Kapitel 6

Das Leben und die Lebensdauer einer ActiveX-Komponente

6.1 Objekte: Sind sie real oder nur Speicher-Gespinste? 132
6.2 Prozeßräume: die unüberwindbaren Grenzen 135
6.3 Der Lebenszyklus eines DLL-Objekts 140
6.4 Der Lebenszyklus eines EXE-Objekts 148
6.5 Der Lebenszyklus eines Remote-Objekts 153

Eines werden Sie sicher bestätigen können, nachdem Sie bis hierhin mitgelesen haben: Ich gehöre zu den Leuten, die den Dingen auf den Grund gehen wollen. Aber das ist nicht bloß Neugier aus reinem Selbstzweck. Ich bin der Meinung, daß es viel leichter fällt, sich eine Sache zunutze zu machen, wenn man versteht, wie sie funktioniert.

Der Trick dabei ist, die richtige Balance zwischen einem Ausloten einer Technologie bis in den letzten Winkel und einem oberflächlichen Überfliegen zu finden. Übertreiben Sie es, kommen Sie unter Umständen gar nicht mehr dazu, die gewonnenen Kenntnisse auch in der Praxis verwenden zu können. Belassen Sie es bei einem flüchtigen Blick unter die Oberfläche, werden Sie kaum mehr zuwege bringen können, als Beispiele anderer Leute abzukupfern. Und letzteres geht nur so lange gut, wie diese Beispiele auch annähernd Ihre Problemstellungen treffen. Sobald Sie jedoch über ein Beispiel hinausgehen wollen, werden Sie auf dem Schlauch stehen, da Sie dann keine Ahnung von den Dingen haben werden, um die es geht.

In bezug auf ActiveX hoffe ich, beim Schreiben dieser Kapitel die richtige Balance gefunden zu haben. Ich habe versucht, Sie mit den konzeptionellen Hintergründen vertraut zu machen, ohne Sie mit den Details zu belästigen, wie COM-Objekte tatsächlich implementiert werden. Falls Sie diese Details allerdings doch interessieren sollten, dann verweise ich Sie auf das Buch von Kraig Brockschmidt, *Inside OLE, Second Edition*, Microsoft Press 1995, in dem die interne Implementierung bis ins Detail abgehandelt wird.

Wir müssen uns noch mit einem Thema beschäftigen, ehe wir uns den Details der Objekterstellung unter Visual Basic zuwenden können. Sie sollten noch etwas darüber wissen, wie ActiveX-Objekte ins Leben gerufen werden, und darüber, wie Sie auf und in Ihrem System leben. Ohne dieses Wissen werden Sie kaum in der Lage sein zu entscheiden, welches der verschiedenen ActiveX-Objekte, die mit Visual Basic erstellt werden können, für die jeweiligen Anforderungen das richtige sein wird.

6.1 Objekte: Sind sie real oder nur Speicher-Gespinste?

Führen wir uns noch einmal die Merkmale eines COM-Objekts vor Augen:

- Es hat einen GUID – eine global eindeutige Kennung;
- es unterstützt mehr als nur ein Interface;
- es können Daten mit ihm assoziiert sein, auf die man jedoch nur über die Interfaces zugreifen kann (denken Sie daran, daß auch auf öffentliche Variablen einer Visual-Basic-Klasse im Hintergrund über separate, zum Klassen-Interface gehörende `Property Let`- und `Property Get`-Prozeduren zugegriffen wird).

Nun, diese Auflistung klingt irgendwie wenig konkret greifbar. Ein »GUID« ist eine Zahl. »Interfaces« sind eine Liste von Funktionsdeklarationen – ein Vertrag,

um es einmal so zu sagen. Der Begriff »Daten« bleibt so lange abstrakt, bis Sie tatsächlich auf eine konkrete Stelle im Arbeitsspeicher weisen und sagen können: »Hier sind sie.«

Das ist ja alles schön und gut – aber man kann keine Zahl laden, keinen Vertrag ausführen oder Abstraktionen manipulieren. (Allerdings hatte ich ein paar College-Professoren, die das alles zustande zu bringen schienen – aber irgendwie machte der Unterricht bei ihnen keinen Spaß). Ein Objekt muß schon irgendwie greifbarer sein, damit wir es in einer Anwendung einsetzen können. Genauer gesagt, hinter einem Interface muß konkreter Code stehen, der in den vorhandenen Arbeitsspeicher geladen und ausgeführt werden kann. Die Daten eines Objekts müssen ebenfalls irgendwo im Speicher existieren. Und es muß irgendeinen Mechanismus des Betriebssystems geben, der einen GUID entgegennimmt und irgendwie daraus das dazugehörende Objekt zaubert.

Schieben wir das GUID-Thema noch ein wenig in diesem Kapitel nach hinten, und konzentrieren wir uns fürs erste auf zwei Aspekte:

- Hinter einem Interface muß konkreter Code stehen.
- Die Daten eines Objekts müssen im Speicher vorhanden sein.

Das klingt einfacher, als es ist. Denn nichts von alledem, was ich bisher zum besten gegeben habe, läßt auf eine Eins-zu-eins-Beziehung zwischen dem Interface eines Objekts und der eigentlichen Implementierung dieses Interface schließen. Dies wirft eine interessante Frage auf: Könnten Dutzende verschiedene Programme und DLLs alle ihre eigene Code-Implementierung eines bestimmten Interface eines bestimmten Objekts enthalten?

Ebenso läßt bisher nichts darauf schließen, daß die Daten eines Objekts im Arbeitsspeicher vorhanden sein und zu irgendeiner Anwendung oder auch nur einem bestimmten System in einem Netzwerk gehören müssen. Wäre es etwa möglich, daß der Code hinter einem Interface auf einem System Funktionen ausführt, wobei Daten manipuliert werden, die auf irgendeinem anderen System auf der anderen Seite des Globus existieren?

Beide Fragen sind zu bejahen – allerdings steckt mehr dahinter, als es auf den ersten Blick scheinen mag.

Das Beruhigende daran ist, daß Sie die eher etwas esoterisch anmutenden Varianten der Geschichte ruhigen Gewissens ignorieren können. Unter Visual Basic werden nur dann verschiedene Code-Implementierungen eines Objekts auftauchen, wenn Sie neue Versionen Ihres Objekts schreiben. Das Management von Objekten über ein Netzwerk hinweg wird ganz ordentlich von DCOM oder Remote-Automation erledigt, einem Thema, dem wir uns in diesem Buch nur am Rande widmen werden.

Es gibt jedoch einen Aspekt, der für Ihre Komponenten von großer Bedeutung ist. Dieser bestimmt nicht nur die möglichen Fähigkeiten Ihrer Objekte, sondern hat

einen erheblichen Einfluß auf deren Performance. Sehen Sie, auf einem 32-Bit-Betriebssystem laufen die einzelnen Anwendungen in ihren eigenen Prozeßräumen. Sie laufen zwar alle auf der gleichen Maschine, könnten aber auch genauso gut auf verschiedenen Maschinen laufen.

Wenn Sie sich mit der Win32-Programmierung auskennen, werden Sie angesichts des Begriffs *Prozeßraum* wissend mit dem Kopf nicken. Wahrscheinlich wird Ihnen der Inhalt des nächstfolgenden Abschnitts dieses Kapitels bestens bekannt sein, so daß Sie diesen überspringen können.

Sollten Sie jedoch zur breiten Schicht derjenigen gehören, die sich zwar ausgiebig auf 16-Bit-Feldern getummelt haben, sich aber in der 32-Bit-Welt noch unsicher fühlen, dann sollten Sie weiterlesen. Doch zunächst folgt ein kleines Intermezzo.

6.1.1 Intermezzo

(Sollte zur Melodie von *The Way We Were* gesungen werden)

Memory

Like the code I've left behind

Misty-banked extended memory

Oh, the way things were.

Six-forty K

And the crashes that we saw.

From UAE to GP fault,

Oh, the way things were.

Could it be that things were all so simple then

Wasting time rebooting one more time?

If we could write each application once again,

Tell me, would we?

Could we?

Memory

Wasn't bountiful and yet

What we used to squeeze in 10K

Now takes over 20 meg!

So let's remember

Whenever we assemble,

Who has time to remember?

The way things were

The way things were...

6.2 Prozeßräume: die unüberwindbaren Grenzen

Ob Sie es mit Code oder mit Daten zu tun haben, eines gilt in jedem Fall: Die Information muß im Arbeitsspeicher existieren, damit der Prozessor etwas damit anfangen kann. Wie jedoch der Speicher organisiert ist, hat einen enormen Einfluß sowohl auf die Performance als auch auf die Stabilität.

In den Tagen der 16-Bit-Betriebssysteme (Windows 3.x) glich der Arbeitsspeicher einem weiten Meer, in dem sich die Anwendungen tummelten. Code und Daten konnten bunt durcheinandergewirbelt werden, selbst wenn sie aus verschiedenen Anwendungen stammten. Abbildung 6.1 illustriert diese Situation anhand zweier Anwendungen und einer DLL. Puristen mögen anmerken, daß die Darstellung der linearen Anordnung des physischen Speichers und die interne Architektur entfallen ist, wobei zwischen unterem und erweitertem Speicher und zwischen physischem und virtuellem Speicher unterschieden werden müßte.

