Kapitel 8

Das Projekt

8.1 Überblick über das Projekt 176
8.2 Instanzierung 192
8.3 Projekt-Eigenschaften 200
8.4 Wie geht es weiter? 208

Nun sind wir soweit, wir können eine Code-Komponente anlegen. Sie haben eine Aufgabe vor Augen – Sie können sich ein gekapseltes Objekt mit einem klar definierten Interface vorstellen, das in eine Anwendung zu integrieren wäre. Wahrscheinlich beabsichtigen Sie auch, daß dieses Objekt von mehreren Anwendungen gleichzeitig oder auch vielleicht von künftigen Anwendungen verwendet werden kann. Sie gehen ebenso davon aus, daß die Implementierung dieses einen Objekts häufiger aktualisiert werden könnte, ohne daß Sie die gesamte Anwendung bei jeder Aktualisierung mitliefern müßten. Und vielleicht sind Sie auch darauf angewiesen, einige der Vorteile von in einer ausführbaren Datei implementierten Objekten in Anspruch zu nehmen.

Dazu gibt es viele Möglichkeiten. Zunächst kommt es darauf an, die richtigen Fragen zu stellen. Sie müssen zuallererst klären, was Sie erreichen wollen, bevor Sie den ersten, besonders kritischen Schritt in Ihrem Projekt unternehmen können.

8.1 Überblick über das Projekt

Ihr Projekt in einer ActiveX-Code-Komponente besteht aus vier Elementen: dem Projekt selbst, den Klassen-Modulen, den Standard-Modulen und den Formularen.

Das Projekt bestimmt über einige Kerneigenschaften der Code-Komponente. So wird im Projekt etwa festgelegt, ob sie In-Process oder Out-of-Process laufen soll, ob sie single- oder multithreaded (darauf kommen wir später noch zu sprechen) sein soll, und wie auf Ihre Komponente von außen zugegriffen werden soll. Ein Projekt kann mehrere Objekte oder auch kein einziges offenlegen.

Die Klassen-Module, soweit vorhanden, stellen die Objekte der Komponente dar. Es sind die Objekte, die nach außen hin sichtbar sind und das Objekt-Modell des Projekts repräsentieren. Darüber hinaus können auch private Objekte innerhalb des Projekts recht nützlich sein.

Standard-Module, soweit vorhanden, sind niemals nach außen hin sichtbar. Sie enthalten meistens innerhalb des Projekts global verwendete Funktionen und Variablen. Die tatsächliche Bedeutung des Begriffs »global« hängt jedoch von der Art der Komponente ab.

Formulare stellen die primäre Benutzeroberfläche in Visual Basic dar. Viele Code-Komponenten haben keine Benutzeroberfläche – es ist tatsächlich möglich, Code-Komponenten zu erstellen, in denen alle Benutzeroberflächen-Elemente gesperrt sind. Code-Komponenten, die etwa zu eigenständigen Anwendungen mit einem offengelegten Objekt-Modell gehören, sind dagegen stark auf Formulare angewiesen. Einige Code-Komponenten verwenden Formulare als Behälter für Controls. Behalten Sie im Blick, daß Formulare ebenfalls Objekte sind, die über Methoden, Eigenschaften und Ereignisse verfügen können.

Bei jedem Projekt sind folgende Schritte auszuführen (wenn auch nicht immer und unbedingt in dieser Reihenfolge):

- Definieren Sie die zu erreichende Zielsetzung.
- Entwerfen Sie ein entsprechendes Objekt-Modell.
- Konfigurieren Sie die Projekt-Einstellungen für die Anforderungen der Komponente.
- Definieren Sie die Klassen und Formulare, die Sie zur Implementierung der Komponente benötigen. Diese werden die Objekte sein, die die Komponente verwenden wird, auch wenn nicht alle nach außen hin offengelegt werden sollen.
- Definieren Sie die globalen Variablen und Funktionen, die in der Komponente benötigt werden.

8.1.1 Entwurf des Objekt-Modells

Einerseits ist es nicht gerade einfach, eine Komponente zu entwerfen, wenn man die Möglichkeiten einer Programmiersprache nicht im Griff hat. Andererseits ist es genauso schwierig, die Möglichkeiten einer Sprache zu verstehen und darzustellen, wenn man nicht weiß, wie die Anforderungen an eine Anwendung zu definieren sind und wie ein Objekt-Modell dafür auszusehen hätte. Womit also nun beginnen?

Microsofts Ansatz ist eine Schritt-für-Schritt-Anleitung, die anhand eines Beispiels viele Sprach-Features aufzeigt. Dabei werden allerdings die Vor- und Nachteile und Wahlmöglichkeiten innerhalb dieses Beispiels meistens nur am Rande gestreift oder ganz übergangen. Danach geht die Dokumentation im Detail auf die Sprach-Features ein.

Dies ist sicher kein schlechter Ansatz. Aber da dieses Buch weniger als Ersatz der Microsoft-Dokumentation, sondern eher als Erweiterung gedacht ist, halte ich einen anderen Ansatz für angebrachter. Da ich ja soeben eine Vorgehensweise für das Anlegen einer Komponente vorgegeben hatte, sollte nun eine Demonstration derselben folgen. Doch statt nur einer simplen Schritt-für-Schritt-Beschreibung, werde ich soweit wie möglich ins Detail gehen und Ihnen sowohl Variationsmöglichkeiten als auch die jeweiligen Vor- und Nachteile aufzeigen.

Zuerst ist die Aufgabe festzulegen.

Wir mußten beispielsweise kürzlich eine Lösung für ein Problem finden, das Sie sicher auch kennen. In unserer Firma hatten einige Leute auf Aktien eines bestimmten Unternehmens gesetzt. Doch plötzlich rutschten die Kurse dieser Aktie in den Keller. Doch da bei uns wegen der Qualität der Produkte des Unternehmens alle zuversichtlich waren, daß sich der Kurs wieder erholen würde, unterbrachen die Leute recht häufig ihre Arbeit, um die neuesten Kursinformationen einzuholen.

Nun, ich denke, daß ich kein kleinlicher Arbeitgeber bin, und auch ich, wie ich zugeben muß, unterbrach hin und wieder meine Arbeit, um einen Blick auf den Aktienmarkt zu werfen. Doch irgendwie lief das Ganze ein wenig aus dem Ruder. Wir benötigten unbedingt ein Programm, das ganz einfach im Hintergrund sitzend den Aktienkurs verfolgen und uns automatisch informieren würde, wenn der Kurs bestimmte Marken überspringen würde. Es müßte ein einfaches Programm sein, das in 10 oder 15 Minuten zusammenzuschustern sein sollte – mehr Zeit damit zu verbringen würde dem Sinn der Aktion widersprechen, nämlich Zeit zu gewinnen. Schließlich wäre das einfach widersinnig, Tage mit der Programmierung eines Kurs-Monitors zu verbringen, bloß um hinterher hier und da ein paar Minuten Zeit zu sparen.

Somit ist das Programm wirklich simpel geworden. Stellen Sie sich ein Formular vor, auf dem eine Textbox zur Eingabe eines Aktienwertes plaziert ist. Es enthält dazu eine Variable zur Aufnahme des momentanen Aktienkurses. Alle paar Minuten fragt die Anwendung den aktuellen Kurs ab und vergleicht ihn mit dem vorherigen Wert. Ändert sich der Preis, macht sich das Programm akustisch oder optisch bemerkbar. Man kann auch einen Kurskorridor angeben, so daß das Programm sich nur dann meldet, wenn der Kurs diesen Korridor verläßt. Eine erweiterte Version dieses Programms könnte sogar mehrere Aktienkurse zugleich verfolgen. Es könnte auch mit einer Datenbank verbunden werden und so eine ganze Liste von Kursen verfolgen – aber damit greife ich nun schon zu weit voraus.

Nun, für Sie als einen einigermaßen erfahrenen Visual-Basic-Programmierer mag die Aufgabenstellung trivial erscheinen. Sie kennen Formulare, Timer und Textboxen. Die Idee eines Programms, das in einem Timer-Ereignis einen gespeicherten Wert mit einem neuen Wert vergleicht, ist auch nichts Weltbewegendes.

Die Chancen stehen somit nicht schlecht, daß Sie erkennen, wo der eigentliche Knackpunkt des Programms liegt. Visual Basic bietet nämlich von Hause aus keine Funktion zur Abfrage eines Aktienkurses.

Bei näherer Betrachtung eines solchen Kursmonitor-Programms gilt es nun, eine wesentliche Entscheidung zu treffen. Denken Sie etwa daran, wie ein Aktienkurs abzufragen und wie diese Funktionalität in das Monitor-Programm einzubauen wäre? Oder spielen Sie mit dem Gedanken, eine irgendwie geartete Komponente zu erstellen, die Aktienkurse abfragen kann?

Um diese Frage zu beantworten, schauen wir uns einmal eine Darstellung des Programms in Form von Pseudocode an. Falls Sie diesen Begriff noch nicht kennen sollten: *Pseudocode* bedeutet hier, daß wir uns einer formalen, Code-ähnlichen Sprache bedienen, die aber nicht als echte Programmiersprache existiert. Dieser Pseudocode ist etwas anderes als Visual Basics *P-Code* – das ist die Sprache eines kompilierten Zwischenstadiums, die für interpretierte Visual-Basic-Programme erzeugt wird. Wenn Sie ein Programm mit Pseudocode beschreiben, beschreiben Sie lediglich in natürlicher Sprache, was das Programm tun wird. Das Aktienkurs-Monitor-Programm könnte in Pseudocode etwa wie folgt aussehen:

```
Hole den aktuellen Kurs und speichere diesen
Bei jedem Timer-Ereignis
    Hole den aktuellen Kurs
    Wenn aktueller Kurs ungleich gespeicherter Kurs, dann
        Benachrichtige den Anwender
    Ende Wenn
Ende Timer-Ereignis
```

Sie sehen, was ich meine? Kein Kompiler (den ich kenne) kann dieses Programm verarbeiten. Es beschreibt jedoch deutlich, was das Programm tut, und kann als Aufhänger für die konkrete Implementierung dienen.

Das einzige, was Sie nicht direkt mit Visual-Basic-Funktionen nachbilden können, ist die erste Zeile »Hole den aktuellen Kurs und speichere diesen«. Diese Aufgabe wird sogar im Programm zweimal abgearbeitet. Beachten Sie, daß das Programm zwar auf diese Operation angewiesen ist, die Operation selbst jedoch von keinem anderen Teil des Programms abhängig ist. Dies bedeutet, daß diese Operation als eigenständige, unabhängige Funktion implementiert werden kann.

Ein weiterer Gesichtspunkte wäre folgender: Würden Sie die Funktion »Hole den aktuellen Kurs und speichere diesen« ändern, müßten Sie auch das übrige Programm ändern. Würden Sie jedoch nur das übrige Programm ändern, hätte dies keine Auswirkungen auf die Kursabfrage-Funktion. Die Abhängigkeit besteht demnach nur in einer Richtung.

Die Frage, die Sie nun zu beantworten haben, lautet also: Kommt die Operation, den aktuellen Preis einer Aktie abzufragen, nur in dieser Anwendung vor? Oder könnten auch andere Anwendungen die Funktion gebrauchen?

Dies ist eine entscheidende Frage. Wenn Sie nun den Code zur Implementierung dieser Funktionalität wiederverwenden wollen, dann müssen Sie ihn auf irgendeine Weise isolieren. Zur Abtrennung von Code zur Wiederverwendung stehen Ihnen mehrere Möglichkeiten offen:

- Plazieren des Codes in einer eigenen Funktion und Kopieren dieser Funktion in andere Projekte nach Bedarf.
- Plazieren des Codes in einem eigenen Modul und Aufnahme dieses Moduls in andere Projekte nach Bedarf.
- Plazieren des Codes in einer ActiveX-Komponente und Zugriff durch jede Anwendung, die diese Funktionalität verwenden will.

