Kapitel 22

Komplexe Techniken

Sie haben es vielleicht schon bemerkt, daß ich jedes Kapitel gerne mit einem einführenden Abschnitt beginne, wo ich Ihnen kurz erzähle, was Sie lesen werden, und warum es so wichtig ist. Das wollte ich auch für dieses Kapitel machen, aber nach 1500 Wörtern erkannte ich, daß ich begann, das eigentliche Kapitel zu schreiben.

Statt der üblichen Einleitung möchte ich also ein paar Wörter zur Vorsicht sagen. Dieses Kapitel beschreibt einige der komplexesten Funktionen, die Windows überhaupt unterstützt. Viele der Techniken, die ich hier vorstellen werde, sind schwierig zu verstehen und zu debuggen, und wenn sie fehlschlagen, werden sie Ihre Applikation oder sogar Ihr System zum Absturz bringen oder eine Speicherausnahme verursachen. Ich werde versuchen, die grundlegenden Konzepte zu den jeweiligen Techniken kurz zu beschreiben, aber Sie werden sie sehr wahrscheinlich nicht wirklich verstehen, wenn Sie sich nicht genau mit Windows und dem Win32 API auskennen. Wenn Sie irgendwo aussteigen, dann kann ich Ihnen nur empfehlen, Dan Appleman's Visual Basic 5.0 Programmer's Guide to the Win32 API zu lesen, auf den im gesamten Buch immer wieder verwiesen wird. Sein einziges Ziel ist, Visual-Basic-Programmierern ein gutes Verständnis für Windows und das Win32 API zu vermitteln. Weil es mehr als 1500 Seiten umfaßt, werden Sie sicher verstehen, warum ich nicht versuche, dieselben Informationen in diesem Kapitel unterzubringen.

22.1 Visual Basic versus Visual C++

Nachdem wir nun gesehen haben, wie man ActiveX-Controls gewisser Länge mit Visual Basic erzeugt, erlauben Sie mir zwei einfache Fragen: Ist es mit Visual Basic einfach, ActiveX-Controls zu erstellen? Ermöglicht Visual Basic es, ernsthafte, robuste, professionelle ActiveX-Controls zu erzeugen? Die Antwort auf beide Fragen ist ein klares Ja – und Nein. Um zu erklären, was ich damit meine, sollte ich zunächst eine völlig andere Sprache erwähnen: Visual C++.

Wenn Sie Visual Basic und Visual C++ besitzen, wissen Sie, daß diese völlig unterschiedlich sind, und das nicht nur wegen der zugrundeliegenden Sprache. Visual C++ ist also nicht wirklich »visuell«, nicht im selben Sinne wie Visual Basic, aber es kommt ihm mit jeder Version näher.

Visual Basic ist eine höchst interaktive Entwicklungsumgebung, wo Sie einfach mit den Elementen der Benutzeroberfläche und anderen Objekten interagieren und ihren Ereignissen direkt Code zuweisen. Die Viusal-Baisc-Umgebung kapselt die zugrundeliegende Windows- und OLE-Technologie ein, so daß Sie nichts damit zu tun haben. Diese Kapselung macht Visual Basic relativ einfach zu erlernen und zu verwenden, führt aber auch zu einem gewissen Funktionalitätsverlust. Sie können nur die Funktionen nutzen, die Visual Basic bereitstellt, oder auf die Visual Basic über API-Aufrufe oder Drittherstellerwerkzeuge Zugriff hat.

Visual C++ ist eine Kombination aus Compiler, Klassenbibliotheken (die eine davon wird als Microsoft Foundation Classes, MFC bezeichnet, die andere als

Active Template Library, ATL), sowie mehreren komplexen Assistenten. Die Klassenbibliothek bietet eine Umgebung für die Arbeit mit Registern und COM, die mit Hilfe der Assistenten programmiert wird. Die Assistenten wissen, wie Code zu erzeugen ist, um in dieser Umgebung die vielfältigsten Aufgaben zu erfüllen. Die Umgebung selbst bietet relativ großen Zugriff auf die zugrundeliegende Funktionalität von Windows und OLE.

Der Haken bei diesem Ansatz ist folgender: Solange Sie versuchen, Funktionalität zu implementieren, die die Assistenten beherrschen, ist Visual C++ einfach zu erlernen und zu verwenden. Sobald Sie jedoch versuchen, über diese eingebaute Funktionalität hinauszugehen, macht Visual C++ fundierte Kenntnisse über Windows und seine Erweiterungsbibliotheken erforderlich, um etwas zu bewerkstelligen. Und während die Assistenten und Klassenbibliotheken einen Großteil der müßigen Arbeit erledigen, wenn es um allgemeine Windows-Aufgaben geht, dann sind sie nur wenig bis gar nicht hilfreich, wenn Funktionen implementiert werden sollen, die speziell auf Ihre Applikation zugeschnitten sind.

Bei Visual Basic müssen Sie einfach nur lernen, einfache Steuerelemente in Visual Basic selbst zu erstellen. Bei Visual C++ und MFC müssen Sie C++, die Windows-API-Funktionen, die MFC-Klassen und OLE beherrschen. ATL ist eine der am schlechtesten dokumentierten Bibliotheken, die es jemals gab. Und letztlich ist Visual C++ viel schwieriger zu erlernen als Visual Basic (und das aus der Perspektive eines Betrachters, der beides häufig einsetzt).

Und noch etwas sollten Sie zu Visual C++ wissen: Wenn Sie die MFC-Klassen und deren Assistenten verwenden, müssen Sie zusammen mit Ihrem Steuerelement die MFC-Laufzeitbibliotheken weitergeben. Trotz der Tatsache, daß Sie C++ verwenden, müssen Sie also immer noch eine Laufzeitbibliothek bereitstellen – genau dieselbe Situation, der Sie in Visual Basic gegenüberstehen. ATL-Steuerelemente machen keine Bereitstellung von Laufzeitbibliotheken erforderlich, aber sie sind viel komplizierter zu entwickeln und bedingen ein ausgezeichnetes Verständnis sowohl der Windows- als auch der OLE-Technologie. Eine detailliertere Diskussion der Abwägungen bei der Auswahl einer Programmiersprache zur Entwicklung von Steuerelementen finden Sie in meinem Artikel »The ActiveX Control Choice – VB, MFC or ATL«, den Sie auf der Web-Site von Desaware unter der URL `www.desaware.com` finden.

22.1.1 »Einschränkungen« von Visual Basic

Ich habe schon viele Leute klagen gehört, die sich über die Einschränkungen von Visual Basic bei der Entwicklung von ActiveX-Steuerelementen beschwert haben. Diese Einschränkungen resultieren aus zwei Tatsachen:

Erstens gibt es einige deutliche Einschränkungen darin, wie Visual Basic die ActiveX-Steuerelemente implementiert. Sie resultieren aus der Kapselung, die es gerade so einfach macht, Steuerelemente in VB zu erzeugen. Die meisten davon

können auch mit Hilfe von API- und OLE-Techniken oder Drittherstellerwerkzeugen wie SpyWorks entwickelt werden.

Zweitens gibt es das Problem der Erwartungshaltung. Als Visual Basic vorgestellt wurde, hielten einige Leute es für eine Spielzeugsprache, weil sie nicht alles konnte, was sie von ihr verlangten. In der Kernsprache fehlten Funktionen, die nicht ohne direkten Zugriff auf das Windows-API oder unter Verwendung von Drittherstellerwerkzeugen implementiert werden konnten.

Heute verstehen wir, daß es eben die Kapselung in VB fordert, daß ein Teil der Funktionalität aus der Kernsprache ausgeklammert wurde. Hätte Visual Basic alles gekonnt, dann wäre es einfach nur ein weiteres Visual C++ gewesen, und ebenso schwierig zu erlernen und zu benutzen. VB-Programmierer haben sich daran gewöhnt, die Funktionen zu nutzen, die direkt in die Sprache eingebaut sind, während sie API-Aufrufe oder Werkzeuge von Drittherstellern einsetzen, um Visual Basic nach Bedarf zu erweitern, wenn das für ihre speziellen Applikationen erforderlich ist.

Das ist vielleicht die größte Stärke von Visual Basic, daß es Ihnen die Möglichkeit bietet, selbst auszuwählen, welche Aufgaben auf oberster Ebene (VB) und welche auf unterster Ebene (API) ausgeführt werden sollen. Ich glaube, daß viele dieser Programmierer, die sich über die Implementierung der ActiveX-Steuerelemente in Visual Basic beschweren, diese Abwägung einfach vergessen haben. Sie haben sich darauf eingefahren, daß Visual Basic, nur weil es nicht alles erlaubt so wie ein ActiveX-Steuerelement unter Visual C++, eben nicht für die ernsthafte Entwicklung von Steuerelementen geeignet ist. Das ist heute ebenso unsinnig wie damals.

Sie sollten außerdem beachten, daß die Implementierung der ActiveX-Steuerelemente in Visual Basic einige einzigartige Vorteile gegenüber dem Ansatz von Visual C++ hat. Es ist außerordentlich schwierig, mit VC++ Steuerelemente aus bereits existierenden Steuerelementen zusammenzusetzen. Die große Vielzahl der VC++-Steuerelemente sind entweder rein vom Benutzer definierte Steuerelemente, oder sie stellen Unterklassen eines Standard-Steuerelements dar; die letztgenannte Technik, das Subclassing, werden Sie später in diesem Kapitel kennenlernen.

Kommen wir zurück zu den Fragen vom Kapitelanfang. Macht es Visual Basic einfach, ActiveX-Steuerelemente zu erzeugen? Das Schlüsselwort ist »einfach«. Die Antwort ist ja, solange das Steuerelement nur die Funktionalität unterstützen soll, die Visual Basic selbst beinhaltet. Die Antwort ist nein, sobald Sie eine Funktionalität implementieren müssen, die über die Dinge hinausgeht, die Visual Basic einfach für Sie macht.

Ist es möglich, mit Visual Basic ernsthafte, robuste, professionelle ActiveX-Steuerelemente zu entwickeln? Das Schlüsselwort ist »möglich«. Die Antwort ist noch einmal ja, so lange Sie gewillt sind, die Mühe auf sich zu nehmen, alle zur Verfügung stehenden Ressourcen zu nutzen. Wenn Sie die von Visual Basic

bereitgestellte Funktionalität kennen, Win32-API-Programmiertechniken beherrschen, die Vorteile komplexer API-Techniken wie Unterklassenbildung und Hooks nutzen können, die Natur von COM-Schnittstellen verstehen und Werkzeuge von Drittherstellern einsetzen wollen, die diese Schnittstellen manipulieren, dann ist alles nur Vorstellbare möglich. Wenn Sie sich jedoch nicht mit diesen zusätzlichen Ressourcen auseinandersetzen und sie nicht nutzen wollen, lautet die Antwort nein. In so einem Fall müssen Sie Visual C++ einsetzen, wo Sie letztlich gezwungen sind, diese Techniken zu nutzen, ob Sie wollen oder nicht!

Diese Situation führt dazu, daß ein komplexes Steuerelement in Visual Basic genau so kompliziert zu schreiben sein kann wie in Visual C++. Ich glaube aber, daß Visual Basic selbst in diesen Situationen einen Vorteil bietet, und sei es nur aufgrund der interaktiveren Entwicklungsumgebung für das Testen und Debuggen.

22.2 Überblick über das Windows-Messaging

Die Entwicklung komplexer Steuerelemente fordert ein fundiertes Verständnis für die Grundlagen des Windows-Messagings, also die Nachrichtenverarbeitung von Windows. Wenn Sie bereits mit dem Windows-Messaging vertraut sind, können Sie diesen Abschnitt überblättern. Für alle anderen sollte er genügend Hintergrundwissen bereitstellen, um den später in diesem Kapitel beschriebenen Techniken folgen zu können.

22.2.1 Was ist eine Nachricht?

Überlegen wir, was passiert, wenn wir mit der Maus auf ein Fenster klicken. Der Maustreiber für das Windows-Betriebssystem erkennt den Mausklick und benachrichtigt das Betriebssystem darüber, daß ein Klick aufgetreten ist. Das Betriebssystem stellt fest, welches Fenster sich unter der Mauszeigerposition befindet, und ermittelt, für welche Applikation die Information vorgesehen ist.

Das Betriebssystem verwaltet für jede Applikation eine Warteschlange, in der eine Liste aller Ereignisse enthalten ist, beispielsweise Klicks und Tastencodes, die an die Applikation geschickt werden sollen. Jedes dieser Ereignisse wird als Nachricht (Message) bezeichnet, die Warteschlange heißt als Nachrichtenwarteschlange. Jeder Nachricht sind bestimmte Informationen zugeordnet, wie in Tabelle 22.1 beschrieben.

Nachrichtenparameter	Beschreibung	Mausklick-Beispiel
HWnd	32-Bit-Handle des Fensters, das Ziel für die Nachricht ist.	Der Handle des Fensters, das angeklickt wurde.

Tab. 22.1: Nachrichtenparameter

Nachrichtenparameter	Beschreibung	Mausklick-Beispiel
Message	Eine 32-Bit-Zahl, die einer bestimmten Nachricht zugeordnet ist.	`&H201`, auch als `WM_LBUTTONDOWN` bezeichnet.
WParam	Ein 32-Bit-Parameter, dessen Bedeutung von der Nachricht abhängig ist.	Flags, die anzeigen, ob die Strg- und/oder Shift-Taste gedrückt ist, und ob eine der anderen Maustasten gleichzeitig gedrückt wurde.
LParam	Ein 32-Bit-Parameter, dessen Bedeutung von der Nachricht abhängig ist.	Die unteren 16 Bit enthalten die horizontale Position des Mausklicks im Fenster, die oberen 16 Bit die vertikale Position.

Tab. 22.1: Nachrichtenparameter

Wie sendet das Betriebssystem eine Nachricht an ein Fenster? In jeder Windows-Applikation gibt es eine Schleife, in der das Programm ständig das Betriebssystem abfragt, um zu prüfen, ob Nachrichten anstehen. Diese Schleife, die Applikationsnachrichtenschleife, wird erst beendet, wenn die Applikation geschlossen wird. Sie bleibt vor den Visual-Basic-Programmierern vollständig verborgen. Wenn die Schleife sieht, daß eine Nachricht ansteht, gibt sie sie an das Fenster weiter, das diese erhalten soll.

Aber wie macht sie das? Für jedes Fenster im System ist eine Funktion definiert, die sogenannte Fensterfunktion. Die Fensterfunktion für das Fenster hat vier Parameter. Eine Fensterfunktion würde in Visual Basic wie folgt aussehen:

```
Public Function MyWindowFunction(ByVal hWnd As Long, ByVal message As _
    Long, ByVal wParam As Long, ByVal lParam As Long) As Long
```

Mit anderen Worten, die Nachrichtenschleife für die Applikation ruft eine API-Funktion auf, die nach der Fensterfunktion für ein Fenster sucht und diese mit den Nachrichtenparametern aufruft.

Es ist auch möglich, die Fensterfunktion für ein Fenster direkt aufzurufen, und damit eine Nachricht an das Fenster zu senden. Es gibt zwei API-Funktionen, die Sie in der Regel dafür verwenden werden, und die beide die Standardparameter verarbeiten (`hWnd`, `Message`, `wParam` und `lParam`):

- `PostMessage` – Diese API-Funktion lädt die Nachricht in die Nachrichtenwarteschlange für die Applikation. Die Funktion kehrt unmittelbar zurück und die Nachricht wird verarbeitet, wenn die Nachrichtenschlange der Applikation an dieser Nachricht angekommen ist. Weil die Nachricht nach der Rückkehr des `PostMessage`-Aufrufs ausgeführt wird, ist es nicht möglich, daß Fenster Werte an Funktionen zurückzugeben, die `PostMessage` aufrufen.

- `SendMessage` – Diese API-Funktion ruft die Fensterfunktion für das angegebene Fenster direkt auf. Die `SendMessage`-Funktion kehrt erst zurück, nachdem die Fensterfunktion die Nachricht verarbeitet hat. Weil die Nachricht verarbeitet wird, bevor `SendMessage` zurückkehrt, ist es damit möglich, daß eine Fensterfunktion einen Wert an die aufrufende Funktion zurückgibt.

Bei der Arbeit mit Windows bezieht sich der Begriff »eine Nachricht übergeben« immer auf die Verwendung von `PostMessage` oder entsprechender API-Funktionen, um eine Nachricht in die Nachrichtenwarteschlange zu übergeben. Der Begriff, eine Nachricht zu senden, bezieht sich immer auf die Verwendung von `SendMessage` oder entsprechender API-Funktionen, die die Fensterfunktion für ein Fenster unmittelbar aufrufen.

22.2.2 Was passiert, wenn ein Fenster erzeugt wird?

Wenn ein Fenster erzeugt wird, egal ob Formular, Steuerelement oder benutzerdefiniertes Fenster, erhält es von der erstellenden Applikation immer eine Fensterfunktion. Das ist eine der Aufgaben, die Visual Basic für Sie übernimmt. Wenn eine Applikation Nachrichten erhält, kann sie diese nach ihren eigenen Vorgaben verarbeiten, oder sie kann sie an eine Klassenfensterfunktion senden, die Windows bereitstellt. Beispielsweise definiert Windows eine Fensterklasse namens `LISTBOX`. Ein Fenster, das dieser Klasse angehört, kann eine von Windows bereitgestellte Klassenfensterfunktion aufrufen, die das Standardverhalten eines Listenfelds implementiert. Dadurch wird es für Applikationen möglich, Listenfelder zu erzeugen, ohne die gesamte Komplexität eines Listenfelds selbst implementieren zu müssen.

Abbildung 22.1 zeigt den Steuerfluß, der stattfindet, wenn eine Nachricht an ein Fenster geschickt wird. Die Applikation (oder das Betriebssystem) ruft die Fensterfunktion auf. Die Fensterfunktion kann entweder die Klassenfensterfunktion aufrufen, um das von der Klasse bereitgestellte Standardverhalten zu implementieren, oder die Nachricht selbst verarbeiten. Die Pfeile mit gestrichelter Linie zeigen einen optionalen Weg für den Programmfluß an.