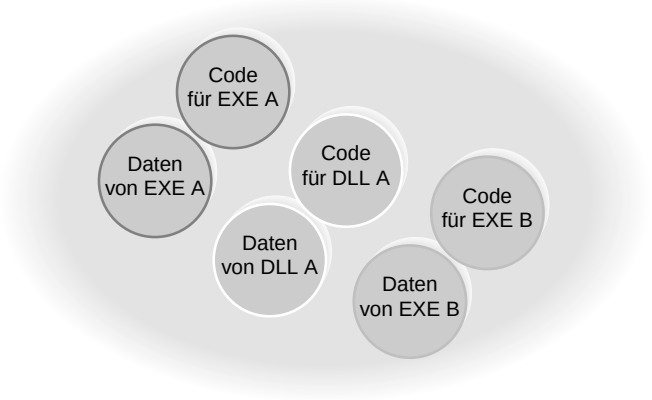

Abb. 6.1: Speicherorganisation unter Windows 3.x

Was Puristen kümmern mag, hat allerdings keinen Einfluß darauf, wie wir Programmierer gewöhnlich den Speicher zu betrachten pflegen. Aus unserer Windows-Perspektive sieht der Speicher aus wie eine große Masse, von der Blöcke mit Beschlag belegt werden können (»Allokieren von Speicher«). Wir kümmern

uns weder darum, wo genau sich diese Blöcke befinden, oder in welcher Reihenfolge sie sich befinden, noch darum, ob sie gerade in den physischen Speicher geladen oder auf die Festplatte ausgelagert sind.

Halten wir einen Moment inne und vergegenwärtigen wir uns die Unterschiede zwischen einer DLL (Dynamic Link Library) und einem ausführbaren Programm (EXE). Zunächst ist es wichtig zu begreifen, daß DLLs und EXEs grundsätzlich dasselbe sind – sie verwenden das gleiche Dateiformat (mit leicht voneinander abweichenden Optionen) und werden auf fast dieselbe Weise geladen. Beide können sowohl Code als auch Daten enthalten. Der einzige tatsächliche Unterschied besteht darin, wie Windows sie behandelt.

Wenn Windows eine DLL lädt, wird nur deren Initialisierungs-Code ausgeführt. Danach bleibt die DLL sich selbst überlassen. Funktionen in einer DLL werden nur dann aufgerufen, wenn eine Anwendung diese ausdrücklich anfordert. Wenn Windows jedoch eine ausführbare Datei (EXE) lädt, ist der Initialisierungs-Code dafür verantwortlich, das Ding anzulegen, was manchmal *die Nachrichtenpumpe* genannt wird – im wesentlichen ist dies eine Programmschleife, die so lange in Betrieb bleibt, wie die Anwendung läuft. Diese Nachrichtenpumpe ruft Nachrichten des Betriebssystems ab. Windows vermerkt die Anwendung als eigenständigen Task, sendet diesem bei Bedarf Nachrichten und weist ihm Teile der Prozessorzeit zu.

DLLs sind dafür konzipiert, von einer Anwendung geladen zu werden, wann immer diese Zugriff auf Funktionen in der DLL benötigt. Im Endeffekt ist eine DLL eine gemeinsam genutzte Bibliothek. Der Aufruf einer Funktion ist eine recht schnelle Angelegenheit, sobald die DLL erst einmal geladen ist. Der Aufruf einer Funktion, die sich in einer anderen Anwendung befindet, sollte doch eigentlich genauso schnell sein. Das gilt jedoch nur, solange ein direkter Zugriff auf den Speicher besteht. Tatsächlich ist dies jedoch nicht der Fall. Der Grund liegt darin, daß Programme nahezu nie Funktionen in anderen Anwendungen direkt aufrufen. Statt dessen senden sie Nachrichten an andere Anwendungen. Das Senden einer Nachricht jedoch bringt im Vergleich zu einem direkten Funktionsaufruf einigen Overhead mit sich.

Die in Abbildung 6.1 dargestellte Speicherorganisation birgt einen schwerwiegenden Mangel in sich. Da sich der gesamte Code und alle Daten aller Anwendungen und DLLs im gleichen Speicher-»Meer« befinden, besteht die Möglichkeit, daß die einzelnen Speicherbereiche interferieren. Solange aller Code fehlerfrei ist und nur auf seine eigenen Speicherbereiche zugreift, ist auch alles in Ordnung (aber wir wissen ja alle, wie fehlerfrei Code gewöhnlich ist). Falls jedoch ein Programm zufällig Speicher modifiziert, der zu einer anderen Anwendung oder gar dem Betriebssystem selbst gehört, droht nicht nur dem betroffenen Programm die Gefahr eines Absturzes, sondern sogar dem gesamten System.

Microsofts 32-Bit-Betriebssysteme verwenden eine radikal andere Speicherarchitektur, die dieses Problem weitgehend eliminiert. Jede ausführbare Datei läuft in ihrem eigenen Prozeß, und das Betriebssystem verteilt die verfügbare CPU-Zeit

zwischen den laufenden Prozessen. Das Speicher-»Meer«, in dem ein Prozeß läuft, wird *Prozeßraum* der Anwendung genannt. Dies wird in Abbildung 6.2 dargestellt. Wie Sie sehen, ist der Prozeßraum jeder Anwendung von den Prozeßräumen der anderen Anwendungen und des Betriebssystems abgeschottet. Eine Anwendung kann nicht auf Speicher zugreifen, der einer anderen Anwendung gehört – es sei denn, beide Prozesse treffen über spezielle, in das Betriebssystem eingebaute Techniken eine ausdrückliche Übereinkunft über die gemeinsame Verwendung eines Speicherbereichs. Darüber hinaus können Anwendungen miteinander kommunizieren, indem sie sich Nachrichten zusenden, und sie können auf Betriebssystemmittel zurückgreifen, um Daten von einem Prozeßraum in einen anderen zu kopieren.

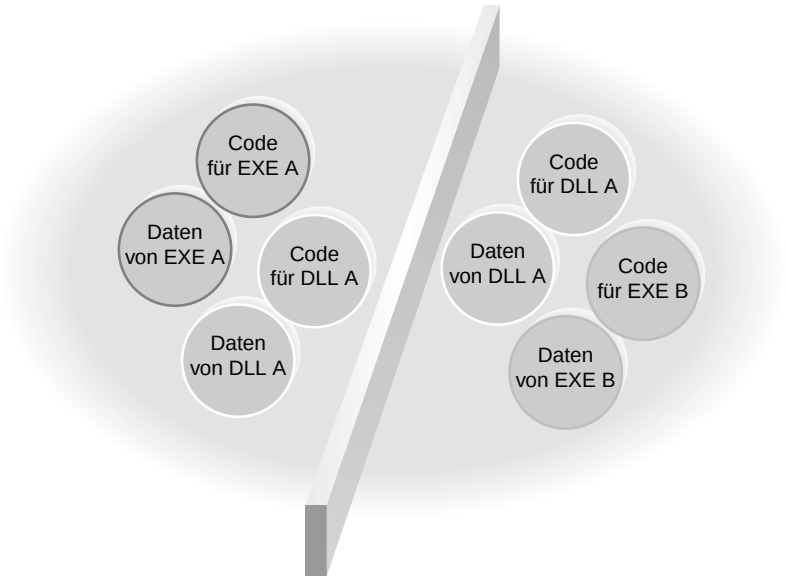

Abb. 6.2: Speicherorganisation unter Win32

Fast schon einer Spitzfindigkeit entspricht die Verwendung von DLLs unter Win32. In Abbildung 6.2 wird suggeriert, daß eine DLL separat für jeden Prozeß, der sie verwendet, in dessen Prozeßraum geladen wird. Vom Konzept her stimmt das so auch, selbst wenn es wie Speicherverschwendung aussieht. Denn ein Vorteil von DLLs soll ja schließlich darin bestehen, daß viele Anwendung den Code der DLL gemeinsam nutzen können sollen. Keine Bange – auch wenn es so aussieht, als ob aus der Sicht der Anwendungen eine DLL mehrfach geladen zu werden scheint, wird auf einer niedrigeren, physischen Ebene der Speicherorganisation die DLL nur einmal geladen und tatsächlich gemeinsam genutzt. Deren Datensegmente werden jedoch für jeden Prozeß separat angelegt, so daß es nicht mehr wie unter 16-Bit-Betriebssystemen möglich ist, Daten über eine DLL gemeinsam zu nutzen. Es muß jedoch angemerkt werden, daß diese Beschreibung

in erster Linie auf Windows NT zutrifft. Unter Windows 95/98 wird der Speicher für DLLs und das Betriebssystem gemeinsam genutzt – dies ist einer der Gründe, weswegen Windows 95/98 nicht ganz so stabil und zuverlässig wie Windows NT ist.

Wenn Sie einen tieferen Einblick in die Organisation von Speicher und Prozessen unter Win32 gewinnen möchten, finden Sie diesen in den Kapiteln 14 und 15 meines Buches *Dan Appleman's Visual Basic 5.0 Programmer's Guide to the Win32 API*, ZD Press 1997.

6.2.1 Zurück zu COM-Objekten

Wie bereits erwähnt, existiert ein COM-Objekt im Speicher sowohl in Form von Code (zur Implementierung der Interfaces) als auch in Form von Daten. Der Code eines COM-Objekts kann sich in einer ausführbaren Datei oder in einer DLL befinden. Die Daten müssen immer innerhalb eines einzelnen Prozeßraums existieren (aus naheliegenden Gründen ist dies derselbe Prozeßraum, der auch den implementierten Code enthält).

Dies bringt wieder ein interessantes Problem auf den Tisch: Wenn doch 32-Bit-Betriebssysteme undurchdringliche Mauern zwischen Prozessen errichten – wie kann denn dann eine Anwendung auf ein Objekt zugreifen, das von einer anderen Anwendung angelegt und implementiert worden ist?

Nun, zunächst steht fest, daß die Behandlung des Datenanteils konsequent erfolgt. Auf die Daten eines Objekts kann nur vom Code aus zugegriffen werden, der sie implementiert. Dieser Code wird sich immer im gleichen Prozeßraum wie die Daten befinden. Wenn wir also davon sprechen, auf ein Objekt einer anderen Anwendung zuzugreifen, dann sprechen wir in Wirklichkeit über die Möglichkeit, einen Interface-Zeiger für das Objekt zu erhalten und Funktionen aufzurufen, die zu diesem Interface gehören. Auf die Daten wird jedoch niemals direkt zugegriffen.

