Kapitel 14

Multithreading

Multithreading – das ist ein Thema, das selbst gestandenen Programmierern die Haare zu Berge stehen lassen kann. Erfreulicherweise jedoch unterstützt Visual Basic Multithreading auf so sichere und einfache Weise, wie es nur eben geht. Holen wir also tief Luft und stürzen wir uns auf das Thema.

14.1 Threads und Prozesse

Bisher haben wir uns mit dem Prinzip der Threads nur stark vereinfacht beschäftigt. In Kapitel 8, »Das Projekt«, führten wir das Prinzip von Ausführungs-Threads ein, als wir einen Thread als die Abfolge der Operationen einer gegebenen Anwendung definierten.

Jede Anwendung läuft in ihrem eigenen Prozeß und hat ihren eigenen Ausführungs-Thread. Das Betriebssystem schaltet in sehr kurzen Intervallen zwischen den verschiedenen Threads um, was jedoch für Sie als Programmierer transparent bleibt.

Jeder Prozeß läuft in seinem eigenen Speicherbereich. Die Möglichkeiten zur Interaktion mit anderen Anwendungen sind strikt kontrolliert. Aus unserer Perspektive erfolgt diese Interaktion über Aufrufe von Methoden und Eigenschaften von COM-Objekten. Wir vertrauen darauf, daß OLE unsere Befehle und Daten von einem Prozeß zum anderen transportiert.

COM-Objekte können als In-Process-Server implementiert sein. Dabei läuft das Objekt im Ausführungs-Thread der aufrufenden Anwendung. Sie können auch als EXE-Server implementiert werden, wobei die Objekte im Ausführungs-Thread des Servers laufen. Soll jedes Objekt in einem EXE-Server in seinem eigenen Ausführungs-Thread laufen, setzen Sie die `Instancing`-Eigenschaft auf `SingleUse`, so daß für jedes Objekt eine eigene Instanz des Servers (mit jeweils seinem eigenen Prozeß und Ausführungs-Thread) gestartet wird.

Dieses Szenario beschreibt soweit, wie die Dinge noch unter Visual Basic 4.0 standen, als die Unterstützung von COM-Objekten in Visual Basic eingeführt wurde. Dieses Szenario hat jedoch zwei wesentliche Schwachstellen:

- Das Starten eines separaten Prozesses für jedes EXE-Server-Objekt, um für jedes Objekt einen eigenen Thread zu erhalten, drückt erheblich auf die System-Performance, vor allem wenn mehr als nur ein paar wenige Objekte in Aktion treten.
- DLL-basierte Objekte vertragen sich nur schlecht mit multithreaded Clients bezüglich der Performance.

Der erste Punkt ist leicht nachzuvollziehen. Das Starten einer neuen Anwendung, um ein einzelnes Objekt zum Leben zu erwecken, ist der ineffizienteste Ansatz, den man sich vorstellen kann. Sie haben bereits Situationen kennengelernt, in denen es wünschenswert ist, jedes Objekt in einem eigenen Ausführungs-Thread

laufen lassen zu können. Dabei geht es in erster Linie darum, zu verhindern, daß sich Anwendungen gegenseitig blockieren können. Es geht aber auch um asynchrone Hintergrundoperationen.

Der zweite Punkt mag vielen Visual-Basic-Programmierern etwas obskur erscheinen. Das mag daran liegen, daß es erst seit dem Erscheinen des 2. Service-Packs zu Visual Basic 5 möglich ist, Multithread-Clients zu erstellen, und weil es bisher nur wenige Multithread-Clients gibt, für die man In-Process-Objekte schreiben könnte. Doch mit der steigenden Zahl sowohl von Client-Server-Datenbanken als auch von Internet-/Intranet-Servern und -Browsern (die sehr häufig multithreaded angelegt sind) rückt die Entwicklung von Komponenten für Multithread-Anwendungen zunehmend in den Vordergrund.

Visual Basic betrifft dies in beiderlei Hinsicht, da Sie ja auch Multithread-Komponenten schreiben können. Doch bevor wir uns mit der Entwicklung von Komponenten für Multithread-Umgebungen befassen, werfen wir einen Blick auf die Natur des Multithreadings und darauf, warum dieses Thema eine solche Herausforderung darstellt.

14.1.1 Multithreading

In Kapitel 8 sahen Sie, wie das Windows-Betriebssystem das Multitasking von mehreren Anwendungen handhabt. In unserem früheren einfachen Ansatz befand sich jeder Thread in einem eigenen Prozeß. Das Betriebssystem schaltet in rascher Folge zwischen den Threads um, so daß diese gleichzeitig zu laufen scheinen.

Wenn jedem Prozeß immer genau ein Thread zugeordnet wäre, könnte man auf die Unterscheidung zwischen Thread und Prozeß verzichten. Jedoch kann es in einem Prozeß auch mehrere Threads geben. Das Betriebssystem teilt die Prozessorzeit zwischen den Threads auf, und nicht zwischen den Prozessen. In Abbildung 14.1 sehen Sie eine Erweiterung des Modells aus Abbildung 8.1 in Kapitel 8, die zwei Single-Thread-Prozesse zeigte.

In diesem Beispiel hat jedoch die Anwendung A zwei separate Ausführungs-Threads. Wenn Sie die Pfeile verfolgen, die die Verteilung der Prozessorzeit darstellen, erkennen Sie, wie das System tatsächlich zwei verschiedene Teile der Anwendung A gleichzeitig in zwei separaten Ausführungs-Threads laufen läßt.

Wozu dient Multithreading bei Anwendungen und Komponenten? Es gibt eine ganze Reihe von häufig auftretenden Situationen, in denen das Multithreading-Prinzip eine sinnvolle Lösung darstellen kann.

Daten- bzw. Informations-Server

Nehmen Sie einmal an, Sie entwickeln einen Informations-Server, der Anfragen von vielen verschiedenen Clients erhält. Zu dieser Kategorie von Informations-Servern gehören beispielsweise Datenbank-Server und Internet-/Intranet-Server. Einige dieser Anfragen können eine geraume Zeit in Anspruch nehmen. Anstatt

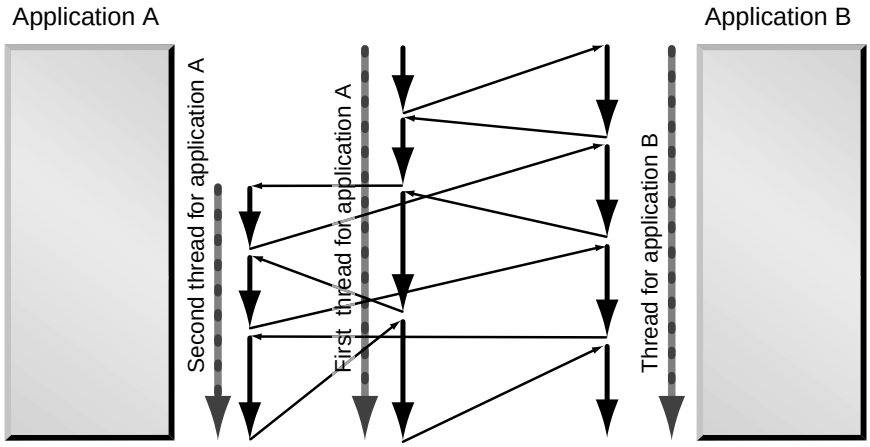

Abb. 14.1: Multithreading

nun während der zeitraubenden Bearbeitung einer einzelnen Anfrage alle übrigen Clients zu blockieren, können Sie für jede Anfrage einen eigenen Thread starten, so daß alle Anfragen anscheinend gleichzeitig bearbeitet werden. Da die Prozessorzeit letztlich jedoch endlich ist, wird bei diesem Ansatz die lange andauernde Anfrage noch länger dauern (da sie ja die Zeit mit den übrigen Anfragen teilen muß), die schneller zu erledigenden Client-Anfragen können dagegen insgesamt schneller abgearbeitet werden, da sie nicht mehr warten müssen.

Geht beispielsweise eine 30 Minuten dauernde Anfrage vier weiteren von je einer Minute Dauer voraus, kann es aus der Sicht der Clients mit den an sich kurzen Anfragen trotzdem 31, 32, 33 oder 34 Minuten dauern, bis sie ihre Antwort erhalten. Auf einem Multithread-Server wird dagegen die Prozessorzeit unter allen fünf Anfragen aufgeteilt. Die lange Anfrage wird dann vielleicht 35 Minuten benötigen, die vier weiteren dagegen jedoch maximal 5 Minuten.

An dieser Stelle werden ein paar interessante Aspekte des Multithreadings offenbar. Multithreading kann einerseits sehr sinnvoll sein, andererseits jedoch nicht das Wunder vollbringen, Ihr System im ganzen zu beschleunigen. Dazu kommt, daß die jeweilige Bearbeitungsdauer in einem Multithreading-System wegen des Overheads zum Umschalten zwischen den Threads länger dauert als die Bearbeitung in einem einfachen System. Die an sich 1minütigen Anfragen im genannten Beispiel werden so wegen dieses Overheads eher fünf Minuten zur Bearbeitung benötigen.

Tatsächlich bremst Multithreading das System. Nehmen wir einmal fünf 1minütige Client-Anfragen an. In einem Nicht-Multithreading-System werden die Anfragen nach 1, 2, 3, 4 und 5 Minuten bearbeitet sein (im Durchschnitt nach

jeweils 3 Minuten). In einem Multithreading-System werden alle Anfragen zugleich nach 5 Minuten erledigt sein – die Durchschnittszeit erhöht sich damit ebenfalls auf 5 Minuten.

Denken Sie also genau darüber nach, was Sie erreichen wollen, wenn Sie eine Server-Komponente entwickeln und dabei auf eine Beschleunigung der Performance abzielen wollen. Die gravierendste Beschleunigung macht sich bei einer bunten Mischung von kurzen und langandauernden Aufgaben bemerkbar.

Hintergrundoperationen

In Kapitel 11 könnten Sie nachlesen, wie Hintergrundoperationen mit Hilfe von OLE-Rückrufen realisiert werden können. Hintergrundoperationen lassen sich im allgemeinen in zwei Kategorien einordnen. Zur einen Kategorie gehören Operationen, die im Hintergrund ausgeführt werden, während der Anwender bereits irgend etwas anderes machen kann. Zur anderen Kategorie gehören die Fälle, in denen eine Anwendung auf ein außerhalb stattfindendes Ereignis wartet, und die den Hauptteil der Anwendung beim Eintreffen des Ereignisses benachrichtigen.

In Kapitel 8 ging es auch darum, wie ein SingleUse-EXE-Server zur Implementierung von Hintergrundoperationen mit Hilfe von Rückrufen (sowohl API- als auch OLE-Rückrufen) verwendet werden kann. Wenn Sie eine Hintergrundoperation mit einem SingleUse-EXE-Server implementieren, läuft diese in einem separaten Thread, da ja per Definition jeder Prozeß seine eigenen Threads verwendet. Wie Sie im Verlauf dieses Kapitels sehen werden, können Hintergrundoperationen allerdings auch in mehreren Threads in ein und demselben Prozeß laufen.

Aufwendigere Benutzeroberflächen und andere Multithread-Clients

Ein Internet-Browser ist ein gutes Beispiel für einen Multithread-Client. Es ist nicht ungewöhnlich, gleichzeitig mehrere Browser-Fenster geöffnet zu haben und in jedem ein anderes Web-Dokument abzurufen, gleichzeitig E-Mail zu versenden und trotzdem sofort auf jede Benutzeraktion zu reagieren. Eine solche Anwendung kann so implementiert werden, daß jeder Dokumenten-Abruf in einem eigenen Thread erfolgt, während ein anderer Thread sich um das E-Mail-Fenster kümmert. Sie können Multithread-Clients mit Visual Basic entwickeln, wie Sie gleich sehen werden.

14.1.2 Fallen bei Multithreading

Zunächst sollten Sie wissen, daß die meisten in diesem Abschnitt beschriebenen Fallgruben Visual Basic nicht betreffen. Ich gehe dennoch darauf ein, damit Sie verstehen können, was Microsoft unternommen hat, um Visual Basic einigermaßen Thread-sicher (Thread-safe) zu machen, aber auch, welche Vorteile und Beschränkungen dieser Ansatz mit sich bringt.

