Kapitel 2

ActiveX – eine historische (aber auch technische) Betrachtung

Ich verstehe ja, daß es durchaus sehr verführerisch ist, eine historische Betrachtung in einem Technik-Buch wie diesem zu überspringen, doch haben Sie bitte ein wenig Geduld mit mir. Denn in Sachen ActiveX kann das Wissen um die historischen Gegebenheiten der gradlinigste Weg zum Verständnis der Technologie sein.

2.1 Die anwendungszentrierte Umgebung

Versetzen Sie sich einmal in die graue Vorzeit zurück, sagen wir mal, in die Mitte der 80er Jahre. DOS war der King, und bei Windows 1.0 handelte es sich um eine lahme Grafik-Oberfläche, die sogar von den Diskettenlaufwerken eines 8086er Rechners betrieben werden konnte (Ok, das war nicht besonders toll, aber immerhin, es lief!). Jede Aufgabe, die auf einem PC erledigt wurde, war *anwendungszentriert*. Genauer gesagt, jede Anwendung arbeitete unabhängig voneinander und verwendete zumeist ein eigenes Datenformat. Abbildung 2.1 veranschaulicht diese Situation. Solange Sie es nur mit dem Datenformat der jeweiligen Anwendung zu tun hatten, war auch alles in Ordnung.

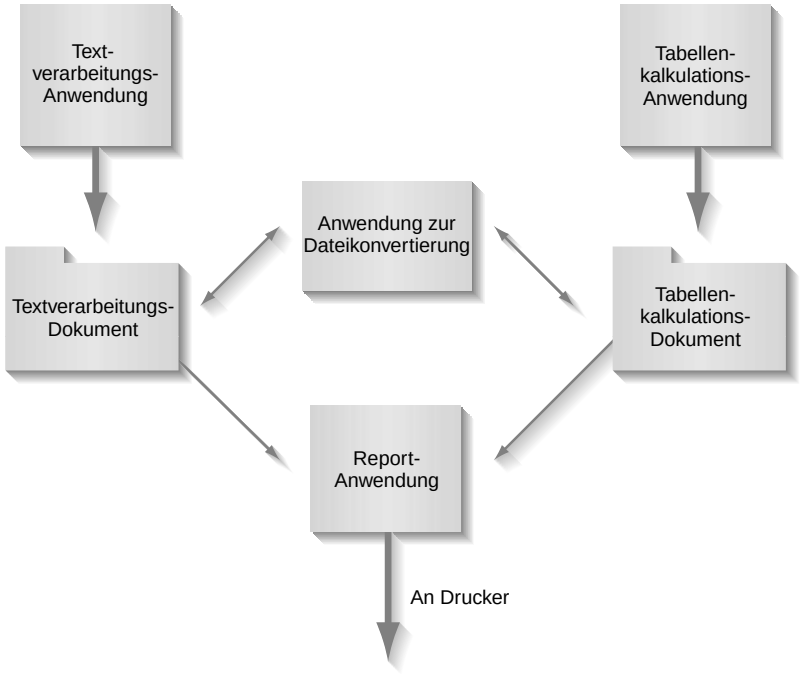

Abb. 2.1: Eine typische anwendungs-zentrierte Umgebung

Was aber war, wenn Sie Daten konvertieren wollten, um damit in einem anderen Programm zu arbeiten? Dann waren Sie auf spezielle Konvertier-Programme angewiesen, oder darauf, daß eine Anwendung eine Import- oder Export-Mög-

lichkeit anbot, um die erforderliche Konvertierung vorzunehmen. Falls Sie einen Bericht erstellen und darin zwei verschiedene Datenquellen – und damit Datentypen – kombinieren wollten, mußten Sie einen gesonderten Report-Generator einsetzen, oder Sie waren auf die Fähigkeiten einer Anwendung angewiesen, mit mehr als nur einem Datenformat umgehen zu können. Einige Anbieter brachten Pakete integrierter Anwendungen heraus, die zusammenarbeiten konnten. Doch oftmals verbarg sich dahinter kaum mehr als die Zusammenfassung von Anwendungen, die die Datenkonvertierung oder die Berichterstellung ein wenig besser als üblich beherrschten.

Betrachten wir einmal die häufig vorkommende Anforderung, Finanzdaten gemeinsam mit Text in einem Newsletter oder einem Bericht unterzubringen. Sie hatten einige Möglichkeiten zur Auswahl.

Die meisten besseren Textverarbeitungssysteme boten bereits so etwas wie das Anlegen von Tabellen an. Heutzutage ist das eine recht komfortable Angelegenheit, bei der sogar einfache Berechnungen in den Tabellenzellen vorgenommen werden können – doch damals zu DOS-Zeiten war das eine recht primitive Geschichte. Sie konnten den Bericht in der Textverarbeitung schreiben, dabei eine einfache Tabelle anlegen und schließlich die Finanzdaten in diese Tabelle kopieren. Logisch, immer wenn Sie in Ihrem Kalkulationsprogramm die Werte änderten, mußten Sie diese erneut von Hand in das Textdokument kopieren.

Andererseits konnte man bei den Tabellenkalkulationsprogrammen auch ein wenig Text einfügen. In einer modernen Tabellenkalkulation, wie etwa Excel, können Sie beträchtliche Textmengen unterbringen und diese dazu auch beliebig formatieren wie in einer Textverarbeitung. In den Tagen von Lotus 1-2-3 jedoch konnten Sie Text nur in einzelne Zellen einfügen. Wenn Sie Ihre Berichte mit so einer Tabellenkalkulation erstellten, sahen die Texte recht langweilig aus.

Sie konnten Informationen aus der Tabellenkalkulation exportieren und diese in das Dokument in der Textverarbeitung einfügen. Hatten Sie die Kalkulationstabelle in Text konvertiert, konnte möglicherweise die Textverarbeitung die Daten importieren und als Tabelle im Dokument darstellen. Aber auch hier mußten Sie das Ganze jedesmal aufs neue abwickeln, wenn sich die Daten änderten. Sie hätten die Daten vielleicht auch als Bild exportieren können, aber dann hätten Sie beim Vergrößern oder Verkleinern des Bildes Probleme bekommen.

Sie konnten einen Report-Generator verwenden oder eine Desktop-Publishing-Anwendung, die die beteiligten nativen Datenformate verstanden. Dieser Weg klang perfekt, abgesehen davon, daß die meisten dieser Programme auf verhältnismäßig wenige Datenformate beschränkt und umständlich zu lernen und zu beherrschen waren – irgendwie auch nicht das Wahre.

Betrachten wir das einmal näher: den Versuch, Kalkulationsdaten in ein Textdokument zu befördern. Abbildung 2.2 illustriert die Konvertierung von Tabellenkalkulationsdaten in Text und das anschließende Kopieren in das Dokument. Das Ergebnis sieht bei weitem nicht so gut aus und paßt auch nicht so recht. Sie hätten

sich natürlich auch mit Hilfe der Formatierungsfähigkeiten des Textverarbeitungsprogramms darüber hermachen können, aber das hätte zusätzlichen Arbeitsaufwand bedeutet.

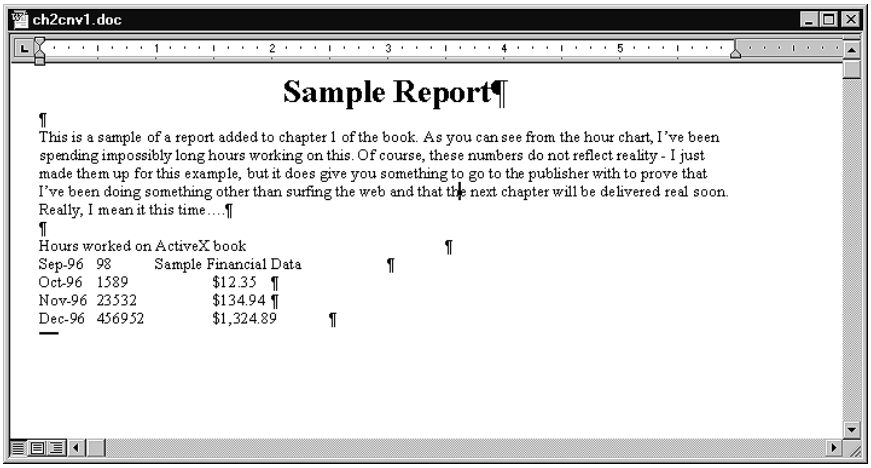

Abb. 2.2: Tabellenkalkulationsdaten als unformatierter Text importiert

Es gibt eine ganze Reihe gebräuchlicher Datenformate zur Übertragung von formatiertem Text. Die Textverarbeitung kann dies erkennen und die importierten Daten in einem Tabellenformat einfügen. Wie Sie unschwer in Abbildung 2.3 erkennen können, läßt auch dieser Weg noch reichlich zu wünschen übrig. Und auch hier erfordert das Vergrößern oder Verkleinern der Tabelle jede Menge Nacharbeit.