Doch selbst ein Interface wirft ein Problem auf. Wegen der Trennung der Prozeßräume ist es nicht möglich, Funktionen, die zu einem anderen Prozeß gehören, direkt aufzurufen. OLE löst dieses Problem auf eine raffinierte Weise. Wenn Sie ein Interface zu einem Objekt eines anderen Prozesses anfordern, legt OLE ein Proxy-Objekt im Prozeßraum Ihrer Anwendung an. Dieses »getürkte« Objekt verfügt über das gleiche Interface wie das eigentliche Objekt. Wenn Sie nun eine Funktion des Proxy-Objekts aufrufen, sammelt Windows alle Parameter der Funktion ein und kopiert diese über die Interprozeß-Fähigkeiten des Betriebssystems in den Prozeß, der die Implementierung des Objekts samt dessen Daten enthält. Anschließend ruft Windows die entsprechende Funktion des echten Interface des Objekts auf. Dieser Prozeß, der *Marshaling* (etwa »Verschieben«) genannt wird, ist in Abbildung 6.3 dargestellt.

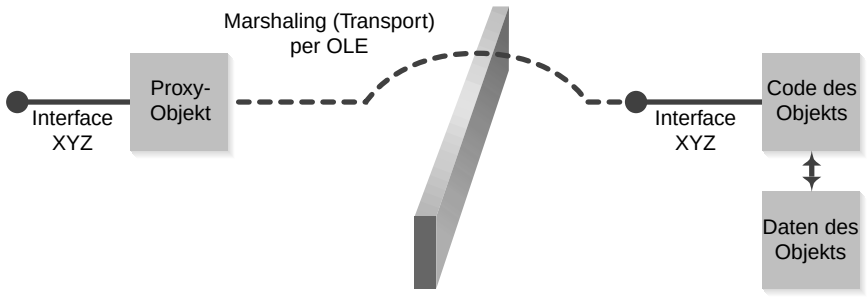

Abb. 6.3: Wie Objekt Prozeßgrenzen überqueren können

Woher weiß Windows genügend über ein Objekt, um dieses Proxy-Objekt anlegen zu können? Letztlich wird dazu eine Liste aller Funktionen eines jeden Interface des Objekts benötigt, zusammen mit allen Parametern jeder Funktion. Je nach verwendeter Programmiersprache gibt es eine Reihe weiterer Antworten auf diese Frage. Am besten begnügen Sie sich mit der Aussage, daß dies exakt die Art der Informationen ist, für deren Lieferung Typbibliotheken entworfen wurden. Und Visual Basic legt automatisch Typbibliotheken für Ihre Visual-Basic-Klassen an, sobald diese angelegt werden. Unter C oder C++ kann eigener Marshaling-Code entwickelt werden. Doch als Visual-Basic-Programmierer werden sie damit wohl nie in Berührung kommen.

Das Konzept des Marshaling kann noch ausgeweitet werden. Denn wenn es erst einmal möglich ist, Funktionsaufrufe über Prozeßgrenzen hinweg zu transportieren – um wieviel größer wäre der Aufwand, die Aufrufe über ein Netzwerk zu einem anderen System zu übertragen?

Dies konfrontiert uns mit einer Reihe von Szenarien, in denen ein COM-Objekt existieren kann.

- *Innerhalb Ihrer eigenen Anwendung* – die Interfaces des Objekts sind in Ihrer ausführbaren Datei implementiert (in der Regel als Klassen-Module). Die Daten werden von der Anwendung allokiert und verwaltet.
- *Innerhalb einer ActiveX-DLL* – die Interfaces des Objekts sind in der DLL implementiert. Die Daten werden von der DLL allokiert und verwaltet. Da die DLL in den Prozeßraum der Anwendung geladen wird, entfällt die Notwendigkeit von Marshaling.
- *Innerhalb einer ActiveX-EXE* – die Interfaces des Objekts sind in der ausführbaren Datei implementiert. Die Daten werden von der Anwendung der ausführbaren Datei allokiert und verwaltet. Windows legt ein Proxy-Objekt immer dann an, wenn der Zugriff auf das Objekt gewünscht wird. Alle Funktionsaufrufe werden in den anderen Prozeß transportiert, was einen gewissen Overhead mit sich bringt.

- *Auf einem Remote-System* – die Interfaces des Objekts sind in der DLL oder der ausführbaren Datei implementiert, die sich auf dem anderen System befindet. Die Daten befinden sich ebenfalls dort. Distributed COM (DCOM) oder Remote-Automation legt das benötigte Proxy-Objekt auf Ihrem System an und sorgt für den Transport der Daten über das Netzwerk. Es fällt nicht nur der Overhead des Marshaling-Vorgangs zwischen den Prozessen an. Es kommt auch noch der Aufwand für den Transport zwischen den Systemen hinzu. Der Hauptvorteil der Remote-Automation kommt zum Tragen, wenn das Remote-System bereits über die Daten verfügt, mit denen Sie arbeiten wollen (etwa in der Form einer großen Datenbank). Der Overhead des Transports einiger weniger Funktionsaufrufe kann unter Umständen gegenüber der Alternative des megabyteweisen Transfers von Daten über das Netzwerk vernachlässigbar gering sein.

Vielleicht ist Ihnen aufgefallen, daß ich ActiveX-Controls und ActiveX-Dokumente nicht aufgezählt habe. Das liegt daran, daß beide lediglich spezialisierte Varianten der oben angeführten Szenarien darstellen. Wenn Sie diese Szenarien erst einmal verstanden haben, dürfte es ein leichtes für Sie sein, Ihr Wissen auf diese spezialisierten ActiveX-Komponenten zu übertragen.

So weit die Theorie. Schauen wir uns einige dieser Szenarien in der Praxis an.

6.3 Der Lebenszyklus eines DLL-Objekts

Meine Ansicht, daß ActiveX-Technologie nahezu jeden Aspekt der Programmierung mit Visual Basic berührt und ActiveX-Komponenten dasselbe wie COM-Objekte sind, egal ob sie innerhalb Ihrer eigenen Anwendung oder ob sie als DLL-Objekte implementiert sind, durchzieht als roter Faden das ganze Buch. Ich finde dies deswegen so wichtig, weil alle diese Objekte im wesentlichen dasselbe sind. Wenn Sie einmal verstanden haben, mit einem zu arbeiten, wissen Sie auch, wie Sie mit jedem anderen zu arbeiten haben. Das impliziert, daß der beste Weg, ActiveX beherrschen zu lernen, nicht über die Beschäftigung mit den einzelnen Typen an ActiveX-Komponenten führt, sondern zunächst über das generelle Verständnis der grundlegenden COM-Technologie. Danach wird es dann relativ einfach, sich mit den besonderen Eigenschaften der einzelnen Komponententypen zu beschäftigen. So beginnen wir die Erforschung eines DLL-basierten Objekts damit, eine bereits existierende Klasse in ein DLL-Objekt zu verwandeln, so daß es von anderen Anwendungen leicht genutzt werden kann, anstatt eine Klasse von Grund auf neu zu entwickeln.

Der naheliegende Kandidat hierfür ist die `clsBankLoan`-Klasse aus Kapitel 5, »Aggregation und Polymorphie«. Das gilt nicht nur deswegen, weil diese Klasse direkt von der Anwendung verwendet wird, sondern auch, weil sie ebenfalls die Basis der Klassen `clsSecurityLoan` und `clsLoanShark` liefert. Die Schritte, diese Klasse in eine eigenständige DLL zu verwandeln, sind einfach:

1. Kopieren Sie das Projekt Loan5 in einen neuen Ordner, und benennen Sie die Dateien gegebenenfalls um. (Sie finden den modifizierten Code unter dem Namen Loan6 im Ordner `samples/ch06` auf der Buch-CD.)
2. Legen Sie ein neues ActiveX-DLL-Projekt namens `gtpBankLoan` an (»`gtp`« steht hier für *Guide to the Perplexed*, dem Untertitel der englischsprachigen Originalausgabe).
3. Fügen Sie die existierende Klasse `clsBankLoan` diesem Projekt hinzu.
4. Löschen Sie die von Visual Basic standardmäßig angelegte Klasse `Class1`.
5. Markieren Sie die Klasse `clsBankLoan` im Projekt-Fenster, und setzen Sie im Eigenschaften-Fenster die INSTANCING-Eigenschaft auf 5-MULTIUSE (dies weist Visual Basic an, das Objekt öffentlich zu machen).
6. Fügen Sie zum Testen das ursprüngliche Loan-Projekt (das nun Loan6 heißt) hinzu.
7. Entfernen Sie aus diesem Loan-Projekt die Klasse `clsBankLoan`. (Die Möglichkeit, mehrere Projekte in eine Instanz der Entwicklungsumgebung zu laden, war in Visual Basic 5.0 neu hinzugekommen. Sollte Ihnen der Vorgang nicht klar sein, sollten Sie in der Dokumentation die das DATEI-Menü betreffenden Kapitel lesen.)
8. Fügen Sie der Loan6-Anwendung über das Menü PROJEKT/VERWEISE einen Verweis auf `gtpBankLoan` hinzu.
9. Klicken Sie mit der rechten Maustaste im Projekt-Fenster auf das Loan6-Projekt, und wählen Sie im Kontextmenü ALS STARTEINSTELLUNG FESTLEGEN.
10. Starten Sie nun das Loan6-Programm. Es funktioniert nach wie vor, abgesehen davon, daß nun das vom GTPBANKLOAN-Projekt offengelegte `clsBankLoan`-Objekt verwendet wird. Sie sehen, daß keine einzige Zeile Code geändert werden mußte.
11. Fahren wir nun fort, indem wir das Programm als DLL kompilieren. Klicken Sie im Projekt-Fenster mit der rechten Maustaste auf das GTPBANKLOAN-Projekt.
12. Wählen Sie im Kontextmenü EIGENSCHAFTEN VON GTPBANKLOAN.
13. Im ALLGEMEIN-Register fügen Sie eine Projektbeschreibung ein.
14. Im Register ERSTELLEN setzen Sie die CheckBox AUTOMATISCH ERHÖHEN (was Sie immer automatisch machen sollten), und fügen Sie nach Bedarf Copyright-Informationen ein.