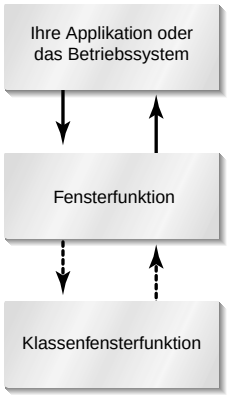

Abb. 22.1: Programmfluß für ein Fenster

22.2.3 Was ist Subclassing?

Visual Basic stellt Fensterfunktionen für die von ihm angelegten Formulare und Steuerelemente bereit. Diese Fensterfunktion implementiert nicht nur das Verhalten von Formularen und Steuerelementen, sondern wirft auch Visual-Basic-Ereignisse für ausgewählte eingehende Nachrichten auf.

Es erzeugt jedoch nicht für jede eingehende Nachricht ein Ereignis. Das sind die Fälle, wo Sie vielleicht das Standardverhalten des Formulars oder Steuerelements, das die von VB bereitgestellte Fensterfunktion bietet, überschreiben möchten. Die API-Funktion `SetWindowLong` kann genutzt werden, um die Fensterfunktion für ein Fenster auf eine von Ihnen definierte Funktion zu setzen. Damit haben Sie die Möglichkeit, gegebenenfalls auch die vorhergehende Fensterfunktion aufzurufen. Diese Situation ist in Abbildung 22.2 dargestellt.

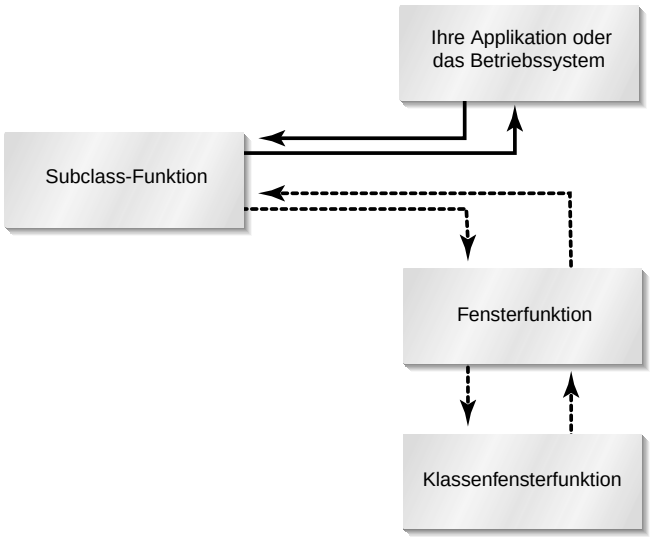

Abb. 22.2: Programmfluß für ein Fenster, für das eine Unterklasse angelegt wurde

22.2.4 Was ist eine Windows-Hook?

Eine Windows-Hook ist eine Technik, Nachrichten an verschiedenen Stellen der normalen Nachrichtenverarbeitungsfolge abzufangen. Hooks werden installiert, indem im Aufruf der Funktion `SetWindowsHookEx` die Adresse einer Hook-Funktion angegeben wird. Einige der gebräuchlichsten Hook-Typen sind im folgenden aufgelistet:

- `WH_GETMESSAGE`-Hook: Windows ruft Ihre Hook-Funktion immer auf, wenn Ihre Applikation eine Nachricht aus der Nachrichtenwarteschlange der Applikation anfordert. Die Hook-Funktion kann die Nachricht ändern oder löschen, bevor die Applikation sie zur Verarbeitung erhält.

- `WH_CALLWNDPROC`-Hook: Windows ruft Ihre Hook-Funktion unmittelbar vor dem Aufruf der Fensterfunktion für jedes Fenster in Ihrem Applikations-Thread auf. Die Hook-Funktion kann die Nachricht ändern oder löschen. Diese Art Hook ist jedoch extrem ineffizient, weil sie für jede Nachricht aufgerufen wird.
- `WH_KEYBOARD`-Hook: Windows ruft Ihre Hook-Funktion für jede Tastendruck-Nachricht auf (Taste nach oben und Taste nach unten), bevor die Nachricht in der Nachrichtenwarteschlange der Applikation abgelegt wird. Die Hook-Funktion kann Tastencodes ändern oder verwerfen.
- `WH_MOUSE`-Hook: Windows ruft Ihre Hook-Funktion für jede Mausnachricht auf, bevor diese in der Nachrichtenwarteschlange der Applikation abgelegt wird. Die Hook-Funktion kann Mausnachrichten ändern oder verwerfen.

Hooks müssen sehr sorgfältig eingesetzt werden, weil sie Windows den falschen Eindruck darüber vermitteln können, was im System passiert.

Außerdem ist es möglich, Hooks einzurichten, die auf systemübergreifender Basis arbeiten, oder die andere als Ihre eigenen Hooks verarbeiten. Ich zweifle jedoch ernsthaft daran, ob das unter Visual Basic (wenn überhaupt) sicher ist. Die Steuerelemente mit prozeßübergreifenden Hooks von SpyWorks sind alle in C++ geschrieben.

22.2.5 Benutzerdefinierte Fenster, Subclassing und Visual Basic

Eine der aufsehenerregendsten Funktionen, die Visual Basic 5 bot, war der `AddressOf`-Operator. Dieser Operator ermöglicht Ihnen, die Adresse einer Funktion in einem Standardmodul zu ermitteln. Sie können diese Adresse der API-Funktion CreateWindow übergeben und Ihre eigenen, privaten Fenster erzeugen und verwalten. Sie können die Adresse auch der API-Funktion `SetWindowLong` übergeben, um eine Unterklasse von einem bereits existierenden Fenster anzulegen. Dieser Ansatz wird auch in der Visual-Basic-Dokumentation aufgezeigt. Das bedeutet, Visual Basic ist jetzt in der Lage, private Fenster zu erzeugen und Unterklassen von existierenden Fenstern anzulegen, eine Aufgabe, für die zuvor Dritthersteller-Werkzeuge erforderlich waren.

Es gibt jedoch zahlreiche entscheidende Aspekte in Hinblick auf diese Techniken, die in der Visual-Basic-Dokumentation nicht erwähnt werden:

- Funktionen, die den `AddressOf`-Operator verwenden, müssen sich in Standardmodulen befinden. Wenn Sie eine Unterklasse eines Formulars oder eines Steuerelements anlegen, brauchen Sie aber eine Objektreferenz, um etwas Sinnvolles mit der eingehenden Nachricht tun zu können. Das bedeutet, Sie müssen eine effiziente Methode implementieren, eine Objektreferenz für ein Fenster zu erhalten – mit der Betonung auf dem Wort »effizient«. Diese Funktion wird für alle eingehenden Nachrichten aufgerufen; das letzte, was Sie an

dieser Stelle in Ihrem Code brauchen, ist ein größerer Overhead. Die Verwendung des `Collection`-Objekts von VB und spät gebundene Referenzen können tödlich sein.

- Es ist nicht unüblich für Visual-Basic-Steuerelemente, daß sie Unterklassen voneinander oder ihren Containern anlegen. Wenn sich eine davon fehlerhaft verhält, kann sie das Verhalten anderer Steuerelemente, die Unterklassen vom selben Fenster angelegt haben, unterminieren, was bis zu einer Speicherausnahme führen kann. Probleme treten insbesondere beim Entfernen eines Fensters oder beim Beenden der Applikation auf. Ihre Aufräum-Operation muß sich ebenfalls entsprechend verhalten.
- Die Verwendung des Subclassings ist von Natur aus gefährlich. Es umgeht alle in Visual Basic eingebauten Sicherheitsmechanismen. Wenn eine Nachrichteneigenschaft fehlerhaft verarbeitet wird, führt das nicht nur zu einem Laufzeitfehler von Visual Basic. Es ist sehr wahrscheinlich, daß eine Speicherausnahme auftritt, was insbesondere die normale Arbeitsweise Ihres Betriebssystems stört (vor allem unter Windows 95, das in dieser Hinsicht sehr viel weniger robust als Windows NT ist).
- Die zuverlässige Verwendung dieser Techniken macht es erforderlich, daß jede Nachricht verarbeitet wird. Aber wenn Sie Ihr Programm anhalten oder in den Break-Modus eintreten, wird Ihr Visual-Basic-Code – und damit auch der Code für Ihre Fensterfunktionen – sofort angehalten. Aus diesem Grund empfehle ich Ihnen wirklich, Ihre Fensterfunktionen in einem separaten ActiveX-Steuerelement oder einer DLL zu implementieren, unabhängig davon, ob Sie ein Drittanbieterprodukt oder Ihren eigenen Code verwenden.

Diese Aspekte führen mich zu einem größeren Dilemma, nämlich wie ich diese Techniken in diesem Buch demonstrieren soll. Also, mein Team hat einen enormen Aufwand betrieben, eine Visual-Basic-Komponente zu schaffen, die das Subclassing, private Fenster-Hooks und mehr in Version 5 unseres SpyWorks-Produkts einzufügen. Dieses Produkt soll die VB-Programmierer mit der sichersten und effizientesten Lösung versorgen, die für diese Low-Level-Fensterfunktionalität möglich ist. Außerdem beinhaltet es einige High-Level-Lösungen, die auf dieser Technologie basieren und insbesondere daraufhin geschaffen wurden, die Entwicklung von ActiveX-Steuerelementen zu unterstützen. Außerdem sollen sie lehrreich sein, weil sie den kompletten Quellcode beinhalten. Damit hatte ich mehrere Möglichkeiten.

Ich hätte eine Untermenge der Komponente mit einem Teil des Quellcodes vorstellen können. Der Rahmen für dieses Buch erlaubte jedoch keine geeignete Erklärung des Codes. Ich hätte die ganze Komponente aufnehmen können, aber offen gesagt, konnte ich mir das nicht leisten. Unsere Kunden verstehen, daß das, was sie für unsere Software zahlen, direkt dazu verwendet wird, ihnen Support zu leisten und immer wieder neue Funktionen und neue Produkte zur Verfügung zu stellen. Ich hätte die Verwendung der Komponente auch ganz weglassen und statt

dessen die in der VB-Dokumentation gezeigte Technik erklären können. Aber dann hätte ich Sie irregeführt, indem ich Ihnen Code gezeigt hätte, den ich in meinen eigenen Projekten nie einsetzen würde. Ich hätte eine Demo-Version der Komponente bereitstellen können, die voll funktional ist, so lange sie in Visual Basic selbst ausgeführt wird.

Ich habe mich für diesen letzteren Ansatz entschieden. Der erste Kunde, für den jede SpyWorks-Komponente entwickelt wird, ist unser eigenes technisches Team. Ich persönlich habe die meisten Funktionen in dieser betreffenden Komponente implementiert, während ich selbst ActiveX-Steuerelemente entwickelte (und dieses Buch schrieb). Ich glaube, Programmierer sollten nie Software verkaufen, die sie nicht auch in ihren eigenen Applikationen einsetzen würden. Dies sind die Komponenten, die ich selbst verwende. Ich hoffe also, Sie finden diese Komponente (und die anderen von SpyWorks) praktisch und kosteneffizient. Bei der Beschreibung werde ich versuchen zu erklären, was die einzelnen Aufrufe der Komponente erledigen, so daß Sie die Funktionalität der Komponente gegebenenfalls selbst nachbilden können. Für die Interessierten unter Ihnen: SpyWorks enthält den gesamten Quellcode für die in diesem Kapitel verwendeten Komponenten.

22.2.6 Messaging-Beispiele

Es ist einfach nicht möglich, das gesamte Feld der komplexen Techniken abzudecken, die durch die Verwendung des API und der Messaging-Techniken möglich geworden sind. Ich kann nur hoffen, Ihnen ein paar repräsentative Beispiele zeigen zu können. In diesem Abschnitt sehen Sie zwei der gebräuchlichsten Beispiele für die Verwendung von ActiveX-Steuerelementen. Im nächsten Abschnitt sehen Sie, wie mehrere dieser Techniken kombiniert werden können, um komplexe Steuerelemente zu erzeugen.

22.2.7 Komplexe Verarbeitung von Tastencodes

Das Steuerelement `dwCounter` zeigt ein häufig auftretendes Problem benutzerdefinierter Steuerelemente auf. Sie haben bereits gesehen, daß Visual Basic die Möglichkeit bietet, sich mit der Tab-Taste zwischen zwei benachbarten Elementen eines Steuerelements zu bewegen. Aber was tun Sie, wenn Sie in Ihrem benutzerdefinierten Steuerelement in der Lage sein möchten, mit der Tab-Taste zwischen den Elementen zu wechseln?

Das ist der Fall beim Steuerelement `dwCounter`, das Sie in Abbildung 22.3 sehen. Dieses Steuerelement kann man sich als Zähler vorstellen, wobei jede der Ziffern einzeln angeklickt oder geändert werden kann.

Das Steuerelement behandelt jede Ziffer unabhängig von der anderen, d.h. es muß möglich sein, nicht nur den Fokus auf das gesamte Steuerelement zu setzen, sondern auch einfach zwischen den Ziffern hin und her zu springen. Das ist typisch für ein komplexeres Problem, wo man ein Steuerelement hat, das ein

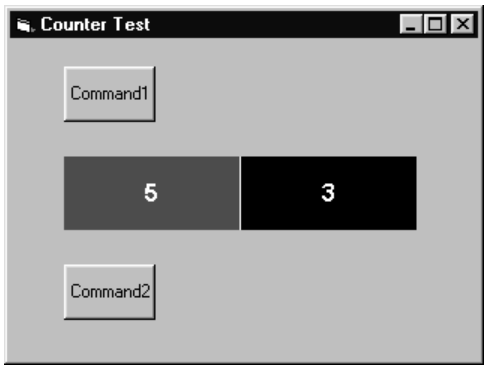

Abb. 22.3: Zähler-Steuerelement

komplexes Formular enthält, für das es möglich sein soll, sich zwischen den verschiedenen Elementen zu bewegen und sie unabhängig voneinander zu verändern, ohne mehrere einzelne Textfelder zu benutzen.

