Kapitel 9

Komponenten erstellen und testen

9.1 Ein kurzer Blick auf die Benutzeroberfläche 213
9.2 Komponenten erstellen und testen 216
9.3 Verweise und Reihenfolge von Verweisen 231
9.4 Fehlerbehandlung 232

Manchmal kann ich mich des Gedankens nicht erwehren, Microsoft hätte ein hochgeheimes Akronym-Labor. Dies wäre das Labor, in dem Buchstabenkombinationen und Abkürzungen wie etwa OLE, ActiveX, COM oder ISAPI ausgetüftelt würden. Angesichts der Stellung Microsofts in der Industrie wundert es einen nur wenig, daß diese Akronyme in Windeseile in den Sprachschatz der meisten Windows-Programmierer eingehen. Dabei ist gar nicht mal gesagt, daß wir auch wirklich deren Bedeutung verstehen ... Es ist einfach so, daß Klappern zum Handwerk gehört, wenn man ernstgenommen werden (und zu Brot und Lohn gelangen) will.

Leider wirken all diese Akronyme besonders auf Einsteiger, aber auch auf erfahrenere Programmierer sehr abschreckend. Ich erinnere mich noch an meine Zeit als Computer-Lehrling, als ich mich bemühte, den Unterschied zwischen einem 8-Bit- und einem 16-Bit-Bus zu verstehen. Ich konnte ja noch nachvollziehen, daß 16 Bit irgendwie doppelt so groß wie 8 Bit sein sollten. Aber was nun Computer mit Massenverkehrsmitteln zu tun haben sollten, ist mir einfach schleierhaft geblieben!

Heute kann ich darüber lachen, wie ignorant ich seinerzeit gewesen war. Ich fühlte mich so weit hinterm Mond, daß ich befürchtete, niemals aufholen zu können. Doch mittlerweile dürfte es nur wenige Programmierer geben, die sich nicht mindestens ein wenig überfordert, ignorant oder zurückgeblieben fühlen. Die mysteriösen Abkürzungen, die aus dem MURKS (Microsofts Unaussprechliche Redewendungen Kreativ-Studio – Sie sehen, was die können, kann ich allemal ... *[Das Wortspiel im engl. Original: Microsoft's Acronym Labs – MAL, was etwa »schlecht« oder »mangelhaft« bedeutet])* kommen, helfen da auch nicht weiter.

Und was hat das nun mit dem Erstellen und Testen von Komponenten zu tun? Nun, zum Teil bezieht sich das auf die Konfusion, die Microsoft mit der Umtaufe von OLE in ActiveX hervorgerufen hat – und die haben wir ja längst entschärft. Aber zum guten Teil geht das auf meine Reaktion zurück, als ich entdeckte, daß es seit Version 5.0 in Visual Basic sogenannte »Designer« gibt. Dies ist ein Tatbestand, der erst mal verstanden sein will, wenn man mit Visual Basic arbeitet. Es ist ja nicht so, daß die früheren Visual-Basic-Versionen keine Designer gehabt hätten. Nur wurden sie damals noch nicht so genannt. Puristen werden nun wieder einwenden, daß der Ausdruck *Designer* doch kein Akronym, sondern ein technischer Begriff sei. In der Sache haben sie recht – der Begriff kommt aus dem MIST (Microsofts Institut für Sprach-Terminologie *[hier im engl. Original: Microsoft Institute of Terminology – MITE, was auch mit »kleines Ding« oder »Würmchen« übersetzt werden kann]*), welches gerade um die Ecke vom MURKS angesiedelt ist.

Sie werden es vielleicht schon bemerkt haben, daß ich mich in vieler Hinsicht um die Entmystifizierung der Terminologie bemühe und Ihren Blick auf die Tatsachen hinter den Kulissen zu richten versuche.

9.1 Ein *kurzer* Blick auf die Benutzeroberfläche

Es ist nicht nur einmal vorgekommen, daß ich mich gerade an eine Entwicklungsumgebung gewöhnt hatte, und nach der Installation einer neueren Version feststellen mußte, daß nicht nur nichts mehr an seinem gewohnten Platz war, sondern diese neuere Version auch schwieriger zu handhaben und weniger intuitiv war. Das mag vielleicht an meiner eher etwas konservativen Natur liegen. Doch ich finde, wenn die Jungs mich schon dazu zwingen, mich mit einer neuen Benutzeroberfläche, einer neuen Menüorganisation und mit neuen Befehlen anzufreunden, dann sollten sie die neue Oberfläche wenigstens so gut machen, daß man die Vorteile eines Umstiegs auch auf der Stelle einsieht. Da dies aber nur selten der Fall ist, zwinge ich mich immer wieder zähneknirschend zum Lernen der neuen Oberfläche, in der Hoffnung, daß diese wenigstens für eine geraume Zeit erhalten bleibt, damit man auch noch mal zum Arbeiten kommt.

Ich vermute, daß während der Entwicklung von Visual Basic 5.0 das Microsoft-Team für Benutzeroberflächen die Finger im Spiel hatte. Keine Frage, Microsofts Team für Benutzeroberflächen ist das beste der Welt. Ich fürchte jedoch, daß es bei ihnen doch ein wenig daran hapert, komplexe Anwendungen einigermaßen intuitiv bedienbar zu gestalten – auch wenn ich zugeben muß, daß es da in Windows ein paar Dinge gibt, wie diese großen, platzfressenden Schaltflächen, die ... ach nein, jetzt schweife ich doch zu weit ab ...

So näherte ich mich der neuen Benutzeroberfläche von Visual Basic 5.0 mit einiger Skepsis und mit einigen Befürchtungen. Doch diese stellten sich als vollkommen unberechtigt heraus. Nach ein paar Stunden der Beschäftigung mit der VB5-Beta graute mir davor, jemals wieder Visual Basic 4.0 anzurühren. Und das beste ist, daß all diese Errungenschaften mit Visual Basic 6.0 nicht wieder den Bach heruntergegangen sind. Ich werde nun nicht alle Features der Entwicklungsumgebung auflisten – Microsoft hat diese recht gut dokumentiert. Aber hier sind ein paar meiner Favoriten:

- Ich liebe die Auflistungsoptionen im Editorfenster. Diese stellen einem alle Methoden und Eigenschaften eines Objekts in einer Pop-up-Listbox zur Verfügung, wenn man den Namen des Objekts, gefolgt von einem Punkt, eintippt. Die QuickInfos zeigen einem die Syntax einer Methode einschließlich der Parameter und Datentypen. Ich mußte mich erst ein wenig daran gewöhnen, aber mittlerweile ersparen sie mir ungemein viel Zeit und helfen dabei, Fehler zu vermeiden.
- Die Möglichkeit, mehrere Projekte gleichzeitig in eine Instanz der Entwicklungsumgebung zu laden, erleichtert das Debuggen von DLL-Code-Komponenten erheblich.
- Ich mag die MDI-Entwicklungsumgebung. Ich verstehe, daß diese die größte Umstellung erfordert, aber ich rate Ihnen, es doch einmal damit zu versuchen.

- Begeistert bin ich auch von den Tooltips, die während des Debuggens den Wert einer Variablen anzeigen, wenn man im Editorfenster mit dem Mauszeiger darüberfährt.

9.1.1 Kein Formular, sondern ein Designer

Erinnern wir uns kurz daran, daß ein Projekt in Visual Basic 4.0 aus vier verschiedenen Typen von Elementen bestehen konnte: aus Formularen, aus MDI-Formularen, aus Standard-Modulen und aus Klassen-Modulen. Ich möchte Sie darum bitten, diese Elemente mit jeweils anderen Augen zu sehen.

Betrachten wir zuerst die Formulare und MDI-Formulare. Ein Formular verfügt über Eigenschaften, die sein Aussehen und Verhalten festlegen. Dazu gehört Code, worin die Methoden, Ereignisse und weitere Eigenschaften je nach Bedarf beschrieben sind. Somit kann man ein Formular als einen Satz von Eigenschaften plus einem Code-Modul ansehen. In Visual Basic können Sie die Eigenschaften eines Formulars über eine durchdachte Benutzeroberfläche setzen. Sie können Controls in das Formular einzeichnen, Menüs hinzufügen und mehr. In einem separaten Code-Fenster bearbeiten Sie das Code-Modul des Formulars.

Klassen-Module und Standard-Module haben in Sachen vordefinierter Eigenschaften nur wenig zu bieten. Lassen Sie mich das anhand eines Beispiels verdeutlichen: Ein Klassen-Modul kann dazu verwendet werden, ein Objekt zu definieren, das über Eigenschaften verfügt. Doch ein (privates) Klassen-Modul bietet zunächst selbst nur zwei Eigenschaften an, `Name` und `Instancing`. Ein Klassen-Modul wird direkt in einem Code-Fenster bearbeitet. Abbildung 9.1 zeigt den Unterschied zwischen dem Ansatz zur Gestaltung eines Benutzeroberflächen-Objekts wie einem Formular, und dem zur Bearbeitung eines Code-Moduls.

Wenn ein Code-Fenster zum Bearbeiten von Code (egal, ob es der Code eines Klassen-Moduls, eines Standard-Moduls oder eines Formulars ist) verwendet wird, wie würden Sie dann ein Fenster nennen, daß zur Gestaltung, also zum Design, eines Formulars verwendet wird? Tja, seit Visual Basic 5.0 heißt so etwas nun *Designer.*

In einem Formular-Designer können Sie die Eigenschaften eines Formulars über das Eigenschaften-Fenster bearbeiten, Controls aus der Werkzeugsammlung einfügen und Menüs mit dem Visual Basic-Menüeditor bearbeiten. Mit einem MDI-Formular-Designer verhält es sich genauso, nur daß die Eigenschaften ein wenig anders aussehen und es Einschränkungen bei den Controls gibt, die Sie auf dem Formular plazieren können.

Visual Basic 6.0 bietet eine Reihe weiterer eingebauter Designer. Der ActiveX-Control-Designer wird zur Gestaltung von ActiveX-Controls verwendet. Mit dem ActiveX-Dokument-Designer erstellen Sie ActiveX-Dokumente. Mit dem IIS-Anwendungs-Designer erstellen Sie Webklassen, eine spezielle Art von Klassen, die mit Active Server Pages für Internet-Anwendungen auf Microsofts Internet Information Server arbeiten.