Schauen wir uns die Vor- und Nachteile in bezug auf unser konkretes Beispiel an.

Die Plazierung des Codes in einer Funktion innerhalb eines Formulars oder eines Moduls der Anwendung hat mehrere Nachteile. Das Kopieren von einem Projekt ins andere ist umständlich; Sie müssen sich immer wieder daran erinnern, wenn Sie die Funktion benötigen, in welchem Modul sie steckt. Natürlich gibt es auf dem Markt Tools wie Code-Bibliotheken, die dieses Problem sich erübrigen las-

sen würden. Wenn Sie allerdings diese Funktion häufiger verwenden, wären in kürzester Zeit viele Kopien der Funktion über Ihr System verstreut. Wenn sich nun nachträglich ein Fehler in dieser Funktion herausstellen sollte, müßten Sie nicht nur jedes Programm, das die Funktion enthält, erneut kompilieren, sondern diese auch in jedem einzelnen Programm-Projekt korrigieren – was Sie natürlich auch wieder durch Kopieren erledigen könnten. Ich will damit ja nun nicht sagen, daß in jeder Ihrer Funktionen ein Fehler steckt, aber ...

Ein anderer schwerwiegender Nachteil dieses Ansatzes tritt in Erscheinung, wenn es sich um eine Funktionalität handelt, die nicht einfach in einer einzigen Funktion unterzubringen ist. Komplexere Aufgaben erfordern eher mehrere Funktionen, die sich über mehrere Module, Objekte oder Controls erstrecken würden. Die Technik des Kopierens wäre in diesem Rahmen nicht mehr praktizierbar.

Aber es gibt doch sicher auch Vorteile, wenn eine Funktion innerhalb einer Anwendung gehalten wird? Nun ja, Sie können sich dann beispielsweise sicher sein, daß keine andere Anwendung oder keines von deren Code-Modulen von einer Änderung betroffen sein würde, die Sie in der einen Funktion vornehmen würden. Dieser Ansatz gibt Ihnen somit die größte Kontrolle über den Code einer einzelnen Anwendung.

Wäre dieser Ansatz sinnvoll für die Aktienkurs-Aufgabe?

Nein, denn die Operation der Kursabfrage läßt sich klar abgrenzen und definieren. Und sollte sich ein Fehler in Ihrer Implementierung herausstellen, sollte es nach dessen Bereinigung auf eine einzige Neukompilierung hinauslaufen, statt auf ein Durchsuchen der Module aller Anwendungen. Dagegen ist die Wahrscheinlichkeit, daß eine Änderung in einem gemeinsam verwendeten Modul Auswirkungen auf das Kursmonitor-Programm hätte, eher als gering anzusehen. Schließlich ist eine Kursabfrage doch eher eine komplexere Aufgabe, die sich – soweit ich das sehe – nicht in eine einzige Visual-Basic-Funktion packen läßt.

Schauen wir uns daher die nächste Möglichkeit an.

Wird der Code in einem mehrfach verwendeten Modul plaziert, erledigen sich die meisten Nachteile des vorigen Ansatzes. Ein einzelnes Modul kann in viele Projekte eingefügt werden. Änderungen des Moduls werden in jeder dieser Anwendungen wirksam, sobald diese neu kompiliert werden. Da der Code in einem einzelnen, gemeinsam genutzten Modul steht, besteht nicht die Gefahr eines Versionswildwuchses. Nach wie vor stellt sich jedoch das Problem, wenn komplexere Aufgaben mehrere Objekte oder Controls benötigen. In diesen Fällen werden Sie einen umfangreicheren Satz an Modulen, Formularen, Klassen und Controls benötigen, um die Funktionalität zu implementieren. Doch mag das nicht unbedingt ein schwerwiegendes Problem darstellen.

Der eigentliche Nachteil dieses Ansatzes liegt darin, daß Änderungen in dem Modul erst dann Wirkung zeigen, nachdem die Anwendung erneut kompiliert worden ist. Sie können das als Vorteil, aber auch als Nachteil ansehen. Verwenden Sie eine ActiveX-Komponente zur Implementierung der Funktionalität, brauchen

Sie nur die neu kompilierte Komponente auszuliefern. Ihre Anwendung wie auch andere Anwendungen, die die Komponente verwenden, sind mit einem Schlag aktualisiert. Natürlich wirken sich genauso auch alle neuen Fehler, die sich in die Komponente eingeschlichen haben könnten, in allen Anwendungen auf einen Schlag aus! Ein gemeinsam verwendetes Modul fügt dagegen den Code in die ausführbare Datei ein (was zu größeren EXE-Dateien führt), erübrigt statt dessen jedoch die Auslieferung einer separaten Komponente und die Umstände der Versionskontrolle.

Ist nun dieser Ansatz eine Lösung für die Aktienkurs-Aufgabe? Möglicherweise ja. Doch eine Reihe von Faktoren, wie die folgenden, lassen den Ansatz der Verwendung einer ActiveX-Komponente günstiger erscheinen:

- Eine ActiveX-Komponente erweitert Ihr System auf effiziente Weise um die Fähigkeit, Aktienkurse abzufragen. Sie kann nicht nur von Visual Basic aus genutzt werden, sondern auch von vielen anderen Anwendungen, die ActiveX-Automation unterstützen, wie Excel beispielsweise.
- Es ist zu erwarten, daß jeder, der Aktienkurse brauchen kann, Bedarf an weiteren Daten über einen Aktienwert haben könnte, wie etwa Tageshöchst- und -niedrigststände, Schlußpreise vom Vortag und ähnlichem. Dieser Satz sollte sämtliche Alarmglocken in Ihrem Gehirn schrillen lassen. Denken Sie darüber nach: Viele verschiedene Datenwerte, mit diesen Daten verbundene Funktionalität (hier: Werte speichern und Berechnungen mit ihnen ausführen). Mittlerweile sollte Ihnen das Begriffspaar »Daten & Funktionalität« das Wort *OBJEKT* geradezu entgegenschreien! ActiveX-Komponenten stellen nunmal offengelegte Objekte dar. Alles klar?
- Nehmen wir an, die von Ihnen verwendete Technik zur Abfrage der Aktienkurse geschieht über das DFÜ-Netzwerk bzw. das Internet. Was dann, wenn die Technik eines Tages versagen sollte, wenn der Kursanbieter nicht mehr existieren sollte? Sie müßten schnellstmöglichst eine neue Technik implementieren! Das letzte, was Sie sich wünschen könnten, wäre die Neukompilierung und Neuauslieferung dutzender Anwendungen, die diese Abfrage-Komponente verwenden. Statt dessen bräuchten Sie nur die Komponente zu aktualisieren, um auf den neuen Kursanbieter zuzugreifen, und zu verteilen. Alle Anwendungen werden nun über diese neue Version der Komponente mit dem neuen Kursanbieter zurechtkommen.

Damit dürfte die Wahl nun klar sein. Wenn Sie an einer Wiederverwendung der Kursabfrage-Funktionalität nicht interessiert sein sollten, können Sie sie in die Anwendung integrieren. Aber wenn Sie sie wiederverwenden wollen, sollte die Lösung definitiv als ActiveX-Komponente implementiert werden – also als Objekt.

Da ich selbst recht faul bin, also nicht gerne dieselbe Arbeit zum wiederholten Male ausführen mag, hat für mich die Wiederverwendbarkeit höchste Priorität.

Wenn ich mich schon einmal mit dem Problem einer Kursabfrage herumschlage, sollte einmal eben auch genug sein. Also bleibt es bei einer ActiveX-Komponente.

Aber welcher Komponenten-Typ? Soll es eine DLL oder eine EXE sein? Eine gute Frage, die Sie sich merken sollten. Denn noch sind wir nicht ganz so weit, diese zu beantworten.

Beschließen wir diesen Abschnitt mit einem Gedanken, der die Objekt-Natur der Aufgabe betrifft. Während der aktuelle Kurs einer Aktie fürs erste reichen sollte, sollte ein Kursabfrage-Programm auch weitere Informationen über einen Aktienwert abfragen können. Sie können sich also ein Objekt für eine bestimmte Kursabfrage mit folgenden Eigenschaften vorstellen:

- `Symbol` – Tickersymbol einer bestimmten Aktie
- `LastPrice` – der aktuelle, zuletzt gehandelte Kurs
- `High` – der Höchststand des Tages
- `PriorClose` – der Schlußkurs des Vortages
- `QuoteTime` – die Uhrzeit der letzten Notierung
- `CompanyName` – der volle Name des Unternehmens

Nun stellen Sie sich vor, daß das Objekt über eine Methode zum Laden der aktuellen Informationen zu der Aktie verfügt. Die Methode könnte beispielsweise `GetQuote` heißen. Von nun an werden wir das Objekt `StockQuote`-Objekt nennen.

Die zu erstellende ActiveX-Komponente erlaubt das einfache Anlegen von `StockQuote`-Objekten. Dies ist nur ein einfaches Objekt-Modell. Wir werden weiter hinten in diesem Kapitel im weiteren Verlauf des Buchs komplexeren Objekt-Modellen begegnen.

8.1.2 Wahl des Projekttyps: EXE oder DLL

Soll der `StockQuote`-Server als In-Process-DLL oder als Out-of-Process-EXE implementiert werden? Aufgrund der Technik-Diskussion in Teil I dieses Buches, scheint die Wahl eindeutig zu sein. In-Process-Code läuft wesentlich schneller als Out-of-Process-Code – somit sollte die Komponente als DLL implementiert werden.

Sollte sie das wirklich?

Wenn die Wahl wirklich so eindeutig wäre, gäbe es keine Existenzberechtigung für EXE-Server. Das heißt also, daß EXE-Server wohl auch ihre Vorteile haben müssen. Es sind da einige Punkte zu berücksichtigen:

- Performance
- Ausführung im Hintergrund

- Gemeinsame Nutzung von Ressourcen durch mehrere Prozesse
- Multithreading-Aspekte

Lassen wir die Multithreading-Aspekte vorläufig außen vor – ihnen ist ein eigenes Kapitel gewidmet, das Kapitel 14.

Performance

Obwohl wir das Thema bereits in Kapitel 1 besprochen haben, gehen wir noch einmal kurz auf den Performance-Aspekt ein. Wichtig wäre hier nachzutragen, daß Sie die relative Performance im Vergleich der beiden Ansätze nicht ganz so einfach auf die Aussage »EXE-Server sind langsamer als DLL-Server« reduzieren können. Zutreffender wäre die Aussage, daß der Zugriff auf in einem EXE-Server implementierte Methoden und Eigenschaften langsamer ist als der Zugriff auf in einem DLL-Server implementierte Methoden und Eigenschaften. Der Code innerhalb einer Komponente läuft so schnell es geht. Sie haben gesehen, daß eine sehr kurze Operation, wie etwa `Property Get`, auf einem EXE-Server wesentlich länger dauert. Doch was gilt, wenn Sie es mit einer Operation zu tun haben, deren Abarbeitung selbst schon einige Minuten in Anspruch nimmt – etwa mit einer umfangreichen Datenbankabfrage oder mit der Erstellung eines Berichts? In diesem Fall ist der Unterschied von ein paar Mikro- oder Millisekunden im Verhältnis zur Gesamtbearbeitungsdauer einer Funktion vernachlässigbar.

Das `StockQuote`-Objekt ist in dieser Hinsicht exemplarisch. Einerseits erwarten Sie, daß die Eigenschaften-Zugriffe, wie etwa die Abfrage der letzten Kursnotierung oder der Abfragezeit, sehr schnelle Operationen sein werden. In diesem Fall würde sich ein Marshaling-Overhead deutlich bemerkbar machen. Andererseits wird die `GetQuote`-Operation relativ langsam ablaufen, da sie von der Kommunikation mit einem externen Provider abhängig ist.

