Kapitel 5

Aggregation und Polymorphie

Haben Sie wirklich die Einführung gelesen? Wahrscheinlich werden Sie sagen, daß es doch wohl ein wenig spät sei, diese Frage noch in Kapitel 5 aufzuwerfen. Doch wenn Sie die Einführung nicht gelesen haben sollten, dann werden Sie dieses Buch vermutlich für eines der seltsamsten technischen Bücher überhaupt halten: Wo sind denn bloß die Schritt-für-Schritt-Anleitungen? Wo ist die Einführung in die Verwendung von Klassen-Modulen, und wie legt man Eigenschaften an?

Als ich mich schon frühzeitig während der Entstehung dieses Buches dazu entschlossen hatte, nicht einfach nur die Visual-Basic-Dokumentation nachzuschreiben, war mir noch nicht bewußt, welchen Luxus dies bedeuten würde. Ich spare ja dabei nicht nur die Zeit und den Aufwand, die Dokumentation in eigenen Worten nachzuerzählen. Der Luxus besteht vielmehr darin, daß ich mich einzig und allein auf das wirklich wichtige konzentrieren kann. Es ist die Chance, die Oberfläche der reinen Syntax der Sprache zu verlassen und tiefer in die Materie der Funktionsweise von Visual-Basic-Programmen hinabzutauchen – darüber zu schreiben, wie Sie sich die Vorteile von Visual Basic zunutze machen können, um effizientere und elegantere Software zu schreiben.

Somit kann ich nun ein ganzes Kapitel für die Themen Aggregation und Polymorphie »verschwenden«. Diese Themen sind in der Visual-Basic-Dokumentation ziemlich gut versteckt untergebracht. Sie sollten sie aber dennoch nicht übersehen – diese Features können und werden die Art und Weise ändern, wie Sie in Visual Basic zu programmieren pflegen. Verzeihen Sie Microsoft, daß man diesen Themen nicht mehr Raum gewidmet hat – schließlich hatte man doch Unmengen an Stoff zu verarbeiten, und man konnte ja auch nicht auf andere Sprach-Dokumentationen zurückgreifen.

5.1 Noch einmal Bindungen

Im vorangegangenen Kapitel haben Sie gesehen, daß Visual-Basic-Objekte tatsächlich COM-Objekte sind. Sie haben gelernt, daß COM-Objekte mehrere Interfaces haben können. Die Methoden und Eigenschaften eines COM-Objekts werden von diesem als duales Interface offengelegt. Ein Interface kann direkt aufgerufen werden. Das andere ist ein Dispatch-Interface, das in seiner Dispatch-Tabelle eine Liste der Methoden und Eigenschaften des Objekts enthält.

Die Beispiele in Kapitel 3, »Objekte und Visual Basic«, verwenden die frühe Bindung. Alle Objektreferenzen wurden mit einer direkten Referenz des Klassen-Typs referenziert, so daß Aufrufe der Methoden und Zugriffe auf die Eigenschaften des Objekts direkt über dessen Klassen-Interface gingen. In Kapitel 4, »Das Component Object Model: Interfaces, Automation und Bindung«, sahen Sie, daß dieses Prinzip der Bindung deutlich schneller als das der späten Bindung ist. Ich habe kurz umrissen, unter welchen Umständen eine späte Bindung sinnvoll sein kann: wenn die Interface-Information nicht im voraus verfügbar ist, wenn ein Objekt seine offengelegten Methoden und Eigenschaften dynamisch ändern könnte, und wenn Sie einer einzelnen Variablen mehrere verschiedene Daten- bzw. Objekttypen zuweisen wollen.

Die beiden ersten Situationen sind offensichtlich. Wenn Sie zur Design-Zeit nicht über die Interface-Informationen verfügen, muß das `IDispatch`-Interface verwendet werden, über das die Methoden und Eigenschaften eines Objekts zur Laufzeit ermittelt werden können. Denken Sie daran, daß Visual-Basic-Objekte zwar über duale Interfaces verfügen, jedoch COM-Objekte von jeder beliebigen Anwendung instanziert werden können, ohne daß duale Interfaces hierfür eine Voraussetzung darstellen würden. Viele Objekte, die Sie in Visual Basic verwenden können, verfügen nur über das `IDispatch`-Interface. Diese Objekte können daher nur spät gebunden werden. Ein Objekt, das nur `IDispatch` anbietet, besitzt die Flexibilität, seine Methoden und Eigenschaften zur Laufzeit ändern zu können. Dies ist jedoch nicht gerade üblich – und Ihre Visual-Basic-Objekte können dies auch gar nicht.

Die dritte Situation ist verzwickter. Wann und warum sollten Sie einer einzelnen Variablen verschiedene Daten- bzw. Objekttypen zuweisen wollen?

Nehmen wir einmal eine einfache Anwendung, die Darlehen verwalten soll. Das Projekt Loan1 (siehe Abbildung 5.1) enthält zwei Listboxen. Die obere Listbox listet die vorhandenen Darlehen auf und die untere Listbox stellt detailliertere Informationen zu einem in der oberen Listbox ausgewählten Darlehen dar.

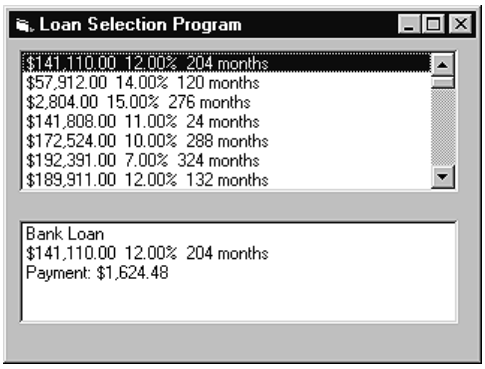

Abb. 5.1: Das Hauptformular der Anwendung Loan1

Jedes Darlehen hat eine Laufzeit, einen Darlehensbetrag und einen Zinsfuß. Listing 5.1 zeigt die Klasse `clsBankLoan`. Neben den Eigenschaften `AmountAvailable`, `Duration` und `Interest` verfügt diese Klasse über die Funktion `Payment`, die die monatlichen Raten zurückgibt, die Funktion `Summary`, die eine kurze Beschreibung des Darlehens liefert, und die Funktion `SourceType`, die einen String liefert, der den Ursprungstyp des Darlehens beschreibt.

```
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.  All Rights Reserved
' Kapitel 5
```

```
Option Explicit

' Darlehensbetrag
Public AmountAvailable As Currency

' Laufzeit
Public Duration As Integer

' Zinsfuß
Public Interest As Double

' Ratenhöhe berechnen
Public Function Payment() As Currency
   Dim factor As Double
   Dim iper As Double
   iper = Interest / 12
   factor = iper * ((1 + iper) ^ Duration)
   Payment = AmountAvailable * factor / (((1 + iper) ^ Duration) - 1)
End Function

' String-Beschreibung für Darlehen
Public Function Summary() As String
   Summary = Format$(AmountAvailable, "Currency") & _
    "  " & Format$(Interest,_
   "Percent") & "  " & Duration & " months"
End Function

Public Function SourceType() As String
   SourceType = "Bank Loan"
End Function
```