15. Im Register KOMPILIEREN wählen Sie die Einstellung KOMPILIEREN ZU SYSTEM-CODE *(NATIVE CODE)*. (Wir werden in Kürze ein paar Performance-Tests durchführen – mit der Wahl von Native Code stellen Sie vorläufig die Entscheidung nativer gegen P-Code hintenan.)
16. Wählen Sie im DATEI-Menü GTPBANKLOAN ERSTELLEN – die DLL wird kompiliert.
17. Öffnen Sie nun das Projekt `Loan6.vbp` alleine und als einziges Projekt. Wenn Sie sich nun die Projekt-Verweise ansehen, werden Sie feststellen, daß der Verweis auf das Objekt `clsBankLoan` von der DLL und nicht mehr von deren Visual-Basic-Projekt implementiert wird.

Auch jetzt werden Sie feststellen, daß das Loan6-Projekt nach wie vor funktioniert.

Kümmern Sie sich vorläufig nicht um die vielen Einstellmöglichkeiten in den Projekt-Eigenschaften – damit werden wir uns im weiteren Verlauf des Buches noch ausführlicher beschäftigen.

Um das Loan6-Beispiel starten zu können, muß sichergestellt sein, daß die Objekte registriert sind. Dieser Vorgang erfordert zwei Operationen:

- Registrieren Sie die Objekte in `gtpbkln.DLL` über den Kommandozeilenbefehl `regsvr32 gtpbkln.DLL`.
- Registrieren Sie die Objekte in `gtpPln.EXE`, indem Sie das Programm starten. (Sie werden über diese Objekte im weiteren Verlauf des Kapitels mehr erfahren.) Sie werden sehen, daß nichts passiert, jedoch wird das Programm registriert.

6.3.1 Ein Blick hinter die Kulissen

Woher weiß das Loan6-Projekt, daß das `clsBankLoan`-Objekt von der DLL `gtpbkln.DLL` implementiert wird? Woher weiß überhaupt eine beliebige Anwendung, welche DLL oder EXE den Code eines Objekts implementiert? Die Antwort ist in der System-Registrierung zu finden.

Eines der Tools, die mit dem Visual Studio 6.0 geliefert werden, heißt OLE View. OLE View ist ein hervorragend dafür geeignetes Werkzeug, hinter die Kulissen der System-Registrierung zu schauen. Es erleichtert auch den Start des Registrierungseditors `regedt32.EXE` (`regedit.EXE` unter Windows 95/98), der sich im System-Ordner (bzw. im Windows-Ordner) des Windows-Systems befindet. Seien Sie vorsichtig, wenn Sie den Registrierungseditor verwenden – eine unachtsame Änderung der Registrierung kann Ihr System beschädigen, unter Umständen so weit, daß es nicht mehr bootet. Der Registrierungseditor bietet die Möglichkeit, einen Eintrag der Registrierung in eine Text-Datei zu exportieren.

Anmerkung

Die Verzeichniseinträge, Zeiten und tatsächlichen GUIDs oder andere CLSID-Werte werden auf Ihrem System von den hier gezeigten abweichen. Die hier gezeigten Werte dienen nur der Illustration.

Wenn Sie den Schlüssel `HKEY_CLASSES_ROOT` der Registrierung öffnen, werden Sie, wenn alles gutgegangen ist, einen Eintrag wie etwa den folgenden für die `clsBankLoan`-Klasse finden:

```
[HKEY_CLASSES_ROOT\gtpBankLoan.clsBankLoan]
@=gtpBankLoan.clsBankLoan

[HKEY_CLASSES_ROOT\gtpBankLoan.clsBankLoan\Clsid]
@={B7989D4A-8AF3-11D0-93CD-00AA0036005A}
```

Die System-Registrierung ist als hierarchische Gliederung aufgebaut. Jeder Ebene der Hierarchie ist ein Name zugewiesen, der *Schlüssel* (Key) genannt wird – ähnlich einem Ordner-Namen auf der Festplatte. Jeder kann einen unbenannten Standardwert haben und beliebig viele weitere benannte Werte enthalten. Wenn Sie einen Eintrag in der Registrierung aus dem Registrierungseditor ausdrucken, stellt das Symbol `@` den Standardwert eines Schlüssels dar. Der Schlüssel `gtpBankLoan.clsBankLoan` hat als Standardwert einen String, der den Namen des Objekts enthält. Er hat einen Unterschlüssel, der `CLSID` heißt. Wie Sie sich erinnern werden, ist `CLSID` die Abkürzung für *Class ID*, also des Klassen-Identifizierers. In diesem Fall enthält `CLSID` die String-Repräsentation der 16-Byte-Zahl, die das Objekt eindeutig identifiziert. Wenn nun eine Anwendung ein Objekt über seinen Namen anfordert – wenn Sie beispielsweise die Funktion `CreateObject` aufrufen –, ist dies die Stelle, an der Windows die Kennung des Objekts finden kann.

Nachdem die GUID vorhanden ist, ist eine ausführbare Datei ausfindig zu machen, die den Code zur Implementierung der Interfaces des Objekts enthält und weiß, wie dieses anzulegen, also ins Leben zu rufen ist. Den Hinweis auf die auszuführende Datei finden Sie unter dem Schlüssel `HKEY_CLASSES_ROOT\CLSID` in der Registrierung:

```
[HKEY_CLASSES_ROOT\CLSID\{B7989D4A-8AF3-11D0-93CD-00AA0036005A}]
@="gtpBankLoan.clsBankLoan"

[HKEY_CLASSES_ROOT\CLSID\{B7989D4A-8AF3-11D0-93CD-00AA0036005A}\Imple-
mented Categories]

[HKEY_CLASSES_ROOT\CLSID\{B7989D4A-8AF3-11D0-93CD-00AA0036005A}\Imple-
mented Categories\_
         {40FC6ED5-2438-
11CF-A3DB-080036F12502}]

[HKEY_CLASSES_ROOT\CLSID\{B7989D4A-8AF3-11D0-93CD-
```

```
00AA0036005A}\InprocServer32]
@="D:\\ZDBOOK4\\NewSamples\\CH06\\gtpbkln.dll"

[HKEY_CLASSES_ROOT\CLSID\{B7989D4A-8AF3-11D0-93CD-00AA0036005A}\ProgID]
@="gtpBankLoan.clsBankLoan"

[HKEY_CLASSES_ROOT\CLSID\{B7989D4A-8AF3-11D0-93CD-00AA0036005A}\Pro-
grammable]

[HKEY_CLASSES_ROOT\CLSID\{B7989D4A-8AF3-11D0-93CD-00AA0036005A}\TypeLib]
@="{CBBEF1B4-2DEE-11D0-92E4-00AA0036005A}"

[HKEY_CLASSES_ROOT\CLSID\{B7989D4A-8AF3-11D0-93CD-00AA0036005A}\VERSION]
@="4.0"
```

Gehen wir die einzelnen Einträge einmal der Reihe nach durch. Der Standardwert des `Schlüssels ...\CLSID\{CBBEF242-2DEE-11D0-92E4-00AA0036005A}` ist der Name des Objekts: `gtpBankLoan.clsBankLoan`. Unter diesem Schlüssel sind drei weitere Unterschlüssel angeordnet (siehe Tabelle 6.1).

Unterschlüssel	Beschreibung
`Implemented Categories`	Enthält einen oder mehrere Unterschlüssel, die dem GUID-Wert von Schlüsseln unter `HKEY_CLASSES_ROOT\Component Categories` entsprechen. Jeder Eintrag unter diesem Schlüssel beschreibt einen Aspekt des Objekts. In diesem Fall wird ausgewiesen, daß das Objekt zur Kategorie der »Automation Objekte« gehört. Als Visual-Basic-Programmierer brauchen Sie sich wahrscheinlich nie um diesen Eintrag zu kümmern
`InProcServer32`	Dieser Eintrag teilt dem System mit, welche DLL den Code enthält, um das Objekt im Prozeß zu implementieren (im gleichen Prozeß wie die Anwendung, die versucht, das Objekt zu verwenden).
`ProgID`	Der »programmatische Identifizierer« des Objekts. Es ist der Name, den Sie in einem Programm zur Identifizierung eines Objekts verwenden. In diesem Fall lautet dieser: `gtpBankLoan.clsBankLoan`.
`Programmable`	Ein anderer Weg der Anzeige, daß dieses Objekt programmierbar ist (über ActiveX-Automation).
`TypeLib`	Dieser Schlüssel enthält die GUID der Typbibliothek, die die von dem Objekt unterstützten Interfaces beschreibt.
`Version`	Die Version des Objekts (in diesem Fall 4.0). Auf die Versionsverwaltung und -kontrolle wird im weiteren Verlauf des Buches noch näher eingegangen.

Tab. 6.1: Objekt-Unterschlüssel

Die wichtigste Information steckt hier im Schlüssel `InProcServer32`. Nun sehen Sie, wie eine Anwendung die DLL finden kann, die ein Objekt implementiert. Wird ein Objekt statt dessen von einer ActiveX-EXE implementiert, lautet der dementsprechende Schlüssel `LocalServer32`.

Sie können den größten Teil dieser Informationen auf einem etwas einfacheren Weg in Erfahrung bringen, indem Sie den OLE Viewer verwenden und dort nach dem Objekt `gtpBankLoan.clsBankLoan` suchen.