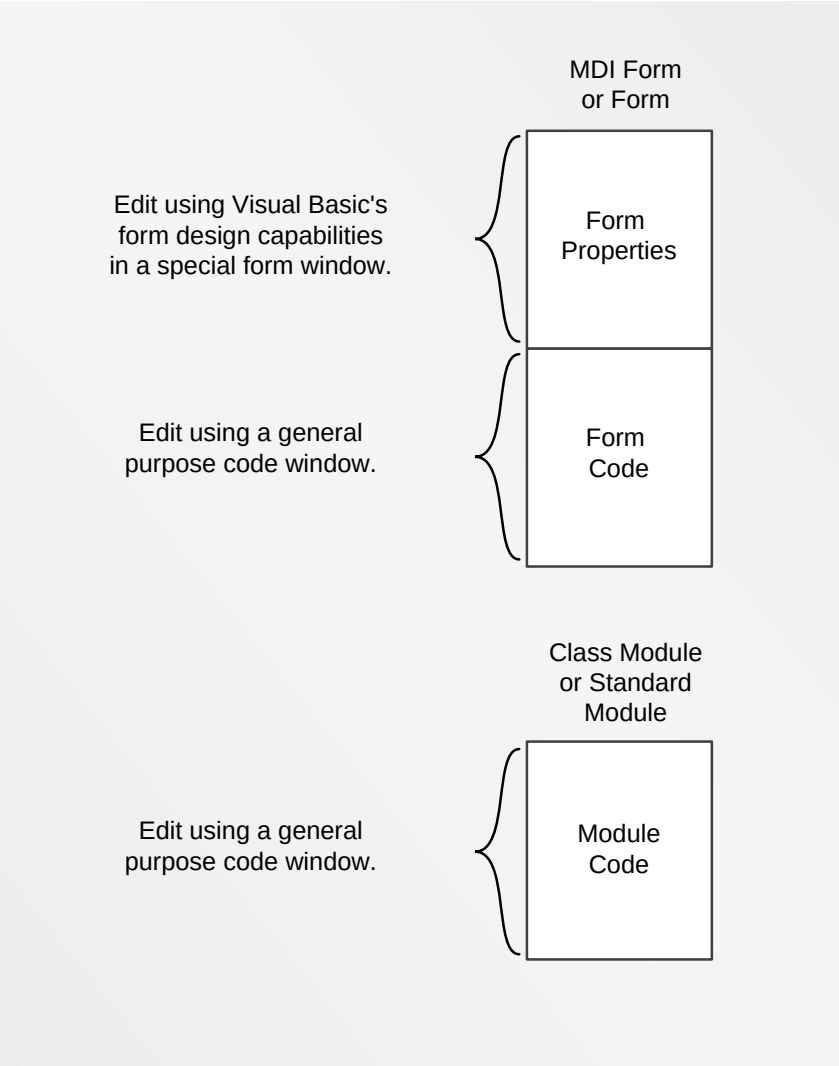

Abb. 9.1: Arbeiten mit Formularen und Modulen

Mit dem DHTML-Designer erstellen Sie dynamische Server-Anwendungen. Mit wiederum anderen Designern erstellen Sie Add-Ins oder Datenbankprojekte. Sie können sogar neue Designer erstellen, allerdings nur in C++.

Der Unterschied zwischen einem Designer-Fenster und einem Code-Fenster ist seit Visual Basic 5.0 von Bedeutung, weil sich seitdem das Verhalten eines Projekts innerhalb der Visual-Basic-Umgebung ändern kann, je nachdem, ob ein Designer geöffnet ist oder nicht.

9.2 Komponenten erstellen und testen

Manchmal fragt mich jemand, woher ich denn die Dinge wissen würde, über die ich schreibe, vor allem, wenn ich zu den ersten gehöre, die über ein Thema wie dieses schreiben, und daher kaum auf Informationen anderer zurückgreifen kann. Habe ich etwa über irgendwelche dunklen Kanäle Zugriff auf den Visual-Basic-Source-Code? Leider nein. Ich lerne die Dinge, indem ich die Dokumentation lese (soweit vorhanden), und durch Experimentieren. Dann kombiniere ich die Ergebnisse der Experimente mit meinem allgemeinen vorhandenen Wissen über Windows und COM-Technologie, und entwickle so meine Theorien und Erklärungen zu dem, was dahintersteckt. Meistens treffe ich das Richtige. In diesem Kapitel lade ich Sie ein nachzuvollziehen, mit welchen Experimenten ich herausgefunden habe, wie Visual Basic mit verschiedenen Komponenten-Typen umgeht.

Die Visual-Basic-Dokumentation bietet Schritt-für-Schritt-Anleitungen für Tests sowohl von In-Process- als auch von Out-of-Process-Komponenten. Diese Anleitungen sind ziemlich gut. Wenn Sie diesen folgen, können Sie Ihre ActiveX-Komponenten mit allen Fähigkeiten der Visual-Basic-Entwicklungsumgebung testen.

Wahrscheinlich werden Sie diese Schritt-für-Schritt-Anleitungen in Griffweite behalten, da sie recht langwierig sind. Im Zweifelsfalle pauken Sie einfach die Schritte, die meistens in einer bestimmten Reihenfolge auszuführen sind. Sie können aber auch ein paar Minuten für das Verständnis aufwenden, was Visual Basic wie und warum macht. Dann wird der Testvorgang einer Komponente so offensichtlich, daß er wie von selbst von der Hand geht.

Wollen wir anfangen?

9.2.1 Die ideale Testumgebung

Einer der großen Vorteile von Visual Basic gegenüber anderen Sprachen wie etwa C++ ist der Interpreter der Entwicklungsumgebung. Interpreter eignen sich hervorragend zum Testen und Debuggen. Sie reichen weiter, als Ihnen lediglich die Möglichkeit zu bieten, Haltepunkte zu setzen und Variablen-Daten im Code zu analysieren – das können Debugger für kompilierte Programme auch. Interpreter erlauben die interaktive Ausführung von Anweisungen und oftmals auch die Änderung des Codes während einer Unterbrechung (Break-Modus). Sie können die Programmausführung von jeder beliebigen Stelle aus fortsetzen und dabei den geänderten Code einbeziehen. Zusammen mit der Möglichkeit, Native-Code zu kompilieren, steht Ihnen das Beste beider Welten zur Verfügung: eine interpretierende Umgebung zum Testen und Debuggen und in Native-Code kompilierte ausführbare Dateien.

Zum Testen einer Komponenten-basierten Anwendung wäre es natürlich ideal, nicht nur das Hauptprogramm in der Visual-Basic-Entwicklungsumgebung laufen lassen zu können, sondern auch zugleich alle beteiligten Komponenten.

Gehen wir noch einen Schritt weiter. Bei einer idealen Testumgebung sollte es möglich sein, alle In-Process-Komponenten auch tatsächlich im gleichen Prozeßraum wie die Container-Anwendung laufen lassen zu können. Wie Sie ja wissen, können sich Komponenten unterschiedlich verhalten, abhängig davon, ob sie In-Process oder Out-of-Process laufen.

In Visual Basic 4.0 konnte nur ein Projekt in einer Visual-Basic-Instanz laufen. Aber immerhin konnte man bereits mehrere Instanzen der Visual-Basic-Entwicklungsumgebung starten. Das war ein großer Fortschritt gegenüber Visual Basic 3.0. Per Definition läuft jedoch jede Instanz in einem eigenen Prozeß, wie damit auch die jeweils darin laufende, zu testende Anwendung. Man konnte so zwar bereits eine DLL-Server-Komponente starten, jedoch lief sie immer in einem anderen Prozeß als die sie verwendende Hauptanwendung. Einige Features konnten auf diese Weise nicht getestet werden – erst mußte die Komponente in eine DLL kompiliert werden. API-Operationen, die auf prozeßspezifische Daten angewiesen sind (z.B. Speicher oder Ressourcen), konnten so nur mit Einschränkungen getestet werden.

Ich stelle fest, daß dies verwirrend klingen mag. Bisher habe ich steif und fest behauptet, daß DLL-Server immer im gleichen Prozeß wie die Anwendung laufen. Nun habe ich gesagt, daß es möglich war, sie in einem eigenen Prozeß zu testen. Wie kann das gehen? Ganz einfach: Visual Basic trickst. Wenn eine ActiveX-DLL-Komponente in der Visual-Basic-Entwicklungsumgebung läuft, wird sie vorübergehend in der Windows-Registrierung registriert, so daß weitere Instanzen von Visual Basic oder andere Anwendungen diese verwenden können. Allerdings wurde sie als ActiveX-EXE-Server registriert, als Out-of-Process-Komponente also.

Dieses Verfahren ist jedoch letzthin nicht tolerierbar für Tests bestimmter Komponenten-Typen. Auch wenn man die meisten Features eines DLL-Servers Out-of-Process testen kann, so müssen doch zumindest per Definition ActiveX-Controls unbedingt In-Process laufen. Microsoft mußte es also einrichten, daß in einer einzelnen Visual-Basic-InstanzVisual-Basic-Instanz sowohl ein ActiveX-Control-Projekt als auch ein Standard-Projekt (das das Control verwendet) gleichzeitig laufen können.

Und wenn schon ein ActiveX-Control zusammen mit einer Anwendung laufen kann, dann ist es auch naheliegend, daß auch mehrere ActiveX-Controls zusammen laufen können. Ebenso naheliegend ist dann auch die Annahme, daß DLL-Server mit einer Anwendung zusammen laufen können, da ja ActiveX-Controls auf der gleichen Technik beruhen.

Genau so hat Microsoft es eingerichtet.

Nach wie vor bildet jedes Standard-Projekt, jedes ActiveX-Control und jeder ActiveX-Server ein eigenes Projekt. Doch innerhalb der Visual-Basic-Entwicklungsumgebung können Sie beliebig viele Projekte hinzuladen (Menü DATEI/PROJEKT HINZUFÜGEN). Sie können auch Projekt-Gruppen anlegen. Eine Projekt-

Gruppe ist nichts anderes als eine Liste von Projekten, die zugleich geladen werden sollen. Sie können jedes einzelne Projekt gesondert laden oder speichern – es bleibt unabhängig. Wollen Sie jedoch beispielsweise eine Anwendung mit fünf Komponenten testen, ist es einfacher, sie alle zugleich in der Visual-Basic-Entwicklungsumgebung zu testen und dazu eine Projekt-Gruppe zu definieren, anstelle die Komponenten jedesmal einzeln hinzuladen zu müssen.

Obwohl die Projekte als Gruppe geladen werden, arbeiten Sie stets nur an jeweils einem einzigen Projekt. Aber woher weiß der Projekt-Eigenschaften-Dialog, welches Projekt dies gerade ist?

