XML Web Service-Anwendungen mit Microsoft .NET

PEARSON Education

ADDISON-WESLEY

Christian Weyer

XML Web Service-Anwendungen mit Microsoft .NET

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

**Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.**

10 9 8 7 6 5 4 3 2 1

05 04 03 02

ISBN 3-8273-1891-2

Einbandgestaltung: atelier für gestaltung, niesner & huber, Wuppertal
Lektorat: Christiane Auf, cauf@pearson.de
Fachlektorat: Dr. Berhard Tritsch, Ober-Ramstadt
Korrektorat: Christina Gibbs
Herstellung: Claudia Bäurle, cbaeurle@pearson.de
CD Mastering: Gregor Kopietz, gkopietz@pearson.de
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Bercker, Kevelaer
Printed in Germany

Inhaltsverzeichnis

D.I.L.Y.A.P.

Vorwort

Nein, ich möchte an dieser Stelle gar nicht viel Worte über das Thema .NET und XML Web Services verlieren. Alle Informationen können Sie in den acht Kapiteln des Buchs nachlesen und anhand der Beispiele in der Praxis ausprobieren.

Vielmehr nutze ich den Platz auf dieser Seite, um mich bei den Personen zu bedanken, die dieses Buch überhaupt erst haben Realität werden lassen.

Allen voran stelle ich Herrn Dr. Bernhard Tritsch, von allen Freunden einfach nur Benny genannt. Nicht nur, dass er als Korrekturleser und kritischer Begutachter der Vorabversionen immer sehr streng, aber durchaus gerecht und gerechtfertigt mit mir umging, sondern vor allem, weil er maßgeblichen Anteil an meinem persönlichen Werdegang hat. Nicht zuletzt ist er ein guter Freund und Kumpel, den man auch mal mit einem Bier weg von der »Computer-Schiene« bringt.

Außerordentliche Erwähnung verdienen auch meine beiden Kumpels Marc Freidhof und Thilo Frotscher. Marc war sehr hartnäckig beim Korrekturlesen und hat bei den Beispielprogrammen genau darauf geachtet, dass sie auch wirklich funktionieren. Mit Thilo habe ich viele anregende Diskussionen vor allem über die neue Welt der XML Web Services geführt. Er ist wohl das, was man einen besten Freund nennt. Seine Meinungen und Ideen sind für mich von unschätzbarem Wert – wir sollten wirklich eine Firma gründen, Thilo.

Ich möchte meinem gesamten Freundeskreis aus Darmstadt und Neustadt und Umgebung danken. Ohne Eure moralische Unterstützung wäre vieles noch schwieriger gewesen. Vor allem die Volleyballer der DJK Karbach waren immer wieder eine Anlaufstation zum »Abschalten«. Kein Buch, kein .NET, keine Web Services. Einfach nur Spaß haben und Volleyball spielen.

Vielen Dank auch an Christiane Auf und ihr Team von Addison-Wesley. Sie hatte sehr viel Geduld mit mir. Wahrscheinlich wurde noch kein Buch so oft nach hinten verschoben, wie das bei mir der Fall war ☺. Es war wirklich eine lange und schwere Geburt …

Meine Familie hatte zeitweise auch sehr mit meinen Launen zu kämpfen. Aber dafür seid Ihr ja da ☺. Ich danke Euch für Eure Unterstützung und Nachsicht.

Alle anderen, die ich hier vergessen (oder ausdrücklich nicht aufgeführt ;-)) habe, mögen es mir nachsehen.

Last but not least: Jones. Du bist lange Zeit mein fachlicher Mentor gewesen und hast sehr viel zu diesem Buch beigetragen. Danke.

So, nun aber genug der warmen Worte. Stürzen wir uns auf die Arbeit.

Christian Weyer, im April 2002
christian.weyer@addison-wesley.de

Einleitung

Mit den folgenden Zeilen möchte ich Ihnen die Intention und den Inhalt des vorliegenden Buchs etwas näher bringen. Zudem weise ich auf die verfügbaren Programmbeispiele, notwendige und hilfreiche Software zur Programmierung und die Website zum Buch hin.

Über dieses Buch

Dieses Buch nimmt sich Microsoft .NET und die Thematik rund um XML Web Services als Inhalt zum Ziel. Sie werden auf den nächsten knapp 320 Seiten erfahren, was .NET ist, warum man es in bestimmten Projektsituationen benötigt, und wie man es für Internet- und Intranet-Anwendungen einsetzt. Dabei wird der Blick ständig auf eines der neuen und interessanten Programmierparadigmen der aktuellen Softwareentwicklung gerichtet sein: XML Web Services.

Allgemeines

.NET und XML Web Services gehören zusammen wie Pfeffer und Salz. Allerdings sind nicht die XML Web Services die große Neuerung in .NET. Auch dieses oft vorherrschende Missverständnis wird in diesem Buch geklärt. Vielmehr hat Microsoft das .NET Framework und alle dazugehörigen Technologien immer mit dem Hintergedanken im Kopf entwickelt, eine Architektur zu schaffen und Entwicklern an die Hand zu geben, welche die Erstellung moderner Applikationen ermöglicht. Und ein wichtiges Teil im großen .NET-Puzzle sind nun mal die XML Web Services.

In diesem Buch werden Sie keine ausführlichen Abhandlungen über Windows DNA, COM, DCOM, COM+ und alle damit zusammenhängenden Technologien und Produkte finden. Ich gehe also davon aus, dass Sie ein Entwickler oder Manager sind, der bereits erste Erfahrungen mit der Softwareentwicklung in diesem Bereich besitzt. Nichtsdestotrotz wird das Grundwissen in diesem Gebieten im ersten Kapitel aufgefrischt, damit alle Leser einen annähernd gleichen Wissensstand für die verbleibenden Kapitel haben.

Wenn man ein Buch über .NET schreibt, besteht immer die Gefahr zwischen die Fronten der Programmiersprachen zu geraten. Die einen bevorzugen C#, die anderen fühlen sich mit der Anwesenheit von Visual Basic .NET-Quelltext sehr wohl. In werde mich dieser teilweise etwas müßigen Diskussion nicht

anschließen. Die Beispiele im Buch und auf der Begleit-CD sind teils in C#, teils in Visual Basic .NET erstellt worden. Eine Portierung von der einen in die andere Sprache ist zum einen sehr einfach und zum anderen eine hervorragende Übung, um beide Sprachen zu lernen und .NET besser zu verstehen.

Die Beispiele und die Vorgehensweisen beruhen in den meisten Fällen auf dem frei verfügbaren .NET Framework SDK. Mit dieser Sammlung an kommandozeilenbasierten Werkzeugen kann man komplette .NET-Anwendungen erstellen, ohne sich eine teure Lizenz für eine Entwicklungsumgebung anschaffen zu müssen. Jedoch wird auch das Visual Studio .NET in den Bereichen mit einbezogen, wo es die Programmierung in .NET erheblich vereinfacht. Beispiele sind hier die Erstellung von Web-basierten grafischen Benutzungsoberflächen oder XML Web Services. Selbstverständlich beruhen alle Informationen und Daten auf der endgültigen deutschen Version von .NET 1.0.

Kapitelübersicht

Damit Sie einen ersten Grobüberblick über den Inhalt dieses Buchs erhalten, finden Sie im Folgenden eine kompakte Beschreibung der jeweiligen Kapitelinhalte.

Kapitel 1:

Um die Intention hinter .NET und hinter XML Web Services verstehen zu können, muss man sich die Entwicklung des Internets und der Softwareentwicklung in den letzten Jahren etwas genauer ansehen. In diesem Kapitel werden die unterschiedlichen Probleme aktueller Softwarearchitekturen beleuchtet. Zudem versuche ich, die neuen und teilweise noch nicht absehbaren Anforderungen an moderne Software durch die Weiterentwicklung von Internet und Intranet mit einzubeziehen.

Kapitel 2:

Ohne Handwerkszeug kann kein Handwerker arbeiten. Und ohne die Grundlagen von .NET kann ein Entwickler keine .NET-Programme erstellen und verstehen. Deshalb stelle ich Ihnen in diesem Kapitel die wesentlichen Neuerungen der .NET-Umgebung vor und versuche an den entsprechenden Stellen, einen Vergleich zu der bisherigen Softwareentwicklung unter Windows zu ziehen.

Kapitel 3:

Web-basierte Anwendungen sind in aller Munde. ASP.NET steht unter .NET eine neue und sehr leistungsfähige Technologie zur Verfügung, um grafische Benutzungsoberflächen für Web-Programme zu erstellen. Sie werden lernen, wie ASP.NET funktioniert und wie man ASP.NET-Seiten erstellt. Es wird auch in diesem Kapitel immer wieder auf die Unterschiede und vor allem die Verbesserungen gegenüber den Vorgängerversionen hingewiesen.

Kapitel 4:

Ohne Daten keine Anwendung. So einfach kann man es formulieren. In diesem Kapitel zeige ich Ihnen, wie man mit ADO.NET auf Datenbanken zugreift: Auslesen, Einfügen, Aktualisieren. Zusätzlich bringe ich noch die wichtigsten Operationen bei der Bearbeitung von XML-Dokumenten ins Spiel. XML wird immer wichtiger und ist vor allem in Web-Applikationen ein gerne verwendetes Mittel zur Speicherung von Daten.

Kapitel 5:

XML, SOAP & Co. werden Ihnen in diesem Kapitel ständig begegnen. Diese Basistechnologien für XML Web Services stelle ich Ihnen vor und spanne den Bogen zur Implementierung und Nutzung von XML-basierten Web Services mit ASP.NET. Denn ASP.NET kann nicht nur für die Erstellung von grafischen Oberflächen herangezogen werden. Es eignet sich auch sehr gut zur Bereitstellung von SOAP-basierten Schnittstellen für eigene Programme.

Kapitel 6:

Mit den Grundlagen aus Kapitel 5 können Sie hier Techniken und Vorgehensweisen kennenlernen, um XML Web Services in Ihrer Anwendungsarchitektur einzusetzen. Wichtige Themen wie Zusammenarbeit mit existierenden COM- und COM+-Anwendungen, Übertragung von komplexen Daten oder aber die Optimierung von Web Services zur schnellen Bereitstellung von Daten sind Gegenstand dieses Kapitels.

Kapitel 7:

Ohne das Thema Sicherheit kommt heutzutage kein Buch mehr aus, das über das Internet und das WWW als Medium für die Anwendungsentwicklung schreibt. Ich werde Ihnen hier die grundlegenden Neuerungen im Bereich Sicherheit aus dem .NET Framework näher bringen. Der Schwerpunkt liegt allerdings auf der Absicherung von ASP.NET-Anwendungen und vor allem von XML Web Services.

Kapitel 8:

Unter anderem die Probleme bei der Programmierung und dem Betrieb verteilter Anwendungen mit DCOM haben zur Entwicklung von .NET Remoting geführt. Remoting ist eine in die Laufzeitumgebung integrierte Technologie, um verteilte Applikationen unter .NET mit einem hohen Maß an Konfigurierbarkeit und Erweiterbarkeit zu erstellen. Nicht zuletzt lassen sich auch mit der Remoting-Technologie XML-basierte Web Services realisieren.

Die CD zum Buch

Diesem Buch liegt eine Begleit-CD bei. Auf ihr finden Sie sowohl die Programmbeispiele aus den einzelnen Kapiteln als auch wichtige und sehr nützliche Anwendungen rund um die .NET- und XML Web Services-Entwicklung.

Programmbeispiele

Es ist mühsam und ungemein fehlerträchtig, Quelltext aus einem Buch abzutippen. Aus diesem Grund sind die Beispiel-Listings auf der beiligenden CD zu finden. Dies hat auch den Vorteil, dass tatsächlich alle Programme in kompletter Form vorliegen, denn im Buch ist an einigen Stellen nur der relevante Teil im Listing abgedruckt.

Für eine erfolgreiche Einrichtung der Betriebsumgebung lesen Sie bitte die auf der CD befindliche Datei *Installation.html*. Zum Starten der Beispiele können Sie nach erfolgter Installation die Seite *Index.html* in einem Web-Browser aufrufen.

Software

Damit Sie als Leser direkt mit der .NET-Programmierung loslegen können, habe ich auf der CD alle notwendigen Werkzeuge, SDKs und weitere hilfreiche Anwendungen für Sie zusammengestellt.

Den genauen Inhalt und eine Beschreibung der Programme entnehmen Sie der Datei *Software.html*.

Die Website zum Buch

Niemand ist perfekt, niemand ist unfehlbar – und ein Buchautor und Softwareentwickler schon gleich gar nicht. Wenn Sie inhaltliche bzw. fachliche Fehler, Ungereimtheiten oder Druckfehler im Buch finden, können Sie diese gerne auf der Website zum Buch eintragen – ich werde mich dann sofort darum kümmern. Natürlich stehe ich in dort auch gerne für Fragen rund um die Themen und Problematiken in diesem Buch zur Verfügung.

Sie können die Website unter folgender Adresse finden: *http://www.eyesoft.de/wsbuch/*.

In einem speziellen Bereich auf dieser Website werden ständig aktualisierte Beispiele und *FAQs* (*Frequently Asked Questions*, regelmäßig gestellte Fragen) für die .NET- und XML Web Service-Programmierung eingepflegt.

Nun wünsche ich Ihnen viel Spaß mit diesem Buch – und denken Sie bitte immer daran: Es ist nur Software!

Dieses Icon enthält einen Tipp und weist Sie auf Techniken hin, mit denen Sie sich das Leben leichter machen und Zeit sparen können.

Hier finden Sie Hinweise auf mögliche Probleme und Fallen. Diese Information hilft Ihnen, bekannte Fehler zu vermeiden - und kann Ihnen ebenfalls viel Zeit sparen.

Das Achtung-Icon enthält Warnungen. Wenn Sie diese Warnung in den Wind schlagen, können Daten verloren gehen, es kann nicht-lauffähiger Code entstehen oder Schlimmeres passieren.

Dieses Symbol weist auf wichtige, weiterführende Information hin, die Sie im Internet finden.

1 Wieso weshalb warum?

Einführung und Motivation

1.1 Überblick

Wieso, weshalb, warum? Diese Frage werden sich einige hundert Tausend bis Millionen von Softwareentwicklern gestellt haben. »*Microsoft .NET* – was ist das, was soll das und wieso brauche ich das?« Wieder nur Marketing, wieder nur ein völlig überzogener Hype, und vielleicht wieder nur leere Worthülsen? Ich möchte Ihnen in diesem ersten Kapitel einen kleinen Überblick über die Entstehung von .NET geben, quasi eine Art Evolutionsgeschichte der Softwareentwicklung im Microsoft-Umfeld. Dieser Überblick soll uns dann anschließend helfen zu verstehen, was .NET wirklich ist und wie man es in seinen Projekten und Produkten einsetzen kann.

1.2 Einführung und Motivation

Wenn man in einem Buch über .NET schreibt, kann man nicht einfach mit der Erklärung beginnen, was .NET ist und wie es programmiert wird. In diesem Fall sind einige einführende Erläuterungen und Erklärungen notwendig, die ein Bild vermitteln sollen, wie es zur Entwicklung von .NET gekommen ist. Schließlich bedeutet .NET teilweise eine Umstellung der Denkweise und der Art und Weise wie man Software entwickelt. Doch dazu später mehr.

1.2.1 Problematik und Hintergrund

Umbruchstimmung. Das ist vielleicht der beste Begriff, der die aktuelle Lage im Internet und der damit verbundenen Softwareentwicklung beschreibt. Die in den letzten Jahren kennen gelernte Art und Weise, mit dem Internet umzugehen und es zu benutzen, verändert sich immer mehr. Während in den Anfangsjahren des *World Wide Web* (*WWW*) dieses Medium hauptsächlich zum Spielen und Herumstöbern verwendet wurde, wandelt sich das Verhalten der Benutzer immer mehr in eine kommerzielle Richtung. Der Spielcharakter muss dem Geschäftstrieb weichen. Das als *E-Commerce* bekannt gewordene Abwickeln von Bestell- und Bezahlvorgängen im Internet und primär im WWW ist eine der großen Chancen des Mediums, sich technisch in jeder Ecke und jedem Winkel des menschlichen Lebens zu etablieren. Das Internet als die »Überall-und-alles-Technologie«.

Mit dem Aufkommen des E-Commerce hat sich auch die zugrunde liegende Technologie und deren Anwendung verändert. Reichten anfangs rein statische HTML-Seiten aus, müssen es nun schon dynamisch erzeugte Inhalte sein. Denn mit von Menschen manuell erstellten und gepflegten HTML-Seiten kommt man in einem Szenario für E-Commerce nicht sehr weit. Abhilfe schaffen hier Server-seitige Mechanismen, welche die Erzeugung von HTML-Ausgaben dynamisch ermöglichen. Basierend auf externen Daten, z.B. aus einer Datenbank, einem Mail-Server oder aus XML-Dateien, wird die resultierende HTML-Seite auf der Server-Seite regelrecht zusammengebaut. Allerdings beschränkt sich die Ausgabe nicht allein auf HTML: Es können beliebige Antworten generiert werden, so auch binäre Informationen, wie sie in Bildern enthalten sind oder etwa reines XML. Eingesetzte Technologien in diesem Umfeld sind beispielweise die *Active Server Pages* (*ASP*) von Microsoft oder die *Java Server Pages* (*JSP*) aus dem Java-Lager. Diese beiden und alle verwandten Techniken ermöglichen die Programmierung einfacher Anwendungen auf der Server-Seite. Mithilfe von Skript-Code werden die notwendigen Berechnungen und Auswertungen kodiert, eventuell auf externe Datenquellen zugegriffen und die Ergebnisse dann an die gewünschte Stelle der Ausgabe eingesetzt. Die Vorteile aber auch die Probleme dieser Vorgehensweise und Technologien werden weiter unten in diesem Kapitel nochmals aufgegriffen.

Zusätzlich wurden die Benutzungsoberflächen im Browser komfortabler und vielseitiger. Anstatt nur reines HTML zu verwenden, kommen nun dynamische Elemente auch auf dem Client zum Einsatz. Mit *JavaScript* lassen sich bestimmte Eingaben bereits auf dem Client überprüfen und mit *Java-Applet*s und *ActiveX*-Control lassen sich sogar komplexe Benutzungsoberflächen mit hohem Komfort erstellen. Allerdings ist bei diesem Thema immer wieder die Browser-Kompatibilität der wunde Punkt. Nicht alle auf dem Markt befindlichen Browser kennen jeweils die Eigenschaften der Konkurrenz.

Mit der Zeit haben sich derart konzipierte Applikationen etabliert, wenn auch noch viele Kinderkrankheiten den richtig großen Erfolg verhindern. Durch den wachsenden Anteil von E-Commerce über das Internet haben sich wiederum neue Rahmenbedingungen ergeben. Das Angebot und die Ausprägung der Transaktionen werden immer vielschichtiger. Wenn ein Schuhhersteller beispielsweise seine Herrenschuhe online verkaufen möchte, stellt dies keine großen Probleme dar. Wenn dieser Hersteller nun allerdings sein Produkt- und Dienstleistungsangebot erweitern, und auch noch die passenden Socken zu den Schuhen anbieten möchte, dann ergeben sich verschiedene Möglichkeiten. Er kann sein Bestellsystem so erweitern, dass er alles selbst abwickelt oder er kann bereits existierende Hersteller bzw. Händler mit in seinen Online-Shop einbeziehen. Diese externen Partner haben meist eine ausgeprägte Kompetenz in ihren Bereichen und eine Kooperation ist oft sinnvoller und auch kostengünstiger als ein kompletter Neuanfang auf eigenen Füßen. Wie kooperiert nun der Schuhhersteller mit dem Sockenhersteller? Die Homepage des Sockenfabrikanten als Basis für eine softwaretechnische Integration heranzuziehen ist keine gute Lösung: Die in HTML verfassten

Seiten bieten keinerlei Struktur und Metainformationen, die man in einer standardisierten Form mechanisch bearbeiten könnte. Für solche Zwecke bietet die *eXtensible Markup Language (XML)* ihre Dienste an. XML bietet strukturierte Informationen und Daten, die man mit entsprechenden Parsern leicht verarbeiten kann. Deshalb sollte eine Integrationslösung für unser Problem auf jeden Fall auf XML aufsetzen, welches über geeignete Kommunikationsprotokolle zwischen den teilnehmenden Partnern verschickt wird. In diesem Zusammenhang werden wir noch des öfteren den Begriff *Web Services* lesen und hören (siehe auch Abschnitt 1.5.2).

Wir sehen also momentan eine Entwicklung weg von der Mensch-zu-Software-Kommunikation, hin zur Software-zu-Software-Kommunikation, welche lediglich eine andere Sichtweise für *Business-to-Business* (*B2B*) darstellt. Doch mit den bisher eingesetzten Techniken und Vorgehensweisen werden uns einige Steine auf dem Weg zum Ziel begegnen.

Die eben aufgezeigten Szenarien und ihre Probleme fordern nach einem Umdenken in der Softwareentwicklung, speziell im Bereich Web-basierte Anwendungen, um moderne Internet-Anwendungen entwickeln zu können. Mit den heute üblichen und weiter unten kurz vorgestellten Techniken stößt ein Entwickler recht schnell an seine Grenzen. Der .NET-Ansatz von Microsoft greift hier an und versucht, möglichst viele Probleme und Grenzen abzuschaffen.

1.2.2 Warum dieses Buch?

Ich habe dieses Buch verfasst, um zum einen den Nebel um .NET herum etwas zu vertreiben und zum anderen eine praxisorientierte Einführung in die Web-basierte Programmierung mit Microsoft .NET zu geben. Dabei sollen die oben erläuterten Probleme nicht aus dem Blickfeld geraten und das zentrale Konzept der Web Services in .NET näher beleuchtet und softwaretechnisch erläutert und implementiert werden.

Bevor wir jedoch über .NET reden können, möchte ich, wie bereits oben angekündigt, einen kurzen Abriss über die Entstehung von Internet-basierten Applikationen allgemein und auf der Microsoft-Plattform im Speziellen geben. Dabei sollen auch die im Laufe der Zeit aufgetretenen Probleme extrahiert und beleuchtet werden.

1.3 Internet-Anwendungen

Wir alle wissen, dass es das Internet nicht erst seit kurzem gibt. Das ehemalige *ARPA-Net* ist mittlerweile gut dreißig Jahre alt. Allerdings hat sich sein Aussehen in den vergangenen Jahren stark verändert. Durch das WWW hat sich das Internet bis in die Unternehmen und sogar in die privaten Haushalte ausgebreitet. Und ein Ende dieser Entwicklung ist noch lange nicht abzusehen. In ein paar Jahren wird man das Internet als Medium ebenso verinnerlicht haben, wie das heute mit dem Telefon oder dem Fernsehen der Fall ist.

Auch die Softwareentwicklung hat parallel zum Werdegang des Internets eine sichtliche Veränderung durchlaufen. In den Anfangsjahren konzentrierten sich die Programmierer vornehmlich darauf, die Bits und Bytes effizient zu verpacken, um sie so schnell wie möglich zwischen den einzelnen Teilnehmern in einem Netzwerk austauschen zu können. In dieser Zeit wurden die Netzwerkprotokolle und die zugehörigen Softwareimplementierungen, die so genannten *Protokoll-Stacks*, geboren. Mit zunehmender Qualität der zugrunde liegenden Software und der Erfahrung der Entwickler konzentrierte man sich mehr und mehr auf die eigentliche Sache: Die Implementierung von verteilten Anwendungen in einem internen Netzwerk (*Intranet*) oder im Internet. Doch mussten die Programmierer über die einzelnen in der Entwicklung involvierten Elemente Bescheid wissen. Angefangen von der zu programmierenden Logik und dem Verhalten der Software, über die Programmierung von Betriebssystemressourcen bis hin zu den verschiedenen Protokoll-Stacks; ja manchmal musste sogar relativ tief in die Hardware über entsprechende Treiber eingegriffen werden. Alles in allem verbrachte ein Programmierer zu viel Zeit damit, sich mit dem Handwerkszeug zu beschäftigen, d.h. der bereitzustellenden Infrastruktur für verteilte Anwendungen, als dass er sich um die eigentliche Programmierlogik kümmern konnte.