Bei der Abwägung der Performance-Auswirkungen kommt auch die Häufigkeit der Zugriffe auf das Objekt durch die Anwendung ins Spiel. Ein ständig benötigtes Objekt wird wahrscheinlich besser in einem DLL-Server aufgehoben sein, da die Marshaling-Vorgänge das System belasten. Die Auswirkung in bezug auf das gesamte Performance-Verhalten einer Anwendung wird jedoch wiederum vernachlässigbar sein, wenn auf das Objekt nur selten zugegriffen wird. Im Fall des `StockQuote`-Objekts werden sich die Bearbeitungszeiten im Bereich von Hunderten von Millisekunden, wenn nicht gar mehreren Sekunden bewegen. Genau vorherbestimmbar ist das wegen der Bandbreite der Performance-Leistungen der verschiedenen, heutzutage verbreiteten Maschinen nicht.

Soll das nun bedeuten, daß das `StockQuote`-Objekt als ActiveX-EXE-Komponente implementiert werden sollte? Nicht unbedingt. Damit soll nur gesagt werden, daß der Performance-Aspekt für dieses eine Objekt von untergeordneter Bedeutung ist. Sie sollten daher noch die anderen durchleuchten, ehe Sie die Entscheidung treffen.

Ausführung im Hintergrund

Wenn Sie bisher aufmerksam mitgelesen haben, werden Sie bemerkt haben, daß ich den Begriff *Multithreading* bereits mehrfach verwendet habe, aber immer darauf verwiesen habe, daß er noch ohne Bedeutung wäre und ich später noch darauf zu sprechen käme. Beides gilt nach wie vor. Aber um einen der Kern-Unterschiede zwischen DLL- und EXE-Servern zu verstehen, sollten Sie ein wenig über Threads und das Multitasking-Verhalten der Windows-32-Bit-Betriebssysteme Bescheid wissen. Wer sich mit Multithreading und Multitasking bereits bestens auskennt, kann das folgende gegebenenfalls überspringen. Für alle anderen fangen wir mit den Grundlagen an.

Sie wissen, daß Sie unter Windows mehrere Anwendungen gleichzeitig laufen lassen können. Sie haben vielleicht auch bemerkt, daß da mehr im Spiel ist, als nur schnell zwischen den Anwendungen hin- und herzuschalten. Das Beispiel `Counter.vbp` demonstriert dies. Es enthält ein Label-Control und einen Timer mit einem Intervall von 200 ms. Hier der Code:

```
Option Explicit

Dim countval&

Private Sub Timer1_Timer()
   countval = countval + 1
   lblCounter.Caption = countval
End Sub
```

Starten Sie zwei Instanzen der kompilierten Version `Counter.EXE`. Beide werden simultan zählen. Wie kann ein einzelner Prozessor zwei Anwendungen gleichzeitig laufen lassen? Bei der Antwort auf diese Frage lassen wir einmal Mehrprozessorsysteme außen vor – dort kommen die gleichen Prinzipien zur Geltung. Nun, es funktioniert, weil das Betriebssystem sehr schnell zwischen den Anwendungen umschalten kann. Tatsächlich laufen zu jedem Zeitpunkt mehrere Prozesse. Einige davon sind vom Betriebssystem selbst gestartet worden und werden von ihm selbst verwaltet.

Abbildung 8.1 zeigt das Multitasking zwischen zwei Anwendungen. Die erste Anwendung läuft eine geraume Zeit, dann unterbricht das Betriebssystem deren Ausführungsfluß und schaltet zur anderen Anwendung um. Dies geschieht sehr schnell, so daß der Eindruck entsteht, als ob beide Anwendungen gleichzeitig laufen würden.

Als Programmierer brauchen Sie sich im allgemeinen nicht um die Tatsache zu kümmern, daß Ihre Anwendung unterbrochen wird. Sie können darauf vertrauen, daß Windows die Ausführung des Codes an genau der Stelle wiederaufnehmen wird, an der sie unterbrochen worden war. Aus Ihrer Sicht als Programmierer besteht Ihr Programm aus einer kontinuierlichen Abfolge von Anweisungen. Die Abfolge wird als *Execution-Thread* (»Ausführungsfaden«) bezeichnet. Das

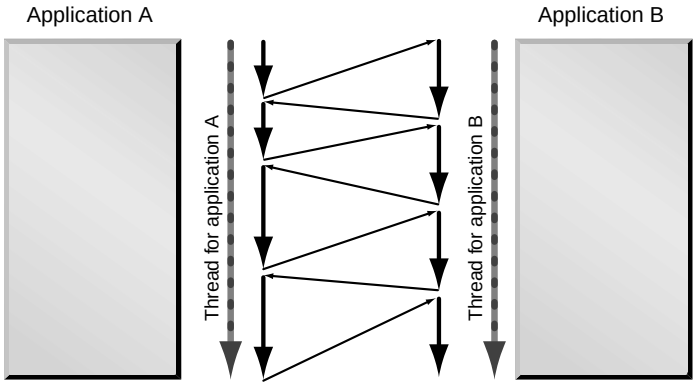

Abb. 8.1: Auswirkung von Multitasking auf den Programmfluß

Betriebssystem sieht zwei verschiedene Prozesse, jeden in seinem eigenen Thread. Es kann zwischen diesen Threads umschalten. Aus Ihrer Sicht jedoch wird der Faden nicht unterbrochen – dies sollen die vertikalen Linien andeuten.

Während in Abbildung 8.1 Multitasking aus der Sicht des Betriebssystems dargestellt ist, zeigt Abbildung 8.2 die Zeitlinie aus der Perspektive des Programmierers. Anwendung A und Anwendung B laufen beide jeweils in ihrem eigenen Thread, und beide Threads laufen gleichzeitig.

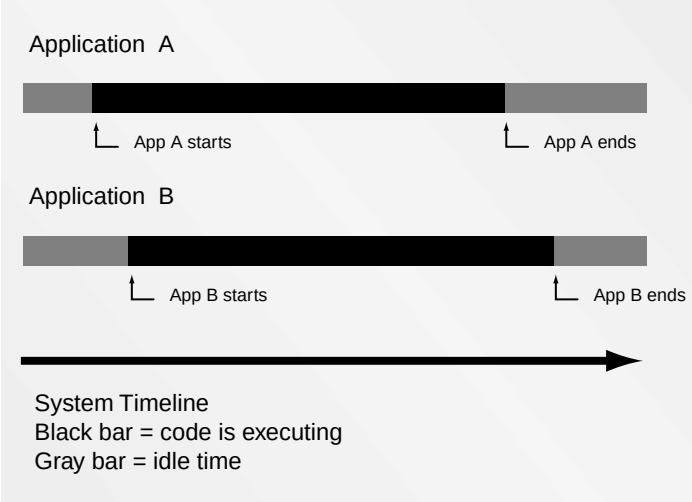

Abb. 8.2: Anwendungen laufen in ihren eigenen Threads

Ein Thread kann sich in einem von drei möglichen Zuständen befinden. Er kann sich in der Ausführung befinden, er kann sich im Ruhezustand befinden (d.h. er ist bereit, ausgeführt zu werden, jedoch führt der Prozessor gerade einen anderen Thread aus) oder er kann blockiert sein (d.h. er wartet auf die Bereitstellung einer System-Ressource oder auf ein System-Ereignis – das Betriebssystem wird den Thread nicht aufrufen, bevor nicht die Ressource verfügbar ist oder das Ereignis auftritt).

Jede Anwendung läuft in ihrem eigenen Prozeß, und jeder Prozeß hat einen Haupt-Thread. Ein Prozeß kann mehrere Threads haben – auf diesen Punkt werde ich unter dem Stichwort Multithreading näher eingehen.

Die Natur des Multithreadings unter Windows beeinflußt die Wahl zwischen EXE-Server und DLL-Servern, da diese in dieser Hinsicht unterschiedlich arbeiten. Schauen wir uns zuerst die DLL-Server an. Abbildung 8.3 zeigt den Programmfluß bei zwei Anwendungen und einem DLL-Server. Anwendung B läuft in ihrem eigenen Thread und ihre Abarbeitung hat keinen Einfluß auf Anwendung A oder den DLL-Server. Sie sehen jedoch, daß die Anwendung A nicht ihren eigenen Code ausführt, wenn sie Code in dem DLL-Server ausführt. Anders gesagt, DLL-Code läuft im gleichen Thread wie der der aufrufenden Anwendung.

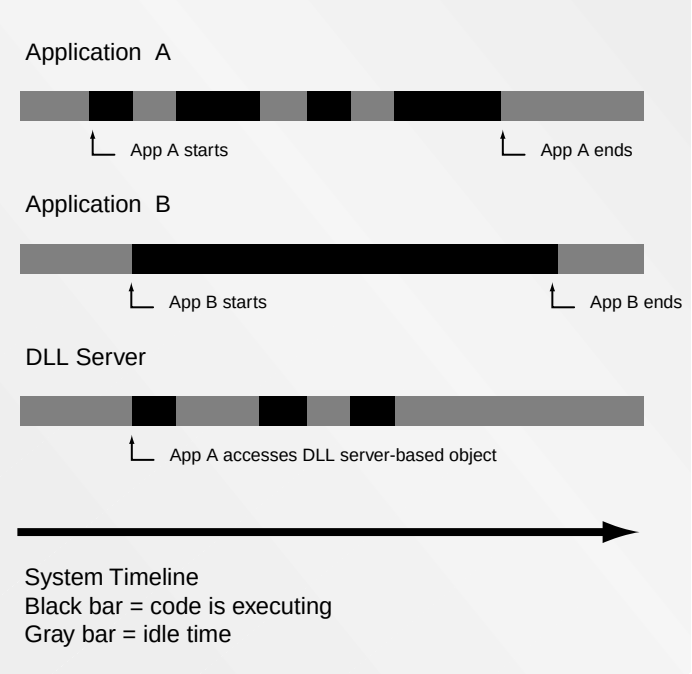

Abb. 8.3: Ausführungsfolge mit einem DLL-Server

Ein EXE-Server läuft als separater Prozeß. Ich habe Ihnen bereits die Auswirkungen auf die Performance erläutert, die daher rühren, daß er in seinem eigenen Prozeßraum läuft. Im Rahmen dieser Einführung in Multitasking können Sie sich wahrscheinlich vorstellen, was als nächstes kommen wird: Ein EXE-Server läuft obendrein in seinem eigenen Thread!

Dies hat zwei bedeutendere Auswirkungen: Erstens wird der EXE-Server blokkiert, wenn eine Anwendung eine seiner Funktionen aufruft, so daß andere Anwendungen solange nicht auf diesen EXE-Server zugreifen können (auf Multithreading-Server gehen wir später noch ein). Der Server wartet so lange, bis die Bearbeitung eines Methoden- oder Eigenschaften-Aufrufs abgeschlossen ist, bevor der nächste erfolgen kann, unabhängig davon, von welcher Anwendung aus der Aufruf erfolgt.

Zweitens kann eine Anwendung eine Hintergrundaufgabe starten, die von einem EXE-Server ausgeführt wird, während die Anwendung weiterhin ihren eigenen Code ausführt. Dies können Sie in Abbildung 8.4 sehen, wo der Code des EXE-Servers simultan zum Code der Anwendung A läuft.