Der Zweig eines weiteren Schlüssels sollte einmal verfolgt werden: `TypeLib`. Eine Typbibliothek können Sie sich ebenfalls mit dem OLE Viewer ansehen. (Die Schlüssel unter `HKEY_CLASSES_ROOT\TypeLib` referenzieren letztlich auch nichts anderes, als die DLL, die die Typbibliothek enthält.) Sie können sich das Objekt in der Liste der Typbibliotheken (Type Libraries) ansehen oder über das DATEI-Menü (*View Type Library*) die Typbibliothek direkt aus der DLL laden. (Auch wenn die voreingestellten Dateierweiterungen `.TLB` und `.OLB` lauten, können Sie dennoch auch DLLs laden.) OLE View kann die Informationen der Typbibliothek im Format der *Interface Description Language* sichern, in einer IDL-Datei. Eine IDL-Datei kann mit Hilfe des mktyplib-Programms (wird mit Visual Basic bzw. dem Visual Studio geliefert) in eine Typbibliothek zurückverwandelt werden. Hier sehen Sie die Zusammenfassung der Typbibliothek der `clsBankLoan`-Klasse:

```
// Generated .IDL file (by the OLE/COM Object Viewer)
//
// typelib filename: gtpbkln.dll

[
  uuid(CBBEF1B4-2DEE-11D0-92E4-00AA0036005A),
  version(4.0),
  helpstring("gtpBankLoan: Chapter 6 sample object for Guide To The Perplexed.")
]
library gtpBankLoan
{
    // TLib :     // TLib : OLE Automation : {00020430-0000-0000-C000-
000000000046}
    importlib("STDOLE2.TLB");

    // Forward declare all types defined in this typelib
    interface _clsBankLoan;

    [
      odl,
      uuid(C8BC6848-F022-11D1-BA59-00001C1B4A69),
      version(1.0),
      hidden,
      dual,
      nonextensible,
```

```
    oleautomation
  ]
  interface _clsBankLoan : IDispatch {
      [id(0x40030000), propget]
      HRESULT AmountAvailable([out, retval] CURRENCY* AmountAvailable);
      [id(0x40030000), propput]
      HRESULT AmountAvailable([in] CURRENCY AmountAvailable);
      [id(0x40030001), propget]
      HRESULT Duration([out, retval] short* Duration);
      [id(0x40030001), propput]
      HRESULT Duration([in] short Duration);
      [id(0x40030002), propget]
      HRESULT Interest([out, retval] double* Interest);
      [id(0x40030002), propput]
      HRESULT Interest([in] double Interest);
      [id(0x60030000)]
      HRESULT Payment([out, retval] CURRENCY* );
      [id(0x60030001)]
      HRESULT Summary([out, retval] BSTR* );
      [id(0x60030002)]
      HRESULT SourceType([out, retval] BSTR* );
      [id(0x60030003)]
      HRESULT MainInterface([out, retval] IDispatch** );
  };

  [
    uuid(B7989D4A-8AF3-11D0-93CD-00AA0036005A),
    version(1.0)
  ]
  coclass clsBankLoan {
      [default] interface _clsBankLoan;
  };
};
```

Das programmierte Interface für dieses Objekt heißt `_clsBankLoan`. Beachten Sie die Zeile, mit der die Definition des Interface beginnt:

```
interface _clsBankLoan : IDispatch {
```

Der Doppelpunkt (`:`) weist darauf hin, daß das Interface von dem dahinter folgenden Interface abgeleitet ist, in diesem Fall von `IDispatch`. Dies bedeutet, daß die ersten sieben Funktionen des Interface den Funktionen von `IUnknown` und `IDispatch` entsprechen. Das ist genau das, was Sie von einem dualen Interface erwarten können. Sie sehen auch, daß die Eigenschaften, die in der Klasse als öffentliche Variablen deklariert worden waren, tatsächlich als Funktionspaar auftreten: die eine Funktion mit dem Attribut `propget`, die andere mit dem Attribut `propput` – entsprechend den `Property Get`- und `Property Let`-Funktionen in einer Klasse. Der OLE Viewer liefert noch mehr Informationen zu jeder Funktion. Vie-

les davon wird jedoch nur OLE-Gurus etwas sagen und ist auch nur für den jeweiligen Container von Bedeutung, wie etwa für Visual Basic oder andere Objekt-Container.

Die Typbibliothek enthält ebenfalls einen Verweis auf die Klassen-GUID (dem Co-Class-Typ am Ende der Liste). Dies erlaubt es Visual Basic, von der Typbibliothek wieder auf die CLSID des Objekts zu schließen.

Vor diesem Hintergrund fällt es nun leicht, den Lebenszyklus eines Objekts zu verfolgen, das in einer DLL implementiert ist.

Die Registrierung

Bevor irgendeine Anwendung (wie auch Loan6) auf das Objekt `gtpBankLoan.clsBankLoan` zugreifen kann, müssen die Informationen über das Objekt in der System-Registrierung abgelegt sein. Visual Basic erledigt dies für Sie auf Ihrem System bereits beim Kompilieren. Wenn Sie Anwendungen vertreiben wollen, werden die verschiedenen Komponenten in der Regel vom Installationsprogramm oder mit Hilfe des Programms `Regsvr32.EXE` registriert.

Visual-Basic-DLLs exportieren die Funktionen `DllRegisterServer` und `DllUnregisterServer`, die zum Registrieren bzw. zum Löschen der Registrierung in der betreffenden DLL enthaltener Objekte aufgerufen werden können. Diese Funktionen werden erforderlichenfalls von `Regsvr32.EXE` oder dem Installationsprogramm aufgerufen.

Zur Design-Zeit

Zum Loan6-Projekt wird ein Verweis auf das `gtpBankLoan.clsBankLoan`-Objekt hinzugefügt:

- Visual Basic kann über die Programm-ID (`ProgID`) `gtpBankLoan.clsBankLoan` in der Registrierung die `CLSID` des Objekts ausfindig machen.
- Visual Basic schaut in der Registrierung nach dem Server für das Objekt. Gefunden wird `gtpbkln.DLL`.
- Visual Basic liest die Typbibliothek der DLL. Nun sind alle Informationen über das Interface des Objekts bekannt.
- Wenn Sie das Projekt sichern, wird der TypeLib-Identifizierer in der Projekt-Datei abgelegt. Wenn Visual Basic das Projekt wieder lädt, kann zuerst die Typbibliothek gelesen werden und daraus die `CLSID` des Objekts gewonnen werden.

Während der Kompilierung

Dieses Stadium wird auch beim Start einer Anwendung innerhalb der Visual-Basic-Entwicklungsumgebung durchlaufen. Da die Informationen über das Interface bereits zur Design-Zeit bekannt sind, können alle Zugriffe über Objektvariablen, die als Objekttyp `clsBankLoan` deklariert sind, früh gebunden werden.

Visual Basic kann direkte Aufrufe der Funktionen des Interface kompilieren, statt über das `IDispatch`-Interface des Objekts zu gehen.

Zur Laufzeit

Wenn Loan6 ein `clsBankLoan`-Objekt anzulegen versucht, wird die CLSID dazu verwendet, in der Registrierung den Namen der DLL, die das Objekt implementiert, ausfindig zu machen. Daraufhin geschieht folgendes:

- Visual Basic lädt die DLL.
- Visual Basic verwendet Funktionen des OLE-Systems, um die Objekte bei Bedarf anzulegen. Wenn OLE ein Objekt anlegt, wird ein Zeiger auf das `IUnknown`-Interface des Objekts zurückgegeben. Visual Basic erfragt nun über QueryInterface das `_clsBankLoan`-Interface, das einer `clsBankLoan`-Variablen zugewiesen werden kann. Für das Objekt wird der Referenzzähler beim Anlegen auf 1 gesetzt.
- Aufrufe von Methoden und Eigenschaften des Objekts werden ausgeführt, indem die Funktionen des Interface direkt aufgerufen werden (frühe Bindung).
- Wenn das Projekt beendet wird oder die Objektvariable auf `Nothing` gesetzt wird, wird der Referenzzähler auf 0 zurückgesetzt. Die DLL `gtpbkln.DLL` erkennt, daß der Wert des Referenzzählers 0 beträgt und gibt den vom Objekt belegten Speicher frei. Sind alle Objekte, die von der DLL implementiert werden, freigegeben, kann die DLL entladen werden.

6.4 Der Lebenszyklus eines EXE-Objekts

Das im vorangegangenen Abschnitt beschriebene DLL-Objekt zeichnet sich durch zwei Merkmale aus: Es ist in einer DLL implementiert, und es läuft im Prozeß einer Anwendung (»InProcess« oder »InProc«).

Was geschieht, wenn ein Objekt in einer ausführbaren Datei implementiert ist, die in einem anderen Prozeß läuft? Um dies zu demonstrieren, wollen wir der Loan6-Anwendung einen weiteren Objekttyp hinzufügen.

Die Zinssätze sind gestiegen, und mehr und mehr Leute kaufen Häuser mit Hilfe von Darlehen ihrer Eltern. Das Loan6-Beispiel soll also erweitert werden, um elterliche Darlehen bearbeiten zu können. Eltern-Darlehen sind in einer Klasse in einem EXE-Server namens `gtpPln.vbp` implementiert. Dieses Projekt enthält ein einziges Klassenmodul namens `clsParentLoan` (`ParLn.cls`), das Sie in Listing 6.1 sehen.

```
' ActiveX: Guide to the Perplexed
' Copyright (c) 1997 by Desaware Inc.  All Rights Reserved
' Kapitel 6

    Implements clsBankLoan
```

```
Option Explicit

' Darlehensbetrag
Public AmountAvailable As Currency

' Laufzeit
Public Duration As Integer

' Zinssatz
Public Interest As Double

' ***Margin requirement
Public Margin As Double

' Berechnung der Raten
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

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

Public Function SourceType() As String
   SourceType = "Loan from parents"
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)
   If RHS > 0.02 Then
      RHS = RHS / 10
   End If
   Interest = RHS
End Property

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

' Referenz auf das andere Interface erhalten
Private Function clsBankLoan_MainInterface() As Object
   Dim myobj As clsParentLoan
   Set myobj = Me  'Die richtige Schnittstelle ermitteln
   Set clsBankLoan_MainInterface = myobj
End Function

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

Public Function LatePenalty() As String
   LatePenalty = "Don't worry about us, we'll be fine"
End Function
```

Listing 6.1: Listing der Klasse clsParentLoan

Diese Klasse basiert auf der Klasse `clsSecurityLoan` der Loan6-Anwendung. Wie Sie sehen können, wird gleichfalls das `clsBankLoan`-Interface implementiert. Damit dies funktioniert, enthält das Projekt auch einen Verweis auf die Klasse `gtpBankLoan.clsBankLoan`. Der Verweis würde über das Menü PROJEKT/VERWEISE hinzugefügt.