In Listing 22.1 beginnt unsere Betrachtung, wie dieses Steuerelement implementiert wird. Dabei wird der Code gezeigt, der die Eigenschaften des Steuerelements implementiert. Das Listing beinhaltet außerdem Ereignis- und API-Deklarationen und wird später in diesem Kapitel beschrieben.

```
' dwCounter-Steuerelement
' Teil von Desaware's ActiveX Gallimaufry
' Copyright © 1997-1998 by Desaware Inc.  All Rights Reserved.

Option Explicit
'Standardeigenschaftswerte:
Const m_def_Digits = 2
Const m_def_Value = 0
Const m_def_FocusColor = &HFF&   ' Rot
'Property Variables:
Dim m_Digits As Integer
Dim m_Value As Long
Dim m_FocusColor As OLE_COLOR

'Ereignis-Deklarationen:
Event Click() 'MappingInfo=UserControl,UserControl,-1,Click
Event KeyDown(KeyCode As Integer, Shift As Integer) _
'MappingInfo=UserControl, UserControl,-1,KeyDown
Event KeyPress(KeyAscii As Integer) 'MappingInfo=UserControl,UserControl _
,-1,KeyPress
Event KeyUp(KeyCode As Integer, Shift As Integer) _
'MappingInfo=UserControl,UserControl,-1,KeyUp
```

```
    ' API-Kram
    Private Const WM_KEYDOWN = &H100
    Private Const WM_KEYUP = &H101
    ' Keine Neudeklaration von InvalidateRect
    Private Declare Function InvalidateRect Lib "user32" (ByVal hwnd As Long, _
    ByVal lpRect As Long, ByVal bErase As Long) As Long

    ' Andere Werte
    Dim m_DigitWidth As Long
    Dim m_FocusIsAt As Integer
    Dim WithEvents PretranslateHook As dwPretranslate
    Dim m_ClickedDigit As Integer
    Dim m_Initializing As Boolean   ' True, während das Steuerelement
                                    ' initialisiert wird
    Dim WithEvents m_Font As StdFont


    Private Sub UserControl_Initialize()
      m_ClickedDigit = 1
      m_Initializing = True
      Set m_Font = New StdFont
      Set UserControl.Font = m_Font
    End Sub

Private Sub m_Font_FontChanged(ByVal PropertyName As String)
  Set UserControl.Font = m_Font
End Sub


    'ACHTUNG! DIE FOLGENDEN AUSKOMMENTIERTEN ZEILEN NICHT
    'LÖSCHEN UND NICHT ÄNDERN!
    'MappingInfo=UserControl,UserControl,-1,BackColor
    Public Property Get BackColor() As OLE_COLOR
      BackColor = UserControl.BackColor
    End Property

    Public Property Let BackColor(ByVal New_BackColor As OLE_COLOR)
      UserControl.BackColor() = New_BackColor
      PropertyChanged "BackColor"
    End Property

    'ACHTUNG! DIE FOLGENDEN AUSKOMMENTIERTEN ZEILEN NICHT
    'LÖSCHEN UND NICHT ÄNDERN!
    'MappingInfo=UserControl,UserControl,-1,BorderStyle
    Public Property Get BorderStyle() As Integer
      BorderStyle = UserControl.BorderStyle
    End Property
```

```
    Public Property Let BorderStyle(ByVal New_BorderStyle As Integer)
      UserControl.BorderStyle() = New_BorderStyle
      PropertyChanged "BorderStyle"
    End Property

    'ACHTUNG! DIE FOLGENDEN AUSKOMMENTIERTEN ZEILEN NICHT
    'LÖSCHEN UND NICHT ÄNDERN!
    'MappingInfo=UserControl,UserControl,-1,Enabled
    Public Property Get Enabled() As Boolean
      Enabled = UserControl.Enabled
    End Property

    Public Property Let Enabled(ByVal New_Enabled As Boolean)
      UserControl.Enabled() = New_Enabled
      PropertyChanged "Enabled"
    End Property

    'ACHTUNG! DIE FOLGENDEN AUSKOMMENTIERTEN ZEILEN NICHT
    'LÖSCHEN UND NICHT ÄNDERN!
    'MappingInfo=UserControl,UserControl,-1,Font
    Public Property Get Font() As Font
      Set Font = m_Font
    End Property

Public Property Set Font(ByVal New_Font As Font)
  With m_Font
     .Bold = New_Font.Bold
     .Charset = New_Font.Charset
     .Italic = New_Font.Italic
     .Name = New_Font.Name
     .Size = New_Font.Size
     .Strikethrough = New_Font.Strikethrough
     .Underline = New_Font.Underline
     .Weight = New_Font.Weight
  End With
  PropertyChanged "Font"
  UserControl.Refresh
End Property



    'ACHTUNG! DIE FOLGENDEN AUSKOMMENTIERTEN ZEILEN NICHT
    'LÖSCHEN UND NICHT ÄNDERN!
    'MappingInfo=UserControl,UserControl,-1,ForeColor
    Public Property Get ForeColor() As OLE_COLOR
      ForeColor = UserControl.ForeColor
    End Property
```

```
Public Property Let ForeColor(ByVal New_ForeColor As OLE_COLOR)
  UserControl.ForeColor() = New_ForeColor
  PropertyChanged "ForeColor"
End Property

Public Property Get FocusColor() As OLE_COLOR
  FocusColor = m_FocusColor
End Property

Public Property Let FocusColor(ByVal New_FocusColor As OLE_COLOR)
  m_FocusColor = New_FocusColor
  PropertyChanged "FocusColor"
End Property

'ACHTUNG! DIE FOLGENDEN AUSKOMMENTIERTEN ZEILEN NICHT
'LÖSCHEN UND NICHT ÄNDERN!
'MappingInfo=UserControl,UserControl,-1,hWnd
Public Property Get hwnd() As Long
  hwnd = UserControl.hwnd
End Property

Public Property Get Digits() As Integer
  Digits = m_Digits
End Property

Public Property Let Digits(ByVal New_Digits As Integer)
  If New_Digits < 0 Or New_Digits > 10 Then
     Err.Raise 380
  End If
  If Ambient.UserMode Then
     ' In dieser Beispielversion stehen die Ziffern nur zur Entwurfszeit zur
     ' Verfügung
     Err.Raise 382
  End If
  m_Digits = New_Digits
  PropertyChanged "Digits"
  SetSize
  UserControl.Refresh
End Property

Public Property Get Value() As Long
  Value = m_Value
End Property

Public Property Let Value(ByVal New_Value As Long)
  Dim MaxValue&
  ' Trick, um den Maximalwert zu erhalten
```

```
    ' Beispiel: 4 Ziffern sind 999
    MaxValue = Val(String$(m_Digits, "9"))
    If New_Value > MaxValue Then
       ' In dieser Beispielversion gibt es keine automatische Einstellung
       ' der Ziffern
       Err.Raise 380
    End If
    m_Value = New_Value
    PropertyChanged "Value"
    UserControl.Refresh
End Property

'Eigenschaften für das Benutzer-Steuerelement initialisieren
Private Sub UserControl_InitProperties()
    Set Font = Ambient.Font
    m_Digits = m_def_Digits
    m_Value = m_def_Value
    m_FocusColor = m_def_FocusColor
End Sub

'Eigenschaftswerte aus dem Speicher laden
Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
    UserControl.BackColor = PropBag.ReadProperty("BackColor", &H8000000F)
    UserControl.BorderStyle = PropBag.ReadProperty("BorderStyle", 0)
    UserControl.Enabled = PropBag.ReadProperty("Enabled", True)
    Set Font = PropBag.ReadProperty("Font", Ambient.Font)
    UserControl.ForeColor = PropBag.ReadProperty("ForeColor", &H80000012)
    m_FocusColor = PropBag.ReadProperty("FocusColor", m_def_FocusColor)
    m_Digits = PropBag.ReadProperty("Digits", m_def_Digits)
    m_Value = PropBag.ReadProperty("Value", m_def_Value)
End Sub

'Eigenschaftswerte in den Speicher schreiben
Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
    Call PropBag.WriteProperty("BackColor", UserControl.BackColor, &H8000000F)
    Call PropBag.WriteProperty("BorderStyle", UserControl.BorderStyle, 0)
    Call PropBag.WriteProperty("Enabled", UserControl.Enabled, True)
    Call PropBag.WriteProperty("Font", Font, Ambient.Font)
    Call PropBag.WriteProperty("ForeColor", UserControl.ForeColor, &H80000012)
    Call PropBag.WriteProperty("FocusColor", m_FocusColor, m_def_FocusColor)
    Call PropBag.WriteProperty("Digits", m_Digits, m_def_Digits)
    Call PropBag.WriteProperty("Value", m_Value, m_def_Value)
End Sub
```

Listing. 22.1: Der Code für die Eigenschaften des Zähler-Steuerelements

Die meisten Eigenschaften werden unter Verwendung der Standardtechniken implementiert und wurden tatsächlich mit dem ActiveX-Interface-Assistenten erzeugt. Diese Version des Steuerelements verwendet drei benutzerdefinierte Eigenschaften:

- `FocusColor` – Die Hintergrundfarbe für die Ziffer, die den Fokus besitzt. Der Fokus wird mit Hilfe der Pfeiltasten oder der Tabulator-Taste weitergegeben.
- `Digits` – Die Anzahl der darzustellenden Ziffern.
- `Value` – Der aktuelle numerische Wert, der angezeigt werden soll.

Die Eigenschaften Digits und Value verwenden eine einfache Bereichsprüfung, um sicherzustellen, daß gültige Werte zugewiesen werden. Sie werfen den Fehler 380 (Ungültiger Eigenschaftswert) auf, falls ein ungültiger Wert zugewiesen wird. Die Eigenschaftsprozedur für Digits und das Resize-Ereignis des Steuerelements rufen außerdem die folgende Funktion auf, um sicherzustellen, daß das Steuerelement groß genug ist, um die Ziffern in der aktuell ausgewählten Schrift anzuzeigen:

```
' Setzt die Mindestgröße für das Steuerelement
Private Sub SetSize()
   Dim minwidth&
   Dim useheight&
   ' Berechnet die Breite jeder Ziffer
   m_DigitWidth = ScaleWidth / Digits
   minwidth = Digits * TextWidth("W")
   useheight = TextHeight("1")
   If ScaleWidth < minwidth Or ScaleHeight < useheight Then
      ' Steuerelement ist zu klein
      If ScaleHeight > useheight Then useheight = ScaleHeight
      UserControl.Size ScaleWidth, useheight
      Exit Sub
   End If
End Sub

Private Sub UserControl_Resize()
   SetSize
End Sub
```

Das Steuerelement verwendet die interne Variable `m_FocusIsAt`, die den Fokusstatus des Steuerelements anzeigt. Wenn das Steuerelement den Fokus nicht besitzt, ist die Variable auf 0 gesetzt. Wenn das Steuerelement den Fokus nicht hat, wird die Variable auf die Nummer des Elements gesetzt (beginnend von 1), das den Fokus besitzt. Die Ereignisse `EnterFocus` und `ExitFocus` steuern den Ausgangswert der Variablen, wenn das Steuerelement den Fokus erhält. Sie setzen sie auf 0, wenn das Steuerelement den Fokus verliert:

```
Private Sub UserControl_EnterFocus()
       m_FocusIsAt = m_ClickedDigit
       UserControl.Refresh
    End Sub

    Private Sub UserControl_ExitFocus()
       m_FocusIsAt = 0
       m_ClickedDigit = 1   ' Auf erste Ziffer zurücksetzen
       UserControl.Refresh
    End Sub

    Private Sub UserControl_MouseDown(Button As Integer, Shift As Integer, _
    X As Single, Y As Single)
       m_ClickedDigit = Int(X / m_DigitWidth) + 1
       If m_FocusIsAt > 0 Then ' Ändern, wenn der Fokus bereits erhalten wurde
          m_FocusIsAt = m_ClickedDigit
          UserControl.Refresh
       End If
    End Sub
```

Die Variable `m_ClickedDigit` beobachtet, welche Ziffer angeklickt wurde, falls der Fokus mit Hilfe eines Mausklicks auf das Steuerelement gesetzt wurde. Diese Ziffer erhält den Fokus unmittelbar, wenn das Steuerelement den Fokus besitzt, oder durch das EnterFocus-Ereignis, wenn das Steuerelement aufgrund des Klicks den Fokus erhält.

Das Erscheinungsbild des Steuerelements wird durch das `Paint`-Ereignis bestimmt, das im folgenden gezeigt ist:

```
' In dieser Beispielversion gibt es keine komplexen
    ' Randeinstellungen
    Private Sub UserControl_Paint()
       Dim ypos&
       Dim txt$, fmt$
       Dim charpos%
       Dim thischar$
       If Not Ambient.UserMode Then
          ' Zur Entwurfszeit wird der Steuerelementname angezeigt
          txt$ = Ambient.DisplayName
          CurrentY = (ScaleHeight - TextHeight(txt)) / 2
          CurrentX = (ScaleWidth - TextWidth(txt)) / 2
          If CurrentX < 0 Then CurrentX = 0
          Print txt
          Exit Sub
       End If
       ypos = (ScaleHeight - TextHeight("1")) / 2
       fmt$ = String$(m_Digits, "0")
       txt$ = Format$(m_Value, fmt$)
```

```
    For charpos = 1 To m_Digits
       If charpos = m_FocusIsAt Then
          UserControl.Line ((charpos - 1) * m_DigitWidth, 0)-(charpos _
          * m_DigitWidth, ScaleHeight), m_FocusColor, BF
       End If
       thischar$ = Mid$(txt$, charpos, 1)
       CurrentX = m_DigitWidth * (charpos - 1) + (m_DigitWidth _
       - TextWidth(thischar)) / 2
       CurrentY = ypos
       UserControl.Print thischar
    Next charpos
    For charpos = 1 To m_Digits - 1
       UserControl.Line (m_DigitWidth * charpos, 0)-(m_DigitWidth * _
       charpos, ScaleHeight)
    Next charpos
End Sub
```

Das Aussehen des Steuerelements zur Entwurfszeit unterscheidet sich wesentlich von dem zur Laufzeit. Zur Entwurfszeit zeigt das Steuerelement einfach in seiner Mitte die `Ambient DisplayName`-Eigenschaft in der aktuellen Schriftart an. Der `DisplayName` ist der Name, den der Entwickler dem Steuerelement zugewiesen hat.

Zur Laufzeit ermittelt die Routine zuerst einen String, wobei jedes Zeichen eine Ziffer im Steuerelement darstellt. Ein Formatstring mit der richtigen Länge wird definiert. Dieser String wird der `Format`-Funktion von VB übergeben, um einen String mit der richtigen Anzahl führender Nullen zu laden.

Die Position der einzelnen Ziffern wird separat berechnet, so daß sie in der Mitte des für sie reservierten Bereichs erscheinen. Wenn eine Ziffer den Fokus bereits besitzt, wird ihre Hintergrundfarbe zunächst auf den in der Eigenschaft `FocusColor` festgelegten Wert gesetzt. Schließlich wird eine Linie zwischen den beiden Ziffern gezogen. Eine spätere Version dieses Steuerelements wird ein interessanteres Aussehen erhalten, aber für den Moment soll es genügen.

Das letzte Problem, das wir mit diesem Steuerelement haben, ist die Bereitstellung einer Methode für den Benutzer, den Fokus zwischen den beiden Ziffern im Steuerelement weiterzuschalten. Für das Umschalten des Fokus sind die Pfeile nach rechts und nach unten definiert. Die Pfeile nach links und nach oben schalten den Fokus zur vorherigen Ziffer. Das wird mit Hilfe des folgenden Codes bewerkstelligt:

```
Private Sub UserControl_KeyDown(KeyCode As Integer, Shift As Integer)
       RaiseEvent KeyDown(KeyCode, Shift)
       If KeyCode = vbKeyRight Or KeyCode = vbKeyDown Then
          If m_FocusIsAt < m_Digits Then
             m_FocusIsAt = m_FocusIsAt + 1
             UserControl.Refresh
             Exit Sub
```

```
      End If
    End If
    If KeyCode = vbKeyLeft Or KeyCode = vbKeyUp Then
      If m_FocusIsAt > 1 Then
        m_FocusIsAt = m_FocusIsAt - 1
        UserControl.Refresh
        Exit Sub
      End If
    End If
  End Sub
  Private Sub UserControl_KeyUp(KeyCode As Integer, Shift As Integer)
    RaiseEvent KeyUp(KeyCode, Shift)
  End Sub
```

Als erstes wirft das `UserControl_KeyDown`-Ereignis das `KeyDown`-Ereignis des Steuerelements auf. Damit ist es dem Entwickler möglich, den Tastencode zu überschreiben. Anschließend überprüft die Routine, ob die Pfeiltasten gedrückt wurden. Wenn das der Fall ist, wird die Variable `m_FocusIsAt` geändert und das Steuerelement neu gezeichnet. Das Steuerelement erlaubt momentan nicht, die Pfeiltasten zu nutzen, um den Fokus an ein anderes Element weiterzuschalten.

Das `KeyPress`-Ereignis, das Sie als nächstes sehen, zeigt, wie die Eingabe einer Zahl, während eine Ziffer den Fokus hat, den Wert dieser Ziffer ändern kann. Ein Entwickler könnte dieses Verhalten leicht deaktivieren, indem er den Parameter `KeyAscii` während des `KeyPress`-Ereignisses des Steuerelements auf 0 setzt. Das ist möglich, weil das `KeyPress`-Ereignis für das Steuerelement aufgeworfen wird, bevor der Wert von `KeyAscii` von dieser Routine verarbeitet wird:

```
Private Sub UserControl_KeyPress(KeyAscii As Integer)
  Dim fmt$, txt$
  RaiseEvent KeyPress(KeyAscii)
  If KeyAscii >= vbKey0 And KeyAscii <= vbKey9 Then
    fmt$ = String$(m_Digits, "0")
    txt$ = Format$(m_Value, fmt$)
    Mid$(txt$, m_FocusIsAt, 1) = Chr$(KeyAscii)
    Value = txt$
  End If

End Sub
```

Ein Problem bei der Implementierung ist, daß Sie bisher die Tab-Taste nicht nutzen können, um den Fokus zwischen den Ziffern des Steuerelements zu verschieben. Die Tab-Taste wird nämlich vom Container aufgefangen, so daß man sich damit zwischen den verschiedenen Steuerelementen bewegen kann. Die `KeyDown`- und `KeyUp`-Ereignisse Ihres Steuerelements sehen sie nie (selbst wenn die `KeyPreview`-Eigenschaft für das Steuerelement auf `True` gesetzt ist).

Die Lösung für dieses Problem ist, die Tab-Taste aufzufangen, bevor sie in den Container gelangt. Das wird mit Hilfe eines `WH_GETMESSAGE`-Hook bewerkstel-

ligt. Die Komponente dewpyvb.dll enthält ein Objekt, dwPretranslate, das diese Hook intern nutzt, um eine Funktion ähnlich der Methode PreTranslate-Message der Visual C++ MFC Steuerelementklasse zu implementieren. Damit ist es möglich, jede Nachricht zu sehen, bevor die Applikation sie verarbeitet. Das Objekt dwPretranslate verwendet einen internen Filter, um Nachrichten zu ermitteln, die nur für das vorgegebene Fenster bestimmt sind, in diesem Fall ist das das Fenster für das Steuerelement. Damit wird der Overhead für das Steuerelement wesentlich reduziert. Weil die Pretranslate-Funktionalität in der Regel für Tastaturnachrichten verwendet wird, nutzt dieses Steuerelement noch einen weiteren Filter, mit dem Sie eine Beschränkung auf die Nachrichten WM_KEYDONW und WM_KEYUP vornehmen können.

Im folgenden Code wird die Variable PretranslateHook initialisiert:

```
Private Sub UserControl_Show()
  m_Initializing = False
  SetSize
  If Ambient.UserMode And PretranslateHook Is Nothing Then
     ' Not yet initialized
     Set PretranslateHook = New dwPretranslate
     PretranslateHook.KeyMessagesOnly = True
     PretranslateHook.hwnd = UserControl.hwnd
  End If
End Sub
```

Weil zur Entwurfszeit keine solche Vorverarbeitung notwendig ist, wird das PretranslateHook-Objekt erst zur Laufzeit gesetzt. Das Objekt wirft ein einziges Ereignis auf, PreTranslateMessage, wie hier gezeigt:

```
Private Sub PretranslateHook_PreTranslateMessage(ByVal hwnd As Long, _
Msg As Long, wParam As Long, lParam As Long, nodef As Boolean)
  Static UpPending As Boolean
  Dim IsShiftPressed As Boolean

  IsShiftPressed = GetKeyState(vbKeyShift) < 0
  ' Auf Tab-Taste prüfen
  If Msg = WM_KEYDOWN And wParam = vbKeyTab Then
     If m_FocusIsAt < m_Digits And Not IsShiftPressed Then
        m_FocusIsAt = m_FocusIsAt + 1
        Call InvalidateRect(UserControl.hwnd, 0, True)
        UpPending = True
        Msg = 0
        nodef = True
     End If
     If m_FocusIsAt > 1 And IsShiftPressed Then
        m_FocusIsAt = m_FocusIsAt - 1
        Call InvalidateRect(UserControl.hwnd, 0, True)
        UpPending = True
```

```
            Msg = 0
            nodef = True
         End If
         Exit Sub
      End If
      If Msg = WM_KEYUP And wParam = vbKeyTab And UpPending Then
         ' Anstehende Taste-nach-oben für Tab verwerfen
         Msg = 0
         nodef = True
         UpPending = False
      End If
   End Sub
```

Dieses Ereignis beinhaltet die Standardnachrichtenparameter. Die Parameter `Msg`, `wParam`, `lParam` und `nodef` werden als Referenz übergeben, das bedeutet, die Funktionen können ihre Werte ändern, bevor sie von der Applikation verarbeitet werden. Wenn Sie den Parameter `nodef` auf `True` setzen, verhindern Sie, daß nachfolgende Steuerelemente, die Windows-Hooks gesetzt haben, die Nachricht auffangen.