Weiterhin sollten Sie wissen, daß sich alle Threads einer Anwendung den gleichen Prozeßraum teilen. In Kapitel 8 haben Sie gelernt, daß einer der großen Vorteile des 32-Bit-Windows gegenüber dem alten 16-Bit-Windows ist, daß jede

Anwendung in ihrem eigenen Prozeßraum läuft. Jede Anwendung verfügt damit über einen eigenen Bereich des Arbeitsspeichers, so daß sich die Prozesse nicht gegenseitig ins Gehege kommen können. Kein Prozeß kann den Speicher eines anderen manipulieren, und im Falle des Absturzes einer Anwendung werden in der Regel weder die anderen Anwendungen noch das Betriebssystem in Mitleidenschaft gezogen.

Dieser Schutz betrifft Prozesse, nicht Threads. Jeder Thread hat freien Zugriff auf den gesamten Speicherbereich eines Prozesses. Das bedeutet, daß ein Thread mit einem anderen interagieren kann (und dies auch häufig tut). Läuft ein Thread fehlerhaft, dürfte die ganze Anwendung terminieren, und nicht nur der betroffene Thread.

Nehmen wir an, in Ihrer Anwendung öffnet eine Routine eine Datei, schreibt ein paar Daten hinein und schließt sie wieder. In Pseudo-Code könnte das etwa so aussehen:

```
Global FileHandle As Long

Procedure SaveFile(filename)
    FileHandle = OpenFile (filename)
    WriteToFile(FileHandle, "This is line 1")
    WriteToFile(FileHandle, "This is line 2")
    CloseFile(FileHandle)
End Procedure
```

Solange Ihre Anwendung in einem einzigen Thread läuft, brauchen Sie nichts zu befürchten. Sobald Sie die Funktion `SaveFile` aufrufen, wird der Thread die Funktion von Anfang bis Ende ausführen. Das System mag gegebenenfalls zwischendurch zu anderen Threads in anderen Prozessen umschalten, die jedoch nicht mit Ihrem Programm ins Gehege kommen (außer Sie haben File-Sharing eingerichtet und greifen auf die gleiche Datei zu).

Nehmen wir nun jedoch einmal an, Sie hätten einen zweiten Thread in Ihrer Anwendung und beide Threads würden gleichzeitig eine Datei zu sichern versuchen. Sie haben keine Möglichkeit vorherzusehen, wie und in welcher Reihenfolge die einzelnen Operationen ausgeführt werden, wenn das System den Prozessor von Thread zu Thread weiterschaltet. Dabei können Ergebnisse zustandekommen, wie Sie sie in Tabelle 14.1 sehen können.

Operation Thread 1	Operation Thread 2	Ergebnis
FileHandle = OpenFile(»Datei1«)		FileHandle enthält Handle für Datei1
WriteToFile(FileHandle, »Dies ist Zeile 1«)		Datei1 enthält »Dies ist Zeile 1«

Tab. 14.1: *Multithreading-Operationen und ihre Ergebnisse*

Operation Thread 1	Operation Thread 2	Ergebnis
	FileHandle = OpenFile(»Datei2«)	FileHandle enthält nun Handle auf Datei2
	WriteToFile(FileHandle, »Dies ist Zeile 1«)	Datei2 enthält »Dies ist Zeile 1«
WriteToFile(FileHandle, »Dies ist Zeile 2«)		Datei2 enthält nun »Dies ist Zeile 1« »Dies ist Zeile 2«
	WriteToFile(FileHandle, »Dies ist Zeile 2«)	Datei2 enthält nun »Dies ist Zeile 1« »Dies ist Zeile 2« »Dies ist Zeile 2«
CloseFile(FileHandle)		Datei2 ist nun geschlossen, Datei1 jedoch weiterhin geöffnet
	CloseFile(FileHandle)	Höchstwahrscheinlich wird ein Fehler ausgelöst, da ein mittlerweile ungültiges FileHandle verwendet wird

Tab. 14.1: *Multithreading-Operationen und ihre Ergebnisse*

Wie Sie sehen können, deuten die Ergebnisse auf ernsthafte Probleme hin. In diesem Beispiel werden zwei möglicherweise beschädigte Dateien hinterlassen. Die ungültige Operation kann vielleicht durch eine Fehlerbehandlung abgefangen werden. Die geöffnete Datei1 verbleibt jedoch geöffnet, bis die Anwendung beendet wird.

Dies ist nur ein kleiner Ausschnitt aus den möglichen Problemen. Wie hoch stehen die Chancen, daß zwei Threads die `SaveFile`-Prozedur zur gleichen Zeit aufrufen? Eins zu fünf? Eins zu 1 Million? Wenn das unregelmäßig vorkommt, haben Sie damit eines dieser wunderhübschen Probleme am Hals, die immer erst dann auftreten, wenn eine Anwendung bereits vielfach auf dem Markt verkauft worden ist. Dazu kommt, daß derartige Probleme nur schwer auszutesten, schwer zu entdecken und schwer zu debuggen sind.

Ein solches Problem ist Visual-Basic-Programmierern nicht ganz unbekannt. Wenn etwa die Funktion `SaveFile` eine `DoEvents`-Anweisung enthalten würde und kein Code verhindert, daß die Prozedur erneut aufgerufen bzw. abgearbeitet wird, können die gleichen Probleme auftreten. Die meisten Visual-Basic-Programmierer vermeiden aus diesem Grund `DoEvents`-Anweisungen und sperren gegebenenfalls Controls oder Funktionen, wenn diese Anweisung benötigt wird.

Die DoEvents-Anweisung öffnet zwar die Tür zu solchen Wiedereintrittsproblemen, hat aber selbst nichts mit Multithreading zu tun. Eine DoEvents-Anweisung erlaubt Windows lediglich im gleichen Thread an eine andere Code-Stelle zu springen und zunächst den dortigen Code auszuführen. Nach dessen Bearbeitung wird der ursprüngliche Code hinter der DoEvents-Anweisung weiter abgearbeitet (ähnlich wie bei einem Funktionsaufruf). Das Ganze spielt sich nach wie vor in ein und demselben Thread ab.

Natürlich war das nur ein sehr einfaches Beispiel – die Probleme ließen sich allein schon dadurch vermeiden, keine globale Variable für das FileHandle zu verwenden. Es ging lediglich darum, die Gefahren bei der Verwendung mehrerer Threads in einer Anwendung aufzuzeigen. Multithreading ist genau deswegen gefährlich, weil Anwendungen globale Variablen und globale Funktionen haben können und anwendungsweit auf Ressourcen wie etwa Formulare, Controls und Dateien zugegriffen werden kann.

Anders als bei der DoEvents-Anweisung, bei der Sie genau festlegen, wann und an welcher Stelle Windows woanders hinspringen darf, kann beim Multithreading Windows nach eigenem Ermessen zwischen den verschiedenen Threads hin- und herschalten.

Programmierer, die auf traditionelle Weise Multithread-Anwendungen schreiben, müssen sorgfältig darauf achten, wie sie auf globale Variablen und Ressourcen zugreifen. Windows bietet eine ganze Reihe von Synchronisationsanweisungen und -Objekten, mit deren Hilfe ein Programmierer kontrollieren kann, wann welcher Teil einer Anwendung laufen darf.

In diesem Fall könnte man ein Objekt namens Mutex verwenden, das die SaveFile-Funktion verriegeln würde. Will ein Thread die Funktion aufrufen, prüft er zunächst, ob das Mutex-Objekt verfügbar ist. Ist dies der Fall, ruft der Thread die Funktion auf, jedoch nicht, bevor er das Mutex-Objekt blockiert hat. Ein anderer Thread, der die gleiche Funktion aufrufen möchte, stößt auf den blockierten Mutex und führt zunächst eine Wartefunktion des Mutex aus. Der Thread wird vorübergehend »schlafengelegt« – Windows wird den Thread nicht eher wieder ausführen, bis der Mutex vom ersten Thread wieder freigegeben worden ist.

Die Tatsache, daß die meisten Probleme des Multithreadings aus der Verwendung von globalen Variablen und gemeinsam genutzten Ressourcen resultieren, wirft eine interessante Frage auf. Wie wäre es, wenn es gelingen würde, auf globale Variablen und gemeinsam genutzte Ressourcen zu verzichten? Würde das nicht die meisten der Multithreading-Gefahren eliminieren, während man weiterhin in den Genuß der Vorteile kommen würde? Die Antwort lautet: ja.

Allerdings würde die Verbannung von globalen Variablen aus Visual Basic eine grundlegende Änderung der Sprache erfordern. Was wäre aber, wenn man jedem Thread einen separaten Satz an globalen Variablen zuweisen könnte? Im Beispiel oben wären dies zwei FileHandle-Variablen, für jeden Thread eine eigene. Der Code bliebe unverändert. Auf die Variable würde weiterhin unter dem Namen

`FileHandle` zugegriffen. Visual Basic würde jedoch automatisch die Daten für jeden Thread getrennt verwalten, so daß jeder Thread nur seine eigene Kopie der Variablen sehen würde. In Tabelle 14.2 sehen Sie diesen Ansatz für das `SaveFile`-Beispiel. Die Kennzeichnungen (T1) und (T2) zeigen an, welche der beiden Kopien der `FileHandle`-Variable jeweils aktiv ist.

Dies ist der erste Teil des Ansatzes, über den Visual Basic für Sicherheit beim Multithreading sorgt. Der zweite, wahrscheinlich wichtigere Teil ist subtiler angelegt und betrifft die Art und Weise, wie COM mit Multithreading umgeht.

Operation Thread 1	**Operation Thread 2**	**Ergebnis**
FileHandle = OpenFile(»Datei1«)		FileHandle(T1) enthält Handle für Datei1
WriteToFile(FileHandle, »Dies ist Zeile 1«)		Datei1 enthält »Dies ist Zeile 1«
	FileHandle = OpenFile(»Datei2«)	FileHandle(T2) enthält nun Handle auf Datei2
	WriteToFile(FileHandle, »Dies ist Zeile 1«)	Verwendet FileHandle(T2) Datei2 enthält »Dies ist Zeile 1«
WriteToFile(FileHandle, »Dies ist Zeile 2«)		Verwendet FileHandle(T1) Datei1 enthält nun »Dies ist Zeile 1« »Dies ist Zeile 2«
	WriteToFile(FileHandle, »Dies ist Zeile 2«)	Verwendet FileHandle(T2) Datei2 enthält nun »Dies ist Zeile 1« »Dies ist Zeile 2«
CloseFile(FileHandle)		Verwendet FileHandle(T1) Datei1 ist nun geschlossen
	CloseFile(FileHandle)	Verwendet FileHande(T2) Datei2 ist nun geschlossen

Tab. 14.2: *Multithreading mit Apartment-Model-Threading*

14.1.3 Multithreading-Modelle in COM

Sie haben gesehen, daß Multithreading immer dann Probleme verursachen kann, wenn Variablen existieren, die von mehreren Threads gemeinsam genutzt werden können. Was bedeutet dies aus der Perspektive eines COM-Objekts? Ein COM-

Objekt legt ein Interface offen und enthält interne Daten. Was geschieht, wenn zwei verschiedene Threads Zugriff auf das Interface eines COM-Objekts haben? Treten Probleme auf, wenn sie gleichzeitig dieselbe Methode eines Interfaces aufrufen und auf die internen Variablen der Komponente zuzugreifen versuchen?

Mit Sicherheit. Solange eine Komponente nicht selbst Code enthält, der den Zugriff auf die internen Daten regelt, bedeutet der Zugriff von verschiedenen Threads aus den sicheren Weg in die Katastrophe. Eine Komponente dagegen, die mit solchen Situationen umgehen kann, kann problemlos von vielen Threads gleichzeitig aufgerufen werden. Eine solche Komponente wird »free-threaded« genannt und ist in Visual Basic ziemlich schwierig zu erstellen. Es ist sogar so kompliziert, daß ich davon abrate, es jemals zu versuchen. Weitere Informationen über freies Threading über API-Techniken von Visual Basic aus können Sie in dem (englischsprachigen) Artikel »A Thread to Visual Basic« auf der Web-Site von Desaware finden (`http://www.desaware.com`).

Das genaue Gegenteil stellen Komponenten dar, die absolut unfähig sind, mit mehreren Threads umzugehen. Sobald eine DLL oder EXE ein Objekt anlegt, muß auf alle Objekte in dieser Komponente vom gleichen Thread aus zugegriffen werden, der die DLL zuerst aufgerufen hat. Dieser Typ einer Komponente wird »single-threaded« genannt. Visual Basic kann natürlich single-threaded Komponenten anlegen.