Listing 5.1: Die Klasse clsBankLoan

Listing 5.2 zeigt das Hauptformular der Beispielanwendung. In der Ereignisprozedur `Form_Load` wird ein Array von `clsBankLoan`-Objekten gefüllt. In einer realen Anwendung könnten Sie das Array aus einer Datenbank oder über einen Online-Dienst füllen. Die Beschreibungen der Objekte (`Summary`) werden in die obere Listbox `lstLoans` geladen. Wenn Sie einen Eintrag in der oberen Liste anklicken, werden die Informationen über das entsprechende Darlehen in der (unteren) Listbox `lstInfo` dargestellt.

```
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.  All Rights Reserved
' Kapitel 5

    Option Explicit

    ' Array der verfügbaren Darlehen
```

```
Dim Loans() As clsBankLoan

' Konstante zu Testzwecken
Const LOANCOUNT = 100

Private Sub Form_Load()
   Dim loannum%

   ' Liste der verfügbaren Darlehen laden
   ' In einer realen Anwendung könnten diese
   ' Informationen aus einer Datenbank oder über
   ' einen Online-Dienst bezogen werden.
   ' In diesem Beispiel erzeugen wir sie zufällig
   ReDim Loans(LOANCOUNT)
   For loannum = 1 To LOANCOUNT
      Set Loans(loannum) = New clsBankLoan
      With Loans(loannum)
         .AmountAvailable = CLng(Rnd() * 200000)
         .Duration = 12 * Int((Rnd * 30) + 1)
         .Interest = (7 + Int((Rnd * 80) * 0.125)) / 100
         lstLoans.AddItem .Summary
      End With
   Next loannum
End Sub

Private Sub lstLoans_Click()
   Dim loannum%
   loannum = lstLoans.ListIndex + 1
   lstInfo.Clear
   With Loans(loannum)
      lstInfo.AddItem .SourceType
      lstInfo.AddItem .Summary
      lstInfo.AddItem "Payment: " & _
       Format$(.Payment, "Currency")
   End With
End Sub
```

Listing 5.2: Listing für Formular LnSel1.frm

Soweit ist das eine klare und einfache Anwendung. Und wenn Sie ein Kreditvermittler sein sollten, könnten Sie damit durchaus zufrieden sein. Doch in der heutigen schnellebigen Zeit können Sie nie sicher sein, daß die Regierung nicht die Bedingungen für das Kreditgewerbe ändert und Wertpapier Händler zuläßt. Denn dann muß die Software an die neuen Darlehensarten angepaßt werden. Was passiert beispielsweise, wenn ein Effekten-Kredit Darlehen auch die Sicherheitsrahmen des Darlehens nachhalten soll?

5.2 Ein Objekt als Objekt

Listing 5.3 enthält Teile einer neuen Klasse, die der Loan-Anwendung hinzugefügt wurde (siehe Beispiel-Programm Loan2.vbp auf der Buch-CD), um Effekten-Kredit einzubringen. Ansonsten ist die Klasse identisch mit der Klasse `clsBankLoan`. Sie werden die Duplizität bemerken, die darin besteht, daß die `Payment`-Funktion in der Klasse `clsSecurityLoan` eine exakte Kopie derjenigen der Klasse `clsBankLoan` ist. Das ist ein wenig verschwenderisch – wir werden uns in Kürze einen Weg anschauen, diesen Overhead zu reduzieren.

```
' ***Margin requirement
Public Margin As Double

' Beschreibungs-String des Darlehens abrufen
Public Function Summary() As String
   Summary = Format$(AmountAvailable, "Currency") & _
    "  " & Format$(Interest,_
   "Percent") & "  " & Duration & " months. Margin: " _
    & Format$(Margin, "Percent")
End Function

Public Function SourceType() As String
   SourceType = "Brokerage Loan"
End Function
```

Listing 5.3: Modifikationen in der Klasse clsSecurityLoan

Das Array `Loans()` im Formular `frmLoan1` ist für den Objekt-Typ `clsBankLoan` deklariert gewesen. Nun soll das Array auch Objekte vom Typ `clsSecurityLoan` aufnehmen können. Jedoch wird jeder Versuch, dem `Loans`-Array ein `clsSecurityLoan`-Objekt zuzuweisen, in einem Typ-Fehler resultieren. Eine mögliche Lösung für dieses Problem wäre, das `Loans`-Array als `As Object` zu deklarieren (siehe Listing 5.4).

```
' Array verfügbarer Darlehen
Dim Loans() As Object

Private Sub Form_Load()
   Dim loannum%

   ' Liste der verfügbaren Darlehen laden
   ReDim Loans(LOANCOUNT)
   For loannum = 1 To LOANCOUNT
      Select Case Int(Rnd() * 2)
         Case 0
            Set Loans(loannum) = New clsBankLoan
         Case 1
            Set Loans(loannum) = New clsSecurityLoan
```

```
            ' Margin trifft nur für diesen Objekt-Typ zu
            Loans(loannum).Margin = Rnd()
        End Select
        With Loans(loannum)
           .AmountAvailable = CLng(Rnd() * 200000)
           .Duration = 12 * Int((Rnd * 30) + 1)
           .Interest = (7 + Int((Rnd * 80) * 0.125)) / 100
           lstLoans.AddItem .Summary
        End With
     Next loannum
End Sub
```

Listing 5.4: Modifikationen im Code des Formulars frmLoan

Ihnen fällt sicher auf, daß die einzige Änderung für die Implementierung des neuen Objekttyps darin besteht, die Deklaration des Objekttyps des `Loans`-Array zu ändern und den korrekten Objekttyp in das Array zu laden. Die Eigenschaft `Margin` wird nur bei einem Objekt der Klasse `clsSecurityLoan` gesetzt.

Überlegen Sie einmal kurz, was beim Aufruf der `Summary`-Funktion eines Objekts im `Loans`-Array vor sich geht. Visual Basic ruft korrekt die entsprechende Funktion des gerade referenzierten Objekts auf. Dies ist eine Demonstration der Polymorphie: Der gleiche Funktionsname kann bei zwei verschiedenen Objekten verwendet werden. Wie Sie hier sehen, geht es um mehr, als nur um die Vereinfachung für Programmierer, die Zahl der zu merkenden Funktionsnamen zu reduzieren. Polymorphie erlaubt die Verwendung des gleichen Codes zur Referenzierung verschiedener Objekttypen beim Aufruf einer Methode oder Eigenschaft gleichen Namens.

5.3 Aggregation

Ihr Darlehensgeschäft ist ins Rollen gekommen, und Sie haben viele neue Kunden gewonnen, von denen einige einer anderen Klientel angehören, die Sie nicht vorausgesehen haben. Nun müssen Sie Ihr Programm erneut anpassen, damit es mit Darlehen für Klienten umgehen kann, die, sagen wir mal, etwas weniger kreditwürdig sind. Diese Darlehen erfordern nicht nur, daß Sie einen neuen Objekttyp hinzufügen, sondern auch, daß dieser neue Objekttyp über eine Methode verfügt, die eine Verzugsstrafe anhand der Darlehenshöhe berechnet.

Sie werden zunächst vermuten, daß dieses Objekt wiederum eine Kopie der `Payment`-Funktion enthalten wird. Doch das wäre nicht nur überflüssig und Verschwendung, sondern öffnet auch Tür und Tor für Wartungsprobleme – etwa wenn Sie eines Tages einen Bug im Payment-Code entdecken sollten. (Ich denke zwar, daß das in diesem kleinen Beispiel weniger der Fall sein wird, jedoch in einer umfangreichen realen Anwendung mit Dutzenden an komplexeren Klassen-Methoden durchaus vorkommen könnte.) Sie könnten ein Standard-Modul hinzufügen und eine globale Funktion zur Berechnung der Tilgungsbeträge anlegen.

Doch dies wäre eine Abweichung vom Weg der Kapselung, die ja die objektorientierte Programmierung ermöglicht. Das Problem wird jedoch erst recht wieder auftauchen, falls Sie sich dazu entschließen sollten, jede dieser Klassen als eigenständige ActiveX-Code-Bibliothek in einer eigenen DLL anzulegen. Sicher gibt es da eine Lösung, die der objektorientierten Methodologie entspringt ...