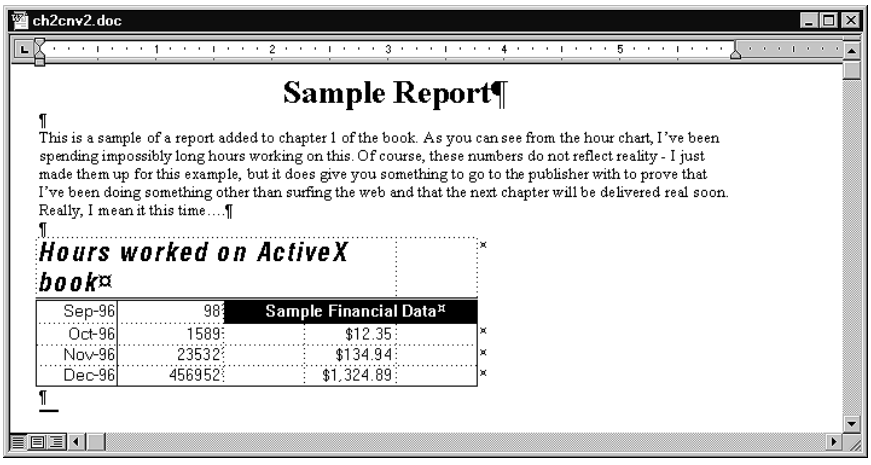

Abb. 2.3: Tabellenkalkulationsdaten in eine Tabelle importiert

Man kann ein perfektes Abbild der Tabellenkalkulation erhalten, indem man die Daten als Bitmap-Bild exportiert, wie in Abbildung 2.4 zu sehen. Die meisten Textverarbeitungen beherrschen Standard-Bitmap-Formate recht gut. Aber hierbei bleiben Sie nun mal auf der fixen Abbildung sitzen – Sie können das Bild nur mit einem Malprogramm verändern, oder Sie müssen die Daten erneut exportieren. Außerdem kranken Bitmap-Images an Skalierungsproblemen, wie in Abbildung 2.5 zu sehen ist.

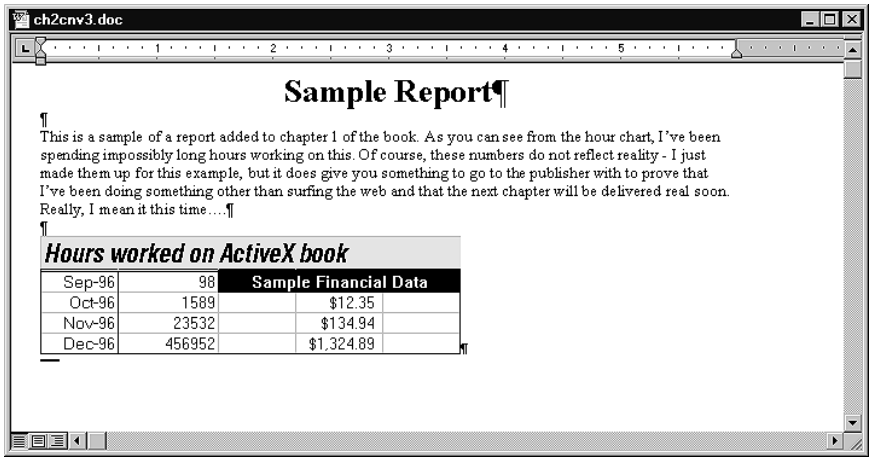

Abb. 2.4: Tabellenkalkulationsdaten als Bitmap importiert

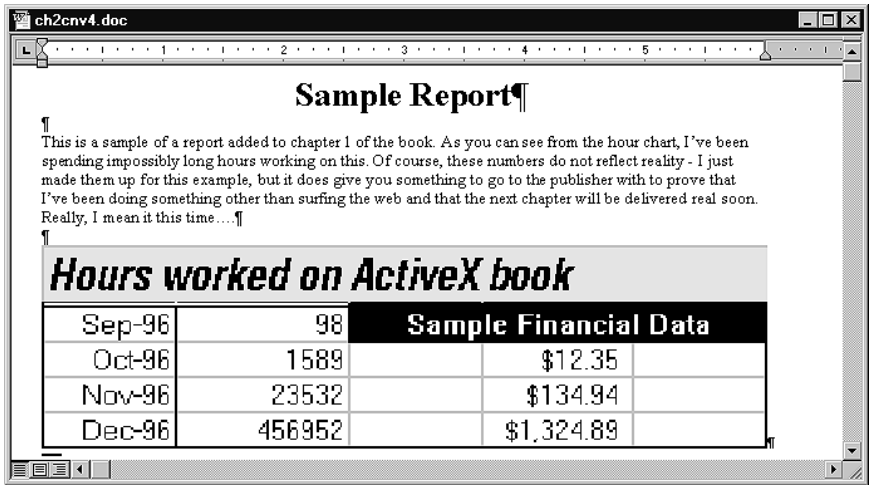

Abb. 2.5: Tabellenkalkulationsdaten als Bitmap importiert und dann skaliert

Dann gibt es noch einen weiteren bedenkenswerten Aspekt, wenn Daten wie in diesen vier Beispielen importiert werden: Die Umwandlung stellt eine Einbahnstraße dar. Nachdem die Daten einmal in ein Format konvertiert sind, mit dem eine Textverarbeitung etwas anzufangen weiß, wird es sehr, sehr schwierig, diese wieder in die Tabellenkalkulation zurück zu importieren. Günstigstenfalls bekommen Sie zwar die Daten wieder zurück, aber jegliche Formatierung bleibt dabei auf der Strecke.

Schlimmstenfalls, nämlich dann, wenn Sie die Daten als Bitmap exportiert haben sollten, wird Ihnen kaum etwas anderes übrigbleiben, als die Daten fein säuberlich von Hand wieder neu einzugeben.

2.1.1 Die dokumentenzentrierte Umgebung

Ein wesentliches Charakteristikum einer anwendungszentrierten Umgebung ist der Konvertierungsprozeß, über den erst die Daten von einer Anwendung zur anderen bewegt werden können. Sie können es *Konvertierung*, *Import*, *Export* oder auch *Ausschneiden-und-Einfügen* nennen – wie auch immer, das Prinzip ist in allen Fällen das gleiche. Eine explizite Operation muß stattfinden, um die Daten von einem in ein anderes Datenformat umzuwandeln. Zudem geht fast immer ein Teil des Informationsgehalts verloren.

»Na toll«, werden Sie sagen, »arbeiten denn nicht heutzutage alle Anwendungen nach diesem Prinzip?«

Sie werden zumeist recht haben, aber langsam sehen wir auch die ersten Auswirkungen einer fundamentalen Bewegung in der Programmierung. Diese Bewegung hat ihre Ursprünge in einer Vision für PCs, die Microsoft vor rund einem Jahrzehnt aufbrachte. Die Idee war, daß die Computerei dokumentenzentriert statt anwendungszentriert sein solle.

Was soll das genau bedeuten? Dahinter steckt der Gedanke, daß sich Anwender nicht länger mit Programmen und dazugehörenden Dateien beschäftigen sollten, sondern nur noch mit den Dokumenten, mit denen sie arbeiten. Dokumente sollten jeden beliebigen Objekttyp enthalten können, von Text über Bilder und Klänge bis hin zu solchen, die man sich derzeit noch gar nicht vorstellen kann. Anwender sollen sich nicht mehr darum kümmern müssen, welche Anwendung sie gerade verwenden. Wenn sie ein Dokument öffnen, würde das Betriebssystem automatisch den zum Betrachten oder Bearbeiten des Dokuments oder jedes beliebigen darin enthaltenen Objekts notwendigen Code starten. Sie würden sich auch keine Gedanken mehr darüber machen müssen, in welcher Form das Dokument auf der Festplatte abgelegt sein würde. Die Objekte des Dokuments könnten alle in einer Datei zusammengefaßt oder aber über Tausende von Dateien verstreut sein, und sie könnten sich alle auf ein und derselben Festplatte befinden oder genauso gut über ein ganzes Netzwerk verteilt sein. Würde eine Datei verschoben, sollte das Betriebssystem den Weg verfolgen, damit das Dokument jederzeit seine Datenbestandteile wiederfinden könnte.