Ein entsprechendes Konzept zur Abschirmung des Entwicklers vor immer wiederkehrenden Abläufen und Vorgehensweisen, die auch nicht immer bis ins kleinste Details einzusehen sein mussten, war gefragt. Das war die Geburt des Begriffs *Middleware*. Wie in Abbildung 1.1 zu sehen ist, abstrahiert eine Middleware den Zugriff eines Programms auf die tieferliegenden Schichten. Der Programmierer muss sich also primär nur noch mit dieser Middleware beschäftigen.

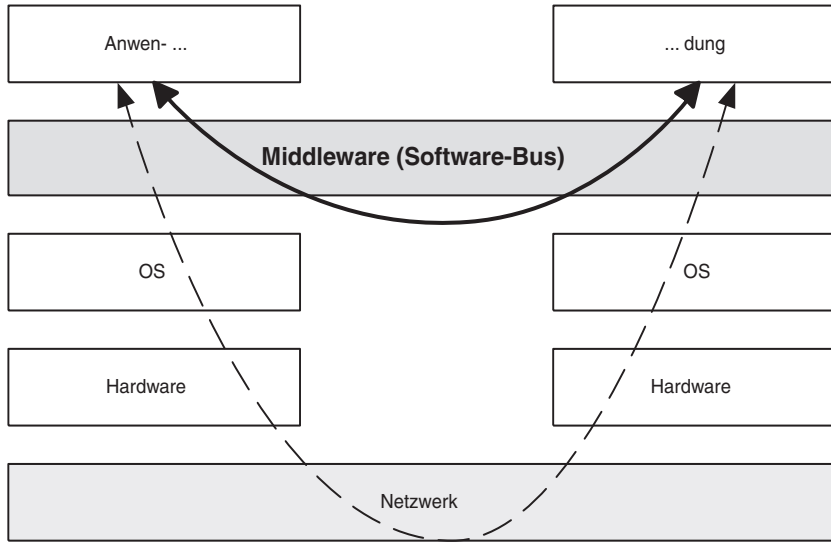


Abbildung 1.1: Das Abstraktionskonzept einer Middleware

Es gibt verschiedene Anbieter von Middleware-Technologien. Einer der bekanntesten ist Microsoft mit seiner *COM/DCOM*-Infrastruktur. Andere Technologien sind weitestgehend herstellerunabhängig. Hierzu gehören *CORBA* (*Common Object Request Broker*) und *EJB* (*Enterprise Java Beans*). Diese beiden Vertreter liegen in Form einer Spezifikation vor. EJB als Spezifikation der Java-Community mit Sun an der Spitze, und CORBA als Spezifikation der *OMG* (*Object Management Group*) als herstellerübergreifendes Konsortium. Die Hersteller von *EJB Application Servern* bzw. *CORBA ORBs* interpretieren die Spezifikation jeweils anders; daher treten in der Praxis auch große Interoperabilitätsprobleme in diesen Bereichen auf. COM/DCOM ist da anders. Microsoft liefert eine Spezifikation und die Referenzumsetzung auf Basis von Windows gleich mit dazu. Zudem ist COM nicht nur ein Middelware-Konzept, sondern stellt auch ein sehr leistungsstarkes Komponentenmodell zur Verfügung. Mehr hierüber können Sie im Abschnitt 1.4.2 nachlesen.

Diese klassischen Vertreter des Middleware-Konzepts wie CORBA sind prädestiniert für den Einsatz im Intranet. Hier gilt es meistens keine besonderen Firewallaspekte oder Interoperabilitätsprobleme zu beachten und die Kommunikationsteilnehmer sind eigentlich immer erreichbar. Allerdings treten sehr große Probleme beim Einsatz dieser Technologien im Internet auf. Hier bedarf es anderer Lösungen.

1.3.1 Client-Server-Modell

Im Folgenden werden die Begriffe Internet und WWW für unsere Zwecke als gleichberechtigt behandelt.

Für die Kommunikation von Partnern im Internet galt lange Zeit das Client-Server-Modell oder 2-Schichten-Modell. Bezeichnenderweise kommunizierte eine Client-Anwendung, wie etwa der WWW-Browser, mit einer Server-Anwendung, etwa einem HTTP-Server. Diese einfache Kommunikationsbeziehung war völlig ausreichend für das Bereitstellen von statischen Informationen wie HTML-Seiten.

Als sich dann aber der Trend zu dynamischen Web-Anwendungen hin entwickelte, bekam man sehr schnell die Probleme dieses Ansatzes zu spüren. Die notwendige Anwendungslogik zur Bearbeitung der Ein- und zur Berechnung der Ausgaben musste einen geeigneten Platz in dieser Architektur finden. Also wurden zwei Ansätze verfolgt: Zum einen, so viel Logik wie möglich auf den Client zu packen, zum anderen, sämtlichen Programm-Code auf die Server-Seite zu verlagern. Vergleichen Sie hierzu auch die ersten beiden Skizzen in Abbildung 1.2.

Im ersten Fall hat man eine relativ gute Skalierbarkeit (siehe Kasten), da die Logik lokal am Client abgearbeitet wird und der Server als dummer Datenlieferant dient. Probleme kommen hier jedoch relativ schnell zum Vorschein: Die Client-Seite muss nicht zwingend ein WWW-Browser sein.

Im Gegenteil sind viele Anwendungen vor allem in Intranets immer noch mit so genannten *Fat Clients* oder *Rich Clients* ausgestattet. Ein Fat Client bezeichnet eine voll ausgestattete und voll funktionsfähige Anwendung, die unter Windows beispielsweise mit Visual Basic oder mit Visual C++ erstellt wurde. Diese Anwendungen erfordern eine lokale Installation und Konfiguration. Eine anfallende Aktualisierung der Anwendung bedeutet also einen immensen Aufwand, wenn man bedenkt, dass leicht eine Benutzeranzahl von mehreren Dutzend bis zu vielen Tausend erreicht werden kann. Weitere Schwierigkeiten treten auf, wenn auf die zentralen Daten auf dem Server gleichzeitig zugegriffen wird. Dabei kann es zu Aktualisierungsproblemen im Datenbestand kommen, die dann unter Umständen in der Client-seitigen Logik nicht beachtet werden können.

Alles in allem ist hier ein Web-basierter Ansatz zu bevorzugen. Hier arbeitet der Benutzer mit einem *Thin Client*, also einer auf dem WWW-Browser basierenden Benutzungsoberfläche. Es ist keine lokale Installation bzw. Konfiguration notwendig. Die anderen beschriebenen Probleme bekommt man aber so noch nicht in den Griff.

Der Begriff *Thin Client* wird hier in einem anderen Kontext verwendet als beispielsweise beim *Thin Client Computing*. Dort ist ein Thin Client eine spezielle Hardware mit zugehörigem Betriebssystem – etwa Windows CE – die auf einen oder mehrere Anwendungs-Server über das Netzwerk zugreifen. Sinn ist die Verringerung der Betriebskosten in einem Unternehmen mit vielen Computer-Arbeitsplätzen.

Im zweiten Fall übernimmt der Server mit einem Verbund von Anwendungs- und Datenzugriffslogik die Hauptaufgabe. Hier besteht die Problematik darin, dass keine klare Trennung zwischen der eigentlichen Programmlogik und dem Datenzugriff besteht. So ist eine Verteilung für den Zweck einer Lastverteilung gar nicht oder nur sehr schwer möglich, was unter Umständen auf Kosten der Skalierbarkeit geht. Eine Skalierung ist nur horizontal möglich: Man kann also den Server mehrfach replizieren und somit mehrere gleichwertige Server mit kombinierter Anwendungs- und Datenlogik bereit stellen. Allerdings ist es in diesem Modell nicht möglich, auch vertikal zu skalieren: Man kann nicht die Datenhaltungslogik von den eigentlichen Geschäftsprozessen in der Anwendungslogik trennen.

Alles in allem hat sich ein weiterführendender Ansatz durchgesetzt, welcher mit dem Begriff *n-Schichten-Modell* tituliert wird.

Das Missverständnis: Skalierbarkeit und Geschwindigkeit

Wenn man im Zusammenhang von verteilten Anwendungen von Skalierbarkeit spricht, werden immer wieder falsche Vorstellungen an den Tag gebracht. Ich will hier versuchen, etwas Licht in diese Grauzone zu bringen:

Bei Geschwindigkeit spricht man häufig auch von empfundener Geschwindigkeit. Ein Beispiel hierfür wäre folgendes Szenario: eine Anwendung ist schnell (für das Empfinden des Benutzers), wenn eine Verarbeitung der Daten mitsamt einer Antwort des Systems in einer angemessenen Zeit erfolgt. Diese Zeit ist abhängig vom Ermessen des Benutzers. Natürlich kann man auch eine Statistik mit den unterschiedlichen Zugriffszeiten erstellen, aber eine Anwendung ist umso schneller, je weniger der Benutzer auf sie warten muss.

Skalierbarkeit hingegen hat eine andere Bedeutung. Ein System bzw. eine Anwendung heißt skalierbar, wenn das Verhalten bei unterschiedlicher Benutzerzahl annähernd gleich bleibt. Wenn also ein einziger Benutzer mit der Anwendung arbeitet, soll sich diese in ihrer Antwortzeit genauso verhalten, wie wenn mehrere hundert oder tausend Benutzer auf sie zugreifen (bzw. soll die Antwortzeit nicht mehr als linear ansteigen, unter der Voraussetzung, dass das System entsprechend erweitert wird).

Daher bedeutet eine hohe Skalierbarkeit einer Anwendung nicht automatisch, dass die Anwendung auch schnell ist!

1.3.2 N-Schichten-Modell

Bei einem n-Schichten-Modell wird die Software schon mit Beginn der Entwurfsphase derart gestaltet, dass die unterschiedlichen Zuständigkeiten in separaten Modulen implementiert werden. Der Begriff Schicht bezeichnet also primär eine logische Einheit und muss sich nicht zwangsweise auf physikalisch unterschiedliche Rechner beziehen. Wie in Abbildung 1.2 zu ersehen ist, lassen sich diese Schichten dann im Bedarfsfall (in der Abbildung »3-Schichten: WWW-Modell«) auf verschiedene Rechner verteilen. Dieses Modell ist als Favorit für moderne, Web-basierte Anwendungen anzusehen, da es entwurfstechnisch die größtmögliche Skalierbarkeit bedeutet und der verbundene Pflegeaufwand am geringsten ist. Zudem ist es die optimale Vorgehensweise für größere Projekte, da man die einzelnen Applikationsbestandteile gut in einem Entwicklerteam verteilen kann.

Was uns jetzt noch zu unserem Glück – sprich einer so weit wie möglich auf die Bedürfnisse verteilter Internet-Anwendungen abgestimmten Architektur und Vorgehensweise – fehlt, sind die entsprechenden Techniken und Standards. Denn wie in vorherigen Abschnitten bereits angedeutet, sind die klassischen Middleware-Technologien aus dem Intranet für solche Anwendungen nicht geeignet.

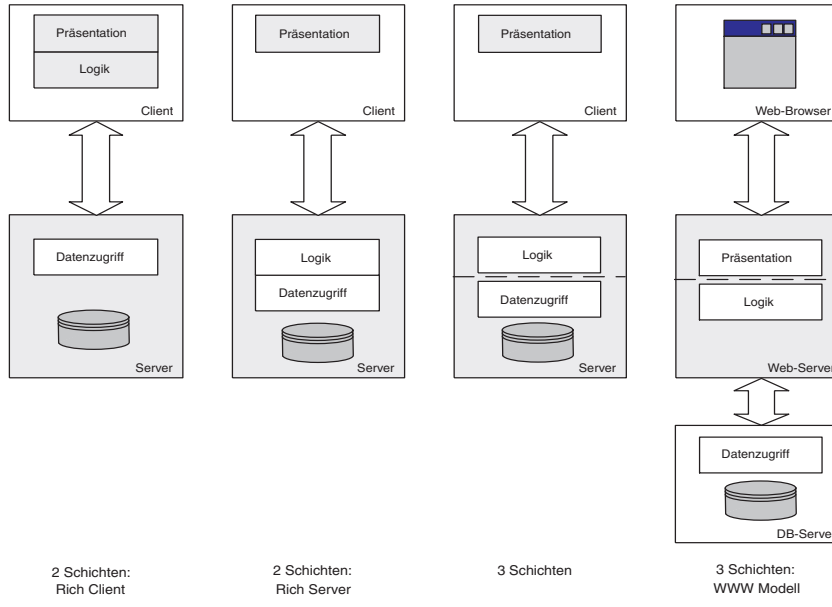


Abbildung 1.2: Schichtenmodelle im Überblick

1.4 Komponenten-basierte Anwendungen mit Windows heute

In diesem Abschnitt möchte ich mit dem Begriff und Konzept der Komponenten aufräumen und aufzeigen, wozu Softwarekomponenten in der Lage sind. Es wird dabei klar gestellt, warum Komponenten-basierter Entwurf und Implementierung zweckmäßige Ansätze für verteilte Anwendungen sind. Der Schwerpunkt liegt hierbei selbstredend auf den Komponententechnologien unter Windows.

1.4.1 Softwarekomponenten

Kinder haben es am einfachsten, ein Spielzeughaus oder eine Burg zu bauen, wenn sie vorgefertigte kleine Bausteine dafür zur Verfügung haben. So können sie aus Holz oder Kunststoff ihre Vorstellungen verwirklichen und in greifbare Resultate umsetzen. Ähnlich ist es auch in der Softwareentwicklung. Mittlerweile tendieren immer mehr Projektverantwortliche und Programmierer dazu, ihre Software nach einem Baukastenprinzip zu entwerfen und zu implementieren. Die grundlegenden Bauteile für diese Software sind die so genannten Softwarekomponenten, oder kurz Komponenten. Doch wie kam es zu diesem Umstand und wo liegen die Vorteile in dieser Vorgehensweise?

Komponenten bringt man mittlerweile primär mit verteilten Internet-Anwendungen in Verbindung. Doch wurde die Idee für Komponenten bereits viel früher geboren. Über die Jahre hinweg haben sich zahlreiche Techniken mehr oder weniger bewährt, um große, verteilte Anwendungen mit einem vernünftigen Aufwand erstellen und pflegen zu können. Ein großes Problem in größeren Anwendungen ist vor allem die Wartbarkeit. Zum einen die Wartbarkeit innerhalb des Entwicklerteams und zum anderen die Wartbarkeit einer Anwendung in Bezug auf Fehlerbereinigung und Hinzufügen neuer Funktionalität. Eine mögliche Lösung für Windows-Applikationen ist die Verwendung von *Dynamic Link Libraries* (*DLLs*). DLLs sind binäre Code-Einheiten, die zur Laufzeit in den aufrufenden Prozess geladen werden. DLLs erfreuen sich einer großen Beliebtheit unter den Windows-Programmierern, bringen jedoch einige Probleme. Die Auflösung der Funktionsaufrufe geschieht über eine einfache Namenszuordnung. Wenn nun eine neue Version einer DLL verabschiedet und für die Anwendung installiert wird, kann es sehr leicht der Fall sein, dass die aufrufende Anwendung den vorher bekannten Einstiegspunkt nicht mehr findet. Noch schlimmer ist, dass DLLs in einer einfachen und komfortablen Art und Weise nur in C bzw. C++ eingebunden werden können. Skriptsprachen wie JavaScript können damit nichts anfangen. Aber die Verwendung von DLLs war ein Schritt in die richtige Richtung. Denn ein statisches Linken mit Bibliotheken zur Kompilierzeit ist nur bedingt eine Lösung für das beschriebene Problem der Wartbarkeit.

Einhergehend mit der Wartbarkeit ist auch der Wunsch nach der Umsetzung der Objektorientierungs (OOP)-Paradigmen zu sehen. Eines der maßgeblichen Prinzipien in der OOP ist das der Kapselung. Dies bedeutet, dass ein Benutzer eines Objektes niemals Details über die interne Implementierung eines Objektes erfährt. Der Benutzer hat lediglich Zugriff über wohldefinierte Methoden, welche ihrerseits den internen Zustand des Objektes manipulieren. Ein Objekt kann also als schwarze Kiste (*black box*) angesehen werden: Der Benutzer kann nicht hinein schauen, bekommt aber das geliefert, was er sich wünscht. Diese Wunschvorstellung mag in der Theorie gut klingen, konnte jedoch in der Praxis selten so umgesetzt werden. Denn wer schon einmal mit größeren, objektorientierten Klassenbibliotheken entwickelt hat, wird viele Probleme kennen. Ein MFC-Programmierer hat für bestimmte Aufgaben immer wieder einen Blick in die Implementierung der jeweiligen Klassen werfen müssen. Manchmal mit dem Ergebnis, dass diese Implementierung so nicht für ihn einsetzbar ist. Also war eine Ableitung dieser Klasse, oder gar ein Neuschreiben, angesagt. Das Konzept der Kapselung und der Vererbung in der OOP kann über eine quelltextbasierte Realisierung nur sehr schwer verwirklicht werden. Zu sehr unterliegt die Implementierung den Änderungen eines Programmierers.

Der größte Vorteil von Softwarekomponenten stellt allerdings die erhöhte Wiederverwendbarkeit dar. Wenn ein bestimmter Algorithmus oder ein Geschäftsprozess in Form einer Komponente realisiert ist, dann kann diese Komponente in den unterschiedlichsten Szenarien mit den unterschiedlichsten Programmierumgebungen eingesetzt werden – vorausgesetzt ein speziell dafür entwickeltes und abgestimmtes Komponentenmodell ist vorhanden.

Im nächsten Abschnitt werde ich Microsofts Lösung zum Problem der wartbaren, wiederverwendbaren Software vorstellen. Zusätzlich werde ich gleich noch einen Schritt weiter gehen und die notwendige Weiterentwicklung bzw. die wichtigen Erweiterungen dieses Modells für den Einsatz in verteilten Anwendungen beschreiben. Die Darstellung wird nicht zu sehr ins Detail gehen, soll aber einen einheitlichen Wissensstand für die Leser schaffen und den Gebrauch eines gemeinsamen Jargons gewährleisten.

1.4.2 COM, DCOM, COM+ im Überblick

Microsofts Ansatz zur Lösung des Komponentenproblems heißt COM. Beim *Component Object Model* (COM) handelt sich um *das* Kommunikationsprinzip für binäre Softwarekomponenten unter Windows-Betriebssystemen und dient als Basis für *Windows DNA* (siehe Abschnitt 1.4.3). Im folgenden möchte ich einen kurzen technischen Einblick in COM und verwandte Technologien geben.

Component Object Model

COM erlaubt es Objekten, ihre Funktionalität nach außen hin offenzulegen, so dass andere Software davon profitieren kann. Dabei wird festgelegt, wie ein Objekt seine Methoden der Außenwelt präsentiert und wie man diese über Prozess- und sogar Maschinengrenzen hinaus nutzen kann. Im letzten Fall spricht man dann vom *Distributed Component Object Model* (DCOM). Im Unterschied zu etwa CORBA stellen COM und DCOM nicht nur eine Spezifikation zum Erstellen von Objekten und deren Kommunikation untereinander dar, sondern liefern neben einem Komponentenmodell auch gleichzeitig eine Implementierung in Form der *COM Runtime*. Einer der Vorzüge von COM ist die Tatsache, dass man es von verschiedenen Programmiersprachen aus nutzen kann. Komponenten können mit C++ (Visual C++), Visual Basic, Java (Visual J++, das von Microsoft nicht mehr weiter entwickelt wird), Delphi und anderen Sprachen und den entsprechenden Entwicklungswerkzeugen erstellt werden. Ein weiteres Plus stellt die Ortstransparenz dar. Es spielt keine Rolle, ob COM-Komponenten als In-Prozess-Komponenten in Form von DLLs, als lokale Server in Form von Executables (EXEs) oder als entfernte Server in Form von EXEs auf einem anderen Computer laufen – es ist bis auf kleine Ausnahmen für den Entwickler völlig transparent (siehe Abbildung 1.3). So kann ein Client-Entwickler eine Applikation derart gestalten, dass diese Anwendung mit einer oder mehreren COM-Komponenten arbeitet. Der Programmierer erzeugt eine Instanz eines COM-Objekts immer gleich: Egal ob es sich um eine DLLoder eine EXE handelt oder ob sich das Objekt und die Komponente auf einem anderen Server befinden. Bei der letzten Variante, die echtes DCOM implementiert, gibt es aber auch eine Möglichkeit, den Ort der Implementierung auch im Quelltext des aufrufenden Clients anzugeben. Ansonsten stehen sämtliche Informationen in der Windows-Registrierungsdatenbank (*Registry*).

Zur Nomenklatur möchte ich noch einige Worte verlieren. Prinzipiell existieren die Bezeichnungen in der COM-Welt, wie sie in Tabelle 1.1 aufgeführt sind.

Begriff	Bedeutung in COM	Herkömmliche Bedeutung
Klasse	Definition eines Typen, welcher ein oder mehrere Schnittstellen implementiert (auch als *CoClass* bezeichnet).	Abstrakter Datentyp. Definiert Methoden und Daten.
Objekt	Instanz einer CoClass.	Instanz einer Klasse.
Schnittstelle/Interface	Zeiger auf eine konstante Gruppe von Funktionen.	Evtl. Exportdefinitionen einer DLL.
Server/Komponente	Programm (EXE oder DLL), das einem COM-Client COM-Objekte zur Verfügung stellt.	Programm, das Dienste anbietet und andere Programme bedient.

Tab. 1.1: Sprechweisen und Bezeichnungen im COM-Umfeld

Jedoch werden Sie auch in der Dokumentation von Microsoft und anderer einschlägiger Literatur immer wieder die Verwechslung zwischen COM-Komponenten und COM-Objekt vorfinden. Ich werde daher versuchen, in diesem Buch die Begriffe Objekt und Komponente soweit wie möglich zu differenzieren, um eine klarere Verständlichkeit zu erreichen.

COM-Objekte realisieren das Prinzip der Kapselung durch Schnittstellen, es werden also keinerlei Implementierungsdetails nach außen gelassen. Ein Client eines COM-Objektes sieht dieses Objekt als Black Box. Schnittstellen oder *Interfaces* stellen das zentrale Kommunikationskonzept von COM dar. Die gesamte Interaktion zwischen Objekten und deren Clients erfolgt ausschließlich über Schnittstellen. Alle Funktionen eines Objektes werden durch Interfaces nach außen zugänglich gemacht. Programmiertechnisch gesehen handelt es sich bei einem Interface um eine Tabelle mit Zeigern auf die Funktionen, die bereit gestellt werden. Im Sinne der Objektorientierung stellt die Tabelle also das Interface und die Funktionen die Methoden des Objektes dar. Dabei kann ein COM-Objekt beliebig viele Interfaces unterstützen. Ein COM-Objekt muss aber eine ganz bestimmte Schnittstelle implementieren, um als COM-Objekt zu gelten. Diese Schnittstelle heißt *IUnknown*.

Um eine weltweit eindeutige Identifizierung der Schnittstellen zu ermöglichen, werden ihnen so genannte *GUIDs* zugewiesen. Diese *Globally Unique Identifiers* vereinfachen zudem die Versionierung von Schnittstellen, denn jede neue Version einer Schnittstelle bekommt eine neue GUID, stellt somit also ein neues Interface dar. Die Informationen über die Schnittstellen werden zentral in der Windows-Registry im Zweig `HKEY_CLASSES_ROOT` abgelegt.

Als Erweiterung der Namensgebung von Komponenten oder Interfaces gibt es die *Friendly Names* oder *ProgID*, mit denen man über einen sprechenden Namen das gewünschte Interface referenzieren kann.