In der Tat gibt es eine solche Lösung. Die Antwort lautet: Aggregation (siehe Listing 5.5). Die neue Klasse `clsLoanShark` basiert auf der Klasse `clsBankLoan`, die als Referenztyp dient. Doch anstatt die Methoden und Eigenschaften der `clsBankLoan`-Klasse zu kopieren, wird eine private Instanz der Klasse `clsBankLoan` immer dann angelegt, wenn ein `clsLoanShark`-Objekt angelegt wird. Mit anderen Worten: Ein `clsLoanShark`-Objekt ist ein Aggregat aus neuem Code und einem Objekt des Typs `clsBankLoan` (daher der Begriff *Aggregation*). Dabei werden die meisten Methoden und Eigenschaften nicht direkt in der `clsLoanShark`-Klasse bearbeitet, sondern an das interne `clsBankLoan`-Objekt delegiert. So wird hier nicht der Code der `Payment`-Funktion eingefügt, sondern die `Payment`-Funktion des `LoanTemplate`-Objekts aufgerufen. `LoadTemplate` ist dabei der Variablenname des privaten `clsBankLoan`-Objekts.

Beachten Sie, daß `LoanTemplate` mit der `New`-Option deklariert ist, weil hier auf jeden Fall eine Instanz des Objekts benötigt wird. Ohne die `New`-Option würde die `LoanTemplate`-Variable leer (auf `Nothing` gesetzt) bleiben, und es ist offensichtlich, daß Sie nicht auf die Methoden und Eigenschaften eines Objekts zugreifen können, das nicht existiert.

```
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.  All Rights Reserved
' Kapitel 5

    Option Explicit

    ' Internes Klassen-Objekt in Aggregation verwendet
    Private LoanTemplate As New clsBankLoan

    Public Property Get AmountAvailable() As Currency
       AmountAvailable = LoanTemplate.AmountAvailable
    End Property

    Public Property Let AmountAvailable(ByVal vNewValue _
     As Currency)
       LoanTemplate.AmountAvailable = vNewValue
    End Property

    Public Property Get Duration() As Integer
       Duration = LoanTemplate.Duration
    End Property

    Public Property Let Duration(ByVal vNewValue As Integer)
```

```
   LoanTemplate.Duration = vNewValue
End Property

Public Property Get Interest() As Double
   Interest = LoanTemplate.Interest
End Property

Public Property Let Interest(ByVal vNewValue As Double)
   If vNewValue < 0.5 Then vNewValue = vNewValue + 0.5
   LoanTemplate.Interest = vNewValue
End Property

Public Function Payment() As Currency
   Payment = LoanTemplate.Payment
End Function

Public Function Summary() As String
   Summary = LoanTemplate.Summary
End Function

Public Function SourceType() As String
   SourceType = "Loan Shark"
End Function

Public Function LatePenalty() As String
   Select Case AmountAvailable
      Case 0 To 25000
         LatePenalty = "Broken Fingers"
      Case 25000 To 75000
         LatePenalty = "Broken arm"
      Case Else
         LatePenalty = "You don't want to know"
   End Select
End Function
```

Listing 5.5: Das Klassen-Objekt clsLoanShark

Sie werden sich vielleicht wundern, warum die Eigenschaften `AmountAvailable`, `Duration` und `Interest` an das `Loan-Template`-Objekt delegiert werden. Wäre es nicht einfacher, sie lediglich als öffentliche Variablen zu deklarieren? Würde das nicht auch die Performance erhöhen, wenn der zusätzliche Funktionsaufruf entfiele?

Es wäre einfacher, würde aber nicht funktionieren. Vergessen Sie nicht, daß Sie ja die `Payment`-Funktion delegieren, und daß diese Funktion alle drei Eigenschaften benötigt, um die monatlichen Raten zu berechnen. Diese Funktion verwendet die

Werte, die sie im LoanTeamplate-Objekt findet, so daß Sie diese Eigenschaften ebenfalls an das interne Objekt delegieren müssen, damit die Payment-Funktion mit den korrekten Werten arbeiten kann.

Die Delegation muß auf jeden Fall exakt sein. Sie können Ihre eigene Fehlerprüfung in die Eigenschaftsfunktionen einfügen, wie Sie es bei der Property Let-Funktion der Interest-Eigenschaft sehen können. Kein ernstzunehmender Kredithai würde einen Zinsfuß unter 50% akzeptieren – dies spiegelt sich in der Bearbeitung dieser Eigenschaft wider.

Das Objekt clsLoanShark braucht nicht alle Methoden und Eigenschaften an das LoanTemplate-Objekt zu delegieren. Die Methode SourceType beispielsweise wird direkt bearbeitet. Das ist sinnvoll, da die Methode des LoanTemplate-Objekts »Bank Loan« liefern würde, was ja falsch wäre. Die Möglichkeit, Methoden oder Eigenschaften des Delegations-Objekts zu übergehen (»überschreiben«) ist einer der Gründe für das Anlegen von Klassen, die auf anderen Klassen basieren. Ein weiterer Grund ist die Möglichkeit, neue Methoden oder Eigenschaften hinzuzufügen, wie es bei der Methode LatePenalty deutlich wird, die ja in der Klasse clsBankLoan nicht vorhanden ist.

Aggregation erlaubt das einfache Wiederverwenden von Code eines eingeschlossenen Objekts. Wie steht es aber mit eventuellen Nachteilen dieser Technik? Es gibt nur einen ins Auge fallenden Nachteil: Jedesmal, wenn ein clsLoanShark-Objekt instanziert wird, wird zugleich ein clsBankLoan-Objekt instanziert. Damit ist ein zusätzlicher Overhead verbunden, und der Einfluß auf die Performance ist nicht vorhersehbar.

Der Overhead beim Anlegen eines eingeschlossenen Objekts ist vernachlässigbar, wenn sich das Objekt in derselben Anwendung befindet. Befindet es sich jedoch in einem ActiveX-Server, kann der Einfluß auf das System substantiell werden. Auf diesen Effekt gehen wir im nächsten Kapitel näher ein. Ein weiterer Punkt kommt ins Spiel, wenn das eingeschlossene Objekt Code in den Initialisierungs- und Terminierungsereignissen enthält. Dieser Code wird immer dann ausgeführt, wenn ein Objekt instanziert bzw. zerstört wird.