Darüber hinaus erlaubt Visual Basic das Erstellen von Komponenten, die zwischen diesen beiden Typen gelagert sind. Deren Art des Threadings wird als »Apartment Model« bezeichnet. Eine DLL oder ein EXE-Server nach diesem Modell kann Objekte anlegen, die in verschiedenen Threads laufen können. Der Haken hierbei ist: Nachdem ein Objekt einmal in einem bestimmten Thread angelegt worden ist, muß jeder weitere Zugriff vom gleichen Thread aus erfolgen. Auf den ersten Blick mag das nach einer Beschränkung aussehen – wenn etwa von einem multithreaded Client aus verschiedenen Threads heraus auf Objekte zugegriffen werden soll. Das ist aber kein Problem. Erinnern Sie sich daran, wie COM es ermöglicht, auf Objekte über Prozeßgrenzen hinweg über ein Proxy-Objekt und das Marshaling von Methoden-Aufrufen zuzugreifen? Nun, COM kann dasselbe auch bei Zugriffen von Thread zu Thread erledigen, indem ein Proxy-Objekt angelegt wird, das für den reibungslosen Transport der Parameter und Rückgabewerte zwischen den Threads sorgt. Dies kostet natürlich mehr Zeit als direkte Aufrufe, erlaubt jedoch die gemeinsame Nutzung von Apartment-threaded wie auch single-threaded Objekten durch verschiedene Threads.

14.2 Multithread-Komponenten

ActiveX-In-Process-Server wie DLL-Server, ActiveX-Controls und In-Process-ActiveX-Dokumente können im ALLGEMEIN-Register der Projekt-Eigenschaften im Kasten MULTITHREADING-MODELL konfiguriert werden. Bei In-Process-Komponenten sollte der einzige Grund sein, der gegen Multithread spricht, daß

Objekte Daten über globale Variablen gemeinsam nutzen können sollen. Ansonsten ist immer Multithreading zu empfehlen.

Bei ActiveX-Servern wird Multithread über die Option THREAD PRO OBJEKT gewählt. Sie können auch die Anzahl der Threads im THREAD-POOL heraufsetzen – die Anzahl 1 ist gleichbedeutend mit single-threaded.

Später werden Sie noch sehen, daß es einige weitere kritische Differenzen zwischen EXE- und DLL-basierten Servern gibt.

14.2.1 Threads und Objekte

Was heißt das genau, wenn wir davon sprechen, daß eine Visual-Basic-Komponente multithreaded ist? Es bedeutet, daß Visual Basic Objekte in verschiedenen Threads laufen lassen kann. Betrachten wir zunächst, wie dies bei einem EXE-Server aussieht und worin der Unterschied zu einem nicht-multithreaded Server liegt.

EXE-Server

In Abbildung 14.2 sehen Sie einen nicht-multithreaded EXE-Server, der ein Objekt implementiert, dessen `Instancing`-Eigenschaft auf `MultiUse` gesetzt ist. Der Server liefert drei Objekte an drei verschiedene Anwendungen – genauer gesagt: Er liefert drei Objekte an drei verschiedene Threads. In diesem Fall befinden sich die drei Threads in verschiedenen Prozessen. Das Beispiel trifft aber genauso für Objekte zu, die von verschiedenen Threads in einem Multithread-Client angefordert wurden.

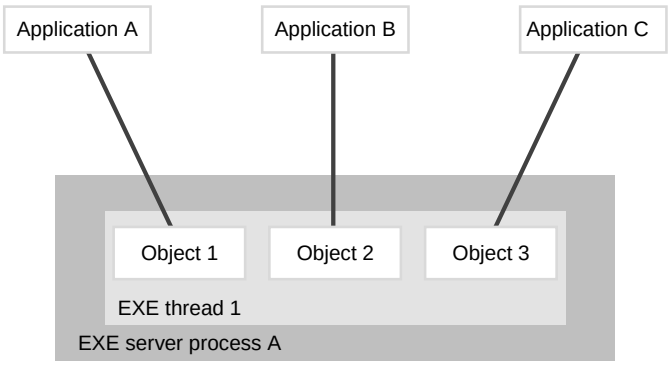

Abb. 14.2: Von einem nicht-multithreaded EXE-Server implementierte Objekte, bei denen die Instancing-Eigenschaft auf MultiUse gesetzt ist

Der Server wird von dem unteren Block repräsentiert. Von den beiden grauen Rechtecken stellt das äußere den Prozeßraum des Servers, das innere dessen Thread dar. Die weißen Rechtecke darin repräsentieren die Objekte. Hier befinden sich alle Objekte in einem einzigen Prozeß und laufen im gleichen Thread.

Wird eines der Objekte von einer der Anwendungen mit einer lang andauernden Operation beschäftigt, wird der Thread des Servers blockiert, so daß die anderen Client-Anwendungen so lange nicht auf Methoden und Eigenschaften der von ihnen angeforderten Objekte zugreifen können, bevor nicht die eine Operation abgearbeitet ist.

Globale Variablen werden von allen Objekten gemeinsam genutzt, da sie sich den gleichen Thread teilen.

In Abbildung 14.3 sehen Sie einen nicht-multithreaded EXE-Server, der ein Objekt implementiert, dessen `Instancing`-Eigenschaft auf `SingleUse` gesetzt ist. Drei separate Instanzen des EXE-Servers laufen als separate Prozesse, in denen jeweils ein einzelnes Objekt angelegt wird. Zwangsläufig befindet sich jedes Objekt in einem eigenen Thread. Da sich zudem jedes Objekt in einem eigenen Prozeßraum befindet, ist auch offensichtlich, daß von den Objekten keine globalen Daten gemeinsam genutzt werden können.

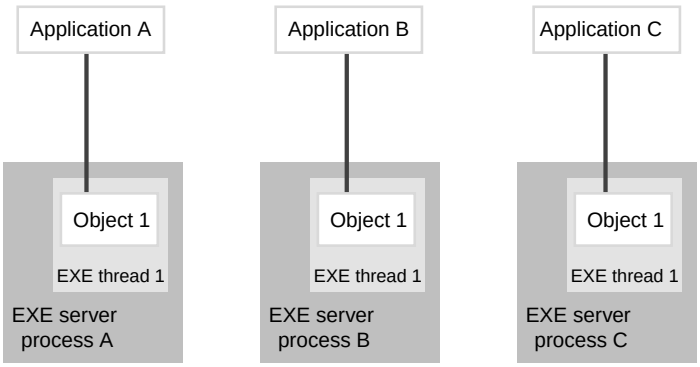

Abb. 14.3: Von einem nicht-multithreaded EXE-Server implementierte Objekte, bei denen die Instancing-Eigenschaft auf SingleUse gesetzt ist

Was passiert, wenn ein EXE-Server mehrere Klassen enthält, deren `Instancing`-Eigenschaft auf `SingleUse` gesetzt ist? Jeder Server kann ein `SingleUse`-Objekt von jeder Klasse anlegen. Wenn der Server zwei `SingleUse`-Klassen enthält, beispielsweise `MeinObjektA` und `MeinObjektB`, kann eine Instanz des EXE-Servers von jeder Klasse genau eine Instanz anlegen. In diesem Fall werden globale Variablen von beiden Instanzen gemeinsam genutzt, da sie sich im gleichen Thread befinden! Dies kann zu recht eigenartigen Konstellationen führen.

In Abbildung 14.4 sehen Sie einen weiteren nicht-multithreaded EXE-Server. Hier hat zunächst Anwendung A ein Objekt der Klasse `MeinObjektA` angefordert – der Server legt eine Instanz dieser Klasse für die Anwendung an. Als nächstes fordert die Anwendung B eine Instanz der Klasse `MeinObjektB` an. Die erste Server-Instanz hat noch kein Objekt dieser Klasse angelegt und kann die gewünschte Instanz anlegen. Beachten Sie, daß diese beiden Objekte die gleichen globalen

Variablen gemeinsam nutzen, da sie sich im gleichen Thread befinden. Eine länger andauernde Operation in `MeinObjektA` wird auch den Zugriff von Anwendung B auf `MeinObjektB` blockieren. Wenn nun Anwendung A wiederum eine Instanz der Klasse `MeinObjektB` anfordert, wird eine neue Instanz des Servers geladen, da ja die erste Instanz des Servers bereits eine Instanz dieser Klasse instanziert hat.

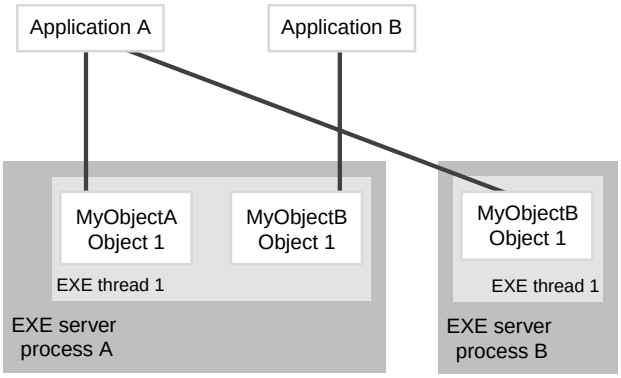

Abb. 14.4: Mehrere als SingleUse instanzierte Objekte

Das bedeutet, daß Sie sich nie sicher sein können, welcher Server-Prozeß schließlich eine Instanz einer Klasse liefert. Seien Sie also bei solchen Konstellationen vorsichtig beim Gebrauch von globalen Variablen.

Beachten Sie auch, daß ein EXE-Server, der `SingleUse`-Objekte liefert, dennoch zusätzliche Instanzen der gleichen Klasse im gleichen Thread liefern kann, wenn er diese Instanzen selbst anlegt und als Referenz an einen Client weiterreicht.

In Abbildung 14.5 sehen Sie einen Multithread-EXE-Server, der ein `MultiUse`-Objekt implementiert. Hier ist der Server so konfiguriert, für jedes Objekt, also jede Instanz, einen eigenen Thread anzulegen. Alle Objekte laufen im gleichen Prozeßraum. Allerdings haben entsprechend dem Apartment-Modell alle Instanzen separate globale Variablen für sich.

Dieser Ansatz ist wesentlich effizienter als der Nicht-Multithread-Ansatz, da der System-Overhead beim Anlegen eines jeweils eigenen Prozesses für jedes Objekt entfällt.

Denken Sie daran, daß jedes Server-Objekt jeweils in einem eigenen Thread und nicht im Thread des aufrufenden Prozesses läuft. Sie laufen daher auch im Server-Prozeß und nicht im aufrufenden Prozeß. Die Effizienz, die sich daraus ergibt, im gleichen Prozeß zu laufen (Vermeidung des Marshaling-Overheads), bleibt DLL-Servern vorbehalten.

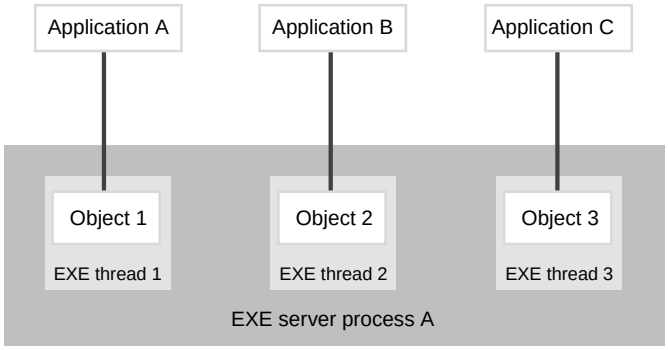

Abb. 14.5: Von einem Multithread-EXE-Server instanzierte Objekte

Die in Abbildung 14.6 dargestellte Situation geht noch einen Schritt weiter und illustriert einige der Möglichkeiten, die sich aus der Implementierung von multithreaded EXE-Server-Objekten ergibt. Wie bei `SingleUse`-Objekten kann ein Multithread-EXE-Server zusätzliche Instanzen selbst in seinem eigenen Thread anlegen und diese Objekte an eine Anwendung zurückgeben. So erhält hier die Anwendung C eine Instanz von Objekt 4, das etwa über einen Methoden-Aufruf in Objekt 3 angelegt worden ist. Um eine Instanz im gleichen Thread zu erzeugen, braucht ein Objekt lediglich einer in ihm deklarierten Variablen des gewünschten Typs eine mit dem `New`-Operator angelegte neue Instanz dieses Typs zuzuweisen. Wird die neue Instanz dagegen per `CreateObject`-Anweisung angelegt, landet sie in einem neuen Thread.