Könnte die `clsParentLoan`-Klasse auf die gleiche Weise wie die `clsLoanShark`-Klasse Aggregation verwenden? Selbstverständlich!

Wiederum tritt hier etwas besonders Wichtiges zu tage. Die Wahl, ob ein Objekt in einer Klasse, in einem DLL- oder in einem EXE-Server implementiert wird, kann von vielen Faktoren abhängen, wie etwa von der logischen Aufteilung der Funktionalität, von Performance-Fragen, von Verbreitungsfragen und vielem mehr (worauf wir später noch zu sprechen kommen werden). Nachdem Sie jedoch erst einmal den Entschluß gefaßt haben, wo und wie ein Objekt implementiert werden soll, ist der Code selbst in jedem Fall im wesentlichen der gleiche.

Ein Objekt in einem EXE-Server kann auf verschiedene Weise registriert werden. Es wird automatisch registriert, wenn das Projekt kompiliert wird. Es wird registriert, wenn die ausführbare EXE-Datei gestartet wird bzw. wenn diese mit der Kommandozeilen-Option `/RegServer` gestartet wird. Die Aufhebung der Registrierung erfolgt, wenn die ausführbare EXE-Datei mit der Kommandozeilen-Option `/UnRegServer` gestartet wird. Sie können ebenfalls das Programm `CliReg32.EXE` verwenden, das mit der Enterprise-Version von Visual Basic geliefert wird, um eine VBR-Datei zu erzeugen und zu registrieren, damit die Komponente per Remote-Automation eingesetzt werden kann. Die VBR-Datei enthält ebenfalls Typbibliothek-Informationen sowie alle notwendigen Informationen, damit das Objekt per Remote-Automation verwendet werden kann. VBR-Dateien werden ebenfalls bereits beim Kompilieren auf Wunsch angelegt, wenn die entsprechende Option in den Projekt-Eigenschaften gesetzt wird.

Die Schlüssel in der Registrierung für einen EXE-Server sehen denen für einen DLL-Server ähnlich. Der wesentlichste Unterschied ist, daß der Dateiname des Servers unter dem Unterschlüssel `\LoacalServer32` anstelle `\InProcServer32` abgelegt ist. Dies zeigt an, daß der Server, der das Objekt implementiert, auf dem lokalen System läuft, aber eben nicht im gleichen Prozeß (als sogenannter »OutofProcess«-Server). Die Einträge in der Registrierung sehen so aus:

```
[HKEY_CLASSES_ROOT\CLSID\{B7989D6C-8AF3-11D0-93CD-00AA0036005A}]
@="gtpParentLoan.clsParentLoan"
"AppID"="{B7989D6C-8AF3-11D0-93CD-00AA0036005A}"

[HKEY_CLASSES_ROOT\CLSID\{B7989D6C-8AF3-11D0-93CD-00AA0036005A}\Imple-
mented Categories]

[HKEY_CLASSES_ROOT\CLSID\{B7989D6C-8AF3-11D0-93CD-00AA0036005A}\Imple-
mented Categories\ _
            {40FC6ED5-2438-11CF-A3DB-080036F12502}]
```

```
 [HKEY_CLASSES_ROOT\CLSID\{B7989D6C-8AF3-11D0-93CD-
00AA0036005A}\LocalServer32]
@="D:\\ZDBOOK4\\NewSamples\\CH06\\gtppln.exe"

[HKEY_CLASSES_ROOT\CLSID\{B7989D6C-8AF3-11D0-93CD-00AA0036005A}\ProgID]
@="gtpParentLoan.clsParentLoan"

[HKEY_CLASSES_ROOT\CLSID\{B7989D6C-8AF3-11D0-93CD-00AA0036005A}\Pro-
grammable]

[HKEY_CLASSES_ROOT\CLSID\{B7989D6C-8AF3-11D0-93CD-00AA0036005A}\TypeLib]
@="{2B8BE8EA-2E0D-11D0-92E4-00AA0036005A}"

[HKEY_CLASSES_ROOT\CLSID\{B7989D6C-8AF3-11D0-93CD-00AA0036005A}\VERSION]
@="3.0"
```

Die Typbibliothek für ein in einem EXE-Server implementiertes Objekt ist im wesentlichen dieselbe wie für ein in einem DLL-Server implementiertes Objekt.

Schauen wir uns nun den Lebenszyklus eines EXE-Objekts an.

Die Registrierung

Die Registrierung erfolgt auf die gleiche Weise wie bei einer DLL, nur daß eine EXE auf die weiter oben beschriebenen Weisen registriert wird.

Zur Design-Zeit

Ein Verweis auf das `gtpParentLoan.clsParentLoan`-Objekt wird zum Loan6-Projekt hinzugefügt. Folgendes geschieht:

- Visual Basic verwendet die `ProgID` (`gtpParentLoan.clsParentLoan`), um in der Registrierung die `CLSID` des Objekts ausfindig zu machen.
- Visual Basic schaut in der Registrierung nach dem Server für das Objekt. Gefunden wird `gtpPln.EXE`.
- Visual Basic liest die Typbibliothek in der EXE-Datei. Nun sind alle Informationen über das Interface des Objekts bekannt.
- Wie auch bei der DLL-Version legt Visual Basic die TypeLib-Kennung in der Projekt-Datei ab, so daß die Verweis-Informationen über die Typbibliothek ermittelt werden können, wenn das Projekt wieder geladen wird.

Während der Kompilierung

Dieses Stadium wird auch beim Start einer Anwendung innerhalb der Visual-Basic-Entwicklungsumgebung durchlaufen. Da die Informationen über das Interface bereits zur Design-Zeit bekannt sind, können alle Zugriffe über Objektvariablen, die als Objekttyp `clsParentLoan` deklariert sind, früh gebunden werden.

Visual Basic kann direkte Aufrufe der Funktionen des Interface kompilieren, statt über das `IDispatch`-Interface des Objekts zu gehen.

Zur Laufzeit

Wenn Loan6 ein `clsParentLoan`-Objekt anzulegen versucht, wird die CLSID dazu verwendet, in der Registrierung den Namen der EXE, die das Objekt implementiert, ausfindig zu machen. Daraufhin geschieht folgendes:

- Visual Basic startet die ausführbare Datei `gtpPlan.EXE`.
- Visual Basic verwendet Funktionen des OLE-Systems, um die Objekte bei Bedarf anzulegen. Wenn das OLE-System ein Out-of-Process-Objekt anlegt, wird das Objekt zuerst im Prozeßraum der Server-Anwendung `gtpPln.EXE` angelegt. Dann wird ein Proxy-Objekt im Prozeßraum der Loan6-Anwendung angelegt und ein Zeiger auf das `IUnknown`-Interface des Proxy-Objekts an Visual Basic zurückgegeben. Das Proxy-Objekt kann die Interfaces des originalen, realen Objekts nachbilden. Visual Basic kann nun das Interface `_clsParentLoan` über `QueryInterface` von dem Proxy-Objekt erhalten und einer als `clsBankLoan` deklarierten Objektvariablen zuweisen. Für das Objekt wird der Referenzzähler beim Anlegen auf 1 gesetzt.
- Aufrufe von Methoden und Eigenschaften des Objekts werden ausgeführt, indem die Funktionen im Interface des Proxy-Objekts aufgerufen werden. Das Proxy-Objekt reicht die Funktionsaufrufe und Parameter weiter, und OLE sendet diese Informationen an das reale Objekt im Prozeßraum von `gtpPln.EXE`. Wenn die Funktion einen Rückgabewert hat oder einer der Parameter als Referenz (implizit oder als `ByRef` deklariert) übergeben worden ist, transportiert Windows diese Ergebniswerte zurück in den Prozeßraum von Loan6, so daß sie an die aufrufende Anwendung zurückgegeben werden können.
- Wenn das Projekt beendet wird oder die Objekt-Variable auf `Nothing` gesetzt wird, wird der Referenzzähler auf 0 zurückgesetzt. Das Programm `gtpPln.exe` erkennt, daß der Wert des Referenzzählers 0 beträgt, und gibt den vom Objekt belegten Speicher frei. Sind alle Objekte, die von der EXE implementiert werden, freigegeben, wird die EXE-Anwendung beendet.

6.5 Der Lebenszyklus eines Remote-Objekts

Bereits weiter oben in diesem Kapitel habe ich die etwas außergewöhnlich klingende Behauptung aufgestellt, daß es möglich sein müßte, wenn man Funktionsaufrufe über Prozeßgrenzen hinweg transportieren könnte, ohne weiteren großen Aufwand diese Funktionsaufrufe auch über ein Netzwerk zu einem anderen System zu transportieren.