Sehen wir uns nun an, wie das Hauptprogramm mit diesem neuen Objekt umgeht. In Abbildung 5.2 sehen Sie das Programm in Aktion. Da Sie einem guten objektorientierten Design folgen, ist es kaum verwunderlich, daß die Änderungen für die Verwendung der neuen Klasse minimal sind. Der Code im Form_Load-Ereignis wird so modifiziert, daß er den neuen Objekttyp laden kann. Auch hier wird natürlich eine reale Anwendung nur simuliert, bei der die Darlehensinformationen aus einer Datenbank oder über einen Online-Dienst geladen würden.

```
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.  All Rights Reserved
' Kapitel 5

    Option Explicit
```

```
' Array verfügbarer Darlehen
Dim Loans() As Object

' Konstante zu Testzwecken
Const LOANCOUNT = 100

Private Sub Form_Load()
   Dim loannum%

   ' Liste verfügbarer Darlehen laden
   ReDim Loans(LOANCOUNT)
   For loannum = 1 To LOANCOUNT
      Select Case Int(Rnd() * 3)
         Case 0
            Set Loans(loannum) = New clsBankLoan
         Case 1
            Set Loans(loannum) = New clsSecurityLoan
            Loans(loannum).Margin = Rnd()
         Case Else
            Set Loans(loannum) = New clsLoanShark
      End Select
      With Loans(loannum)
         .AmountAvailable = CLng(Rnd() * 200000)
         .Duration = 12 * Int((Rnd * 30) + 1)
         .Interest = (7 + Int((Rnd * 80) * 0.125)) / 100
         lstLoans.AddItem .Summary
      End With
   Next loannum
End Sub

Private Sub lstLoans_Click()
   Dim loannum%
   Dim LatePenaltyValue$
   loannum = lstLoans.ListIndex + 1
   lstInfo.Clear
   With Loans(loannum)
      lstInfo.AddItem .SourceType
      lstInfo.AddItem .Summary
      lstInfo.AddItem "Payment: " & _
       Format$(.Payment, "Currency")
      LatePenaltyValue = GetLatePenalty(Loans(loannum))
      If LatePenaltyValue <> "" Then
         lstInfo.AddItem "Late payment penalty:"
         lstInfo.AddItem "   " & .LatePenalty
      End If
   End With
```

```
End Sub

' Generische Funktion zur Ermittlung
' des LatePenalty-Wertes
Public Function GetLatePenalty(obj As Object) As String
   On Error GoTo nofunction
   GetLatePenalty = obj.LatePenalty
   Exit Function
nofunction:
End Function
```

Listing 5.6: Listing für LnSel3.frm

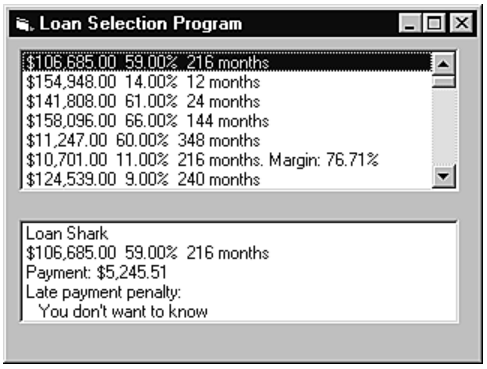

Abb. 5.2: Das Hauptformular der Loan3-Anwendung

Die wichtigste und interessante Änderung ist hier wahrscheinlich die Bearbeitung der `LatePenalty`-Methode. Jeder Versuch, diese Methode bei einem `clsBankLoan`- oder einem `clsSecurityLoan`-Objekt aufzurufen, wird in einem Fehler resultieren, da diese Objekte die Methode nicht unterstützen. Somit können Sie die neue Methode auf zweierlei Weise behandeln:

- Prüfung, ob das Objekt vom Typ `clsLoanShark` ist. Wenn ja, Aufruf der `LatePenalty`-Methode. Der Typ eines Objekts kann mit der `If Typeof... Is`-Anweisung oder mit der `TypeName`-Funktion getestet werden.
- Versuch, die Methode `LatePenalty` bei jedem Objekt aufzurufen, wobei auftretende Fehler ignoriert werden.

In diesem Beispiel machen wir es auf die zweite Weise. Die Funktion `GetLatePenalty` bearbeitet dies und liefert einen leeren String zurück, wenn der Versuch des Aufrufs fehlgeschlagen ist. Das Schöne an diesem Weg ist, daß er allgemeingültig ist. Er wird auch mit jedem weiteren zukünftigen Objekt funktionieren, das die `LatePenalty`-Funktion enthält, ohne daß weitere Änderungen am Code des Hauptformulars notwendig würden.

Ein dritter Ansatz wäre, auf einem direkten Weg das Vorhandensein eines Methodennamens zu prüfen. Die Hilfs-DLL `Apigid32.DLL`, die auf der Buch-CD mitgeliefert wird, enthält die Funktion `agIsValidname()`, mit deren Hilfe Sie feststellen können, ob der Name einer Methode oder einer Eigenschaft von einem Objekt unterstützt wird. Diese Funktion greift intern auf das `IDispatch`-Interface zurück und verwendet deren Funktionen `GetIdOfNames`, um festzustellen, ob zu dem gefragten Namen eine gültige Dispatch-ID existiert.

5.4 Ein implementiertes Objekt

Sie genießen die Vorzüge des `Object`-Datentyps bei der Implementierung von Aggregation und Polymorphie. Leider ist dieser `Object`-Datentyp mit zwei Nachteilen behaftet:

- Da eine als `As Object` deklarierte Variable jeden Objekttyp referenzieren kann, öffnet dies die Tür zu neuen Fehlerquellen. Was passiert mit Ihrem Programm, wenn Sie so einer Variablen zufällig ein Formular oder ein Control zuweisen? Oder ein anderes Objekt, das kein Darlehen repräsentiert? Diese Art von Fehlern kann nicht zur Design-Zeit aufgedeckt werden, sondern nur zur Laufzeit, wenn das Programm versucht, auf eine Methode oder Eigenschaft zuzugreifen, über die das fälschlich zugewiesene Objekt gar nicht verfügt. Dies bedeutet einen erhöhten Testaufwand. Letztlich ist es jedoch immer besser, möglichst bereits den Compiler auf derartige Probleme hinweisen zu lassen.
- Alle Zugriffe auf diese Objekte erfolgen unter später Bindung, was meistens zu Lasten der Performance geht. Dies mag in einem kleinen Programm nicht weiter auffallen, kann aber kritisch werden in einem Programm, das Millionen an Transaktionen abwickelt.

Mit Visual Basic 5.0 wurde eine hervorragende Möglichkeit zur Lösung dieser Probleme eingeführt. Erinnern Sie sich an die Interface-Diskussion in Kapitel 4, »Das Component Object Model: Interfaces, Automation und Bindung«. Ein Objekt kann mehr als nur ein Interface offenlegen. Zum Beispiel legt unsere `clsBankLoan`-Klasse mindestens drei Interfaces offen: `IUnknown`, `IDispatch` (das Automation-Interface, das bei der späten Bindung verwendet wird) und das Interface `_clsBankLoan` (das früh gebundene duale Interface, das die Methoden und Eigenschaften des Objekts enthält. Einer Konvention entsprechend wird dessen Namen ein Unterstrich »_« vorangestellt, um zu verdeutlichen, daß es verborgen ist).

Diese Interfaces sehen Sie in Abbildung 5.3. Diese Abbildung illustriert außerdem das Standard-Schema, das Microsoft zur Beschreibung von Interfaces verwendet. Jeder Kreis repräsentiert ein Interface eines Objekts.

Im obenstehenden Beispiel war die als `As Object` deklarierte Objekt-Variable notwendig, da sie den einzigen Weg darstellte, über eine Variable auf mehr als nur ein Interface zugreifen zu können.

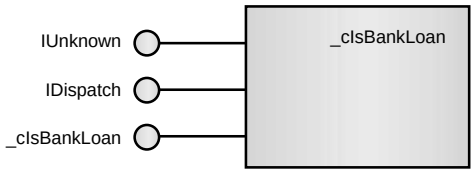

Abb. 5.3: Die Interfaces des clsBankLoan-Objekts

Es gibt aber noch einen weiteren Ansatz. Anstatt über eine Objekt-Variable auf viele verschiedene Interfaces zuzugreifen, bietet jedes Objekt ein allen gemeinsames Interface an.

Wenn das `clsLoanShark`-Objekt zusätzlich das `clsBankLoan`-Interface anbieten würde, könnten Sie weiterhin und durchgängig eine Objekt-Variable vom Typ `clsBankLoan` verwenden. Diesen Ansatz zeigt Abbildung 5.4.