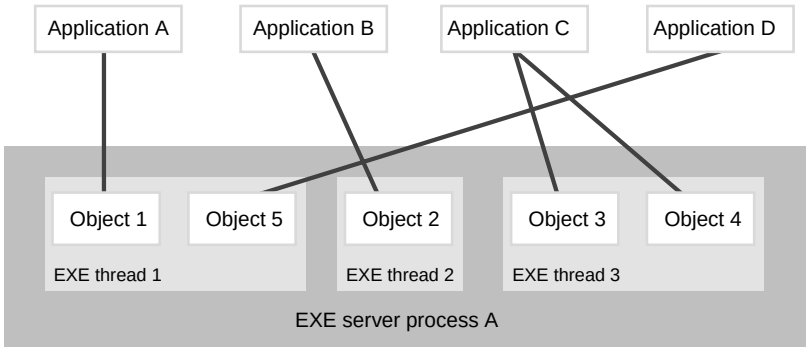

Abb. 14.6: Ein Multithread-EXE-Server, der weitere Objekte anlegt

Wie kam jedoch Anwendung D zur Instanz des Objekts 5 im ersten Thread? Für dieses Beispiel wurde die Anzahl der Threads im Thread-Pool auf 3 heraufgesetzt. Diese Option weist den Server an, die Anzahl der verfügbaren Threads auf drei zu begrenzen. Sobald die drei Threads »verbraucht« sind, werden neue Objekt-Instanzen der Reihe nach in den vorhandenen Threads angelegt.

Wenn Sie etwa vier Threads hätten, in jedem 10 Objekte laufen würden, und schließlich alle 10 in einem der Threads beendet würden, wäre es sinnvoll, zunächst diesen Thread wieder mit Objekten zu füllen, um die Verteilung ausgeglichen zu halten. Leider nimmt Visual Basic keinerlei Ausgleich der Verteilung vor – die neuen Objekte werden reihum auf die Threads verteilt. Objekte, die von Objekten des EXE-Servers selbst angelegt werden, bleiben unberücksichtigt – sie werden jeweils im gleichen Thread angelegt.

Hier war es also so, daß das neue von Anwendung D angeforderte Objekt im nächsten Thread in Folge angelegt wurde, also Thread 1.

Einer der Seiteneffekte dieser Vorgehensweise ist, daß nun sowohl Anwendung A als auch Anwendung D Objekte im gleichen Thread haben. Diese Objekte verwenden die globalen Variablen gemeinsam, und Operationen in einem Objekt können das andere Objekt blockieren.

Dies verdeutlicht einen der größten Nachteile des Thread-Poolings. Sie können nicht vorhersagen, in welchem Thread ein bestimmtes Objekt angelegt wird oder mit welchen anderen Objekten es sich den Thread teilen muß. Warum sollte man dann überhaupt Thread-Pooling wählen? Weil jeder neue Thread die Systembelastung erhöht, wird irgendwann der Punkt erreicht, daß die Vorteile des Multithreadings von der Systembelastung wieder aufgefressen werden. Thread-Pooling erlaubt somit die Begrenzung der Thread-Anzahl und damit auch eine Begrenzung der System-Ressourcen, die der Komponenten-Server verbraucht. Sie sollten also bei der Verwendung von globalen Variablen mit Bedacht vorgehen, wenn Sie Thread-Pooling wählen – besser noch, Sie verzichten vollkommen auf globale Variablen.

DLL-Server

DLL-Server unterscheiden sich von EXE-Servern dahingehend, daß die Objekte eines DLL-Servers nie in einem eigenen Thread laufen. Bei DLL-Servern stellt sich die Frage, ob ein Objekt in dem Thread läuft, der die DLL gestartet hat, oder ob es in dem Thread läuft, der das Objekt angelegt hat. Ist eine DLL nicht multithreaded, laufen alle Objekte in dem Thread, der die DLL geladen hat. Dieses Szenario sehen Sie in Abbildung 14.7. Alle Objekte laufen im gleichen Thread und verwenden die gleichen globalen Variablen gemeinsam.

Was stimmt nicht an diesem Szenario? Sie sollten mittlerweile mit dem Stoff aus Kapitel 6 vertraut sein, in dem es auch um die Performance-Auswirkungen von Marshaling über Prozeß-Grenzen hinweg ging. Nun, ich habe es bisher noch zurückgehalten, aber nun kann ich Ihnen sagen, daß die Auswirkungen des Marshalings von Daten zwischen Threads deutlich spürbar sind. OLE selbst nutzt ein Apartment-Model-Threading. Unter NT 4.0 gibt es weitere Modelle, die jedoch von Visual Basic nicht unterstützt werden. Ein unkontrollierter Zugriff auf Objekte in anderen Threads würde die Gefahr der oben beschriebenen Probleme bei jedem Methoden-Aufruf und Eigenschaften-Zugriff bedeuten. Da die Objekte jedoch im gleichen Prozeß laufen, geht das Marshaling ein wenig schneller von-

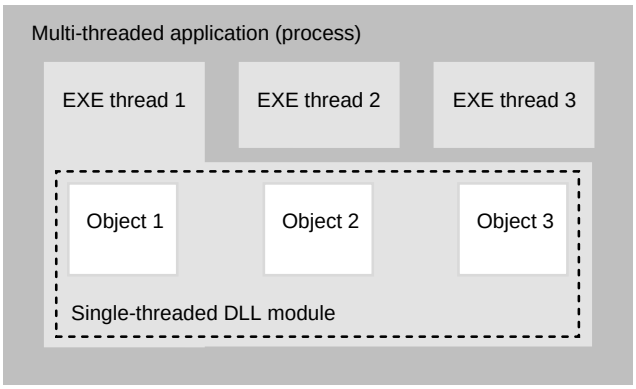

Abb. 14.7: Von einem nicht-multithreaded DLL-Server implementierte Objekte

statten als bei Out-of-Process-Objekten. Auf jeden Fall werden allerdings interne Proxy-Objekte benötigt, damit der Zugriff auf die Objekte sauber synchronisiert werden kann – nur ein Thread darf zur gleichen Zeit auf ein Objekt zugreifen.

Im Szenario in Abbildung 14.7 fällt das prozeßübergreifende Marshaling immer dann an, wenn von den Threads EXE-Thread 2 und EXE-Thread 3 aus auf die Objekte 2 und 3 zugegriffen werden soll. Angenommen EXE-Thread 2 führt eine langandauernde Operation in Objekt 2 aus und EXE-Thread 3 versucht, auf Objekt 3 zuzugreifen: EXE-Thread 3 wird blockiert, da Thread 1 mit der Ausführung der von EXE-Thread 2 gestarteten Operation beschäftigt ist.

Das Problem kann umgangen werden, indem die DLL Multithreading-fähig gemacht wird (siehe Abbildung 14.8). Die DLL legt dabei jedoch keine neuen Threads an. Ist eine DLL multithreaded, bedeutet dies lediglich, daß jedes Objekt in dem Thread läuft, der es angelegt hat. Wenn wie hier Objekt 1 von EXE-Thread 1, Objekt 2 von EXE-Thread 2 usw. angelegt wurden, fällt kein prozeßübergreifendes Marshaling an, solange auf die Objekte nur aus den Threads zugegriffen wird, von denen sie angelegt worden sind. Objekte in verschiedenen Threads können zwar weiterhin aufeinander zugreifen, jedoch wird dann wieder Marshaling notwendig.

In diesem Beispiel befindet sich jedes Objekt in dem Thread, von dem es angelegt wurde. Globale Variablen werden nur von den Objekten gemeinsam verwendet, die sich im gleichen Thread befinden.

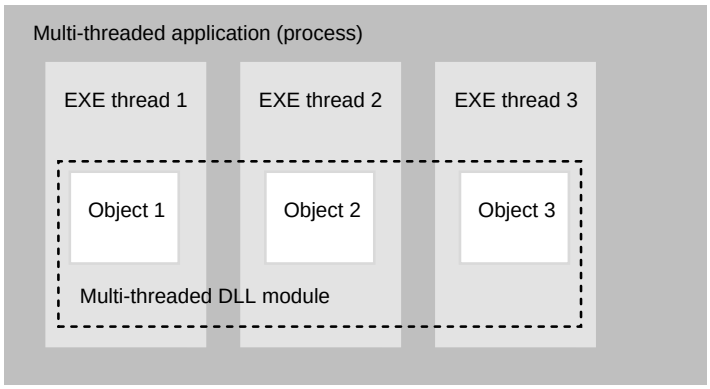

Abb. 14.8: Von einem Multithread-DLL-Server angelegte Objekte

14.2.2 Gültigkeitsbereich

In Kapitel 10, »Code und Klassen«, behandelten wir das Thema der Gültigkeitsbereiche – die Regeln, die die Lebensdauer und Sichtbarkeit von Variablen in Abhängigkeit vom Ort ihrer Deklaration festlegen. Vielleicht sollten Sie sich dieses Kapitel noch einmal anschauen, bevor Sie weiterlesen.

In den vorangegangenen Abschnitten dieses Kapitels ist mehrfach die Rede von gemeinsam verwendeten globalen Variablen gewesen. Die Tatsache, daß globale Variablen nicht von Objekten in verschiedenen Threads gemeinsam genutzt werden, scheint eine gründliche Revision der Gültigkeitsbereichsregeln notwendig werden zu lassen.

Einiges hat sich nicht geändert. Variablen, die auf Modul-Ebene einer Klasse deklariert werden, werden weiterhin für jede Instanz der Klasse separat angelegt und existieren nur während deren Lebenszeit. In nicht als `Static` deklarierten Prozessen deklarierte Variablen oder einfache in Prozessen deklarierte Variablen werden weiterhin bei jedem einzelnen Aufruf einer Prozedur angelegt und existieren so lange, bis die Prozedur verlassen wird. Auf Modul-Ebene eines Formulars angelegte Variablen – Sie haben es erraten: Es gibt keine Formulare in einer Multithread-Komponente, diese Situation kommt also nicht in Betracht.

Die einzigen Variablen, die in Betracht kommen, sind also in einem Standard-Modul deklarierte globale Variablen oder als `Static` deklarierte Variablen in Prozeduren. Diese Variablen werden ja einmal in jeder Anwendung angelegt und existieren, bis die Anwendung beendet wird.

Bei Multithread-Komponenten werden diese Variablen einmal je Thread angelegt und existieren, solange der Thread existiert. Die Sichtbarkeit einer globalen Variablen ist auf den betreffenden Thread begrenzt. Alle Objekte in ein und demselben Thread nutzen die gleichen globalen Variablen gemeinsam. Objekte in anderen Threads sehen nur ihre eigenen Kopien der Daten.

14.3 Testen und Debuggen von Multithread-Komponenten

Jede Instanz der Visual-Basic-Entwicklungsumgebung kann mit einem einzelnen Thread arbeiten. Daher können Sie die Effekte des Multithreadings in Ihren Komponenten nicht in der Visual-Basic-Entwicklungsumgebung testen. Zum Testen müssen Sie daher kompilierte Versionen Ihrer Komponenten verwenden. Sie können die Komponente zwar nach wie vor in der Entwicklungsumgebung starten, jedoch werden die Multithreading-Effekte nicht zum Tragen kommen.

Da Sie das Multithreading nicht in der Entwicklungsumgebung testen können, gibt es natürlich auch kein Direktfenster, in dem Sie beobachten könnten, was in Ihren Komponenten vor sich geht. Eine Möglichkeit wäre, die Komponente in Native-Code zu kompilieren, Debug-Informationen einzuschließen und einen Stand-alone-Debugger wie etwa den aus Visual C++ zu verwenden. Es gibt jedoch Alternativen.

14.3.1 Ein Debug-Monitor

Das Debug-Monitor-Projekt im Ordner zu Kapitel 14 auf der Buch-CD (`DmDebMon.vbp`) enthält ein Klassen-Modul und zwei Formulare. Diese Komponente ist als ActiveX-EXE-Server angelegt, da sie Nachrichten von mehreren Anwendungen und Threads erhalten soll. Die Objekte werden von vielen Threads verwendet. Trotzdem sollen sie Daten zentral gemeinsam verwenden, damit der Monitor-Server die von den Objekten eingehenden Nachrichten in der Reihenfolge des Eingangs anzeigen kann. Die `Instancing`-Eigenschaft des Klassen-Moduls `Trace` ist auf `MultiUse` gesetzt, so daß alle Instanzen von der gleichen Server-Instanz angelegt werden. Die Komponente `DebugMonitor` selbst ist single-threaded, damit alle Objekte die gleichen globalen Daten verwenden.

Den Code der `Trace`-Klasse sehen Sie in Listing 14.1. Die Klasse bietet eine einzige öffentliche Methode namens `Add` mit einem `String`-Parameter. Anwendungen rufen diese Methode auf, um eine im Hauptformular des Monitors anzuzeigende Nachricht zu übermitteln. Die Methode selbst ruft lediglich die `Add`-Methode des Formulars `frmDebug` auf, dem Hauptformular der Komponente.