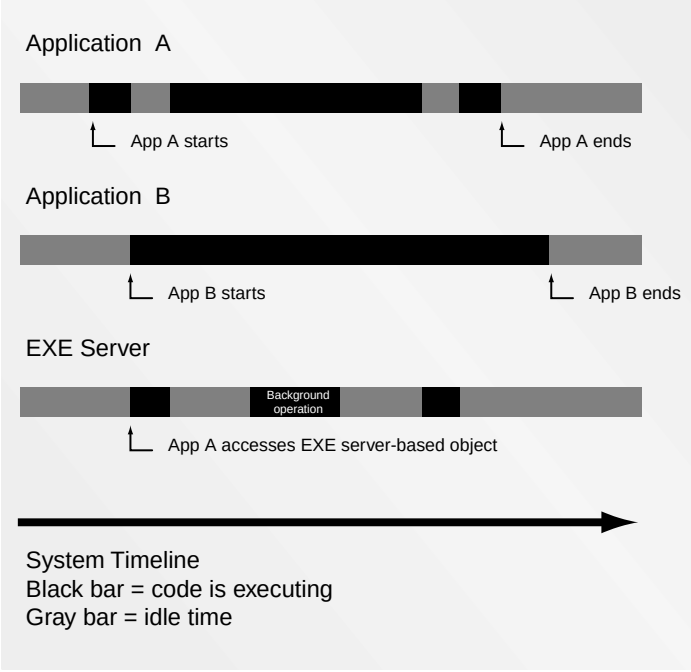

Abb. 8.4: Ausführungsfolge mit einem EXE-Server

Schauen wir uns das noch einmal aus einer anderen Perspektive an:

Ein in einem DLL-Server implementiertes Objekt läuft im Thread der aufrufenden Anwendung. Wenn 50 verschiedene Anwendungen alle dasselbe Objekt in einem DLL-Server verwenden, existiert jedes dieser 50 Objekte im Prozeßraum der jeweils aufrufenden Anwendung und läuft jeweils in deren Thread. Auch wenn sie alle den gleichen Server verwenden, besteht keinerlei Kommunikationsverbindung zwischen diesen Objekten, und sie kommen sich auch keinesfalls ins Gehege.

Ein in einem EXE-Server implementiertes Objekt läuft im Thread der Server-Anwendung. Wenn 50 verschiedene Anwendungen alle jeweils ein Objekt aus der gleichen Instanz eines single-threaded EXE-Servers anlegen, dann existiert jedes dieser Objekte im Prozeßraum des Servers und läuft im Thread des Servers. Während der Server einen Methoden- oder Eigenschaften-Aufruf durch eine Anwendung abarbeitet, kann er keinen weiteren Aufruf einer Methode oder Eigenschaft einer anderen Anwendung ausführen – der Thread führt bereits Code aufgrund des Aufrufs der ersten Anwendung aus. Die nachfolgenden Aufrufe werden in einer Warteschlange abgelegt und in einer vom Betriebssystem gewählten Reihenfolge ausgeführt.

Gibt es einen Weg, diese Blockade zu umgehen? Ja, eine Möglichkeit ist, für jedes anzulegende Objekt eine eigene Instanz des EXE-Servers zu starten. So läuft jedes Objekt in seinem eigenen Prozeß, hat somit seinen eigenen Prozeßraum und auch seinen eigenen Thread. Auch wenn dies in manchen Fällen sinnvoll sein mag, insbesondere dann, wenn dieses eine Objekt als Anfangspunkt eines komplexeren Objekt-Modells dient, ergibt dieser Ansatz jedoch eine ganze Menge an Overhead. Im Falle des Beispiels mit den 50 Objekten würden schließlich 50 Instanzen des EXE-Servers laufen, was mit Sicherheit Auswirkungen auf die System-Performance hätte.

Die andere Lösung wäre, einen EXE-Server im Multithreading-Betrieb laufen zu lassen. Ich weiß, ich werde Sie hier bezüglich dieses Themas auf ein späteres Kapitel vertrösten. Der Grund hierfür liegt darin, daß die Implementierung von Multithreading in Visual Basic nur in bestimmten Fällen eine sinnvolle Lösung bietet und auch etwas Kopfzerbrechen bei der Verwendung von globalen Variablen mit sich bringt. Es ist auch eine fortgeschrittenere Technik, die die Gefahr von Nebeneffekten mit sich bringt. Das Dilemma ist, daß ich kaum über Nebeneffekte sprechen kann, solange ich nicht einmal die »normalen« Effekte zu Ende besprochen habe. Wenn es Ihnen tatsächlich unter den Nägeln brennen sollte, mehr über Multithreading in Erfahrung zu bringen, dann finden Sie den Stoff dazu in Kapitel 14, »Multithreading«.

Der Ordner auf der Buch-CD zum Kapitel 8 enthält ein paar Programme, die diese Situation demonstrieren. Zunächst gibt es da zwei Server-Anwendungen. Beide Server implementieren ein öffentliches Objekt namens `TestObject1` mit dem folgenden Klassen-Code:

```
Option Explicit

Public Sub SlowOperation()
   Dim counter&
   For counter = 1 To 10000000
   Next counter
End Sub

Public Sub FastOperation()
End Sub
```

Das Projekt `MTTstSv1.vbp` implementiert dieses Objekt als ActiveX-EXE-Server mit dem Projekt-Namen `MTTestServer1`. Das Projekt `MTTstS1D` implementiert das Objekt als ActiveX-DLL-Server mit dem Projekt-Namen `MTTestServer1DLL`.

Die Klasse `TestObject1` legt zwei Funktionen offen. Die Methode `SlowOperation` führt eine Schleife mit einer sehr hohen Anzahl an Durchläufen aus, was eine zeitkonsumierende Operation simulieren soll. Die Methode `FastOperation` dagegen kehrt sofort zurück und demonstriert somit eine sehr schnelle Operation.

Ein Testprogramm (`MTTest1.vbp`) testet das Verhalten dieser Objekte. Das folgende Code-Listing enthält den Code für das Formular `frmMTTest1`, das ein Label-Control und vier Schaltflächen enthält. Jede Schaltfläche ruft eine Methode in einem der `TestObject1`-Objekte auf, erhöht nach der Rückkehr einen Zähler und zeigt dessen Stand im Label-Control an. Das Projekt legt zwei Objekte an. Beide heißen `TestObject1`, so daß Sie die Herkunft jeweils durch das Voranstellen des Objekt-Namens (mit ».« dazwischen) kennzeichnen müssen. Abbildung 8.5 zeigt die Gestaltung des Formulars.

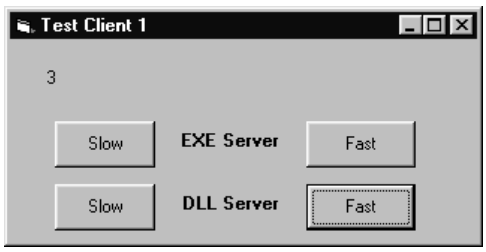

Abb. 8.5: Das Projekt MTTest1 in Aktion

```
' Threading Demonstrations-Programm #1
' Guide to the Perplexed
' Copyright (c) 1997, by Desaware Inc. All Rights Reserved

    Option Explicit
```

```
Dim counter&

' Wir können zwei verschiedene Test-Objekte
' referenzieren
Dim mtserve As New MTTestServer1.TestObject1
Dim mtserveDLL As New MTTestServer1DLL.TestObject1

Private Sub cmdFast_Click()
   mtserve.FastOperation
   counter = counter + 1
   lblCount.Caption = counter
End Sub

Private Sub cmdFastDLL_Click()
   mtserveDLL.FastOperation
   counter = counter + 1
   lblCount.Caption = counter
End Sub

Private Sub cmdSlow_Click()
   mtserve.SlowOperation
   counter = counter + 1
   lblCount.Caption = counter
End Sub

Private Sub cmdSlowDLL_Click()
   mtserveDLL.SlowOperation
   counter = counter + 1
   lblCount.Caption = counter
End Sub
```

Probieren Sie nun folgendes aus:

- Vergewissern Sie sich, daß die Server `MTTstS1D.DLL` und `MTTstSv1.EXE` registriert sind (verwenden Sie `regsvr32.EXE` zum Registrieren der DLL bzw. starten Sie einfach den EXE-Server einmal).
- Starten Sie zwei Instanzen des Testprogramms `MTTest1.EXE` und ordnen Sie sie auf dem Bildschirm nebeneinander an.
- Klicken Sie auf die Schaltfläche SLOW (EXE) bei einer der beiden Anwendungen, um den EXE-Server vorab zu laden. Klicken Sie nun noch einmal auf die Schaltfläche, um ein Gefühl dafür zu bekommen, wie lange die Ausführung dauert. Sie sollte schon ein paar Sekunden dauern, andernfalls sollten Sie einen höheren Schleifenzähler verwenden. Hier geht es nicht nur um den Unterschied alleine, sondern auch darum, daß genügend Zeit bleibt, andere Operationen auszuführen, während diese eine gerade in Bearbeitung ist.

- Klicken Sie auf die Schaltflächen SLOW und FAST (DLLs), um auch ein Gefühl für deren Geschwindigkeit zu bekommen. Wie Sie sehen, scheint die Performance die gleiche wie bei der EXE-Server-Version zu sein. Die war zu erwarten, da Sie den Marshaling-Overhead kaum wahrnehmen können.
- Klicken Sie nun auf die Schaltfläche SLOW (EXE) in der einen Instanz des Programms und klicken Sie dann sofort ein paarmal hintereinander auf die Schaltfläche FAST (EXE) in der anderen Instanz. Sie werden sehen, daß der Zähler dort nicht eher aktualisiert wird, ehe nicht die vorher begonnene langsame EXE-Ausführung beendet worden ist. Dies liegt an der Blockierung des Zugriffs auf den EXE-Server. Auch wenn jede Instanz ihr eigenes Objekt `TestObject1` hat, werden doch beide Objekte vom gleichen EXE-Server ausgeführt, der den Code nicht mehrmals parallel ausführen kann.

Nun klicken Sie einmal auf die Schaltfläche SLOW (DLL) in einer der Instanzen und sofort danach auf die Schaltfläche FAST (DLL) in der anderen Instanz. Sie werden sehen, daß der Zähler dieser Instanz sofort aktualisiert wird. Dies liegt daran, daß die Objekte beider Anwendungen im Thread ihrer Anwendung laufen.

Ich habe erwähnt, daß die Fähigkeit eines EXE-Servers, in einem eigenen Thread laufen zu können, Operationen im Hintergrund erlaubt. Ich habe aber bis jetzt noch nicht gezeigt, wie das geht. Im Prinzip sieht das so aus:

- Rufen Sie eine Methode in einem EXE-basierenden Objekt auf, die ein anderes Objekt initialisiert, das später ein Ereignis auslösen wird (ein Timer-Objekt beispielsweise).
- Der Aufruf der Methode kehrt zurück – der Thread der Anwendung läuft weiter.
- Einige Zeit später wird das Ereignis im Server-Objekt ausgelöst. Dieses Ereignis läuft im Thread des Servers zur gleichen Zeit, während die Anwendung selbst in ihrem eigenen Thread weiterläuft.

Auf dieses Thema gehe ich im nächstfolgenden Kapitel, »Erstellen und Testen von Komponenten«, im Detail ein. Unser `StockQuote`-Beispiel beruht auf dieser Technologie, die viel zum Entschluß beigetragen hat, einen EXE-Server anstelle eines DLL-Servers zu wählen.

Gemeinsame Nutzung von Ressourcen von mehreren Prozessen

Ein EXE-Server, der mehrfach instanzierte Objekte unterstützt, kann bei der Verwendung begrenzter System-Ressourcen nützlich sein, wie auch bei Ressourcen, die nur von einem Objekt zur gleichen Zeit verwendet werden können.

Vergegenwärtigen Sie sich wieder die Aufgabenstellung, einen Aktienkurs abzurufen. Ich bin noch nicht darauf eingegangen, wie diese Aufgabe konkret zu lösen wäre, doch nehmen wir vorläufig einmal an, daß dafür ein ActiveX-Control, ein Formular und irgendeine Kommunikations-Ressource benötigt wird.