Die Beschreibung von Interfaces für eine sprachenübergreifende Kommunikation erfolgt mit einer speziellen Sprache, der *Interface Definition Language* (*IDL*). IDL ist keine zusätzliche Programmiersprache, sondern dient zur Beschreibung respektive Spezifizierung der Schnittstellen eines Objektes in einer programmiersprachenunabhängigen Form. Somit kann ein Client über die Interface-Definitionen einen wohldefinierten Zugriff über die Methoden eines Interfaces auf ein Objekt ausführen.

Ein essentieller Bestandteil von COM ist die Behandlung gleichzeitiger Zugriffe. Wenn mehrere Clients auf das gleiche COM-Objekt zugreifen wollten und keine Regelungen für die Reihenfolge des Zugriffs vorhanden wären, würde es unter Umständen zu unbeabsichtigten Nebeneffekten kommen: Client 1 schreibt Daten im COM-Objekt während Client 2 sie liest. In diesem Fall wäre das jeweilige Ergebnis nicht das gewünschte Ergebnis. COM synchronisiert standardmäßig die Zugriffe auf COM-Objekte, um sie so zu schützen. Wenn aber ein COM-Objekt auf Multithreading (gleichzeitige Behandlung mehrerer Programmfäden) ausgelegt ist, dann muss der COM-Programmierer selbst Sorge dafür tragen, dass diese Objekte auch Thread-sicher sind. Hier kommt der Begriff des *Apartments* ins Spiel.

Ein Apartment lässt sich als Gruppe von COM-Objekten innerhalb eines Prozesses bezeichnen, die alle die gleichen Anforderungen bezüglich des gleichzeitigen Zugriffs auf ihre Daten besitzen. Ein Prozess kann dabei ein oder mehrere Apartments beinhalten. Ein COM-Objekt gehört immer genau einem Apartment an. Man kann sich, vereinfacht gesagt, Apartments als Zuordnungshilfe zwischen Threads in einem Prozess und den darin existierenden COM-Objekten vorstellen. Wenn also beispielsweise zwei Objekte in einem Apartment sind, dann haben diese Objekte die gleichen Anforderungen bezüglich der Anzahl und der Art und Weise wie Threads auf sie zugreifen können. Es gibt in COM zwei verschiedene Apartment-Typen: das *Single Threaded Apartment* (*STA*) und das *Multi Threaded Apartment* (*MTA*). Ersteres legt fest, dass es pro Apartment nur einen Thread geben darf, dafür aber beliebig viele STAs in einem Prozess existieren können. Das MTA kann dagegen eine unbeschränkte Anzahl von Threads haben, es kann aber nur ein MTA pro Prozess vorkommen.

Wenn nun ein Aufruf eines Clients an ein COM-Objekt aus einem anderen Apartment kommt, dann tritt das so genannte *Marshaling* in Aktion. Marshaling bedeutet hier, dass kein direkter Zugriff über einen Zeiger auf den Speicherbereich des COM-Objektes möglich ist. Vielmehr muss dafür gesorgt werden, dass die Daten über die Apartment-Grenzen hinweg verpackt werden, so dass sie im Ziel-Apartment entsprechend ausgepackt und weiterverarbeitet werden können. Das gleiche Prinzip gilt natürlich auch über Prozessgrenzen und Maschinengrenzen hinweg. Im letzteren Fall spricht man dann explizit von DCOM (vgl. Abbildung 1.3). Marshaling

funktioniert in COM prinzipiell über das *Proxy*-Entwurfsmuster. Ein Proxy dient als Stellvertreter für eine Schnittstelle, die sich in einem anderen Apartment oder einem anderen Prozess befindet. Der Proxy ist dann für das Verpacken der Datentypen zuständig. Es gibt auch hier Ausnahmen von der Regel, allerdings würde deren Beschreibung den Rahmen dieses Abschnittes sprengen.

Abschließend möchte ich noch auf das v. a. in der Visual Basic-Welt verbreitete *COM Automation Marshaling* hinweisen. In diesem Fall übernimmt die *COM Automation Runtime OLEAUT32.DLL* das Marshaling von Standardtypen gemäß der Automation-Spezifikation. COM Automation-Objekte implementieren die Schnittstelle *IDispatch,* um u.a. auch mit Skriptsprachen zusammenarbeiten zu können. IDispatch ermöglicht eine späte Bindung an die Schnittstelle bzw. das Objekt, was über IUnknown nicht möglich ist. Und Skriptsprachen arbeiten von ihrem Konzept her mit später Bindung. Die Bindung und das Marshaling bei COM Automation wird durch den Einsatz von Typbibliotheken (*Type Libraries*) vereinfacht. Type Libraries sind als kompilierte Versionen der Schnittstellenbeschreibungen zu verstehen und werden auch als Metdadatenbehälter in Entwicklungsumgebungen eingesetzt. Diese können dann anhand der Type Library beispielsweise heraus finden, welche Typen und Parameter in einem COM-Server implementiert sind und diese Informationen dem Entwickler an die Hand geben.

Dieser kompakte Überblick über die Funktionsweise von COM soll für unsere Zwecke genügen. Um nun mit dem Component Object Model auch unternehmenskritische verteilte Anwendungen erstellen zu können, bedarf es noch zusätzlicher Funktionalität und Dienste.

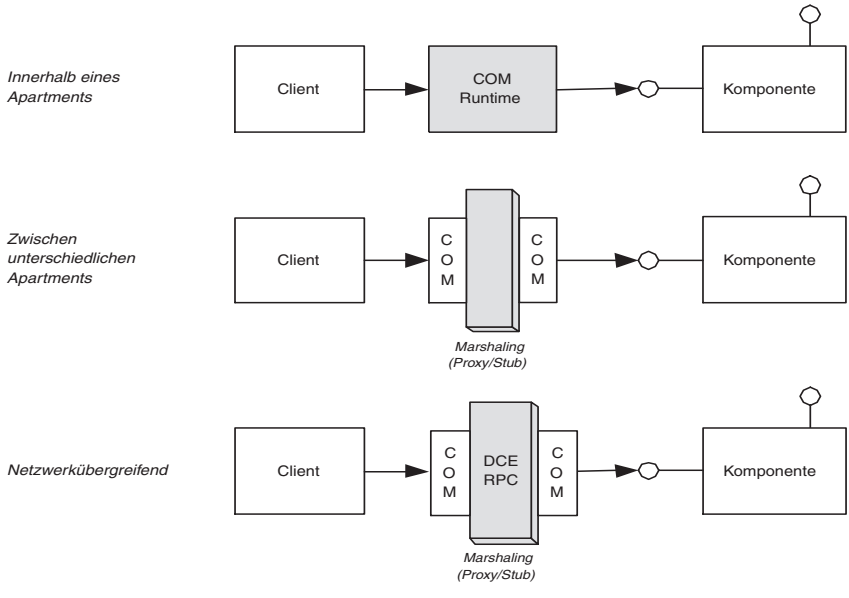


Abbildung 1.3: Kommunikationsprinzip von COM und DCOM

Microsoft Transaction Server

Zwei der kritischsten Anforderungen an eine verteilte Anwendung sind die Skalierbarkeit und die Robustheit von Server-Anwendungen. Diese lassen sich im Windows NT-Umfeld mit dem *Microsoft Transaction Server* (*MTS*) implementieren. Der MTS ist unter Windows NT eine wichtige Komponente des Windows DNA-Frameworks (siehe Abschnitt 1.4.3) von Microsoft und stellt allgemein gesprochen eine Erweiterung von COM dar. Damit man große verteilte Anwendungen auf Basis von Windows NT entwerfen und implementieren kann, bedarf es wichtiger Applikationsdienste auf der mittleren Schicht der Anwendungs- und Datenzugriffslogik. Das ureigentliche COM ist ein reines Komponentenmodell und war nie dazu bestimmt, als echte Middleware für verteilte Anwendungen zu dienen. Für eine gute Middleware werden Dienste wie Transaktionsverarbeitung, asynchrone Kommunikation oder gleichzeitige Bedienung von mehreren Clients benötigt. Diese Dienste stellt der MTS zur Verfügung, wobei die Transaktionsverarbeitung nur eine Eigenschaft repräsentiert. Aus diesem Grund ist der Name Microsoft Transaction Server sehr unglücklich, weil irreführend gewählt. Der Begriff *Microsoft Component Services* umschreibt die Rolle des MTS viel besser. Mit COM+ (siehe folgender Abschnitt), welches seit Windows 2000 verfügbar ist, wird die Funktionalität und das Programmiermodell des MTS in die Laufzeitumgebung von COM integriert und erweitert.

Mit der Einführung des MTS war es möglich, bestimmte Aufgaben zu vereinfachen, die ein Entwickler für die Erstellung verteilter Applikationen beachten musste. Ein großes Problem in COM war und ist die relativ aufwändige Programmierung von COM-Servern. Kritische Eigenschaften wie Multithreading oder Teilnahme an Transaktionen ist nur sehr umständlich und mit viel tiefschürfendem praktischem Wissen zu bewerkstelligen. Der MTS hat hier ein Konzept etabliert, mit dem die Server-Entwicklung wesentlich vereinfacht wird.

Das zentrale Konzept im MTS ist das der *Interception*. Um einer COM-Komponente zur Laufzeit nur durch Konfiguration bestimmter Parameter neue Mehrwertdienste wie automatische Transaktionen oder Rollen-basierte Sicherheitseinstellungen bereitstellen zu können, klinkt sich der MTS in die Objektinstanziierung und den Methodenaufruf ein. Siehe hierzu auch Abbildung 1.4. Durch ein Kontextobjekt kann er verschiedene Konfigurationsparameter für ein Objekt bereit halten und so beispielsweise die Teilnahme an einer verteilten Transaktion gewährleisten.

Um eine größtmögliche Skalierbarkeit der Server-Anwendungen zu erreichen, arbeitet der MTS mit diversen Ressourcen-Pools. Das bedeutet, dass eine Menge teurer Ressourcen, wie eine Datenbankverbindung oder ein Thread, in einem Bereich des Speichers gehalten werden. Objekte, die dann eine dieser Ressourcen benötigen, werden mit einer Ressource aus diesem Pool bedient. Im Falle des Datenbankverbindungs-Poolings wird dies bei-

spielsweise durch die Datenzugriffsschnittstelle *OLEDB* realisiert. Ein weiteres Mittel zur Erreichung hoher Skalierbarkeit ist die Verwaltung von Objekten und deren Ressourcen zur Laufzeit.

Da der MTS sich bildlich gesprochen in die Kommunikation zwischen einem Client und einer Komponente einklinkt, kann er bestimmen, wie und wann die eingehende Client-Anfrage an das eigentliche Objekt weiter geleitet wird. Um so wenig wie möglich Speicher und Ressourcen auf dem Server zu belegen, wird ein Objekt erst dann aktiviert, wenn wirklich ein Methodenaufruf erfolgt. Dies bezeichnet man auch als *Just In Time (JIT)-Aktivierung*. Außerdem wird das Objekt sofort nach der Benutzung wieder freigegeben – wenn der Entwickler sich an dieses neue, doch ungewohnte Programmierparadigma hält und seine Objekte gemäß dieser Richtlinien entwirft und implementiert.

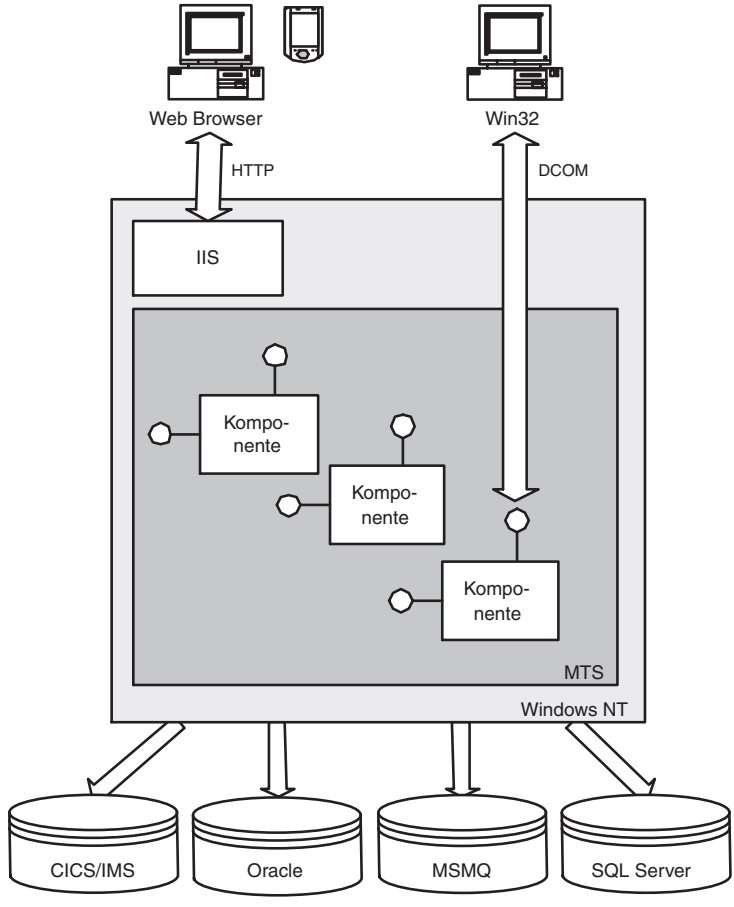


Abbildung 1.4: Interception-Prinzip des Microsoft Transaction Server als Laufzeitumgebung

Eine weitere, erwähnenswerte Neuerung des MTS ist die Attribut-basierte Entwicklung. Anstatt alle gewünschten Dienste und Funktionalitäten immer und immer wieder neu zu schreiben, stellt der MTS – wie oben beschrieben - gewisse Applikationsdienste von Haus aus zur Verfügung. Um diese Dienste nutzen zu können, müssen Komponenten zunächst beim MTS registriert werden. Anschließend kann man über die Zuweisung gewisser Attribute einfach deklarieren, wie sich ein Objekt innerhalb einer Transaktion verhält oder welche Benutzer Zugriff auf bestimmte Schnittstellen haben. Dies erleichtert die Entwicklung und vor allem die Verteilung und die Pflege von Server-Anwendungen ungemein.

COM+ – Komponentendienste

COM+ ist die Vereinigung zweier essentieller Technologien und deren Anreicherung mit neuen nützlichen Eigenschaften. Die beiden wichtigen Technologien sind das oben beschriebene Komponentenmodell von COM sowie die Programmiermethodik und Laufzeitumgebung des MTS (vgl. Abbildung 1.5). Im Folgenden werden die wichtigsten Konzepte und Dienste von COM+ aufgeführt und erläutert.

Konfigurierte und nicht konfigurierte Komponenten. Nichtkonfigurierte Komponenten sind COM-Komponenten, die kein Gebrauch von COM+-Diensten machen; ihre Metadaten (Lokalisierung, Apartment-Typ, etc.) sind nach wie vor einzig und allein in der Windows-Registry zu finden. Konfigurierte Komponenten werden im COM+-Katalog beschrieben und bedienen sich der COM+-Dienste, um spezielles Verhalten zu implementieren. Konfigurierte Komponenten können nur als DLLs existieren. Man kann zusammengehörige DLLs in einen Prozess packen und sie als eine so genannte COM+-Applikation installieren.

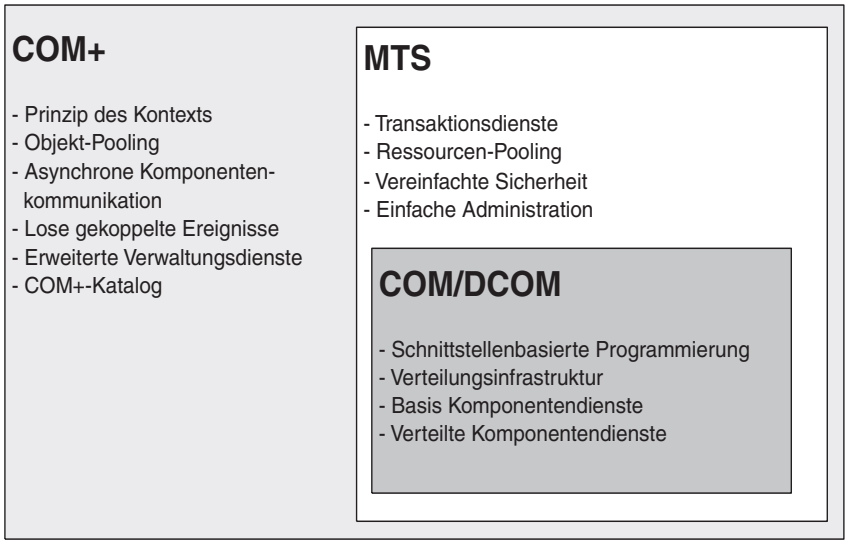


Abbildung 1.5: COM+ als Evolution

Kontext & Interception. Der Begriff Kontext wurde erstmals mit der Einführung des MTS bekannt. Einen Kontext kann man als Menge von zur Laufzeit benötigten und verwendeten Eigenschaften für ein oder mehrere Objekte bezeichnen. Diese Eigenschaften können sich z. B. auf die Art und Weise der Beteiligung an Transaktionen oder auf die Sicherheitseinstellungen beziehen. Wenn ein Objekt erzeugt wird, wird auch automatisch ein zugehöriges Kontextobjekt instanziiert. In COM+ unterscheidet man hierzu auch konfigurierte und nicht konfigurierte Komponenten und Objekte. Wenn ein COM+-Objekt nicht konfiguriert ist (es wurde nicht im COM+-Katalog registriert), dann verhält es sich wie ein »gewöhnliches« COM-Objekt – in diesem Fall stellt das Kontextobjekt auch nur eine Art leere Hülle dar. Ist es hingegen konfiguriert, dann wird das zugehörige Kontextobjekt mit den entsprechenden Eigenschaften aufgebaut. Erst durch die Einführung und die Nutzung von Kontexten können die COM+-Objekte die zahlreichen Dienste der COM+-Laufzeitumgebung nutzen. Das Kontextobjekt schiebt sich bildlich gesprochen während eines Methodenaufrufes zwischen den Client und das eigentliche COM+-(Server)-Objekt. Diese Vorgehensweise bezeichnet man auch als Interception und sie zieht sich durch die gesamte COM+-Programmierung.

COM+-Katalog. Wer bisher unter COM Komponenten entwickelt hat, wird intensive Bekanntschaft mit der Windows Registrierungsdatenbank gemacht haben. Mit COM+ wird der Katalog eingeführt. Informationen über die Eigenschaften einer Komponente findet COM+ im COM+-Katalog. Dieser Katalog ist eine neue, auf sehr schnelle Leseoperationen optimierte Datenbank für das Speichern von Metadaten über Komponenten. Hierbei handelt es sich um eine neue systemeigene Datenbank und der Entwickler kann entweder mit der KOMPONENTENDIENSTE-Anwendung oder über eine Anzahl von COM-Schnittstellen diese Datenbank bearbeiten und nutzen.

Die Administration und Konfiguration von COM+ erfolgt in den meisten Fällen über eine grafische Benutzungsoberfläche. Diese Anwendung ist ein Plug-In für die *Microsoft Management Console* (*MMC*). Sie befindet sich in der Programmgruppe VERWALTUNG und heißt KOMPONENTENDIENSTE.

COM+ integriert eine ganze Reihe neuer Dienste. Sie sollen im Folgenden kurz vorgestellt und umrissen werden. Dienste (*services*) stellen bestimmte, vorgefertigte Funktionalität für Komponenten zur Verfügung.

Objekt-Pooling. Mit dieser neuen Einrichtung werden auf Wunsch mehrere Objektinstanzen im Speicher gehalten, damit sie sofort für Anwendungsprogramme zur Verfügung stehen. Der zeitraubende Instanziierungsprozess entfällt also. Dies bedeutet eine erhöhte Effizienz für Komponenten, deren Instanziierungszeit geringer ist als die Zeit zum Auslösen eines Objektes aus dem Pool. Objekt-Pooling ist vor allem dann sinnvoll, wenn es sich entweder um Objekte mit sehr zeitaufwendiger Instanziierung oder um Objekte mit einem generischen Erzeugungsprozess handelt – in

beiden Fällen bringt das Zurückstellen in einen Vorrat und das Wiederauslösen aus einem Vorrat von Objekten einen signifikanten Geschwindigkeitsvorteil. Die Anzahl der Objekte, die in einen Pool gestellt werden kann der Entwickler oder Administrator über die KOMPONENTENDIENSTE-Anwendung einstellen. Objekt-Pooling wird auch in der zu entwerfenden Architektur als Mittel zur Erhöhung der Skalierbarkeit eingesetzt.

Transaktionsverarbeitung. Durch die Verschmelzung der Programmiermodelle von COM und MTS findet sich auch in COM+ die Unterstützung zur automatischen Transaktionverarbeitung wieder. Eine Transaktion ist eine Folge mehrerer Aktionen auf einem Datenbestand, die als eine in sich abgeschlossene Menge von Aktionen angesehen wird. Eine Transaktion ist dann erfolgreich, wenn jede Aktion innerhalb dieser Menge erfolgreich war. Sie ist nicht erfolgreich, wenn auch nur eine Aktion aus der Menge fehlschlägt. Dann werden alle beteiligten Aktionen wieder zurückgesetzt. In Verbindung mit dem *Distributed Transaction Coordinator* (*DTC*) verwaltet und steuert die COM+-Laufzeitumgebung verteilte Transaktionen, um einen konsistenten Zustand aller Daten zu gewährleisten. Dabei wird der Entwickler von den lästigen Details abgeschirmt – er muss nur festlegen, in welchem Fall die Transaktion erfolgreich oder erfolglos beendet wird. Auf Basis dieser Entscheidung wird dann entsprechend im Erfolgsfall ein *Commit* oder aber ein *Rollback* ausgelöst. Dies bedeutet u.a., dass der Programmierer nicht jedes Mal den Algorithmus des *2-Phasen-Commit-Protokolls* für verteilte Transaktionen implementieren muss. Die Konfiguration einer Komponente bezüglich ihrer Teilnahme an Transaktionen erfolgt ebenfalls über KOMPONENTENDIENSTE-Anwendung. Diese Informationen befinden sich dann zur Laufzeit des Objektes im zugehörigen Kontextobjekt, über das der Entwickler dann auch den Ausgang der Transaktion steuern kann. Dieser Dienst wird ebenfalls eingesetzt, um die Anwendungsarchitektur auf Komponentenebene besser zu skalieren.

Asynchrone Komponentenkommunikation. In einigen Anwendungsfällen ist es wünschenswert, dass die Client-Anwendung und der COM+-Server nicht über die synchronen Standardmechanismen miteinander kommunizieren, sondern eine asynchrone, Nachrichten-basierte Kommunikation stattfindet. Dieses war im bestimmten Umfang schon früher unter Windows NT 4.0 mit dem *Microsoft Message Queue Server (MSMQ)* möglich. Im Zuge der Einführung von COM+ wurde dieses Merkmal wie der MTS ein integraler Dienst innerhalb der Laufzeitumgebung. Der große Unterschied zur Programmierung mit und für den MSMQ besteht darin, dass in COM+ durch die *Queued Components* (*QC*) die Details für asynchrones Messaging vor dem Entwickler versteckt werden. Es ist zu beachten, dass die Kommunikation immer noch über DCOM stattfindet. Es besteht jedoch die Möglichkeit, MSMQ und QC mit anderen Messaging-Systemen wie z.B. von IBM zu verbinden.