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

Option Explicit

' Nachricht dem Formular übergeben
Public Sub Add(msg As String)
    frmDebug.Add msg
End Sub
```

Listing 14.1: Die Klasse Trace (DMTrace.cls)

Listing 14.2 zeigt den Code des Hauptformulars. Es enthält ein Menü mit den drei Punkten UPDATE, CLEAR und OPTIONS. Weiterhin enthält es die ListBox `lstDebug`, in der die eingehenden Nachrichten in der Reihenfolge des Eingangs angezeigt werden. Dieses Formular wird in der `Sub Main`-Prozedur des Moduls `DmDebug.bas` geöffnet, die als Start-Objekt des Projekts gesetzt ist. So wird sie automatisch geladen und angezeigt, sobald die Komponente selbst geladen wird. Das Formular enthält dazu noch den Timer `Timer1`.

```
' DebugMonitor trace program
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved
Option Explicit
Dim StartPos&
Dim EndPos&
Dim MaxStrings&

Dim StringArray() As String

Private Sub Form_Load()
    SetArraySize 200
End Sub

' Anzahl der Strings festlegen
' Löscht den aktuellen Inhalt
Public Sub SetArraySize(ByVal maxsize%)
    If maxsize < 10 Then maxsize = 10
    If maxsize > 1000 Then maxsize = 1000
    MaxStrings = maxsize
    ' Array-Größe aktualisieren
    ReDim StringArray(MaxStrings)
    StartPos = 1
    EndPos = 1
    UpdateList
End Sub

' ListBox aktualisieren
Public Sub UpdateList()
    Dim counter&
    Dim currenttop&
    currenttop = lstDebug.TopIndex
    lstDebug.Clear
    counter = StartPos
    Do While counter <> EndPos
        lstDebug.AddItem StringArray(counter)
        counter = counter + 1
        If counter > MaxStrings Then
            counter = 1
        End If
```

```
    Loop
    If currenttop < lstDebug.ListCount Then
        lstDebug.TopIndex = currenttop
    End If
End Sub

' String in Liste einfügen
Public Sub Add(newstring$)
    StringArray(EndPos) = newstring
    EndPos = EndPos + 1
    If EndPos > MaxStrings Then EndPos = 1
    If EndPos = StartPos Then
        StartPos = StartPos + 1
        If StartPos > MaxStrings Then StartPos = 1
    End If
End Sub

Private Sub mnuClear_Click()
    StartPos = 1
    EndPos = 1
    UpdateList
End Sub

' Optionen-Dialog anzeigen
Private Sub mnuOptions_Click()
    Hide
    frmOptions.Show 1
    Show
End Sub

Private Sub mnuUpdate_Click()
    UpdateList
End Sub

Private Sub Timer1_Timer()
    UpdateList
End Sub

' Inhalt in Clipboard kopieren
Private Sub mnuCopy_Click()
    Dim counter&
    Dim s$
    counter = StartPos
    Do While counter <> EndPos
        s$ = s$ & StringArray(counter) & vbCrLf
        counter = counter + 1
        If counter > MaxStrings Then
```

```
            counter = 1
        End If
    Loop
    Clipboard.SetText s$
End Sub
```

Listing 14.2: Code des Hauptformulars frmDebug (DmDeb.frm)

Eine einfachere Implementierung dieser Art eines Programms wäre das einfache Einfügen der Informationen direkt in die ListBox. Dieser Ansatz hätte jedoch zwei schwerwiegende Nachteile:

- Es bestünde die Gefahr, daß die Performance sinkt, wenn die ListBox mit Nachrichten gefüllt wird. Oder Sie müßten den Overhead einkalkulieren, wenn Sie die Anzahl der ListBox-Einträge prüfen und sicherstellen wollen, daß die Füllmenge nicht unhandlich wird.
- Es entstünde ein grundsätzlicher Overhead beim Einfügen von Daten in die ListBox. Vor allem, wenn der Monitor während eines Benchmark-Laufs einer zu verfolgenden Anwendung in Betrieb ist, könnte sich der Overhead bei jedem Einfügen von Daten in die ListBox verfälschend auswirken.

Um diese Probleme zu vermeiden, verwaltet die Komponente ein separates Array namens `StringArray`. Die Methode `SetArraySize` legt die maximale Größe des Arrays fest und initialisiert die Variablen `StartPos` und `EndPos`. Über diese Variablen wird ein Ringpuffer (First-in-first-out) implementiert. Die Variable `StartPos` zeigt auf das erste Element im Puffer, die Variable `Endpos` zeigt auf die Stelle, an der das nächste zu ladende Element eingefügt werden soll. Wenn beide Variablen gleich sind, ist der Puffer leer.

Die Methode `UpdateList` leert die ListBox und lädt den Inhalt des aktuellen Puffers in einer Schleife von `StartPos` bis `EndPos`. Die Prozedur hält die aktuelle Anzeigeposition nach, so daß kein exzessives Scrolling auftritt. Diese Methode wird aufgerufen, wenn der Timer abgelaufen ist oder wenn im Menü UPDATE gewählt wird (`mnuUpdate_Click`).

Der Befehl `mnuClear` löscht den aktuellen Pufferinhalt. Der Befehl `mnuCopy` kopiert den Inhalt des Puffers in die Zwischenablage. Im Formular `frmOptions` (siehe Listing 14.3) können Sie den Timer ein- und ausschalten und das Timer-Intervall auf einen Wert von 1 bis 60 Sekunden einstellen.

```
' DebugMonitor trace program
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved
Option Explicit

Private Sub cmdOK_Click()
    Dim delayval&
    delayval = Val(txtInterval)
```

```
    If chkAutoUpdate.Value Then
        frmDebug.Timer1.Enabled = True
        If delayval < 1 Or delayval > 60 Then
            MsgBox "Select a delay between 1 and 60", _
             vbOKOnly, "Invalid value"
            Exit Sub
        End If
        frmDebug.Timer1.Interval = _
         Val(txtInterval.Text) * 1000
    Else
        frmDebug.Timer1.Enabled = False
    End If
    Unload Me
End Sub

Private Sub Form_Load()
    If frmDebug.Timer1.Enabled Then
        chkAutoUpdate.Value = 1
    End If
    txtInterval.Text = frmDebug.Timer1.Interval / 1000
End Sub
```

Listing 14.3: Das Formular frmOptions (Datei DmOpt.frm)

Die CheckBox `chkAutoUpdate` setzt die `Enabled`-Eigenschaft des Timers. In der TextBox `txtInterval` wird das Timer-Intervall angegeben. Die Schaltfläche `cmdOK` übernimmt die Einstellungen ins Hauptformular.

Im nächsten Abschnitt werden Sie sehen, wie die Komponente eingesetzt wird.

14.4 Multithreading-Beispiele

Mit Multithreading muß man erst vertraut werden. Sie haben zwar bereits grafische Darstellungen gesehen, aber noch keine Umsetzung in Code. Wir werden uns den Code und die Ergebnisse ansehen, um zu verstehen, was tatsächlich vor sich geht.

14.4.1 Der Beweis liegt im Timing

Ein Multithreading-EXE-Server ist von Visual Basic aus leicht zu testen. Legen Sie einfach von verschiedenen Anwendungen aus dessen Objekte an und Sie sehen, daß diese einander nicht blockieren. Doch wie können Sie einen Multithreading-DLL-Server testen? Der einzige Weg, die Vorteile von Multithreading in einer DLL wahrzunehmen, führt über eine Multithreading-Client-Anwendung.

Für unsere ersten Tests erstellen wir einen scheinbaren Multithreading-Client. Wir verwenden eine Standard-EXE-Anwendung (die ja kein Multithread-Client sein kann) und lassen von dieser Objekte in einem Multithreading-EXE-Server

mit der Einstellung THREAD PRO OBJEKT anlegen. Dann testen wir den DLL-Server über die verschiedenen Threads des EXE-Servers. Sie werden dann später noch sehen, wie man in Visual Basic einen echten Multithreading-Client erstellen kann.

Wir werden neben dem bereits beschriebenen Debug-Monitor vier einzelne Projekte zur Durchführung dieser Tests erstellen.

MT3.VBP und MT4.VBP

Beginnen wir auf der DLL-Seite mit den Projekten `MT3.VBP` und `MT4.VBP`. Dies sind beides ActiveX-DLL-Server.

Alle diese Projekte verwenden zur Zeitmessung die Klasse `clsElapsedTime` aus Kapitel 12. Diese wird hier nicht noch einmal beschrieben.

In Listing 14.4 sehen Sie den Code der Klasse `ClassMT4`. Er ist identisch mit dem der Klasse `ClassMT3`. Die einzigen Unterschiede zwischen ihnen in den Projekten `MT3` und `MT4` bestehen in dem jeweiligen Klassennamen, der Projekt-Beschreibung und darin, daß für das Projekt `MT4` Apartment-Threading eingestellt ist – dieses ist also multithreaded.

```
Option Explicit

Private CurrentMessage$
Private InProgress As Boolean
Private DebugMon As DebugMonitor.Trace

Private Declare Function GetCurrentThreadId _
 Lib "kernel32" () As Long

' langandauernde Operation ausführen
' Ausführungsdauer messen und melden
Public Sub LongOp(msg$)
    Dim ctr&
    Dim x&
    Dim s$
    Dim Elapsed As New clsElapsedTime
    CurrentMessage = msg
    ' Eine richtig lange dauernde Operation,
    ' die nicht optimiert werden kann.
    Elapsed.StartTheClock

    For ctr = 1 To 5000
        For x = 1 To 255
            s$ = Chr$(x)
        Next x
    Next ctr
    Elapsed.StopTheClock
```

```
    Report Elapsed
End Sub

' Wir tun nichts weiter, als den String zu übergeben
Public Sub ShortOp(ByVal msg$)

End Sub

Public Sub ShowTID(ByVal msg$)
    DebugMon.Add msg & " TID: " & GetCurrentThreadId()
End Sub

Private Sub Class_Initialize()
    Set DebugMon = New DebugMonitor.Trace
End Sub

Private Sub Class_Terminate()
    Set DebugMon = Nothing
End Sub

' Aktuelle Nachricht und verstrichene Zeit melden
Friend Sub Report(elp As clsElapsedTime)
    Dim msg$
    msg$ = CurrentMessage$ & " Time: " & elp.Elapsed _
     & " TID: " & GetCurrentThreadId()
    DebugMon.Add msg
End Sub
```

Listing 14.4: Die Klassen ClassMT4 (mt4cls4.cls) und ClassMT3 (mt2cls3.cls)

Die Methode `LongOp` führt eine langandauernde Operation aus, die auf einer mittleren Pentium-Maschine etwa 5 Sekunden dauern dürfte. Sie können den Wert des Schleifenzählers variieren, um gegebenenfalls auf eine ähnliche Dauer zu kommen. Die `String`-Operationen in dieser Funktion erfüllen keinen besonderen sinnvollen Zweck. Ohne diese `String`-Operationen würde die Native-Code-Optimierung den größten Teil der Schleife einfach wegoptimieren.

Die Methode `ShortOp` tut schlichtweg gar nichts. Damit wird eine Methode demonstriert, die zwar schnell bearbeitet wird, aber wegen des `String`-Parameters ein Marshaling der Daten erfordert und somit die Performance-Auswirkungen von Thread-übergreifendem Marshaling zeigt.

Das `DebugMon`-Objekt steht dem Projekt über eine Referenz auf die `DebugMonitor`-Komponente zur Verfügung. Das `Trace`-Objekt wird zur Übergabe einer Nachricht an die `DebugMonitor`-Komponente verwendet. Die Methode `ShowTID` übergibt der `DebugMonitor`-Komponente die Identifikation des jeweiligen Threads.

Jeder Thread im System trägt eine individuelle Kennung. Diese Kennung erhalten Sie wie hier über die API-Funktion `GetCurrentThreadID()` oder über die Eigenschaft `ThreadId` des `App`-Objekts.

MT2.VBP

Die beiden DLL-Server werden mit Hilfe des EXE-Servers getestet (unter Anweisung eines Test-Programms). Das Projekt `MT2.VBP` stellt einen ActiveX-EXE-Server dar, der einen neuen Thread für jedes Objekt anlegen soll. Den Code des Klassen-Moduls `ClassMT2` sehen Sie in Listing 14.5. Die `Instancing`-Eigenschaft ist für diese Klasse auf `MultiUse` gesetzt. Das Projekt enthält Verweise auf das DebugMonitor-Objekt und die Projekte `MT3` und `MT4`.