Als erstes prüft die Routine den aktuellen Status der Shift-Taste, so daß sie feststellen kann, in welche Richtung der Fokus weitergeschaltet werden soll. Die aktuelle Implementierung setzt den Fokus immer auf die erste Ziffer, wenn das Steuerelement den Fokus erhält; mit Shift-Tab in das Steuerelement gelangt man also zur ersten Ziffer, und nicht zur letzten. Dieses Verhalten kann geändert werden, das ist aber relativ aufwendig, weil Sie dann zuerst herausfinden müssen, welches Steuerelement zuvor den Fokus besaß. Es handelt sich hier jedoch um eine kleinere Einschränkung, weil nur wenige Leute die Shift-Tab-Kombination verwenden, um sich rückwärts zu bewegen.

Wenn eine Tab-Taste gedrückt wurde, stellt das Steuerelement fest, ob eine Fokusänderung erfolgen soll. Falls ja, ändert es den Wert der Variablen `m_FocusIsAt`.

Anschließend setzt es die statische Variable `UpPendig` auf `True`. Dies ist eine statische Variable, die zu der Ereignisprozedur gehört. Sie wird genutzt, wenn die `WM_KEYUP`-Nachricht ankommt, und darauf hinweist, daß die Tab-Taste losgelassen wurde. Sie sehen, die Ereignisprozedur verwirft die `WM_KEYDOWN`-Nachricht, denn Windows könnte verwirrt sein, wenn es feststellt, daß die Tab-Taste losgelassen wurde, ohne daß sie zuvor gedrückt wurde. Diese Variable wird also auf `True` gesetzt, bevor die Nachricht, daß die Tab-Taste gedrückt wurde, verworfen wird, als Signal, daß auch die Nachricht verworfen werden soll, daß sie losgelassen wurde.

Die Nachricht wird verworfen, indem der `Msg`-Wert auf 0 gesetzt wird. Die Null-Nachricht wird von allen Fenstern ignoriert. Der Parameter `nodef` wird ebenfalls auf `True` gesetzt, um zu verhindern, daß auch andere Hooks die Nachricht sehen.

Die API-Funktion `InvalidateRect` wird verwendet, um die Anzeige des Steuerelements zu aktualisieren. Warum verwendet man dazu nicht die `Refresh`-Methode von `UserControl`? Weil dieses Ereignis während einer Windows-Nachrichten-Hook auftritt, was ein sehr gefährlicher Zeitpunkt für viele Operationen ist. Es ist immer sinnvoll, Hook-Funktionen so einfach wie möglich zu halten und komplexe Operationen zu vermeiden, die das System verwirren könnten, wenn sie mitten in einer Operation zum Versenden einer Nachricht auftreten. Eine `Refresh`-Operation mit dem gesamten Code aus dem `Paint`-Ereignis ist weitgehend unbekannt. Sie kann sicher arbeiten, stellt aber dennoch ein Risiko dar. Die API-Funktion `InvalidateRect` ist ein schneller Aufruf, der einfach nur Windows darüber informiert, daß der gesamte Bereich des Steuerelementfensters ungültig ist und neu gezeichnet werden muß.

Windows selbst stellt eine `WM_PAINT`-Nachricht für das Fenster in die Nachrichtenwarteschlange der Applikation, die dann wiederum das `Paint`-Ereignis des Steuerelements auslöst.

Eines, was dieses Steuerelement nicht erledigt, ist, den Fokus für eine Ziffer korrekt zuzuteilen, wenn Sie auf das Steuerelement klicken. Das bleibt dem Leser als Übung überlassen. (Hinweis: Verwenden Sie das `MouseDown`-Ereignis von `UserControl`.)

22.2.8 Verwaltung der zu aktualisierenden Bereiche

Eine der Einschränkungen des Visual-Basic-Modells für die ActiveX-Entwicklung in Hinblick auf benutzerdefinierte Steuerelemente betrifft das `Paint`-Ereignis. Das Problem dabei ist, daß Sie, wenn dieses Ereignis auftritt, das gesamte Steuerelement neu zeichnen müssen. Das kann für ein komplexes Steuerelement eine sehr zeitaufwendige Operation sein, und ein Großteil dieser Zeit kann gar vergeudet sein, wenn das `Paint`-Ereignis ausgelöst wird, aber nur ein kleiner Teil des Steuerelements neu gezeichnet werden muß. Das passiert häufig, wenn Fenster übereinander geschoben werden, oder wenn ein Dialogfeld oder ein Meldungsfeld verborgen werden.

Wenn die Zeichenroutine für Ihr Steuerelement so eingerichtet werden kann, daß nur die Teile des Fensters aktualisiert werden, die sich verändert haben, können Sie daraus einen wesentlichen Zeitvorteil erzielen. Sie ermitteln den Aktualisierungsbereich eines Fensters durch Bildung von Unterklassen, indem Sie die API-Funktion `GetUpdateRect` während der `WM_PAINT`-Nachricht für das Fenster aufrufen. Listing 22.2 zeigt ein Steuerelement, das diese Technik nicht nutzt, um sich selbst zu aktualisieren. Statt dessen legt es eine Unterklasse von seinem Container an, so daß der Container den Aktualisierungsbereich ermittelt.

```
' Guide to the Perplexed
    ' Beispiel für die Aktualisierung eines Fensterbereichs
    ' Copyright © 1997 by Desaware Inc. All Rights Reserved
```

```
Option Explicit

Private Type RECT
        Left As Long
        Top As Long
        Right As Long
        Bottom As Long
End Type

Private Declare Function GetUpdateRect Lib "user32" (ByVal hwnd As Long, _
lpRect As RECT, ByVal bErase As Long) As Long

Private Const WM_PAINT = &HF

Dim m_UpdateRect As RECT

Dim WithEvents UpdateHook As dwSubClass

Private Sub UpdateHook_WndMessage(ByVal hwnd As Long, Msg As Long, wp As _
Long, lp As Long, retval As Long, nodef As Boolean)
   Call GetUpdateRect(Extender.Parent.hwnd, m_UpdateRect, False)
End Sub

Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
   If Ambient.UserMode Then
      Set UpdateHook = New dwSubClass
      UpdateHook.AddMessage WM_PAINT
      UpdateHook.HwndParam = Extender.Parent.hwnd
   End If
End Sub

Private Sub UserControl_Resize()
   UserControl.Size UserControl.Picture.Width, UserControl.Picture.Height
End Sub

Private Sub UserControl_Terminate()
   Set UpdateHook = Nothing
End Sub

Public Property Get UpdateLeft() As Long
   UpdateLeft = m_UpdateRect.Left * Screen.TwipsPerPixelX
End Property

Public Property Get UpdateTop() As Long
   UpdateTop = m_UpdateRect.Top * Screen.TwipsPerPixelY
End Property
```

```
Public Property Get UpdateRight() As Long
  UpdateRight = m_UpdateRect.Right * Screen.TwipsPerPixelX
End Property

Public Property Get UpdateBottom() As Long
  UpdateBottom = m_UpdateRect.Bottom * Screen.TwipsPerPixelY
End Property
```

Listing. 22.2: Der Code für das Steuerelement PaintUpdate

Das Objekt `dwSubClass` ist Teil der Komponente `dwspyvb.dll`. Die Variable UpdateHook wird während des `ReadProperties`-Ereignisses initialisiert, wobei die `Extender`-Eigenschaft gültig ist. Die `HwndParam`-Eigenschaft des `UpdateHook`-Objekts setzt das Fenster, von dem eine Unterklasse angelegt werden soll, in diesem Fall unter Verwendung des Fensterhandles des Containers. Das Steuerelement könnte seinen eigenen Aktualisierungsbereich ermitteln, indem es statt dessen die `hWnd`-Eigenschaft des `UserControl`-Objekts verwendet.

Die Methode `AddMessage` setzt die `WM_PAINT`-Nachricht als einzige Nachricht, die abgefangen werden soll. Das Objekt filtert die Nachrichten auf unterster Ebene, um eine bestmögliche Leistung zu gewährleisten. Dadurch wird es auch überflüssig, während des `WndMessage`-Ereignisses des Objekts Nachrichten abzufangen. Die `WM_PAINT`-Nachricht ist die einzige, die ankommt.

Während des `WndMessage`-Ereignisses wird das zu aktualisierende Rechteck mit Hilfe der Funktion `GetUpdateRect` ermittelt und in der Variablen `m_UpdateRect` abgelegt. Die Felder dieser Variablen werden unter Verwendung der Eigenschaften UpdateLeft, UpdateTop, UpdateRight und UpdateBottom gelesen.

Das Programm UpdateTest demonstriert dies, indem es die oberen Koordinaten eines Anzeigefeld-Steuerelements im Formular `frmUpdate` anzeigt. Versuchen Sie, ein Fenster über einen Teil des Formulars zu ziehen, um zu sehen, wie das Ganze funktioniert.

22.3 Das Märchen mit den vier Listenfeldern

Wir werden unsere Beschreibung der komplexen Steuerelemente und unsere Diskussion der Steuerelemente im allgemeinen mit einer Betrachtung vier verschiedener Ansätze für die Implementierung eines benutzerdefinierten Listenfeld-Steuerelements fortsetzen. Unter anderem geht es dabei um:

- ein komponenten-basiertes Steuerelement
- ein vom Benutzer gezeichnetes Steuerelement
- ein besseres vom Benutzer gezeichnetes Steuerelement
- Ein benutzerdefiniertes fenster-basiertes Steuerelement

Bevor ich anfange, sollte ich betonen, daß diese Steuerelemente nicht als vollständig oder robust betrachtet werden sollen. Größtenteils werden dabei nur ein paar wenige Eigenschaften und Ereignisse implementiert – gerade genug, um die Techniken für den jeweiligen Ansatz zu demonstrieren. Sie wurden nur in geringem Umfang getestet und es gibt wenig bis keine Fehlerprüfung.

Die Projektgruppe `ListCtls.vbg` enthält zwei Projekte. `ListCtls.vbp` enthält die vier Steuerelemente, die die oben aufgeführten Ansätze demonstrieren. `ListTest.vbp` enthält vier Formulare, die einige der Charakteristika der vier Steuerelemente aufzeigen.

22.3.1 Ein komponenten-basiertes Steuerelement

Listing 22.3 enthält den Code für das Beispiel mit dem komponenten-basierten Steuerelement. Dieses Steuerelement zeigt die wichtigste Einschränkung dieses Ansatzes auf – daß es für den Entwickler unmöglich ist, zur Entwurfszeit die Eigenschaften eines komponenten-basierten Steuerelements zu setzen. Ihr Steuerelement befindet sich nämlich im Laufzeitmodus, auch wenn sich der Container in der Entwurfszeit befindet. Die Eigenschaft `MultiSelect` ist ein Beispiel für eine Eigenschaft, die zur Laufzeit nicht geändert werden kann. `ListCtlA` unterstützt die `MultiSelect`-Eigenschaft, indem es zwei separate Listenfeld-Steuerelemente verwendet, eines, das sich im Modus zur Mehrfachauswahl befindet, der andere für die Einfachauswahl.

```
' Beispiel für ein komponenten-basiertes Listenfeld
' Copyright (c) 1997-1998, by Desaware Inc. All Rights Reserved

Option Explicit
'Ereignis-Deklarationen:
Event Click()
Event DblClick()
Event KEYDOWN(KeyCode As Integer, Shift As Integer)
Event KeyPress(KeyAscii As Integer)
Event KEYUP(KeyCode As Integer, Shift As Integer)
Event MouseDown(Button As Integer, Shift As Integer, x As Single, _
Y As Single)
Event MouseMove(Button As Integer, Shift As Integer, x As Single, _
Y As Single)
Event MouseUp(Button As Integer, Shift As Integer, x As Single, Y As Single)

Dim CurrentListBox As ListBox
Dim m_MultiSelect As Boolean
Dim WithEvents m_Font As StdFont


Public Property Get MultiSelect() As Boolean
  MultiSelect = m_MultiSelect
End Property
```

```
    Public Property Get BackColor() As OLE_COLOR
      BackColor = CurrentListBox.BackColor
    End Property

    Public Property Let BackColor(ByVal New_BackColor As OLE_COLOR)
      List1.BackColor = New_BackColor
      List2.BackColor = New_BackColor
      PropertyChanged "BackColor"
    End Property

    Public Property Get ForeColor() As OLE_COLOR
      ForeColor = CurrentListBox.ForeColor
    End Property

    Public Property Let ForeColor(ByVal New_ForeColor As OLE_COLOR)
      List1.ForeColor = New_ForeColor
      List2.ForeColor = New_ForeColor
      PropertyChanged "ForeColor"
    End Property

    Public Property Get Enabled() As Boolean
      Enabled = CurrentListBox.Enabled
    End Property

    Public Property Let Enabled(ByVal New_Enabled As Boolean)
      List1.Enabled = New_Enabled
      List2.Enabled = New_Enabled
      PropertyChanged "Enabled"
    End Property

Public Property Get Font() As Font
  Set Font = CurrentListBox.Font
End Property

Public Property Set Font(ByVal New_Font As Font)
  With m_Font
     .Bold = New_Font.Bold
     .Charset = New_Font.Charset
     .Italic = New_Font.Italic
     .Name = New_Font.Name
     .Size = New_Font.Size
     .Strikethrough = New_Font.Strikethrough
     .Underline = New_Font.Underline
     .Weight = New_Font.Weight
  End With
  Set List1.Font = m_Font
  Set List2.Font = m_Font
```

```
    PropertyChanged "Font"
End Property


Public Sub Refresh()
  CurrentListBox.Refresh
End Sub

Private Sub List1_Click()
  RaiseEvent Click
End Sub

Private Sub List2_Click()
  RaiseEvent Click
End Sub


Private Sub List1_DblClick()
  RaiseEvent DblClick
End Sub

Private Sub List2_DblClick()
  RaiseEvent DblClick
End Sub

Private Sub List1_KeyDown(KeyCode As Integer, Shift As Integer)
  RaiseEvent KEYDOWN(KeyCode, Shift)
End Sub

Private Sub List2_KeyDown(KeyCode As Integer, Shift As Integer)
  RaiseEvent KEYDOWN(KeyCode, Shift)
End Sub

Private Sub List1_KeyPress(KeyAscii As Integer)
  RaiseEvent KeyPress(KeyAscii)
End Sub

Private Sub List2_KeyPress(KeyAscii As Integer)
  RaiseEvent KeyPress(KeyAscii)
End Sub

Private Sub List1_KeyUp(KeyCode As Integer, Shift As Integer)
  RaiseEvent KEYUP(KeyCode, Shift)
End Sub

Private Sub List2_KeyUp(KeyCode As Integer, Shift As Integer)
  RaiseEvent KEYUP(KeyCode, Shift)
End Sub
```

```
Private Sub List1_MouseDown(Button As Integer, Shift As Integer, _
x As Single, Y As Single)
  RaiseEvent MouseDown(Button, Shift, x, Y)
End Sub

Private Sub List2_MouseDown(Button As Integer, Shift As Integer, _
x As Single, Y As Single)
  RaiseEvent MouseDown(Button, Shift, x, Y)
End Sub

Private Sub List1_MouseMove(Button As Integer, Shift As Integer, _
x As Single, Y As Single)
  RaiseEvent MouseMove(Button, Shift, x, Y)
End Sub

Private Sub List2_MouseMove(Button As Integer, Shift As Integer, _
x As Single, Y As Single)
  RaiseEvent MouseMove(Button, Shift, x, Y)
End Sub

Private Sub List1_MouseUp(Button As Integer, Shift As Integer, x As Single, _
                      Y As Single)
  RaiseEvent MouseUp(Button, Shift, x, Y)
End Sub

Private Sub List2_MouseUp(Button As Integer, Shift As Integer, x As Single, _
                      Y As Single)
  RaiseEvent MouseUp(Button, Shift, x, Y)
End Sub

Public Sub AddItem(item As String, Optional Index As Variant)
  CurrentListBox.AddItem item, Index
End Sub

Public Sub Clear()
  CurrentListBox.Clear
End Sub


Public Property Get hwnd() As Long
  hwnd = CurrentListBox.hwnd
End Property

Public Property Get ItemData(ByVal ItemIndex As Long) As Long
  ItemData = CurrentListBox.ItemData(ItemIndex)
End Property
```

```
Public Property Let ItemData(ByVal ItemIndex As Long, ByVal New_ItemData _
                        As Long)
  CurrentListBox.ItemData(ItemIndex) = New_ItemData
  PropertyChanged "ItemData"
End Property

Public Property Get List(ByVal ItemIndex As Long) As String
  List = CurrentListBox.List(ItemIndex)
End Property

Public Property Let List(ByVal ItemIndex As Long, ByVal New_List As String)
  CurrentListBox.List(ItemIndex) = New_List
  PropertyChanged "List"
End Property

Public Property Get ListCount() As Integer
  ListCount = CurrentListBox.ListCount
End Property

Public Property Get ListIndex() As Integer
  ListIndex = CurrentListBox.ListIndex
End Property

Public Property Let ListIndex(ByVal New_ListIndex As Integer)
  CurrentListBox.ListIndex = New_ListIndex
  PropertyChanged "ListIndex"
End Property

Public Property Get MouseIcon() As Picture
  Set MouseIcon = CurrentListBox.MouseIcon
End Property

Public Property Set MouseIcon(ByVal New_MouseIcon As Picture)
  Set List1.MouseIcon = New_MouseIcon
  Set List2.MouseIcon = New_MouseIcon
  PropertyChanged "MouseIcon"
End Property

Public Property Get NewIndex() As Integer
  NewIndex = CurrentListBox.NewIndex
End Property

Public Sub RemoveItem(Index As Integer)
  CurrentListBox.RemoveItem Index
End Sub

Public Property Get Selected(ByVal ItemIndex As Long) As Boolean
```

```
        If CurrentListBox Is List2 Then
           Selected = CurrentListBox.Selected(ItemIndex)
        Else
           ListError 1000
        End If
     End Property

     Public Property Let Selected(ByVal ItemIndex As Long, ByVal New_Selected _
     As Boolean)
        If CurrentListBox Is List2 Then
           CurrentListBox.Selected(ItemIndex) = New_Selected
           PropertyChanged "Selected"
        Else
           ListError 1000
        End If
     End Property

Private Sub m_Font_FontChanged(ByVal PropertyName As String)
   Set List1.Font = m_Font
   Set List2.Font = m_Font
End Sub

Private Sub UserControl_Initialize()
   Set m_Font = New StdFont
   Set List1.Font = m_Font
   Set List2.Font = m_Font
End Sub


     Private Sub UserControl_InitProperties()
        Set CurrentListBox = List1
     End Sub

     'Eigenschaftswerte aus dem Speicher laden
     Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
        m_MultiSelect = PropBag.ReadProperty("MultiSelect", False)
        If m_MultiSelect Then
           Set CurrentListBox = List2
           List1.Visible = False
           List2.Visible = True
        Else
           Set CurrentListBox = List1
           List2.Visible = False
           List1.Visible = True
        End If
        BackColor = PropBag.ReadProperty("BackColor", &H80000005)
        ForeColor = PropBag.ReadProperty("ForeColor", &H80000008)
```

```
    Enabled = PropBag.ReadProperty("Enabled", True)
    Set Font = PropBag.ReadProperty("Font")
    Set CurrentListBox.MouseIcon = PropBag.ReadProperty("MouseIcon", Nothing)
End Sub


'Eigenschaftswerte in den Speicher schreiben
Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
    Call PropBag.WriteProperty("MultiSelect", m_MultiSelect, False)
    Call PropBag.WriteProperty("BackColor", _
    CurrentListBox.BackColor, &H80000005)
    Call PropBag.WriteProperty("ForeColor", _
    CurrentListBox.ForeColor, &H80000008)
    Call PropBag.WriteProperty("Enabled", CurrentListBox.Enabled, True)
    Call PropBag.WriteProperty("Font", m_Font)
    Call PropBag.WriteProperty("MouseIcon", CurrentListBox.MouseIcon, Nothing)
End Sub
```

Listing. 22.3: Listing für das Steuerelement ListCtlA.ctl

Wie Sie sehen, werden die meisten der Standardeigenschaften und Ereignisse so implementiert, wie Sie es in diesem Buch bereits mehrfach gesehen haben.