Klingt das nicht allmählich vertraut? Auch wenn die meisten Anwender nach wie vor mehr oder weniger anwendungszentriert denken, ist der Übergang zu dokumenten-zentrierten System in vollem Gange. Eine der Manifestationen dieser Veränderung zeigt sich darin, daß Anwender sich zunehmend daran gewöhnen, verschiedenste Objekttypen in ihre Text- oder Kalkulationsdokumente einzubetten. Dies zeigt Abbildung 2.6; hier wurde ein Excel-Tabellenobjekt in ein Textdokument eingefügt. Nicht nur das Erscheinungsbild stimmt, sondern auch die Größenanpassung. Es scheint schon ein wenig wie Magie, daß eine Textverarbeitung weiß, wie ein Tabellenobjekt handzuhaben und auf diese Weise darzustellen ist. Doch um genau diese Form der Magie geht es in diesem Buch. Der dokumentenzentrierte Vorgang der Einbettung eines Tabellenobjekts in ein Textdokument wird in Abbildung 2.7 dargestellt.

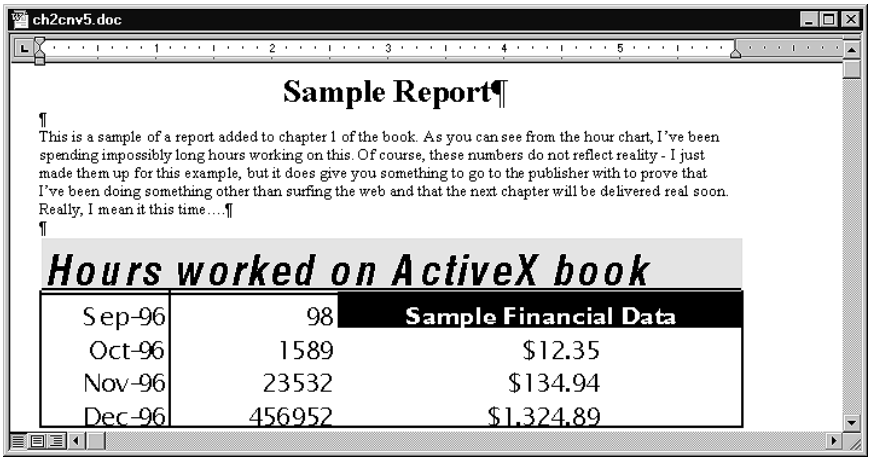

Abb. 2.6: Ein Excel-Tabellenobjekt eingebettet in ein Word-Dokument

Das vielleicht größte derzeit verwendete dokumentenzentrierte System ist das World Wide Web, wo eine HTML-Seite Bilder, Videos, Klänge, Applets und viele weitere Datentypen enthalten kann. Diese Objekte können über die ganze Welt verteilt sein – und trotzdem brauchen sich die Anwender dieses Systems keine Gedanken um das Anwerfen eines Browsers zu machen, sondern lediglich nach einem Dokument zu fragen. Das System kümmert sich um das Starten des Browser und der Browser startet wiederum die jeweils zur Darstellung eines Objekts einer Seite benötigte Anwendung.

Ist eine dokumentenzentrierte Umgebung wirklich besser für Computer-Anwender als die gewohnte anwendungszentrierte? Dies ist eine gute Frage, allerdings eine, die ich nicht zu beantworten vermag. Es ist wahrscheinlich noch zu früh, um eine sichere Antwort geben zu können. Die heutigen Implementierungen von dokumentenzentrierten Systemen sind alles andere als perfekt.

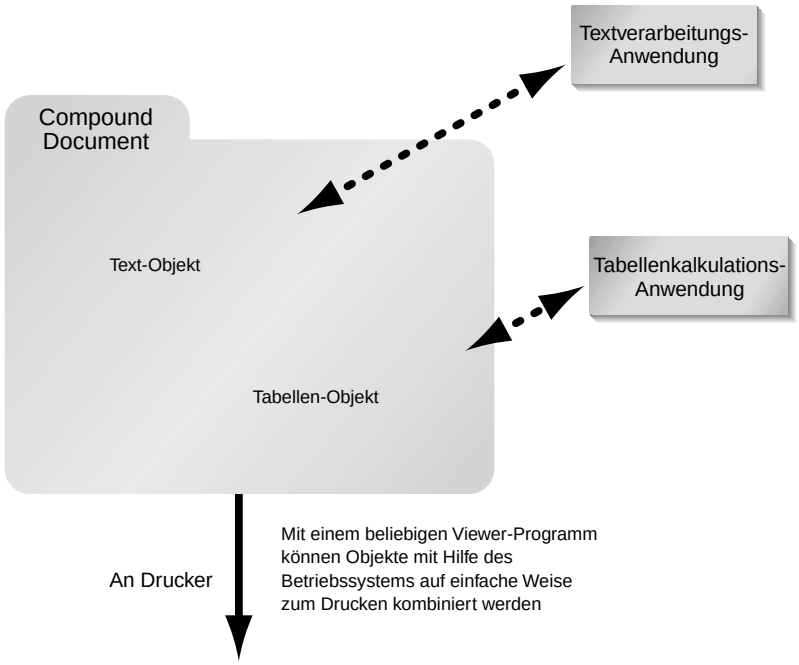

Abb. 2.7: Eine typische dokumenten-zentrierte Umgebung

Verlorene Verknüpfungen, wenn Dateien verschoben oder Server vom Netz genommen werden, und System-Konfigurationsprobleme zeigen dies nur allzu deutlich. Für uns Programmierer jedoch gibt es offensichtlich nur eine wahre Antwort, nämlich die, daß uns nichts anderes übrigbleibt, als auf den Zug aufzuspringen und mitzufahren. Der Übergang zum dokumentenzentrierten Modell schreitet rasch voran und wird zunehmend Einfluß darauf haben, wie wir unsere Anwendungen gebrauchen und wie wir sie schreiben.

Frage: Was hat das alles mit ActiveX zu tun?

Antwort: Microsoft treibt die Vision des dokumentenzentrierten Computing voran. ActiveX ist Microsofts Implementierung dieser Vision.

2.2 Der Weg zu ActiveX

Was wir heute unter ActiveX verstehen, erschien nicht über Nacht auf der Bildfläche. Nun, der Name »ActiveX« erschien über Nacht, jedoch nicht die Technologie. Diese ist im Laufe der Jahre entstanden. Wenn Sie diese Evolution nachvollziehen können, werden Sie die heutige Technologie und neue Technologien bei ihrem Auftauchen besser verstehen können.

2.2.1 DDE

Der erste Schritt auf dem Weg zu ActiveX war »Dynamic Data Exchange« (DDE). Diese Technik wird immer noch von einigen Anwendungen und auch von Visual Basic unterstützt, allerdings kaum mehr als zur Wahrung der Rückwärtskompatibilität zu älteren Anwendungen. Nur sehr wenige neuere Anwendungen setzen noch substantiell auf DDE.

DDE war die erste Möglichkeit für Anwendungen, miteinander kommunizieren zu können. Diese Kommunikation hatte zwei Erscheinungsformen: Daten und Befehle. Anwendungen konnten Daten an andere Anwendungen senden oder von dort empfangen. Die Daten wurden über den Namen der Anwendung, ein individuelles Kennwort (»Topic«) und Element-Namen innerhalb der Anwendung identifiziert. Es war auch möglich, sogenannte »heiße« Verbindungen (Hot Links) einzurichten, so daß eine Anwendung eine andere über Änderungen an Informationen benachrichtigen konnte.

Anwendungen konnten ebenfalls Befehle an andere Anwendungen absetzen. Wenn etwa in einer Textverarbeitung ein Makro namens `OpenFile` über DDE zugänglich implementiert gewesen war, konnte man über DDE einen `OpenFile`-Befehl an diese Textverarbeitung senden, und diese hätte eine angegebene Datei geöffnet.

DDE war ziemlich beschränkt, bereitete Schwierigkeiten bei der sauberen Implementierung und war unzuverlässig. Es war auch noch ein anwendungszentrierter Blickwinkel auf die gemeinsame Verwendung von Daten. Es erlaubte zwar Anwendungen, Daten gemeinsam zu nutzen und auszutauschen, erlaubte es aber Dokumenten nicht, Daten jenseits der in die Anwendungen eingebauten Mechanismen zu teilen.

Nachdem Sie nun ein wenig über DDE wissen, sind Sie frei, das meiste dessen, was Sie gerade gelesen haben, auch wieder schnell zu vergessen. DDE war ein großer Schritt, nur daß es sich in einem Punkt als in die falsche Richtung gehend erwies. Dieser Punkt betraf die Fähigkeit, Befehle in anderen Anwendungen auszuführen – der Vorläufer-Technologie der heutigen ActiveX-Automation.