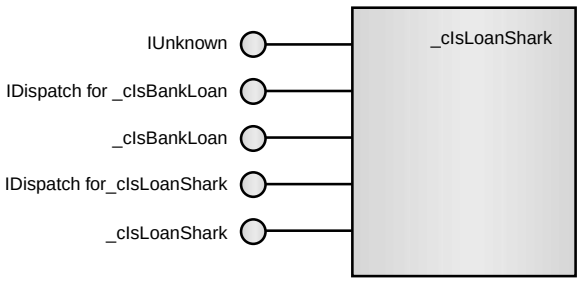

Abb. 5.4: Die Interfaces des clsLoanShark-Objekts

Visual Basic ermöglicht es einem Objekt über die `Implements`-Anweisung, mehrere Interfaces offenzulegen. Dazu wird am Anfang des Klassen-Codes folgende Anweisung eingefügt:

```
Implements AndererKlassenName
```

Wenn Sie beispielsweise eine Klasse namens `Klasse2` haben, die `Klasse1` implementiert, und diese `Klasse1` verfügt über die Methode `MeineMethode()`, muß auch `Klasse2` über Code für diese Methode der `Klasse1` verfügen. Die Deklaration erfolgt im Code von `Klasse2` so:

```
Private Sub Klasse1_MeineMethode()
```

Die Voranstellung von `Klasse1` weist darauf hin, daß die genannte Methode aus dem Interface der `Klasse1` stammt. `Klasse2` muß in dieser Form alle Methoden und Eigenschaften des Interface von `Klasse1` enthalten – nur so kann ein COM-Interface ordnungsgemäß funktionieren. Wie gesagt, stellt ein Interface einen Vertrag dar. Und wenn `Klasse2` das Interface der `Klasse1` implementiert, muß es vollständig implementiert werden.

Nun schauen wir uns einmal an, wie wir dieses Prinzip Schritt für Schritt in der Anwendung Loan4 umsetzen können.

Zunächst sehen Sie in Listing 5.7 das `clsbankLoan`-Objekt.

```
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.  All Rights Reserved
' Kapitel 5

    Option Explicit

    ' Darlehensbetrag
    Public AmountAvailable As Currency

    ' Laufzeit
    Public Duration As Integer

    ' Zinsfuß
    Public Interest As Double

    ' Raten berechnen
    Public Function Payment() As Currency
       Dim factor As Double
       Dim iper As Double
       iper = Interest / 12
       factor = iper * ((1 + iper) ^ Duration)
       Payment = AmountAvailable * factor / (((1 + iper) ^_
        Duration) - 1)
    End Function

    ' Beschreibungs-String
    Public Function Summary() As String
       Summary = Format$(AmountAvailable, "Currency") & _
        "  " & Format$(Interest,_
       "Percent") & "  " & Duration & " months"
    End Function

    Public Function SourceType() As String
       SourceType = "Bank Loan"
    End Function
```

Listing 5.7: Das clsBankLoan-Objekt, Version 2

Verschwenden Sie keine Zeit damit, nach Unterschieden zwischen diesem Code und Listing 5.1 zu suchen – es gibt keine.

Der größte Teil der Visual-Basic-Dokumentation zur `Implements`-Verwendung befaßt sich mit dem Anlegen einer abstrakten Klasse – einer Klasse, die lediglich

leere Rümpfe von Methoden und Eigenschaften enthält. Eine Klasse, die ein Interface implementiert, braucht nämlich tatsächlich nur die Definitionen der Methoden und Eigenschaften, und eventuell dahinterstehender Code des »Originals« wird nicht benötigt. Statt dessen wird eigener Code für die implementierten Methoden und Eigenschaften eingefügt.

Sie sind jedoch nicht dazu gezwungen, unbedingt eine leere, abstrakte Klasse als Ursprung der Interface-Definition anzulegen. Enthält die Ursprungsklasse selbst Code, stört das keineswegs. So können Sie nun eine allgemeine Darlehensklasse anlegen, die von allen anderen Darlehensobjekten implementiert werden kann. Hier werden wir nun also das `clsBankLoan`-Objekt als Ursprung für ein Standard-Interface verwenden. Und so können wir auch wieder die Aggregation einsetzen, um den Code der Ursprungsklasse zu nutzen. Dies ermöglicht eine Form der Vererbung, bei der ein Objekt sowohl die Interface-Definiton als auch Teile des Codes einer Ursprungsklasse wiederverwenden kann. Erinnern Sie sich an Kapitel 3, »Objekte und Visual Basic«, zurück: Vererbung ist das dritte Merkmal einer objektorientierten Sprache.

Diese Form der Vererbung ist keine echte Vererbung auf Sprachebene, wie es die theoretische Definition des Begriffs eigentlich verlangt. Dennoch können Sie damit die meisten der per Vererbung zu lösenden Aufgaben in den Griff bekommen. Die Antwort auf die Frage, ob Visual Basic nun eine objektorientierte Sprache ist oder nicht, können Sie dann getrost akademischen Zirkeln überlassen.

Microsoft bevorzugt ganz klar abstrakte Klassen als Ursprünge für Interface-Definitionen. Ich kann dieser Sicht nicht ganz zustimmen, da ich den Verdacht habe, daß viele Programmierer vollständig implementierte Klassen wesentlich nützlicher finden. Am besten lernen Sie daher beide Möglichkeiten kennen und entscheiden dann selbst, was Ihnen lieber ist.

Listing 5.8 enthält die Überarbeitung der `clsSecurityLoan`-Klasse mit der Implementierung des `clsBankLoan`-Interface. Zwei Aspekte in diesem Listing sollten einmal näher beleuchtet werden.

Erstens nutzt dieses Objekt keine Aggregation. Daher werden die Methoden und Eigenschaften der `clsBankLoan`-Klasse von dieser Klasse nicht aufgerufen. Sie implementiert ihre eigenen Versionen der Funktionen (wie auch die Vorgänger in den früheren Beispielen). Das heißt also, dieses Objekt benutzt die `clsBankLoan`-Klasse nur zur Definition des implementierten Interface. Der Code der `clsBankLoan`-Klasse wird dagegen nicht genutzt.

Zweitens werden Sie feststellen, daß alle Methoden und Eigenschaften zweimal vorhanden sind. Sie sollen nämlich in beiden Interfaces des Objekts enthalten sein. Beispielsweise wird die Eigenschaft `Interest` sowohl vom `clsBankLoan`-Interface als auch vom `clsSecurityLoan`-Interface angeboten. Sie werden in Kürze sehen, daß diese Duplizität nicht unbedingt notwendig ist.