Ereignisverarbeitung. Um der teilweise umständlichen Behandlung von Ereignissen in verteilten Anwendungen unter (D)COM ein Ende zu bereiten, implementierte Microsoft ein neues Konzept zur Ereignisver-

waltung. Es wird wegen der sich ständig ändernden Rollen von Client und Server auch von Herausgeber und Abonnent gesprochen (*publish/subscribe*). Mit der COM+-Ereignisklasse wird die enge Bindung zwischen einem Herausgeber und einem Abonnenten aufgelockert. Stellt ein Herausgeber Ereignisse nach außen zur Verfügung, so muss er dies über die *IEventClass*-Schnittstelle der COM+-Laufzeitumgebung tun. Dadurch wird eine Ereignisklasse erzeugt. Analog muss ein Abonnent ein Abonnementobjekt erzeugen, um Ereignisse anderer Objekte zu erhalten.

Sicherheit. Auch im Bereich Sicherheit in verteilten Komponenten-basierten Applikationen bietet COM+ interessante Eigenschaften. Wieder in systemeigene Dienste eingebettet, braucht der Entwickler nur die Sicherheitsfunktionen zu aktivieren und zu benutzen; das teilweise doch recht komplexe Arbeiten mit Schnittstellen für Sicherheit entfällt weitgehend. Dennoch können sie benutzt werden, um die vorhandene Funktionalität eigenen Bedürfnissen anzupassen. Die größte Erleichterung ist die so genannte Rollen-basierte Sicherheit. Durch sie wird im Programmcode lediglich eine bestimmte Rolle festgelegt und später über die KOMPONENTENDIENSTE-Anwendung die entsprechenden Benutzer dieser Rolle zugewiesen.

Neben diesen neuen Diensten, die direkt die Entwicklung Komponenten-basierter verteilter Systeme für den Programmierer erleichtern, finden sich auch noch weitere Verbesserungen wie die integrierte Verwaltung für COM+-Komponenten in der neuen Ausgabe von Microsofts Middleware. Die Administration wird auf alle COM+-Komponenten erweitert.

Mit dem Erscheinen von Windows 2000 und COM+ sind skalierbare Serveranwendungen unter Windows tatsächlich Realität geworden. Vorausgesetzt man beachtet die vielen Rahmenbedingungen und hält sich an die teilweise neuen Programmierparadigmen, kann man mit COM+ gute Intranet-Anwendungen bauen. Beim Schritt ins Internet hingegen ergeben sich einige Probleme und auch bei erhöhten Anforderungen an die Anwendung erreicht man vor allem mit COM und DCOM schnell seine Grenzen.

1.4.3 Windows DNA

Windows DNA steht für *Windows Distributed interNet Applications Architecture*. Mit der DNA-Architektur stellt Microsoft ein Anwendungsmodell und Framework für Windows zur Verfügung. Sie ermöglicht dadurch Entwicklern von unternehmensweiten verteilten Anwendungen auf bereits existierende Windows-Systemdienste zuzugreifen und über standardisierte Technologien und Schnittstellen Benutzungsoberflächen zu erstellen oder sich zu Datenbanksystemen zu verbinden. DNA basiert auf einer drei- oder mehrschichtigen Architektur und verlässt sich im Kern auf die Fähigkeiten von COM zur binären Kommunikation zwischen Softwarekomponenten. Für nähere Informationen über n-Schichten-Architekturen lesen Sie bitte Abschnitt 1.3.

DNA ist kein Produkt, sondern eine Strategie oder Vorgehensweise bei der Entwicklung von verteilten Anwendungen unter Windows. Oftmals spricht man auch von einer Blaupause, wenn man über Windows DNA im Zusammenhang mit dem Erntwurf von verteilten Anwendungen redet. Dabei wird die zugrunde liegende Betriebssystemplattform, eine Komponententechnologie, Systemdienste und eine Anwendungsinfrastruktur als Rahmen zur Verfügung gestellt. Damit der Entwickler all dies effektiv nutzen kann, werden ihm Entwicklungsumgebungen zur Seite gestellt, die unmittelbar ins Windows DNA-Gerüst integriert sind (siehe Abbildung 1.6). Mit Windows DNA 2000 hat Microsoft im September 1999 eine Erweiterung von DNA um einige wichtige und zentrale Technologien und Server-Produkte vorgestellt.

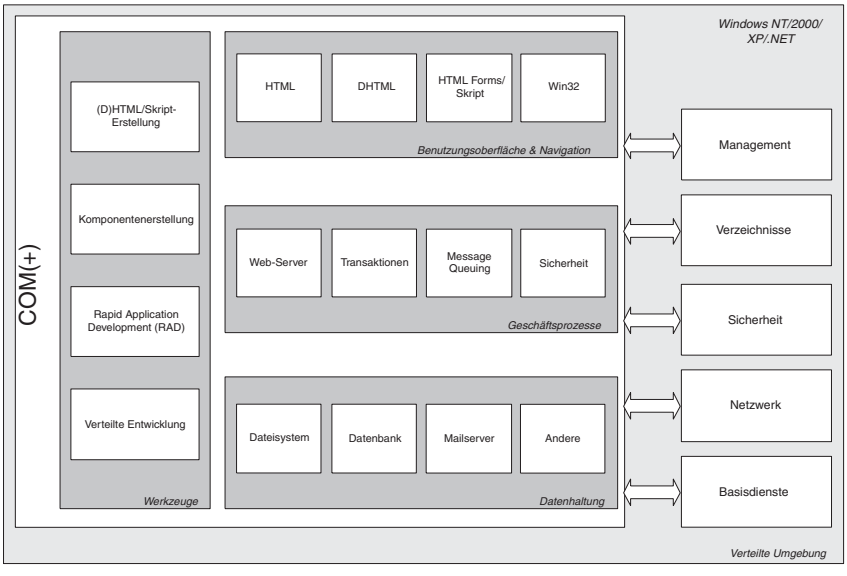


Abbildung 1.6: Überblick über Windows DNA

1.5 Warum .NET?

Um die Frage in der Kapitelüberschrift hier wieder aufzunehmen: Wieso, weshalb, warum brauchen wir als Entwickler Web-basierter Anwendungen jetzt etwas Neues; warum brauchen wir .NET? Sind die in diesem Kapitel vorgestellten Vorgehensweisen und Technologien nicht ausreichend? Diese letzte Frage muss man leider mit »*Nein, sind sie nicht*« beantworten. Im folgenden werde ich die Probleme und Unzulänglichkeiten der existierenden Lösungen in einer Zusammenfassung beleuchten. Weiterhin dient der folgende Abschnitt auch der Sensibilisierung in Bezug auf die sich ändernden Anforderungen an Internet-Anwendungen.

1.5.1 Probleme existierender Lösungen

Durch die in Abschnitt 1.2.1 beschriebene Situation in Bezug auf Webbasierte Anwendungen kann man verschiedene Faktoren als kritisch für erfolgreiche Applikationen ausmachen. Diese Faktoren werden in den folgenden Abschnitten beleuchtet:

- Interoperabilität,
- Implementierung von Standards,
- Einheitliches Programmiermodell,
- Einheitliche Schnittstellen,
- Unkomplizierte Installation und Updates.

Was ist also bezüglich der oben aufgeführten Punkte schlecht an der jetzigen Situation? Mit der Vielzahl von Plattformen, Betriebssystemen und Applikationen im Internet ist eines der großen Hauptziele die plattformübergreifende Integrationsmöglichkeit. Angefangen bei lokalen Anwendungen, die mit unterschiedlichen Sprachen entwickelt wurden, bis hin zu echter Interoperabilität über Rechner- und Betriebssystemgrenzen hinweg ist Interoperabilität ein altes und zum Teil nur unzureichend gelöstes Problem.

Das Problem auf Programmiersprachenebene ist, dass die unterschiedlichen, auf dem Markt befindlichen Sprachen völlig verschiedene Typsysteme besitzen. Das bedeutet, dass Programmiersprachen wie Visual Basic, C, C++ oder Delphi eine unterschiedliche interne Repräsentation ihrer Typen wie beispielweise Integer oder String haben. COM versuchte, mit einer sprachenübergreifenden binären Standardisierung diese Problematik zu überwinden. Allerdings kann man diesen Versuch lediglich als ein ambitioniertes Hilfskonstrukt bezeichnen. Denn wenn eine Client-Anwendung in Visual Basic und die Serverkomponente in C++ realisiert wurden, dann existieren drei unterschiedliche Darstellungen eines verwendeten Typs: Die Darstellung innerhalb von Visual Basic, innerhalb von C++ und die Repräsentation in COM zum Austausch zwischen den Parteien. Also alles in allem eine nicht optimale Lösung eines bekannten Problems. Hier bedarf es einer besseren Lösung.

Bei der rechnerübergreifenden Interoperabilität geht es vornehmlich um die Integration von vorhandenen Diensten bzw. Plattformen. Die üblichen Anforderungen in einer heterogenen Landschaft sind die, dass existierende Programme auf unterschiedlichen Rechnern auf mitunter unterschiedlichen Betriebssystemen mit unterschiedlichen Softwareplattformen miteinander kooperieren müssen. Diese Programme, die einem Client eine gewisse Dienstleistung anbieten, kann man auch abstrakt als Dienste bezeichnen. Das Problem in diesem Bereich ist der Mangel an Standards bzw. deren Umsetzung. Proprietäre Lösungen wie DCOM sind für solche Unterfangen absolut ungeeignet. Auch CORBA als scheinbarer Standard ist ebenso wenig eine befriedigende Lösung. Ein echter Standard sollte die guten Eigenschaften der

genannten Protagonisten in sich vereinen, ohne dabei die Schwierigkeiten und Unzulänglichkeiten wie Probleme mit Firewalls oder plattformübergreifende Kommunikation aus dem Auge zu verlieren.

Die nächste Kategorie von Problemen bei COM/DCOM-basierten Anwendungen ist die Installation und Pflege der Anwendung respektive Komponenten. Die Idee der einfachen Austauschbarkeit von Komponenten und der Wunsch nach Plug&Play-Software durch Zusammenfügen von diversen Komponenten wurde in der Realität des Component Object Model leider nicht gut umgesetzt. Sämtliche Informationen über eine COM-Komponente und der darin enthaltenen Schnittstellen, Objekte, Typen werden in der Windows-Registry verankert. Die Registrierung einer solchen Komponente findet durch das folgende Kommando statt:

```
regsvr32.exe myCOM.dll
```

Mit diesem Befehl werden diverse Einträge in der Registry vorgenommen, die aber nicht allein an einer einzigen Stelle, sondern an unterschiedlichen Punkten in der Registry verankert werden. Ebenso muss eine Komponente auch wieder manuell deregistriert werden. Die Syntax hierzu wir in der folgenden Zeile exemplarisch aufgezeigt:

```
regsvr32.exe /u myCOM.dll
```

Und eben dieser Zwang zur Festschreibung von Komponenten im System ist eine sehr große Einschränkung. Diese als »DLL-Hölle« bekannte Situation unter Windows kann einen Entwickler bzw. Systemadministrator zur Verzweiflung treiben. Vor allem wenn sich unterschiedliche Versionen einer Schnittstelle oder eines Objekts im System befinden, gibt es gehäuft Probleme für Client-Anwendungen, wenn sie auf die Schnittstellen zugreifen wollen. Zudem bringen viele COM-Komponenten noch eine zusätzliche Datei mit, in der sich die zugehörige Type Library befindet. Die darin befindlichen Metadaten werden ebenso in der Registry verankert. Eine Lösung mit einer nicht notwendigen Registrierung ist angesichts der beschriebenen Sachlage unabdingbar.

Weiterhin ist mit der momentanen Technologie ein großes Sammelsurium an unterschiedlichen Programmierschnittstellen vorhanden. Visual Basic-Entwickler arbeiten mit Visual Basic-Formularen und hauptsächlich mit COM- bzw. ActiveX-Komponenten. Die C++-Programmierer realisieren ihre Software mithilfe von *MFC*, *ATL*, *STL* oder anderen Klassenbibliotheken. Als Basis dient in den meisten Fällen das *Win32 –API* (Application Programming Interface). Für eine sprachenübergreifende Klassenbibliothek, die im System verankert arbeitet, würden viele Entwickler und Projektleiter die bisher eingesetzte Entwicklungsplattform wechseln.

Wie wir an den obigen Ausführungen sehen können, sind die Probleme einer COM/DCOM-basierten Architektur keineswegs zu vernachlässigen. Wenn man hierzu noch die anfangs des Kapitels beschriebenen neuen Anforderungen an Internet-Applikationen addiert, so kommt man schnell zu der Einsicht, dass eine neuartige, auf diese Umstände angepasste Technologie ein logischer Schritt in die richtige Richtung ist.

1.5.2 Möglicher Lösungsansatz

Es gilt also eine Lösung für dieses neu aufgetretene Problem zu finden. Wenngleich das anfangs dieses Kapitels illustrierte Paradigma für das Arbeiten des Menschen mit dem Internet gut geeignet ist und sich auch zurecht etabliert hat, birgt es doch einige mehr oder weniger schwerwiegenden Probleme für den Einsatz in E-Commerce-Architekturen. Hier kann man mehrere Aspekte ausmachen.

Zum einen sind moderne E-Commerce-Anwendungen derart gestaltet, dass für eine bestimmte Dienstleistung ein oder mehrere spezialisierte Anbieter existieren. Diese Anbieter verbreiten ihre Dienste meistens über Browserbasierte Schnittstellen. Jedoch ist dies nicht ausreichend. Wenn beispielsweise ein Online-Marktplatz Dienste eines Anbieters zum Vertrieb von Werbeschildern anbieten möchte, sollten die übertragenen Informationen und deren Zugang in einer standardisierten Art und Weise, aber ohne unstrukturierte Darstellungselemente vorgenommen werden. Hier drängt sich die XML-Technologie als Datenbeschreibungsstandard auf.

Zum anderen werden in Zukunft immer mehr neue Benutzungsgeräte, so genannte *Appliances*, zum Einsatz kommen. Der Manager möchte etwa eine speziell für sein Unternehmen entwickelte Personalplanungssoftware mit seinem PDA von unterwegs aus nutzen, um den Einsatzplan für den kommenden Monat zu erstellen. Die Software ist zwar speziell auf seine Firma abgestimmt, jedoch benutzt sie auch Standardkomponenten. Diese Komponenten bietet der Hersteller als über das Internet aufrufbare Web-Dienste an.

Dies ist nur ein Beispiel für zukünftig denkbare Anwendungen. Viele - wenn nicht gar alle Programme - werden Internet-Zugangsmöglichkeiten besitzen, um mit bereits vorhandenen Online-Ressourcen zu kommunizieren. Dieser universelle Internet-Zugang sollte idealerweise vom verwendeten Betriebssystem unabhängig sein. Jener oben erwähnte PDA könnte also auf Windows CE- oder PalmOS-Basis arbeiten – dies sollte für die Nutzung der Applikation keinen Unterschied darstellen. Hinzu kommt noch, dass die Art und Weise des Zugriffs für den Entwickler nicht von einer gewissen Programmiersprache abhängen sollte. Jede verfügbare Sprache sollte mit so wenig Aufwand wie möglich die vorhandenen Möglichkeiten (sprich Dienstleistungen) im Internet nutzen können.

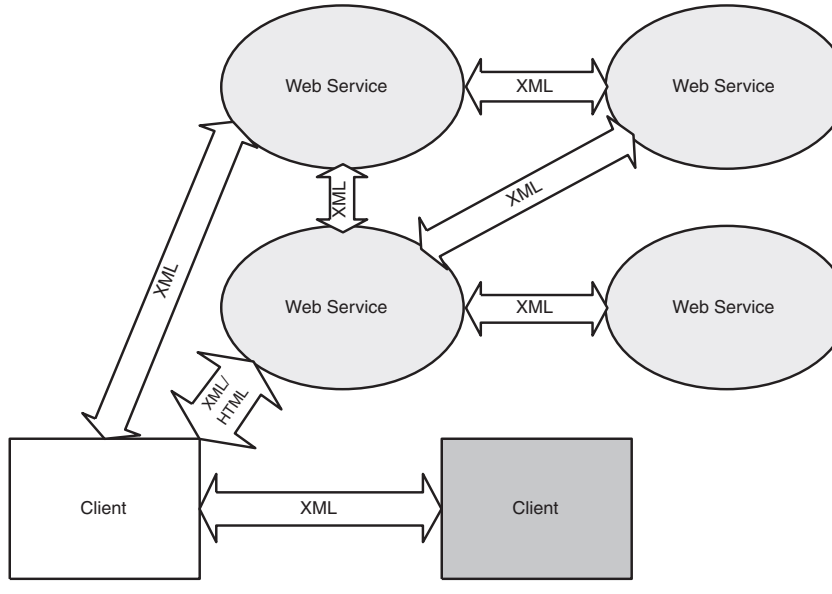
Für dieses technische Problem gab es in der Vergangenheit mehrere Lösungsansätze, die weiter oben erwähnt und teilweise auch genauer vorgestellt wurden. Vor allem die Crux der Interoperabilität war bis vor kurzem eine scheinbar unlösbare Angelegenheit: DCOMs DCE RPC, CORBAs IIOP und Javas RMI konnten aufgrund ihrer proprietären Herkunft und Probleme mit Firewalls keine zufriedenstellende Lösung vorweisen. Auf der Suche nach geeigneten Protokollen und Technologien für eine verbesserte Interoperabilität stieß man schnell auf XML. Denn seit dem Erscheinen des plattformübergreifenden *SOAP* (*Simple Object Access Protocol*) ist ein Ausweg in Sicht. Durch

die Verwendung von Standardprotokollen wie HTTP und dem Datenbeschreibungsstandard XML ermöglicht SOAP eine plattformübergreifende Kommunikation zwischen Internet-Anwendungen. Und eben dieser Standard wird als Basis für die so genannten Web Services herangezogen.

Web Services sind programmierbare *URIs* (*Uniform Resource Identifier*), d.h. dass sie über ein vertrautes Adressierungsschema angesprochen werden. Sie liefern jedoch keine gewöhnlichen HTML-Ausgaben, sondern berechnen auf Basis von übermittelten Daten eine Menge von Ausgabedaten innerhalb eines Programms. Somit kann man sich beispielsweise eine Anwendung vorstellen, die als Eingabeparameter ein Wort entgegen nimmt, welches auf korrekte Schreibweise überprüft wird. Ein frei verfügbarer Web Service könnte die interne Implementierung dieser Überprüfung entweder auf Basis einer eigenen Datenbank vornehmen oder die in Microsoft Word verfügbare Rechtschreibüberprüfung benutzen.

Somit lässt sich ein abstrakter Überblick über eine mit Web Services aufgebaute Anwendung wie in Abbildung 1.7 darstellen. Diese Vorgehensweise stellt den Schritt weg vom gewohnten Browser-Modell hin zum Software-zu-Software-Modell dar, welches sich in der Zukunft immer weiter verbreiten wird.

Abbildung 1.7: Web Services Szenario



Ziel dieses Buches ist die Vermittlung der Technologien aber auch der neuen Programmierparadigmen in .NET mit Schwerpunkt auf Web Services-basierten Architekturen.

1.6 Zusammenfassung

In diesem Kapitel habe ich Ihnen die Probleme existierender Plattformen zur Entwicklung Internet-basierter Lösungen aufgezeigt. Die aktuell eingesetzten Technologien wie COM und DCOM hatten zwar in den vergangenen Jahren durchaus ihre Daseinsberechtigung, allerdings konnten sie über die Zeit hinweg nicht mit dem Fortschritt Schritt halten. Es lastet zu viel Altlast auf ihren Schultern. Durch die Einführung von .NET, Web Services und SOAP als Mittel zur Plattformübergreifung und verbesserten Interoperabilität zwischen Anwendungen bieten sich dem Softwareentwickler viel mehr Möglichkeiten, die immer komplexer werdenden Ansprüche an die Programme auch in einem vernünftigen Zeitrahmen umsetzen zu können.

2 Das Handwerkszeug

.NET Grundlagen

2.1 Überblick

.NET. Dieser Begriff und alles, was Microsoft sowie andere Parteien damit in Verbindung bringen, wird in diesem Kapitel geklärt. Wir werden auf viele neue und unvertraute Techniken und Begriffe stoßen. .NET ist aber vor allem ein sehr weitläufiger Marketingbegriff, mit dem man von der Programmiererseite her aufräumen muss.

Nach der Erörterung der unterschiedlichen Probleme heutiger Internetanwendungen im vorigen Kapitel wird auf den folgenden Seiten die aus Redmond stammende Lösung im Überblick beschrieben. Vor allem aber die .NET-Aspekte für Entwickler und verantwortliche Manager werden tiefer beleuchtet, um eine gute Ausgangslage für die kommenden Kapitel zu schaffen. In diesen werden jeweils verschiedene Spezialthemen und –technologien für den Einsatz in einer auf .NET basierenden verteilten Internet-Architektur beleuchtet.

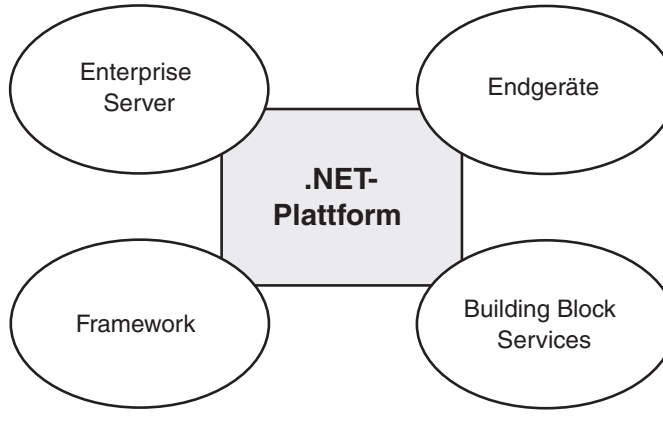
2.2 Idee und Gesamtkonzept von .NET

»*Software as a Service.*« Mit dieser Aussage lässt sich die eigentliche Grundidee von Microsoft .NET vermutlich am besten beschreiben. Dass Software nicht mehr immer nur zwangsläufig lokal auf einem PC läuft, sondern sich gewisse Teile der Anwendungslogik auch im weiten Internet oder innerhalb des Intranets befinden können, dies versucht .NET mit einigen neuen Ansätzen zu realisieren. Dabei ist .NET nicht, wie viele befürchten, etwas völlig Neues, sondern eine logische und konsequente Weiterentwicklung bisher eingesetzter Techniken und Lösungen, die nun an ihre Grenzen gestoßen sind. .NET ist also eher eine Evolution als eine Revolution.

Man spricht im Zusammenhang von .NET auch meistens von der .NET-Plattform. Dies ist eine Plattform zur Entwicklung und zum Betrieb hauptsächlich von XML Web Service-basierten Internet-Anwendungen. Damit eine solche neue Vision auch entsprechend verwirklicht werden kann, bedarf es einer gewissen Ausrichtung der Produkte und Technologien innerhalb der Plattform.

Für eine funktionsfähige Web Service-Plattform werden Internet-fähige Server-Applikationen benötigt. Solche Server wie Email-, Datenbank oder Dokumentenmanagement-Server werden für die Speicherung und das Publizieren von Daten und Informationen über das Internet benötigt. Microsoft will in diesem Segment in den kommenden Jahren seine Server-Palette ausbauen und vervollständigen. Damit soll eine relativ homogene Server-Infrastruktur als Rückgrat für Web Service-Anwendungen garantiert werden. Außer den eigentlichen Hauptdarstellern in dieser Inszenierung – den Web Services – werden auch die richtigen und passenden Entwicklungswerkzeuge benötigt. Denn eine Technologie und ein Standard sind nur so gut und erfolgreich wie die zugehörigen integrierten Entwicklungsumgebungen (*Integrated Development Environment – IDE*). Hier sorgt vor allem *Visual Studio .NET* für Aufmerksamkeit und wird sich vermutlich als Standardwerkzeug in diesem Bereich durchsetzen. Die Web Services, ob eigens entwickelt oder über einen externen Anbieter bezogen, müssen in einer für den Entwickler einfachen und reproduzierbaren Art und Weise in die übrige Anwendung und deren Logik mit eingebunden werden. So soll eine Windows-basierte Anwendung ebenso einfach auf einen Web Service zugreifen können wie eine über Server-seitige Mechanismen erstellte Web-Anwendung. Diese Endbenutzeranwendungen werden dann schließlich auf einem PC, einem Handheld-Gerät (besser bekannt als *PDA*: Personal Digital Assistant), einem Mobiltelefon oder irgendeinem heute noch nicht bekannten Endgerät bedient.

