zuweisen, wie etwa

```
Dim A As MeineKlasse, B As New MeineKlasse
Set A = B
```

dann macht Visual Basic im Hintergrund folgendes:

- Variable `B` enthält einen Zeiger auf das Standard-Interface von `MeineKlasse` auf ein neu angelegtes Objekt.
- Visual Basic erfragt über `QueryInterface` des Interface einen weiteren Zeiger auf das Standard-Interface von `Meine Klasse`. Dies erhöht zugleich den Referenzzähler des Objekts.
- Der Zeiger wird in Variable `A` geladen.

Was geschieht im Hintergrund, wenn folgender Code ausgeführt wird?

```
Dim A As Object, B As New MeineKlasse
Set A = B
```

- Variable `B` enthält einen Zeiger auf das Standard-Interface von `MeineKlasse` auf ein neu angelegtes Objekt.
- Visual Basic erfragt über `QueryInterface` einen weiteren Zeiger auf das `IDispatch`-Interface des Objekts. Dies erhöht zugleich den Referenzzähler des Objekts.
- Der Zeiger wird in Variable `A` geladen.

Und was ist, wenn die Objekt-Typen nicht übereinstimmen, wie im folgenden Beispiel?

```
Dim A As AndereKlasse, B As New MeineKlasse
Set A = B
```

- Variable `B` enthält einen Zeiger auf das Standard-Interface von `MeineKlasse` auf ein neu angelegtes Objekt.
- Visual Basic erfragt über `QueryInterface` von `MeineKlasse` einen weiteren Zeiger auf das `AndereKlasse`-Interface des Objekts. Dieser Versuch schlägt jedoch fehl, da `MeineKlasse` nicht über das `AndereKlasse`-Interface verfügt.
- Visual Basic meldet als Fehler, daß die Datentypen nicht übereinstimmen.

Über dieses Thema werden Sie in Kapitel 13 mehr erfahren.

4.3.1 Auswirkungen der Bindung auf die Performance

Welchen Unterschied macht die Bindungsweise beim Zugriff auf die Methoden und Eigenschaften eines Objekts aus? Das Beispielprogramm zur Bindung im Ordner zu Kapitel 4 auf der Buch-CD demonstriert dies. Das Hauptformular enthält eine `ListBox` und eine Schaltfläche. Listing 4.1 enthält den Code des Programms. Das Projekt enthält eine einzelne Klasse mit zwei Methoden. Eine der Methoden gibt einfach einen Integer-Wert zurück – eine sehr schnelle Operation. Die andere Methode führt eine größere Reihe von String-Operationen aus, um eine etwas komplexere Operation zu simulieren.

```
' ActiveX: A Guide to the Perplexed Beispiel-Programm Bindung
' Beispiel zur Bindung
' Copyright (c) 1997 by Desaware Inc.

Option Explicit

' Eine sehr schnelle Operation
Public Function FastOperation() As Integer
   FastOperation = 1
End Function

' Eine langsamere Operation
Public Function SlowOperation() As Integer
   Dim x&
   Dim s$
   For x = 1 To 50
      s$ = s$ & "X"
   Next x
End Function
```

Listing 4.1: Das Beispielprogramm zur Bindung

Das Formular legt eine einzelne Instanz von `Class1` an. Den zwei Objekt-Variablen wird während des `Form_Load`-Ereignisses die Referenz auf dieses Objekt zugewiesen. Die Variable `EarlyBound` ist als `Class1` deklariert. Da Visual Basic die Klasse `Class1` bereits beim Kompilieren kennt, kann das direkte Interface des Objekts verwendet werden, also die frühe Bindung. Die Variable `LateBound` ist dagegen als `As Object` deklariert. Beim Kompilieren weiß Visual Basic daher noch nicht, welcher Objekttyp von dieser Variablen referenziert werden soll, da zur Laufzeit jeder beliebige Objekttyp zugewiesen werden könnte. Das bedeutet, daß Visual Basic bei allen Methoden- und Eigenschaftsaufrufen auf das Automations-Interface zugreifen muß – also späte Bindung.