Alle Projekte einer Projekt-Gruppe werden im Projekt-Fenster aufgelistet. Es ist immer nur ein Element im Projekt-Fenster ausgewählt. Dadurch wird automatisch festgelegt, mit welchem Projekt Sie gerade arbeiten. Wenn Sie ein Objekt im Objekt-Katalog untersuchen, hängt die Darstellung vom gerade aktiven Projekt ab. Wenn beispielsweise ein Komponenten-Projekt gewählt ist, können Sie sowohl öffentliche als auch Friend-Funktionen eines Objekts sehen (über Friend-Funktionen erfahren Sie mehr in Kapitel 10). Ist ein Projekt, das eine Komponente verwendet, gewählt, werden Sie nur die öffentlichen Funktionen der Komponente zu sehen bekommen. Die Friend-Funktionen bleiben richtigerweise verborgen. In Abbildung 9.2 sehen Sie die Projekt-Gruppe `Conflict.vbg`, die drei separate Projekte umfaßt. Das Projekt `CFLTest.vbg` ist eine Standard-EXE. Hinzu kommen in der Gruppe zwei zusätzliche ActiveX-DLL-Projekte.

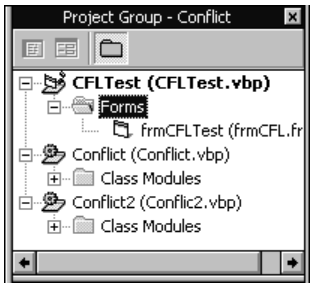

Abb. 9.2: Projekt-Fenster mit drei Projekten

Der Name eines der Projekte im Projekt-Fenster wird fett dargestellt (hier ist es das Projekt `CFLTest.vbp`). Damit wird angezeigt, welches Programm beim Ausführen in der Entwicklungsumgebung gestartet wird. Sie können das Start-Projekt über das Kontextmenü des Projekt-Namens festlegen. Achten Sie darauf, daß immer das Hauptprogramm (in der Regel eine Standard-EXE oder ein ActiveX-EXE-Server) als Start-Projekt markiert ist.

Ein weithin verbreiteter Fehler bei der Entwicklung von ActiveX-Komponenten ist, an der Komponente zu arbeiten und erst dann ein Standard-EXE-Testprojekt hinzuzuladen. Versuchen Sie dann, die Anwendung zu starten, geschieht offen-

sichtlich nichts. Es dauert manchmal eine ganze Weile, bis man dahinterkommt, daß die Komponente anstelle der Anwendung als Start-Projekt gesetzt ist.

Warum braucht die In-Process-Komponente nicht gestartet zu werden? Und wenn wir schon Fragen stellen: Woher weiß Visual Basic, ob die als Projekt in die Entwicklungsumgebung geladene Komponente oder ob eine bereits früher kompilierte DLL verwendet werden soll?

Zur Beantwortung dieser Fragen und zum tieferen Verständnis über Anwendungen, die sowohl In-Process- als auch Out-of-Process-Objekte verwenden, müssen wir uns näher mit der Art und Weise beschäftigen, wie Visual Basic Komponenten lädt.

9.2.2 Eine Komponente wird geboren

In Kapitel 6, »Das Leben und die Lebensdauer einer ActiveX-Komponente«, haben wir recht ausführlich darüber gesprochen, wie eine Komponente im System erscheint. Sie haben gelernt, daß CLSIDs in der Registrierung abgelegt werden, damit eine Komponente identifiziert werden kann. Die IIDs werden in der Registrierung abgelegt, damit Interfaces identifiziert werden können. Ebenso werden Typbibliothek-Kennungen in der Registrierung abgelegt, damit das System weiß, welche Dateien die Typbibliothek-Ressourcen enthalten. Sie haben auch gelernt, daß alle diese Kennungen nichts anderes als 16 Byte lange GUIDs sind.

Verfolgen wir nun den Lebenszyklus einer ActiveX-Komponente vom Erstellen bis zum Testen derselben. Wir beginnen mit einer ActiveX-DLL. Sie können gerne die folgenden Schritt-für-Schritt-Experimente nachvollziehen. Das hier verwendete Beispiel finden Sie auf der Buch-CD im Ordner des Kapitels 9.

Zuerst legen wir ein ActiveX-DLL-Projekt in der Visual-Basic-Entwicklungsumgebung an. Geben Sie dem Projekt einen Namen, damit Sie es wiederfinden können. Fügen Sie eine Eigenschaft in das Klassen-Modul des Projekts ein.

Fügen Sie ein Standard-EXE-Projekt hinzu. Von diesem aus greifen wir zum Testen auf die eben im ActiveX-DLL-Projekt angelegte Eigenschaft zu. Vergewissern Sie sich, daß Ihr Test-Code auf eine Eigenschaft der Komponente zugreift, damit dieses Objekt auch tatsächlich angelegt wird. Sie können zur Kontrolle `Debug.Print`-Anweisungen in die Ereignisse `Class_Initialize` und `Class_Terminate` einfügen.

Frage: Wird die Komponente nun in der Registrierung registriert?

Antwort: Nein. Sie können aber dennoch in Ihrem Standard-EXE-Projekt einen Verweis auf die DLL-Komponente einfügen!

Wie ist das möglich? Es ist möglich, weil Visual Basic nicht auf die Registrierung zuzugreifen braucht, um Informationen über gerade in die Entwicklungsumgebung geladene Projekte zu erhalten. Visual Basic sieht zuerst unter den geladenen Projekten nach.

Fügen Sie also über den Verweis-Dialog einen Verweis auf das Komponenten-Projekt in das Test-Projekt ein. Markieren Sie dann das Standard-EXE-Projekt als Start-Projekt. Starten Sie nun das Projekt. Der Start des Standard-EXE-Projekts führt dazu, daß der Code der DLL erstmals ausgeführt wird.

Frage: Ist nun die Komponente in der Registrierung abgelegt?

Antwort: Ja! Visual Basic braucht eine Komponente nicht zu registrieren, um sie verarbeiten zu können. Doch sobald eine DLL-Komponente in der Visual-Basic-Entwicklungsumgebung läuft, steht sie auch anderen Anwendungen zur Verfügung, einschließlich anderen Instanzen der Visual-Basic-Entwicklungsumgebung. Somit ist es auch möglich, eine Komponente wie seinerzeit in Visual Basic 4.0 zu testen – in zwei separaten Instanzen der Visual-Basic-Entwicklungsumgebung. Die DLL-Komponente erscheint in der Registrierung als In-Process-Server unter dem Namen `VB6DEBUG.DLL`. Diese DLL steht zwischen dem Client-Prozeß und der Visual-Basic-Instanz, in der die Komponente läuft. Denken Sie jedoch daran, daß die Komponente selbst nicht wirklich als In-Process-Server läuft, auch wenn dies der Client-Anwendung vorgegaukelt wird. Somit werden alle API-Funktionen oder anderer Code scheitern, wenn davon ausgegangen wird, daß die DLL im gleichen Prozeßraum läuft. Visual Basic kann eine echte In-Process-Unterstützung nur dann gewährleisten, wenn sowohl das Komponenten-Projekt als auch das Client-Projekt, das auf die Komponente zugreift, in der gleichen Instanz der Visual-Basic-Entwicklungsumgebung laufen.

Starten Sie eine zweite Instanz der Visual-Basic-Entwicklungsumgebung. Legen Sie auch dort ein Standard-EXE-Projekt an und versuchen Sie, einen Verweis auf die Komponente in der anderen Instanz zu setzen. Es funktioniert! Legen Sie in diesem Projekt eine Instanz des Komponenten-Objekts an – es funktioniert ebenfalls.

Frage: Wenn ein DLL-Server als Out-of-Process-Komponente getestet wird – wer stellt dann den Server des Objekts dar? Wo befindet sich die Typbibliothek?

Antwort: Der Server für das Objekt ist Visual Basic selbst. (Schauen Sie selbst in der Registrierung nach – in Kapitel 6 war beschrieben, wie das geht.) Die Typbibliothek wird ebenfalls von Visual Basic entsprechend der Projekt-Datei zur Verfügung gestellt – deren Name wird für das Objekt in der Registrierung im Typbibliothek-Abschnitt abgelegt. Wenn Sie ein Projekt testen, das noch nicht kompiliert worden ist, legt Visual Basic eine temporäre Datei mit einem Dateinamen der Form VB??.TMP an, um die Projekt-Informationen festzuhalten.

Stoppen Sie nun das Projekt in der Visual-Basic-Instanz, die die Komponente enthält (schließen Sie das Test-Projekt).

Frage: Ist die Information weiterhin in der Registrierung abgelegt?

Antwort: Nein. Visual Basic löscht die temporäre Datei und entfernt die Information aus der Registrierung. Wenn Sie nun in der anderen Visual-Basic-Instanz die Komponente anzulegen versuchen, wird das nicht gelingen.

Sichern Sie nun die Projekt-Gruppe, die das DLL- und das EXE-Projekt aus dem ersten Versuch enthält. Sie werden sie weiter hinten in diesem Kapitel wieder benötigen, wenn Sie dem Experiment weiter folgen wollen.

Fassen wir die Ergebnisse dieses ersten Versuchs zusammen. Wenn Sie eine In-Process-Komponente (ActiveX-DLL-Server, ActiveX-Controls) testen wollen, laden Sie deren Projekte in einer Visual-Basic-Instanz hinzu, setzen die Hauptanwendung in dieser Instanz als Start-Projekt und starten die Anwendung. Sie können nun alle von Visual Basic gebotenen Debug-Möglichkeiten für alle in derselben Entwicklungsumgebung laufenden Projekte wahrnehmen. Sie können auch sowohl die Hauptanwendung als auch die Komponenten im Einzelschrittmodus durchlaufen.

9.2.3 Erneuter Versuch mit EXE-Komponenten

Bei EXE-Komponenten stellt sich die Situation in einigen Punkten ähnlich dar, in anderen dagegen radikal anders.

Zuallererst wäre folgendes festzuhalten: ActiveX-EXE-Server-Komponenten laufen in einem eigenen Prozeßraum. Das hat zur Folge, daß jede EXE-Server-Komponente in einer eigenen Visual-Basic-Instanz getestet werden muß.

Beginnen wir nun ein ähnliches Experiment, diesmal mit zwei Visual-Basic-Instanzen. Mein Beispiel mit den Projekten `EXETest2.vbp` und `gtpTest2.vbp` finden Sie im Ordner des Kapitels 9 auf der Buch-CD. Damit Sie jedoch die hier beschriebenen Schritte und Fehlversuche nachvollziehen können, müssen Sie ein Projekt ganz von Anfang an neu anlegen, da einige der Einstellungen meines Test-Projekts bereits die späteren Stadien der Versuchsanordnung widerspiegeln.

In der ersten Instanz erstellen Sie ein ActiveX-EXE-Projekt, geben diesem einen eigenen Namen und fügen eine Eigenschaft in die Klasse ein. Vergewissern Sie sich, daß die `Instancing`-Eigenschaft der Klasse auf *MultiUse* gesetzt ist. In der zweiten Instanz legen Sie ein Standard-EXE-Projekt an, mit dem Sie das Objekt in der ersten Instanz testen können.