2.2.2 OLE 1.0

OLE steht für »Object Linking and Embedding«. OLE 1.0 war der erste Schritt zur Implementierung eines der bedeutendsten Konzepte der dokumentenzentrierten Umgebung: die Idee, daß ein Dokument verschiedene Typen von Objekten enthalten könne. Der Bestandteil »Linking and Embedding« verweist auf die möglichen Unterbringungsorte der Objekte, die Teil eines Dokuments sind. Sie können entweder in ein Dokument eingebettet, oder mit diesem verknüpft sein. Im letzteren Falle enthält das Dokument lediglich den Namen einer Datei oder einen anderen Verweis auf den Ort, an dem das Objekt gespeichert ist.

OLE 1.0 bot ebenfalls einen Weg für Anwendungen, mit diesen zusammengesetzten (»compound«) Dokumenten zu arbeiten. Jede Anwendung konnte ein Doku-

ment darstellen, das aus vielen verschiedenen Objekten bestand, und nach einem Doppelklick auf eines der Objekte konnte dieses bearbeitet werden – unter Verwendung der damit assoziierten Anwendung. Sehen Sie sich das Beispiel in Abbildung 2.7 noch einmal an. Das Dokument enthält ein Textobjekt und ein Tabellenobjekt. Wenn Sie das Dokument mit einer Textverarbeitung öffnen, werden Sie den Text und die Tabelle zu Gesicht bekommen. Die Textverarbeitung wird die Tabellenkalkulation zur Bearbeitung des Objekts starten. Wenn Sie die Bearbeitung des Tabellenobjekts abschließen, wird das Dokument mit allen von Ihnen vorgenommenen Änderungen aktualisiert.

Dieser Weg unterscheidet sich in zweierlei Hinsicht deutlich von den oben beschriebenen anwendungszentrierten Techniken. Zum einen findet keinerlei Konvertierung statt. Das Kalkulationstabellenobjekt bleibt unverändert ein Kalkulationstabellenobjekt, auch wenn es auf geheimnisvolle Weise mitten in einem Textverarbeitungsdokument auftaucht. Da keine Konvertierung stattfindet, gehen keine Daten verloren – die originale Formatierung der Tabelle bleibt erhalten. Zum anderen betreffen alle Änderungen der Tabelle, die innerhalb der Anzeige im Textdokument vorgenommen werden, gleichfalls die originale Kalkulationstabelle. Das Problem der Rückkonvertierung vom Textdokument zur Kalkulationstabelle ist damit auch hinfällig geworden.

OLE 1.0 krankte jedoch an einem großen Problem: Es war eine Software-Technologie, die der seinerzeit verfügbaren Hardware weit voraus war. Es war die Ära von Windows 2.x und 3.0, einer Zeit, zu der die 640 KByte-Speichergrenze durchaus noch gegenwärtig war (auch 3.0 benötigte den Adreßraum bis 640 KByte für verschiedene Belange, trotz der virtuellen Speicherverwaltung). Der Versuch, ein umfangreiches Textverarbeitungsprogramm wie Microsoft Word gleichzeitig mit einer umfangreichen Tabellenkalkulation wie Excel laufen zu lassen, war ein zeitraubendes, frustrierendes und oftmals auch gefährliches Unterfangen. Na ja, das war noch die Zeit der UAEs, der »Unrecoverable Application Errors«, im Gegensatz zur Windows-3.1-Ära mit den GPFs, den »General Protection Faults«, die in der heutigen, modernen Zeit durch die Ausnahmefehler ersetzt worden sind – diese treten anders auf und melden sich mit »Your program has performed an illegal operation«.

Ungeachtet dieser Grenzen war OLE 1.0 der erste echte Schritt in Richtung des dokumentenzentrierten Computing unter Windows. Dennoch war dieser Technologie kein allzu großer Erfolg beschieden, was aber auch angesichts der großen Aufgabe, eine dokumentenzentrierte Umgebung zu implementieren, nicht weiter wundert.

Eine dokumentenzentrierte Umgebung stellt folgende Anforderungen:

- Eine Anwendung muß Objekte darstellen können, über die sie nichts weiß: Ein Dokument einer dokumenten-zentrierten Anwendung soll beliebige Objekte aufnehmen können, auch wenn diese unter Umständen zum Zeitpunkt der Entwicklung der Anwendung noch gar nicht bekannt sein sollten.

- Die Anwendung muß Dokumente einschließlich ihr unbekannter Objekte laden und sichern können.
- Die Anwendung muß eine visuelle Benutzeroberfläche zur Bearbeitung der ihr unbekannten Objekte anbieten können.

Und wenn wir schon einmal dabei sind – wie wäre es mit folgenden Ergänzungen:

- Die Anwendung soll die Möglichkeit haben, Befehle zur Manipulation von ihr unbekannten Objekten ausführen zu können.
- Die Anwendung sollte Drag&Drop-Operationen für ihr unbekannte Objekte unterstützen.

Für einen Programmier-Neuling ist es oftmals schon schwierig genug, mit bekannten Daten und Formaten umzugehen. Um wieviel komplizierter dürfte es dann erst sein, mit Daten zu hantieren, die eine Anwendung noch nicht einmal richtig zu Gesicht bekommt? Aber genau auf diese Frage sollte dann OLE 2.0 die Antwort liefern.

2.2.3 OLE 2.0

Auf den folgenden Seiten werde ich eine schier unlösbare Aufgabe in Angriff nehmen, nämlich Ihnen eine klare, verständliche Beschreibung von OLE 2.0 vorzulegen. OLE 2.0 ist die wahrscheinlich komplexeste Software-Technologie, die mir je begegnet ist. Ich glaube nicht, daß Sie allzu viele Leute finden werden, die sie vollständig verstanden haben, und ich erhebe nicht den Anspruch, zu diesen zu zählen.

Aber lassen Sie sich bitte keinen Schrecken einjagen, und bitte überspringen Sie diesen Abschnitt nicht. Es spricht vieles dafür, daß Sie ActiveX kaum verstehen können, ohne zuvor einen gründlichen Überblick über OLE 2.0 gewonnen zu haben (warum das so ist, wird Ihnen sicher spätestens am Ende des Kapitels einleuchten).

Beruhigend ist immerhin, daß Sie OLE 2.0 nicht von vorne bis hinten komplett verstehen müssen, um Gebrauch davon zu machen. Denn von dem größten Teil der Komplexität schirmt Visual Basic Sie ab (wie auch die Microsoft Foundation Classes – MFC – es für Visual-C++-Programmierer tun). Letztendlich ist die Arbeit mit ActiveX in Visual Basic so einfach, daß Sie ActiveX-Komponenten erstellen können, auch ohne weiteren Durchblick zu haben. (Ok, so entstehen vielleicht keine guten ActiveX-Komponenten, aber Sie würden sich ja kaum mit diesem Buch befassen, wenn Sie sich nicht darüber im klaren wären.)

Ich habe gute Gründe für meine Bitte an Sie, sich gründlich mit der OLE 2.0- und der ActiveX-Technologie auseinanderzusetzen:

- Sie werden die »Sprache« von COM verstehen lernen. Zum Beispiel: Was ist ein Interface? Was bedeutet `IDispatch`?
- Es wird Ihnen nicht nur dabei helfen, gute Komponenten zu schreiben, sondern Sie werden ein Experte in Sachen Komponenten-Entwicklung unter Visual Basic werden (und es ist meine Absicht, Sie zu nichts geringerem als einem Guru darin zu machen).
- Es ist ein interessanter Stoff. Denn abgesehen davon, daß Visual Basic die einfachste, effizienteste und kostengünstigste Entwicklungsumgebung für Windows-Programme darstellt, ist der Spaß, den man mit Visual Basic hat, weithin bekannt.

Ich werde diese Technologie in zwei Etappen in Angriff nehmen – in der einen geht es um die Philosophie von OLE, in der anderen um deren Implementierung. In diesem Abschnitt wird es nun weiterhin um die Philosophie von OLE 2.0 gehen – um die Ideen, auf denen es beruht und um die damit erreichte Funktionalität. Informationen über die Implementierung werden im verbleibenden Rest von Teil I enthalten und ebenfalls über das ganze Buch verteilt sein. Ich hoffe, daß ein gutes Verständnis des Sinns und Zwecks der Technologie Ihnen bei der Implementierung und bei der Anwendung behilflich sein wird.

Ein technologischer Eintopf

Eigentlich ist OLE 2.0 nicht nur eine einzelne Technologie, sondern vielmehr eine Sammlung von Technologien, die sogar nur wenig miteinander zu tun haben. Der wesentliche Kern der Gemeinsamkeiten besteht darin, daß sie alle auf die gleiche Weise mit Objekten arbeiten.