```
' Guide to the Perplexed - Multithreading EXE example
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved
'
Option Explicit

Private CurrentMessage$
Private InProgress As Boolean
Private DebugMon As DebugMonitor.Trace

' Führt eine langandauernde Operation aus
' Mißt die Dauer und meldet sie
Public Sub LongOp(msg$)
    Dim ctr&
    Dim x&
    Dim s$
    Dim Elapsed As New clsElapsedTime
    CurrentMessage = msg
    ' Eine lang andauernde Operation,
    ' die nicht optimiert werden kann
    Elapsed.StartTheClock

    For ctr = 1 To 1000
        For x = 1 To 255
            s$ = Chr$(x)
        Next x
    Next ctr
    Elapsed.StopTheClock
    Report Elapsed
End Sub

Private Sub Class_Initialize()
    ' Objekte vorher laden, damit der Ladevorgang nicht
    ' in die Zeitmessung eingeht
    Set DebugMon = New DebugMonitor.Trace
End Sub
```

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

' Aktuelle Nachricht und verstrichene Zeit weitergeben
Friend Sub Report(elp As clsElapsedTime)
    Dim msg$
    msg$ = CurrentMessage$ & " Time: " & elp.Elapsed _
     & " TID: " & App.ThreadID
    DebugMon.Add msg
End Sub

' Langandauernde Operation in einer
' Nicht-Multithread-DLL aufrufen
Public Sub CallDllNoMTLong(ByVal msg$)
    Dim dllNonMT As New gtpMT3.ClassMT3
    dllNonMT.LongOp msg
End Sub

' Langandauernde Operation in einer Multithread-DLL
' aufrufen
Public Sub CallDllMTLong(ByVal msg$)
    Dim dllMT As New gtpMT4.ClassMT4
    dllMT.LongOp msg
End Sub

' Nicht-Multithread-DLL aufrufen
Public Sub CallDllNoMT(ByVal msg$)
    Dim Elapsed As New clsElapsedTime
    Dim dllNonMT As New gtpMT3.ClassMT3
    Dim x&
    CurrentMessage = msg
    dllNonMT.ShowTID "CallDllNoMT in TID: " _
     & App.ThreadID & " on DLL object"
    Elapsed.StartTheClock
    For x = 1 To 4000
        dllNonMT.ShortOp "This string must be marshaled"
    Next x
    Elapsed.StopTheClock
    Report Elapsed
End Sub

' Multithread-DLL aufrufen
Public Sub CallDllMT(ByVal msg$)
    Dim Elapsed As New clsElapsedTime
    Dim x&
```

```
    Dim dllMT As New gtpMT4.ClassMT4
    CurrentMessage = msg
    dllMT.ShowTID "CallDllMT in TID: " & App.ThreadID _
     & " on DLL object"
    Elapsed.StartTheClock
    For x = 1 To 400000  ' Achtung: um Faktor 10
                         ' langsamer
        dllMT.ShortOp "This string must be marshaled"
    Next x
    Elapsed.StopTheClock
    Report Elapsed
End Sub
```

Listing 14.5: Die Klasse ClassMT2 (mt2cls2.cls)

Die Methode `LongOp` dieser Klasse ist im wesentlichen identisch zu derjenigen der Klassen in den Projekten `MT3` und `MT4`. Mit ihr wird gezeigt, daß sich jedes Objekt in einem Thread befindet. Das Objekt `DebugMon` dient auch hier wieder zum Zugriff auf die `DebugMonitor`-Komponente.

Die Methoden `CallDllMTLong` und `CallDllNoMTLong` legen jeweils `MT3`- und `MT4`-Objekte an und rufen deren `LongOp`-Methoden auf. Somit können wir die Unterschiede zwischen Objekten in Multithread-DLL-Servern und Nicht-Multithread-DLL-Servern sehen.

Die Methoden `CallDllMT` und `CallDllNoMT` führen wiederholte Aufrufe der `ShortOp`-Methoden der Multithread- und Nicht-Multithread-DLL-Server-Objekte auf. Damit zeigen wir die Performance-Auswirkungen des Thread-übergreifenden Marshalings. Der einzige Unterschied zwischen den beiden Methoden besteht darin, daß die Methode auf einem Multithread-Server 100mal öfter aufgerufen wird, um einen Eindruck davon zu vermitteln, wie groß die Marshaling-Auswirkungen tatsächlich sind.

MTTest1.VBP

Dieses Projekt enthält Verweise auf die Komponenten `DebugMonitor` und `MT2`. Es enthält dazu die Klasse `clsElapsedTime` zur Messung der bei jedem Test verstrichenen Zeit. Das Hauptformular dieses Projekts sehen Sie in Abbildung 14.9. Den Code des Formulars sehen Sie in Listing 14.6.

Die Schaltflächen führen folgende Aufgaben aus:

- EXE LOOP TEST mißt die Zeit für einen `LongOp`-Aufruf beim MT2-Server.
- DLL (MANY SHORT) mißt die Zeit für Aufrufe von `CallDllNoMT` und `CallDllMT` beim MT2-Server.
- DLL (LONG) mißt die Zeit für Aufrufe von `CallDllNoMTLong` und `CallDll-MTLong` beim MT2-Server.

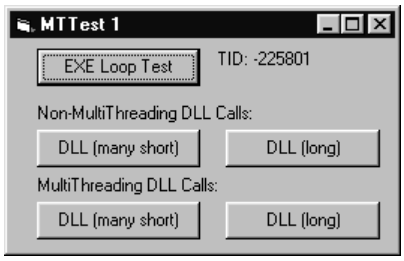

Abb. 14.9: Das Hauptformular des Projekts MTTest1 in Aktion

```
' Guide to the Perplexed - Multithreading test program
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit
Dim MT2obj As ClassMT2
Dim elapse As New clsElapsedTime
Dim debugmon As DebugMonitor.Trace

Private Sub cmdDLLMT_Click()
    elapse.StartTheClock
    MT2obj.CallDllMT "Called from TID: " & App.ThreadID
    elapse.StopTheClock
    debugmon.Add "Elapsed on TID: " & App.ThreadID _
     & " was " & elapse.Elapsed
End Sub

Private Sub cmdDllMTLong_Click()
    elapse.StartTheClock
    MT2obj.CallDllMTLong "Long - Called from TID: " _
     & App.ThreadID
    elapse.StopTheClock
    debugmon.Add "Elapsed on TID: " & App.ThreadID _
     & " was " & elapse.Elapsed
End Sub

Private Sub cmdDLLNoMT_Click()
    elapse.StartTheClock
    MT2obj.CallDllNoMT "Called from TID: " _
     & App.ThreadID
    elapse.StopTheClock
    debugmon.Add "Elapsed on TID: " & App.ThreadID _
     & " was " & elapse.Elapsed
End Sub

Private Sub cmdDllNonMTL_Click()
    elapse.StartTheClock
```

```
    MT2obj.CallDllNoMTLong "Long - Called from TID: " _
     & App.ThreadID
    elapse.StopTheClock
    debugmon.Add "Elapsed on TID: " & App.ThreadID _
     & " was " & elapse.Elapsed
End Sub

Private Sub cmdTest1_Click()
    elapse.StartTheClock
    MT2obj.LongOp "Called from TID: " & App.ThreadID
    elapse.StopTheClock
    debugmon.Add "Elapsed on TID: " & App.ThreadID _
     & " was " & elapse.Elapsed
End Sub

Private Sub Form_Load()
    ' Die Ladezeit wird nicht mitgemessen
    Set debugmon = New DebugMonitor.Trace
    Set MT2obj = New ClassMT2
    lblTID = "TID: " & App.ThreadID
End Sub

Private Sub Form_Unload(Cancel As Integer)
    Set MT2obj = Nothing
    Set debugmon = Nothing
End Sub
```

Listing 14.6: Listing des Formulars frmMTTest1 (MTTest1.frm)

14.4.2 Test-Beginn

Stellen Sie zunächst sicher, daß die Server alle kompiliert und registriert sind. Starten Sie dazu die Programme `MT2.EXE` und `DmDebMon.EXE` und registrieren Sie die DLLs `MT3.DLL` und `MT4.DLL` mit Hilfe von `regsrv32.EXE`.

Testen des Multithreading-EXE-Servers

Starten Sie drei Instanzen des Programms `MTTest1`. Dadurch wird auch das Fenster des Debug-Monitors geöffnet (dessen Titel lautet TRACE DISPLAY). Ordnen Sie die drei Testfenster so an, daß Sie leicht darauf zugreifen können.

Jedes `MTTest1`-Programm zeigt seine eigene Thread-Kennung an. Da es Visual Basic-Standard-EXEs sind, haben sie nur jeweils einen Thread, so daß die Thread-Kennung auch das jeweilige Programm eindeutig kennzeichnet.

Klicken Sie nun auf die Schaltfläche EXE LOOP TEST in allen drei Instanzen so schnell Sie können. Beachten Sie, daß die hier angeführten Zeiten auf meinem System gemessen wurden – sie werden auf Ihrem System sicher davon abweichen. Es ergaben sich folgende Resultate:

```
Called from TID: 243 Time: 1733. TID: 241
Elapsed on TID: 243 was 1933.
Called from TID: 108 Time: 2304. TID: 242
Elapsed on TID: 108 was 2474.
Called from TID: 236 Time: 2003. TID: 213
Elapsed on TID: 236 was 2063.
```

Die erste Nachricht kam wie folgt zustande: Die Nachricht »Called Form TID: 243« wurde in der Prozedur cmdTest1_Click erzeugt. Dies identifiziert den ursprünglichen Thread. Die ermittelte Zeit für den LongOp-Aufruf beim Server MT2 betrug 1733 ms. Diese Zeit hat der Server in der Schleife verbracht. Die LongOp-Routine lief im Thread 241. Nach Abschluß der Operation wurde als Gesamtzeit 1933 ms gemessen.

Um eine Vorstellung davon zu bekommen, wie lange eine einzelne LongOp-Operation dauert, klicken Sie auf die Schaltfläche EXE LOOP TEST einer der MTTest1-Instanzen. Die Ergebnisse:

```
Called from TID: 243 Time: 1052. TID: 241
Elapsed on TID: 243 was 1062.
```

Ungefähr 1062 ms. Würden die drei LongOp-Operationen nacheinander ausgeführt, sollte jede also etwa 1 Sekunde lang dauern. Tatsächlich ergaben sich jedoch in allen drei Fällen gleichermaßen etwa 2 Sekunden, wobei die Gesamtzeit auch etwa 2 Sekunden betrug. Aber dies ist genau das zu erwartende Ergebnis unter der Annahme, daß jedes Objekt in einem eigenen Thread läuft – was auch die ausgegebenen Thread-Kennungen bezeugen.

Das vielleicht wichtigste Ergebnis des Tests ist, daß wir nachgewiesen haben, daß wir es tatsächlich mit einem einzigen Prozeß (dem EXE-Server) zu tun haben, in dem drei Threads zur gleichen Zeit liefen. Wir werden nun diese Threads zum Test der DLL-Server verwenden.

Testen der Auswirkungen von thread-übergreifendem Marshaling

Klicken Sie nun auf die Schaltfläche DLL (MANY SHORT) unter NON-MULTITHREADING DLL CALLS bei den drei Instanzen von MTTest1. Die Ergebnisse lauten (auch hier können wieder die Werte von den Ihren etwas abweichen):

```
CallDllNoMT in TID: 241 on DLL object TID: 247
CallDllNoMT in TID: 242 on DLL object TID: 247
CallDllNoMT in TID: 213 on DLL object TID: 247
Called from TID: 243 Time: 8603. TID: 241
Elapsed on TID: 243 was 9173.
Called from TID: 108 Time: 8922. TID: 242
Elapsed on TID: 108 was 9063.
Called from TID: 236 Time: 6329. TID: 213
Elapsed on TID: 236 was 11106.
```

Jede Instanz generiert drei Nachrichten. Verfolgen wir einmal Thread 108.

In `cmdDLLNoMT_Click` wird zuerst die Nachricht »`Called Form TID: 108`« an den Server übergeben. Der Server speichert diese Nachricht. Über die Methode `ShowTID` läßt er Sie wissen, wer ihn aufgerufen hat:

```
CallDllNoMT in TID: 242 on DLL object TID: 247
```

Dann ruft er die Methode `ShortOp` des DLL-Objekts 4000mal auf. Danach meldet er die gemessene Zeit:

```
Called from TID: 108 Time: 8922. TID: 242
```

Er verwendet die früher gespeicherte Nachricht. Wir wissen, daß Thread 108 in der `MTTest1`-Instanz das Server-Objekt in Thread 242 zum Aufruf der `ShortOp`-Methode eines DLL-Server-Objekts in Thread 247 verwendet. Die `ShortOp`-Schleife dort dauert 8,9 Sekunden. Schließlich kehrt der Aufruf zum `MTTest1`-Programm zurück und gibt folgende Nachricht aus:

```
Elapsed on TID: 108 was 9063
```

Die Gesamtzeit zwischen Klick und Rückkehr beträgt also etwa 9 Sekunden.