Frage: Kann man jetzt schon einen Verweis auf den ActiveX-EXE-Server anlegen?

Antwort: Nein. In der Test-Anwendung kann erst dann ein Verweis auf die Komponente angelegt werden, wenn diese registriert worden ist. Dies unterscheidet sich von der In-Process-Situation, in der Visual Basic bereits alle mit der Test-Anwendung geladenen Komponenten kannte.

Starten Sie nun die Komponente. Jetzt können Sie in der anderen Instanz einen Verweis darauf in Ihre Test-Anwendung aufnehmen. Mit dem Start registriert Visual Basic die Komponente (wie das auch bei DLL-Komponenten geschieht). Wie auch zuvor registriert sich Visual Basic als Server des Objekts.

Starten Sie nun die Test-Anwendung. Sie sollten nun das von der Komponente offengelegte Objekt instanzieren können. Gehen Sie im Einzelschrittmodus durch den Zugriff auf die Eigenschaft der Komponente. Sie werden feststellen, daß automatisch zwischen den beteiligten Visual-Basic-Instanzen umgeschaltet wird, wenn die Bearbeitung des Codes zwischen der Komponente und der Test-Anwendung wechselt.

Sichern Sie nun beide Projekte. Sie werden sie in Kürze wieder benötigen.

Fassen wir wieder zusammen: Wir wissen nun, daß zum Testen einer Out-of-Process-Komponente (ActiveX-EXE-Server, ActiveX-EXE-Dokument) die Komponente wie auch die Test-Anwendung in einer eigenen Visual-Basic-Instanz laufen muß. Dazu müssen Sie sicherstellen, daß ein EXE-Server auch tatsächlich in seiner Visual-Basic-Instanz gestartet wird, damit dessen Objekte bei Bedarf angelegt werden können.

Denken Sie daran, daß Sie durchaus beide Wege kombinieren können! Wenn etwa eine ActiveX-EXE-Komponente eine ActiveX-DLL-Komponente verwendet, kann diese ActiveX-DLL-Komponente in die gleiche Visual-Basic-Instanz wie die EXE-Komponente geladen und mit ihr zusammen getestet werden.

Bis jetzt erscheint die Testerei eine recht einfache Angelegenheit zu sein. Freuen Sie sich aber nicht zu früh – es wird schon noch komplizierter werden. Doch werfen wir nun noch einen schnellen Blick auf die Situation bei ActiveX-Controls.

9.2.4 ActiveX-Controls

Auf ActiveX-Controls gehe ich im dritten Teil, »ActiveX-Controls«, näher ein – daher gestatte ich mir hier nur einen kleinen Vorgriff. ActiveX-Controls werden genauso wie ActiveX-DLL-Server angelegt und getestet, mit folgenden Ausnahmen:

- Verweise auf Controls werden in Visual Basic über den KOMPONENTEN-Befehl des PROJEKT-Menüs hinzugefügt (oder über das Kontextmenü der Werkzeugsammlung).
- Wird ein ActiveX-Control-Projekt in die Visual-Basic-Entwicklungsumgebung geladen, wird es automatisch als Komponenten-Verweis bei allen weiteren in die gleiche Entwicklungsumgebung geladenen Projekten hinzugefügt.
- ActiveX-Controls sind gesperrt, wenn deren Designer geöffnet ist. Daher müssen Sie diesen Designer schließen, bevor Sie das betreffende Control in ein anderes Formular einfügen können.
- Visual Basic 6 legt für ActiveX-Controls keine temporären Registrierungseinträge an, auch wenn sie normalerweise nicht Out-of-Process laufen können. Warum? Weil einige Container, wie etwa der Internet Explorer, Out-of-Process-Controls laden können. Wie das geht? Visual Basic legt das Control über eine In-Process-Komponente namens `VB6DEBUG.DLL` offen, über die der Inter-

net Explorer die Out-of-Process-Komponente wie eine In-Process-Komponente behandeln kann. Visual Basic 6 selbst scheint jedoch diese Fähigkeit nicht zu haben.

Sie werden mehr darüber in Teil III lesen.

IIS-Anwendungen (Webklassen)

IIS-Anwendungen verhalten sich ein wenig anders als andere ActiveX-Komponenten. Als erstes werden Sie darauf stoßen, daß Sie eine IIS-Anwendung nicht testen können, solange Sie das Projekt noch nicht gesichert haben. Das Sichern des Projekts teilt Visual Basic mit, wo die ASP-Datei (Active Server Page) angelegt werden soll, wenn das Programm läuft oder kompiliert wird. Moment mal, Active Server Pages? Was haben die denn mit Webklassen zu tun? Eine ganze Menge: Visual-Basic-IIS-Anwendungen stellen ein Framework dar, das unter den Bedingungen der Active Server Pages des Internet Information Servers läuft. Active Server Pages konnten schon immer Objekte anlegen und Methoden dieser Objekte aufrufen. Webklassen stellen lediglich einen einfacheren Weg für Active Server Pages dar, auf COM-Objekte zuzugreifen, die Sie in Visual Basic im Webclass-Designer anlegen. Diese Objekte werden letztlich als ActiveX-DLLs implementiert, die zusätzlich einige für IIS-Anwendungen notwendige Interfaces unterstützen. Wenn ein Browser eine Active Server Page anfordert, liest der Internet Information Server die Datei und bearbeitet (auf dem Server) das Script. Ein typisches Script sieht etwa so aus:

```
If (VarType(Application("~WC~WebClassManager")) = 0) Then
    Application.Lock
    If (VarType(Application("~WC~WebClassManager")) = 0) Then
        Set Application("~WC~WebClassManager") _
         = Server.CreateObject _
         ("WebClassRuntime.WebClassManager")
    End If
    Application.UnLock
End If

Application _
 ("~WC~WebClassManager").ProcessNoStateWebClass _
 "Project1.WebTest1", Server, Application, Session, _
 Request, Response
```

Das ASP-Script legt keines der Objekte in Ihrer IIS-Anwendung direkt an. Statt dessen wird ein `WebClassRuntime.WebClassManager`-Objekt angelegt. Dieses Objekt, das im Visual Studio enthalten ist und nun wahrscheinlich auch im Internet Information Server, legt Ihr IIS-Anwendungs-Objekt an – hier das Objekt `Project1.WebTest1`. IIS-Objekte werden in der Registrierung wie jeder andere ActiveX-DLL-Server registriert. Welche Dateien stellen nun den Server des Objekts dar? Wenn Sie Ihre IIS-Anwendung in der Visual-Basic-Entwicklungsumgebung testen, tritt auch hier wieder die bereits erwähnte `VB6DEBUG.DLL` in

Erscheinung, die dem Internet Explorer eine In-Process-Komponente als Out-of-Process vorgaukelt. Wenn auch IIS-Anwendungen ein wenig anders als ActiveX-DLL- und ActiveX-EXE-Server aussehen, so ist die zugrundeliegende Technologie dieselbe. Der Webclass-Designer ist eine spezielle Benutzeroberfläche zur Erstellung von COM-Objekten. Sie können in Teil V, »Dies und das ...«, mehr über IIS-Anwendungen lesen.

9.2.5 Die kompilierte Komponente

Beim ersten Anlegen und Testen einer Komponente wird alles problemlos klappen, wie in den zuvor beschriebenen Experimenten. Nun werden wir es ein wenig komplizierter machen – versuchen Sie folgendes:

- Schließen Sie zunächst alle Visual-Basic-Instanzen, damit Sie »rückstandsfrei« neu beginnen können.
- Öffnen Sie die DLL-Test-Projekt-Gruppe, die Sie vorhin angelegt haben (oder öffnen Sie die Projekt-Gruppe `Test1.vbg` von der Buch-CD).
- Starten Sie das Test-Projekt. Alles wird wie zuvor funktionieren.
- Schließen Sie das DLL-Projekt.
- Öffnen Sie die ActiveX-EXE-Komponente in einer Visual-Basic-Instanz und starten Sie diese.
- Öffnen Sie das Test-Projekt für die EXE-Komponente in einer zweiten Visual-Basic-Instanz.
- Versuchen Sie, das Test-Projekt zu starten – es wird fehlschlagen.

Abhängig davon, wie Sie auf die Komponente zugegriffen haben, werden Sie einen Kompiler-Fehler oder einen Fehler beim Zugriff auf die Eigenschaft der Komponente erhalten. Der Verweis-Dialog öffnet sich und zeigt an, daß die Komponente fehlt.

Löschen Sie den das Fehlen anzeigenden Eintrag in der Verweis-Liste und schließen Sie den Dialog. Öffnen Sie den Verweis-Dialog erneut und suchen Sie den Eintrag für die Komponente. Markieren Sie diesen, damit der Verweis erneut aufgenommen wird. Starten Sie nun das Test-Projekt. Es wird wieder anstandslos funktionieren. Was ist da passiert?

Visual Basic und GUIDs

Suchen Sie im Registrierungs-Editor die ProgID Ihrer Komponente im Schlüssel `HKEY_CLASSES_ROOT` (siehe Kapitel 6). Notieren Sie den Wert des Unterschlüssels `CLSID`.

Laden Sie nun erneut das Komponenten-Projekt. Sie können dazu die Visual-Basic-Instanz komplett schließen, dann wieder starten und das Projekt laden, oder Sie laden es einfach erneut über das Menü (DATEI/PROJEKT ÖFFNEN).

Starten Sie das Komponenten-Projekt und versuchen Sie wieder, das Test-Projekt zu starten. Sie werden die Meldung erhalten, daß die Verbindung zur Typ- oder Objekt-Bibliothek des Remote-Prozesses nicht gefunden werden kann. Bestätigen Sie mit OK, um den Verweis zu entfernen. Das ist effektiv der gleiche Fehler, den Sie vorhin schon erhalten haben.

Schauen Sie sich wieder den `CLSID`-Schlüssel in der Registrierung an (am besten die Registrierung erst aktualisieren). Der Wert der `CLSID` ist ein anderer!

Na, das ist aber seltsam ...!

Sie werden sich von der Besprechung von COM her daran erinnern, daß eine wesentliche Bedingung des COM-Systems ist, daß Objekt-Typen und deren Interfaces eindeutig identifiziert werden können, nicht über ihren Namen, sondern über ihre Klasse oder ihre Interface-Kennung. Ändert sich die Klassen-Kennung, sieht es für Windows so aus, als ob es sich um ein völlig neues Objekt handeln würde. Kein Wunder, daß das Test-Projekt die Komponente nicht wiederfinden konnte – mit der Änderung der Klassen-Kennung existierte das alte Objekt einfach nicht mehr.