Bei dieser Implementierung werden die Eigenschaften beiden Steuerelementen zugeordnet. Es könnte effizienter sein, Variablen zu verwenden, die die Eigenschaftswerte aufnehmen, und nur die Komponenteneigenschaften für das aktuell sichtbare Steuerelement zu setzen. Dafür wäre jedoch auch zusätzlicher Code erforderlich, um beim Wechsel zwischen den Steuerelementen alle Eigenschaften des jeweils anderen Steuerelements zu laden.

Die Variable `CurrentListBox` dient als praktische Referenz auf das Komponenten-Steuerelement, das gerade verwendet wird. Sie wird während des `ReadProperties`-Ereignisses und der `Property Let`-Prozedur für die `MultiSelect`-Eigenschaft gesetzt, wie im folgenden gezeigt:

```
Public Property Let MultiSelect(ByVal vNewValue As Boolean)
    If UserControl.Ambient.UserMode Then
       ' Can't change list type at runtime
       SetNotSupportedAtRuntime
    End If
    m_MultiSelect = vNewValue
    If m_MultiSelect Then
       Set CurrentListBox = List2
       List1.Visible = False
       List2.Visible = True
    Else
       Set CurrentListBox = List1
       List1.Visible = True
       List2.Visible = False
```

```
    End If

    PropertyChanged "MultiSelect"
End Property
```

Beachten Sie, daß es mit diesem Ansatz auch möglich ist, dem Benutzer zur Laufzeit des Containers einen Wechsel zwischen dem Mehrfachauswahl- und dem Einfachauswahl-Modus zu bieten; diese Funktionalität ist jedoch hier nicht implementiert. Wenn Sie diese Funktion unterstützen möchten, müßten Sie auch entscheiden, ob der aktuelle Inhalt des Listenfelds bei dem Wechsel von dem einen in das andere Listenfeld kopiert werden soll. Dieses Beispiel wirft in diesem Fall einen Fehler auf, um mit dem Standard-Listenfeld kompatibel zu bleiben.

Das `Resize`-Ereignis positioniert beide Listenfelder, so daß sie den Client-Bereich Ihres Steuerelements ausfüllen, wie hier im `UserControl_Resize`-Ereignis gezeigt:

```
Private Sub UserControl_Resize()
    List1.Move 0, 0, ScaleWidth - Screen.TwipsPerPixelX, _
    ScaleHeight - Screen.TwipsPerPixelY
    List2.Move 0, 0, ScaleWidth - Screen.TwipsPerPixelX, _
    ScaleHeight - Screen.TwipsPerPixelY
End Sub
```

Was tun Sie, wenn Sie die `Appearance`-Eigenschaft des zusammengesetzten Steuerelements ebenfalls verarbeiten möchten? Dabei handelt es sich auch um eine Eigenschaft, die nur zur Entwurfszeit gesetzt werden kann, Sie bräuchten also vier separate Listenfelder, die zusammen das Steuerelement bilden, um diese Aufgabe zu erledigen. Die `Integral`-Eigenschaft führt Sie zu acht Elementen. Und wenn Sie dann noch die `Sorted`-Eigenschaft einfügen, sind Sie schnell bei 16 Komponenten.

Wie Sie sehen, kann dieser Ansatz sehr schnell sehr unpraktikabel werden. Der Overhead für die Unterstützung von 17 Fenstern für jedes Element (die 16 Listenfelder plus das `UserControl`-Objekt) ist gewaltig. Ein Programmierer, der diesen Ansatz wählt, sollte seinen Beruf (hoffentlich) an den Nagel hängen.

Das bedeutet jedoch nicht, daß Sie diesen Komponenten-Ansatz nie verwenden sollten, sondern nur, daß er unpraktisch ist, wenn Sie fordern, daß der Entwickler in der Lage sein sollte, mehr als eine (oder höchstens zwei) Entwurfszeit-Eigenschaft zu ändern.

22.3.2 Ein vom Benutzer gezeichnetes Steuerelement

Betrachten Sie einen Augenblick lang die von Windows standardmäßig implementierte Listenfeld-Fensterklasse, die Windows implementiert. Dies ist die Klasse, die Visual Basic verwendet, um sein Standard-Listenfeld-Steuerelement zu implementieren. Wie wird die eigentliche Klasse implementiert?

Wenn ein Listenfeld-Fenster erzeugt wird, erzeugt das System als erstes ein Fenster (ein ganz normales Fenster, wie alle anderen im System). Es weist dem Fenster die Klassenfensterfunktion für die Listenfeldklasse zu. Diese Fensterfunktion nimmt eingehende Nachrichten entgegen, die an das Fenster gesendet oder ihm übergeben wurden. Wenn eine Paint-Nachricht ankommt, zeichnet der Code das Listenfeld unter Verwendung von API-Funktionen. Maus- oder Tastatur-Nachrichten bewirken bestimmte Auswahl- und Scrolling-Operationen. An das Fenster können Steuernachrichten geschickt werden, um Aufgaben zu erledigen, wie etwa das Hinzufügen oder das Löschen von Strings. Dieses Listenfeld ist in der Lage, Nachrichten an sein Elternfenster zu senden, um es zu benachrichtigen, wenn ein bestimmtes Ereignis auftritt, beispielsweise ein Wechsel des Auswahlmodus. Die Charakteristika des Listenfeldes, beispielsweise, ob es sortiert ist, oder ob es eine Einzel- oder Mehrfachauswahl unterstützt, werden durch die Stileigenschaften des Fensters festgelegt. Jedes Fenster hat zwei 32-Bit-Stilvariablen. Einige der Bits in diesen Variablen sind standardisiert, sie geben beispielsweise an, ob das Fenster einen Rahmen hat. Andere sind vom Fenstertyp abhängig.

Wir werden später noch detaillierter auf dieses Thema zurückkommen. Überlegen Sie hier nur folgendes: Ein in Visual Basic geschriebenes ActiveX-Steuerelement nimmt Ereignisse entgegen. Es kann interne Variablen definieren. Es kann der Außenwelt Methoden und Eigenschaften bereitstellen. Es kann beliebige Dinge in sein Steuerelementfenster zeichnen. Der Mechanismus ist möglicherweise nicht derselbe wie eine vom System definierte Fensterklasse, aber der Effekt ist derselbe.

Ein Listenfeld-Fenster ist ein Fenster mit Code, der eingehende Nachrichten verarbeitet. Ein ActiveX-Steuerelement ist ein Fenster mit Code, der eingehende Nachrichten verarbeitet. Offensichtlich könnten Sie Ihr eigenes Listenfeld-Steuerelement von Grund auf neu implementieren, indem Sie die gesamte oder einen Teil der Funktionalität des Standard-Listenfeld-Fensters duplizieren. Wenn Ihnen das alles etwas verwirrend erscheint, dann vielleicht, weil die Aufgabe relativ komplex ist. Aber sie ist machbar, und mit der neuen Unterstützung, die Visual Basic für die Kompilierung von eigenem Code bietet, können Sie mit diesem Ansatz eine ausgezeichnete Performance erzielen.

Beachten Sie einen wichtigen Aspekt: Wenn Sie diesen Ansatz wählen, sind Sie nicht mehr auf die Funktionen oder Charakteristika des Standard-Listenfeld-Steuerelements von Visual Basic oder der Listenfeld-Fensterklasse beschränkt. Ihr Listenfeld-Steuerelement könnte Bitmaps aufnehmen, und sogar Audio- oder Video-Clips.

Das Beispiel `ListCtlB` demonstriert diesen Ansatz. In diesem Fall wurde das Steuerelement so entworfen, daß es das Erscheinungsbild und einen Großteil des Verhaltens des Standard-Listenfelds nachbildet, nur um zu zeigen, was möglich wäre. Auch hier möchte ich noch einmal betonen, daß es sich um ein Beispiel-Steuerelement handelt, das nur für Demonstrationszwecke entwickelt wurde. Es basiert auf Code, den ich in Konferenzen und Übungs-Sitzungen in den letzten

paar Jahren verwendet habe, um einige der von Visual Basic gebotenen Möglichkeiten zu demonstrieren. Es hat sich in dieser Zeit schrittweise weiterentwickelt, und dies ist die erste Inkarnation als ActiveX-Steuerelement. Beachten Sie, daß die Mehrfachauswahl, wie sie hier implementiert ist, nur ein schneller Versuch ist, der hinzugefügt wurde, um die Möglichkeit zu zeigen, wie in einem Steuerelement sowohl die Einfach- als auch die Mehrfachauswahl unterstützt werden kann. Die Mehrfachauswahl ist nicht mit der eines Standard-Listenfelds vergleichbar, und stellt auch keinen repräsentativen Entwurf dar. Eine Überarbeitung und Neuimplementierung soll in zukünftigen Versionen stattfinden.

Hier sind die Deklarationen und Variablen für das `ListCtlB`-Steuerelement gezeigt. Wie Sie sehen, nutzt dieses Steuerelement zahlreiche Win32-API-Funktionen:

```
' Beispiel für ein vom Benutzer gezeichnetes Steuerelement
    ' Copyright (c) 1997, by Desaware Inc. All Rights Reserved
    Option Explicit

    Private Type RECT   '16 Bytes
       left As Long
       top As Long
       right As Long
       bottom As Long
    End Type

    Private Type POINTAPI  '8 Bytes - Synonymous with LONG
       x As Long
       Y As Long
    End Type

    Private Declare Function DrawFocusRect& Lib "user32" (ByVal hdc As Long, _
    lpRect As RECT)
    Private Declare Function DrawText& Lib "user32" Alias "DrawTextA" _
    (ByVal hdc As Long, ByVal lpStr As String, ByVal nCount As Long, _
    lpRect As RECT, ByVal wFormat As Long)
    Private Declare Function GetSysColor& Lib "user32" (ByVal nIndex As Long)
    Private Declare Function InflateRect& Lib "user32" (lpRect As RECT, ByVal x _
    As Long, ByVal Y As Long)
    Private Declare Function ScrollWindowByNum& Lib "user32" Alias _
    "ScrollWindow" (ByVal hwnd As Long, ByVal XAmount As Long, ByVal _
    YAmount As Long, lpRect As RECT, ByVal lpClipRect As Long)
    Private Declare Function GetCursorPos& Lib "user32" (lpPoint As POINTAPI)
    Private Declare Function SetCursorPos& Lib "user32" (ByVal x As Long, _
    ByVal Y As Long)
    Private Declare Function SetCapture& Lib "user32" (ByVal hwnd As Long)
    Private Declare Function ReleaseCapture& Lib "user32" ()
    Private Declare Function GetFocus Lib "user32" () As Long
    Private Declare Function SetTextColor& Lib "gdi32" (ByVal hdc As Long, _
```

```
ByVal crColor As Long)
Private Declare Function SelectObject Lib "gdi32" (ByVal hdc As Long, ByVal _
hObject As Long) As Long

Private CurrentTop As Long
Private TotalLines As Long
Private PixelsPerLine As Integer
Private Selected() As Long    ' Nummer der ausgewählten Zeile
Private HasFocus As Long
Private highlight As Long
Private HighlightText As Long
' -1, -1 bedeutet, daß alle gezeichnet werden sollen
Private LowDrawRange As Long    ' Niedrigste Zeilennummer, für die ein
                                ' Zeichnen erforderlich ist
Private HighDrawRange As Long   ' Höchste Zeilennummer, für die ein Zeichnen
                                ' erforderlich ist
Private ClickLine As Long       ' Zeile, in der ein Klick erkannt wird
Private InContext As Integer    ' Reentrante Mausbewegungs-Ereignisse
                                ' verhindern
Private Light3D As Long       ' 3D Hervorheben
Private Dark3D As Long        ' 3D Schatten
Private DarkShadow As Long    ' 3D dunkler Schatten
Private FixNextScroll As Long

Private Const KEYDOWN = 40
Private Const KEYUP = 38
Private Const COLOR_HIGHLIGHT = 13
Private Const COLOR_HIGHLIGHTTEXT = 14
Private Const COLOR_BTNSHADOW = 16
Private Const COLOR_BTNHIGHLIGHT = 20
Private Const COLOR_3DDKSHADOW = 21
Private m_BorderStyle As Integer  ' 1 = Single, 2 = 3D, 0 = None
Private m_MultiSelect As Boolean  ' Dialog für die Mehrfachauswahl
Private WithEvents m_Font As StdFont



Private Const DT_LEFT = &H0
Private Const DT_SINGLELINE = &H20
Private Const DT_NOPREFIX = &H800
```

Das Steuerelement verwendet mehrere interne Variablen, die vorab berechnete Werte enthalten, beispielsweise die Anzahl der Pixel pro Zeile. Das Steuerelement besitzt eine Bildlaufleiste, so daß im Listenfeld geblättert werden kann.

Das Listen-Steuerelement enthält keine `String`-Daten. Statt dessen fordert es die String-Daten für jedes Element mit Hilfe des `GetText`-Ereignisses vom Container an. Dieser Ansatz bedingt zusätzliche Arbeit auf Seiten des Containers, kann aber

extrem effizient sein, weil es dabei nicht notwendig ist, daß Sie das Listenfeld vorab mit Daten laden. Das kann für Datenbankanwendungen ideal sein, wo nur die paar Elemente geladen werden müssen, die angezeigt werden sollen, wenn das Steuerelement zum ersten Mal angezeigt wird. Außerdem wird es dadurch ganz einfach, eine sehr große Anzahl von Einträgen effizient zu verarbeiten.

Der Code für die Eigenschaften und Methoden für das Steuerelement folgt. Wie Sie sehen, ist es möglich, die Eigenschaften `BorderStyle` und `MultiSelect` zur Entwurfszeit und zur Laufzeit des Containers zu setzen, weil diese Einfluß auf das Verhalten und das Erscheinungsbild des Steuerelements haben und vollständig im Code des Steuerelements implementiert sind:

```
'-----------------------------------------------------
'
'  Öffentliche Methoden und Ereignisse des Steuerelements
'
Event GetText(ByVal location As Long, ListBoxString As String)
Event Click()
Event DblClick()

Public Property Get BorderStyle() As dwBorderStyle
  BorderStyle = m_BorderStyle
End Property

Public Property Let BorderStyle(vNewBorder As dwBorderStyle)
  If vNewBorder < 0 Or vNewBorder > 2 Then
    ListError 380
  End If
  m_BorderStyle = vNewBorder
  PropertyChanged "BorderStyle"
  CalculateValues
  PositionScrollBar
  lbInvalidateRange
  Refresh
End Property

Public Property Get MultiSelect() As Boolean
  MultiSelect = m_MultiSelect
End Property

Public Property Let MultiSelect(ByVal vNewValue As Boolean)
  m_MultiSelect = vNewValue
  ReDim Preserve Selected(0)
  lbInvalidateRange
  Refresh
  PropertyChanged "MultiSelect"
End Property
```

```
    Public Property Get BackColor() As OLE_COLOR
      BackColor = UserControl.BackColor
    End Property

    Public Property Let BackColor(vNewColor As OLE_COLOR)
      UserControl.BackColor = vNewColor
      PropertyChanged "BackColor"
    End Property

    Public Property Get Font() As Font
      Set Font = m_Font
    End Property

Public Property Set Font(ByVal New_Font As Font)
  With m_Font
    .Bold = New_Font.Bold
    .Charset = New_Font.Charset
    .Italic = New_Font.Italic
    .Name = New_Font.Name
    .Size = New_Font.Size
    .Strikethrough = New_Font.Strikethrough
    .Underline = New_Font.Underline
    .Weight = New_Font.Weight
  End With
  Set UserControl.Font = m_Font
  PropertyChanged "Font"
  CalculateValues
  lbInvalidateRange
  Refresh
End Property
```

Das Ereignis `UserControl_Initialize` initialisiert die internen Listenfeldvariablen mit ihren Standardwerten. Die Listenfeldfarben werden zu diesem Zeitpunkt abhängig von den Standard-Systemfarben geladen. Sie können gegebenenfalls neue Eigenschaften hinzufügen, um diese Werte zu überschreiben.