Abbildung 2.1: Die .NET-Plattform und ihre Pfeiler



Alle diese geschilderten Anforderungen und Voraussetzungen will Microsoft mit seiner .NET-Plattform erfüllen. Wie in Abbildung 2.1 ersichtlich ist, will man mit verschiedenen Schwerpunkten diese Vision verwirklichen.

Zu erwähnen sind hier die *.NET Enterprise Server* als Basis für eine Serverzentrierte Architektur. Allerdings existierte zum Zeitpunkt des Schreibens dieser Zeilen noch kein einziger echter .NET-Server. Zwar werden alle neuen Server-Produkte als .NET-Server ausgezeichnet, doch ist dies hauptsächlich als Marketingstempel anzusehen. Als »echt« bezeichne ich hierbei Applika-

tionen, die in das Gesamtgefüge von .NET eingebunden sind. So wird der nächste SQL Server (Codename *Yukon*) einer der ersten .NET Enterprise Server sein, da er beispielsweise die neue .NET-Laufzeitumgebung CLR (siehe auch Abschnitt 2.4.1) enthält.

Als Programmiermodell und –werkzeuge bietet Microsoft mit dem *.NET Framework*, dem *.NET Framework SDK* und dem Visual Studio .NET die gesamte Palette für die Entwicklung und Pflege von .NET-Anwendungen. Während das .NET Framework vornehmlich die Laufzeitumgebung darstellt, handelt es sich beim .NET Framework SDK um die Grundlage der Entwicklung .NET-basierter Anwendungen. Sämtliche Software kann im Prinzip alleine mit dem frei verfügbaren SDK entwickelt werden, da fast alle benötigten Werkzeuge hierin enthalten sind. Selbstverständlich gibt es aber auch eine komfortable integrierte Entwicklungsumgebung in Form von Visual Studio .NET.

Mit diesen Programmierwerkzeugen kann der Entwickler dann Web Services und andere auf .NET aufsetzende Programme schreiben. Diese Web Services wiederum werden in vielen Anwendungsfällen andere, teils auch im Internet verfügbare, Web Services für ihre Anwendungslogik verwenden.

Es soll hier erwähnt werden, dass Web Services nicht immer die optimale Lösung für jedes Problem sind. In vielen Fällen hängt es ganz eindeutig von den Anforderungen der Anwendung ab, ob und wie Web Services und verwandte Techniken zum Einsatz kommen oder nicht.

Dieses Buch will daher zwar hauptsächlich auf die Entwicklung von Web Services-basierten Anwendungen abzielen; es will aber auch immer mit einem kritischen Auge die Situation und den Sachverhalt durchleuchten.

Die Endanwender schließlich arbeiten mit ihrem heute vertrauten PC oder aber mit neuen, zum Teil heute noch nicht verfügbaren Bediengeräten. Microsoft nennt diese Benutzbarkeit von auf .NET-basierenden Applikationen *.NET User Experience* und will damit suggerieren, dass ein Benutzer, der mit .NET-Anwendungen arbeitet, ein besseres »Benutzererlebnis« genießt, als andere Anwender. Auch hier will ich die Marketingschiene verlassen und darauf hinweisen, dass man auch mit anderen Ansätzen und Plattformen gut benutzbare Anwendungen erstellen kann.

Somit lässt sich ein Gesamtbild der .NET-Plattform zeichnen wie es in Abbildung 2.1 dargestellt ist. Die tragenden Säulen von .NET sind die .NET Enterprise Server, die neuen Endgeräte, die so genannten *Building Block Services* und das .NET Framework. Die Building Block Services sind von Microsoft zur Verfügung gestellte Web Services (bisher bekannt als *.NET My Services* bzw. unter Codenamen *Hailstorm*). Diese kann der Programmierer und Benutzer verwenden, um Standardaufgaben wie Terminverwaltung und Authentifizierung mittels existierender Lösungen zu realisieren.

2.3 .NET Framework

Das .NET Framework bildet die technische Basis für alle .NET-Anwendungen und bietet viele Neuerungen und Verbesserungen hinsichtlich der im vorhergehenden Kapitel beschriebenen Probleme existierender Lösungen wie COM/DCOM. Zudem bildet es auch die Basis für die Entwicklung und den Betrieb von XML Web Services-Anwendungen auf der Microsoft-Plattform. Das Framework wird uns im Verlauf dieses Buchs immer wieder begegnen und wir werden intensiv mit ihm arbeiten.

Einen allgemeinen Überblick über das .NET Framework können Sie sich in Abbildung 2.2 verschaffen. Als Basis für die Ausführung von Code ist die *Common Language Runtime* (*CLR*) eine der wichtigsten Komponenten im .NET Framework. Die CLR werden wir genauer in Abschnitt 2.4.1 in Augenschein nehmen. Ein .NET-Programmierer entwickelt nicht mehr mit mehreren, unterschiedlichen Klassenbibliotheken, sondern stützt sich ausnahmslos auf die neue *Base Class Library* (*BCL*) – bis auf die Ausnahmefälle, bei denen man mit Win32- oder COM-Implementierungen interagieren muss. Diese BCL bietet eine einheitliche Sicht auf das .NET Framework und seine mannigfaltige Funktionalität. Nähere Informationen zur BCL stehen in Abschnitt 2.4.3. Aufbauend auf der BCL kann ein Entwickler Datenbank- und XML-orientierte Applikationen erstellen. Diese Art von Programmen werden wir in Kapitel 4 erörtern.

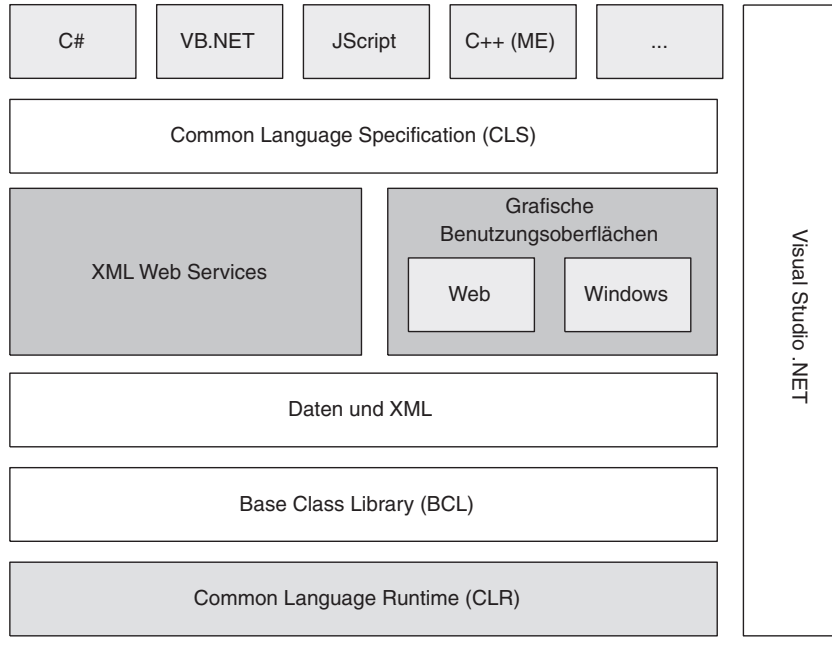


Abbildung 2.2: Das .NET Framework im Überblick

Ein großer und wichtiger Bestandteil der Programmierung mit dem .NET Framework ist die Implementierung Web-basierter Applikationen. Man kann zwar mithilfe von .NET auch Anwendungen auf Basis von Windows mit dem neuen Konzept der *Windows Forms* erstellen, allerdings liegt der Fokus auch von Seiten Microsofts klar auf der WWW-Programmierung. Diese Art von Programmen unterteilt sich grob in zwei Kategorien: Applikationen mit grafischer Oberfläche und direkter Interaktion durch den menschlichen Benutzer auf der einen und abstrakte, auf standardisierte Datenformate und –protokolle ausgelegte Programmschnittstellen – besser bekannt als Web Services oder XML Web Services – auf der anderen Seite. Wenn Sie mehr über die Programmierung von Web-basierten Oberflächen erfahren möchten, werfen Sie bitte einen Blick auf Kapitel 3. Die verbleibenden Kapitel beschäftigen sich fast ausnahmslos mit den Technologien hinter und rund um XML Web Services.

Damit diese modernen Softwareanwendungen auch in einer modernen Art und Weise implementiert werden können, stehen dem Programmierer mehrere Sprachen zur Auswahl. Diese Sprachen entsprechen den Richtlinien der *Common Language Specification* (*CLS*, siehe Abschnitt 2.4.2) und werden innerhalb von .NET als gleichberechtigt angesehen. Egal, ob Sie mit Visual Basic .NET oder der neuen Sprache C# programmieren: Im Endeffekt wird der erzeugte Code immer von der CLR ausgeführt.

In den verbleibenden Abschnitten dieses Kapitels werde ich Ihnen zunächst die wichtigsten Konzepte und Begriffe rund um das .NET Framework vorstellen. Anschließend gehen wir dann intensiver in spezielle Teilbereiche, deren Verständnis für die Programmierung mit .NET unbedingt notwendig sind.

2.4 .NET-Grundlagen für Programmierer

Viele werden sich fragen, warum eine neue Vorgehensweise und Architektur überhaupt benötigt wird. Außer den in Kapitel 1 angeführten Gründen für eine neue Entwicklungsbasis gibt es auch noch die technischen Feinheiten, die man in diesem Zusammenhang erwähnen sollte. Der Wunschtraum der Interoperabilität zwischen den Programmiersprachen und Paradigmen innerhalb der Microsoft-Plattform hatte mit COM und DCOM ein etwas hinkendes Bein. .NET räumt nun mit vielen der auf Kompromissen aufsetzenden Notlösungen von COM auf und führt eine allgegenwärtige Laufzeitumgebung ein. Auf deren Basis laufen sämtliche Softwareentwicklung und die Ausführung von Programmen ab.

Wenn wir uns den Ablauf einer Programmentwicklung unter .NET ansehen (vgl. Abbildung 2.3), dann stellen wir fest, dass alles in der CLR mündet. Sämtliche Implementierungen in C#, Visual Basic .NET oder JScript werden in eine Zwischendarstellung kompiliert (die so genannte *Intermediate Langu-*

age, *IL* oder *MSIL*). Diese IL wird in ausführbare und versendbare Pakete verpackt, die so genannten *Assemblies*. Diese Assemblies werden dann von der CLR geladen und ausgeführt. Eine detailliertere Beschreibung dieses Vorgangs finden Sie in den folgenden Abschnitten.

Dies hat zur Folge, dass man mit Visual Basic .NET keine nativen Windows-Programme (Win32) mehr erstellen kann (im Grunde wird auch klassisches Visual Basic in einen Zwischencode – den P-Code - übersetzt, welcher anschließend von der Visual Basic-Laufzeitumgebung interpretiert wird). Sämtliche Kompilierung geht in IL-Code über. Einzig mit Visual C++ kann man im Rahmen des Visual Studio .NET noch originale Win32-Applikationen programmieren. Diese werden dann eben nicht in MSIL umgesetzt, sondern münden wie gewohnt in einer Assembler-Darstellung der Ziel-Prozessorarchitektur.

Jeglicher von der CLR verwalteter und ausgeführter Code heißt *managed Code* (verwalteter Code).

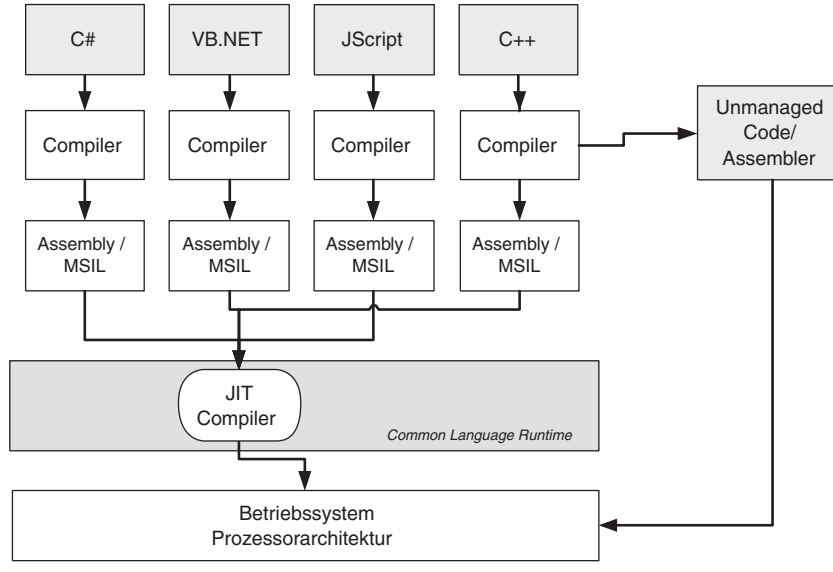


Abbildung 2.3: Entwicklungsablauf unter .NET

Wir wollen nun einen Blick auf die Common Language Runtime und die essentiellen Bestandteile der neuen .NET-Architektur werfen.

2.4.1 Common Language Runtime (CLR)

Ein Ziel der Common Language Runtime ist die Vereinheitlichung der unterschiedlichen Laufzeitumgebungen in der Windows-Welt. Wenn man sich den Wandel der Laufzeitsysteme über die letzten Jahre hinweg betrachtet (vgl. Abbildung 2.4), dann stellt man einen Trend zu einer einzigen, homogenisierten Laufzeitumgebung fest.

Bei der Programmierung unter Windows NT 4.0 hatte man es prinzipiell noch mit drei Laufzeitsystemen zu tun: Die COM-Runtime, die MTS-Runtime und die verschiedenen Laufzeitumgebungen für die jeweilige Programmiersprache. Mit dem Einzug von COM+ unter Windows 2000 hat sich diese Situation aber verbessert. Die Laufzeitsysteme von COM und des MTS wurden gegenseitig integriert und vereinheitlicht. Das Ziel von .NET ist es nun auf mittelfristige Sicht hin, nur eine einzige Laufzeitumgebung, eben die CLR, zu etablieren. Zum Zeitpunkt der Version 1.0 von .NET ist dies natürlich noch nicht gelungen, da die Basisbetriebssysteme ja noch alle auf den herkömmlichen Technologien basieren. Daher wird es wohl in den ersten Jahren eine Mixtur aus CLR (repräsentiert durch *MSCOREE.DLL* und *MSCORLIB.DLL*) und vor allem der für COM und COM+ stehenden *OLE32.DLL* geben.

Microsoft hat die Spezifikationen für eine allgemeine Laufzeitumgebung (*Common Language Infrastructure, CLI*) und für die Programmiersprache C# einem Standardisierungsgremium übergeben. Dieses Gremium heißt ECMA und überprüft diese Spezifikationen auf Hersteller- und Plattformunabhängigkeit. Sämtliche Dokumente sind öffentlich verfügbar und einsehbar.

Die CLI ist somit die grundsätzlich gültige Spezifikation und das theoretische Rahmenwerk für die allgemeine Laufzeitumgebung, während die Common Language Runtime die spezielle Implementierung der CLI von Microsoft für Windows ist. Es arbeiten einige andere Hersteller bzw. Open Source-Projekte an CLI-Implementierungen für alternative Betriebssysteme. Hierzu zählt beispielsweise das *Mono*-Projekt für UNIX- und Linux-Distributionen (Website: *http://www.go-mono.com/*).

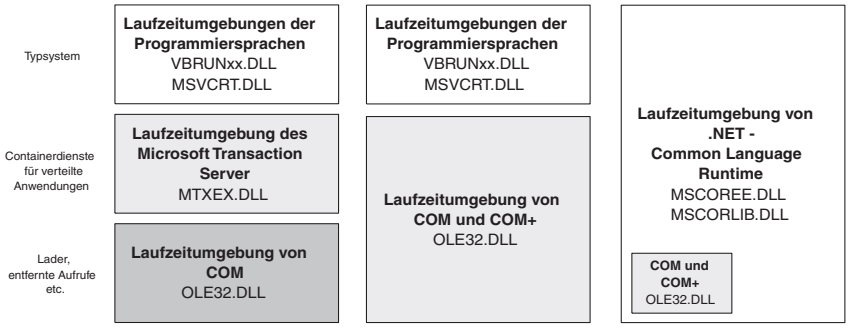


Abbildung 2.4: Vereinheitlichung der Laufzeitsysteme

Damit eine auf .NET basierende Applikation überhaupt ausgeführt werden kann, muss sich auf jedem Ziel-Computer die CLR befinden. Hierfür bietet Microsoft ein so genanntes *.NET Redistributable* an, welches ein Größe von ca. 21 MB aufweist. Es ist anzunehmen, dass dieses Redistributable in künftige Windows Service Pack- und Internet Explorer-Versionen als optionales Paket mit einfließen wird.

Die CLR muss in aktuellen Windows-Versionen in so genannten Host-Umgebungen ablaufen. Ein Host ist eine Wirtsapplikation, die vom Betriebssystem gestartet werden kann, und welche dann ihrerseits die CLR lädt und ausführt. Folgende CLR-Hosts sind verfügbar:

- Windows Shell (Kommandozeile, Explorer),
- Internet Explorer,
- ASP.NET,
- Eigene Host-Implementierungen

Für herkömmliche Windows-EXE-Anwendungen sorgt ein kleiner *Stub* für den Aufruf der CLR. Dieser Stub besteht aus nativen Assembler-Anweisungen, die für das Laden der CLR sorgen. In Windows .NET Server und allen zukünftigen Versionen von Windows wird der Stub nicht mehr nötig sein. Diese Betriebssysteme werden neue *OS Loader* besitzen, die ohne den Stub auskommen. Der OS Loader sorgt für das Laden und die Ausführung von Dateien im *PE-Format* (*Portable Executable*) unter Windows.

Man kann sich relativ einfach auch seinen eigenen CLR-Host erstellen. Dies liegt daran, dass die CLR auch über COM-Schnittstellen ansprechbar ist. Das wichtigste zugehörige COM-Objekt heißt *CorRunTimeHost*.

Schauen wir uns die CLR einmal etwas genauer an. Wie wir weiter unten in Abschnitt 2.5.1 sehen werden, verwendet .NET und somit die CLR so genannte Assemblies als Verpackungs- und Ausführungseinheiten. Damit die CLR auch alle Assemblies auffindet, die an der Ausführung eines Programms beteiligt sind, sorgt der *Assembly Resolver* dafür, dass die benötigten und korrekten Assemblies nach bestimmten Richtlinien gefunden werden und vorliegen (siehe Abbildung 2.5). Ist die entsprechende Assembly gefunden, tritt der Klassenlader in Aktion und lädt alle benötigten Typen aus der Assembly. Der Assembly Resolver wird auch oft als *Fusion* bezeichnet.

Fusion ist also die Technologie, welche in .NET für das Auffinden und Laden von Assemblies verantwortlich ist. Mit Fusion soll die alte Problematik der DLL-Hölle beseitigt oder zumindest entschärft werden. Die DLL-Hölle ist eine Umschreibung für den Zustand des Wildwuchses von externen Bibliotheken und Ressourcen unter Windows. Mit der CLR und Fusion hat Microsoft nun einen neuen Weg eingeschlagen. In der klassischen Win32-Umgebung werden externe Module entweder dynamisch mittels *LoadLibraryEx* bzw. *CoCreateInstanceEx* (im Falle von COM) geladen. Alternativ werden sie statisch durch Import-Bibliotheken dazu gebunden. Der von *LoadLibraryEx* verwendete Suchalgorithmus kann allerdings zu falschen DLL-Versionen führen. Abhilfe schafft hier das Kopieren der DLLs in das Verzeichnis *system32* der Windows-Installation. Hierdurch werden aber eventuell vorhandene ältere Versionen überschrieben, was zu Problemen mit anderen Programmen oder bei der Deinstallation führen kann. Alternativ kann man Win32 durch eine *.local*-Datei anweisen, dass nur die DLLs geladen werden, die sich im gleichen Verzeichnis wie die Anwendung befinden. Allerdings hat dieser letzte Schritt das Problem, dass die DLLs so nicht gemeinsam von mehreren Programmen genutzt werden können.

Mit dem Einzug von COM wanderten die Informationen über den Ort der Implementierung in die Windows-Registrierungsdatenbank. Jede COM-Klasse hat ihre *ClassID* und diese ID zeigt auf die eigentliche Implementierung. Werden Änderungen am Code vorgenommen, dann muss – laut Regeln von COM – eine neue ClassID erzeugt werden. Dieser Umstand bedeutet aber vor allem bei kleineren Änderungen sehr viel Aufwand. Es gibt unter COM so etwas Ähnliches wie *side-by-side*-Installationen. Allerdings wird hierdurch die Registrierung unnötig erschwert. Alles in allem ist die Vorgehensweise mit der Festschreibung von Metadaten über Typen und ausführbare Dateien in der Registry keine gute Lösung.

Mit Fusion erhält nun ein neuer Ansatz Einzug in die .NET-Welt. Fusion schaut immer zuerst im lokalen Verzeichnis und allen Unterverzeichnissen nach einer Assembly. Dieses Verhalten kann aber durch Konfigurationsdateien überschrieben werden. Will man eine Assembly mit anderen Anwendungen teilen, dann muss diese in den *Global Assembly Cache* (*GAC*) installiert werden (siehe Abschnitt 2.5.1). Fusion sieht automatisch im GAC nach und ermöglicht somit eine echte *side-by-side* Installation von Assemblies.

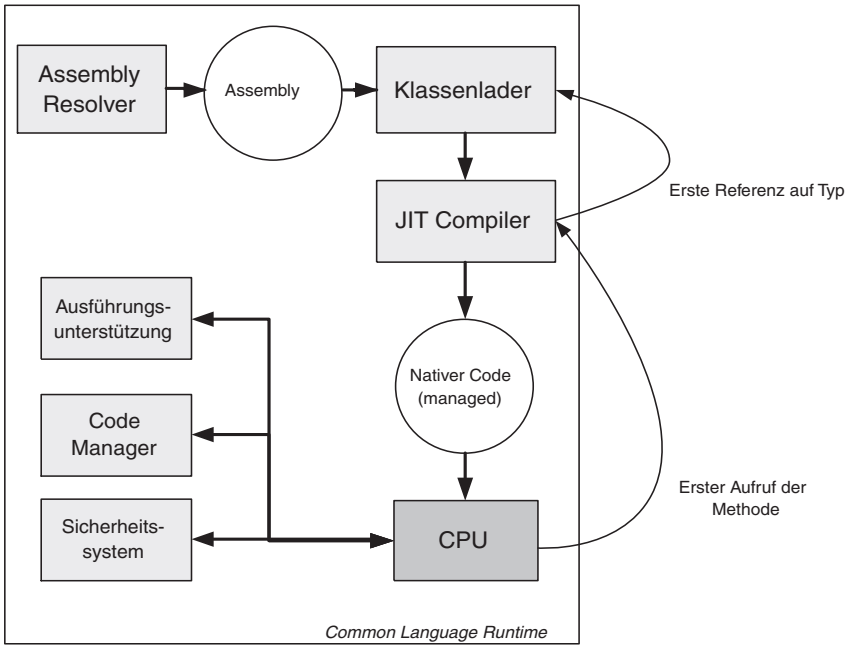


Abbildung 2.5: Kontrollfluss innerhalb der CLR

Wenn nun also die Assembly und die Typen geladen sind, sorgt der *Just-In-Time* (*JIT*) *Compiler* dafür, dass der in MSIL vorliegende Code in eine native Darstellung übersetzt wird. Der JIT Compiler arbeitet im Übrigen auf Methodenbasis. So wird bei jedem ersten Aufruf einer Methode diese durch den JIT Compiler geführt, ehe sie in nativer Form vorliegt. Diese native Darstellung wird dann von der CLR-CPU ausgeführt. Hierbei arbeitet die CPU Hand in

Hand mit einigen Hilfskomponenten, wie z. B. dem Sicherheitssystem. Denn die CLR ist immer auf die sichere Ausführung von Code bedacht, um etwaige Pufferüberläufe oder unberechtigte Zugriffe und Aktionen zu unterbinden.