Nun, unter dem Begriff *Objekt* kann alles mögliche verstanden werden. Da gibt es etwa die objektorientierte Programmierung mit Objekten als Datenstrukturen innerhalb eines Programms. Wenn allerdings von Objekten im Zusammenhang mit OLE 2 oder ActiveX die Rede ist, geht es um einen ganz spezifischen Typ an Objekten, dessen Bezeichnung *Component Object* oder auch manchmal *Window Object* lautet. Die Objekte unterliegen einem Standard, dem *Component Object Model (COM)*. Der COM-Standard definiert folgendes:

1. Einen allgemeingültigen Weg für Anwendungen zum Zugriff und zur Ausführung von Operationen auf Objekte. Um diesen Punkt wird es in Kapitel 3, »Objekte und Visual Basic«, und in Kapitel 4, »Das Component Object Model: Interfaces, Automation und Binding« gehen.
2. Einen Mechanismus zur Verfolgung, welche Objekte gerade in Verwendung sind, um nicht länger benötigte Objekte entfernen zu können.
3. Einen Standard-Mechanismus zur Fehlermeldung und einen Satz an Fehlercodes und -werten.

4. Einen Mechanismus, der Anwendungen den Austausch von Objekten ermöglicht.
5. Einen Weg, Objekte zu identifizieren und mit Anwendungen zu verknüpfen, die wissen, wie ein jeweiliges Objekt implementiert ist.

Warum sind diese Faktoren von so großer Bedeutung und in welchem Zusammenhang stehen sie mit der Idee des dokumentenzentrierten Computing? Vergegenwärtigen Sie sich noch einmal das erwähnte Beispiel eines Dokuments, das Text und Kalkulationstabellen enthält und von einer Textverarbeitungsanwendung geöffnet wurde. Wie kann eine Textverarbeitung ein Tabellenobjekt darstellen, ohne etwas darüber zu wissen?

Wenn es sich bei dem Tabellenobjekt um ein COM-Objekt handelt, ist das eine einfache Angelegenheit. Das COM-Objekt kann einen Standard-Satz an Funktionen unterstützen, die ein Objekt anweisen, sich selbst darzustellen (COM-Standard Nr. 1). Doch um was geht es, wenn wir davon sprechen, daß ein Objekt einen Satz an Funktionen unterstützt? Das Textdokument enthält die Daten der Kalkulationstabelle (darauf, wie die Tabellendaten vom eigentlichen, eigenen Text des Dokuments separiert bleiben, werden ich in Kürze eingehen). Das Dokument enthält ebenfalls einen Identifikationsschlüssel, der dem System mitteilt, daß es sich um ein Tabellenobjekt handelt (*dieser* informiert das System und *nicht* die Textverarbeitung!). Das Dokument enthält *keinesfalls* die zur Darstellung des Objekts benötigten Funktionen. Die sind in einer Anwendung oder einer DLL enthalten, die das Objekt kennt und versteht – in diesem Fall die Tabellenkalkulationsanwendung.

Und wo findet die Textverarbeitungsanwendung nun die Tabellenkalkulationsanwendung, die die Funktionen zur Darstellung enthält? Sie greift auf die COM-Mechanismen zurück, über die Anwendungen, die Objekte enthalten, ausfindig gemacht werden (COM-Standard Nr. 5). Natürlich wird es erforderlich sein, diese Anwendung zu starten und ihr den Zugriff auf die Daten des Objekts zu ermöglichen (COM-Standard Nr. 4). Tritt während der Bearbeitung der Darstellung des Tabellenobjekts durch die andere Anwendung ein Fehler auf, wird die Tetverarbeitung den Fehler verstehen (COM-Standard Nr. 3). Und wenn beide mit dem Tabellenobjekt fertig sind, wird es freigegeben und entfernt (COM-Standard Nr. 2).

Diese Beschreibung ist eine starke Vereinfachung dessen, was sich hinter den Kulissen abspielt, aber das Prinzip dürfte klar geworden sein, wie ich hoffe. Das Component Object Model macht es Anwendungen möglich, unbekannte Objekte zu manipulieren. Es ist die Technologie für dokumentenzentriertes Computing, die die Grundlage aller künftigen Microsoft-Betriebssysteme bilden wird, bis hin zu ActiveX-Controls und sogar Visual Basic.

Ich habe OLE einen »technischen Eintopf« genannt. Aus welchen weiteren wohlschmeckenden Zutaten besteht denn dieser COM genannte Eintopf? In den folgenden Abschnitten geht es nun um weitere wesentliche Bestandteile von OLE.

UUID (oder GUID oder CLSID)

Eine der essentiellen Anforderungen an OLE besteht darin, Objekte identifizieren zu können. Wenn eine Anwendung mit einem Dokument arbeitet, das mehrere Objekte enthält, muß sie dazu in der Lage sein, den Typ eines jeden einzelnen Objekts zu erkennen, so daß das System genau die Anwendung findet, die sich um das Objekt kümmern kann.

Dazu weist COM jedem Objekttyp einen 16 Byte langen Wert zu. Dieser Wert taucht unter verschiedenen Bezeichnungen auf, je nach Verwendungszweck. *UUID* bedeutet in Langform »Universally Unique Identifier«, *GUID* heißt »Globally Unique Identifier«, *CLSID* steht für »Class Identifier« und *IID* für »Interface Identifier«.

Wenn Sie sich einen GUID in der System-Registrierung ansehen, sieht er gewöhnlich etwa so aus:

```
{97OEDBA1-111C-11d0-92B0-00AA0036005A}
```

Wenn Microsoft von »Globally Unique«, also »weltweit einmalig« spricht, dann ist das kein Scherz. Nachdem ein GUID einmal einem Objekt zugewiesen worden ist, ist ziemlich effektiv garantiert, daß dieser einmalig im ganzen Universum ist und bleibt, auf immer und ewig. Zwei Faktoren stellen sicher, daß die Zahl einmalig ist. Zum einen wird jeder Teil eines GUIDs anhand der Adresse der Netzwerkkarte in Ihrem System berechnet (vorausgesetzt, es ist eine vorhanden). Jede jemals gebaute Netzwerkkarte verdankt den Industriestandards bereits eine einmalige Adresse. Zum anderen ist ein GUID eine riesengroße Zahl, so daß auch ohne Netzwerkkarte ein GUID-Generator-Programm eine Zahl erzeugen kann, bei der die Wahrscheinlichkeit, einen identischen Zwilling zu haben, geradezu mikroskopisch klein ist.

Visual Basic generiert automatisch GUIDs für Ihre Objekte – Sie werden wohl eher Probleme damit bekommen, überflüssig gewordene GUIDs zu beseitigen, statt neue zu bekommen. Der wichtige Punkt, den Sie sich merken sollten, ist hier, daß jedes Objekt, das Sie in Visual Basic erschaffen, über einen GUID identifiziert wird – und nicht über den Namen des Objekts –, so daß Ihr Programm in jedem Fall funktionieren wird, auch wenn Sie zufällig den gleichen Namen wie jemand anderes verwendet hätten (was Sie aber trotzdem vermeiden sollten). Die Objekte werden nicht durcheinandergeraten.

GUIDs werden auch dazu verwendet, Sätze von Funktionen zu identifizieren, die *Interfaces* genannt werden. Darüber werden Sie später mehr erfahren.

Darstellung von Objekten

OLE definiert Standard-Mechanismen zur Darstellung von Objekten. Das bedeutet, daß eine Container-Anwendung, wie etwa Microsoft Word, eine Fläche auf dem Bildschirm oder auf einer zu druckenden Seite zur Verfügung stellen und dem Objekt erlauben kann, sich selbst in diese Fläche hineinzuzeichnen. Wie geht

das vor sich? Word kennt den GUID des Objekts und die Standard-Funktionen, die das Objekt zur Darstellung (und einige weitere Aufgaben) verwendet. Das System kann die Registrierung nach dem GUID durchsuchen und so die Anwendung oder die DLL ausfindig machen, die den eigentlichen Code dieser Funktionen enthält.

OLE 2.0 geht sogar noch weiter. Hier wird ein Mechanismus definiert, über den eine Objekt-Anwendung (ein »Server«) Teile eines Dokument-Containers übernehmen kann, so daß Ihnen die Werkzeuge dieser Anwendung zum Bearbeiten des Objekts direkt zur Verfügung stehen, auch wenn Sie sich gerade eigentlich im Rahmen einer ganz anderen Anwendung befinden. Dies wird als *In-Place-Editing* bezeichnet. Die Funktionsweise von ActiveX-Controls beruht auf einer Variante dieses Prinzips. Glücklicherweise erledigt Visual Basic alle Details der Implementierung dieser ziemlich komplexen Technologie.

Objekt-Marshaling

OLE definiert einen Mechanismus, über den Objekte von Anwendung zu Anwendung verschoben werden können – das »Marshaling«. Wenn Sie noch neu in der 32-Bit-Programmierung sein sollten, wird Ihnen der Hintergrund der Angelegenheit nicht viel zu sagen, da es unter 16-Bit-Windows recht einfach ist, Speicherblöcke zwischen Anwendungen hin und her zu schieben. Sie werden jedoch recht schnell mitbekommen, daß das Verschieben von Objekten zwischen Prozessen unter 32-Bit-Windows eine viel diffizilere Angelegenheit ist. Zum Glück erledigt bereits OLE das meiste für Sie. Auf diesen Punkt geht Kapitel 6, »Das Leben und die Lebensdauer einer ActiveX-Komponente«, näher ein.