Nun, dies sind nicht gerade schlanke Ressourcen. Sie werden kaum einen Unterschied in Ihrem System bemerken, wenn nur eine Instanz eines solchen Servers läuft. Doch können Sie sich vorstellen, daß ohne weiteres 20 verschiedene Anwendungen gleichzeitig, aus welchem Grund auch immer, Aktienkurse abfragen könnten. Wenn Sie einen DLL-Server verwenden würden, hätten Sie dann 20mal das gleiche Control auf 20mal dem gleichen Formular, und alle schlagen sich um die einzige vorhandene Kommunikationsverbindung. Selbst wenn der Durchsatz der Verbindung und der Server dahinter vollauf ausreichen mögen, ein einzelnes Control zu versorgen, würden die übrigen 19 Controls und Formulare derweil Speicher und Ressourcen blockieren und verschwenden.

Wenn Sie jedoch einen EXE-Server verwenden, der mehrere `StockQuote`-Objekte gleichzeitig verwalten kann, können all diese Objekte ein einziges Control und ein einziges Formular gemeinsam nutzen. Dieser Server könnte die Anfragen der einzelnen Objekte in eine Warteschlange einreihen und dem Objekt eine Mitteilung zukommen lassen, sobald der gewünschte Kurs verfügbar ist. Da der EXE-Server in seinem eigenen Thread läuft, kann er die Kurse im Hintergrund abrufen, ohne die Anwendungen zu blockieren, die seine Objekte verwenden. Da hier der Performance-kritische Faktor die relativ langsame Kommunikationsverbindung ist, fällt der durch das Marshaling entstehende Overhead bei der Verwendung eines Out-of-Process-EXE-Servers kaum noch ins Gewicht.

Aus all diesen Überlegungen ergibt sich also unterm Strich, daß das `StockQuote`-Objekt als EXE-Server zu implementieren wäre. In diesem Fall implementiert ein einziger Server alle benötigten `StockQuote`-Objekte.

8.2 Instanzierung

Jedes Klassen-Modul verfügt über eine Reihe von Eigenschaften, von denen an dieser Stelle zwei von näherem Interesse sind: der Klassen-Name und die Eigenschaft `Instancing`. `Instancing` hat weniger speziell mit unserem Projekt zu tun, doch hat diese Eigenschaft einen bedeutenden Einfluß auf das Verhalten von Objekten in einem Projekt. Außerdem gehört sie zu den weniger leicht zu verstehenden Eigenschaften – ich denke aber, Sie werden sich mit Ihrem bisher in diesem Buch gewonnenen Wissen ein ganzes Stück leichter tun.

Die Eigenschaft `Instancing` kann folgende Werte annehmen:

- `Private`
- `PublicNotCreatable`
- `SingleUse`
- `MultiUse`
- `GlobalMultiUse`
- `GlobalSingleUse`

Schauen wir uns diese Einstellungen der Reihe nach an, wenn auch nicht in dieser Reihenfolge – warum dies nicht, wird Ihnen sicher sehr bald klar werden.

8.2.1 Private

Auf `private` Objekte kann nur innerhalb einer Komponente oder Anwendung zugegriffen werden. Es ist zwar theoretisch möglich, ein privates Objekt anzulegen und es nach draußen weiterzugeben. Doch das wäre ein gefährliches Unterfangen, da Visual Basic nicht für eine stabile und saubere Referenzierung von außen garantiert. Weiterhin können andere Anwendungen private Objekte einer Komponente nicht von außen her anlegen.

Alle Objekte, die Sie innerhalb einer Anwendung oder einer Komponente verwenden, sollten Sie per Voreinstellung zunächst privat halten.

Private Objekte werden in allen ActiveX-Komponenten und in Standard-EXE-Programmen unterstützt.

8.2.2 PublicNotCreatable

Diese Objekte werden in der Typbibliothek der Komponente offengelegt und auf sie kann von Anwendungen, die die Komponente verwenden, zugegriffen werden. Doch kann nur die Komponente selbst solche Objekte anlegen und nach außen hin anbieten. Andersherum gesagt bedeutet dies, daß andere Anwendungen solche Objekte selbst nicht anlegen können. Hat jedoch eine Anwendung Zugriff auf ein anderes Objekt der Komponente, kann dieses Objekt ein `PublicNotCreatable` intern anlegen und an die Anwendung über eine Eigenschaft oder als Ergebnis eines Funktionsaufrufs nach außen reichen.

Dies wird in der Projekt-Gruppe `PubTest.vbg` gezeigt, die zwei Projekte enthält. Das eine Projekt ist ein ActiveX-DLL-Server-Projekt namens `PubTest.vbp`. Es enthält zwei Klassen. Diese Demo-Projekt-Gruppe kann komplett innerhalb der Visual-Basic-Entwicklungsumgebung gestartet werden.

Die eine Klasse heißt `PublicTest` und deren `Instancing`-Eigenschaft ist auf 5-MULTIUSE gesetzt. Dies bedeutet, daß die Klasse von außen her instanziert werden kann. Sie enthält folgenden Code:

```
Public Function GetOtherClass() As PublicNotCreatable
   Dim obj As New PublicNotCreatable
   Set GetOtherClass = obj
End Function
```

Die andere Klasse heißt `PublicNotCreatable` und deren `Instancing`-Eigenschaft ist auf 2-PUBLICNOTCREATABLE gesetzt. Sie enthält folgenden Code:

```
Public Sub Message()
   MsgBox "Public Not Creatable Object"
End Sub
```

Das andere Projekt heißt PubTest1.vbp und ist ein Standard-EXE-Projekt. Es enthält ein Formular mit einer Schaltfläche darauf, in deren Click-Ereignis folgender Code ausgeführt wird:

```
' Ein von außen nicht anlegbares Objekt über ein
' öffentliches Objekt erhalten
Private Sub cmdPublic_Click()
   Dim pubobj As PublicClass
   Dim pubNCobj As PublicNotCreatable
   ' Instanz der öffentlichen Klasse anlegen
   Set pubobj = New PublicClass
   ' Nun darüber eine Instanz der
   ' PublicNotCreatable-Klasse erhalten
   Set pubNCobj = pubobj.GetOtherClass()
   ' Zeigen, daß es geklappt hat
   pubNCobj.Message

   ' Bei folgendem würde der Kompiler streiken:
   ' Set pubNCobj As New PublicNotCreatable
End Sub
```

Die Methode GetOtherClass des öffentlichen Objekts legt ein PublicNotCreatable-Objekt an und übergibt es nach außen.

Dieser Typ einer Klasse ist sehr gebräuchlich. Mit ihm können Sie Objekt-Hierarchien erstellen, bei dem der Zugriff auf das Objekt-Modell über eine begrenzte Anzahl an von außen her anlegbaren Objekten beschränkt ist. Diese Objekte dienen gewissermaßen als Vermittler zu den übrigen Objekten in der Hierarchie. PublicNotCreatable-Klassen werden von allen ActiveX-Komponenten unterstützt, können jedoch in Standard-EXE-Projekten nicht verwendet werden.

8.2.3 SingleUse

Wir haben in diesem Kapitel bereits gesehen, daß ein EXE-Server Objekte auf zweierlei Weise unterstützen kann. Ein einzelner EXE-Server kann Objekte mehrfach unterstützen, wobei alle diese Objekte im gleichen Prozeß laufen. Oder es kann für jedes Objekt eine eigene Instanz des Servers gestartet werden, wobei jedes Objekt in seinem eigenen Prozeß läuft und damit zugleich auch in seinem eigenen Thread. Über die Instancing-Eigenschaft legen Sie fest, auf welche Weise der Server gestartet wird.

Wählen Sie SingleUse, wird für jedes neu angelegte Objekt eine neue Instanz des EXE-Servers gestartet. Dies ist ein ziemlich ineffizienter Weg, Objekte zu implementieren und er wird in der Regel nur für das oberste Objekt in einem komplexen Objekt-Modell gewählt. Die übrigen Objekte sind dann zumeist SingleNotCreatable. Wegen des hohen Overheads sollten Sie es möglichst vermeiden, allzuviele Objekte dieses Typs gleichzeitig auf einem System zu aktivieren.

SingleUse-Objekte können von einer Anwendung über den `New`-Operator oder über die Funktion `CreateObject` angelegt werden. Diese `Instancing`-Variante kann nur in ActiveX-EXE-Servern verwendet werden.

8.2.4 MultiUse

Ist diese `Instancing`-Einstellung gewählt, kann eine einzelne Instanz eines Servers beliebig viele Objekte der besagten Klasse liefern. Die Objekte können von Anwendungen über den `New`-Operator oder über die Funktion `CreateObject` angelegt werden.

Bei einem EXE-Server laufen alle Instanzen eines Objekts im Prozeßraum einer einzigen Instanz des Servers. Bei einem DLL-Server läuft jedes Objekt im Prozeßraum der aufrufenden Anwendung, wie wir ja bereits wissen.

`MultiUse`-Objekte können sowohl in ActiveX-DLL-Servern als auch in ActiveX-EXE-Servern verwendet werden.

8.2.5 GlobalMultiUse

Wenn Sie sich schon längere Zeit mit Visual Basic beschäftigen, werden Sie wahrscheinlich bereits etwas kennen, das sich *Globales Objekt* nennt, auch wenn Ihnen dieser Begriff nicht besonders geläufig sein sollte. Wenn Sie beispielsweise ein Formular namens `Form1` in Ihre Anwendung einfügen, können Sie mit einem Aufruf nach dem Muster `Form1.Methode` ganz einfach darauf zugreifen. So öffnet etwa die Anweisung `Form1.Show` das Formular.

Ebenso können Sie direkt auf das globale `Printer`-Objekt zugreifen. Sie können das Objekt immer referenzieren, ohne vorher ausdrücklich eine Instanz davon anzulegen. Sie brauchen nicht erst ein `Printer`- oder ein `Form1`-Objekt zu instanzieren, um es verwenden zu können.

In Visual Basic können Sie auch Ihre eigenen globalen Objekte erstellen.

Die Projekt-Gruppe `GlblTest.vbg` enthält zwei Projekte, die das Erstellen und Verwenden von globalen Objekten demonstrieren. Diese Demo-Projekt-Gruppe kann komplett innerhalb der Visual-Basic-Entwicklungsumgebung gestartet werden. Das Projekt `GlobalTest` (`GlblTst.vbp`) enthält die Klasse `MyGlobalClass`, deren `Instancing`-Eigenschaft auf `GlobalMultiUse` gesetzt ist. Die Klasse enthält folgenden Code:

```
Public Sub Message()
   MsgBox "I've been accessed"
End Sub

Private Sub Class_Initialize()
   Debug.Print "MyGlobalClassObject created"
End Sub
```

```
Private Sub Class_Terminate()
   Debug.Print "MyGlobalClassObject deleted"
End Sub
```

Die Ereignisse `Initialize` und `Terminate` werden zur Verfolgung der Lebensdauer des Objekts verwendet.

Das Projekt `GlobalTestClient` enthält eine Form mit zwei Schaltflächen. Hier der Code des Formulars:

```
' Demonstration der global Instanzierung
Private Sub cmdGlobal_Click()
   ' Kurze Referenzierung - funktioniert!
   Message
   ' vollständige Referenzierung - funktioniert ebenso!
   GlobalTest.Message
End Sub

' Wie üblich geht es natürlich auch
Private Sub cmdObject_Click()
   Dim gbtest As New MyGlobalClass
   gbtest.Message
End Sub
```

Wie Sie sehen, können Sie auf das Objekt sowohl über eine vollständige Referenzierung, die sogenannte *qualifizierte* Referenzierung einer Methode zugreifen (`GlobalText.Message`), oder einfach über den puren Namen der Methode. In beiden Fällen instanziert Visual Basic intern eine Instanz des Objekts, sobald Sie darauf zugreifen. Diese Instanz lebt daraufhin offensichtlich genauso lange wie die ganze Anwendung und wird bei allen globalen Zugriffen innerhalb der Anwendung verwendet.