2.4.2 Einheitliches Typensystem

Um wieder auf die Probleme aus der COM-Welt zu sprechen zu kommen: Eine Vielzahl an Typen in jeder Programmiersprache machte die Interoperabilität über COM nicht gerade einfach. Für .NET und die CLI hat Microsoft das so genannte *Common Type System* (*CTS*) geschaffen. Das CTS besteht aus einer Anzahl von Typen, die .NET und die CLR verstehen. Die Typen des CTS können von allen .NET-Anwendungen genutzt werden. Allerdings muss erwähnt werden, dass nicht alle Programmiersprachen und deren Compiler die CTS unterstützen. Für die gemeinsame Definition einer Untermenge des CTS, welche alle Sprach- und Compilerhersteller unterstützen müssen, existiert die Common Language Specification (CLS). Die Grundidee hinter der CLS ist, dass alle Sprachen, die CLS-kompatible Typen verwenden, mit anderen .NET-Sprachen und -Programmen interagieren können. So kann beispielsweise eine Visual Basic .NET-Klasse von einer C#-Klasse erben, weil beide CLS-konform sind. Den Unterschied zwischen CTS und CLS habe ich in Abbildung 2.6 grafisch noch einmal dargestellt.

Abbildung 2.6: CLS und CTS im Vergleich

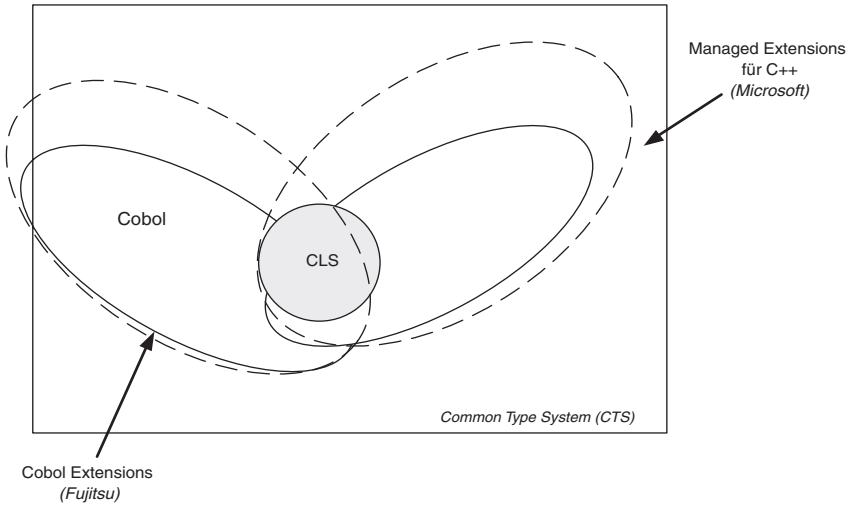


Abbildung 2.6 zeigt auch, dass die *Managed Extensions for C++ (ME)* eine Obermenge der Standard-ANSI-C++-Spezifikation darstellen. Mit den ME ist C++ in der Lage, CLR-konforme Programme zu erstellen. Allerdings muss man dabei beachten, dass diese Programme nicht mehr kompatibel mit C++-Anwendungen gemäß dem C++-Standard sind. Ähnlich sieht die Sachlage mit der .NET-Version für Cobol aus, die von Fujitsu vertrieben wird.

Wert- und Referenztypen

Das CTS definiert verschiedene Typen, von denen ich nun die wichtigsten vorstellen möchte. Prinzipiell unterscheidet .NET zwischen zwei großen Typfamilien: Den Werttypen (*value types*) und den Referenz- oder Verweistypen (*reference types*). Diese Aufteilung wird in Abbildung 2.7 grafisch in einer Baumstruktur verdeutlicht.

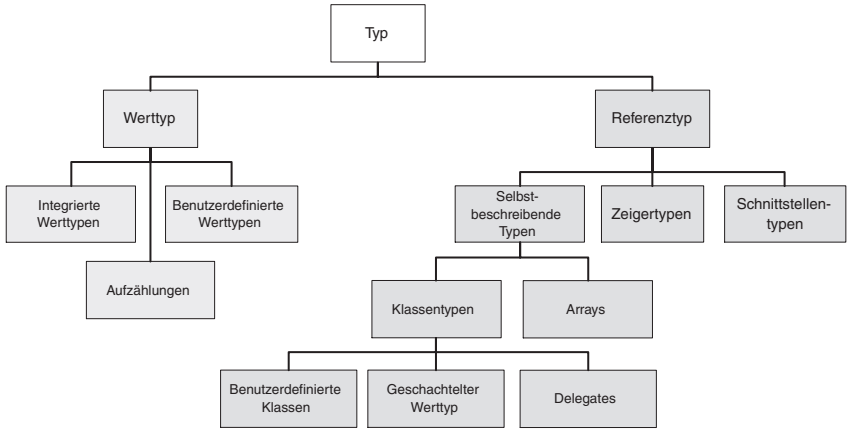


Abbildung 2.7: CLR-Typsystem

Die beiden Typarten unterscheiden sich primär in ihrem Verhalten im Speicher. Wie jeder handelsübliche Prozessor hat auch die CLR zwei maßgebende Speicherbereiche für die Haltung von Daten und die Abarbeitung von Befehlen: den *Stack* (zu deutsch Stapel- oder Kellerspeicher) und den *Heap* (zu deutsch Halde). Werttypen werden immer auf dem Stack angelegt und können nicht Null sein. Wenn ein Werttyp einer Funktion übergeben wird, dann wird immer eine Kopie des Werts angelegt, bevor er der Funktion übergeben wird. Dadurch ändert sich der ursprüngliche Wert nicht. Da die meisten eingebauten Werttypen klein sind, ist die Erstellung einer Kopie kein echtes Problem. Zu den eingebauten Werttypen zählen primitive Typen wie Integer, Strukturen und Aufzählungen. Das Beispiel für einen Werttyp in Abbildung 2.8 verwendet eine Struktur, um den Unterschied zwischen Werttyp und einem Referenztyp (im Beispiel eine Klasse) zu erläutern. Ein Entwickler kann auch eigene Werttypen erzeugen, in dem er von der Klasse *System.ValueType* erbt.

Referenztypen hingegen werden auf dem Heap allokiert. Diese Typen werden vor allem dann verwendet, wenn ein Typ mehr Speicherplatz verbraucht als die simplen Datentypen. Genauer gesagt werden Referenztypen auf dem Heap gehalten und eine Referenz zu dieser Adresse wird auf dem Stack eingetragen, Referenztypen können also Null sein. Auf diese Weise bleibt der kostbare Platz auf dem Stack erhalten. Somit stammt der Name dieser Typen von den Referenzen, die sie auf dem Heap haben. Wenn man einen Referenztyp an eine Funktion übergibt, dann wird immer ein Verweis auf den Typ übergeben. Es wird gewissermaßen die Adresse des Typs und

keine Kopie weitergegeben. Einer der Vorteile von Referenztypen liegt somit klar auf der Hand: Ein Referenztyp kann als Rückgabeparameter verwendet werden, ohne dass zusätzliche Ressourcen belegt werden müssen. Allerdings haben Referenztypen auch ihre Nachteile: Da sie auf dem verwalteten Heap der CLR angelegt werden, verbrauchen sie mehr CPU-Zyklen als andere Typen. Zu den Referenztypen zählen im CTS u.a. Klassen, Schnittstellen, Arrays und Delegates (siehe weiter unten in diesem Abschnitt).

Abbildung 2.8 zeigt anhand einer Grafik genauer, wie man sich die Speicherbelegung von Wert- und Referenztypen vorzustellen hat.

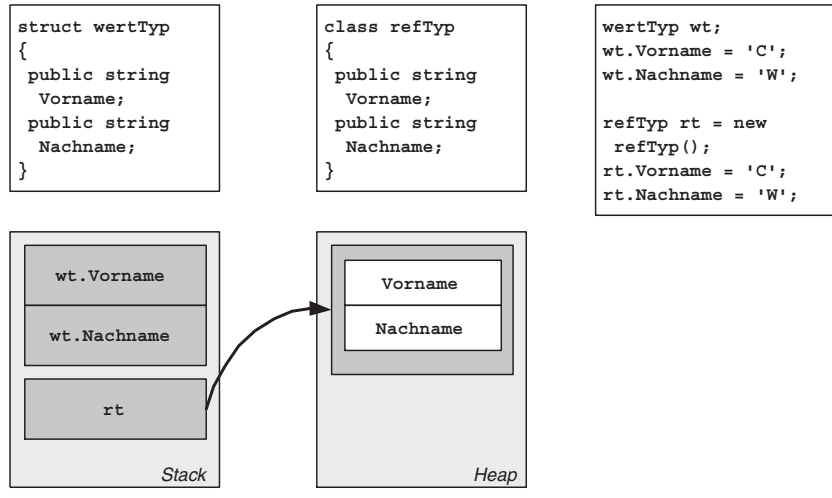


Abbildung 2.8: Speicherallokation von Wert- und Referenztypen

Microsoft bietet einen Mechanismus, mit dem man Wert- in Referenztypen und umgekehrt umwandeln kann. Denn unter der Haube ist in .NET alles ein Objekt und die Werttypen werden hauptsächlich aus Geschwindigkeitsgründen verwendet. Der Vorgang der Umwandlung eines Werttyps in einen auf dem Heap lagernden Referenztyp wird *boxing* genannt. Analog wird das Gegenstück als *unboxing* bezeichnet.

Delegates

Ein wichtiger, spezieller Referenztyp im Umfeld von .NET sind die *Delegates*. Delegates werden innerhalb des .NET Frameworks sehr oft verwendet. Sie gleichen von der prinzipiellen Funktionsweise den Funktionszeigern aus C und C++. Funktionszeiger erlauben einem Programmierer, Programme mit Eingriffspunkten zu schreiben, die dann an anderer Stelle von anderen Programmierern implementiert werden können. Somit ermöglichen Funktionszeiger und eben auch Delegates die Erstellung von erweiterbarer Software. Der Unterschied von Delegates zu den klassischen Funktionszeigern ist, dass Delegates im Rahmen von .NET typsicher sind.

Als Erstes muss man einen Prototypen für eine Rückruffunktion (*callback function*) definieren und diesen mit dem Schlüsselwort *delegate* versehen. Hierdurch wird dem Compiler angezeigt, dass es sich um einen objektorientierten Funktionszeiger handelt. Unter der Haube wird eine Klasse erzeugt, die von *System.Delegate* abgeleitet ist. Anschließend wird eine Methode mit der gleichen Signatur wie der Funktionsprototyp implementiert. Abschließend bleibt nur noch, diese Methode dem Konstruktor des Delegates zu übergeben und die Rückruffunktion indirekt aufzurufen. Ein erklärendes Beispiel für Delegates finden Sie in Listing 2.1. Hier wird keine Methode explizit definiert, sondern die *ToString*-Methode verwendet, die für jedes Objekt existiert. Das Beispiel ist ein über Delegates realisierter Weg, um die *ToString*-Methode aufzurufen.

```
using System;

public class DelegateDemo
{
   private delegate string WandleInZeichenkette();

   static void Main(string[] args)
   {
      int a = 74;
      WandleInZeichenkette zA = new
         WandleInZeichenkette(a.ToString);
      Console.WriteLine(zA());
   }
}
```

Listing 2.1: Einfaches Beispiel für Delegates

Ausnahmen

Eine weitere wichtige Eigenschaft ist die gleiche Behandlung von Fehlern über Sprachgrenzen hinweg. In COM werden Fehler nicht über Ausnahmen (*exceptions*), sondern über Fehler-Codes, die so genannten *HRESULTs*, propagiert. In Visual Basic kommt man gar nicht an diese Werte heran und kann Fehlerbehandlung nur eingeschränkt implementieren.

.NET verwendet im Prinzip das gleiche Paradigma zur Behandlung von Fehlern wie Java oder C++. Aber im Gegensatz zu C++ bedeutet die Nutzung von Exceptions nicht unbedingt einen Mehraufwand, da das Typsystem daraufhin optimiert wurde. Eine Klasse oder Anwendung kann einen Fehler anzeigen, indem sie eine Ausnahme erzeugt. Diese Ausnahme ist immer abgeleitet vom Typ *System.Exception*. Eine Klasse, die diesen Typ verwendet, kann über einen *try/catch/finally*-Block etwaige Fehler abfangen und entsprechend darauf reagieren. Dieses Konzept gilt gleichwohl in Managed C++, Visual Basic .NET und allen anderen .NET-Sprachen.

2.4.3 Base Class Library

Wie weiter oben bereits angedeutet, wird bei der Entwicklung mit .NET immer wieder auf die Base Class Library (BCL) zugegriffen. Die BCL ist eine umfangreiche und mächtige Klassenbibliothek, die Microsoft mit dem .NET Framework ausliefert. Im Gegensatz zur bisherigen Entwicklung mit Visual Basic oder Visual C++ muss ein .NET-Programmierer nicht mehr mehrere unterschiedliche Bibliotheken kennen und beherrschen. Die Unterschiede beispielsweise zwischen der *Microsoft Foundation Classes* (*MFC*), der *Active Template Library* (*ATL*) und der Visual Basic Laufzeitbibliothek waren für viele Programmierer nur schwer unter einen Hut zu bekommen.

Die BCL ist in so genannte Namensräume (*namespaces*) unterteilt. Das Konzept der Namensräume ist nicht neu, sondern existiert auch in anderen Programmierumgebungen wie C++ oder Java. Der alles überspannende Namensraum in .NET heißt *System*. Ausgehend von dieser Wurzel spannt sich ein komplexer und weit verzweigter Baum von Namensräumen über die BCL.

Man darf das Konzept der Namensräume nicht mit den Assemblies verwechseln. Zwar ist es oft so, dass ein Namensraum innerhalb einer Assembly implementiert ist, allerdings ist dies keine Beschränkung. Im .NET Framework selbst gibt es zahlreiche Beispiele, in denen ein Namensraum über mehrere Assemblies verstreut ist. Auch der umgekehrte Fall, nämlich dass innerhalb einer Assembly mehrere Namensräume existieren, ist sehr häufig zu finden.

Das Schaubild in Abbildung 2.9 zeigt einen etwas detaillierteren Überblick der Base Class Library im Hinblick auf ihre Unterteilung in Namensräume.

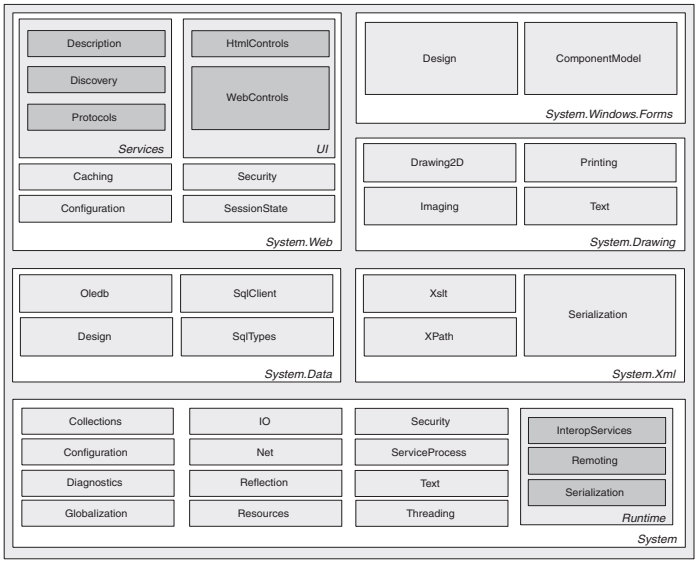


Abbildung 2.9: Wichtige Namensräume der .NET Base Class Library

Die BCL ist auf dem Schaubild in sechs große Bereiche unterteilt. Angefangen bei den essentiellen Systemfunktionen (*System*), über die Programmierung mit Daten aller Art (*System.Data* und *System.Xml*) bis hin zu den Klassen für die Web-basierte Programmierung (*System.Web*).

Das Konzept der Namensräume in .NET ist auch erweiterbar. Ein Programmierer kann seine eigenen Namensräume definieren und nutzen.

2.5 Der Blick ins Innere

Nach diesem ersten Blick auf das .NET Framework werden wir in den kommenden Abschnitten etwas tiefer eintauchen und wichtige Techniken kennen lernen, die für die Arbeit mit dem .NET Framework unerlässlich sind.

2.5.1 Assemblies

Hat man im Zusammenhang mit der COM-basierten Entwicklung immer wieder von COM-Komponenten gesprochen, so ändert sich die Sprechweise bei .NET etwas. Unter COM-Komponenten versteht man die Verpackungseinheit von COM-Schnittstellen und -objekten. Unter .NET bezeichnet man die Verpackungseinheit für Code nun als Assembly.

Definition

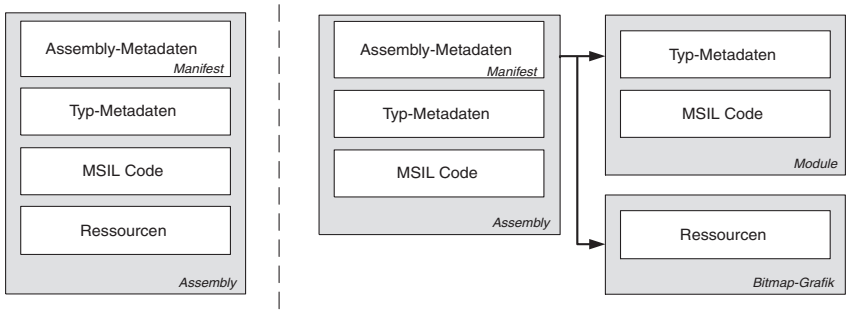
Eine Assembly ist aber nicht primär als physikalische Einheit im Dateisystem zu sehen, sondern es handelt sich vielmehr um einen logischen Verbund mehrerer Ressourcen aus dem .NET-Umfeld. Denn Assemblies haben einen ganz großen Vorteil gegenüber COM-Komponenten oder anderen Win32-Dateien. Sie sind in sich abgeschlossen und selbstbeschreibend. Dies bedeutet, dass jegliche Informationen, Daten und Metadaten, die man für die .NET-Programmierung auf Basis einer Assembly benötigt, innerhalb der Assembly verfügbar sind. Somit ist keine Registrierung jedweder Daten in der Registry notwendig.

Üblicherweise besteht eine Assembly aus Metadaten der Assembly selbst, aus den Metadaten der sich in der Assembly befindenden .NET-Typen, aus dem eigentlichen Code in Form von MSIL und aus Ressourcen, die beispielsweise aus Bildern oder Zeichenketten bestehen können. Diesen am häufigsten auftretenden Fall bezeichnet man auch als Eindatei-Assembly (siehe Abbildung 2.10, linke Seite) und diese Eindatei-Assembly wird dann auch auf eine Datei im Dateisystem abgebildet. Alternativ kann eine Assembly bzw. deren Bestandteile über mehrere Dateien verstreut sein. Dann spricht man von Mehrfachdatei-Assemblies (siehe Abbildung 2.10, rechte Seite). Bei diesem Modell sind z. B. Ressourcen wie Bilder in externe Dateien ausgelagert und diese Ressourcen werden innerhalb eines bestimmten Bereichs der Assembly - dem *Manifest* (siehe unten) - referenziert. Diese externen Res-

sourcen können aber auch in so genannten Modulen (*modules*) vorliegen. Allgemein gesprochen, besteht eine Assembly aus einer Anzahl von Modulen, die über das Assembly Manifest zusammen gehalten werden.

Eine Assembly bzw. der Hauptteil einer Assembly mit dem Manifest kann in Form einer *.exe*-Datei oder einer *.dll*-Datei im Windows-Dateisystem vorliegen. Die Klassenbibliotheken der Base Class Library liegen beispielsweise in Form zahlreicher DLLs im .NET-Installationsverzeichnis vor (z.B. *System.Web.Services.dll*).

Abbildung 2.10: Zwei verschiedene Arten von Assemblies

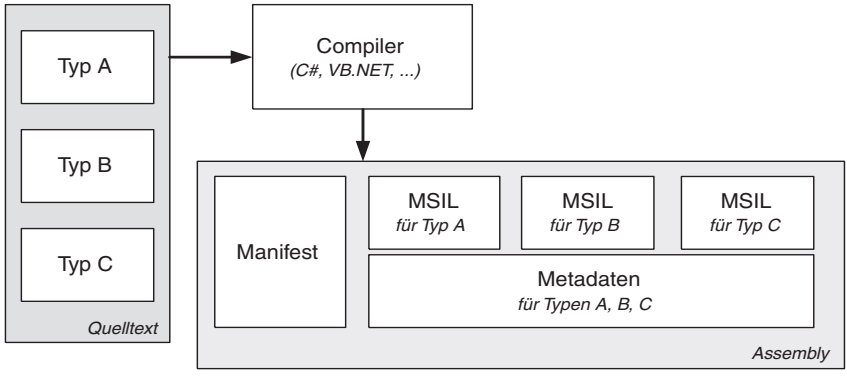


Nachdem wir nun gesehen haben, was Assemblies sind und wie sie aussehen können, kümmern wir uns im Folgenden um deren Erstellung.

Erzeugung

Eine Assembly wird durch einen der .NET-Compiler erzeugt. Der allgemeine Ablauf bei der Kompilierung der Quelltexte in eine ausführbare Form ist in Abbildung 2.11 dargestellt. Die resultierende Assembly enthält bei der Standardausgabe des Compiler-Aufrufs also die Daten für jeden im Quelltext vorkommenden Typ, die oben erwähnten Metadaten für die Typen und das Manifest.

Abbildung 2.11: Kompilierung von Quelltext in Assemblies



Das Assembly Manifest ist ein spezieller Bereich innerhalb einer Assembly und enthält u.a. eine Auflistung der Elemente der Assembly und wie sie miteinander verknüpft sind. Zusätzlich sind im Manifest noch Daten über die Version und die Kultur festgehalten (vgl. Abbildung 2.15).

Die im .NET Framework befindlichen Compiler sind:

- *vbc.exe* – für Visual Basic .NET,
- *csc.exe* – für C#,
- *jsc.exe* – für JScript .NET.
- *cl.exe* – für C/C++ (*managed* und *unmanaged*)

Damit kann man also ohne den Erwerb einer teuren Lizenz mit dem Programmieren beginnen, allerdings nur mit der Kommandozeile (siehe Abbildung 2.12).

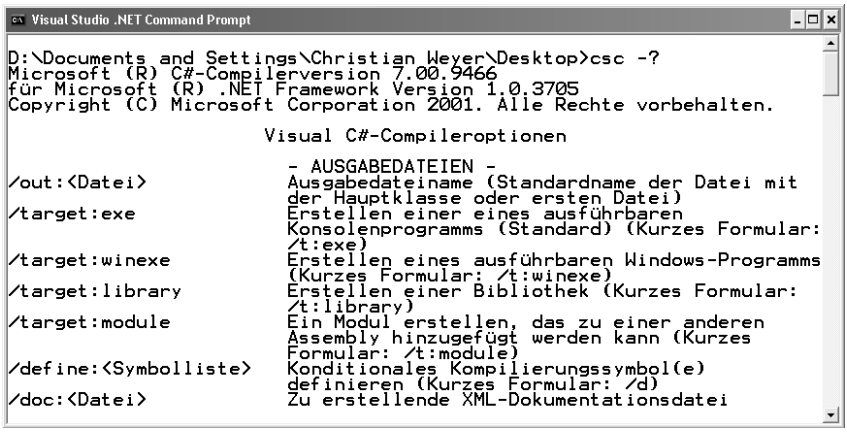


Abbildung 2.12: C#-Compiler csc.exe in der Kommandozeile

Der C++-Compiler *cl.exe* aus dem Framework SDK befindet sich nicht im Standardinstallationsverzeichnis wie die anderen Compiler. Vielmehr ist er in einem Unterordner von *Microsoft Visual Studio .NET* zu finden, auch wenn Visual Studio gar nicht installiert wurde

Die Ausgabe des Compilers kann über Schalter beeinflusst werden. Wir schauen uns den C#-Compiler als Beispiel an. In Tabelle 2.1 sind die möglichen Werte für den Schalter */target* aufgeführt. Dieser Schalter sorgt dafür, dass dem Compiler mitgeteilt wird, welche Art von Datei er aus den Quelltexten erstellen soll.