Es ist interessant zu beobachten, daß alle Tests gleich lange dauern und auch die Gesamtzeit die gleiche ist. Wie kann das sein, wo wir hier doch keinen Multithread-DLL-Server verwendet haben? Ganz einfach: Wir führen keine langandauernden Operationen aus. Objekte in einem Thread blockieren den Thread nur während des eigentlichen Methoden-Aufrufs. Die Methoden-Aufrufe in der DLL sind extrem kurz, so daß es keinen Grund dafür gibt, daß nicht alle drei EXE-Server-Threads zum Zuge kommen und dabei jeder wiederum sein DLL-Objekt aufruft.

Beachten Sie, daß alle DLL-Objekte in Thread 247 laufen, in dem EXE-Server-Thread, der die DLL-Komponente geladen hat. Es gibt zwar noch einen anderen Thread in dem EXE-Server, der jedoch nur für die »Innereien« der Server-Verwaltung und nicht für einzelne Objekte zuständig ist.

Versuchen Sie dieselbe Operation über die Schaltfläche DLL (MANY SHORT) in der Multithreading-Gruppe. Dies sind nun die Ergebnisse:

```
CallDllMT in TID: 241 on DLL object TID: 241
CallDllMT in TID: 242 on DLL object TID: 242
CallDllMT in TID: 213 on DLL object TID: 213
Called from TID: 243 Time: 3044. TID: 241
Elapsed on TID: 243 was 3435.
Called from TID: 108 Time: 3786. TID: 242
Elapsed on TID: 108 was 4376.
Called from TID: 236 Time: 2914. TID: 213
Elapsed on TID: 236 was 3895.
```

Der größte Unterschied wird hier bei der Thread-Kennung der DLL-Objekte offensichtlich. Sie sehen, daß jedes Objekt in dem gleichen Thread läuft wie das aufrufende EXE-Server-Objekt. Dies ist in den ersten drei Nachrichten zu sehen.

Auf den ersten Blick scheint die Operation nur unmerklich schneller zu sein – 3 Sekunden anstatt 9 Sekunden. Doch schauen Sie sich noch einmal genau das Listing an: Wir führen hier nicht nur 4000 Aufrufe von `ShortOp` aus, sondern 400.000 – der Performance-Gewinn beträgt also das 200fache!

Daraus folgt unweigerlich: Wenn Sie einen DLL-Server schreiben (ActiveX-Controls und Web-Klassen eingeschlossen), der von einem Multithread-Client wie einem Web-Server oder einem Web-Browser verwendet werden soll, kann Multithreading die Performance erheblich steigern. Natürlich ist das hier nur ein Ideal-Szenario gewesen, in dem eine leere Funktion aufgerufen wurde. Der Gewinn in einer realen Anwendung wird um einiges geringer ausfallen, abhängig auch davon, wie hoch der Anteil des thread-übergreifenden Marshalings an der Gesamtdauer eines Methoden- bzw. Eigenschaften-Zugriffs ist.

Testen von langandauernden Operationen bei DLL-Servern

Klicken Sie nun die Schaltfläche DLL (LONG) in der Gruppe NON-MULTITHREADING DLL-CALLS an. Die Ergebnisse:

```
Long - Called from TID: 243 Time: 5698. TID: 247
Long - Called from TID: 236 Time: 5508. TID: 247
Elapsed on TID: 236 was 8803.
Elapsed on TID: 243 was 11427.
Long - Called from TID: 108 Time: 5508. TID: 247
Elapsed on TID: 108 was 15793.
```

Hier sehen Sie ganz klar die Auswirkungen des Blockierens. Die Dauer eines `LongOp`-Aufrufs eines DLL-Objekts beträgt in allen Fällen etwa 5,5 Sekunden. Die Gesamtdauer jeder Operation hat sich verlängert, da jede von der vorhergehenden langen Operation blockiert wird. Nachdem die erste langandauernde Operation abgeschlossen ist, entsteht eine weitere Verzögerung, bevor die Gesamtzeit gemessen werden kann, da der andere Thread noch CPU-Zeit bindet. Sie sehen, daß die Gesamtdauer für alle drei Operationen etwa 15,7 Sekunden beträgt, nur etwas mehr als die Summe der jeweiligen Dauer der drei Einzeloperationen.

Probieren Sie nun die Schaltfläche DLL (LONG) für Multithread-Aufrufe.

```
Long - Called from TID: 243 Time: 14621. TID: 241
Elapsed on TID: 243 was 15162.
Long - Called from TID: 108 Time: 15302. TID: 242
Elapsed on TID: 108 was 15963.
Long - Called from TID: 236 Time: 15072. TID: 213
Elapsed on TID: 236 was 15813.
```

Die Dauer jedes einzelnen Aufrufs und die Gesamtzeit sind nahezu gleich. Diese Situation habe ich bereits weiter oben in diesem Kapitel beschrieben. Der Nachweis wird tatsächlich durch die Messungen erbracht.

14.5 Hintergrundoperationen

Sie haben gesehen, daß Sie mit Visual Basic Multithread-Komponenten erstellen können. Schauen wir uns einmal einen Ansatz dafür an, wie eine Anwendung Hintergrundoperationen in einem separaten Thread nutzen kann.

Was geschieht, wenn Sie ein Objekt in einem Multithread-Server anlegen, eine Methode dieses Objekts aufrufen und sofort zurückkehren, während das Objekt mit einer Hintergrundoperation beginnt und Sie benachrichtigt, wenn diese abgeschlossen ist? Nun, aus praktischer Sicht haben Sie lediglich einen neuen Thread Ihrer Anwendung ins Leben gerufen.

Sie haben dies bereits mit einem `SingleUse`-EXE-Server im Projekt `Tick1` in Kapitel 11 gesehen. Funktioniert dies auch mit `MultiUse`-Klassen in einem Multithread-EXE-Server? Ja, aber mit ein paar Einschränkungen.

In Listing 14.7 sehen Sie eine Implementierung, die derjenigen des `Tick1`-Projekts aus Kapitel 11 sehr ähnelt. Das Standard-Modul zur Implementierung des Timers sehen Sie in Listing 14.8.

```
' Guide to the Perplexed: Background Thread Launcher
' Copyright (c) 1997 by Desaware Inc.
Option Explicit

Private CurrentMessage$
Private InProgress As Boolean
Private debugmon As DebugMonitor.Trace
Private CallerToNotify As Object

' langandauernde Operation ausführen
' Dauer messen und melden
Public Sub LongOp(msg$)
    Dim ctr&
    Dim x&
    Dim s$
    Dim Elapsed As New clsElapsedTime
    CurrentMessage = msg
    ' langandauernde, nicht optimierbare Operation
    Elapsed.StartTheClock

    For ctr = 1 To 5000
        For x = 1 To 255
            s$ = Chr$(x)
        Next x
```

```
    Next ctr
    Elapsed.StopTheClock
    Report Elapsed
End Sub

Private Sub Class_Initialize()
    ' Objekte vorher laden, um den Ladevorgang aus
    ' der Zeitmessung herauszuhalten
    Set debugmon = New DebugMonitor.Trace
End Sub

Private Sub Class_Terminate()
    Set debugmon = Nothing
End Sub

' Aktuelle Nachricht und gemessene Dauer melden
Friend Sub Report(elp As clsElapsedTime)
    Dim msg$
    msg$ = CurrentMessage$ & " Time: " & elp.Elapsed _
     & " TID: " & App.ThreadID
    debugmon.Add msg
End Sub

' Thread-Kennung des Objekt ermitteln
Public Function ObjectThreadId() As Long
    ObjectThreadId = App.ThreadID
End Function

' Hintergrundoperation eines Timers starten
Public Sub StartBackground1(ToNotify As Object)
    Set CallerToNotify = ToNotify
    debugmon.Add "Starting timer from TID: " _
     & App.ThreadID
    StartTimer Me
End Sub

' Hintergrundoperation gestartet
Friend Sub TimerExpired()
    debugmon.Add "Starting Background Op in TID: " _
     & App.ThreadID
    LongOp "Background Op Done "
    ' OLE-Rückruf an Objekt
    If CallerToNotify Is Nothing Then Exit Sub
    CallerToNotify.BackgroundNotify
    Set CallerToNotify = Nothing
End Sub
```

Listing 14.7: Das Klassen-Modul ClssMt5 (Datei cls5Methode.cls) des Projekts MT5

```
' Guide to the Perplexed: MT5
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved

Option Explicit

' Timer-Kennung
Dim TimerID&

' Objekt für diesen Timer
Dim TimerObject As ClassMT5
Declare Function SetTimer Lib "user32" _
 (ByVal hwnd As Long, ByVal nIDEvent
As Long, ByVal uElapse As Long, _
 ByVal lpTimerFunc As Long) As Long
Declare Function KillTimer Lib "user32" _
 (ByVal hwnd As Long, ByVal nIDEvent
As Long) As Long

Public Sub StartTimer(callingobject As ClassMT5)
   Set TimerObject = callingobject
   TimerID = SetTimer(0, 0, 100, AddressOf TimerProc)
End Sub

' Rückruf-Funktion
Public Sub TimerProc(ByVal hwnd&, ByVal msg&, _
 ByVal id&, ByVal currentime&)
   Call KillTimer(0, TimerID)
   TimerID = 0
   TimerObject.TimerExpired
   ' Objekt-Referenz freigeben
   Set TimerObject = Nothing
End Sub
```

Listing 14.8: Modul modMT5.bas

Die Client-Anwendung legt ein Objekt der Klasse `ClassMT5` an und ruft die Methode `StartBackground1` auf, um den Hintergrund-Thread zu starten. Diese Methode erhält eine Referenz auf das Formular-Objekt, das für den OLE-Rückruf verwendet werden soll. Die Methode `StartBackground1` sendet eine Debug-Nachricht an die DebugMonitor-Komponente und ruft dann die `StartTimer`-Funktion im Modul `modMT5` auf. Diese startet einen Kurzzeit-Timer. Danach kehrt die Methode `StartBackground1` zurück.

Wenn der Timer abgelaufen ist, wird die Prozedur `TimerProc` aufgerufen. Diese löscht den Timer und ruft die Methode `TimerExpired` des Objekts `ClassMT5` auf. Diese Methode meldet eine weitere Debug-Nachricht und beginnt danach eine

langandauernde Operation. Diese Operation läuft in einem separaten Thread, dem Thread des EXE-Server-Objekts.

Nach Erledigung der Operation wird eine weitere Debug-Nachricht ausgegeben und es wird die OLE-Rückruf-Methode `BackgroundNotify` aufgerufen.

Das Projekt `MTTest2` sehen Sie in Listing 14.9. Darin wird ein Objekt der `MT5`-Komponente angelegt und die Hintergrundoperation mit einem Klick auf die TEST-Schaltfläche gestartet. Wenn die Operation abgeschlossen ist, wird dies über einen Aufruf der Methode `BackgroundNotify` gemeldet, und es wird die Gesamtdauer im Debug-Monitor angezeigt.

```
' Guide to the Perplexed: Background thread demonstration
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved
Option Explicit

Dim MTTestObj As gtpMT5.ClassMT5
Dim Debugmon As DebugMonitor.Trace
Dim Elapsed As New clsElapsedTime

Private Sub cmdTest_Click()
    Elapsed.StartTheClock
    MTTestObj.StartBackground1 Me
End Sub

Private Sub Form_Load()
    Set MTTestObj = New gtpMT5.ClassMT5
    Set Debugmon = New DebugMonitor.Trace
    lblTID.Caption = "TID: " & App.ThreadID
End Sub

Private Sub Form_Unload(Cancel As Integer)
    Set MTTestObj = Nothing
    Set Debugmon = Nothing
End Sub

Public Sub BackgroundNotify()
    Elapsed.StopTheClock
    Debugmon.Add "Total elapsed in TID: " _
     & App.ThreadID & " was: " & Elapsed.Elapsed
End Sub
```

Listing 14.9: Das Projekt MTTest2

Nachdem Sie die Komponente `MT5.EXE` registriert haben, starten Sie zwei Instanzen des Programms `MTTest2`. Klicken Sie dann in beiden auf die Schaltfläche BACKGROUND USING TIMER. Hier einige typische Resultate:

```
Starting timer from TID: 59
Starting Background Op in TID: 59
Starting timer from TID: 213
Starting Background Op in TID: 213
Background Op Done  Time: 9794. TID: 59
Total elapsed in TID: 223 was: 10024.
Background Op Done  Time: 9824. TID: 213
Total elapsed in TID: 77 was: 10184.
```

Wie Sie sehen, dauern beide Hintergrundoperationen etwa 10 Sekunden, wobei die Gesamtdauer auch etwa 10 Sekunden beträgt – beide Operationen sind in verschiedenen Threads ausgeführt worden.