Objekte, die als `GlobalMultiUse` gesetzt sind, haben die gleichen Merkmale wie im gleichen Prozeß instanzierte Objekte, auf die allerdings global zugegriffen werden kann. Dieser Instanzierungs-Typ wird von ActiveX-EXE-Servern und von ActiveX-DLL-Servern unterstützt.

8.2.6 GlobalSingleUse

Dieser Instanzierungs-Typ ist nun ganz einfach zu verstehen. Er entspricht exakt der `SingleUse`-Variante, nur daß das Objekt nun wie bei `GlobalMultiUse` global ist.

Dieser Instanzierungs-Typ wird nur von ActiveX-EXE-Servern unterstützt.

Nun können wir uns einigen interessanten Aspekten im Zusammenhang mit diesen Instanzierungs-Varianten zuwenden.

8.2.7 Varianten der Instanzierung

Ein wichtiger anzumerkender Punkt ist, daß sich die `Instancing`-Eigenschaft nur darauf auswirkt, wie ein Objekt offengelegt wird und wie es extern verwendet werden kann. Innerhalb einer Komponente oder Anwendung können alle Objekte unbeachtet der Einstellung der `Instancing`-Eigenschaft ohne Beschränkungen verwendet werden. Dies öffnet Tür und Tor für einige interessante Möglichkeiten.

Das Projekt `EXESingle` (`EXESgle.vbp`) enthält eine einzige Klasse namens `EXESingleUse`, deren `Instancing`-Eigenschaft auf `SingleUse` gesetzt ist. Vergewissern Sie sich, daß die ausführbare Datei `EXESgle.EXE` registriert ist, bevor sie das Test-Programm starten (ein Versuch, die hier zu demonstrierende Funktionalität innerhalb der Visual-Basic-Entwicklungsumgebung zu erleben, wird zwecklos sein, da die Entwicklungsumgebung nur ein einziges `SingleUse`-Objekt zur gleichen Zeit starten kann). Die Klasse `EXESingleUse` enthält folgenden Code:

```
' Guide to the Perplexed
' SingleUse-EXE-Beispiel
' Copyright (c) 1997 by Desaware Inc.
' All Rights Reserved

Option Explicit

Private Declare Function GetCurrentProcessId _
 Lib "Kernel32" () As Long

Public Sub Message()
   MsgBox "Accessed EXESingle use in process " & _
    Hex$(GetCurrentProcessId())
End Sub

' Anlegen und Zurückgeben
' einer neuen Instanz des Objekts
Public Function GetNewObject() As EXESingleUse
   Dim newobj As New EXESingleUse
   Set GetNewObject = newobj
End Function
```

Die Message-Anweisung in diesem Beispiel zeigt die eindeutige Prozeß-Kennung (`ProcessID`) des Prozesses an, in dem die Objekt-Methode ausgeführt wird. Daraus ist leicht zu ersehen, ob zwei Objekte von demselben Server instanziert werden oder nicht.

Das Projekt `ExeSingleTest` (`EXESgTst.vbg`) enthält ein Formular mit zwei Schaltflächen. Im `Click`-Ereignis dieser Schaltflächen wird folgender Code ausgeführt:

```
' Guide to the Perplexed
' SingleUse-Instanz Test-Programm
```

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

Option Explicit

Private Sub cmdTest1_Click()
   Dim newobj1 As New EXESingleUse
   Dim newobj2 As New EXESingleUse
   ' Diese Objekte laufen in verschiedenen Prozessen
   newobj1.Message
   newobj2.Message
End Sub

Private Sub cmdTest2_Click()
   Dim newobj1 As New EXESingleUse
   Dim newobj2 As EXESingleUse
   Set newobj2 = newobj1.GetNewObject()
   ' Beide Objekte laufen in demselben Prozeß!
   newobj1.Message
   newobj2.Message
End Sub
```

Wenn Sie die Schaltfläche TEST1 betätigen, werden zwei verschiedene Objekte instanziert. Wie Sie sehen werden, läuft jedes Objekt in einer eigenen Instanz des Servers – das geht aus den unterschiedlichen Prozeß-Kennungen hervor.

Wenn Sie jedoch die Schaltfläche TEST2 betätigen, werden Sie feststellen, daß beide Objekte in demselben Prozeß laufen. Wieso das denn nun? Das liegt daran, daß die vom Objekt `newobj1` in der `GetNewObject`-Methode angelegte zweite Instanz im gleichen Prozeßraum angelegt wird (wie bei einer einfachen, gewöhnlichen Klasse). Da diese Klasse jedoch öffentlich ist, kann sie als Funktions-Ergebnis an die aufrufende Anwendung übergeben werden.

Diese Technik gibt Ihnen eine weiterreichende Kontrolle über die Instanzierung von Objekten im Rahmen Ihres Objekt-Modells, wenn Sie `SingleUse`-Objekte in ActiveX-EXE-Servern verwenden.

8.2.8 Komplikationen bei gemeinsamer Verwendung von Klassen-Modulen in mehreren Projekten

Die `Instancing`-Eigenschaft berührt die gemeinsame Nutzung von Klassen-Modulen in mehreren Projekten. Weiter oben in diesem Kapitel haben wir mehrere mögliche Mechanismen der gemeinsamen Verwendung von Code untersucht, wie etwa das Kopieren, die gemeinsame Verwendung von Klassen in Projekten und das Erstellen von Komponenten.

Angenommen, Sie haben eine ActiveX-DLL erstellt, die unter anderem ein perfektes Text-Validierungs-Objekt enthält. Dieses Objekt wird von der DLL offengelegt und daher ist seine `Instancing`-Eigenschaft auf irgendeinen Wert außer `Private` gesetzt.

Später arbeiten Sie dann an einem Standard-EXE-Projekt. Sie stellen fest, daß Sie dieses Text-Validierungs-Objekt in diesem Projekt bestens gebrauchen könnten. Jedoch ist es das einzige Objekt, das Sie von der ActiveX-DLL gebrauchen können, so daß Sie es vermeiden möchten, die ganze DLL mitliefern zu müssen. Sie denken, daß es wohl das einfachste wäre, die besagte Klasse einfach direkt in Ihr Standard-EXE-Projekt einzufügen.

Wenn Sie dies jedoch tun, wird Visual Basic die `Instancing`-Eigenschaft der Klasse auf `Private` setzen (und Sie mit einer »höchst informativen« Nachricht darüber in Kenntnis setzen). In diesem Projekt wird die Klasse ihre Funktion bestens erfüllen, doch wenn Sie das Projekt speichern, wird diese Änderung mitgesichert. Wenn Sie nun das nächste Mal die ActiveX-DLL neu kompilieren wollen, wird das nicht gelingen – wahrscheinlich werden Sie es mit einem Kompatibilitätsproblem zu tun bekommen (später mehr dazu).

Wie können Sie nun eine Klasse mit einer bestimmten Einstellung der `Instancing`-Eigenschaft problemlos in einem Projekt verwenden, das diese Einstellung gerade nicht unterstützt?

Das geht gar nicht. Sie müssen schon eine zweite Kopie des Klassen-Moduls verwenden. Oder die Klasse in einem Komponenten-Server implementieren und so Ihrem Projekt zur Verfügung stellen. *(Ergänzung d. Übersetzers: Speichern Sie eine Kopie einer Klasse, bei der dieses `Instancing`-Problem auftreten könnte, im Ordner \Templates\Classes Ihres Visual Basic-Ordners und laden Sie diese Vorlage in Ihre weiteren Projekte. Somit wird wenigstens die Gefahr gebannt, daß die `Instancing-Einstellung` der originalen Klasse versehentlich überschrieben werden kann. Beim Speichern verlangt dann Visual-Basic nämlich von Ihnen die Angabe eines neuen, eigenen Speicherorts für die Klasse – die Vorlage kann so nicht überschrieben werden und ihr ist es egal, wie Sie die Instancing-Eigenschaft der auf diese Weise in ein Projekt eingebundenen Kopie setzen.)*

Wenn Sie das Problem früh genug absehen können (was der Fall sein sollte, wenn Sie dem Entwurf Ihres Objekt-Modells genügend Aufmerksamkeit geschenkt haben), läßt es sich im Handumdrehen lösen. Legen Sie eine private Klasse an, die die benötigte Funktionalität implementiert. In Ihrer ActiveX-Komponente greifen Sie dann auf die Technik der Aggregation zurück, um eine neue öffentliche Klasse zu erstellen, die als Hülle um die private Klasse dient.

8.3 Projekt-Eigenschaften

Nachdem Sie den Projekttyp Ihrer Komponente festgelegt und sich ausgiebig mit dem Objekt-Modell beschäftigt haben, können Sie nun darangehen, das Projekt zu erstellen. Zugegeben, Sie müssen nicht um jeden Preis zuerst diesen Entwurfsprozeß hinter sich bringen. Sie können den Projekttyp jederzeit nachträglich ändern. Dies wird jedoch Visual Basic dazu veranlassen, die Einstellungen der `Instancing`-Eigenschaft jeder öffentlichen Klasse notfalls Projekttyp-verträglich eigenmächtig anzupassen.

Dies führt uns zu den Eigenschaften eines Projekts. Außer bei wirklich trivialen Projekten werden Sie diesen Eigenschaften einige Aufmerksamkeit zukommen lassen müssen.

Zugriff auf diese Eigenschaften erhalten Sie über den Menüpunkt EIGENSCHAFTEN VON PROJEKT... im PROJEKT-Menü oder im Kontextmenü nach einem Klick mit der rechten Maustaste auf das Projekt im Projekt-Fenster. Achten Sie darauf, daß Sie dabei auch wirklich das gewünschte Projekt erwischen, wenn Sie mehrere Projekte als Projekt-Gruppe geladen haben. Es ist ziemlich frustrierend, nach dem Schließen des Eigenschaften-Dialogs festzustellen, daß die ganze Mühe umsonst war, weil Sie das falsche Projekt gewählt hatten ...

8.3.1 Allgemeine Eigenschaften

In Abbildung 8.6 sehen Sie das Allgemein-Register des Projekt-Eigenschaften-Dialogs. Gehen wir die einzelnen Punkte der Reihe nach durch.

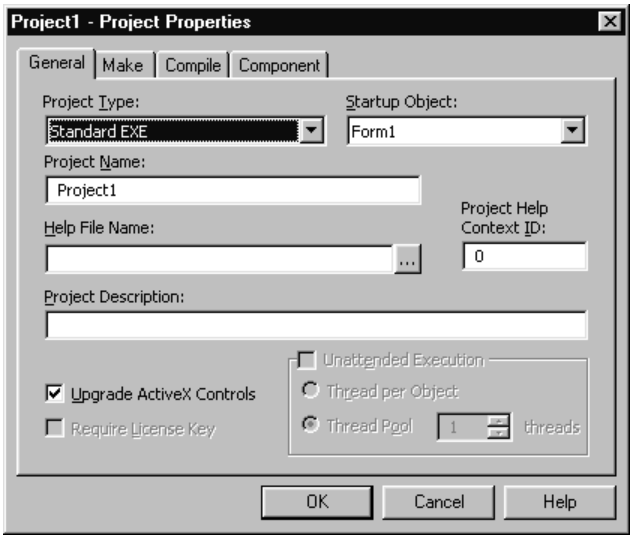

Abb. 8.6: Allgemeine Projekt-Eigenschaften

Projekttyp

Die Wahl des Projekttyps ist eines der Hauptthemen dieses Kapitels. Jedes weitere Wort dazu an dieser Stelle erübrigt sich wohl.

Startobjekt

Sie können je nach Typ der zu erstellenden Komponente und abhängig von den gerade in Ihrem Projekt vorhandenen Modulen das Startobjekt auswählen. Damit erhalten Sie ein wenig Kontrolle über das, was beim Start der Anwendung bzw. Komponente geschieht.

Standard-EXE-Projekte erfordern auf jeden Fall ein Startobjekt. Dies kann jedes beliebige Formular oder eine `Sub Main`-Prozedur sein. Legen Sie ein Formular fest, wird es geladen und es werden die Ereignis-Prozeduren `Initialize` und `Load` ausgeführt. Legen Sie `Sub Main` fest, wird die Main-Prozedur in einem Standard-Modul des Projekts ausgeführt. In dieser Prozedur können beispielsweise weitere Formulare je nach Bedarf angelegt werden.

ActiveX-Server können ohne Startobjekt auskommen – wählen Sie dazu die Option (KEINE). Als erster Code beim Start der Komponente wird dann der Code im `Initialize`-Ereignis der allerersten angelegten Klasse oder, wenn dieses dort nicht implementiert ist, der Code der zuerst in dieser Klasse aufgerufenen Eigenschaft oder Methode ausgeführt.

Auch für ActiveX-Server können Sie `Sub Main` als Startobjekt festlegen. Die `Sub Main`-Prozedur wird dann vor allen `Initialize`-Ereignissen von Klassen ausgeführt. Denken Sie aber daran, daß der Code der `Sub Main`-Prozedur nicht ausgeführt wird, bevor nicht ein Objekt des Servers angelegt wird. Nachdem der Server einmal gestartet worden ist, wird `Sub Main` kein zweites Mal ausgeführt, es sei denn, er wird vollständig beendet und neu gestartet. Dieses Verhalten ist jedoch nicht eindeutig vorherbestimmbar, da Visual Basic einen Server nicht unbedingt sofort nach der Freigabe seines letzten Objekts entlädt.

In Visual Basic kann bei einem ActiveX-EXE-Server ein Formular nicht als Startobjekt festgelegt werden. Möchten Sie ein Formular nach dem Start anzeigen, wählen Sie `Sub Main` als Startobjekt. Darin können Sie das gewünschte Formular öffnen. Sie können über die `StartMode`-Eigenschaft des `App`-Objekts feststellen, ob die Komponente als Server oder ob sie eigenständig gestartet worden ist.

Zu Testzwecken in der Visual Basic-Entwicklungsumgebung können Sie den passenden Startmodus im KOMPONENTE-Register der Projekt-Eigenschaften festsetzen. Denken Sie auch hier wieder daran, daß eine `Sub Main`-Prozedur erst ausgeführt wird, wenn das erste Objekt vom Server angelegt wird.

Projektname

Dies ist eine der wichtigsten festzulegenden Eigenschaften, die Sie unbedingt bei jedem Projekt setzen sollten. Denn diese Eigenschaft enthält den Namen der Komponente in der Typbibliothek. Wenn Sie Ihr Projekt beispielsweise `MeinPro-`

jekt nennen und es die Objekte MeinObjekt1 und MeinObjekt2 enthält, werden diese Objekte per Programm-Code in der Form MeinProjekt.MeinObjekt1 und MeinProjekt.MeinObjekt2 referenziert. Sind diese Objekte öffentlich, kann eine solche Referenzbezeichnung (die sogenannte *ProgID*) an die Funktion CreateObject übergeben werden, um Instanzen des betreffenden Objekts anzulegen:

```
Set newobj = CreateObject("MyProject.MyObject1").
```

Ich kann Ihnen für die Wahl des Projektnamens nur den Rat geben, einen einigermaßen verständlichen, beschreibenden und auch eindeutigen Namen zu geben. Der StockQuote-Server, der im nächsten Kapitel beschrieben wird, trägt beispielsweise den Projektnamen »StockMonitor«.

Projektbeschreibung

Sie werden sicher denken, daß dieses Feld lediglich Dokumentationszwecken dienen soll. Doch es ist von größerer Bedeutung. Den Inhalt dieses Feldes verwenden Objekt-Kataloge (Objekt-Browser) zur Beschreibung einer Komponente. Immer wenn Sie über einen Objekt-Browser einen Verweis auf eine Komponente oder ein Control einfügen möchten, werden Sie diesen Namen zu Gesicht bekommen.

Der Visual-Basic-Objekt-Katalog und der Verweise-Dialog sortiert Komponenten in alphabetischer Folge der Beschreibung – nur wenn eine Beschreibung fehlt, wird statt dessen der Projektname verwendet (trotz der ansonsten alphabetischen Reihenfolge werden im Verweise-Dialog bereits referenzierte Komponenten an erster Stelle ganz oben angeführt). Ein Seiteneffekt Ihrer Wahl für die Beschreibung ist damit die relative Position Ihrer Komponente in diesen Listen.

Heißt das nun, daß Sie Ihre Komponenten mit Beschreibungen wie »AAA-Alpha das supertolle Control« versehen sollen? Wenn Sie glauben sollten, daß Programmierer blindlings die Komponente wählen, die in einer Liste an erster Stelle steht, dann tun Sie das ruhig. Mit Verlaub gesagt, ich putze Controls mit derartigen Namen auf der Stelle von meiner Festplatte, ohne auch nur eine Sekunde zu zögern ...

Mit der Zeit hat sich die recht praktische Konvention entwickelt, der Beschreibung den Namen des Herstellers voranzustellen. So lautet das erste Wort nahezu aller Objekte und Controls von Microsoft eben »Microsoft«, und daher erscheinen alle in Objekt-Katalogen und Verweis-/Komponenten-Listen aufeinanderfolgend. Alle Projekte und Controls dieses Buchs sind mit einem von zwei Präfixen versehen. Beginnt die Beschreibung mit »gtp« (für *Guide to the Perplexed*, dem originalen Untertitel dieses Buchs), handelt es sich um ein triviales Projekt, das speziell für dieses Buch zur Demonstration von diversen Techniken entwickelt worden ist. Darüber hinaus dürften diese Projekte wohl nur wenig Nutzen haben. Beginnt die Beschreibung jedoch mit *Desaware*, handelt es sich um eine Komponente oder ein Control, das zum Teil oder komplett aus Desawares ActiveX Gallimaufry stammt, einem separaten Produkt.

Name der Hilfedatei – Kontext-ID für Projekthilfe

Unter NAME DER HILFEDATEI wird der Name der Hilfedatei für das Projekt angegeben. Zu keinem der Projekte zu diesem Buch gibt es eine Hilfedatei. Zu einer kommerziell vertriebenen Komponente sollte es allerdings eine Hilfedatei geben. Doch zum Themenkreis dieses Buches gehört nicht die Entwicklung von Hilfedateien.

Die KONTEXT-ID FÜR PROJEKTHILFE sollte die Kontext-ID für die Hilfeanfrage zu der Komponente aus einem Objekt-Browser enthalten.

ActiveX-Steuerelemente aktualisieren

Diese Option weist Visual Basic an, eventuell veraltete ActiveX-Controls bei einem späteren Laden des Projekts zu aktualisieren.

Lizenzierungsschlüssel erforderlich

Die Lizenzierung betrifft ActiveX-Controls und wird in Kapitel 26, »Lizenzierung und Auslieferung«, behandelt.

Unbeaufsichtigte Ausführung

Diese Option zeigt an, daß die Komponente keine Benutzeroberfläche haben soll. Sie dürfen keine Formulare oder Messageboxen anzeigen, wenn diese Option gesetzt ist. Wichtiger zu wissen ist jedoch, daß Fehler (wie Laufzeitfehler), die normalerweise mit einer MessageBox gemeldet werden, dies dann auch nicht mehr tun. Statt dessen schreibt Visual Basic eine Nachricht in das System-Event-Log.

Threading-Modell

Die Optionen, die hier zu setzen sind, werden in Kapitel 14, »Multithreading«, besprochen.

In Speicher erhalten

Normalerweise entlädt Visual Basic eine DLL, sobald alle Objekte der DLL freigegeben worden sind. Dies kann beträchtliche Auswirkungen auf die Performance haben, wenn die Objekte häufig angelegt und freigegeben werden. Bei IIS-Anwendungen ist dies beispielsweise oft der Fall. Ist diese Option gesetzt, verbleibt eine DLL solange im Arbeitsspeicher, bis der Prozeß, also die Anwendung selbst, terminiert. Dies liefert Performance-Vorteile auf Kosten eines höheren Bedarfs an Systemressourcen.

8.3.2 Registerkarte Erstellen

In Abbildung 8.7 sehen Sie die Registerkarte ERSTELLEN. Die hier zu plazierenden Informationen werden verwendet, wenn Visual Basic die Komponente kompiliert.

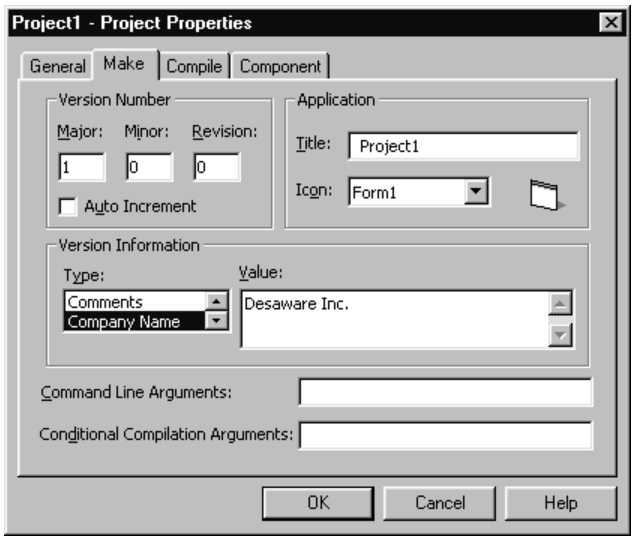

Abb. 8.7: Die Registerkarte Erstellen

Versionsnummer – Automatisch erhöhen

Wenn Sie das erste Mal ein Control oder eine Komponente kompilieren, sollten Sie die Option AUTOMATISCH ERHÖHEN setzen. Dann wird die Revisionsnummer (auch »Build« genannt) jedesmal automatisch inkrementiert, wenn das Projekt kompiliert wird. Meine Meinung ist, daß Microsoft diese Option als Voreinstellung hätte setzen sollen. Die Versionsnummer wird von Installationsprogrammen zur Prüfung verwendet, ob eine zu installierende Version neuer als eine eventuell bereits vorhandene ist.

Anwendung – Name

Der Name einer Anwendung ist etwas anderes als der Projektname. Letzterer wird Bestandteil der ProgIDs der enthaltenen Objekte. Der Name ist dagegen der Name, unter dem das Betriebssystem die Anwendung kennt und der in der Taskliste des Systems erscheint. Er wird auch von der Eigenschaft `Title` des `App`-Objekts in Visual Basic zurückgegeben. Meistens wird als Name entweder der Projektname oder der Dateiname der Anwendung verwendet. Von Bedeutung ist der Name in erster Linie für eigenständige Anwendungen (Standard-EXE-Anwendungen) und für EXE-Server.

Anwendung – Symbol

Dies ist das Symbol (Icon), das die Anwendung in der Taskbar, auf dem Desktop oder im Explorer repräsentiert (abhängig von den jeweiligen Systemeinstellungen). Es wird bei Standard-EXE-Anwendungen oder bei EXE-Servern verwendet.

Versionsinformationen

In der Listbox TYP können Sie unter einer ganzen Reihe von Strings wählen. Unter WERT tragen Sie dann den Inhalt zu jedem String ein. Es empfiehlt sich sehr, die Eintragungen hier vorzunehmen. Sie haben jedoch keinerlei Einfluß auf das Verhalten oder die Funktion einer Komponente bzw. Anwendung. Diese Versionsinformationen können beispielsweise vom Windows-Explorer oder vom Dateimanager in den Eigenschaften der ausführbaren Datei angezeigt werden.

Befehlszeilenargumente

Ein Eintrag hier wird nur wirksam, wenn die Anwendung in der Entwicklungsumgebung läuft. Er simuliert die Übergabe von Befehlszeilenargumenten.

Argumente für bedingte Kompilierung

Hier können Sie Konstanten für eine bedingte Kompilierung angeben. Dies bietet fantastische Möglichkeiten beim Debuggen eines Projekts – mit diesen Konstanten kann Debug-Code eingeschaltet und für die Auslieferungsversion einfach ausgeschaltet werden.

Noch wichtiger waren diese Konstanten in Visual Basic 4, da auf diese Weise Anwendungen sowohl für 16-Bit- als auch für 32-Bit-Plattformen mit einer einzigen Code-Basis geschrieben werden konnten. Die späteren Visual-Basic-Versionen unterstützen jedoch die 16-Bit-Kompilierung nicht mehr, so daß dieser spezielle Verwendungszweck hinfällig geworden ist.

Informationen zu nicht verwendeten ActiveX-Steuerelementen entfernen

Jeder zu Ihrem Projekt hinzugefügte Verweis kostet ein wenig Overhead. Normalerweise entfernt Visual Basic diese Informationen für Controls, die zwar in der Werkzeugleiste im Projekt enthalten sind, aber auf keinem Formular (`UserControl`, `UserDocument`, `PropertyPage`) verwendet werden.