Tab. 2.1: Mögliche Ausgabeformate des C#-Compilers

Schalterwert	Beschreibung
/target:exe	Erstellen eines ausführbaren Konsolenprogramms als *.exe*-Datei (Standard) (Kurzform: */t:exe*).
/target:winexe	Erstellen eines ausführbaren Windows-Programms als *.exe*-Datei (Kurzform: */t:winexe*).
/target:library	Erstellen einer Bibliothek als *.dll*-Datei (Kurzform: */t:library*).
/target:module	Erstellen eines Moduls, das zu einer anderen Assembly hinzugefügt werden kann (Kurzform: */t:module*).

Wenn wir uns das einfache *Hello World*-Programm aus Listing 2.2 her nehmen, können wir es mit folgender Kommandozeile übersetzen:

```
csc.exe /t:exe HelloWorld.cs
```

Als Resultat ergibt dieser Aufruf eine *.exe*-Datei mit der vollständigen .NET Assembly. Diese Assembly können wir uns nun einmal genauer betrachten.

Damit die Werkzeuge aus dem .NET Framework SDK in der Kommandozeile erkannt werden, müssen mehrere Pfade in die Umgebungsvariable *Path* eingefügt werden. Dies kann man manuell erledigen oder die Stapeldatei *corvars.bat* (im *bin*-Verzeichnis des SDKs) verwenden. Wenn man zusätzlich noch Visual Studio .NET installiert hat, gibt es das VISUAL STUDIO .NET COMMAND PROMPT, mit dem man gleich loslegen kann, denn hier werden die Umgebungsvariablen bereits durch die Datei *vsvars32.bat* gesetzt.

```
using System;

class HelloWorld
{
   static void Main(string[] args)
   {
      Console.WriteLine("Hallo, dies ist .NET!");
   }
}
```

Listing 2.2: Hello World-Programm in C#

Mit dem Werkzeug *al.exe* (*Assembly Linker*) kann man unterschiedliche Module – die beispielsweise von unterschiedlichen Compilern erzeugt wurden – zu einer Assembly mit Assembly Manifest zusammenfügen.

Aufbau

Die oben erzeugte Assembly *HelloWorld.exe* wollen wir uns in diesem Abschnitt genauer betrachten. Für die Inspektion von Assemblies liefert das .NET Framework SDK das Kommandozeilenwerkzeug ILDASM (*ildasm.exe*) mit.

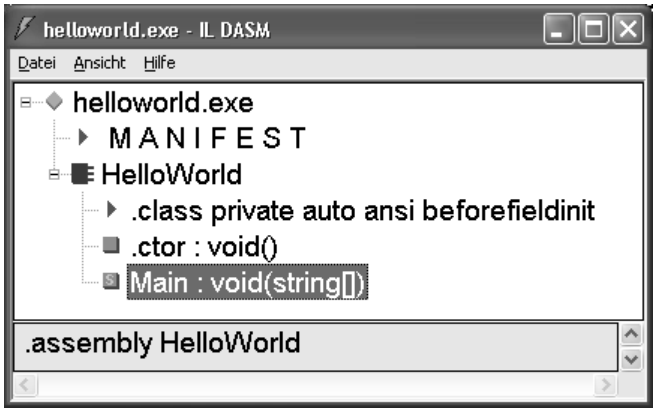


Abbildung 2.13: Assembly in ILDASM

Das Bildschirmfoto aus Abbildung 2.13 zeigt uns die Assembly und deren Inhalt in einer Baumstruktur. Mehr über die Bedeutung der einzelnen Symbole und über IL-Code können Sie in Abschnitt 2.5.2 nachlesen. Wenn wir direkt unterhalb des Assembly-Namens auf MANIFEST klicken, dann erscheint die tatsächliche Darstellung des Manifests in einem neuen Fenster (vgl. Abbildung 2.14). Dies ist die textuelle Darstellung der Manifest-Informationen wie sie grafisch in Abbildung 2.15 dargestellt sind.

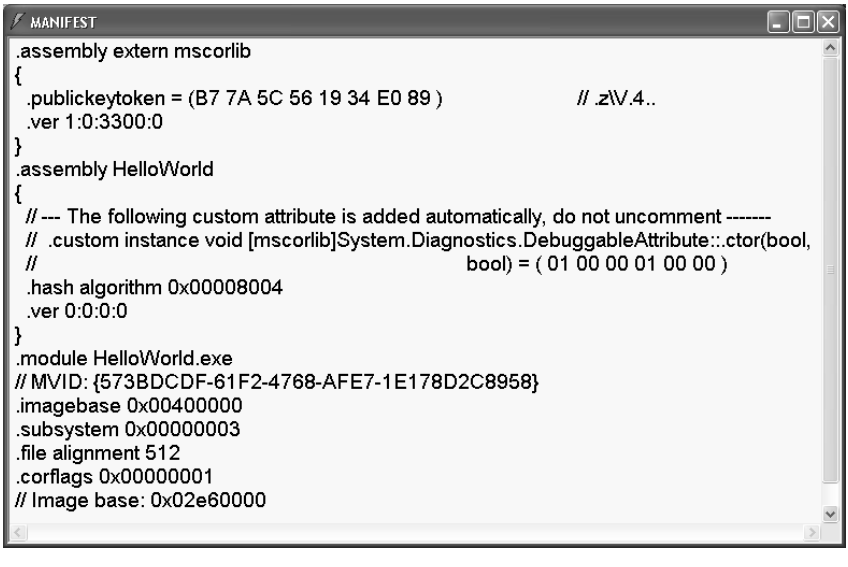


Abbildung 2.14: Assembly Manifest in ILDASM

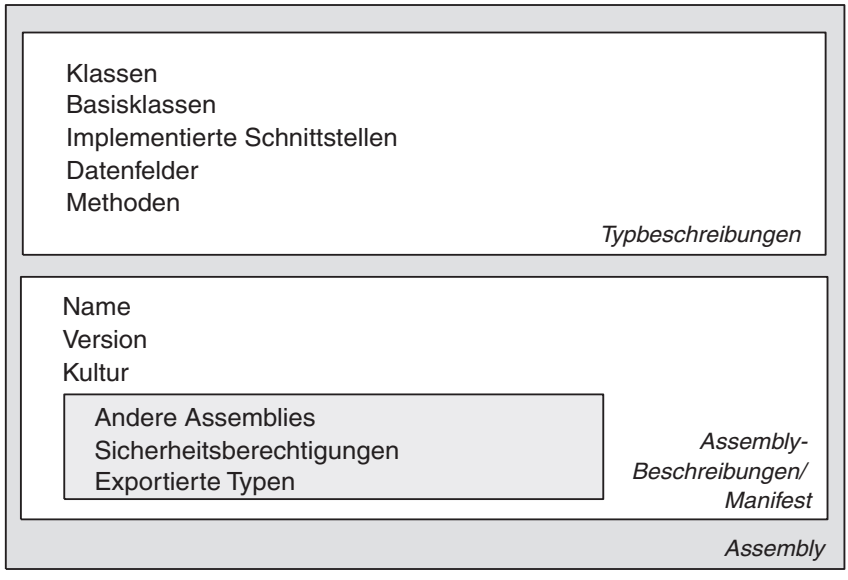


Abbildung 2.15: Informationen in einer Assembly

Man kann mit Tools wie ILDASM oder dem *.NET Reflector* von Lutz Roeder die mit .NET-Technologie erstellten Assemblies ansehen. Man sieht den kompletten MSIL-Code. Wenn man sich mit der MSIL-Syntax beschäftigt, ist man sehr schnell in der Lage, ein Reverse Engineering erster Güte zu betreiben.

Noch einfacher macht es das Werkzeug *Anakrino* für den Entwickler: Anakrino ist ein so genannter Decompiler und versucht anhand der IL-Darstellung des Codes, eine lesbare C#- oder vergleichbare Sprachvariante anzubieten.

Abhilfe können hier die so genannten Obfuskatoren (*obfuscators*) schaffen. Sie wenden spezielle Algorithmen auf eine Assembly an, um den IL-Code in eine unleserliche, aber immer noch gültige Form zu bringen. Selbst die Anwendung von Akarino funktioniert dann entweder gar nicht mehr oder liefert sinnlose Ergebnisse.

Nutzung

Sind die Assemblies erst einmal erstellt, dann müssen Client-Anwendungen häufig mit Typen aus diesen Assemblies arbeiten. Wenn wir z. B. einen sicheren und schnellen Verschlüsselungsalgorithmus mit C# programmiert haben, dann wollen andere Programme diesen Algorithmus verwenden. Also kompilieren wir den Quelltext in eine DLL-Assembly und stellen sie den Clients zur Verfügung.

Die übliche Vorgehensweise innerhalb von .NET ist die lokale Referenzierung und Verwendung von Assemblies. Dies bedeutet, dass eine Assembly im gleichen Verzeichnis wie die Anwendung oder in einem zugehörigen Unterverzeichnis vorhanden sein muss, um sie benutzen zu können. Damit hat auch nur diese Anwendung Zugriff auf die Assembly – denn die anderen Applikationen in anderen Verzeichnissen wissen nichts von und über diese Assembly. Allerdings kann die Suche nach geeigneten Assemblies über die Angabe eines Eintrags in einer XML-basierten, anwendungsspezifischen Konfigurationsdatei kontrolliert werden. Dieses Element heißt *assemblyBinding*. Eines der möglichen Unterelemente heißt *codeBase* und kann anzeigen, an welcher Stelle die Assembly mit einer bestimmten Versionsnummer gefunden werden kann. Es ist sogar die Angabe eines HTTP-URL möglich. Somit wird der automatische Download von Assemblies durch die CLR ermöglicht. Die hierdurch entstehenden Sicherheitsprobleme können Sie in Kapitel 7 nachlesen. Diese Art von Assemblies werden auch als private Assemblies bezeichnet.

Es kommt aber des Öfteren vor, dass eine Assembly und deren Funktionalität anwendungsübergreifend und somit systemweit verwendet werden muss. Hier passt unser Beispiel vom Verschlüsselungsalgorithmus besonders gut. Wie wir weiter oben bereits gelernt haben, ist aber keine Registrierung einer Assembly in der Registry notwendig. Vielmehr werden diese globalen Assemblies in einem systemweit verfügbaren Speicherbereich bekannt gemacht: Dem *Global Assembly Cache* (*GAC*).

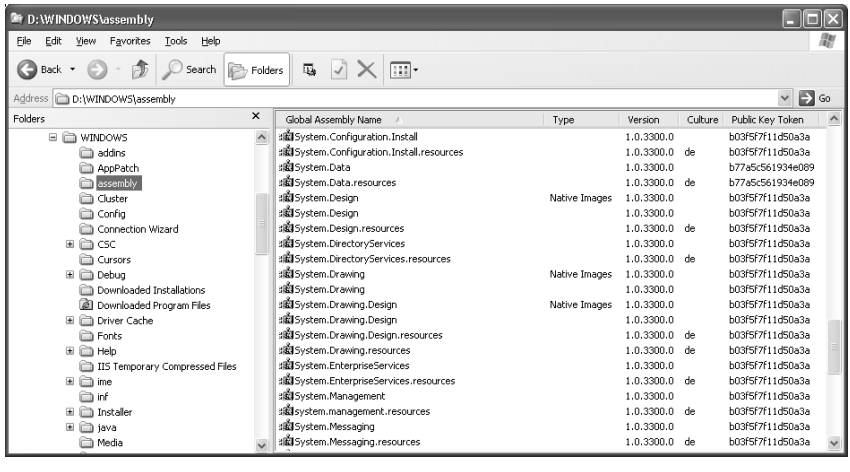


Abbildung 2.16: Der Inhalt des Global Assembly Caches im Windows Explorer

Der GAC wird von der Win32-DLL *fusion.dll* implementiert und speichert die relevanten Assemblies im Verzeichnis *<WINDOWS>\Assembly* – also z.B. *c:\winnt\Assembly*. Wenn man bei installiertem .NET Framework mit dem Windows Explorer auf dieses Verzeichnis geht, dann sieht man eine übersichtliche Anordnung sämtlicher Assemblies, die sich im GAC befinden (vgl. Abbildung 2.16). Diese Ansicht wird durch eine *Windows Shell Extension*

namens *shfusion.dll* erzeugt. Dies ist eine COM-Komponente, die man auch deregistrieren kann. Wenn man dies tut kann man auch über den Windows Explorer sehen, dass der GAC aus einer Vielzahl von Unterverzeichnissen für jede Assembly besteht. Wird eine Assembly in mehreren Versionen in den GAC installiert, dann wird für jede neue Version ein eigenes Unterverzeichnis angelegt. Diese Sicht erhält man übrigens auch, wenn man in der Kommandozeile zu besagtem Verzeichnis navigiert.

Damit ein Entwickler oder ein Administrator eine Assembly im GAC registrieren kann, benötigt diese Assembly einen starken Namen (*strong name*). Ein starker Name setzt sich aus der Identität der Assembly – dem einfachen Textnamen, der Versionsnummer und den Kulturdaten (sofern vorhanden) – sowie einem öffentlichen Schlüssel und einer digitalen Signatur zusammen. Das kryptografische Schlüsselpaar wird mit dem Werkzeug *sn.exe* erzeugt. Das folgende Kommando erzeugt ein Schlüsselpaar und speichert es in der Datei *keys.snk* ab:

```
sn.exe -k keys.snk
```

Ist dieses Schlüsselpaar einmal generiert, müssen wir unserer noch zu erstellenden Assembly mitteilen, wie sie bzw. der Compiler an die wichtigen Schlüsseldaten kommt. Denn eine Assembly kann einen starken Namen nur bei der ersten Erstellung erhalten – später unter normalen Umständen nicht mehr. Diese Informationen halten wir im Quelltext mit folgendem Attribut fest:

```
[assembly:AssemblyKeyFile("keys.snk")]
```

Wenn der Quelltext nun in eine Assembly kompiliert wird, dann werden die Schlüsseldaten aus der referenzierten Datei mit eingebunden und die resultierende Assembly besitzt einen starken Namen. Alternativ können Sie auch das Werkzeug *al.exe* (*Assembly Linker*) verwenden. Allerdings arbeitet dieses nur auf Basis von Modulen.

Die Registrierung im GAC erfolgt dann abschließend über die Kommandozeilenanwendung *gacutil.exe*. Man kann die erzeugte Assembly auch per Drag&Drop in das oben erwähnte Assembly-Verzeichnis ziehen. Mit dem folgenden Aufruf wird eine Assembly mit dem Namen *SuperSicher.dll*, die einen starken Namen besitzt, in den Global Assembly Cache installiert:

```
gacutil.exe /i SuperSicher.dll
```

Versionierung

Eine Assembly kann in mehreren Versionen im System vorliegen. Die zugehörige Versionsnummer ist gleichzeitig auch Bestandteil ihrer Identität. Wenn eine neue Funktion zu einer Bibliothek hinzugefügt und eine alte Funktion gelöscht wird, dann erstellt man eine neue Version der Assembly. Die Angabe einer Versionsnummer erfolgt am einfachsten durch die Anwendung eines Attributs wie im folgenden Beispiel:

```
[assembly:AssemblyVersion("2.0.3.7")]
```

Wenn man mit einem der .NET-Compiler eine Assembly ohne explizite Angabe einer Version erzeugt, dann wird immer die Version 0.0.0.0 erstellt. Doch was bedeuten die einzelnen Zahlen? Schauen wir uns die Erklärung der einzelnen Stellen in Abbildung 2.17 an. Die physikalische Unterteilung in vier Zahlenstellen wird logisch auf drei Möglichkeiten abgebildet. Die erste Stelle ist die Hauptversion, gefolgt von der Nebenversion. Wenn diese beiden Stellen bei zwei Assemblies unterschiedlich sind, dann sind diese definitiv inkompatibel. Ist hingegen die dritte Stelle unterschiedlich, kann keine konkrete Aussage über die Kompatibilität der Assemblies gemacht werden. Ein Beispiel ist, wenn eine neue Eigenschaft hinzugefügt wurde und die Assembly im Prinzip kompatibel ist, man aber keine Abwärtskompatibilität gewährleisten kann oder will. Die letzte Stelle schließlich deutet auf einen *QFE* (*Quick Fix Engineering*) hin und zeigt an, dass eine kleine und fast unbedeutende technische Korrektur vorgenommen wurde.

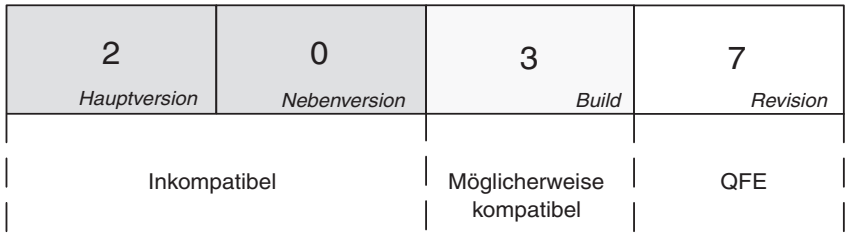


Abbildung 2.17: Aufbau der Versionsnummern von Assemblies

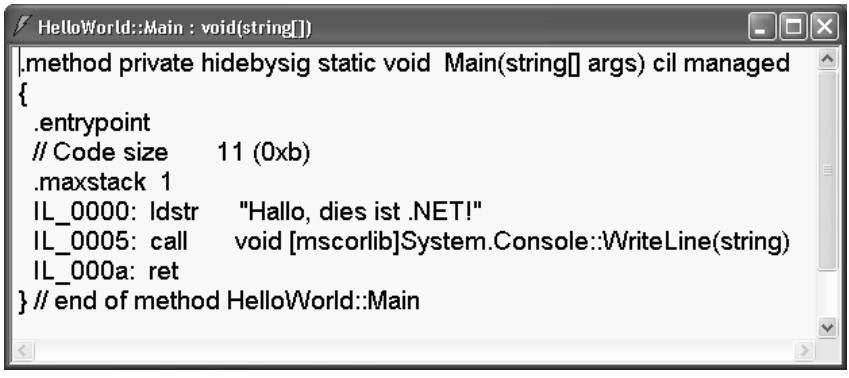
Somit ist eine Assembly mit der Version 2.1.3.7 inkompatibel mit unserer Beispiel-Assembly. Eine Assembly mit der Version 2.0.4.7 könnte inkompatibel sein, muss es aber nicht. Und eine Version 2.0.3.8 zeigt uns, dass die beiden Assemblies kompatibel sind, aber in der neueren Version eine kleine Änderung vorgenommen wurde.

2.5.2 Intermediate Language (IL)

Ich habe nun schon an mehreren Stellen die Microsoft Intermediate Language (IL oder MSIL) erwähnt. Die IL ist eine Zwischenrepräsentation von ausführbarem Code, der aber immer durch einen JIT Compiler in den Assembler-Code der Zielarchitektur übersetzt wird.

Wenn wir uns die weiter oben erstellte Assembly *HelloWorld.exe* noch einmal in ILDASM ansehen, werden wir auch endlich den ersten IL-Code zu sehen bekommen. Nach dem Start von ILDASM und dem Laden der Assembly präsentiert sich uns das Bild aus Abbildung 2.13. Doppelklicken wir jetzt aber auf den Eintrag MAIN, dann zeigt sich uns die IL-Darstellung der `Main`-Methode unseres *Hello World*-Programms (siehe Abbildung 2.18). Ich werde an dieser Stelle keine Einführung in IL geben, will aber darauf hinweisen, dass die Sprache IL einen recht umfangreichen Befehlssatz hat und wesentlich einfacher zu lesen ist als beispielsweise x86 Assembler.

Abbildung 2.18: IL-Code des Hello World-Programms



Man kann sogar – ähnlich wie bei herkömmlichen Maschinensprachen – rein in IL programmieren. Wenn man den kompletten IL-Code aus ILDASM extrahiert und in den Assembler von .NET steckt, dann wird daraus eine neue Assembly mit dem gleichen Verhalten erzeugt. Der Assembler heißt *ilasm.exe* und ist ebenfalls eine Kommandozeilenanwendung. Man kann sogar so weit gehen und eine Klasse in IL programmieren, von der dann eine andere in C# geschriebene Klasse erbt. Dies gilt alles aufgrund des allgemein gültigen Typsystems.

Damit die Arbeit mit ILDASM einfacher wird, habe ich in Tabelle 2.2 die wichtigsten Symbole mitsamt ihrer Bedeutung innerhalb von ILDASM als Referenz aufgelistet.

Tab. 2.2: Symbole in ILDASM und deren Bedeutung

Symbol	Beschreibung
	Mehr Informationen
	Namensraum
	Klasse
	Werttyp (Value Type)
	Schnittstelle (Interface)
	Methode
	Statische Methode
	Feld
	Statisches Feld
	Ereignis (Event)
	Eigenschaft

2.5.3 Metadaten in .NET

Es wurde schon an verschiedenen Stellen erwähnt: .NET arbeitet sehr viel und intensiv mit Metadaten – also mit Daten über die eigentlichen Daten. Zu diesen Metadaten zählen auf unterster Ebene die Metadaten innerhalb einer Assembly. Dort betrifft dies insbesondere das Manifest. Weiterhin wird in den Klassen und Methoden der Base Class Library sehr oft das Prinzip der Attribute verwendet. Attribute stehen dabei im Einklang mit der Verwendung von Metadaten. Die Attribut-basierte Programmierung hat in der Microsoft-Welt mit COM+ und den entsprechenden Erweiterungen von Visual C++ Einzug gehalten. Im Prinzip spart die Verwendung von Attributen viel unnötigen Programmieraufwand. Attribute werden meist dann eingesetzt, wenn eine Standardfunktionalität auf eine Klasse oder eine Methode übertragen werden soll. Im Falle von COM+ ist ein Beispiel für die Anwendung eines Attributs, wenn eine Klasse an einer Transaktion teilnehmen soll und diese Transaktion automatisch erfolgreich abgeschlossen wird, sofern keine Ausnahme auftritt. Somit sind Attribute Mechanismen, um einer Entität Metadaten zuzuweisen.

Ein typisches Beispiel für die Verwendung von Attributen findet man im Bereich Interoperabilität mit Win32-Funktionen aus .NET heraus. Durch einfaches Anbringen des folgenden Attributs an einer Methode, wird der ausführenden Einheit klar gemacht, dass hier Funktionalität zur Zusammenarbeit mit einer Win32-System-DLL zur Verfügung gestellt werden soll:

```
[DllImport("user32.dll")]
```

Attribute werden übrigens in C# von eckigen Klammern (*[]*) und in Visual Basic von spitzen Klammern (<>) umschlossen. Das System bietet eine Vielzahl von Attributen, die alle von einer Basisklasse abgeleitet sind: *System.Attribute*. Jede Klasse, die ein Attribut implementiert, muss nach Konvention das Suffix *Attribute* tragen, also z. B. *MeinAttributAttribute*.