Die Meßtechnik ist schnell erklärt. Die aktuelle Zeit wird festgehalten und sowohl die `FastOperation`- als auch die `SlowOperation`-Methode des Objekts werden aufgerufen, jeweils über die Variablen `EarlyBound` und `LateBound`. Die Aufrufe erfolgen mehrfach hintereinander, damit die durchschnittlichen Bearbeitungszeiten je Aufruf ermittelt werden können – die Auflösung des System-Zeitgebers ist ja nicht besonders fein. Trotzdem lassen sich die Aufrufe der schnellen Operation bei der Anzahl der Wiederholungen in diesem Beispiel kaum exakt messen. Variieren Sie gegebenenfalls die Zahl der Wiederholung, um sie den Gegebenheiten Ihres Systems anzupassen.

Bei einem dritten Test wird die Geschwindigkeit der `CallByName`-Anweisung gemessen. Die Funktion ist per Definition spät gebunden, da der Aufruf der im Text-String angegebenen Methode über Automation stattfindet.

```
     ' ActiveX: A Guide to the Perplexed
     ' Beispiel zur Bindung
     ' Copyright (c) 1997 by Desaware Inc.

     Option Explicit

Private Declare Function GetTickCount& Lib "kernel32" ()

     ' Zeit merken
     Dim CurrentTime As Long

     Dim EarlyBound As Class1
     Dim LateBound As Object

     Const repeats = 50000

     ' Ein konkretes Objekt
     Dim TheObject As New Class1

Private Sub cmdTest_Click()
     Dim ctr&
     Dim res&
     Dim EarlyFast As Long
```

```
Dim EarlySlow As Long
Dim ByNameFast As Long
Dim ByNameSlow As Long
Dim LateFast As Long
Dim LateSlow As Long

Screen.MousePointer = vbHourglass

lblStatus.Caption = "Early-Fast"
lblStatus.Refresh

CurrentTime = GetTickCount()
For ctr = 1 To repeats
   res = EarlyBound.FastOperation
Next ctr
EarlyFast = (GetTickCount() - CurrentTime)

lblStatus.Caption = "Early-Slow"
lblStatus.Refresh

' Nun die langsame Operation
CurrentTime = GetTickCount()
For ctr = 1 To repeats
   res = EarlyBound.SlowOperation
Next ctr
EarlySlow = (GetTickCount() - CurrentTime)

lblStatus.Caption = "Late-Fast"
lblStatus.Refresh

' Späte Bindung - schnell
CurrentTime = GetTickCount()
For ctr = 1 To repeats
   res = LateBound.FastOperation
Next ctr
LateFast = (GetTickCount() - CurrentTime)

lblStatus.Caption = "Late-Slow"
lblStatus.Refresh

' Nun die langsame Operation
CurrentTime = GetTickCount()
For ctr = 1 To repeats
   res = LateBound.SlowOperation
Next ctr
LateSlow = (GetTickCount() - CurrentTime)
```

```
    lblStatus.Caption = "ByName-Fast"
    lblStatus.Refresh

    ' CallByName - schnell
    CurrentTime = GetTickCount()
    For ctr = 1 To repeats
       res = CallByName(LateBound, _
        "FastOperation",  VbMethod)
    Next ctr
    ByNameFast = (GetTickCount() - CurrentTime)

    lblStatus.Caption = "ByName-Slow"
    lblStatus.Refresh

    ' Nun die langsame Operation
    CurrentTime = GetTickCount()
    For ctr = 1 To repeats
       res = CallByName(LateBound, "SlowOperation", _
        VbMethod)
    Next ctr
    ByNameSlow = (GetTickCount() - CurrentTime)

    Screen.MousePointer = vbNormal
    lblStatus.Caption = ""

    lstResults.AddItem "Early Binding"
    lstResults.AddItem "  Fast: " & GetTime(EarlyFast)
    lstResults.AddItem "  Slow: " & GetTime(EarlySlow)
    lstResults.AddItem "Late Binding"
    lstResults.AddItem "  Fast: " & GetTime(LateFast)
    lstResults.AddItem "  Slow: " & GetTime(LateSlow)
    lstResults.AddItem "CallByName"
    lstResults.AddItem "  Fast: " & GetTime(ByNameFast)
    lstResults.AddItem "  Slow: " & GetTime(ByNameSlow)
    lstResults.AddItem "Binding overhead:"
    lstResults.AddItem "  Fast: " & Format$((LateFast - _
     EarlyFast) / LateFast, "Percent")
    lstResults.AddItem "  Slow: " & Format$((LateSlow - _
     EarlySlow) / LateSlow, "Percent")
    lstResults.AddItem "  Byname Fast: " & Format$( _
     (ByNameFast - EarlyFast) / ByNameFast, "Percent")
    lstResults.AddItem "  Byname Slow: " & Format$( _
     (ByNameSlow - EarlySlow) / ByNameSlow, "Percent")

End Sub
```

```
Private Sub Form_Load()
     Set EarlyBound = TheObject
     Set LateBound = TheObject
End Sub

' Zeit in Millisekunden in einem formatierten String
Public Function GetTime(timeval As Long) As String
     ' timeval ist die Zeitdifferenz in Millisekunden
     GetTime = Format$(CDbl(timeval) / repeats * 1000#, _
      "0.###")
     End Function
```