Um dies auszuprobieren, nehmen Sie das Programm `gtpPln.EXE`, und kopieren Sie es auf ein anderes System in Ihrem Netzwerk. Starten Sie das Programm dort,

damit es auf diesem System registriert wird. Öffnen Sie nun den VERBINDUNGSMANAGER FÜR REMOTE-AUTOMATISIERUNG auf Ihrem lokalen System, und markieren Sie das Objekt `gtpParentLoan.clsParentLoan` in der Liste der COM-Klassen. Wählen Sie im Menü REGISTRIEREN/REMOTE, und markieren Sie dann auf der Registerkarte SERVER-VERBINDUNG die CheckBox DISTRIBUTED COM. Geben Sie nun den Namen des Remote-Servers ein.

Schauen Sie nun in der Registrierung nach den Eintragungen für dieses Objekt – Sie werden eine kleine, unscheinbare Änderung feststellen:

```
[HKEY_CLASSES_ROOT\CLSID\{B7989D6C-8AF3-11D0-93CD-
00AA0036005A}\_LocalServer32]
@="C:\\ZDBook4\\CH06\\gtpPLn.exe"
```

Erkennen Sie es? Der Bezeichnung `LocalServer32` wurde ein Unterstrich vorangestellt. Dieses Zeichen sorgt dafür, daß der Ort des Servers verborgen wird – der ursprüngliche Schlüsselname `LocalServer32` wird nun nicht mehr gefunden. Aber was unternimmt das COM-System, wenn es für ein Objekt keinen Server unter der `CLSID` ausfindig machen kann? Es schaut an einer anderen Stelle in der Registrierung nach, an der eine Liste der Remote-Server abgelegt ist – diese Liste wird vom Verbindungsmanager für Remote-Automatisierung angelegt. In dieser Liste kann nun der von Ihnen eingegebene Server-Name für das Objekt gefunden werden.

Die Registrierung

Der eigentliche Trick der Funktionsweise von DCOM oder Remote-Automation ist vom Konzept her einfach, kann aber in der Praxis recht komplex werden. Wichtig ist, daß die Komponente auf beiden Systemen registriert sein muß. Sie muß auf dem Server-System registriert sein, damit DCOM weiß, welcher EXE-Server zu laden ist und wie die per Marshaling transportierten Daten auf der Server-Seite zu behandeln sind. Die Komponente muß auf dem Client-System registriert sein, damit dort ein Proxy-Objekt der Komponente angelegt werden kann und die Marshaling-Daten auf der Client-Seite korrekt behandelt werden können.

Die serverseitige Registrierung ist einfach – genau wie bei der lokalen Registrierung braucht die EXE-Komponente nur einmal gestartet zu werden. Auf der Client-Seite können Sie `CliReg32.EXE` verwenden, um die Komponente zu registrieren, wobei die VBR-Datei verwendet wird, die beim Kompilieren erzeugt worden ist. Sie können auch die Komponente selbst registrieren. Jedoch wird Sie dann so lange ausdrücklich auf dem lokalen System laufen, bis Sie ebenso ausdrücklich einen anderen Ort über den Verbindungsmanager für Remote-Automatisierung festlegen. Das Projekt `perform.vbp`, das Sie in Kürze kennenlernen werden, demonstriert, wie Sie eine Komponente gleichzeitig auf dem lokalen und einem Remote-System laufen lassen können, und wie Sie auch einige Objekte auf dem lokalen System anlegen lassen, andere hingegen auf dem Remote-System.

Während der Design-Zeit und der Kompilierung

Nachdem Sie das Remote-Objekt registriert haben, können Sie das Objekt wie jede andere ActiveX-Komponente verwenden. Visual Basic ist es egal, wo sich das Objekt befindet.

Zur Laufzeit

Visual Basic verwendet die `CLSID` des Objekts, um den Server ausfindig zu machen, der das Objekt implementiert. Wird ein lokaler Server nicht gefunden, wird nach einem Remote-Server gesucht. Dann geschieht folgendes:

- Wird der Remote-Server gefunden, wird versucht, eine Verbindung zu dem Server herzustellen und das Anlegen des gewünschten Objekts anzufordern. Entschließt sich der Server, Ihnen das Anlegen des Objekts zu erlauben, wird die EXE des Servers auf dem Remote-System gestartet. In der Zwischenzeit legt das COM-Subsystem ein Proxy-Objekt auf dem lokalen System auf die gleiche Weise an, wie dieses auch für eine Out-of-Process-Komponente angelegt werden würde.
- Genau wie bei einem lokalen EXE-Server kann das Proxy-Objekt alle Interfaces des realen Objekts nachbilden, so daß Sie frühe Bindung verwenden können, indem Sie Objektvariablen des entsprechenden Objekttyps deklarieren. Wahrscheinlich wird der Performance-Vorteil der frühen Bindung angesichts des Netzwerk-Overheads kaum Wirkung zeigen.
- Aufrufe von Methoden und Eigenschaften des Objekts werden ausgeführt, indem die Funktionen im Interface des Proxy-Objekts aufgerufen werden. Das Proxy-Objekt reicht die Funktionsaufrufe und Parameter über die Remote-Procedure-Aufruf-Mechanismen des Betriebssystems über das Netzwerk weiter. Auf der Server-Seite wird das Objekt direkt aufgerufen. Verwenden Sie Remote-Automation anstelle von DCOM, kommt noch eine weitere Schicht hinzu. Dabei wird das Remote-Objekt nicht direkt aufgerufen, sondern Windows ruft eine Anwendung namens Remote-Automation-Manager auf, die alle Remote-Automation-Objekte verwaltet. Diese Anwendung greift dann auf das Objekt über die gewohnten Out-of-Process-Marshaling-Funktionen zu. Hier treffen Sie auf zwei Marshaling-Schichten, eine für prozeßübergreifende Vorgänge über das Netzwerk und eine zweite für prozeßübergreifende Vorgänge auf der Server-Seite. Dies ist einer der Gründe für die bessere Performance von DCOM gegenüber Remote-Automation.
- Alle Ergebniswerte oder als Referenz übergebene Parameter werden zum lokalen System über das Proxy-Objekt zum Client zurücktransportiert.
- Wenn das Projekt beendet wird oder die Objekt-Variable auf `Nothing` gesetzt wird, wird der Referenzzähler auf 0 zurückgesetzt, und das Proxy-Objekt wird gelöscht. Wenn der Remote-Server erkennt, daß alle bei ihm angeforderten Objekte freigegeben worden sind, wird er beendet.

6.5.1 Konfigurationsaspekte

Das wichtigste, was bei beiden Arten von Remote-Objekten zu bedenken wäre, ist die Tatsache, daß der Code jeweils auf dem Server läuft (auf Ihrem Server!) und daß der Code vollen Zugriff auf Ihr System hat. Der schnellste Weg zu einem DCOM-System ist, es einfach einzurichten und jedermann ohne Einschränkung zu erlauben, auf Ihrem System Objekte zu starten. Das ist natürlich auch der einfachste Weg, Hackern und anderen böswilligen Zeitgenossen Tür und Tor für ihre Untaten zu öffnen, angefangen vom Lesen all Ihrer privaten Daten bis hin zum Formatieren Ihrer Festplatte(n).

Denken Sie also zuallererst an Sicherheit, wenn Sie sich dazu entschließen, Remote-Objekte zuzulassen.

Das war die schlechte Nachricht. Und es gibt schlechtere.

Die Konfiguration von Remote-Automation oder DCOM ist in beiden Fällen eine recht komplexe Angelegenheit, die ein hohes Verständnis von Systemsicherheit, Netzwerkarchitektur usw. erfordert – alles Themen, die über den Horizont dieses Buches weit hinausreichen. Ich will Ihnen hier nichts weiter als ein Grundverständnis dafür vermitteln, was tatsächlich vor sich geht, wenn Sie Remote-Objekte über DCOM oder Remote-Automation verwenden. Dazu will ich Ihnen zeigen, daß es nicht kompliziert ist, Remote-Objekte zu verwenden, wenn man von den Konfigurationsfragen und Sicherheitsproblemen einmal absieht. Aus der Sicht Ihrer Programme ist es eine transparente Angelegenheit, was bedeutet, daß es Ihrem Programm im Prinzip egal ist, wo die angeforderten Objekte letztlich laufen – der programmäßige Umgang mit ihnen ist immer derselbe. Die Technologie sicher einzurichten und die Verwicklungen bei der Konfiguration der Systeme zu bewältigen – das steht auf einem anderen Blatt und ist jetzt nicht das Thema.

6.5.2 Was ist mit DLLs?

Sie werden bemerkt haben, daß in der Diskussion über Remote-Objekte bisher nur die Rede von ActiveX-EXE-Servern war. Wie steht es denn um Objekte, die Sie in einer DLL implementieren möchten?

Das Problem ist folgendes: Eine DLL kann nur in einem vorhandenen Prozeß laufen. Wenn Sie nun ein Remote-Objekt anlegen – in welchem Prozeß wird es tatsächlich geladen? Sicher, es könnte einen Prozeß geben, der DLLs für Sie lädt und die Daten per Marshaling zu Ihrem System transportiert. Doch dann hätten Sie es mit den gleichen Aspekten zu tun, wie bei EXE-Servern. Zum Beispiel mit der Frage, in welchem Programmfaden (Thread) die Objekte laufen. Wenn sich alle Komponenten einen einzigen Thread teilen würden, könnten Sie Performance- und Blockade-Probleme bekommen. DCOM ist (noch) nicht ganz so ausgeklügelt, so daß es erst einmal beim Betrieb von EXE-Servern bleibt.

Um DLL-Server laufen zu lassen, würden Sie ein komplexeres System benötigen, das Objekten Threads zuweisen könnte, deren Laden und Entladen verwalten

würde und viele weitere Features anbieten könnte, etwa Objektsicherheit und die Fähigkeit, Objektinstanzen wiederzuverwenden, so daß mit relativ wenigen Objektinstanzen eine größere Zahl an Clients bedient werden könnte.