Wenn Sie den alten, »fehlenden« Verweis löschen und einen neuen Verweis hinzufügen, teilen Sie damit Ihrem Test-Projekt mit, daß Sie eine völlig andere Komponente zu verwenden beabsichtigen. Diese neue Komponente funktioniert jedoch, da die Namen von Klassen und Eigenschaften dieselben geblieben sind. Doch wenn Visual Basic das Projekt nun sichert oder kompiliert, werden die Kennungen der neuen Komponente verwendet. Wenn Sie die Komponente aus dem System entfernen, indem Sie das Projekt erneut laden, wird Ihr Test-Projekt erneut streiken, da schon wieder die erwartete Komponente nicht gefunden wird. Sie können dieses Problem umgehen, indem Sie Objekte vorläufig dynamisch anlegen (über die `CreateObject`-Anweisung) und vorübergehend nur späte Bindung verwenden. In diesem Fall speichert Visual Basic keine GUID-Informationen mit dem Projekt, sondern erhält diese erst, wenn das Objekt tatsächlich angelegt wird. Es ist aber offensichtlich, daß dies keine glückliche Lösung ist.

Bei ActiveX-DLL-Servern tritt das Problem nicht auf, weil Visual Basic Verweise auf Komponenten in der gleichen Instanz der Entwicklungsumgebung in eigener Regie aktualisieren kann. Wenn Sie eine Komponente hinzuladen, werden die GUIDs in der Test-Anwendung entsprechend aktualisiert.

Soll das bedeuten, daß es beim Testen von DLLs niemals derartige Probleme gibt? Nein, nicht ganz. Auch bei DLL-Servern werden die GUIDs in der Registrierung geändert. Wenn Sie die DLL als Out-of-Process-Server testen (wie vorhin beschrieben), werden Sie auf exakt dieselben Probleme wie bei einem EXE-Server stoßen. Allerdings sind diese hier schwerwiegender. Was geschieht, wenn Sie die DLLs und EXEs kompilieren? Wenn die GUIDs jedesmal geändert werden, können Sie eine Komponente eigentlich niemals neu kompilieren, weil ja die Client-Anwendungen die Objekte einer neueren Version nicht wiederfinden könnten.

Es muß also einen Weg geben, Visual Basic anzuweisen, die GUIDs von einer Version zur nächsten beizubehalten. Nun, warten wir's ab ...

Projekt-Kompatibiliät

Laden Sie Ihre EXE-Server-Komponente erneut und kompilieren Sie sie diesmal. Damit erreichen Sie zweierlei. Zum einen wird die Komponente nun in der Registrierung registriert. Die neue ausführbare Datei wird als Server für die Komponente geführt. Zum anderen werden Sie bei einem Blick ins Register KOMPONENTE der Projekt-Eigenschaften feststellen, daß die Option unter VERSIONSKOMPATIBILITÄT auf PROJEKT-KOMPATIBILITÄT gesetzt und der Pfad der ausführbaren Datei eingetragen wurde.

Starten Sie das Komponenten-Projekt. Aktualisieren Sie den Verweis auf diese Komponente in Ihrem Test-Projekt. Dies ist noch einmal notwendig, da Sie das EXE-Projekt erneut geladen und damit einen neuen Objekt-Typ erzeugt hatten. Da nun die Projekt-Kompatibilität gesetzt ist, brauchen Sie künftig den Verweis solange nicht mehr zu aktualisieren, solange Sie nicht die Rückwärtskompatibilität der Komponente selbst durch Änderungen an deren Interfaces aufheben.

Starten Sie nun das Test-Projekt. Es wird funktionieren. Sie können es wie zuvor auch im Einzelschrittmodus testen.

Schauen wir uns das näher an.

In meiner Implementierung dieses Experiments heißt die Komponente `gtpEXETest2.EXEClass2`. Aber wenn dieser Name doch als von der kompilierten ausführbaren Datei implementiert registriert wird, wie kann dann das Objekt zugleich von Visual Basic implementiert werden, so daß eine Anwendung damit getestet werden kann?

Auch hier schummelt Visual Basic wieder. Wenn eine Komponente in einer Instanz von Visual Basic läuft, ändert Visual Basic den Namen des Servers der betreffenden Komponente von der ausführbaren Datei in Visual Basic selbst. An diesem Verhalten hat Microsoft interessanterweise seit den ersten Versionen von Visual Basic 5 herumgebastelt. Zeitweise änderte Visual Basic den Registrierungs-Eintrag der `ProgID`, um auf eine temporäre, von Visual Basic implementierte CLSID zu verweisen. Die Änderung des Registrierungs-Eintrags macht folgendes möglich:

- Programme, die die CLSID zum Zugriff auf das Objekt verwenden, verwenden weiterhin die kompilierte Version.
- Programme, die die `CreateObject`-Funktion und die ProgID verwenden, greifen auf die in der Visual-Basic-Entwicklungsumgebung laufende Version zu.

Der zur Zeit gültige Ansatz der Änderung des Server-Namens in der Registrierung hat zur Folge, daß jeder Client, der auf die Objekte des EXE-Servers zugreifen will, auf das in der Visual-Basic-Entwicklungsumgebung laufende Objekt zugreift.

Warum sich also überhaupt mit der ausführbaren Datei aufhalten? Sie wird doch nie von dem Test-Projekt verwendet, das in einer Instanz der Visual-Basic-Entwicklungsumgebung läuft, richtig?

Richtig, die kompilierte Datei wird vom Test-Projekt tatsächlich nicht benötigt. Sie wird aber von Visual Basic zur Ermittlung der GUID-Werte der Interfaces der Komponente für die Client-Anwendung benötigt.

Wenn Sie ein Projekt das erste Mal kompilieren, wird der Anwendung ein kompletter Satz an GUIDs für den Komponenten-Typ, die Interfaces und die Typbibliotheken zugewiesen. Visual Basic setzt zugleich die Option Projekt-Kompatibilität, die Visual Basic anweist, diese GUIDs bei zukünftigen Kompilaten möglichst wieder zu verwenden. Visual Basic selbst verwendet diese GUIDs, wenn die Komponente im Design-Modus läuft. Das besondere ist hier, daß Visual Basic immer die gleichen Werte verwenden kann. Versuchen Sie einmal folgendes: Stoppen Sie die Instanz, in der der EXE-Server läuft, und laden Sie das Server-Projekt erneut. Starten Sie das Server-Projekt wieder, und ebenso das Test-Projekt. Das Problem des fehlenden Verweises hat sich erledigt. Visual Basic verwendet die Informationen aus der ausführbaren Datei, um sicherzustellen, daß der Server dieselben GUID-Werte erhält, auch wenn er in der Entwicklungsumgebung läuft. Wenn das Server-Projekt nicht gestartet ist, wird automatisch die ausführbare Datei referenziert und verwendet.

Solange Sie mit Test-Projekten (Clients) arbeiten, die in der Visual-Basic-Entwicklungsumgebung laufen, klappt das alles recht gut. Wenn Sie jedoch Ihren EXE-Server mit anderen Client-Programmen oder mit in anderen Programmiersprachen geschriebenen Programmen testen wollen, empfehle ich Ihnen, die Option BINÄR-KOMPATIBILITÄT zu setzen. Dies zwingt Visual Basic dazu, beim Wechsel zwischen der kompilierten und der in einer Visual-Basic-Entwicklungsumgebung laufenden Version nur die GUID-Informationen aus der ausführbaren Datei zu verwenden.

Der Beispiel-EXE-Server `gtpExeDg` kann Ihnen bei der Feststellung helfen, welcher Server die Komponente tatsächlich implementiert. Er verwendet die API-Funktion `GetModuleFileName`, um den Namen der ausführbaren Datei zu ermitteln, die das Objekt implementiert. Kompilieren Sie das Projekt (oder registrieren Sie die bereits kompilierte Version `gtpExeDg.EXE`). Starten Sie dann das Programm `TstExeDg.EXE`. Versuchen Sie, das `gtpExeDg`-Projekt zu starten und wieder zu stoppen – Sie werden sehen, daß sich der Name der Anwendung von `gtpExeDg.EXE` in `VB6.EXE` ändert. Versuchen Sie das erneut, aber ändern Sie vorher die Versionskompatibilität von Binär- zu Projekt-Kompatibilität. Es sollte ein Automations-Fehler auftreten (was allerdings ein Bug in Visual Basic ist, der vielleicht schon behoben ist, wenn Sie dieses hier lesen).

Die Fähigkeit von Visual Basic, GUID-Informationen zu bewahren, ist äußerst wichtig. Nur so können Komponenten aktualisiert werden und trotzdem mit älteren Anwendungen, die diese Komponenten verwenden, weiterhin einwandfrei zusammenarbeiten. Allerdings haben Sie bisher nur die Spitze des Eisbergs in

Sachen Kompatibilität zu Gesicht bekommen. Wir werden in Kapitel 25, »Versionen«, noch weiter darauf eingehen. Hier geht es nur darum, Sie für das Anlegen, Testen und Debuggen von Anwendungen zu wappnen.

SingleUse-EXE-Server

Legen Sie zwei Projekte an, so wie Sie das ActiveX-EXE-Server-Beispiel zuvor angelegt haben. Sie finden die Projekte `gtpSingleExe.vbp` (den Komponenten-Server) und `EXETest3.vbp` (das Test-Projekt) im Ordner zu Kapitel 9 auf der Buch-CD. Das Test-Projekt legt jedesmal eine Instanz des Server-Objekts an, wenn Sie auf eine Schaltfläche klicken.

Der einzige Unterschied zwischen diesem und dem vorherigen EXE-Server-Projekt besteht darin, daß die `Instancing`-Eigenschaft der Objekt-Klasse auf `SingleUse` gesetzt ist. Nun probieren Sie folgendes aus:

- Starten Sie den Server.
- Starten Sie das Test-Projekt.
- Klicken Sie auf die Schaltfläche, um ein Objekt anzulegen.
- Klicken Sie erneut auf die Schaltfläche, um ein weiteres Objekt anzulegen.

An dieser Stelle werden Sie eine Fehlermeldung erhalten, die besagt, daß die Visual-Basic-Entwicklungsumgebung nur eine einzige Instanz einer Klasse zur Verfügung stellen kann. Die Tatsache, daß Ihre Test-Anwendung die erste Instanz bereits wieder freigegeben hatte, ändert nichts daran.

Wenn eine Visual-Basic-Instanz einmal eine `SingleUse`-Instanz eines `SingleUse`-Objekts angelegt hat, kann keine weitere angelegt werden, bevor die betreffende Komponente nicht in der Visual-Basic-EntwicklungsumgebungVisual-Basic-Entwicklungsumgebung gestoppt und erneut gestartet worden ist.

9.2.6 Tips und Techniken

Hier folgen nun einige Tips, die Ihnen beim Anlegen und Testen von ActiveX-Komponenten helfen können.