Windows enthält mittlerweile eine erweiterte Form von COM, nämlich DCOM *(Distributed Component Object Model)*. DCOM-Objekte können sogar zwischen Anwendungen verschoben werden, die auf verschiedenen Systemen eines Netzwerks laufen.

Verbunddokumente (Compound Documents, OLE Structured Storage)

Wenn ein Objekt mehrere verschiedene Objekte enthalten kann – wie kann ein bestimmter Container solch ein Dokument speichern? Er müßte das Dateiformat für jedes einzelne Objekt kennen – eine unlösbare Aufgabe angesichts dessen, daß die Anwendung absolut ahnungslos in bezug auf die Natur und Funktionsweise eines Objekts ist. Oder ist die Aufgabe doch lösbar?

OLE handhabt die Persistenz (Dauerhaftigkeit) von Objekten (das Laden und Speichern von Objekten) auf eine der Darstellung vergleichbare Weise: Es ist Angelegenheit eines Objekts zu wissen, wie es sich selbst aus einer Datei laden oder dort hinein speichern kann. Genauso, wie OLE einen Satz an Funktionen zur Darstellung eines Objekts definiert, wird auch ein Satz an Funktionen definiert, auf den ein Objekt zurückgreifen kann, um sich selbst zu »persistieren«, also die Zeiten zwischen seiner »lebendigen« Existenz im Arbeitsspeicher zu überleben, gewissermaßen zu »überwintern«.

Sie nun vermuten vielleicht, daß dies zu einigen schwerwiegenden Problemen führen könnte. Falls irgendeines der Objekte einen Bug im Datei-I/O-Code aufweist, könnte das zu Überschneidungen mit den Dateiteilen kommen, die von anderen Objekten mit Beschlag belegt werden – Teile der Datei könnten überschrieben und damit zerstört werden. Die Aufgabenstellung des Speicherns von Objekten in einer Datei wird von einer OLE-Technologie erledigt, die *OLE Structured Storage* genannt wird. Hierbei wird eine Datei in eine Hierarchie von Speicherbereichen *(Storages)* und Datenströmen *(Streams)* gegliedert. Ein Storage können Sie sich analog zu einem Datei-Verzeichnis bzw. -Ordner vorstellen, wobei ein Stream dann einer Datei entspräche. Im Endeffekt erhalten Sie dabei so etwas wie ein eigenes Dateisystem innerhalb einer einzigen Datei. Ein Container wie Word kann einen Storage oder Stream anlegen, diesen dem Objekt übergeben und es anweisen, sich selbst in diesem Storage bzw. Stream abzulegen. Meistens schreibt der Container den GUID des Objekts dazu, so daß er beim Laden des Objekts aus einem einzelnen Storage oder Stream dessen Typ ermitteln kann.

Drag&Drop

Es gibt ein paar Operationen, die eine weitergehendere Kooperation zwischen Anwendungen notwendig werden lassen, als die reine Fähigkeit, Objekte per Drag&Drop von einer Anwendung zu einer anderen zu ziehen. Jede Anwendung muß entscheiden, welche Objekte ein Anwender auswählen und ziehen kann, und sie muß dem System einen Verweis auf diese Objekte zur Verfügung stellen. Sie muß ebenfalls entscheiden, welche Objekttypen sie von anderen Anwendungen akzeptieren und wie sie damit umgehen kann. OLE 2.0 definiert einen Mechanismus für Drag&Drop-Operationen nicht nur zwischen Anwendungen, sondern auch zwischen Anwendungen und dem Betriebssystem.

OLE Automation (ActiveX Automation)

Hiermit habe ich mir das beste für den Schluß aufgehoben. OLE-Automation ist der Nachfolger der DDE-Fähigkeiten, d.h., einer Anwendung die Ausführung von Befehlen in einer anderen Anwendung zu ermöglichen – allerdings auf einer wesentlich leistungsfähigeren Ebene. Wie Sie nun wissen, ermöglicht OLE einer Anwendung den Export von Objekten in alle Welt. OLE-Automation ermöglicht das Ausführen von Befehlen, die diese Objekte zur Verfügung stellen, und den Transfer von Daten von und zu diesen Objekten.

OLE-Automation bietet einer Anwendung nicht nur den Aufruf dieser Funktionen eines Objekts, sondern sogar die Möglichkeit, zur Laufzeit festzustellen, welche Funktionen dies sind und welche Parameter sie benötigen (falls ein Objekt diese Informationen öffentlich zugänglich gemacht haben sollte).

OLE-Automation legt den Grundstein für fast die gesamte Funktionsweise von ActiveX-Komponenten und ist in der Tat das zentrale Thema der nächsten paar Kapitel. Bevor wir uns jedoch näher damit befassen, müssen wir noch einen weiteren Schritt zurücklegen. Wie kommen wir von OLE 2.0 nach ActiveX? Wie passen ActiveX-Controls in dieses Szenario?

Bevor ich darauf eine Antwort geben kann, müssen wir uns noch mit einer weiteren Technologie befassen – einer Technologie, die alle bezüglich ihres Erfolgs überrascht, und absolut nichts mit OLE zu tun hat.

Vorhang auf für VBX

Im Jahre 1991 wurde Visual Basic von Microsoft auf den Markt gebracht. In der heutigen Zeit des visuellen Programmierens ist es kaum noch nachvollziehbar, welche Revolution das für die Programmierung von Windows-Anwendungen seinerzeit bedeutet hat. Zuvor bestand jede noch so simple Windows-Anwendung aus hunderten von Zeilen C-Code. Für Einsteiger in die Windows-Programmierung waren sechs Monate harten Studiums üblich, um sich die wichtigsten Kenntnisse und Fähigkeiten anzueignen. Überhaupt war die Windows-Programmierung damals eine komplexe und düstere Angelegenheit.

Visual Basic stellte dies auf den Kopf. Man konnte von nun an einfache Windows-Anwendungen in wenigen Minuten erstellen. Es war erstmals möglich, triviale Windows-Anwendungen für den Wegwerf-Gebrauch zu schreiben. Windows-Programmierung machte auf einmal richtig Spaß.

Visual Basic versteckte das meiste der Windows-Komplexität in der Basic-Sprache. Die Formulargestaltung ließ das Anlegen einer Benutzeroberfläche einfach werden. (Sie sehen, daß ich nicht gesagt habe, daß das Anlegen einer *guten* Benutzeroberfläche leichtgemacht wird. Mit Visual Basic kann man mit gleicher Leichtigkeit sowohl schlechte als auch gute Benutzeroberflächen gestalten.) Man zog einfach Controls auf das Formular. Diese Controls verfügten über Eigenschaften, die von der Anwendung aus gesetzt werden konnten. Die Werte dieser Eigenschaften konnten zur Design-Zeit gesetzt und mit der Anwendung gespeichert werden. Die meisten Controls hatte im Prinzip ihre jeweils eigene Benutzeroberfläche; sie wurden auf dem Formular dargestellt und konnten angeklickt oder sonstwie zur Laufzeit manipuliert werden.

Selbst wenn das alles gewesen wäre, was Visual Basic gebracht hat, wäre es allein deswegen schon ein bemerkenswertes Produkt gewesen. Doch die Väter von Visual Basic gingen noch einen beachtlichen Schritt weiter: Sie sorgten für eine Erweiterbarkeit der Sprache. Zunächst ermöglichten sie den direkten Zugriff auf Funktionen in DLLs (Dynamic Link Libraries), insbesondere auf die der Windows-API. (Mein erstes Buch *Visual Basic Programmer's Guide to the Win32 API* und dessen Nachfolger *Dan Appleman's Visual Basic Programmer's Guide to the Win32 API*, beide von *Ziff-Davis Press*, befassen sich umfassend mit diesem Thema.) Darüber hinaus ermöglichten sie es, Visual Basic mit externen Controls (Custom Controls) zu ergänzen. Diese VBX-Controls sahen aus der Sicht des Programmierers so aus, als ob sie in der Entwicklungsumgebung selbst vorhanden gewesen wären. Sie erschienen in der Werkzeugsammlung und verhielten sich exakt wie die in die Sprache von vornherein eingebauten Controls.