Beachten Sie auch, wie die Klasse sämtliche Funktionen des `clsBankLoan`-Interface implementiert.

```
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.  All Rights Reserved
' Kapitel 5

    ' Implementiert das clsBankLoan-Interface
    Implements clsBankLoan

    Option Explicit

    ' Darlehensbetrag
    Public AmountAvailable As Currency

    ' Laufzeit
    Public Duration As Integer

    ' Zinsfuß
    Public Interest As Double

    ' ****Margin requirement
    Public Margin As Double

    ' Raten berechnen
    Private Function Payment() As Currency
       Dim factor As Double
       Dim iper As Double
       iper = Interest / 12
       factor = iper * ((1 + iper) ^ Duration)
       Payment = AmountAvailable * factor / (((1 + iper) _
        ^ Duration) - 1)
    End Function

    ' Beschreibungs-String
    Public Function Summary() As String
       Summary = Format$(AmountAvailable, "Currency") & _
        "  " & Format$(Interest,_
       "Percent") & "  " & Duration & " months. Margin: " _
        & Format$(Margin, "Percent")
    End Function

    Public Function SourceType() As String
       SourceType = "Brokerage Loan"
    End Function

    ' Das clsBankLoan-Interface
    Private Property Let clsBankLoan_AmountAvailable( _
     ByVal RHS As Currency)
       AmountAvailable = RHS
    End Property
```

```
Private Property Get clsBankLoan_AmountAvailable() _
 As Currency
   clsBankLoan_AmountAvailable = AmountAvailable
End Property

Private Property Let clsBankLoan_Duration(ByVal RHS _
 As Integer)
   Duration = RHS
End Property

Private Property Get clsBankLoan_Duration() As Integer
   clsBankLoan_Duration = Duration
End Property

Private Property Let clsBankLoan_Interest(ByVal RHS _
 As Double)
   Interest = RHS
End Property

Private Property Get clsBankLoan_Interest() As Double
   clsBankLoan_Interest = Interest
End Property

Private Function clsBankLoan_Payment() As Currency
   clsBankLoan_Payment = Payment()
End Function

Private Function clsBankLoan_SourceType() As String
   clsBankLoan_SourceType = SourceType()
End Function

Private Function clsBankLoan_Summary() As String
   clsBankLoan_Summary = Summary()
End Function
```

Listing 5.8: Das clsSecurityLoan-Objekt, Version 2

Die Klasse `clsLoanShark` verwendet Aggregation (siehe Listing 5.9). Dies kommt dem Objekt zugute, da die Methoden und Eigenschaften der `clsBankLoan`-Klasse tatsächlich nur über das `clsBankLoan`-Interface angesprochen werden – diese in das `clsLoanShark`-Interface aufzunehmen, wäre Overkill. Dies reduziert die Code-Menge in der Klasse. So bleibt als einzige Funktion des `clsLoanShark`-Interface die Methode `LatePenalty` übrig, über die auch nur dieses Objekt verfügt.

```
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.  All Rights Reserved
' Kapitel 5
```

```
Option Explicit

' Implementiert das clsBankLoan-Interface
Implements clsBankLoan

' Internes Objekt in Aggregation genutzt
Private LoanTemplate As New clsBankLoan

Public Function LatePenalty() As String
   Select Case LoanTemplate.AmountAvailable
      Case 0 To 25000
         LatePenalty = "Broken Fingers"
      Case 25000 To 75000
         LatePenalty = "Broken arm"
      Case Else
         LatePenalty = "You don't want to know"
   End Select
End Function

Private Property Let clsBankLoan_AmountAvailable(ByVal _
 RHS As Currency)
   LoanTemplate.AmountAvailable = RHS
End Property

Private Property Get clsBankLoan_AmountAvailable() _
 As Currency
   clsBankLoan_AmountAvailable = _
    LoanTemplate.AmountAvailable
End Property

Private Property Let clsBankLoan_Duration(ByVal RHS _
 As Integer)
   LoanTemplate.Duration = RHS
End Property

Private Property Get clsBankLoan_Duration() As Integer
   clsBankLoan_Duration = LoanTemplate.Duration
End Property

Private Property Let clsBankLoan_Interest(ByVal RHS _
 As Double)
   If RHS < 0.5 Then RHS = RHS + 0.5
   LoanTemplate.Interest = RHS
End Property

Private Property Get clsBankLoan_Interest() As Double
   clsBankLoan_Interest = LoanTemplate.Interest
End Property
```

```
    Private Function clsBankLoan_Payment() As Currency
       clsBankLoan_Payment = LoanTemplate.Payment
    End Function

    Private Function clsBankLoan_SourceType() As String
       clsBankLoan_SourceType = "Loan Shark"
    End Function

    Private Function clsBankLoan_Summary() As String
       clsBankLoan_Summary = LoanTemplate.Summary
    End Function
```

Listing 5.9: Das clsLoanShark-Objekt, Version 2

Nun bleibt uns nur noch übrig, uns dem Beispielprogramm Loan4 selbst zuzuwenden. Listing 5.10 enthält den Code des Hauptformulars, der so modifiziert wurde, daß er die Vorteile der Implementierung des `clsBankLoan`-Interface in allen Klassen nutzt.

```
' ActiveX: Guide to the Perplexed
    ' Copyright (c) 1997 by Desaware Inc.  All Rights Reserved
    ' Kapitel 5

    Option Explicit

    ' Array der verfügbaren Darlehen
    Dim Loans() As clsBankLoan

    ' Konstante zu internen Testzwecken
    Const LOANCOUNT = 100

    Private Sub Form_Load()
       Dim loannum%
       Dim secLoan As clsSecurityLoan

       ' Liste der verfügbaren Darlehen laden
       ReDim Loans(LOANCOUNT)
       For loannum = 1 To LOANCOUNT
          Select Case Int(Rnd() * 3)
             Case 0
                Set Loans(loannum) = New clsBankLoan
             Case 1
                Set Loans(loannum) = New clsSecurityLoan
                ' Margin kommt nur bei diesem Typ vor
                Set secLoan = Loans(loannum)
                secLoan.Margin = Rnd()
             Case Else
                Set Loans(loannum) = New clsLoanShark
```

```
      End Select
      With Loans(loannum)
         .AmountAvailable = CLng(Rnd() * 200000)
         .Duration = 12 * Int((Rnd * 30) + 1)
         .Interest = (7 + Int((Rnd * 80) * 0.125)) / 100
         lstLoans.AddItem .Summary
      End With
   Next loannum
End Sub

Private Sub lstLoans_Click()
   Dim loannum%
   Dim LatePenaltyValue$
   Dim LoanSharkObject As clsLoanShark

   loannum = lstLoans.ListIndex + 1
   lstInfo.Clear
   With Loans(loannum)
      lstInfo.AddItem .SourceType
      lstInfo.AddItem .Summary
      lstInfo.AddItem "Payment: " & _
       Format$(.Payment, "Currency")
      ' LatePenaltyValue = _
      ' GetLatePenalty(Loans(loannum))
      ' Dies funktioniert nun nicht mehr!
      'If LatePenaltyValue <> "" Then
      '   lstInfo.AddItem "Late payment penalty:"
      '   lstInfo.AddItem "   " & LatePenaltyValue
      'End If
      If TypeOf Loans(loannum) Is clsLoanShark Then
         Set LoanSharkObject = Loans(loannum)
         lstInfo.AddItem "Late payment penalty:"
         lstInfo.AddItem "   " _
          & LoanSharkObject.LatePenalty
      End If
   End With
End Sub

' Allgemeine Funktion zum Auslesen
' des LatePenalty-Wertes
'Public Function GetLatePenalty(obj As Object) As String
'   On Error GoTo nofunction
'   GetLatePenalty = obj.LatePenalty
'   Exit Function
'nofunction:
'End Function
```

Listing 5.10: Listing für lnSel4.frm

Die erste und wichtigste Änderung ist, daß das Array `Loans()` jetzt wieder als `clsBankLoan` deklariert wird. Das geht nun, da alle Objekte über das `clsBankLoan`-Interface verfügen, das diesem Array zugewiesen werden kann. Im `Form_Load`-Ereignis werden nach wie vor die verschiedenen Objekttypen instanziert. Doch immer wenn ein Objekt zugewiesen wird, führt Visual Basic einen Zugriff auf die Funktion `QueryInterface` aus, um einen Zeiger auf das `clsBankLoan`-Interface zu erhalten. Dieser Zeiger kann nun dem Array zugewiesen werden.