Listing 4.2: Die CallByName-Funktion

Sie sollten einmal die kompilierte Version dieses Beispiels starten – es ist in Native-Code kompiliert und daher sehr schnell. Sie können es auch in der Visual-Basic-Umgebung starten, aber dann müssen Sie sich auf Wartezeiten gefaßt machen. Abbildung 4.3 zeigt das Programm in Aktion mit einigen typischen Ergebnissen. Wie Sie sehen, bewirkt der Bindungs-Overhead bei der schnellen Funktion eine spürbare Verzögerung – annähernd 100% der Zeit wird allein für die Operation der späten Bindung aufgewendet. Bei der langsamen Funktion fällt der Bindungs-Overhead dementsprechend weniger ins Gewicht. Die Aufrufe der `CallByName`-Anweisung sind noch langsamer als die Aufrufe mit später Bindung. Das liegt an dem obendrein dazukommenden Aufwand für den Aufruf der Anweisung selbst und das Vorbereiten der Parameter für den internen Aufruf der `Invoke`-Methode des Objekts.

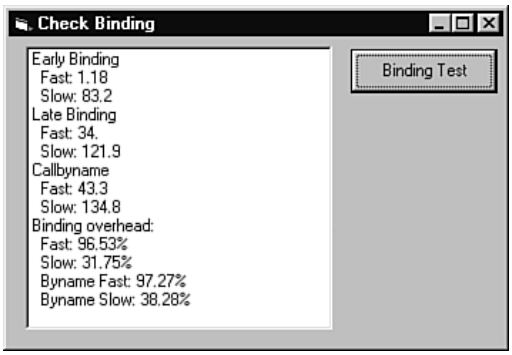


Abb. 4.3: Das Bindungsbeispiel in Aktion

Sie werden vermuten, daß die langsamere Funktion eher repräsentativ für Zugriffe auf Objektmethoden sein könnte. Dies ist aber weniger der Fall. Nicht nur, daß viele Operationen hinter Objektmethoden kurz sind – vor allem Eigenschaften-Zugriffe neigen mehr dazu, einfach und schnell zu sein. Daher kann die frühe Bindung in vielen Fällen erhebliche Vorteile bringen.

Warum verwenden wir dann nicht immer die frühe Bindung? Nun, in manchen Situationen werden Sie vielleicht nicht umhin kommen, eine einzige Variable für mehrere Objekttypen vorzusehen. Oder Sie bekommen ein Objekt von einem anderen Server, wobei die Typ-Definitionen nicht vorab erhältlich sind oder ein Objekt die offengelegten Eigenschaften und Methoden ändert. Manche Objekte, die mit anderen Entwicklungswerkzeugen als Visual Basic erstellt worden sind, bieten zwar Typ-Bibliotheken, jedoch keine dualen Interfaces an. In diesen Fällen kann Visual Basic eine eingeschränkte Form der frühen Bindung vornehmen, die *DispID-Bindung* genannt wird. Hierbei werden die Dispatch-IDs der Funktionen und Parameter zur Design-Zeit vorberechnet. Zur Laufzeit werden dann diese Werte für die Methoden-Aufrufe über die `Invoke`-Methode des `IDispatch`-Interface verwendet.

Ein letzter Kommentar zum `IDispatch`-Interface: Ein Objekt kann mehr als nur ein solches Interface haben. Dies war schon immer so unter COM, und nun trifft dies auch unter Visual Basic zu – eine tatsächlich sehr interessante Aussicht. Sie können bei einem Objekt mehrere Interfaces implementieren, über die `Implements`-Anweisung. In Kapitel 5, »Aggregation und Polymorphie«, werden wir in das Thema der frühen und späten Bindung und der Verwendung mehrerer Interfaces bei mit Visual Basic erstellten COM-Objekten tiefer eintauchen.