Müll in der Registrierung

Visual Basic registriert Komponenten temporär, wenn Sie in der Visual-Basic-Entwicklungsumgebung gestartet werden, und hebt die Registrierung wieder auf, wenn die Ausführung beendet wird. Wird die Visual-Basic-Entwicklungsumgebung irregulär beendet, verbleiben die temporären Einträge als Müll in der Registrierung.

Microsoft hat ein Programm namens RegClean entwickelt, das Registrierungs-Einträge von nicht mehr existierenden Objekten beseitigen soll. Sie sollten die von Visual Basic angelegten TMP-Dateien vor dem Start von RegClean löschen. Ich möchte Sie allerdings warnen: Ich habe verschiedentlich gehört, daß

RegClean die Registrierung unter Umständen zerstören kann. Ich habe das Programm bisher ohne Probleme verwendet, so daß ich das nicht untermauern kann. Ich kann Ihnen daher nur empfehlen, vor allem unter Windows NT eine Notfall-Diskette anzulegen, bevor Sie RegClean starten, so daß Sie im Falle des Falles Ihr System wieder rekonstruieren können. Für diese Aufgabe dient das NT-Programm `RDISK.EXE`.

Vollständige Kompilierung

Visual Basic bietet innerhalb der Entwicklungsumgebung drei verschiedene Kompilierungs-Modi. Diese Modi wählen Sie über die Optionen BEI BEDARF und IM HINTERGRUND des Kastens KOMPILIEREN im ALLGEMEIN-Register der Visual-Basic-OptionVisual-Basic-Optionen. Die drei Modi lauten:

- VOLLSTÄNDIGE KOMPILIERUNG*:* Dieser Modus wird gewählt, wenn die Option BEI BEDARF nicht gesetzt ist. Alle in der Entwicklungsumgebung geladenen Projekte werden vor jedem Start kompiliert.
- BEI BEDARF*:* Dieser Modus wird gewählt, wenn die Option BEI BEDARF gesetzt, die Option IM HINTERGRUND jedoch nicht gesetzt ist. In diesem Modus wird der Code nach Bedarf kompiliert. Wenn Ihr Programm unter Umständen nie eine Instanz eines Klassen-Objekts anlegt, wird der Code dieser Klasse auch nie kompiliert.
- IM HINTERGRUND*:* Dieser Modus wird gewählt, wenn beide Optionen, BEI BEDARF und IM HINTERGRUND, gesetzt sind. Dieser Modus gleicht dem vorigen, nur daß nun nach dem Kompilieren des Start-Moduls und nach dem Start des Projekts eventuelle Leerlaufzeiten genutzt werden, die übrigen Module zu kompilieren.

Sie können jede dieser drei Modi bei der Arbeit mit Objekten wählen. Es kann jedoch verwirrend sein, wenn ein Projekt läuft und plötzlich ein Objekt einen Kompilierungsfehler meldet (es ist schon schlimm genug, wenn sie wegen echter Bugs Fehler melden). Ich selbst bevorzuge meistens den Modus der vollständigen Kompilierung. So kann ich bereits alle Syntax-Fehler ausmerzen, bevor der eigentliche Test- und Debug-Vorgang beginnt.

Dies kann jedoch bei sehr großen Projekten lästig werden, wenn das Kompilieren sehr lange dauern sollte. Daher sollten Sie die vollständige Kompilierung zumindest bei Ihrem EXE-Server-Projekt wählen. Sie vermeiden damit am ehesten Timeout-Fehler beim Umschalten zwischen den Instanzen. Die vollständige Kompilierung starten Sie mit der Tastenkombination Strg + F5 oder über das Menü AUSFÜHREN/STARTEN, VOLLSTÄNDIGE KOMPILIERUNG.

Startmodus und Debugging-Optionen

Wann wird die Ausführung einer Visual-Basic-AnwendungVisual-Basic-Anwendung beendet? Wenn das letzte Formular geschlossen wird.

Wann wird die Ausführung eines Visual-Basic-ActiveX-EXE-Servers beendet? Wenn dessen letztes Objekt freigegeben und dessen letztes Formular geschlossen wird.

Was geschieht, wenn Sie eine ActiveX-Server-Anwendung direkt starten? Wenn eine `Sub Main`-Prozedur in einem Standard-Modul vorhanden ist, wird diese ausgeführt. Ist die Bearbeitung der Prozedur erledigt und es ist kein Formular geladen worden, wird die Anwendung beendet. Werden keine Formulare geladen und ist keine `Sub Main`-Prozedur vorhanden, wird die Anwendung unmittelbar nach dem Start wieder beendet. Das einzige, was damit erreicht wird, ist die Registrierung in der System-Registrierung.

Was geschieht, wenn Sie eine ActiveX-Server-Anwendung in der Visual-Basic-Entwicklungsumgebung starten? Ziemlich genau dasselbe. Sie haben aber bereits gesehen, daß eine Komponente erst in den Ausführungszustand versetzt werden muß, damit Visual Basic Instanzen der Komponente anderen Anwendungen zur Verfügung stellen kann. Wie verhindern Sie, daß die Ausführung des Servers unmittelbar beendet wird?

Die Antwort liegt im KOMPONENTE-Register der Projekt-Eigenschaften. Wenn Sie unter STARTMODUS die Option auf ACTIVEX-KOMPONENTE setzen, hält Visual Basic die Komponente solange in der Ausführung, bis Sie das Projekt ausdrücklich stoppen, auch wenn keine Formulare geöffnet oder Objekte referenziert sind. Diese Option wirkt sich nur auf das Verhalten einer Komponente innerhalb der Visual-Basic-Entwicklungsumgebung aus.

In der `Sub Main`-Prozedur eines ActiveX-EXE-Servers können Sie die Eigenschaft `App.StartMode` auslesen. Wenn Sie als Wert `vbSModeAutomation` erhalten, ist die Anwendung auf die Objekt-Anfrage eines Clients hin gestartet worden. Erhalten Sie als Wert `vbSModeStandalone`, ist die Anwendung vom Anwender direkt gestartet worden. Denken Sie aber daran, daß die `Sub Main`-Prozedur eines ActiveX-EXE-Servers bei `vbSModeAutomation` erst dann bearbeitet wird, wenn das erste Objekt angelegt wird.

In Visual Basic 6 ist ein neues Register zu den Projekt-Eigenschaften hinzugekommen, das DEBUGGEN-Register. In diesem Register können Sie einige weitere Optionen zum Debuggen setzen. Ist die Option WARTEN, BIS KOMPONENTE ERSTELLT IST gesetzt, verhält sich die Komponente nach wie vor wie in Visual Basic 5. Ist der Startmodus auf ACTIVEX-KOMPONENTE gesetzt, verbleibt die Komponente in der Ausführung und wartet darauf, daß ein Client auf sie zugreift. Bei einigen Komponenten-Typen können Sie die Komponente tatsächlich starten. Da aber Komponenten eigentlich nicht selbständig laufen können, startet Visual Basic einen Clienten, der die Komponente bereits einmal instanziert. Wenn Sie etwa ein ActiveX-Control starten, legt Visual Basic eine Dummy-HTML-Seite an, die eine Referenz auf das Control enthält, und stellt das Control in einem Browser dar. Sie können auch eine ausführbare Datei starten, von der das Control verwendet wird. Sie können auch eine weitere Instanz von Visual Basic starten.

Und Sie können einen Web-Browser starten und ihn auf eine URL setzen oder einen bereits aktiven Web-Browser verwenden.

Auch wenn das DEBUGGEN-Register auf den ersten Blick als umfangreichere Änderung gegenüber früheren Visual-Basic-Versionen erscheinen mag, steckt jedoch keine größere Änderung im Verhalten oder in der Funktionalität dahinter. Es werden lediglich ein paar praktische Zugriffe auf häufige Operationen nach dem Start einer Anwendung offeriert. Wenn Sie beispielsweise Ihr Control in einer Webseite testen wollen, brauchen Sie nicht unbedingt erst das Control starten, in FrontPage oder einem ähnlichen Werkzeug das Control in eine leere Seite einfügen, dann die Seite in einem Browser anzeigen – nunmehr starten Sie einfach das Control und Visual Basic erledigt das alles für Sie. Die neuen Debuggen-Optionen bieten also nichts weiter als praktischen Test-Komfort.

9.3 Verweise und Reihenfolge von Verweisen

Was passiert, wenn Methoden oder Eigenschaften in zwei Objekten in Konflikt geraten? Dies kann der Fall sein, wenn zwei Objekte globale Methoden, Eigenschaften oder enumerierte Konstanten offenlegen.

In solch einem Fall verwendet Visual Basic die Komponente, die in der Reihenfolge der Verweise an vorderer Stelle steht. Das Beispiel `Conflict.vbg` im Ordner auf der Buch-CD zu diesem Kapitel demonstriert dies.

Die Projekte `Conflict` und `Conflict2` enthalten beide eine Eigenschaft mit dem Namen `MyGlobal`, die global zur Verfügung stehen soll (die `Instancing`-Eigenschaft der Klassen ist auf `GlobalMultiUse` gesetzt).

Der Code für diese Eigenschaft lautet im Projekt `Conflict`:

```
Public Property Get MyGlobal() As Variant
    MyGlobal = "Global from Conflict"
End Property
```

Im `Conflict2`-Projekt lautet er:

```
Public Property Get MyGlobal() As Variant
    MyGlobal = "Global from Conflict2"
End Property
```

Das Projekt `CFLTest` enthält folgenden Code zu Prüfung, auf welche der beiden Implementierungen der Eigenschaft tatsächlich zugegriffen wird:

```
Private Sub cmdMyGlobal_Click()
    MsgBox MyGlobal
End Sub
```

Welcher Wert wird angezeigt? Das hängt von der Reihenfolge der Verweise ab, wie sie im Verweise-Dialog angezeigt wird. Erscheint dort `Conflict` vor

Conflict2, wird auf die erstere Komponente zugegriffen. Sie können mit den Prioritäts-Schaltflächen die Positionen der Komponenten-Verweise in der Liste ändern. Sie können allerdings keinen hinzugefügten Verweis vor die Verweise auf die Visual-Basic-Bibliotheken schieben. Diese Möglichkeit ist besonders in bezug auf enumerierte Konstanten von Bedeutung, die immer global sind.

9.4 Fehlerbehandlung

Die Visual-Basic-Dokumentation von Microsoft befaßt sich mit zwei verschiedenen Ansätzen zur Behandlung von Fehlern. Beim API-Stil wird die Rückgabe eines Fehlerstatus-Werts verlangt, der anzeigt, ob die Ausführung einer Funktion erfolgreich war oder ob sie gescheitert ist. Der Basic-Stil beruht auf der in Visual Basic eingebauten Fehlerbehandlung (das sind die notorischen »On Error...«-Anweisungen).