Das Objekt `clsSecurityLoan` erfordert eine spezielle Behandlung, da im `Form_Load`-Ereignis zusätzlich die `Margin`-Eigenschaft gesetzt werden muß. Das Problem besteht darin, daß `Margin` nicht zum `clsBankLoan`-Interface gehört. Um diese Eigenschaft setzen zu können muß eine Hilfsvariable, `secLoan` genannt, angelegt werden, über die der volle Zugriff auf das `clsSecurityLoan`-Interface erfolgen kann. Nachdem das Objekt auch dieser Variablen zugewiesen worden ist, kann die Margin-Eigenschaft gesetzt werden.

Es gibt noch eine weitere wesentliche Änderung in diesem Code. Die frühere Technik zur Prüfung des `LatePenalty`-Wertes funktioniert nicht länger. (Versuchen Sie doch einmal, die Auskommentierung des Codes rückgängig zu machen und das Projekt zu starten.) Die Eigenschaft `LatePenalty` wird nicht gefunden. Nun, das mag verwirrend erscheinen. Immerhin akzeptiert doch die Funktion `GetLatePenalty` eine Objektreferenz, die das `IDispatch`-Interface verwendet, das spät gebunden ist. Und das `IDispatch`-Interface kann doch auf sicherem Wege prüfen, ob die Eigenschaft `LatePenalty` vorhanden ist, nicht wahr? So hatten wir es doch zuvor erklärt.

Was da los ist, sehen Sie nach einem zweiten Blick auf Abbildung 5.4. Ja, die Funktion `GetLatePenalty` verwendet das `IDispatch`-Interface des `clsLoanShark`-Objekts, aber welches nun wirklich? Das `clsLoanShark`-Objekt verfügt nämlich über zwei `IDispatch`-Interfaces: eines für das `clsBankLoan`-Interface und eines für das `clsLoanShark`-Interface. (Sie erinnern sich: Bei einem dualen Interface treffen Sie sowohl das direkte Interface als auch das entsprechende `IDispatch`-Interface an.)

Die `GetLatePenalty`-Funktion bekommt das `IDispatch`-Interface des `clsBankLoan`-Interface geliefert, das nie über die `LatePenalty`-Funktion verfügte – dieser Ansatz funktioniert hier also nicht. Sie müssen dazu zurückkehren, bei jedem einzelnen Objekt nachzuprüfen, ob es sich um ein Objekt handelt, das die `LatePenalty`-Funktion unterstützt.

Wie entscheidet Visual Basic darüber, welches `IDispatch`-Interface in Fällen wie diesem zu verwenden ist? Gar nicht – Visual Basic gibt immer einen Zeiger auf das zuletzt verwendete Interface zurück. In dieser Anwendung fand der letzte Zugriff auf ein Objekt über das `Loans()`-Array statt, das das `clsBankLoan`-Interface verwendet. Sie sollten niemals davon ausgehen, daß ein bestimmtes Interface verwendet wird, wenn Sie ein Objekt mit mehreren Interfaces an eine Funktion übergeben, die einen allgemeinen `Object`-Parameter akzeptiert, solange Sie nicht

ein bestimmtes Interface ausdrücklich dadurch festlegen, daß Sie das Objekt zunächst einer Variablen zuweisen, die dem gewünschten Interface entsprechend deklariert ist.

Gibt es einen Weg, die Prüfung der einzelnen Objekte auf das Vorhandensein der `LatePenalty`-Funktion zu umgehen?

Ja: Sie können dem `clsBankLoan`-Interface eine Funktion hinzufügen, die immer eine Referenz auf das andere `IDispatch`-Interface des Objekts zurückgibt. Sie könnten diese Funktion etwa `MainInterface` nennen und folgendermaßen implementieren:

```
Private Function clsBankLoan_MainInterface() As Object
            Dim myobj As clsLoanShark
            Set myobj = Me
            Set clsBankLoan_MainInterface = myobj
     End Function
```

Diese Prozedur wird im Beispiel Loan5 demonstriert. Hier verwendet das `lstLoans_Click`-Ereignis den folgenden Code, um die `LatePenalty`-Funktion zu finden, falls diese vorhanden ist:

```
LatePenaltyValue = GetLatePenalty(Loans(loannum).MainInterface)
     If LatePenaltyValue <> "" Then
        lstInfo.AddItem "Late payment penalty:"
        lstInfo.AddItem "   " & LatePenaltyValue
     End If
```

Eines ist bei diesem Ansatz zu beachten: Sie sollten sich rechtzeitig dafür entscheiden. Wenn Sie sich das Loan5-Beispiel genau anschauen, werden Sie sehen, daß die Erweiterung des `clsBankLoan`-Interface das nachträgliche Einfügen der `MainInterface`-Funktion in allen Klassen nach sich zieht, die dieses Interface implementieren. In diesem Beispiel ist das kein großes Unglück – aber doch eines, wenn Sie auf der Basis dieses Interface ActiveX-Komponenten entwickelt haben sollten. Wenn andere Projekte diese Klasse verwenden, laufen Sie Gefahr, daß diese nicht mehr funktionieren, da Sie durch die Ergänzung den Interface-Vertrag verletzt haben!

Die COM-Technologie, die von Visual Basic und ActiveX genutzt wird, bietet viele Fähigkeiten und Vorteile. Sie erfordert es im Gegenzug jedoch, daß Sie sich frühzeitig gründliche Gedanken über die Definition Ihrer Objekte machen. Es stellt kein Problem dar, ein Interface während der Entwicklung zu ändern oder zu ergänzen. Haben Sie jedoch ein Objekt, das auf diesem Interface beruht, erst einmal veröffentlicht und »auf die ganze Welt losgelassen«, werden Sie für immer und ewig mit diesem Interface leben müssen.

Werfen wir noch einen letzten Blick auf diese Beispiele. Sie werden erkennen, daß das Beispiel Loan4 mit einem großen Vorteil gegenüber dem Beispiel Loan3 aufwartet: Alle Objektmethoden und -eigenschaften sind früh gebunden. Das Bei-

spiel Loan5 bewahrt diesen Vorteil mit Ausnahme des `LatePenalty`-Tests, bei dem in diesem Fall nur eine späte Bindung möglich ist.

5.4.1 Vor- und Nachteile

In diesem Kapitel werden eine Reihe von Techniken für die Entwicklung von Objekten mit Visual Basic gezeigt. Ich habe mich bemüht, die Vor- und Nachteile der einzelnen Ansätze aufzuzeigen. Sie sollten sich über folgendes im klaren sein:

- über den Einfluß, den frühe und späte Bindung auf die Performance haben;
- über die Wiederverwendbarkeit von Code und die langfristige Zuverlässigkeit bei der Nutzung von Aggregation und der Delegation an vorhandene Funktionen;
- über die Kosten an Performance und Ressourcen, die beim Anlegen und Freigeben von eingeschlossenen Objekten bei der Nutzung von Aggregation entstehen.

Alle diese Techniken stehen Ihnen als Visual-Basic-Programmierer zur Verfügung. Es bleibt jedoch Ihnen überlassen, die für Sie geeignetste zu wählen.

5.4.2 Über ActiveX-Interfaces

Nun haben Sie in diesem Kapitel gesehen, wie Sie mehrere Interfaces Ihrer Objekte auf verschiedene Weise nutzen können:

- Sie können ein gemeinsames Interface definieren, das von verschiedenen Objekttypen zur Unterstützung von gemeinsamer Funktionalität genutzt werden kann.
- Sie können die Funktionalität eines Objekts erweitern, indem Sie ein neues Interface hinzufügen, während Sie die bisherige Funktionalität über ältere Interfaces weiterhin unterstützen.
- Sie können die Vorteile der frühen Bindung zur Performance-Steigerung nutzen.