Seit Visual Basic 6 ist es möglich, ActiveX-Controls dynamisch in ein Formular einzufügen. Dazu benötigt Visual Basic bestimmte Informationen zu diesen Controls, auch wenn es so scheinen mag, daß sie in der zu kompilierenden Version nicht verwendet werden. Das Setzen dieser Option zwingt Visual Basic dazu, die Informationen mitzukompilieren, so daß ActiveX-Controls zur Laufzeit angelegt werden können.

8.3.3 Register Kompilieren

In Abbildung 8.8 sehen Sie das Register KOMPILIEREN der Projekt-Eigenschaften. Die Einstellungen hier werden von Visual Basic beim Kompilieren eines Projekts verwendet. Schauen wir uns die Einstellungen in der Reihenfolge ihrer Bedeutung an.

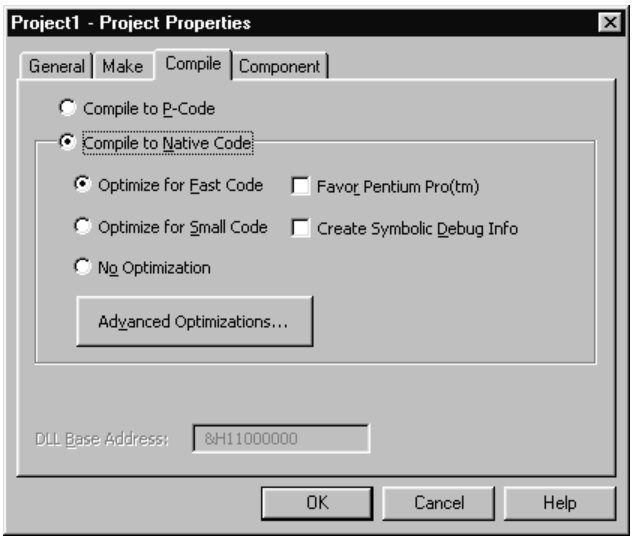

Abb. 8.8: Das Register Kompilieren der Projekt-Eigenschaften

DLL-Basisadresse

Bevor Sie eine DLL-basierte Komponente ausliefern (dazu gehören auch ActiveX-Controls), sollten Sie den hier eingetragenen Wert in irgendeinen Wert zwischen &H10000000 und &H80000000 ändern. Wählen Sie dabei einen glatten 64K-Wert, ein Wert, bei dem die vier letzten (rechten) Stellen alle 0 sind. Microsoft schlägt vor, eine zufällige Basisadresse in diesem Bereich für alle Ihre Controls zu wählen, und dann von dieser Adresse an weiterhin aufsteigende Werte zu verwenden – keine schlechte Idee.

Der Grund hierfür ist folgender: Wenn Windows eine DLL lädt, wird zunächst versucht, diese an der angegebenen Basisadresse zu laden. Ist dieser Speicherbereich im Prozeßraum der Anwendung, die die DLL lädt, verfügbar, wird die DLL äußerst schnell geladen. Belegt jedoch bereits eine andere DLL den angegebenen Speicherbereich, muß Windows die DLL-Daten und den Code an eine andere Stelle verschieben. Dieser Vorgang kostet Zeit und zugleich verhindert Windows die gemeinsame Nutzung des DLLs durch andere Anwendungen.

Da viele Visual Basic-Programmierer die Stelle der Dokumentation, an der dies erklärt wird, noch nie gelesen haben (genausowenig, wie dieses Buch), können Sie darauf wetten, daß die allermeisten mit Visual Basic erstellten Komponenten an der voreingestellten Basisadresse &H11000000 geladen werden wollen. Die Wahrscheinlichkeit, daß Windows die besagte nachteilige Verschiebung vornehmen muß, ist also extrem hoch.

Kompilieren zu P-Code / Kompilieren zu System-Code (Native Code)

Die erste Reaktion auf die Möglichkeit, Visual-Basic-Anwendungen in Native Code zu kompilieren, mag Freude oder Erleichterungsseufzer ausgelöst haben. Ihre Reaktion könnte aber nach einem näheren Test auch leicht in Depression umschlagen.

Wenn nämlich Ihre Anwendung starken Gebrauch von Visual-Basic-eigenen Funktionen, von API-Aufrufen, von anderen externen Controls oder von Datenbankzugriffen macht, wird es Sie womöglich schockieren, daß der Vorteil von Native Code verschwindend gering ist. Tatsächlich werden Sie am Ende lediglich größere EXE-Dateien erhalten.

Viele Anwendungen verbringen nur wenig Zeit in der Abarbeitung von P-Code (dem Zwischencode, der von Visual Basic interpretiert wird), so daß der Geschwindigkeitsvorteil sich nicht auswirken täte, wenn dieser Code in Native Code kompiliert würde. Microsoft hat selbst eine ganze Reihe von Anwendungen untersucht und dabei festgestellt, daß diese Anwendungen selten mehr als 5% der Zeit P-Code ausführen. Die übrige Zeit wird in Aufrufen der Support-Dateien verbracht (Visual Basic-Runtime-Dateien usw.). Dies überrascht mich nicht.

Wenn in Ihrem Code jedoch umfangreiche Berechnungen durchgeführt werden, sollten Sie Native-Code wählen – dann kann der Zeitgewinn beträchtlich sein. Dies gilt besonders für Komponenten wie ActiveX-DLLs, ActiveX-EXE-Server und ActiveX-Controls, wenn diese intern umfangreiche Berechnungen ausführen.

Mein Vorschlag lautet, eine Komponente in beiden Varianten zu kompilieren. Stellen Sie keinen wesentlichen Unterschied fest, sollten Sie die Variante wählen, die in einer kleineren ausführbaren Datei mündet.

Wie einige Visual-Basic-Programmierer sind auch Sie vielleicht enttäuscht, daß Sie nach wie vor eine große Runtime-Datei zusammen mit Ihren Anwendungen und Komponenten ausliefern müssen. Bevor Sie sich jedoch frustriert von Visual Basic abwenden, lassen Sie mich zwei bedenkenswerte Punkte anführen:

- Gäbe es die Visual-Basic-Runtime-Datei nicht, wäre die ausführbare Datei selbst der trivialsten Visual-Basic-Anwendung einige hundert Kilobyte groß, wie das beispielsweise bei Visual-C++ Programmen der Fall ist, die statisch mit den MFC-Bibliotheken (Microsoft Foundation Classes) gelinkt sind.
- Die meisten Visual-C++ Komponenten erfordern sowohl die C++-Runtime als auch die MFC-Runtime. Visual Basic bietet Ihnen dagegen nicht ganz so viele Möglichkeiten, wie sie sich daraus ergeben, aber der Ansatz hier ist in dieser Hinsicht nicht ganz unbegründet.

Individuelle Native Code-Kompileroptionen

Wenn Ihnen ein externer Debugger zur Verfügung steht (wie etwa mit Visual C++) und Sie Debug-Informationen in Ihre ausführbaren Dateien einbinden möchten, setzen Sie die Option DEBUG-INFORMATIONEN FÜR SYMBOLISCHEN

DEBUGGER GENERIEREN und die Option KEINE OPTIMIERUNG. Dann können Sie Ihre mit Visual Basic entwickelte Komponente auch bei der Verwendung mit einer Visual-C++ Komponente debuggen.

Ansonsten können Sie noch nach Belieben die Option CODE-AUSFÜHRUNGSGESCHWINDIGKEIT OPTIMIEREN oder die Option CODE-GRÖSSE OPTIMIEREN setzen.

Derzeit neige ich noch dazu, die Option FÜR PENTIUM PRO(TM) OPTIMIEREN zu vermeiden – vielleicht im nächsten Jahr ...

Weitere Optimierungen – Native Code

In der Online-Hilfe finden Sie zu diesen Optionen nähere Details.

Ich empfehle, alle diese Optimierungen so lange außer acht zu lassen, bis Sie Ihre Komponente gründlich getestet und fehlerbereinigt haben. Setzen Sie dazu in der Visual-Basic-Entwicklungsumgebung die Option BEI JEDEM FEHLER UNTERBRECHEN, damit nicht versehentlich Überlauffehler u.ä. von einer Fehlerbehandlungsroutine geschluckt werden.

Wenn Sie sicher sind, daß Ihre Komponente keine Überlauffehler mehr oder Fehler bei Arraygrenzen enthält, sollten Sie die Optionen KEINE ÜBERPRÜFUNG DER ARRAYGRENZEN, KEINE ÜBERPRÜFUNG BEI GANZZAHLÜBERLAUF und KEINE ÜBERPRÜFUNG AUF FEHLER BEI FLIESSKOMMABERECHNUNGEN setzen.

Ich rate davon ab, eine der übrigen Optionen zu setzen. Die erzielbaren Vorteile sind in den meisten Fällen vernachlässigbar. Sie riskieren damit höchstens äußerst schwer auffindbare Bugs einschließlich unregelmäßig auftretender Fehler, falscher Ergebnisse und datenabhängiger Fehler.

8.3.4 Register Komponenten und Debuggen

Diese Register enthalten einige der wichtigsten Projekt-Eigenschaften für die Entwicklung und zum Testen von ActiveX-Komponenten. Daher halte ich es für sinnvoll, diese Punkte für das nächstfolgende Kapitel aufzusparen, das sich mit dem Erstellen, dem Testen und der Versionskontrolle von Komponenten befaßt – dem Kapitel 9, »Komponenten erstellen und testen«.

8.4 Wie geht es weiter?

Wir haben nun in diesem Kapitel viele Aspekte der Entwicklung von ActiveX-Komponenten beleuchtet, von den Vor- und Nachteilen des Grundentwurfs einer Komponente bis hin zu konkreten Projekt-Eigenschaften. Dabei haben Sie eine Menge über die verschiedenen Komponenten-Typen und wie sie funktionieren gelernt.

Eines ist dabei etwas zu kurz gekommen. Außer ein paar simplen Projekten haben Sie hier nicht viel Code zu Gesicht bekommen. Was ist aus der StockQuote-Komponente geworden, die ich versprochen hatte?

Keine Bange – die ist in der Mache. Alle Beispiele in diesem Kapitel verwendeten die grundlegendsten Code-Konstruktionen: Anweisungen, mit denen jeder Visual-Basic-Programmierer vertraut sein sollte, bevor er dieses Buch zur Hand nimmt. Die StockQuote-Komponente beruht aber auf einigen fortgeschritteneren Sprachfähigkeiten.

Doch schauen wir uns als nächstes das Erstellen und Testen von ActiveX-Komponenten näher an.