Microsoft empfiehlt darüber hinaus, konsequent bei einer einmal getroffenen Entscheidung für einen der beiden Stile zu bleiben. Funktionen sollten immer Fehlerwerte zurückgeben, oder einen ByRef-Parameter verwenden, um einen Fehlerwert zurückzugeben, oder sie sollten Fehler auslösen, die vom Client bearbeitet werden können.

Dieses Thema ist eine ziemlich frustrierende Angelegenheit. Denn egal welchen Stil ich auch empfehlen werde, ein großer Teil der Leserschaft würde behaupten, daß ich falsch läge. Es gibt tatsächlich keinen grundsätzlich falschen oder richtigen Stil. Anstelle einer Empfehlung wende ich mich daher zunächst einer näheren Betrachtung der Behandlung von ActiveX-Fehlern zu.

Die Projekt-Gruppe ErrTest im Ordner zu Kapitel 9 auf der Buch-CD enthält zwei Projekte, ErrTest.vbp und ErrClient.vbp. Das ErrTest-Projekt enthält einen DLL-Server mit einer einzigen Klasse, in der die Instancing-Eigenschaft auf GlobalMultiUse gesetzt ist (um die Umstände des ausdrücklichen Anlegens von Objekten bloß zu Testzwecken zu umgehen). Diese Klasse enthält drei Funktionen, die eine einfache Division vornehmen. Die erste, DivideBy1, verwendet den API-Stil der Fehler-Prüfung. Da die Funktion selbst einen numerischen Wert zurückgeben soll, muß der Fehler in einem separaten ByRef-Parameter zurückgegeben werden, in errval. Die Funktion sieht so aus:

```
' Dies demonstriert den API-Stil
Public Function DivideBy1(numerator As Long, _
 denominator As Long, errval As Long) As Long
    If denominator = 0 Then
        ' Fehler abfangen
        errval = -1
        Exit Function
    End If
    DivideBy1 = numerator / denominator
End Function
```

Wenn diese Funktion verwendet wird, muß im Client zuerst eine `Long`-Variable zur Übergabe als `errval`-Parameter dimensioniert werden. Dessen Wert ist dann nach dem Aufruf zu prüfen. Sie können `errval` optional machen, damit Clienten, die sich um den Rückgabewert nicht zu kümmern brauchen, auch keinen gesonderten Parameter übergeben brauchen.

Die Funktion `DivideBy2` folgt dem Ansatz der Client-seitigen Fehlerbehandlung. Sie sieht so aus:

```
' Der Client soll den Fehler behandeln
Public Function DivideBy2(numerator As Long, _
 denominator As Long) As Long
    DivideBy2 = numerator / denominator
End Function
```

Die Funktion `DivideBy3` verwendet die OLE-Fehlerbehandlung. Sie enthält eine interne Fehlerbehandlung zur Entdeckung des Divisionsfehlers (man könnte hier auch die Technik der Funktion `DivideBy1` verwenden und den Wert des Nenners zuerst prüfen). Wenn nun ein Fehler entdeckt wird, wird ein Fehler in der Client-Anwendung ausgelöst. Wir werden uns die Operation des Auslösens gleich noch näher ansehen. Dies ist die Funktion `DivideBy3`:

```
' Fehlerbehandlung im Basic-Stil
Public Function DivideBy3(numerator As Long, _
 denominator As Long) As Long
On Error GoTo problem3:
    DivideBy3 = numerator / denominator
    Exit Function
problem3:
    Err.Raise vbObjectError + 1000, "clsErrorMaker", _
     "Error Maker Numeric Error"
End Function
```

Abbildung 9.3 zeigt das Programm `ErrClient`, das zum Testen dieser Komponente dient. Das Array von Options-Schaltflächen entscheidet, welche Funktion aufgerufen wird. Die Schaltflächen führen jede eine einfache Division aus (die keinen Fehler hervorruft) oder eine Division durch Null. Den Code sehen Sie im folgenden Listing:

```
' Guide to the Perplexed: ErrTest - Error testing program
' Copyright (c) 1996 by Desaware Inc. All Rights Reserved
Option Explicit

Dim CurrentOptionIndex As Integer

' Aktuelle Option setzen
Private Sub cmdOp_Click(Index As Integer)
    Dim numerator As Long
```

```
    Dim denominator As Long
    Dim errval&
    Dim result&
    Select Case Index
        Case 0  ' Gültige Werte für korrekte Operation
            numerator = 10
            denominator = 5
        Case 1  ' Fehler per Division durch Null
            numerator = 10
            denominator = 0
    End Select

    Select Case CurrentOptionIndex
        Case 0  ' API-Stil
            result = DivideBy1(numerator, _
             denominator, errval)
            If errval = -1 Then
               MsgBox "Error occurred"
            End If
        Case 1  ' keine Behandlung
            result = DivideBy2(numerator, denominator)
        Case 2  ' Basic-Stil
            result = DivideBy3(numerator, denominator)
    End Select
End Sub

Private Sub optFunction_Click(Index As Integer)
    CurrentOptionIndex = Index
End Sub
```

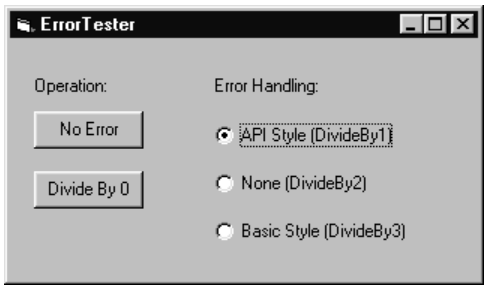

Abb. 9.3: Die Anwendung ErrClient in Aktion

Im ALLGEMEIN-Register der allgemeinen Visual-Basic-Optionen finden Sie den Kasten UNTERBRECHEN BEI FEHLERN. Er enthält drei Optionen. Die Einstellung BEI JEDEM FEHLER veranlaßt Visual Basic, das Programm immer sofort zu unterbrechen, wenn ein Fehler aufgetreten ist, unabhängig von der im Code stehenden

Fehlerbehandlung. Die Option IN KLASSENMODUL veranlaßt Visual Basic, bei unbehandelten Fehlern in einem Klassen-Modul dort zu unterbrechen. Bei der Option BEI NICHT VERARBEITETEN FEHLERN unterbricht Visual Basic im Client an der Stelle, an der der Fehler aufgetreten ist.

Gehen wir diese Varianten der Reihe nach durch.

Der Fall `DivideBy1` ist der einfachste. Es wird nie ein Visual-Basic-Fehler ausgelöst. Es bleibt dem Client überlassen, auf Fehler zu achten.

Der Fall `DivideBy2` ist etwas verzwickter. Ist die Fehlerunterbrechung auf BEI JEDEM FEHLER oder IN KLASSENMODUL eingestellt, wird Visual Basic in der Klasse `clsErrorMaker` unterbrechen, wenn der Divisionsfehler auftritt. Anderenfalls wird in der Zeile des Clients unterbrochen, in der die `DivideBy2`-Methode aufgerufen wird. Dies geschieht, weil der Client automatisch von der Komponente generierte Fehler behandelt, so daß auch ein Fehler in einer Komponente wie vorgesehen im Client behandelt wird.

Im Fall `DivideBy3` ist eine Fehlerbehandlung enthalten, die ihrerseits einen Fehler an die Anwendung weiterreicht. Ist die Fehlerunterbrechung auf BEI JEDEM FEHLER eingestellt, wird Visual Basic auch hier wieder in der Zeile in der Klasse `clsErrorMaker` unterbrechen, in der der Fehler auftritt. Ist jedoch IN KLASSENMODUL eingestellt, wird der Fehler in der folgenden Zeile ausgelöst:

```
Err.Raise vbObjectError + 1000, "clsErrorMaker", _
 "Error Maker Numeric Error"
```

Warum dieses? Die Zeile mit der Division führt nicht zu einem Fehler im Klassen-Modul, da er im Klassen-Modul abgefangen wird. Jedoch die Methode `Err.Raise` löst einen Fehler aus, der nicht im Klassen-Modul behandelt wird und somit zum Client »nach oben« weitergereicht wird. Dies ist der erste »echte« Fehler im Klassen-Modul, der daher entsprechend der Fehlerunterbrechungs-Einstellung eine Unterbrechung verursacht. Ist die Fehlerunterbrechung auf BEI NICHT VERARBEITETEN FEHLERN eingestellt, wird Visual Basic in der Zeile der `Err-Client`-Anwendung unterbrechen, in der die `DivideBy3`-Methode aufgerufen wird.

9.4.1 OLE-Fehlerbehandlung und »Basic«-Fehlerbehandlung

In Kapitel 4, »Das Component Object Model: Interfaces, Automation und Bindung«, hatten wir als fundamentales Merkmal der OLE-Technologie einen Standard-Mechanismus zur Fehlermeldung und einen Satz an Fehler-Codes und -Werten angeführt.

Dies ist notwendig, damit COM-Komponenten in den verschiedensten Programmiersprachen erstellt werden können. Ohne einen standardisierten Mechanismus zur Meldung von Fehlern könnte kein Client mit Fehlern in Komponenten umgehen, ohne über genaue Kenntnisse jeder Komponente zu verfügen. Und schließ-

lich steht doch hinter der ganzen COM-Geschichte die Idee, daß Clients auf allgemeine Weise mit Objekten umgehen können sollen, ohne derartige spezielle Informationen.

Wie OLE tatsächlich Fehler auslöst, ist weniger interessant. Wichtiger ist, daß jeder OLE-Fehlerwert ein 32-Bit-Wert ist. Der Fehler ist in mehrere Abschnitte unterteilt, die Sie in Tabelle 9.1 sehen können.

Bit-Werte (31 = High, 0 = Low)	Bedeutung
31	1 = Fehler, 0 = Erfolg
27 – 30	Reserviert bzw. OLE-intern
16 – 26	Herkunft: zeigt das Subsystem an, das den Fehler generiert hat; z. B.: Windows = 8, ActiveX-Automation = 2
0 – 15	Die tatsächliche Fehlernummer

Tab. 9.1: Aufteilung der OLE-Fehlerwerte

Vielen Fehlern sind Standard-Fehlerwerte zugeordnet. Wenn Sie beispielsweise versuchen sollten, auf eine nichtexistierende Methode über ein Dispatch-Interface zuzugreifen, bekämen Sie wahrscheinlich den Fehlerwert `&H80020003` zurück. Dieser setzt sich aus folgenden Werten zusammen:

- `&H80000000`: Bit 31 ist gesetzt, also Fehler
- `&H00020000`: Bits 16 – 26 sind 2, es wird also ein Automations-Fehler angezeigt
- `&H00000003`: Bits 0 – 15 sind 3, der Fehler heißt also: »Mitglied nicht gefunden«

Sie können auch Ihre eigenen Fehler definieren. Das `clsErrorMaker`-Objekt gibt beispielsweise den Fehlerwert `vbObjectError + 1000` zurück.