Nun, tatsächlich nutzen die wenigsten Visual-Basic-Programmierer die Möglichkeiten mehrfacher Interfaces. Schauen wir der Wahrheit ins Gesicht: Die meisten Visual-Basic-Programmierer sind derzeit noch damit beschäftigt, mit dem Gedanken der objektorientierten Programmierung vertraut zu werden. Die Unterstützung mehrerer Interfaces sieht dabei erst einmal nach einem höheren Grad an Komplexität aus, die scheinbar keinen allzu großen Vorteil für gewöhnliche Anwendungen zu bieten hat.

Wenn Sie lediglich eigenständige Anwendungen entwickeln, stellen mehrere Interfaces wohl tatsächlich nur einen höheren Grad an Komplexität dar, ohne wirklich Vorteile zu bieten, vor allem bei der Entwicklung gewöhnlicher Datenbankanwendungen.

Nichtsdestotrotz ist es dennoch wichtig, daß Sie mit diesen Konzepten und Techniken vertraut werden. Denn früher oder später werden Sie doch wiederverwendbare Software-Komponenten entwickeln wollen oder müssen. Ich kann es Ihnen auch »auf die harte Tour« klarmachen: Wenn Sie nicht baldmöglichst damit anfangen, wiederverwendbare Software-Komponenten zu entwickeln, werden diejenigen unter Ihren Konkurrenten davonziehen, die damit schon längst begonnen haben – und Sie werden Ihre Marktanteile schwinden sehen. Spätestens dann werden Sie dazu gezwungen sein, wiederverwendbare Software-Komponenten anzubieten, ob Sie wollen oder nicht. Daher kann es nie zu früh sein, sich mit all diesem Stoff vertraut zu machen. Aber da Sie sich ja die Mühe machen, dieses Buch zu lesen, werden Sie das längst wissen.

Bei der Entwicklung wiederverwendbarer Software-Komponenten jedoch wird die Nutzung mehrerer Interfaces zur Erweiterung der Fähigkeiten dieser Komponenten sowohl essentiell als auch zur Routine. Wenn Sie diese Techniken nicht nutzen, werden neuere Versionen Ihrer Komponenten nicht mehr mit Anwendungen funktionieren, die die alten Versionen nutzen, oder diese Anwendungen werden gar nicht mehr funktionieren – auf dieses Problem bin ich in diesem Buch bereits eingegangen.

Denken Sie nun jedoch nicht, daß Sie immer und überall mehrere Interfaces unterbringen müßten. Erst bei komplexeren Geflechten von Objekten werden Sie einen Punkt erreichen, an dem gemeinsam genutzte Funktionalität über gemeinsame Interfaces wirklich sinnvoll und notwendig wird. In der Regel ergibt sich das Hinzufügen von Interfaces für neue Funktionalität erst dann, wenn auch die Komponenten erweiterte Funktionalität zu bieten haben. Aber dennoch gibt es eine Stelle, an der mehrfache Interfaces eine große Bedeutung bekommen: an einer Stelle, an der Sie nicht nur einen hohen Grad an Komplexität antreffen werden, sondern auch rasch aufeinanderfolgende Versionswechsel. Wo anders könnte diese Stelle anzutreffen sein als beim Betriebssystem selbst?

Windows und COM

Sie sind es höchstwahrscheinlich gewohnt, daß Windows zum großen Teil über Aufrufe der API-Funktionen gesteuert wird, die von den verschiedenen Windows-DLLs exportiert werden. Windows wird jedoch in zunehmendem Maße über COM-Objekte gesteuert. Es gibt Gerüchte, die da lauten, daß es sich eines Tages erübrigen wird, jemals eine API-Funktion aufzurufen – das gesamte Betriebssystem soll über ein komplexes Objektmodell offengelegt werden. ActiveX (bzw. OLE) ist ein wenig mehr als nur ein Satz von Standard-Interfaces, die unter COM definiert werden und von jedem COM-Objekt verwendet werden können. Das bedeutet, daß im Idealfall Visual-Basic-Anwendungen Referenzen auf System-Objekte erhalten können und die von diesen offengelegten Methoden aufrufen können, und daß Visual-Basic-Objekte Standard-Interfaces implementieren können, die von anderen Anwendungen genutzt werden können.

Nun, das stimmt so nicht ganz.

Es gibt ein paar wenige, aber wesentliche Beschränkungen in der Art und Weise, wie Visual Basic mit Interfaces umgeht, die dem Erreichen dieses Ideals im Wege stehen.

In Kapitel 3 haben Sie von Automation-Interfaces gehört und davon, wie hinter Visual-Basic-Klassen Objekte stehen, die duale Interfaces unterstützen – das verborgene Klassen-Interface und das `IDispatch`-Interface. Nicht jedes Objekt unterstützt duale Interfaces – tatsächlich fehlt den meisten System-Objekten die Unterstützung des `IDispatch`-Interface. Das alleine wäre kein Problem, da Visual Basic auch mit Nicht-`IDispatch`-Interfaces glücklich werden kann, jedoch besteht Visual Basic darauf, daß die Interfaces automationskompatibel sein müssen.

Was soll das heißen?

Sie wissen, daß es per Automation möglich ist, Methoden und Eigenschaften indirekt anhand eines gegebenen Namens einer Funktion aufzurufen. Dabei kommt jedoch zum Vorschein, daß das `IDispatch`-Interface nicht alle für Funktionsparameter möglichen Datentypen unterstützt. Es werden lediglich Datentypen für Parameter unterstützt, die in einem `Variant` abgelegt werden können. Wenn nun ein Interface nur Datentypen erfordert, die mit dem `IDispatch`-Interface verwendet werden können, wird es automationkompatibel genannt. Werden jedoch Datentypen verwendet, die vom `IDispatch`-Interface nicht unterstützt werden, kann Visual Basic Funktionen eines solchen Interface nicht aufrufen.

Aus der Sicht eines Visual-Basic-Programmierers ist es von größerer Bedeutung, daß Visual Basic selbst nicht jeden Funktions-Parametertyp unterstützt. So werden beispielsweise vorzeichenlose Long-Werte nicht unterstützt. Visual Basic kann nur Funktionen von Objekt-Interfaces aufrufen, die automationkompatibel sind und Visual-Basic-Datentypen unterstützen.

Leider sind viele Standard-Interfaces nicht für Visual Basic verwendbar. So bleiben nur zwei Möglichkeiten für Sie offen:

- Sie lernen, wie man Typbibliotheken anlegt, die alternative Funktionen für Standard-Interfaces anbieten. Sie sollten diesen Weg mit Bedacht gehen und darauf achten, nicht versehentlich die originalen Definitionen des Systems zu ändern. Andernfalls könnten Sie die Systeme, auf denen Ihre Anwendungen laufen sollen, ganz schön durcheinanderbringen.
- Sie können ein Produkt eines Drittanbieters verwenden, wie etwa SpyWorks von Desaware, mit dessen Hilfe Sie jedes beliebige Interface implementieren oder verwenden können, unabhängig von dessen Automationskompatibilität.

Sie werden Beispiele für die Implementierung von nützlichen System-Interfaces im Teil II, »Code-Komponenten«, kennenlernen, wo System-Interfaces zur Erweiterung der Funktionalität von mit Visual Basic geschriebenen ActiveX-Controls dienen werden.

Da Sie nun ein wenig darüber wissen (nein, eigentlich wissen Sie jetzt bereits eine ganze Menge darüber), wie COM-Objekte funktionieren, wird es Zeit, die nächste Stufe zu erklimmen und einen Blick darauf zu werfen, wo COM-Objekte »leben« können: in Dynamic Link Libraries (DLLs) und auch in anderen Anwendungen.