Fast über Nacht entstand ein komplett neuer Markt mit einem Angebot von einer Vielzahl von (Custom-)Controls für nahezu jeden erdenklichen Zweck. (Ich grün-

dete selbst ein Unternehmen, Desaware, einzig mit dem Ziel, Controls für Visual Basic zu entwickeln.) Die Verfügbarkeit eines breiten Spektrums an speziellen Controls ließen Visual Basic zunehmend mächtiger und populärer werden. Die Existenz eines breiten Visual-Basic-Markts gestattete es VBX-Entwicklern, ihre Kosten auf eine größere Anzahl Kunden umzulegen, so daß sie die Controls an die Entwickler zu Preisen weit unter den Kosten für deren Eigenentwicklungsaufwand weitergeben konnten. Dies war ein wesentlicher Aspekt, da VBX-Controls in C oder C++ geschrieben werden mußten, was eine diffizile Angelegenheit war – weitaus schwieriger als das Programmieren in Visual Basic. Mit der Zeit förderte Microsoft diesen Markt und unterstützte die Anbieter in nicht unerheblichem Maße.

Visual Basic realisierte also nicht nur den Traum der einfachen Windows-Programmierung, sondern zugleich den Traum des komponentenbasierten Entwikkeln. Die Entwicklung von komplexen Anwendungen aus kostengünstigen, wiederverwendbaren Software-Komponenten wurde möglich. Diese Kombination war es, die Visual Basic den überwältigenden Erfolg bescherte, so daß weitaus mehr Visual-Basic-Pakete als solche für C oder C++ verkauft wurden.

Betrachten wir einmal im Rückblick die Charakteristiken eines Custom-Controls. Es ist gewissermaßen ein Objekt. Es enthält Daten. Die Daten, die zur Design-Zeit gesetzt wurden, können in einer Projekt-Datei gesichert werden, und jedes Control weiß, wie es seine eigenen Daten zu speichern hat. Visual Basic kann mit jedem VBX-Control umgehen, da jedes VBX-Control einen Standard-Satz an Funktionen unterstützt, die Visual Basic manipulieren kann. Ein Custom-Control kann eine visuelle Erscheinung haben, und jedes VBX-Control kann sich selbst und eigenverantwortlich darstellen, wenn es dazu von der Visual-Basic-Umgebung aufgefordert wird. Eine ausführbare Datei enthält die persistenten Daten der Custom-Controls, jedoch wird die Implementierung des Controls – die eigentliche Funktionalität – in einer separaten DLL mit der Dateierweiterung VBX gehalten. Visual Basic speichert mit einem Formular zusammen nicht nur die Daten jeder einzelnen Instanz der Controls (jedes Control-Objekts), sondern auch Informationen zur Identifikation der Controls, so daß die richtige VBX-Datei für jedes Objekt geladen werden kann.

Klingt das nicht alles irgendwie vertraut? Das sollte es. Es sind die gleichen Charakteristiken, die ich zuvor für OLE 2.0 beschrieben hatte. Lassen Sie mich aber betonen, daß ein VBX *kein* COM-Objekt ist. VBX beruhen auf einer eigenen Visual-Basic-spezifischen Technologie, die nur unter 16-Bit implementiert worden ist – in Visual Basic 3, dem 16-bittigen Teil von VB4 und anderen Umgebungen, die sich um Kompatibilität mit dem VBX-Standard bemühten.

Bühne frei für Visual Basic 4.0

Visual Basic 1.0 beruhte trotz aller Revolution auf einer Reihe bekannter Technologien. Es wurde eine Basic-Sprach-Engine verwendet, die in 16-Bit-Assembler geschrieben worden war. Diese Engine wurde mit einem Formular-Designer-

Paket namens »Ruby« zusammengebracht, die ursprünglich von Alan Cooper, dem Vater von Visual Basic, entwickelt worden war. Es war klar, daß Visual Basic, wie es in den Versionen 1.0 bis 3.0 implementiert worden war, kein langes Leben beschieden sein würde, insbesondere im Hinblick auf die neuen 32-Bit-Betriebssysteme, die sich damals in Entwicklung befanden. Microsoft begann daher mit der Entwicklung einer Sprach-Engine, die man »Object Basic« nannte und heute als VBA, »Visual Basic for Applications«, bekannt ist. Diese Engine wurde nicht nur das Fundament für Visual Basic 4.0 und seine Nachfolger, sondern für alle Microsoft-Anwendungen, zumindest für die programmierbaren darunter.

VBA war ein komplettes Re-Design, für das die Sprach-Engine neu geschrieben wurde. Die meisten Änderungen hinter den Kulissen werden wohl für immer nur den Microsoft-Entwicklern bekannt bleiben, die wichtigste Änderung ist jedoch offen bekannt: VBA basiert auf COM.

Welche Bedeutung hat dies? Es bedeutet, daß VBA aus Objekten besteht, die echte OLE-Objekte sind. VBA verwendet nicht nur OLE-Automation beim Programmieren anderer Objekte und Anwendungen, sondern verwendet auch OLE intern, um Befehle an seine eigenen Objekte auszuführen. Genauso wird OLE verwendet, wenn Sie Ihre eigenen Objekte erstellen, die dann sowohl innerhalb von VBA verwendet oder veröffentlicht als auch von anderen Anwendungen genutzt werden können.

Als die Microsoft-Entwickler Visual Basic 4.0 auf der Basis der COM-Technologie neu erstellten, wußten sie, daß sie Custom-Controls weiterhin unterstützen mußten. Sie waren sich aber bewußt, daß die VBX-Technologie obsolet geworden war. Sie benötigten ein OLE-Äquivalent. Zum Glück existierte es bereits. Wie Sie wissen, stellt OLE bereits die Möglichkeit zur Verfügung, Objekte in einem Container zu plazieren. Diese Objekte können per OLE-Automation programmiert werden, und sie haben bereits die Fähigkeit, sich selbst in Dateien zu schreiben. Das einzige, was diese Objekte noch nicht beherrschten, war das Auslösen von Ereignissen.

Die Antwort lag nahe: OLE erweitern. Ein neuer Objekt-Typ wurde definiert und »OLE-Control« (OCX) genannt. Dabei wurde ein Mechanismus festgelegt, über den Controls Ereignisse in der Container-Anwendung auslösen können. Weiterhin wurden einige neue Funktionen zur Performance-Verbesserung und für einige weitere Fähigkeiten hinzugefügt. Daraus ergab sich, daß OLE-Controls nicht bloß mit Visual Basic 4.0 kompatibel waren, sondern nach nur wenigen Modifikationen in nahezu jedem OLE-Container verwendet werden konnten.

Die Aussage, daß die Anwort einfach war, ist vielleicht ein wenig irreführend. Die Implementierung dieser Technologie war komplex und benötigte eine geraume Zeit zur Reife. Sie reift immer noch weiter, aber OLE ist ja von Natur aus erweiterbar. Wenn Sie weiterlesen, werden Sie das Wie und Warum in Erfahrung bringen – weil Sie die gleichen Techniken einsetzen werden, um Ihre eigenen ActiveX-Objekte zu erweitern.

2.3 ActiveX: Technologie oder Marketing?

Fassen wir einmal kurz zusammen.

Wir begannen mit der Betrachtung der Probleme bezüglich komplexer Dokumente in einer anwendungszentrierten Umgebung.

Dann schauten wir uns an, wie eine dokumentenzentrierte Umgebung aussehen könnte und welche Anforderungen an diese gestellt werden müßten.

Wir sahen, wie COM die Implementierung vieler der notwendigen Technologien für eine dokumentenzentrierte Umgebung ermöglicht.

Wir lernten, daß OLE keine einzelne Technologie, sondern ein ganzes Bündel an Technologien darstellt, die kaum etwas miteinander zu tun haben, abgesehen davon, daß sie alle auf COM basieren und auf dokumentenzentrierte Umgebungen abzielen.

Wir sahen, daß Visual Basic im Zusammenspiel mit der VBX-Custom-Control-Technologie zu einer äußerst erfolgreichen Plattform für die komponentenbasierte Software-Entwicklung wurde, jedoch in der Form nicht länger überleben konnte.

Schließlich sahen wir, wie Visual Basic 4.0 auf VBA aufbaute, einer COM-basierten Sprach-Engine, und wie OLE erweitert wurde, um OLE-Controls einzubeziehen, die OLE-basierte Custom-Control-Technologie.

Wie Sie auch erkannt haben und erkennen werden, hat sich die Technologie über Jahre hinweg immer mehr dokumentenzentriert ausgerichtet, und trotzdem befindet sich der Prozeß des Wandels immer noch in einem frühen Stadium.