Die Benutzer-Ereignisse, die die Operation des Listenfelds steuern, rufen mehrere private Funktionen mit dem Präfix `LB` auf. Diese Namengebung wurde gewählt, um die Lesbarkeit und die Modularität des Codes zu verbessern. Einige Ereignisse, beispielsweise Resize, machen es erforderlich, daß bestimmte Listenfeldvariablen neu berechnet werden. Eine der wichtigsten Aufgaben, die Sie bei jedem vom Benutzer gezeichneten Listenfeld haben, ist zu prüfen, welchen Einfluß jede Variable auf das Erscheinungsbild des Steuerelements hat, und sie korrekt zu verarbeiten. Dieses Steuerelement demonstriert einen guten Ansatz für die Verarbeitung, wobei die Berechnungen in einigen Routinen isoliert sind, die bei Bedarf aufgerufen werden.

```
'-----------------------------------------------------
'
'  UserControl - Methoden und Ereignisse
'


Private Sub UserControl_Initialize()
   set m_Font = new StdFont
   CurrentTop = 0
   LowDrawRange = -1
   HighDrawRange = -1
   m_BorderStyle = dw3d
   UserControl.BackColor = &HFFFFFF
   ReDim Selected(0)
   Selected(0) = -1
   FixNextScroll = -1
   PixelsPerLine = TextHeight("W") + 1
   highlight = GetSysColor(COLOR_HIGHLIGHT)
   HighlightText = GetSysColor(COLOR_HIGHLIGHTTEXT)
   Light3D = GetSysColor(COLOR_BTNHIGHLIGHT)
   Dark3D = GetSysColor(COLOR_BTNSHADOW)
   DarkShadow = GetSysColor(COLOR_3DDKSHADOW)
End Sub

Private Sub UserControl_Click()
   RaiseEvent Click
End Sub

Private Sub UserControl_DblClick()
   RaiseEvent DblClick
End Sub

Private Sub UserControl_MouseDown(Button As Integer, Shift As Integer, x _
As Single, Y As Single)
    LBMouseDown x, Y
End Sub

Private Sub UserControl_MouseMove(Button As Integer, Shift As Integer, x _
As Single, Y As Single)
    LBMouseMove Button, x, Y
End Sub

Private Sub UserControl_MouseUp(Button As Integer, Shift As Integer, x _
As Single, Y As Single)
    LBMouseUp
End Sub
```

```
    Private Sub UserControl_Paint()
        LBDraw
    End Sub

    Private Sub UserControl_Resize()
      CalculateValues
      PositionScrollBar
      lbInvalidateRange
    End Sub

    Private Sub UserControl_EnterFocus()
      LBGotFocus
    End Sub

    Private Sub UserControl_ExitFocus()
      LBLostFocus
    End Sub

Private Sub m_Font_FontChanged(ByVal PropertyName As String)
  Set UserControl.Font = m_Font
  CalculateValues
  lbInvalidateRange
  Refresh
End Sub


    Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
      m_BorderStyle = PropBag.ReadProperty("BorderStyle", dw3d)
      m_MultiSelect = PropBag.ReadProperty("MultiSelect", False)
      UserControl.BackColor = PropBag.ReadProperty("BackColor", &HFFFFFF)
      Set Font = PropBag.ReadProperty("Font", Ambient.Font)
      CalculateValues
    End Sub

    Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
      Call PropBag.WriteProperty("BorderStyle", m_BorderStyle, dw3d)
      Call PropBag.WriteProperty("MultiSelect", m_MultiSelect, 0)
      Call PropBag.WriteProperty("BackColor", UserControl.BackColor, &HFFFFFF)
      Call PropBag.WriteProperty("Font", m_Font, Ambient.Font)
    End Sub

    Private Sub UserControl_InitProperties()
      m_BorderStyle = dw3d
    End Sub
```

Das `ListCtlB`-Steuerelement verwendet eine Standard-Bildlaufleiste, um ein Scrolling für das Listenfeld zu unterstützen. Die Perspektive, eine benutzerdefinierte Bildlaufleiste im Steuerelementfenster zu erzeugen, war zu deprimierend,

um sie zu realisieren. Dieser Ansatz hat einen großen Nachteil. Wenn Sie ein sichtbares Standardelement in einem in Visual Basic geschriebenen Steuerelement verwenden, kann dieses Steuerelement selbst den Fokus nicht erhalten. Der Fokus geht dann immer an das Standard-Steuerelement. Das bedeutet, alle Tastencode-Ereignisse gelangen zum Bildlaufleisten-Element, statt zum `UserControl`-Objekt. Außerdem bedeutet es, daß die Bildlaufleiste mit dem Fokus erscheint, wenn Ihr Steuerelement den Fokus hat, was nicht dem Erscheinungsbild eines Standard-Listenfelds entspricht. Das ist unvermeidbar in der hier verwendeten Implementierung, aber im nächsten Kapitel werden Sie eine einfache Lösung dafür kennenlernen.

```
'-------------------------------------------------------
    '
    '  VScroll1: Methoden und Ereignisse
    '

    Private Sub VScroll1_Change()
      If FixNextScroll >= 0 Then
        If VScroll1.Value <> FixNextScroll Then
          VScroll1.Value = FixNextScroll
          Exit Sub
        End If
        FixNextScroll = -1
        Exit Sub
      End If
      Debug.Print "Change"
      LBScrollChange
    End Sub

    Private Sub VScroll1_KeyDown(KeyCode As Integer, Shift As Integer)
      FixNextScroll = VScroll1.Value
      LBArrow KeyCode
    End Sub

    Private Sub VScroll1_Scroll()
      Debug.Print "Scroll"
      LBScroll
    End Sub

    Private Sub VScroll1_KeyPress(KeyAscii As Integer)
      If KeyAscii = vbKeySpace And m_MultiSelect Then
        LBSelect ClickLine
        lbInvalidateRange ClickLine, ClickLine
        Refresh
      End If
    End Sub
```

Das benutzerdefinierte Listen-Steuerelement wird unter Verwendung der folgenden Funktionen implementiert. Ich gebe zu, daß dieser Code etwas kompliziert aussieht. Und ich hätte bestimmt den größten Teil des Kapitels mit seiner Beschreibung füllen können, aber weder Zeit noch Platz haben das erlaubt. Ich schlage vor, Sie nehmen das Ganze so in Angriff wie jeden anderen komplizierten Codeabschnitt. Versuchen Sie, der Logik zu folgen, und durchlaufen Sie den Code zur Laufzeit im Einzelschrittmodus:

```
'-----------------------------------------------------
'
'  Funktionen, die das benutzerdefinierte Listen-Steuerelement implementieren
'
'
' Für das Keydown-Ereignis im Steuerelement aufrufen
'
Private Sub LBArrow(keyid%)
   Dim newClickLine&
   Dim NewScrollValue&
   Select Case keyid%
      Case KEYDOWN
         newClickLine = ClickLine + 1
      Case KEYUP
         newClickLine = ClickLine - 1
      Case Else
         Exit Sub
   End Select
   If newClickLine < 0 Or newClickLine > VScroll1.Max Then
      ' Keine Änderung; vom Container angefordertes Neuzeichnen unterdrücken
      Exit Sub
   End If
   ClickLine = newClickLine
   HasFocus = ClickLine
   If Not m_MultiSelect Then LBSelect ClickLine

   If ClickLine < CurrentTop Then
      FixNextScroll = -1
      NewScrollValue = VScroll1.Value - 1
      VScroll1.Value = NewScrollValue
      FixNextScroll = NewScrollValue
      Exit Sub
   End If

   If ClickLine > CurrentTop + TotalLines - 1 Then
      FixNextScroll = -1
      NewScrollValue = VScroll1.Value + 1
      VScroll1.Value = NewScrollValue
      FixNextScroll = NewScrollValue
```

```
        Exit Sub
      End If
      LBDraw
    End Sub
```

Jede Zeile erhält den zu zeichnenden Test über das GetText-Ereignis. Es berechnet den genauen Bereich des Fensters, der von der Zeile bedeckt wird, wobei der aktuelle Rahmen und die Position der Bildlaufleiste berücksichtigt werden. Wenn die Zeile nicht sichtbar ist, muß sie auch nicht gezeichnet werden:

```
' Angegebene Zeilennummer in das Listenfeld schreiben
    '
    Private Sub LBDrawLine(tloc&)
      Dim rc As RECT
      Dim rc2 As RECT
      Dim di&
      Dim oldcolor&
      Dim txt$
      Dim usefont As IFont
      Dim oldfont As Long
      If tloc < CurrentTop Or tloc > CurrentTop + TotalLines Then
        ' Nicht sichtbar
        Exit Sub
      End If
      If Ambient.UserMode Then
        RaiseEvent GetText(tloc, txt$)
      Else
        If tloc = 0 Then txt = Extender.Name
      End If

      rc.left = m_BorderStyle + 1
      rc.right = ScaleWidth - VScroll1.Width - m_BorderStyle * 2
      rc.top = (tloc - CurrentTop) * PixelsPerLine + m_BorderStyle
      rc.bottom = rc.top + PixelsPerLine - 1
      Line (m_BorderStyle, rc.top + m_BorderStyle)-(ScaleWidth - _
      VScroll1.Width, rc.bottom + m_BorderStyle), BackColor, BF
      If IsSelected(tloc) Then
        If VScroll1.hwnd = GetFocus() And HasFocus = tloc Then   _
        ' Diese Zeile hat den Fokus
          LSet rc2 = rc
          rc2.left = m_BorderStyle
          rc2.right = ScaleWidth - VScroll1.Width - m_BorderStyle
          di = DrawFocusRect(hdc, rc2)
          di = InflateRect(rc2, -1, -1)
          Line (rc2.left, rc2.top)-(rc2.right - 1, rc2.bottom - 1), _
          highlight, BF
        Else
          Line (m_BorderStyle, rc.top)-(ScaleWidth - VScroll1.Width - _
```

```
            m_BorderStyle, rc.bottom), highlight, BF
         End If
         oldcolor = SetTextColor(hdc, HighlightText)

      Else
         If VScroll1.hwnd = GetFocus() And HasFocus = tloc Then
            ' Diese Zeile hat den Fokus
            LSet rc2 = rc
            rc2.left = m_BorderStyle
            rc2.right = ScaleWidth - m_BorderStyle
            di = DrawFocusRect(hdc, rc2)
            di = InflateRect(rc2, -1, -1)
            Line (rc2.left, rc2.top)-(rc2.right - 1, rc2.bottom - 1), _
            BackColor, BF
         Else
            Line (m_BorderStyle, rc.top)-(ScaleWidth - m_BorderStyle, _
            rc.bottom), BackColor, BF
         End If
      End If

      'Set usefont = UserControl.Font
      'oldfont = SelectObject(hdc, usefont.hFont)
      di = DrawText(hdc, txt, Len(txt), rc, DT_LEFT Or DT_SINGLELINE Or _
                  DT_NOPREFIX)
      'Call SelectObject(hdc, oldfont)
      If IsSelected(tloc) Then
         oldcolor = SetTextColor(hdc, oldcolor)
      End If

   End Sub
```

Es wäre sehr ineffizient, würde das Steuerelement bei jeder Änderung jeden Eintrag im Listenfeld anzeigen. Wird die Liste beispielsweise um ein Element nach oben oder unten geblättert, sollte es möglich sein, den Fensterinhalt nur zu verschieben und nur die neue Zeile zu zeichnen. Diese Implementierung unterstützt diese Funktion. Die Funktion lbInvalidateRange markiert, welche Einträge in dem Listenfeld wirklich gezeichnet werden müssen:

```
'
   ' Einen Bereich mit Einträgen unterdrücken
   '
   Private Sub lbInvalidateRange(Optional ByVal lowval& = 0, Optional _
   ByVal highval& = &H7FFFFFFF)
      Dim tval&
      Dim highest&
      If LowDrawRange = -1 And HighDrawRange = -1 Then
         LowDrawRange = &H7FFFFFFF
      End If
```

```
    If highval < lowval Then
       ' Ggf. vertauschen, um die Reihenfolge im Bereich beizubehalten
       tval = lowval
       lowval = highval
       highval = tval
    End If
    If lowval < LowDrawRange Then LowDrawRange = lowval
    If highval > HighDrawRange Then HighDrawRange = highval
    If LowDrawRange < CurrentTop Then LowDrawRange = CurrentTop
    highest = CurrentTop + TotalLines
    If HighDrawRange > highest Then HighDrawRange = highest

End Sub
```

Wenn ein Eintrag angeklickt wird, muß er möglicherweise neu gezeichnet werden, wenn sich der Auswahlstatus ändert. Das erfolgt in der hier gezeigten `LBMouseDown`-Funktion. Beachten Sie, wie sie auch den Eintrag mit dem Fokus verschiebt. Die Funktion nimmt außerdem Mauseingaben entgegen, was später mit der Funktion `LBMouseMove` noch beschrieben wird.

```
'
    ' Durch das MouseDown-Ereignis für das Steuerelement aufgerufen
    '
    Private Sub LBMouseDown(ByVal x&, ByVal Y&)
      Dim newClickLine&
      Dim li&
      li = SetCapture(hwnd)
      newClickLine = (Y - m_BorderStyle) \ PixelsPerLine + CurrentTop
      ' Listenfeld hat sich nicht geändert
      If HasFocus <> newClickLine Then
        HasFocus = newClickLine
        lbInvalidateRange HasFocus, HasFocus
      End If
      If IsSelected(newClickLine) And Not m_MultiSelect Then
         Exit Sub
      End If
      ClickLine = newClickLine
      LBSelect ClickLine
      LBDraw
    End Sub
```

Die Funktion `LBScroll` verarbeitet die Scrolling-Operation, die bereits beschrieben wurde. Sie berechnet, wie weit die Liste geblättert werden soll, und ruft die API-Funktion `ScrollWindow` auf, um den Inhalt des Fensters um den richtigen Betrag zu verschieben. Anschließend verwendet sie die Funktion `lbInvalidateRange`, um die Zeilen zu markieren, die ganz neu berechnet werden müssen. Dieser Ansatz erlaubt ein sehr schnelles und glattes Blättern:

```
'
    ' Aufruf durch das Scroll-Ereignis der vertikalen Bildlaufleiste
    '
    Private Sub LBScroll()
      Dim howmuch&
      Dim newtop&
      Dim dl&
      Dim rc As RECT
      rc.left = m_BorderStyle
      rc.right = ScaleWidth - VScroll1.Width - m_BorderStyle
      rc.top = m_BorderStyle
      rc.bottom = ScaleHeight - m_BorderStyle
      newtop& = VScroll1.Value
      howmuch& = CurrentTop - newtop&
      If howmuch& = 0 Then
        Exit Sub
      End If
      If Abs(howmuch) >= TotalLines Then
        ' Set to redraw it all
        LowDrawRange = -1
        HighDrawRange = -1
        CurrentTop = newtop
        LBDraw
        Exit Sub
      End If
      CurrentTop = newtop
      dl& = ScrollWindowByNum(hwnd, 0, howmuch& * PixelsPerLine, rc, 0)
      ' Jetzt ungültig machen
      If howmuch < 0 Then ' Wir haben nach oben gescrollt
          lbInvalidateRange newtop + TotalLines + howmuch, newtop + TotalLines
      Else
          lbInvalidateRange newtop, newtop + howmuch
      End If
      LBDraw

    End Sub
```

Die Auswahl-Operation ist relativ einfach. Im Einfachauswahl-Modus gibt der erste Eintrag im ausgewählten Feld den aktuell ausgewählten Eintrag an. Im Mehrfachauswahl-Modus hat das `Selected`-Feld eine dynamische Größe. Mit der Funktion `SetSelection` wird eine Liste der Zeilen angelegt, die aktuell selektiert sind:

```
'
    ' Auswahl des angegebenen Eintrags
    '
    Private Sub LBSelect(ByVal tloc&)
      If m_MultiSelect Then
```

```
        SetSelection tloc, Not IsSelected(tloc)
        lbInvalidateRange
      Else
        lbInvalidateRange Selected(0), tloc
        Selected(0) = tloc
      End If
    End Sub
```

Die Funktion `LBDraw` zeichnet die Zeilen, die aktuell als ungültig markiert sind. Am interessantesten ist jedoch die Implementierung des Rahmens. Wie Sie sehen, ist es ganz einfach, mit normalem VB-Code 3D-Effekte zu implementieren. Beachten Sie das beim nächsten Mal, wenn Sie ein ActiveX-Steuerelement einsetzen wollen, um ein 3D-Erscheinungsbild zu realisieren. Der Code-Ansatz ist viel effizienter in Hinblick auf Speicher und Ressourcen, und seine Performance ist ausgezeichnet:

```
Private Sub LBDraw()
  Dim startline&, lastline&
  Dim x&
  Dim sw As Long
  Dim sh As Long
  sw = ScaleWidth
  sh = ScaleHeight

  If LowDrawRange = -1 And HighDrawRange = -1 Then
       startline& = CurrentTop
       lastline& = TotalLines + CurrentTop
  Else
       startline& = LowDrawRange
       lastline& = HighDrawRange
  End If
  For x& = startline& - 1 To lastline& + 1
    LBDrawLine x
  Next x&
  LowDrawRange = -1
  HighDrawRange = -1

  Select Case m_BorderStyle
    Case dw3d
      UserControl.Line (0, 0)-(sw, 0), Dark3D
      UserControl.Line (1, 1)-(sw, 1), DarkShadow
      UserControl.Line (0, 0)-(0, sh), Dark3D
      UserControl.Line (1, 1)-(1, sh), DarkShadow
      UserControl.Line (sw - 1, 0)-(sw - 1, sh), Light3D
      UserControl.Line (sw - 1, sh - 1)-(1, sh - 1), Light3D
      UserControl.Line (sw - 2, sh - 2)-(2, sh - 2), Dark3D
    Case dwSingle
      UserControl.Line (0, 0)-(sw, 0), 0
```

```
            UserControl.Line (0, 0)-(0, sh), 0
            UserControl.Line (sw - 1, 0)-(sw - 1, sh), 0
            UserControl.Line (sw - 1, sh - 1)-(1, sh - 1), 0
        End Select
    End Sub
```

Was passiert, wenn Sie auf einen Eintrag in einem Listenfeld klicken und die Maus außerhalb des unteren oder oberen Rahmens des Listenfelds ziehen? Das Listenfeld wird weitergeblättert. `LBMouseMove` verwendet dazu einen API-Trick. Wenn ein `MouseMove`-Ereignis von einer Koordinate außerhalb des Listenfelds auf auftritt, prüft es, ob die Position oberhalb oder unterhalb des Steuerelements liegt. Ist das der Fall, blättert der Code das Listenfeld zuerst. Anschließend führt er eine `DoEvents`-Operation aus, um die Aktualisierung zu erlauben. Eine Kontextvariable auf Modulebene, `incontext`, soll zu diesem Zeitpunkt reentrante Ereignisse verhindern. Alle anderen `MouseMove`-Ereignisse werden dann zu diesem Zeitpunkt ignoriert. Als nächstes verwendet die Funktion `GetCursorPos` und `SetCursorPos`, um die Mauszeigerposition auf die aktuelle Position zu setzen. Damit wird die Maus nicht bewegt, sondern ein weiteres `MouseMove`-Ereignis erzeugt, das bewirkt, daß die Scrolling-Operation wiederholt wird.