Sie könnten einen EXE-Server erstellen, der dies alles kann. Sie können aber auch, wenn Sie es eilig und trotzdem die Zeit haben, die Handhabung eines weiteren komplexen Software-Pakets zu lernen, den Microsoft Transaction Server (MTS) einsetzen, der genau zur Bewältigung dieser Aufgabenstellung entwickelt worden ist.

Der Lebenszyklus eines Out-of-Process- bzw. eines Remote-Objekts ist nahezu identisch zu dem eines DLL-Objekts. Und die wenigen Unterschiede sind für einen Programmierer ohne Belang. Warum ist es dann so wichtig, daß Sie den Unterschied zwischen diesen Objekten verstehen? Ein Grund ist, daß Ihre Wahl der Implementierung einen nicht unerheblichen Einfluß auf die Performance hat.

6.5.3 Performance-Aspekte

Das Projekt `perform.vbp` im Ordner `samples\ch06` der Buch-CD demonstriert den Performance-Einfluß von Out-of-Process-Objekten. Den Code des Hauptformulars sehen Sie in Listing 6.2. Der Code ähnelt stark dem Code des Beispiels `binding.vbp` in Kapitel 4, »Das Component Object Model: Interfaces, Automation und Bindung«.

```
' ActiveX: A Guide to the Perplexed

' Performance-Beispiel
' Copyright (c) 1997 by Desaware Inc.

Option Explicit

' System-API-Aufruf, um Zeit in Millisekunden zu erhalten
Private Declare Function GetTickCount& Lib "kernel32" ()

' Zeitmarke
Dim CurrentTime As Long

Dim InApp As New clsSecurityLoan
Dim InProcDLL As New clsBankLoan
Dim OutProcEXE As New clsParentLoan
Dim DComProcEXE As clsParentLoan

Const repeats = 1000000
Const oprepeats = 10000&

Private Sub cmdTest_Click()
   Dim ctr&
   Dim res%
```

```
Dim InAppTime As Long
Dim InProcTime As Long
Dim OutProcTime As Long
Dim DComProcTime As Long

' Auf jedes Objekt zugreifen, um sicherzustellen,
' daß es geladen ist.
' Die Ladezeit soll nicht mitgemessen werden
res = InApp.Duration
res = InProcDLL.Duration
res = OutProcEXE.Duration
Set DComProcEXE = _
 CreateObject("gtpParentLoan.clsParentLoan", _
 txtRemote.Text)
res = DComProcEXE.Duration

Screen.MousePointer = vbHourglass
CurrentTime = GetTickCount()
For ctr = 1 To repeats
   res = InApp.Duration
Next ctr
InAppTime = (GetTickCount() - CurrentTime)
' Nun InProc-Operation
CurrentTime = GetTickCount()
For ctr = 1 To repeats
   res = InProcDLL.Duration
Next ctr
InProcTime = (GetTickCount() - CurrentTime)

' Nun Out-of-Process-Operation
CurrentTime = GetTickCount()
For ctr = 1 To oprepeats   ' Weniger Zeit Out-of-Process
   res = OutProcEXE.Duration
Next ctr
OutProcTime = (GetTickCount() - CurrentTime)

' Nun Remote-Operation
CurrentTime = GetTickCount()
For ctr = 1 To oprepeats   ' Weniger Zeit Out-of-Process
   res = DComProcEXE.Duration
Next ctr
DComProcTime = (GetTickCount() - CurrentTime)

Screen.MousePointer = vbNormal

lstResults.AddItem "Within application"
lstResults.AddItem "  " & GetTime(InAppTime) & _
 " microseconds"
```

```
   lstResults.AddItem "In Process DLL"
   lstResults.AddItem "   " & GetTime(InProcTime) & _
    " microseconds"
   lstResults.AddItem "Out of Process EXE"
   lstResults.AddItem "   " & GetTime(OutProcTime) _
    * CDbl(repeats / oprepeats) & " microseconds"
   lstResults.AddItem "Remote EXE via DCOM"
   lstResults.AddItem "   " & GetTime(DComProcTime) _
    * CDbl(repeats / oprepeats) & " microseconds"
End Sub

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

Listing 6.2: Listing des Perform-Projekts

Dieses Objekt verwendet die eigentlich für die Loan6-Anwendung erstellten Objekte. Es werden Verweise auf die Objekte `clsBankLoan` und `clsParentLoan` hinzugefügt. Die Klasse `clsSecurityLoan` ist direkt in das Projekt eingefügt worden.

Vier Objekte werden definiert, eines für jede Klasse und ein gesondertes `clsParentLoan`-Objekt als Remote-Objekt. Die Anzahl der Wiederholungen des Out-of-Process- und des Remote-Beispiels ist wesentlich niedriger als bei den beiden InProcess-Beispielen, da anderenfalls die Tests womöglich stundenlang dauern könnten. (Sie können die Werte aber nach Belieben entsprechend Ihrer Systemleistung anpassen.)

Im `cmdTest_Click`-Ereignis wird als erstes auf eine Eigenschaft jedes Objekts zugegriffen. Dies ist notwendig, da Visual Basic mit dem tatsächlichen Anlegen eines Objekts so lange wartet, bis das erste Mal darauf zugegriffen wird. Die New-Option in der Dimensionierungsanweisung legt ein Objekt noch nicht tatsächlich an – es wird damit nur gesagt, daß das Objekt angelegt werden soll, sobald darauf zugegriffen wird. In diesem Fall wollen wir aber die Objekte angelegt wissen, bevor wir mit der Zeitmessung beginnen, um Verzerrungen derselben zu vermeiden. Das `DComProcEXE`-Objekt wird als Remote-Objekt angelegt, indem in der `CreateObject`-Funktion direkt der Name des Servers angegeben wird, auf dem das Objekt angelegt werden soll (dies ist ein neues Feature in Visual Basic 6.0). Der Name des Servers wird in eine Textbox auf dem Formular eingegeben.

Da Sie die Zeit für einen Funktionsaufruf messen wollen, ist es sinnvoll, eine Funktion zu wählen, die selbst so wenig wie möglich tut. Die Funktion zur Abfrage der `Duration`-Eigenschaft ist eine solche, da sie nichts weiter als einen Integer-Wert zurückgibt.

Abbildung 6.4 illustriert eine typische Ergebnisfolge eines Testlaufs. Stellen Sie sicher, daß die ausführbare Version verwendet wird, oder kompilieren Sie das Projekt selbst in Native Code, bevor Sie den Test starten. Anderenfalls würden Sie nämlich zusätzlich die Differenz zwischen der Ausführung als P-Code und als Native Code messen, was das Ergebnis natürlich verfälschen dürfte. Achten Sie auch darauf, daß beide Server, `gtpPln.EXE` und `gtpBkln.DLL`, registriert sein müssen, damit das Programm laufen kann. Wenn Sie keinen Server zur Verfügung haben sollten, auf dem das Programm `gtpPln.EXE` für den Remote-Test registriert werden könnte, oder wenn Sie nicht die Enterprise-Version von Visual Basic (diese wird für DCOM und Remote-Automation benötigt) zur Verfügung haben sollten, entfernen Sie alle Verweise auf die `DComProcEXE`-Variable in dem Programm, bevor Sie es starten.

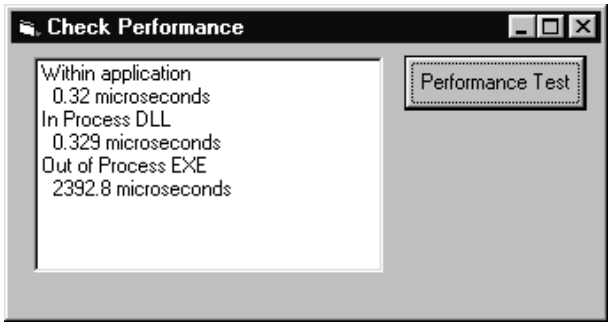

Abb. 6.4: Hauptformular des Perform-Projekts in Aktion

Wie Sie sehen können, gibt es einerseits keinen signifikanten Unterschied zwischen der Implementierung als DLL oder direkt in einem Projekt. Die auftretenden geringfügigen Unterschiede sind auf die geringe Auflösung des System-Timers zurückzuführen und eventuell auch auf die aktuelle übrige Belastung des Gesamtsystems. (Windows-32-Bit-Betriebssysteme sind Multitasking-Systeme, so daß andere Operationen stattfinden können, während Ihre Messungen laufen.)

Andererseits treten die Unterschiede zwischen In-Process- und Out-of-Process-Aufrufen überwältigend deutlich zutage. Die Minimierung von prozeßübergreifenden Operationen ist daher eines der wichtigsten Ziele, wenn Sie die Performance einer Anwendung erhöhen wollen. Daß Remote-Aufrufe noch schlechter dastehen, ist sicher weniger verwunderlich.

Eine wichtige Kleinigkeit am Rande: Auch wenn der Zugriff auf das Objekt `clsParentLoan` Out-of-Process erfolgt, ist es dennoch früh gebunden! Denn da dem Perform-Projekt ein Verweis auf das Objekt hinzugefügt worden ist und eine Variable des Typs `clsBankLoan` verwendet wird, kann die direkte Bindung zum Klassen-Interface des Objekts erfolgen. Da das Objekt allerdings Out-of-Process läuft, wird tatsächlich das Klassen-Interface des Proxy-Objekts gebunden.

Wenn Sie die hier gezeigten Ergebnisse mit denen der Binding-Anwendung in Kapitel 4 vergleichen, werden Sie feststellen, daß die Performance-Verluste bei der Verwendung von Out-of-Process-Objekten deutlich größer ausfallen als bei der späten Bindung. Würde hier obendrein auch noch späte Bindung verwendet, würden die Performance-Verluste natürlich noch höher ausfallen.

Alle ActiveX-Komponenten sind entweder der Kategorie der InProc- oder der Out-of-Process-Server zuzuordnen. Aber innerhalb dieser Kategorien treffen Sie auf eine ganze Reihe an Varianten, wie Sie im nächsten Kapitel sehen werden.