Was bedeutet `vbObjectError`? Diese Konstante stellt den Wert `&H80040000` dar. Dieser Wert setzt sich folgendermaßen zusammen:

- `&H80000000`: Bit 31 ist gesetzt, also Fehler
- `&H00040000`: Bits 16 – 26 sind 4, zeigen also an, daß der Fehler Interface-spezifisch ist (jedes Interface definiert seine eigenen Fehlerwerte)

Welche Fehlerwerte können Sie nun also verwenden? Sie teilen sich `vbObjectError` mit Visual Basic selbst, so daß Sie keinen Wert unter 512 verwenden sollten. Sie können aber auch keine Werte über 65536 nehmen, da diese einen Überlauf in das Herkunftsfeld bedeuten würden. Somit ergibt sich ein Wertebereich für Ihre selbstdefinierten Fehler von `vbObjectError + 512` bis `vbObjectError + 65536`.

Nun stellt sich die Frage: Könnte man auch API-Techniken dazu verwenden, einen Fehlerwert in diesem Bereich zurückzugeben, anstelle einen Fehler auszulösen?

Natürlich können Sie das. Wenn Sie nämlich OLE-DLL-Funktionen direkt aufrufen (was durchaus möglich ist), werden Sie häufig auf einen `HRESULT`-Wert als Rückgabewert treffen. Und ein `HRESULT` ist ein OLE-Fehlerwert.

Das Auslösen eines Fehlerereignisses über die `Err.Raise`-Methode bietet einen effizienten Weg zum Auslösen von Fehlern in Clients, die Ihre Komponente verwenden. Sie sollten einen Namen für die Fehlerquelle und gleichfalls eine Beschreibung angeben. Ich verwende hier gerne den Objekt-Namen als Namen der Quelle.

9.4.2 EXE-Server-Komponenten

EXE-Server-Komponenten erfordern besondere Aufmerksamkeit, da sie einen eigenen Satz von Fehlern auslösen können, die damit zu tun haben, daß Sie in einem eigenen Prozeßraum laufen. Das Projekt `EXEErr.vbp` ist ein einfacher Server, der einige Operationen ausführt, die Sie eigentlich niemals in einem Server unterbringen sollten:

```
' GTP EXEErr - EXE Error Server
' Copyright (c) 1997 by Desaware Inc.
' All Rights Reserved

Option Explicit

' Machen Sie dies niemals!
Public Sub KillThisComponent()
    End
End Sub

' Dies sollten Sie auch nicht machen:
Public Sub NeverReturns()
    Do
    Loop While True
End Sub

' Lädt ein Formular, das nach ein paar Sekunden das
' Programm beendet.
Public Sub LoadBadForm()
    Load BadForm
End Sub

' Funktion, die nichts tut
Public Sub SafeFunction()
End Sub
```

Die Methode `KillThisComponent` beendet den Server während des Methoden-Aufrufs. Ich denke, ich kann mir den ausdrücklichen Hinweis sparen, daß Sie das niemals machen sollten. Aber hier stellt das eine perfekte Simulation der Situation dar, die eintritt, wenn beispielsweise ein Speicherfehler oder eine illegale Operation (ehemals General Protection Fault – Allgemeine Schutzverletzung) in einer Komponente auftritt. Nicht, daß so etwas gerade in Ihren Komponenten auftreten würde, aber ...

Die Methode `NeverReturns` simuliert eine Komponente, die hängt, egal, ob wegen einer Endlosschleife oder einer überlangen Operation. Dieses Problem ist nicht gerade unbekannt.

Die Methode `LoadBadForm` lädt ein unsichtbares Formular, das ein Timer-Control enthält mit einem 2-Sekunden-Timeout. Wird der Timeout erreicht, wird die `End`-Anweisung ausgeführt, die den Server terminiert. Wenn Sie den Server in der Visual-Basic-Entwicklungsumgebung laufen lassen, kann dies zur Demonstration dienen, was bei einem Versuch eines Zugriffs auf ein nicht anlegbares Objekt passiert.

Die Funktion `SafeFunction` stellt eine fehlerfrei aufrufbare Funktion zum Testen dar.

Das Projekt `ErrTest2.vbp` des folgenden Listings zeigt den Code für einige Tests, die Sie mit dem Server durchführen können, um diese Fehler zu illustrieren und zu bearbeiten.

Das Formular dieses Projekts enthält fünf Schaltflächen mit den folgenden einfachen Click-Ereignis-Prozeduren:

```
' gtp - Tests gtpEXEError
' Copyright (c) 1997 by Desaware Inc. All Rights Reserved
Option Explicit

Private Sub cmdBadError2_Click()
    Dim obj As New BadClass
On Error GoTo BadError2
    obj.KillThisComponent
    Exit Sub
BadError2:
    MsgBox "The error was caught"
End Sub

Private Sub cmdBadError_Click()
    Dim obj As New BadClass
    obj.KillThisComponent
End Sub

Private Sub cmdEnd_Click()
    Dim obj As New BadClass
```

```
    obj.LoadBadForm
End Sub

Private Sub cmdNever_Click()
    Dim obj As New BadClass
    obj.NeverReturns
End Sub

Private Sub cmdSafe_Click()
    Dim obj As New BadClass
On Error GoTo safeerror
    obj.SafeFunction
    Exit Sub
safeerror:
    MsgBox "Couldn't call the safe method"
End Sub
```

Die Ereignisse `cmdBadError_Click` und `cmdBadError2_Click` demonstrieren, was passiert, wenn eine Server-Operation fehlschlägt. Glücklicherweise kommt dies nicht allzu häufig vor. Müssen Sie darauf achten, überall eine Fehlerbehandlung einzubauen, wo auf einen EXE-Server zugegriffen wird? Das hängt von Ihrer Anwendung ab. In vielen Fällen kann es ausreichen, wenn Ihre Anwendung in solch einem seltenen Fall terminiert. Sie sollten aber auch bedenken, daß jede Fehlerbehandlung auch von diesem Fehlertyp aktiviert wird – beachten Sie dies im Code Ihrer Fehlerbehandlung.

Die Prozedur `cmdNever_Click` befördert den Server in eine Endlosschleife. Der Trick zum Testen dieser Situation besteht darin, eine zweite Instanz der `ErrTest2`-Anwendung zu starten und dort auf die Schaltfläche `cmdSafe` zu klikken. Da dieser EXE-Server in einem Single-Thread läuft, verhindert die von der ersten Instanz gestartete Endlosschleife den Zugriff auf den Server durch die zweite Instanz, und es wird ein »Server Busy«-Fehler ausgelöst (»Server beschäftigt«).

In der Prozedur `cmdEnd_Click` wird der Server dazu veranlaßt, nach ein paar Sekunden zu terminieren. Versuchen Sie einmal diese Anweisung und danach die Operation aus `cmdSafe_Click`. Dies verdeutlicht, was geschieht, wenn der Server kein Objekt des gewünschten Typs zur Verfügung stellen kann.

9.4.3 Hinweise und Vor- bzw. Nachteile

Hier folgen nun einige Hinweise, die Sie hoffentlich nützlich finden.

Die wichtigste Technik zur Vermeidung von Fehlern ist, zu versuchen, diese bereits beim Entwurf zu vermeiden. Die Behandlung von Fehlern, die von ungültigen Client-Operationen oder -Parametern ausgelöst werden, ist eine Sache. Aber Sie möchten bestimmt nicht, daß Fehler ausgelöst werden, nur weil in Ihrem Code Bugs stecken.

Wenn Ihre Komponente selbst wiederum andere Komponenten verwendet, die Fehler auslösen können, sollten Sie eine Fehlerbehandlung in Ihrer Komponente einführen und Ihre eigenen Fehler an den Client Ihrer Komponente melden. Die Anwender Ihrer Komponente wollen die von Ihnen definierten Methoden und Eigenschaften nutzen. Das letzte, was sie wollen, ist, sich mit Fehlern herumzuschlagen, die von Subkomponenten ausgelöst werden. Sie sollten daher die Fehler dokumentieren, die Ihre Komponente auslösen kann.

Anstelle des Auslösens von Fehlern in jeder Funktion, in der ein Fehler auftreten kann, sollten Sie über die Einführung einer zentralisierten Fehlerbehandlung in der Klasse nachdenken. Damit würde sich die Angabe der Quelle in jeder einzelnen Funktion erübrigen. Sie können so auch Beschreibungstexte zentralisieren, Ich stecke z.B. sehr oft alle Fehlertexte in ein separates Standard-Modul.

Verwenden Sie Fehlernummern, die den Kontext-IDs in Ihrer Hilfe-Datei entsprechen oder einen festgelegten Offset zu diesen haben. Wenn etwa Ihre Kontext-IDs von 1000 bis 2000 reichen, könnten Sie Fehlernummern von 600 bis 1600 verwenden und bräuchten lediglich 400 zu addieren, um Übereinstimmung zu erzielen. Das erleichtert die Aufnahme der Kontext-ID in die `Err.Raise`-Anweisung.

Die Fehlerbehandlung im API-Stil hat den Vorteil, daß Sie häufig leichter zu lesen und zu debuggen ist. Anders als beim Basic-Stil gibt es keine plötzlichen Sprünge im Programmfluß (einer der Gründe, weswegen die `GoTo`-Anweisung bei Programmierern so verhaßt ist). Ich bevorzuge meistens den API-Stil, wenn ich meine eigenen Anwendungen oder ActiveX-Komponenten erstelle. Den Basic-Stil verwende ich eher bei ActiveX-Controls, da Visual-Basic-Programmierer daran gewöhnt sind, daß Controls Fehler auslösen. Hoffentlich wird Visual Basic eines Tages eine strukturierte Fehlerbehandlung bieten – eine mehr block-strukturierte Form der Fehlerauslösung, die nicht die radikalen Wechsel im Programmfluß hervorruft wie die `On Error GoTo`-Anweisung.

Im Zweifelsfalle können Sie auch beide Stile einführen. Implementieren Sie einfach eine Eigenschaft, über die der Anwender zwischen den beiden Stilen wählen kann. Dann lösen Sie nur dann Fehler aus, wenn der Basic-Stil eingestellt ist.

Nun wissen Sie, wie Komponenten in Ihrem System arbeiten. Damit wird es Zeit, sich näher mit Visual-Basic-Klassen-Modulen zu befassen. Denn diese – und deren Verwandte, wie Formular, ActiveX-Control und ActiveX-Dokument – bilden die Grundlage jedes mit Visual Basic erstellten Objekts.