Wie kann ein `MouseMove`-Ereignis von einem Punkt außerhalb des Steuerelements aus auftreten?

In der Funktion `LBMouseDown` wurde die API-Funktion `SetCapture` aufgerufen, um die Mauseingaben aufzufangen. Unter Win32 bleiben die Mauseingaben solange gültig, wie die Maustaste gedrückt ist, was für diese Applikation perfekt geeignet ist. Die Funktion `LBMouseUp` gibt die aufgefangenen Eingaben unter Verwendung der API-Funktion `ReleaseCapture` frei:

```
'
    ' Für das MouseMove-Ereignis des Steuerelements aufrufen
    '
    Private Sub LBMouseMove(ByVal Button%, ByVal x&, ByVal Y&)
       Dim pt As POINTAPI
       Dim dl&
       If InContext% Then Exit Sub
       If (Button And 1) = 0 Then Exit Sub
       If Y < 0 Or Y > ScaleHeight Then ' Wir scrollen
           InContext% = True
           If Y < 0 Then
              If VScroll1.Value > 0 Then VScroll1.Value = VScroll1.Value - 1
           Else
              If VScroll1.Value < VScroll1.Max Then VScroll1.Value = _
              VScroll1.Value + 1
           End If
           DoEvents
           ' Weiteres Maus-Ereignis erzwingen, so daß erneut geblättert wird
           InContext% = False
```

```
        dl& = GetCursorPos(pt)
        dl& = SetCursorPos(pt.x, pt.Y)
        Exit Sub
    End If
    If Not m_MultiSelect Then LBMouseDown x, Y
End Sub

Private Sub LBMouseUp()
    Dim dl&
    dl& = ReleaseCapture()
End Sub

Private Property Get LBClickLine()
    LBClickLine = ClickLine
End Property


'
' Wird für das GotFocus-Ereignis aufgerufen
'
Private Sub LBGotFocus()
  If m_MultiSelect Then
     lbInvalidateRange
  Else
     lbInvalidateRange Selected(0), Selected(0)
  End If
  LBDraw
End Sub

'
' Wird für das Change-Ereignis der Bildlaufleiste aufgerufen
'
Private Sub LBScrollChange()
    LBScroll
End Sub

'
' Wird für das LostFocus-Ereignis aufgerufen
'
Private Sub LBLostFocus()
  If m_MultiSelect Then
     lbInvalidateRange
  Else
     lbInvalidateRange Selected(0), Selected(0)
  End If
  LBDraw
End Sub
```

```
Private Sub PositionScrollBar()
  Dim slidescroll%
  If m_BorderStyle = dw3d Then slidescroll = 1
  VScroll1.Move ScaleWidth - (VScroll1.Width + slidescroll), _
  slidescroll, VScroll1.Width, ScaleHeight - m_BorderStyle
End Sub

Private Function IsSelected(ByVal location&) As Boolean
  Dim x&
  For x = 0 To UBound(Selected)
    If Selected(x) = location Then
      IsSelected = True
      Exit Function
    End If
  Next x
End Function
```

Die Routine `SetSelection` verwaltet das Auswahlfeld. Das ist eine der schnell hingeschriebenen Routinen, die nicht unbedingt die effizienteste Lösung für diese Aufgabe darstellen, wie bereits oben erwähnt. Aber so sollte es gehen:

```
Private Sub SetSelection(ByVal location&, ByVal newstate As Boolean)
    Dim x&
    Dim firstfree&
    firstfree = -1
    For x = 0 To UBound(Selected)
      If Selected(x) = location Then
        If Not newstate Then Selected(x) = -1
        Exit Sub
      End If
      If firstfree < 0 And Selected(x) = -1 Then firstfree = x
    Next x
    If newstate Then
      If firstfree >= 0 Then
        Selected(firstfree) = location
        Exit Sub
      End If
      ReDim Preserve Selected(UBound(Selected) + 1)
      Selected(UBound(Selected)) = location
    End If

End Sub


Private Sub CalculateValues()
  PixelsPerLine = UserControl.TextHeight("W") + 1
  TotalLines = (ScaleHeight - m_BorderStyle * 2) \ PixelsPerLine
End Sub
```

Wie Sie sehen, ist für den benutzerdefinierten Ansatz wesentlich mehr Aufwand erforderlich, aber Sie erhalten damit gleichzeitig eine enorme Flexibilität in dem Prozeß. Bevor wir diesen Ansatz verlassen, betrachten wir noch ein Beispiel, das das Fokusproblem löst, das es bei dieser Implementierung gibt.

22.3.3 Ein besseres benutzerdefiniertes Steuerelement

Das Beispiel `ListCtlC.ctl` behebt das Fokusproblem, das entsteht, wenn man eine sichtbare Standard-Elementkomponente verwendet. Es stellt sich heraus, daß Windows eine eingebaute Bildlaufleiste unterstützt, die in fast jedem Fenster eingesetzt werden kann. Sie können diese eingebauten Bildlaufleisten auch für das UserControl-Fenster aktivieren. Das zählt nicht als sichtbare Standard-Elementkomponente. Damit erhält also das Steuerelement selbst den Fokus.

Es weist sich, daß die Aktivierung von Bildlaufleisten und die Verwaltung von eingehenden Nachrichten ein bißchen Aufwand bedeutet, aber die Komponente `dwspyvb.dll` beinhaltet ein Objekt, das Ihnen die Arbeit abnimmt: das `dwScrollBars`-Objekt. Es kann für viele verschiedene Fenstertypen eingesetzt werden, unter anderem auch für Fenster für ActiveX-Steuerelemente, Formulare und Bildfeld-Steuerelemente. Das Objekt wird wie folgt deklariert und initialisiert:

```
Private WithEvents VScroll As dwScrollBars
```

Das Objekt wird im UserControl_Initialize-Ereignis initialisiert:

```
Set VScroll = New dwScrollBars
```

Die vertikale Bildlaufleiste wird während des `UserControl_Show`-Ereignisses aktiviert, wie hier gezeigt:

```
Private Sub UserControl_Show()
  If Ambient.UserMode Then
    VScroll.hwnd = UserControl.hwnd
    VScroll.ScrollBars = sbeVerticalScrollbar
  End If
End Sub
```

Es ist nicht erforderlich, die Bildlaufleiste zur Entwurfszeit anzuzeigen.

Das Steuerelement selbst erhält jetzt Tastatureingaben, die wie folgt verarbeitet werden:

```
Private Sub UserControl_KeyDown(KeyCode As Integer, Shift As Integer)
  FixNextScroll = VScroll.VValue
  LBArrow KeyCode
End Sub

Private Sub UserControl_KeyPress(KeyAscii As Integer)
  If KeyAscii = vbKeySpace And m_MultiSelect Then
    LBSelect ClickLine
    lbInvalidateRange ClickLine, ClickLine
```

```
      Refresh
    End If
End Sub
```

Die Eigenschaften und Ereignisse des `VScroll`-Objekts sind ähnlich denen des Standardbildlaufleistenelements. Der einzige Unterschied ist, daß jedem Ereignis- oder Eigenschaftsnamen ein H oder V vorausgehen, woran man erkennt, ob sie zu der vertikalen oder zu der horizontalen Bildlaufleiste gehören. Dieses Steuerelement verwendet nur die vertikale Bildlaufleiste, aber das Objekt unterstützt beide, entweder einzeln oder gleichzeitig.

```
'-----------------------------------------------------
'
'  VScroll1': Methoden und Ereignisse
Private Sub VScroll_VChange()
   If FixNextScroll >= 0 Then
      If VScroll.VValue <> FixNextScroll Then
         VScroll.VValue = FixNextScroll
         Exit Sub
      End If
      FixNextScroll = -1
      Exit Sub
   End If
   Debug.Print "Change"
   LBScrollChange
End Sub

Private Sub VScroll_VScroll()
   LBScroll
End Sub
```

Damit beschließen wir unsere Beschreibung der benutzerdefinierten Steuerelemente und kommen zu einem weiteren Ansatz: dem vierten Modell zur Entwicklung von Steuerelementen, das fensterbasierte Steuerelement.

22.3.4 Ein angepaßtes fensterbasiertes Steuerelement

Die größte Einschränkung eines aus Standardelementen zusammengesetzten Steuerelements ist, daß Sie die Entwurfseigenschaften der betreffenden Elemente nicht ändern können. Diese Eigenschaften basieren nämlich auf Fensterstilen, die beim Anlegen des Fensters definiert werden müssen. Visual Basic bietet keinen Mechanismus, um ein Steuerelement zu zerstören und neu anzulegen. Na gut, eigentlich schon – wenn Sie das Steuerelement von Anfang an dynamisch anlegen. Aber Steuerelemente, die dynamisch angelegt werden, werden zur Laufzeit angelegt, so daß Sie ihre Entwurfs-Eigenschaften nicht festlegen können. Das dynamische Anlegen von Steuerelementen hilft Ihnen in diesem Kontext also nicht weiter. Wenn Sie dagegen Ihr eigenes Fenster von Grund auf neu erzeugen würden, könnten Sie das Fenster beliebig zerstören und neu erstellen, und zwar

sowohl zur Entwurfszeit als auch zur Laufzeit des Containers. Damit müßte Ihr Steuerelement natürlich mehrere Verantwortlichkeiten übernehmen:

- Sie müßten eingehende Nachrichten an das Fenster weitergeben.
- Sie müßten Nachrichten auffangen, die das Fenster an sein Elternfenster schickt (Ihr UserControl-Objekt).
- Die gesamte Kommunikation mit dem Fenster müßte über Nachrichten erfolgen.

Das ist gar nicht so schwer, wie es sich anhört, wie Sie gleich sehen werden.

Das Beispiel-Steuerelement `ListCtlD` erzeugt ein Standard-Listenfeldfenster und plaziert es über dem UserControl-Fenster für Ihr Steuerelement. Das Beispiel beginnt mit mehreren Standard-API-Deklarationen.

Die Konstanten mit den Präfixen `WS` und `LBS` beschreiben die Fensterstile. Die Bedeutung der meisten dieser Fensterstile kann von ihrem Namen abgeleitet werden. Detaillierte Erklärungen finden Sie in jedem guten API-Nachschlagewerk. Nur ein paar der hier gezeigten Stile werden in diesem Steuerelement wirklich verwendet:

```
' Listenfeld-Beispiel CreateWindow
    ' Copyright (c) 1997-1998, by Desaware Inc. All Rights Reserved

    Option Explicit

    Private Const WS_CHILD = &H40000000
    Private Const WS_VISIBLE = &H10000000
    Private Const WS_CLIPSIBLINGS = &H4000000
    Private Const WS_BORDER = &H800000
    Private Const WS_GROUP = &H20000
    Private Const WS_TABSTOP = &H10000
    Private Const WS_VSCROLL = &H200000
    Private Const WS_EX_APPWINDOW = &H40000
    Private Const WS_EX_CLIENTEDGE = &H200&
    Private Const WS_EX_CONTEXTHELP = &H400&
    Private Const WS_EX_CONTROLPARENT = &H10000
    Private Const WS_EX_LEFT = &H0&
    Private Const WS_EX_LEFTSCROLLBAR = &H4000&
    Private Const WS_EX_LTRREADING = &H0&
    Private Const WS_EX_MDICHILD = &H40&
    Private Const WS_EX_RIGHT = &H1000&
    Private Const WS_EX_RIGHTSCROLLBAR = &H0&
    Private Const WS_EX_RTLREADING = &H2000&
    Private Const WS_EX_STATICEDGE = &H20000
    Private Const WS_EX_TOOLWINDOW = &H80&
    Private Const WS_EX_WINDOWEDGE = &H100&
```

```
' Listenfeld-Stile
Private Const LBS_NOTIFY = &H1&
Private Const LBS_SORT = &H2&
Private Const LBS_NOREDRAW = &H4&
Private Const LBS_MULTIPLESEL = &H8&
Private Const LBS_OWNERDRAWFIXED = &H10&
Private Const LBS_OWNERDRAWVARIABLE = &H20&
Private Const LBS_HASSTRINGS = &H40&
Private Const LBS_USETABSTOPS = &H80&
Private Const LBS_NOINTEGRALHEIGHT = &H100&
Private Const LBS_MULTICOLUMN = &H200&
Private Const LBS_WANTKEYBOARDINPUT = &H400&
Private Const LBS_EXTENDEDSEL = &H800&
Private Const LBS_DISABLENOSCROLL = &H1000&
Private Const LBS_NODATA = &H2000&
Private Const LBS_NOSEL = &H4000&
Private Const LBS_STANDARD = (LBS_NOTIFY Or LBS_SORT Or WS_VSCROLL _
Or WS_BORDER)
```

Benachrichtigungscodes sind WM_COMMAND-Nachrichtentypen, die vom Fenster an sein Elternfenster geschickt werden, in diesem Fall das UserControl-Fenster:

```
' Benachrichtigungscodes für das Listenfeld
Private Const LBN_ERRSPACE = (-2)
Private Const LBN_SELCHANGE = 1
Private Const LBN_DBLCLK = 2
Private Const LBN_SELCANCEL = 3
Private Const LBN_SETFOCUS = 4
Private Const LBN_KILLFOCUS = 5
```

Die Nachrichten mit dem Präfix LB_ werden genutzt, um die Operation des Listenfeldfensters zu steuern. Nur ein paar der hier gezeigten Befehle werden in diesem Beispiel auch implementiert:

```
' Listenfeld-Nachrichten
Private Const LB_ADDSTRING = &H180
Private Const LB_INSERTSTRING = &H181
Private Const LB_DELETESTRING = &H182
Private Const LB_SELITEMRANGEEX = &H183
Private Const LB_RESETCONTENT = &H184
Private Const LB_SETSEL = &H185
Private Const LB_SETCURSEL = &H186
Private Const LB_GETSEL = &H187
Private Const LB_GETCURSEL = &H188
Private Const LB_GETTEXT = &H189
Private Const LB_GETTEXTLEN = &H18A
Private Const LB_GETCOUNT = &H18B
Private Const LB_SELECTSTRING = &H18C
Private Const LB_DIR = &H18D
```

```
Private Const LB_GETTOPINDEX = &H18E
Private Const LB_FINDSTRING = &H18F
Private Const LB_GETSELCOUNT = &H190
Private Const LB_GETSELITEMS = &H191
Private Const LB_SETTABSTOPS = &H192
Private Const LB_GETHORIZONTALEXTENT = &H193
Private Const LB_SETHORIZONTALEXTENT = &H194
Private Const LB_SETCOLUMNWIDTH = &H195
Private Const LB_ADDFILE = &H196
Private Const LB_SETTOPINDEX = &H197
Private Const LB_GETITEMRECT = &H198
Private Const LB_GETITEMDATA = &H199
Private Const LB_SETITEMDATA = &H19A
Private Const LB_SELITEMRANGE = &H19B
Private Const LB_SETANCHORINDEX = &H19C
Private Const LB_GETANCHORINDEX = &H19D
Private Const LB_SETCARETINDEX = &H19E
Private Const LB_GETCARETINDEX = &H19F
Private Const LB_SETITEMHEIGHT = &H1A0
Private Const LB_GETITEMHEIGHT = &H1A1
Private Const LB_FINDSTRINGEXACT = &H1A2
Private Const LB_SETLOCALE = &H1A5
Private Const LB_GETLOCALE = &H1A6
Private Const LB_SETCOUNT = &H1A7
Private Const LB_MSGMAX = &H1A8
Private Const WM_CTLCOLORLISTBOX = &H134


Private Const WM_LBUTTONDOWN = &H201
Private Const WM_LBUTTONUP = &H202
Private Const WM_SETFONT = &H30
Private Const WM_SETFONT = &H30
Private Const WM_SETFOCUS = &H7


Private Const WHITE_BRUSH = 0
Private Declare Function GetStockObject Lib "gdi32" (ByVal nIndex As Long) _
As Long

Event Click()
Event DblClick()
```

Dieses Beispiel verwendet zwei Objekte aus der Komponente `dwSpyvb.dll`. Die Klasse `dwPrivateWindow` wird genutzt, um das Anlegen und die Verwendung privater Fenster zu verwalten. Das Objekt `dwSubClass`, das Sie bereits kennengelernt haben, dient dazu, die Benachrichtigungen für das Listenfeldfenster aufzufangen.

```
    Dim WithEvents listwnd As dwPrivateWindow
    Dim WithEvents cmdhook As dwSubClass
    Dim WithEvents listhook As dwSubClass
    Dim WithEvents pretrans As dwPretranslate ' Tastatur-Umsetzung

    Private Const defClassName = "LISTBOX"

    Private m_MultiSelect As Boolean
    Private m_BorderStyle As Integer
    Private WithEvents m_Font As StdFont
```