Bis jetzt habe ich über OLE 2.0 gesprochen. Sie werden sich schon längst gefragt haben, wo denn nun ActiveX ins Spiel kommt. Die Antwort mag Sie vielleicht überraschen. Doch bevor ich fortfahre, möchte ich noch eines klarstellen. Ich bin nicht bei Microsoft angestellt und habe auch, bis auf eine kurze Vertragsarbeit, nie für Microsoft gearbeitet. Vieles von dem, was ich über Microsoft und die Arbeitsweisen dort weiß, beruht auf informellen Gesprächen mit Microsoft-Mitarbeitern, auf veröffentlichten Informationen, auf Berichten in den Medien, auf einem Verständnis der Technologie – und manchmal auch auf ein wenig schierer Spekulation. So habe ich nur wenig Insider-Wissen im Vergleich zu Microsoft-Mitarbeitern, und ich genieße daher weitaus größere Freiheit, meine Meinung zu äußern. Ich brauche mich schließlich an keine »Parteilinie« zu halten. Nachdem ich dies losgeworden bin, hier also die Wahrheit über ActiveX, so wie ich sie kenne.

Erinnern Sie sich an die Zeit Ende 1995 und Anfang 1996. Visual Basic 4.0 erschien, und OLE war auf dem Weg, ein dominierender Standard für Objekt-Einbettung und -Programmierung zu werden – angetrieben von Microsofts Vision eines dokumentenzentrierten Computing.

Zu dieser Zeit begann der Aufstieg des Internet, insbesondere des World Wide Web, sich in hyperbolischen Kurven niederzuschlagen. Ich glaube zwar, daß das meiste, was Sie über das Internet lesen, übertriebener Medien-Hype ist. Einige der Investionen in Internet-bezogene Produkte und Firmen könnten sich als Geldverschwendung herausstellen. Es ist noch immer zu früh, vorauszusagen, was mit dem Internet geschehen wird, welche Arten von Märkten sich entwickeln werden, und welchen Einfluß das alles langfristig auf die Gesellschaft haben wird.

Ich weiß genügend über Microsoft, um sagen zu können, daß dieser Organisation viele extrem helle Köpfe angehören. Trotzdem scheint von außen gesehen deren kollektive Reaktion auf das Internet von nichts anderem als Panik diktiert worden zu sein. Urplötzlich hatte Microsoft eine Strategie, die anscheinend lautete: »Alles, was wir machen, muß eine Internet-Komponente enthalten.« Fürchtete sich Microsoft offensichtlich vor Netscape? Könnte das Internet tatsächlich eine Herausforderung an Microsofts Vision von einem Computer auf jedem Schreibtisch mit Microsoft-Software unter Microsoft-Betriebssystemen sein? Wenn ich das so als deren Vision herausstelle, geschieht das ohne kritische Hintergedanken – es ist einfach, soweit ich das zu sagen weiß, die gültige Vision des Unternehmens.

Erst als ich die Umrisse dieses Buchs entwickelte, wurde es mir klar. Ich glaube nicht, daß sich Microsoft vor der Firma Netscape fürchtete. Man fürchtete vielmehr das gesamte World Wide Web als eigenständige Vision. Sehen Sie, der Trend zu einem Windows-basierten, dokumentenzentrierten Programmier-Paradigma war im Anrollen begriffen – und da platzte das World Wide Web hinein, eine von Natur aus eindeutig dokumentenzentrierte Umgebung. Denn auf HTML-Seiten sind Ansammlungen verschiedenster Objekte, von formatiertem Text über Bilder und Sound bis hin zu Video, nichts Ungewöhnliches. Mit Java können Web-Seiten Code-basierte Objekte enthalten. HTML ist leicht erweiterbar und kann ohne weiteres weitere Objekte aufnehmen. HTML-Dokumente sind leicht zu erzeugen – und das wird noch viel leichter werden, je mehr fortschrittliche Tools dazu verfügbar werden.

In einer Hinsicht repräsentiert das Web eine Überschneidung mit Microsofts Ansinnen. Anwendungsentwickler haben sich gerade erst einmal an die Idee gewöhnt, verschiedene Objekt-Typen in Textdokumente oder Kalkulationstabellen einzufügen, wobei diese Objekte gewöhnlich auf dem gleichen System oder immerhin in einem lokalen Netzwerk beheimatet sind. Web-Programmierer hingegen starten mit einem Grundverständnis, daß Dokumente aus Seiten verschiedenster Typen zusammengefügt werden können und sollten, wobei sich jede beliebige Seite an beliebiger Stelle irgendwo auf der Welt befinden kann. HTML ist beschränkt im Vergleich zur OLE-Technologie, aber es bereitet einem Denken den Weg, der Microsofts Ansinnen bezüglich des dokumentenzentrierten Computing gefährden könnten.

Ich vermute, daß Microsoft deswegen in Panik geriet. Es ging nicht um Netscape. Man fürchtete, daß das World Wide Web (samt HTML, Java und einem nicht

COM-basierten Objekt-Standard) die dominierende Implementierung für dokumentenzentrierte Umgebungen werden könnte.

Folgerichtig wandelte Microsoft den eigenen Standpunkt: Man adoptierte das Internet. Dies war einfacher, als es zunächst schien. OLE war von Natur aus erweiterbar. Es würde nicht viel Aufwand erfordern, ein paar Internet-Erweiterungen einzubauen und OLE zu einem sehr konkurrenzfähigen Mechanismus für die Implementierung von Verbunddokumenten auf dem Internet zu machen, der auch noch voll kompatibel zu HTML wäre.

Aber Technologie ist nicht alles. Erwartungshaltungen zählen genauso. Microsoft brauchte eine Internet-Message. Man mußte der Welt zeigen, daß man es mit dem Internet ernst meinte. Man hatte noch keine Internet-spezifische Technologie, aber man brauchte einen dramaturgischen Effekt, um zu zeigen, daß man diese in Kürze hätte. Dieser Effekt sollte nach einem technischen Fundament klingen, das auf dem Internet seine Lebensberechtigung haben würde.

Genau das machte Microsoft dann auch: Man taufte OLE schlichtweg in ActiveX um.

OLE *ist* ActiveX. So einfach ist das. ActiveX-Automation ist OLE-Automation. Alle OLE-Controls wurden auf einen Schlag zu ActiveX-Controls. Nicht alle ActiveX-Controls sind internetfähig (es gibt einige sehr nützliche Controls, die nichts mit dem Internet zu tun haben). ActiveX-Code-Komponenten sind OLE-Server. Der Name hat sich geändert, aber die Technologie ist dieselbe. ActiveX-Documents sind DocObjects.

Ich kann dies gar nicht oft genug wiederholen. Microsofts Marketing-Team unternahm große Anstrengungen, ActiveX als Internet-Technologie zu promoten. Das war lebenswichtig aus deren Sicht, da Microsoft ActiveX mit aller Macht als dominierende Objekttechnologie im Internet und in Unternehmens-Intranets etablieren wollte. Als Visual-Basic-Programmierer mögen Sie der Meinung sein, daß der Internet-Aspekt von ActiveX für Sie ganz hintenan stehen mag. Und doch mag sich vielleicht die Fähigkeit von Visual Basic, ActiveX-Controls erstellen zu können, als guter Weg zu abwechslungsreichen Web-Seiten herausstellen. Es ist aber auf jeden Fall zum mächtigsten Tool zur Entwicklung von komponentenbasierten Anwendungen geworden. Wenn Sie in den Medien auf Meinungen stoßen sollten, daß Microsofts ActiveX-Strategie fehlgeschlagen sei, dann glauben Sie das, wenn Sie wollen. Interpretieren Sie es aber bitte nicht dahingehend, daß die dahinterstehende Technologie selbst fehlgeschlagen sei. Selbst wenn ActiveX-Controls von sämtlichen Web-Seiten verschwinden sollten, wird ActiveX-Technologie dennoch die dominierende Technologie für dokumenten-zentrierte Anwendungen im Internet bleiben. Angenommen, jedes Bit an ActiveX-Technologie würde von heute auf morgen von unseren Rechnern verschwinden – die allermeisten Anwendungen auf Ihrem System würden unweigerlich den Dienst verweigern, und sogar das Betriebssystem selbst würde seinen Geist aufgeben und nicht mehr booten.

Sie können sichergehen, daß es in diesem Buch auch darum gehen wird, wie Sie ActiveX-Controls und ActiveX-Dokumente (früher hießen diese *DocObjects*) ins Internet bringen können. Doch der Schwerpunkt wird auf der allgemeinen Technologie liegen, und darauf, wie Sie diese Technologie einsetzen, großartige Anwendungen zu erstellen – seien diese nun Internet-basiert oder nicht.

Damit ist die historische Betrachtung abgeschlossen. Von nun an werde ich mich auf die Technologie unter dem Begriff ActiveX beziehen. Sie wissen jedoch, daß ActiveX und OLE ein und dasselbe sind. Sie wissen auch, daß ActiveX nichts weiter ist als ein Bündel von Technologien, die alle auf dem Windows Component Object Model (COM) beruhen. Schauen wir uns nun an, wie dies implementiert ist – am besten auf einem Visual-Basic-Level.