14.5.1 Eine Multithread-Benutzeroberfläche

Mit Visual Basic können Sie echte Multithread-Client-Anwendungen schreiben. Sie sollten aber vorher gründlich darüber nachdenken. Ein Multithread-Client-Programm kann auf Anwender recht verwirrend wirken, wenn es nicht sorgfältig entworfen worden ist, so daß die einzelnen Benutzeroberflächen-Elemente deutlich voneinander unabhängig erscheinen. Bei einem Web-Browser ist es dem Anwender klar, daß verschiedene Browser-Fenster in verschiedenen Threads laufen können. Jedes Browser-Fenster verhält sich wie eine eigene Instanz des Programms. Tatsächlich kann ein Anwender nicht unterscheiden, ob die verschiedenen Browser-Fenster in separaten Threads oder in separaten Prozessen laufen.

Eine Standard-EXE-Anwendung kann nicht als Multithread-Client fungieren. Ich habe jedoch behauptet, daß Sie mit Visual Basic Multithread-Clients erstellen können. Wie denn nun? Der Trick liegt darin, daß ein ActiveX-EXE-Server auch als eigenständige Anwendung laufen kann.

Das Projekt `MTDemo2.vbp` zeigt, wie das realisiert werden kann. In Listing 14.10 sehen Sie das Klassen-Modul `clsMTDemo2`. Das Modul tut nur sehr wenig, doch das wenige ist von Bedeutung. Die Methode `DummyOperation` tut wirklich nichts. Sie existiert nur, damit der Client sicher sein kann, daß das Objekt tatsächlich angelegt wird. Warum ist dies notwendig? Erinnern Sie sich daran, daß bei einer `New`-Operation ein Objekt nicht vor dem ersten Methoden-Aufruf tatsächlich angelegt wird. Wenn die Instanz angelegt wird, lädt und öffnet sie ein Formular. In welchem Thread läuft das Formular? Visual Basic verwendet das Apartment-Modell, so daß jedes Objekt in dem Thread läuft, von dem es angelegt wird. Daher wird das Formular im gleichen Thread laufen wie das `clsMTDemo`-Objekt.

```
' MTDemo2 - Multithreading demo program
' Copyright © 1997 by Desaware Inc.  All Rights Reserved

Option Explicit

Private Sub Class_Initialize()
   Dim f As New frmMTDemo2
```

```
   f.Show
   Set f = Nothing
End Sub

Public Sub DummyOperation()
End Sub
```

Listing 14.10: Die Klasse clsMTDemo2

Schauen wir uns das Formular in Abbildung 14.10 an. Das Formular zeigt den Thread an, in dem es läuft. Es enthält drei Schaltflächen. Den Code des Formulars sehen Sie in Listing 14.11.

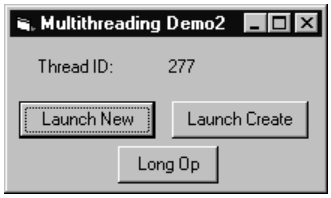

Abb. 14.10: Formular frmMTDemo2 zur Laufzeit

```
' MTDemo2 - Multithreading demo program
' Copyright © 1997 by Desaware Inc.  All Rights Reserved

Option Explicit

Private Sub cmdLaunch1_Click()
   Dim c As New clsMTDemo2
   c.DummyOperation
End Sub

Private Sub cmdLaunch2_Click()
   Dim c As clsMTDemo2
   Set c = CreateObject("MTDemo2.clsMTDemo2")
   c.DummyOperation
End Sub

Private Sub cmdLongOp_Click()
   Dim l&
   Dim s$
   For l = 1 To 2000000
      s = Chr$(l And &H7F)
   Next l
End Sub
```

```
Private Sub Form_Load()
   lblThread.Caption = Str$(App.ThreadID)
End Sub
```

Listing 14.11: Code des Formulars frmMTDemo2

Zwei der Schaltflächen können neue Instanzen des `clsMTDemo2`-Objekts starten. Die eine verwendet die `New`-Anweisung, die andere die `CreateObject`-Anweisung. Sie werden gleich sehen, daß das Verhalten unterschiedlich ist. Sie werden feststellen, daß das Objekt sofort freigegeben wird, sobald die Ereignis-Prozedur verlassen wird. Bedeutet dies, daß der neue Thread (falls einer angelegt worden ist), beendet wird? Nein, da das Objekt während seiner Initialisierung ein Formular geöffnet hat. Das Objekt wird tatsächlich zerstört, das Formular existiert jedoch solange, bis der Anwender es schließt. Der Befehl `cmdLongOp` führt lediglich eine längere, nicht unterbrochene Operation aus. Wählen Sie den Wert des Schleifenzählers so, daß die Dauer wahrnehmbar wird. Dies bindet den Thread des Formulars für eine gewisse Zeit.

Das Kern-Modul dieses Programms ist `modMTDemo2`:

```
' MTDemo2 - Multithreading demo program
' Copyright © 1997 by Desaware Inc.  All Rights Reserved
Option Explicit
Declare Function FindWindow Lib "user32" Alias _
 "FindWindowA" (ByVal lpClassName As String, _
 ByVal lpWindowName As String) As Long

Sub Main()
   Dim f As frmMTDemo2
   ' Notwendig, da Sub Main für jeden neuen Thread
   ' aufgerufen wird
   Dim hwnd As Long
   hwnd = FindWindow(vbNullString, "Multithreading Demo2")
   If hwnd = 0 Then
      Set f = New frmMTDemo2
      f.Show
      Set f = Nothing
   End If
End Sub
```

Das Programm `MTDemo2` startet mit `Sub Main` und läuft als eigenständige Anwendung. Beim ersten Start wird über die API-Funktion `FindWindow` geprüft, ob das Formular `frmMTDemo2` bereits sichtbar ist. Wenn nicht, zeigt die Anwendung das Formular an. Die `Main`-Prozedur wird bei jedem Start eines neuen Threads aufgerufen. Wir wollen jedoch nicht jedesmal hier ein neues Formular öffnen, da es während der Initialisierung der Klasse geöffnet wird.

14.5.2 Testen des Client-Programms

Starten Sie das Programm zunächst in der Visual-Basic-Entwicklungsumgebung. Klicken Sie auf die Schaltflächen LAUNCH NEW und LAUNCH CREATE. Sie sehen, daß die beiden neuen Formulare im gleichen Thread wie das erste laufen. Wie kommt das? Das liegt daran, daß die Visual-Basic-Entwicklungsumgebung, wie bereits erwähnt, selbst kein Multithreading unterstützt. Klicken Sie auf die Schaltfläche LONG OP auf einem der Formulare. Versuchen Sie, eines der anderen Formulare zu verschieben, während die Operation noch ausgeführt wird. Sie werden feststellen, daß es sich erst dann verschiebt, wenn die Operation abgeschlossen ist.

Versuchen Sie nun dasselbe mit einer kompilierten Version des Projekts. Das über die Schaltfläche LAUNCH NEW geöffnete Formular befindet sich im gleichen Thread wie das Original-Formular. Das liegt daran, daß beim New-Operator neue Objekte jeweils im aufrufenden Thread angelegt werden. Das von der Schaltfläche LAUNCH CREATE angelegte Formular läuft dagegen in einem eigenen Thread. Klicken Sie nun auf die Schaltfläche `LONG OP` auf dem Original-Formular und versuchen Sie dann die beiden anderen Formulare zu verschieben. Das über LAUNCH NEW geöffnete Formular ist eingefroren, da der Thread blockiert ist. Das andere, über LAUNCH CREATE geöffnete Formular kann dagegen verschoben werden – es läuft in einem unabhängigen Thread.

Ein Problem gibt es aber noch bei Multithread-Clients in Visual Basic. Wenn die Objekte doch das Apartment-Modell verwenden – wie können sie dann Informationen austauschen oder Daten gemeinsam verwenden? Jeder Thread hat ja seine eigene Kopie der globalen Daten der Anwendung. Eine Möglichkeit ist, API-Techniken zum Austausch von Daten zu verwenden. Sie können Nachrichten (Windows-Messages) von Formular zu Formular senden und dazu die SubClassing-Techniken einsetzen, die Sie in Kapitel 22, »Fortgeschrittene Techniken«, kennenlernen werden. Sie können auch die API-Techniken zur gemeinsamen Nutzung von Arbeitsspeicher verwenden. Die gleichen Techniken zur gemeinsamen Nutzung von Arbeitsspeicher zwischen Prozessen können auch zur gemeinsamen Nutzung von Arbeitsspeicher zwischen Threads verwendet werden. Darüber erfahren Sie mehr in den Kapiteln 13 und 14 meines Buchs *Dan Appleman's Visual Basic Programmer's Guide to the Win32 API*. Es werden API-Synchronisationstechniken notwendig, um sicherzustellen, daß nur jeweils ein einziger Thread zur gleichen Zeit auf ein Datenelement zugreifen kann.

Auf eines sollten Sie achten: Vermeiden Sie bei der Verwendung dieser API-Techniken übergreifende Verweise auf Objekte. Dies würde die COM-Threading-Regeln verletzen und Ihre Anwendung extrem instabil werden lassen (auch wenn es zu funktionieren scheint). Das gleiche Problem tritt bei einem API-Aufruf `CreateThread` auf. Details zu diesen Ansätzen, warum Sie sie vermeiden sollten und wie Sie sie dennoch halbwegs sicher verwenden können, wenn Sie meine Warnung zu ignorieren wünschen, finden Sie in meinem Artikel »A Thread to Visual Basic« auf Desawares Web-Site unter `http://www.desaware.COM`.

14.5.3 Multithreading ist keine Hexerei

In diesem Kapitel haben Sie gesehen, daß es eine ganze Reihe mehr an Möglichkeiten für die Implementierung von Komponenten gibt. Ich gebe Ihnen hier noch zwei Punkte als Stoff zum Nachdenken.

Vor Visual Basic 5 wünschten sich viele Visual-Basic-Programmierer nichts sehnlicher als die Möglichkeit, Native-Code zu kompilieren als Lösung all ihrer Performance-Probleme. Aber nur ein kleiner Teil davon, nämlich diejenigen, die sehr Code-intensive Anwendungen entwickelten, erkannte wirklich die Vorteile, die sie erwarteten. Die übrigen fanden, daß Native-Code nicht die alleinseligmachende Lösung sei und erst recht kein Ersatz für gute Programm-Architekturen – und dasselbe gilt auch für Multithreading.

Multithreading kann in bestimmten Fällen seinen Zweck bestens erfüllen. Bei In-Process-Komponenten in Verbindung mit Multithread-Clients ist es sehr nützlich. Auch als Ersatz für SingleUse-EXE-Server-Komponenten, bei denen Objekte keinen übergreifenden, gemeinsamen Datenzugriff benötigen, eignen sie sich bestens. In anderen Fällen kann Multithreading Ihre Anwendung bremsen. Wägen Sie Ihre Entscheidung also sorgfältig ab.

14.5.4 Sie haben größere Kontrolle als Sie denken

Vergessen Sie nicht, daß Sie neben der Festlegung auf Multithreading und den Einstellungen der Klassen-Instancing-Eigenschaft, die zusammen vorgeben, wie Visual Basic Objekte anlegt, immer auch den Server die Objekte für Sie anlegen lassen können. Dies bedeutet eine große Flexibilität in bezug auf die Zuordnung von Objekten zu Threads. Gefällt Ihnen die Visual-Basic-eigene Zuordnung nicht, steuern Sie sie selbst.

Wenn Sie es darauf anlegen, können Sie Ihren Server sowohl als EXE-Server als auch als DLL-Server erstellen (mit unterschiedlichen Server- und Programm-Namen natürlich). Der Server kann eingehende Anfragen prüfen und deren Komplexität feststellen. Langandauernde, komplexe Operationen können von Objekten erledigt werden, die ein Multithread-EXE-Server anlegt. Kurze Operationen können von Objekten erledigt werden, die von einem Nicht-Multithreading-DLL-Server angelegt werden.

Damit schließen wir das Thema »Multithreading« ab. In Kapitel 15, »Die ganze Mannschaft: Der StockQuote-Server«, werden wir vieles aus den vorangegangenen sieben Kapiteln zusammenbringen und uns endlich an die Erstellung der lange versprochenen StockQuoting-Komponente begeben.