Für die Änderung der Eigenschaften `MultiSelect` und `BorderStyle` ist es erforderlich, daß das Listenfeldfenster zerstört und neu angelegt wird, wie Sie in ihren Eigenschaftsprozeduren sehen:

```
Public Property Get MultiSelect() As Boolean
      MultiSelect = m_MultiSelect
    End Property

    Public Property Let MultiSelect(ByVal vNewValue As Boolean)
      m_MultiSelect = vNewValue
      PropertyChanged "MultiSelect"
      InitTheWindow  ' Das Erscheinungsbild ändern - Jetzt!
    End Property

    Public Property Get BorderStyle() As dwBorderStyle
      BorderStyle = m_BorderStyle
    End Property

    Public Property Let BorderStyle(ByVal vNewValue As dwBorderStyle)
      If vNewValue > 2 or vNewValue < 0 Then
         Err.Raise 380
      End If
      m_BorderStyle = vNewValue
      PropertyChanged "BorderStyle"
      InitTheWindow
    End Property

Public Property Get Font() As Font
  Set Font = m_Font
End Property

Public Property Set Font(ByVal New_Font As Font)
  With m_Font
     .Bold = New_Font.Bold
     .Charset = New_Font.Charset
     .Italic = New_Font.Italic
     .Name = New_Font.Name
```

```
      .Size = New_Font.Size
      .Strikethrough = New_Font.Strikethrough
      .Underline = New_Font.Underline
      .Weight = New_Font.Weight
   End With
   Set UserControl.Font = m_Font
   PropertyChanged "Font"
   UpdateTheFont
End Property

Private Sub m_Font_FontChanged(ByVal PropertyName As String)
   Set UserControl.Font = m_Font
End Sub
```

Wenn Ihr Steuerelement den Fokus erhält, müssen Sie ihn auf das enthaltene Listenfeldfenster setzen. Das ist erforderlich, weil ein privates Fenster nicht wirklich ein Standard-Element ist, so daß Visual Basic nicht weiß, daß es den Fokus erhalten soll, wenn das Steuerelement ihn erhält. Die Methode `SetFocus`, die Sie hier sehen, ist eine Methode des Objekts `dwPrivateWindow`, die den API-Befehl `SetFocus` aufruft:

```
   '-------------------------------------------
    '
    '    UserControl: Ereignisse und Eigenschaften
    '
    Private Sub UserControl_EnterFocus()
      listwnd.SetFocus
    End Sub


    Private Sub UserControl_Initialize()
      Set m_Font = New StdFont
      Set listwnd = New dwPrivateWindow
      Set cmdhook = New dwSubClass
      Set listhook = New dwSubClass
     m_BorderStyle = 1
    End Sub
```

Die `InitTheWindow`-Funktion wird aufgerufen, wenn das Steuerelement initialisiert oder geladen wird:

```
Private Sub UserControl_InitProperties()
       ' Kindfenster anlegen, nachdem die Eigenschaften gelesen wurden
       InitTheWindow
    End Sub

    Private Sub UserControl_ReadProperties(PropBag As PropertyBag)
      ' Kindfenster anlegen, nachdem die Eigenschaften gelesen wurden
      m_MultiSelect = PropBag.ReadProperty("MultiSelect", False)
```

```
    m_BorderStyle = PropBag.ReadProperty("BorderStyle", False)
    Set Font = PropBag.ReadProperty("Font", Ambient.Font)
    InitTheWindow
End Sub

Private Sub UserControl_WriteProperties(PropBag As PropertyBag)
    Call PropBag.WriteProperty("MultiSelect", m_MultiSelect, False)
    Call PropBag.WriteProperty("BorderStyle", m_BorderStyle, False)
    Call PropBag.WriteProperty("Font", m_Font, Ambient.Font)
End Sub
```

Die folgenden Aufräumarbeiten beim Beenden des Steuerelements sind eine Stilfrage. Die beiden Objekte `dwPrivateWindow` und `dwSubClass` wissen, wie sie aufräumen müssen, nachdem sie zerstört wurden:

```
Private Sub UserControl_Terminate()
    listwnd.DestroyWindow
    cmdhook.HwndParam = 0
    Set listwnd = Nothing
    If Not (pretrans Is Nothing) Then pretrans.hwnd = 0
    Set cmdhook = Nothing
End Sub
```

Es ist nicht möglich, direkt vom `Font`-Objekt einen Schrift-Handle zu erhalten, aber das `Form`-Objekt implementiert die Schnittstelle IFont, die ermöglicht, den Handle für die Schrift zu ermitteln. Dieser kann mit der Nachricht `WM_SETFONT` an das Listenfeldfenster geschickt werden:

```
'------------------------------------
'
'     Intern verwendete Funktionen
'


Private Sub UpdateTheFont()
    Dim usefont As IFont
    Set usefont = m_Font
    Call listwnd.SendMessageNumber(WM_SETFONT, usefont.hFont, 1)
End Sub
```

Die Funktion `GetListboxStyle` ist eine `Utility`-Funktion, die den korrekten Fensterstil für die vorgegebenen Mehrfachauswahl- und Rahmeneinstellungen ermittelt:

```
'Private Function GetListboxStyle() As Long
    Dim LS As Long
    LS = WS_CHILD Or WS_TABSTOP Or WS_CLIPSIBLINGS Or WS_VISIBLE Or WS_VSCROLL
    LS = LS Or LBS_NOTIFY Or LBS_HASSTRINGS
    If m_MultiSelect Then LS = LS Or LBS_MULTIPLESEL Or LBS_EXTENDEDSEL
    If m_BorderStyle Then LS = LS Or WS_BORDER
    GetListboxStyle = LS
```

```
End Function
```

Die Funktion `InitTheWindow` erzeugt das enthaltene Listenfeldfenster. Das `cmdhook`-Objekt erzeugt eine Unterklasse des UserControl-Fensters. Es wird so eingerichtet, daß es Steuernachrichten mit Hilfe des `CommandMessage`-Ereignisses ermittelt, wobei die Parameter der `WM_COMMAND`-Nachricht in ihre einzelnen Komponenten zerlegt werden. Außerdem wird es eingerichtet, um die `WM_CTLCOLORLISTBOX`-Nachricht zu erkennen, so daß Sie die Hintergrundfarbe des Listenfelds ermitteln können. Die `CreateWindowEx`-Methode von listwnd erzeugt das neue Fenster. Es ist nicht erforderlich, das vorhergehende Fenster zu zerstören – das passiert automatisch, wenn Sie das neue Fenster erzeugen. Das pretrans-Objekt wird genutzt, um Tastencode-Nachrichten zu erkennen, und sicherzustellen, daß die Tastencodes an das Steuerelementfenster geschickt werden. Das listhook-Objekt bildet eine Unterklasse des privaten Fensters, und beobachtet, wann es den Fokus erhält. Ich werde gleich dazu kommen, warum das so ist.

```
Private Sub InitTheWindow()
  cmdhook.EnableCommandEvent = True
  cmdhook.HwndParam = UserControl.hwnd
  cmdhook.AddMessage WM_CTLCOLORLISTBOX
  Call listwnd.CreateWindowEx(0, defClassName, "", GetListboxStyle, _
       0, 0, ScaleWidth, ScaleHeight, hwnd, 0)
  If Not Ambient.UserMode Then
       AddItem Ambient.DisplayName
 Else
       Set pretrans = New dwPretranslate
       pretrans.hwnd = listwnd.hwnd
       pretrans.KeyMessagesOnly = True
  End If
  listhook.HwndParam = listwnd.hwnd
  listhook.AddMessage WM_SETFOCUS
  listhook.SubclassType = dwPostedUpdateTheFont
End Sub
```

`WM_COMMAND`-Nachrichten von dem Listenfeldfenster werden auf ihrem Weg zum UserControl-Fenster abgefangen. Das UserControl-Fenster erhält `WM_COMMAND`-Nachrichten von vielen Quellen, auch von anderen Steuerelementen und Menüs. In diesem Beispiel gibt es kein anderes Fenster, die einzige Quelle für `WM_COMMAND`-Nachrichten ist also das Listenfeldfenster. Es ist aber dennoch sinnvoll, eine Prüfung vorzunehmen. Die Nachrichten werden direkt auf Control-Ereignisse abgebildet.

```
' Hier kommen Befehlsbenachrichtigungen an
Private Sub cmdhook_CommandMessage(ByVal hwnd As Long, ByVal wID As Long, _
ByVal wNotifyCode As Long, ByVal hwndCtl As Long, retval As Long, nodef _
As Boolean)
  If hwndCtl <> listwnd.hwnd Then Exit Sub  ' Sie ist nicht für das
                                            ' Listenfeld bestimmt
```

```
    Select Case wNotifyCode
       Case LBN_SELCHANGE
          RaiseEvent Click
       Case LBN_DBLCLK
          RaiseEvent DblClick
    End Select
    ' Standardergebnis für Benachrichtigungen
    nodef = True
    retval = 0
End Sub

Private Sub cmdhook_WndMessage(ByVal hwnd As Long, Msg As Long, _
wp As Long, lp As Long, retval As Long, nodef As Boolean)
    If lp <> listwnd.hwnd Then Exit Sub  ' It's not for the list box
    Select Case Msg
       Case WM_CTLCOLORLISTBOX
          retval = GetStockObject(WHITE_BRUSH)   ' Weißen Pinsel zurückgeben
          nodef = True   ' Standardblockverarbeitung
    End Select
End Sub
```

Die Funktion `cmdhook_WndMessage` fängt die `WM_CTLCOLORLISTBOX`-Nachricht auf und gibt einen Handle auf einen weißen Pinsel zurück, so daß die Hintergrundfarbe des Listenfelds weiß wird. Sie können diesen Wert natürlich auch abhängig von der `BackColor`-Eigenschaft setzen, falls Sie diese implementiert haben. Der Ereignisparameter `nodef` muß auf `True` gesetzt werden, um die Farbe auf die Standardhintergrundfarbe für Listenfelder zu setzen.

Das Einfügen und Entfernen eines Strings erfolgt einfach durch Senden der entsprechenden Nachricht:

```
' String in das Listenfeld einfügen
Public Sub AddItem(item As String, Optional Index As Long = -1)
    Dim res&
    If Index = -1 Then
       res = listwnd.SendMessageString(LB_ADDSTRING, 0, item)
       ' TODO - on fail raise error
    Else
       res = listwnd.SendMessageString(LB_INSERTSTRING, Index, item)
       ' TODO - on fail raise error
    End If
End Sub

' String aus dem Listenfeld entfernen
Public Sub RemoveItem(ByVal Index As Long)
    If Index < 0 Or Index > ListCount Then
       Err.Raise 380
    Else
```

```
        Call listwnd.SendMessageNumber(LB_DELETESTRING, Index, 0)
      End If
    End Sub

    Public Property Get ListCount() As Long
      ListCount = listwnd.SendMessageNumber(LB_GETCOUNT, 0, 0)
    End Property
```

Was ist mit den Nachrichten, die an das private Listenfeldfenster geschickt werden? Wenn nicht anders von Ihnen angegeben, werden sie automatisch durch das `listwnd`-Objekt an die Standardfensterfunktion geschickt – die Klassenfensterfunktion, die die Funktionalität des Listenfelds implementiert. Sie können die Verarbeitung dieser Nachrichten bei Bedarf überschreiben, aber in diesem Fall ist das nicht nötig. Schön an diesem Ansatz ist, daß wenn Sie Ihre Applikation unterbrechen, die Fensternachrichten weiterhin ihre Standardverarbeitung erfahren, womit die Windows-Nachrichtenverarbeitung auf normale Weise fortgesetzt wird. Das ist möglich, weil die Standardnachrichtenverarbeitung in das `dwPrivateWindow`-Objekt eingebaut ist. Sie können sie überschreiben, wenn Sie möchten, aber wenn Ihr Code unterbrochen wird, wird immer die Standardnachrichtenverarbeitung ausgeführt.

Das Listenfeld muß die Tastencodes für den Pfeil nach unten und nach oben empfangen können, um richtig zu arbeiten. Die einfachste Methode ist die Verwendung der Vorverarbeitung von Nachrichten, also diese Tastencodes aufzufangen und die Tastaturnachrichten mit Hilfe der `SendMessageNumber`-Methode des `listwnd`-Objekts direkt an das Listenfeldfenster zu senden, wie hier gezeigt:

```
Private Sub pretrans_PreTranslateMessage(ByVal hwnd As Long, Msg As Long, _
              wParam As Long, lParam As Long, nodef As Boolean)
  If wParam = VK_UP Or wParam = VK_DOWN Then
    Call listwnd.SendMessageNumber(Msg, wParam, lParam)
    nodef = True
    Msg = 0
  End If
End Sub
```

Es gibt noch eine Geschichte, um die wir uns kümmern müssen, und die mit dem Fokus zu tun hat. Dieses Listenfeld-Steuerelement besteht aus zwei Fenstern, dem UserControl-Fenster und dem privaten Listenfeldfenster. Wenn Sie mit der Tab-Taste auf das Steuerelement gehen, wird das `EnterFocus`-Ereignis aufgeworfen, das die SetFocus-Methode des `listwnd`-Objekts aufruft und den Fokus auf das private Listenfeldfenster setzt. Was passiert aber, wenn Sie den Fokus auf das Fenster setzen, indem Sie einfach auf das Fenster klicken? Windows übergibt den Fokus direkt dem privaten Fenster. Das `EnterFocus`-Ereignis wird nie aufgeworfen. Auf den ersten Blick scheint das keine Rolle zu spielen; schließlich erhält das private Fenster den Fokus. Dabei umgeht Visual Basic jedoch die normale Fokus-Verarbeitung und es entsteht ein Chaos in der Tab-Reihenfolge.

Die Lösung dafür ist, das WM_SETFOCUS-Ereignis mit Hilfe eines Unterklassenobjekts abzufragen, in diesem Fall mit dem listhook-Objekt. Der Parameter wp enthält den Handle des Fensters, das den Fokus verloren hat. Wenn es sich dabei um das UserControl-Fenster handelt, wissen Sie, daß die Fokusweitergabe durch den Aufruf von listwnd.SetFocus im EnterFocus-Ereignis passiert. Andernfalls ist es eine direkte Weitergabe. In diesem Fall setzen wir den Fokus explizit zurück auf das UserControl-Objekt. Visual Basic sieht eine normale Fokusweitergabe und aktualisiert seine Fokus-Reihenfolge korrekt. Das EnterFocus-Ereignis wird aufgeworfen, und der Fokus geht zurück an das private Fenster – und diesmal auf sichere Weise.

```
Private Sub listhook_WndMessage(ByVal hwnd As Long, Msg As Long, wp As Long, _
lp As Long, retval As Long, nodef As Boolean)
   Select Case Msg
      Case WM_SETFOCUS
         If wp <> UserControl.hwnd Then
            ' Wird aufgerufen, wenn Sie direkt auf das Listenfeld klicken,
            ' um zu verhindern, daß Visual Basic den Überblick verliert,
            ' wer gerade den Fokus hat
            Debug.Print "Special case"
            UserControl.SetFocus
         End If
      ' We can handle window messages here
   End Select
End Sub
```

Der hier gezeigte Ansatz kann noch praktischer werden, wenn die Technologie der benutzerdefinierten Listenfelder verwendet wird. In diesem Fall zeichnet nicht das Standard-Listenfeldfenster das Listenfeld. Statt dessen sendet es eine Nachricht an das Elternfenster und teilt ihm mit, welche Zeile ausgegeben werden soll, und ob sie selektiert ist oder den Fokus hat. Außerdem stellt es einen Gerätekontext bereit, der für das Zeichnen verwendet werden kann.

Sie denken vielleicht, Sie müßten Windows-Experte sein, um den Ansatz mit dem privaten Fenster wirklich nutzen zu können. Das stimmt auch irgendwo. Aber hier sollte er vor allem dazu dienen, meine Aussage vom Kapitelanfang zu bestärken: Sie können fast alle Techniken zur Entwicklung von Steuerelementen verwenden, die in anderen Sprachen möglich sind, wenn Sie unter Visual Basic Steuerelemente erstellen. Dazu müssen Sie jedoch diese Techniken auf derselben Ebene erlernen und verarbeiten – der Ebene von API und Windows-Nachrichtenverarbeitung. Visual Basic kann die einfachste Methode darstellen, komplexe Steuerelemente zu entwickeln, aber man kann nicht unbedingt behaupten, daß das einfach wäre.

Für diejenigen unter Ihnen, die sich für weitere Beispiele komplexer Steuerelemente interessieren, bietet Desaware das Produkt Desaware ActiveX Gallimaufry, das eine Vielzahl von Steuerelementen mit technischer Beschreibung und vollständigem Quellcode beinhaltet. Die Steuerelemente enthalten ein perspektivi-

sches Listen-Steuerelement, ein Hexadezimaleditor-Steuerelement, ein Spiralkunst-Steuerelement, ein Bitmap-Dreh-Steuerelement, ein MDI-Taskleisten-Steuerelement (fügt MDI-Applikationen eine Taskleiste hinzu), eine Standarddialogkomponente (bietet Standardarddialoge, die weit über die Funktionalität der normalen Standarddialog-Steuerelemente hinausgehen). Details finden Sie auf der CD-ROM zum Buch und auf unserer Web-Seite unter der Adresse `http://www.desaware.com`.

Damit beschließen wir die Beschreibung der Entwicklung von ActiveX-Steuerelementen unter Visual Basic. Wir kehren in Teil V noch einmal zu diesem Thema zurück, wo wir über verwandte Aspekte für die Versionsverwaltung und Lizenzierung sprechen. Bis dahin werden wir über einen ähnlichen Komponententyp sprechen: das ActiveX-Dokument.