So kann ein Entwickler seine eigenen Attribute erstellen. Ein schönes Anwendungsbeispiel ist die Erstellung eines Attributes, welches – auf eine Klasse angewendet – diese Klasse zur Laufzeit inspiziert und die gerade aktuellen Daten der Klasse in eine Datenbanktabelle schreibt. Diese Tabelle wird sogar noch automatisch erstellt, falls das Schema in der Datenbank noch nicht existiert. Wenn man einmal dieses Attribut angelegt hat, kann man es in beliebigen Projekten anwenden, um einen automatisierten Persistenzmechanismus verfügbar zu machen.

Die entsprechende Technologie zur Untersuchung von Assemblies und Typen zur Laufzeit heißt *Reflection* (Namensraum *System.Reflection*). Reflection hat auch ein Gegenstück namens *Reflection Emit* (Namensraum *System.Reflection.Emit*), mit dem man Typen und Assemblies dynamisch zur Laufzeit generieren kann. Mittels Reflection kann man Typen auch dynamisch zur Laufzeit laden und binden. Dies ist das typische Anwendungsszenario für den Einsatz von später Bindung unter .NET.

2.5.4 Garbage Collection

Wer kennt diese Probleme nicht? Angelegter Speicher wird nicht frei gegeben: Es treten Speicherlecks auf. Auf nicht angelegten oder bereits frei gegebenen Speicher wird versucht zuzugreifen: Es gibt eine Zugriffsverletzung. Die Programmierung und die Verwaltung des zur Verfügung stehenden Speichers kann mitunter zu einer eigenen Wissenschaft werden, vor allem bei umfangreicheren Projekten. Mit .NET erhält der Entwickler hier einen Helfer an die Hand, der sich um die automatische Speicherverwaltung kümmern soll.

Der so genannte *Garbage Collector* (*GC*) der Common Language Runtime verwaltet für eine Anwendung die Reservierung und Freigabe von Arbeitsspeicher. Das bedeutet, dass beim Entwickeln verwalteter Anwendungen kein Code für Aufgaben der Speicherverwaltung mehr geschrieben werden muss. Objekte werden innerhalb eines Prozesses immer sequenziell auf dem Heap und im Falle von .NETs CLR dem verwalteten Heap angelegt. Wenn der Heap nun zu voll wird, dann tritt der Garbage Collector in Aktion. Der zugrunde liegende Algorithmus ist in zwei Phasen aufgeteilt: Markieren und Einsammeln. Die erste Phase ist dafür zuständig, die Speicherbereiche zu finden, die nicht mehr benötigt werden. In der zweiten Phase sorgt er dann dafür, dass alle Objekte, die noch benutzt werden, auf dem Heap nach unten wandern, und so eine Komprimierung des Heaps stattfindet. Dies ist – sehr vereinfacht formuliert – der Algorithmus des Garbage Collectors in .NET, der nach einer heuristischen Methode arbeitet. Dabei ist das Verhalten des Garbage Collectors nicht-deterministisch, man kann also nicht vorhersagen, wann ein Objekt noch oder nicht mehr verfügbar ist, nachdem es dereferenziert wurde.

Das bedeutet also, dass das Setzen einer Objektreferenz auf *Null* diese nicht löscht oder aus dem Speicher entfernt. Das aus Visual Basic bekannte Szenario hat somit nicht mehr den ursprünglichen Effekt:

```
Set myObject = Nothing
```

Der interne Ablauf des Garbage Collectors ist noch wesentlich komplexer. Ein Programmierer hat auch begrenzten Einfluss auf dessen Steuerung – für unsere Zwecke soll diese kurze Einführung jedoch genügen.

2.6 Zusammenfassung

Es ist vieles neu in .NET. Aber hier hat keine Revolution sondern eher eine Evolution stattgefunden. Viele Konzepte und Technologien, die nun unter dem Mantel .NET zur Verfügung stehen, sind schon an anderer Stelle mehr oder weniger bemerkbar in Aktion getreten. Aber für einen Windows- und COM-Entwickler wird sich dennoch einiges ändern.

Angefangen bei der IL-Sprache für die plattformübergreifende Darstellung von Code bis hin zur automatischen Speicherverwaltung mit ihren Tücken: Es muss an verschiedenen Stellen umgedacht und umgeschult werden. Wenn diese Hürden aber erst einmal genommen sind, wird sich die Entwicklung mit und unter .NET sehr viel einfacher gestalten, als man es bisher gewohnt war.

3 Schöne neue WWWelt

ASP.NET und Web Forms

3.1 Überblick

Web-basierte Applikationen als *Thin Clients* sind seit einiger Zeit stark im Kommen und in Mode. Für die Entwicklung Web-basierter Anwendungen bedarf es allerdings nicht nur eines einfachen HTTP-Servers. Vielmehr werden Erweiterungstechnologien benötigt, mit denen man dynamische Inhalte auf Seiten des Servers erstellen kann. Dies steht im Gegensatz zum einfachen Bereitstellen statischer HTML-Seiten. In diesem Kapitel wird mit *ASP.NET* die entsprechende Technologie auf Basis der .NET-Plattform vorgestellt. Angefangen mit einem Vergleich zu bisherigen ASP-Versionen, über die genaue Beschreibung der Architektur hin zur Programmierung von Web-basierten Formularen mitsamt praktischer Tipps, soll auf den folgenden Seiten genügend Wissen zum Erstellen eigener Web-Anwendungen auf Basis von ASP.NET vermittelt werden. Dabei werden die Beispiele sowohl mit einem einfachen Texteditor wie Notepad als auch gegen Ende des Kapitels mit dem Visual Studio .NET erstellt.

3.2 Web-basierte Entwicklung mit ASP

Die Programmierung von Anwendungen, die auf einem WWW-Server ausgeführt werden und Inhalt sowie Informationen in Form von HTML zur Darstellung im WWW-Browser erzeugen, ist nicht gerade neu. Schlagworte wie *Common Gateway Interface* (CGI), *Perl* und *PHP* sind den meisten Entwicklern geläufig, ohne dass sie unbedingt größere Erfahrungen damit haben müssen. In der Windows-Umgebung haben sich über die Jahre hinweg die *Active Server Pages* (*ASP*) etabliert, deren neueste Entwicklungsstufe sich im .NET Framework wiederfindet. Doch was ist ASP und warum ist es jetzt nicht mehr zeitgemäß?

3.2.1 ASP als Revolution

Mit dem Erscheinen von ASP 1.0 unter Windows NT 4.0 im Herbst 1996 änderte sich die Herangehensweise an und Umsetzung von dynamischen Web-Applikationen schlagartig. Mit den bis dahin vertrauten und oben angesprochenen Techniken konnte man mehr oder weniger komfortabel

kleinere Anwendungen entwerfen. Durch die Integration von ASP in den *Microsoft Internet Information Server* (*IIS*) wurde ein spürbarer Zugewinn an Leistungsfähigkeit erzielt. Zudem trat es mit einem einheitlichen Objektmodell zur Programmierung auf die Bühne. Mit ASP war es möglich, bereits vorgefertigte COM-Komponenten zu referenzieren und deren Anwendungslogik in die eigene Web-Anwendung einzubauen. Somit konnte man einerseits viel Zeit bei der Implementierung sparen und andererseits auch die Geschwindigkeit und Skalierbarkeit verbessern, da anstatt des üblichen Skript-Codes hier binäre Einheiten zum Einsatz kommen können.

Doch wie sieht ASP aus? Das folgende Listing zeigt eine sehr einfache ASP-Seite, welche verschiedene Server-Variablen ausliest und dynamisch in die HTTP-Antwort zurück schreibt.

```
<%@ Language = "VBScript" %>
<HTML>
<BODY>
<%
Dim strServername, strLocalname
strServername = Request.ServerVariables("SERVER_NAME")
strRemoteIP = Request.ServerVariables("REMOTE_ADDR")
Response.Write "Sie sprechen den Server " & _
   strServername & " vom Rechner " & _
   strRemoteIP & " aus an."
%>
</BODY>
</HTML>
```

Listing 3.1: Einfache ASP-Seite zur Ausgabe von Server-Variablen

Eine ASP-Seite ist als Mixtur aus Skript-Code und HTML-Code zu sehen, die weitestgehend miteinander vermischt werden. Die Ausgabe im obigen Beispiel wird über das Objekt *Response* geregelt. *Response* und *Request* sind zwei von vielen Objekten die dem ASP-Entwickler von der Laufzeitumgebung an die Hand gegeben werden. Was ist die Laufzeitumgebung und wie funktioniert die Abarbeitung einer Seite auf dem IIS? Antworten auf diese Fragestellungen sind wichtig, um gegebenenfalls Probleme bei eigenen Anwendungen verstehen oder beheben zu können. Deshalb werden wir jetzt die Architektur von ASP unter die Lupe nehmen. Außerdem werden wir anhand dieser Architektur auch deren Schwachstellen und die verbesserungswürdigen Details ausfindig machen können.

Die Architektur und den internen Aufbau der klassischen ASP-Umgebung kann man in Abbildung 3.1 ersehen. Es sei an dieser Stelle ausdrücklich darauf hingewiesen, dass man den Begriff Active Server Pages wörtlich nehmen muss: Sämtliche aktiven Aktionen in diesem Modell finden auf dem Web-Server statt. Dies bedeutet, dass die Erstellung der HTML-Seite auf Seiten des IIS erfolgt – und prinzipiell keine aktiven Client-Technologien wie *ActiveX*-Kontrollelemente oder Java-Applets involviert sind.

Gestartet wird die Server-seitige Abarbeitung durch eine gewöhnliche HTTP-Anfrage. Diese Anfrage kann durch einen Browser oder aber durch eine Windows-Anwendung mit HTTP-Unterstützung angestoßen werden. In der Abbildung wird exemplarisch die Seite *default.asp* angefordert. An der Dateiendung *.asp* erkennt die im IIS registrierte ASP-Laufzeitumgebung, dass es sich um keine übliche statische HTML-Seite, sondern um eine ASP-Seite handelt. Die Laufzeitumgebung ist in Form einer *ISAPI* (*Internet Server Application Programming Interface*)-Erweiterungs-DLL implementiert und wird über die Konfigurationsschnittstelle des IIS im System bekannt gemacht (vgl. Abbildung 3.2). Die Laufzeitumgebung lädt die angeforderte Seite vom Festspeicher, indem sie die im IIS erstellte Verknüpfung vom virtuellen Pfad zum physikalischen Pfad auflöst. Einmal in den Speicher der *ASP.DLL* geladen, kann diese Seite nun durch einen Parser verarbeitet werden. Die Skriptblöcke werden extrahiert und den entsprechenden zuständigen Komponenten zugeführt. Diese Komponenten heißen *ActiveX-Scripting-Engines* und sind für das Parsen eines Skriptes in einer speziellen Programmiersprache zuständig. Standardmäßig sind ASP-Seiten in *VBScript* verfasst, eine Untermenge von Visual Basic speziell für die Web-Programmierung und Windows-Administration entworfen. Somit wird also diese Engine in den Speicher geladen (alle Engines sind COM-Komponenten in Form von DLLs). Gelangt die Engine beim Abarbeiten des Skriptes beispielsweise an eine Stelle, die nach der Erzeugung eines COM-Objektes verlangt, dann wird von der Laufzeitumgebung versucht, dieses Objekt zu erzeugen. In der Abbildung ist dies das *ADO Recordset*-Objekt für das Arbeiten mit Daten aus einer Datenbank. Das *Recordset*-Objekt interagiert über seine Schnittstellen gekapselt mit dem eigentlichen Datenbehälter (z.B. SQL Server) und liefert die Abfrageergebnisse für das Skript. Nach Abarbeitung aller Skriptteile wird die daraus resultierende HTML-Seite zusammengesetzt und über die Standard HTTP-Antwort an den aufrufenden Client zurückgegeben.

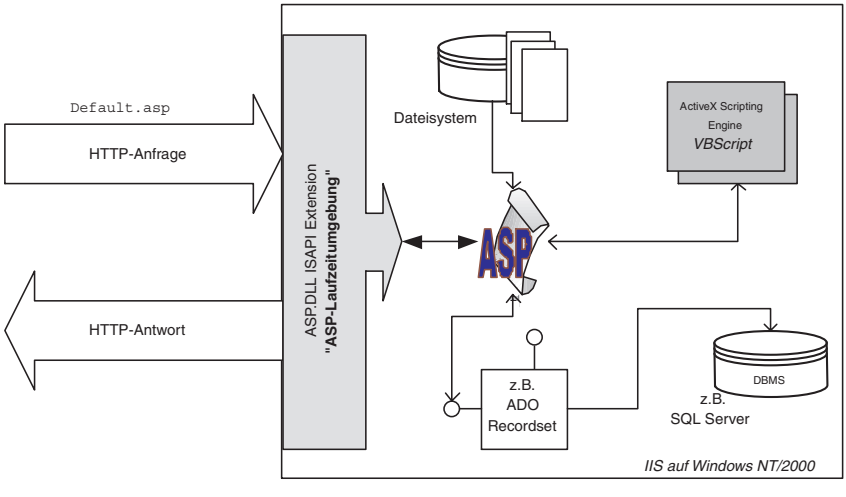


Abbildung 3.1: Verarbeitungsablauf in der klassischen ASP-Architektur

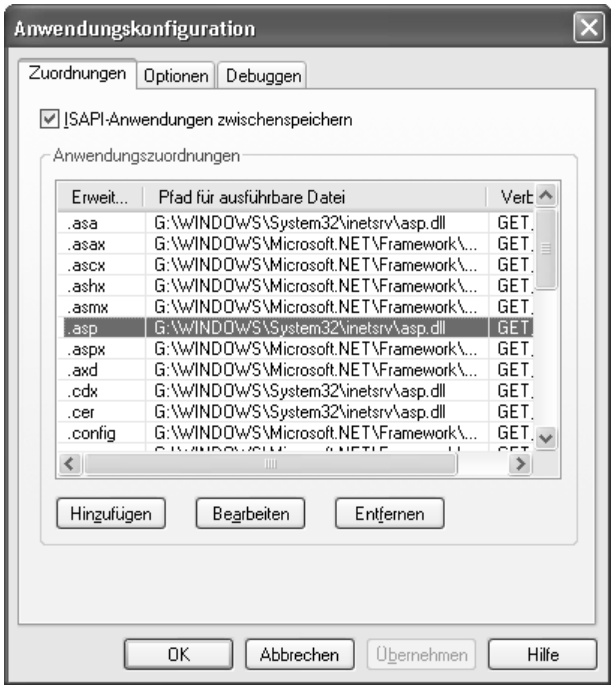


Abbildung 3.2: Registrierung der ASP-Laufzeitumgebung

Es sei erwähnt, dass mit ASP nicht nur einfache HTML-Seiten dynamisch erzeugt werden können. Auf der Server-Seite kann man auch bestimmen, dass in den HTML-Code Verweise auf ActiveX-Elemente oder Java-Applets eingebettet werden sollen. Durch den Einsatz von Drittkomponenten etwa kann man ebenfalls PDF-Dokumente oder GIF-, JPG- und TIFF-Bilder generieren. Der Phantasie und der technischen Umsetzung sind auf Seiten des IIS mit COM-Komponenten keine Grenze gesetzt

ASP ist also eine für den Benutzer einer Web-Site transparente Technologie, welche aber dem Entwickler einige Hürden aus dem Weg räumt. Weiterhin gibt ihm ASP ein Programmiermodell an die Hand, mit dem er auf schnelle Art und Weise leistungsfähige Web-basierte Anwendungen erstellen kann. Doch gibt es einige Punkte in ASP, bei denen viele Programmierer und Administratoren die Nase rümpfen. Denn der Weisheit letzter Schluss ist ASP auch in der Version 3.0 noch nicht.

3.2.2 Probleme mit ASP

Wie schon oben erwähnt ist ASP zwar ein Schritt in die richtige Richtung, aber über die Jahre haben sich im praktischen Einsatz doch sehr viele Unzulänglichkeiten heraus kristallisiert. Und schließlich ist eine Technologie immer verbesserungswürdig, vor allem wenn sie ihre Tauglichkeit im tägli-

chen Einsatz unter realen Bedingungen beweisen muss. Im Folgenden werden die wichtigsten und offensichtlichsten Schwachstellen der klassischen ASP-Architektur dargestellt.

- *Spaghetti-Code*: durch die Verschmelzung von Skript- und HTML-Quelltext in eine Datei kann man sehr schnell den Überblick verlieren. HTML-Tags inmitten eines Wirrwarrs von Skript-Code und dynamisches Generieren von HTML aus einem Stück Skript: All diese immer wieder gesehenen Unarten der ASP-Programmierung führen zu einer Masse von ASP-Seiten innerhalb eines Projektes, die ab einem bestimmten Umfang nicht mehr zu verwalten sind. Auch die Auslagerung von immer wieder benötigten Prozeduren in *include*-Dateien verbessert dieses Bild nicht: Mit einer Reihe von *include*-Dateien wird der Quelltext zwar zunächst übersichtlicher, doch bekommt man mit dieser Vorgehensweise große Probleme beim Zurückverfolgen der eigentlichen Fehler- oder Ausgabeursache. Nicht zuletzt ist die Verschachtelung von *include*-Dateien ohne die Erzeugung von zyklischen Referenzen eine Wissenschaft für sich.
- *COM-Komponenten*: auch wenn die Nutzung von COM innerhalb von ASP zunächst als großer Fortschritt anmutet, hat man doch sehr schnell gemerkt, dass die Verwendung dieser Komponenten meistens mehr Probleme aufwirft als sie löst. Dies liegt an einigen COM-Eigenheiten. Hierzu zählt die notwendige Registrierung auf dem Server in der Registrierungsdatenbank von Windows und die schlechte Versionsverwaltung. Beim Austausch der COM DLL gegen eine neue Version kann unter Umständen eine ASP-Seite, welche diese COM-Komponente nutzt, ihren Dienst versagen. Dies geschieht dann oftmals ohne Vorwarnung.
- *Cookie-basierte Sitzungen*: Wer mit ASP programmiert wird das Problem kennen. Um in einer ASP-Anwendung benutzerspezifische Daten zwischen mehreren Aufrufen auf dem Server halten zu können, bedient man sich der von ASP angebotenen *Session* (Sitzung). Allerdings ist die Sitzungsimplementierung in ASP so realisiert, dass der IIS immer ein *Cookie* auf Seiten des Browsers setzen muss. Ohne Cookie gibt es keine Sitzung. In Anbetracht der Tatsache, dass viele Unternehmen schlicht und einfach die Verwendung von Cookies verbieten, ist dieser Umstand ein großes Minus. Es gibt zwar andere Lösungen, die dies ermöglichen, sie sind aber nicht in ASP integriert.
- *Interpretierung*: Sämtliche Seiten in ASP werden durch die Scripting-Engines interpretiert. Es findet also kein Kompilierungsvorgang vor dem Gebrauch der Programme statt, wie das bei anderen Programmiersprachen und Entwicklungsumgebungen durchaus der Fall ist. Es werden zwar so genannte ASP-Schablonen (*Templates*) für ASP-Dateien erzeugt und im Hauptspeicher zwischengelagert, dieser Speicher ist aber sehr begrenzt und wird bei Abarbeitung einer Vielzahl von ASP-Seiten mit neuen Templates gefüllt. Aus Sicht der Leistungsfähigkeit ist dies natürlich nicht die optimale Lösung. Eine Seite immer und immer wieder zu interpretieren kostet zu viele Ressourcen auf dem Server.

- *Skriptsprachen*: Dieser Punkt geht eng einher mit dem vorhergehenden. Zur Programmierung werden in ASP nur bestimmte Skriptsprachen zugelassen, für die auch eine ActiveX-Scripting-Engine existiert. Dies sind standardmäßig VBScript und *JScript* (ein Microsoft-Ableger von JavaScript). Zudem bieten Skriptsprachen aufgrund ihrer Arbeitsweise keine strenge Typisierung, was im Umgang mit ASP oft als Nachteil erkannt wird. Ein Integer verhält sich wie eine Zeichenkette oder wie ein Objekt. Es ist alles ein Typ. Diese Vereinfachung trägt auch weiter zu schwer les- und verstehbaren Quelltexten bei. Die fehlende Typisierung ist oftmals die größte Quelle für Fehler.
- *Wiederverwendung*: Mit ASP ist es nicht möglich, bestimmte grafische Elemente Server-seitig zu kapseln und einfach in anderen ASP-Seiten zu integrieren. Ein Beispiel ist die Darstellung eines Login-Dialogs. Im Großen und Ganzen sehen diese Dialoge immer gleich aus. Es wäre daher wünschenswert, wenn man die grafische Darstellung (und am besten auch gleich die damit verbundene Logik) so kapseln könnte, dass man diese Komponente in andere Projekte einbauen und durch Setzen von Eigenschaften deren Ausprägung beeinflussen könnte. Hierdurch würde sich auch die Erstellung von Seiten für mehrere Browser vereinfachen, da diese Kontrollelemente selbst dafür sorgen könnten, welches HTML nun gerade erzeugt werden soll.
- *Behandlung von Formularen*: Die Verarbeitung von HTML-Formularinhalten kann sich in ASP als schwieriges Unterfangen herausstellen. Ein Problem ist das Verschicken von Formularen an die gleiche Seite. Hier muss man extra Aufwand betreiben, um heraus zu finden, was nun genau angezeigt werden soll. Eine andere Schwachstelle ist das Merken von Formularinhalten zwischen HTTP-Anfragen des Clients. Auch hier gibt es keine gute Lösung im klassischen ASP.

Für fast alle diese Probleme gibt es in ASP.NET und dem .NET Framework eine Lösung. Und wenn noch keine Lösung existiert, dann kann sich ein Programmierer diese Lösung selbst schaffen. Es lohnt sich also, auf ASP.NET umzusteigen. Aber keine Bange: ASP wird die nächsten Jahre so bestehen bleiben, wie man es kennen und schätzen gelernt hat. Denn vor allem für die Migrationsphase hin zu .NET kann man beide Versionen, das klassische ASP und ASP.NET, gemeinsam auf einem Rechner betreiben. Wollen wir uns nun also der neuen Welt von ASP.NET zuwenden. Ähnlich wie im vorangegangenen Teil über ASP werden wir zunächst mit der Analyse der zugrunde liegenden Architektur beginnen.

3.3 ASP.NET

Es ist müßig herausfinden zu wollen, was nun eigentlich zuerst in den Köpfen der Microsoft-Entwickler vorhanden war: Die .NET-Plattform als Basis oder ASP.NET als neue Umgebung für die Entwicklung von Web-Anwendungen. Fakt ist, dass ASP.NET nicht einfach nur die neue Version von ASP ist, sondern von Grund auf komplett neu geschrieben wurde und nahtlos in

das .NET Framework integriert ist. Um diesem radikalen Schnitt mehr Ausdruck zu verleihen, ist man in Redmond wohl auch von der ursprünglichen Namensgebung *ASP+* abgekommen und hat den Namen im Rahmen der gesamten .NET-Initiative angepasst. In diesem Abschnitt werden wir uns die Architektur und Funktionsweise von ASP.NET genauer ansehen, bevor wir zur eigentlichen Programmierung schreiten.

3.3.1 Überblick

Als neue Version von ASP auf Basis von .NET hat ASP.NET auf technischer Ebene wirklich nicht mehr viel mit dem klassischen ASP gemein. So ist es auch ein angenehmer Nebeneffekt, dass man beide Varianten bedenkenlos nebeneinander betreiben kann – sie beißen sich nicht auf einem System. Wie wir im vorherigen Kapitel gesehen haben, spielt sich fast alles in .NETs neuer verwalteter Welt ab. Die CLR ist allgegenwärtig und übernimmt so wichtige Aufgaben wie Codeverifizierung, Klassenladen und JIT-Kompilierung. Die ASP.NET-Umgebung stellt hier eine kleine Ausnahme dar, da sie auf den Windows Systemen eng mit dem IIS verbunden ist. Und der IIS ist in heute verfügbaren Windows-Versionen noch rein auf Basis von Win32 entwickelt worden. Wie funktioniert nun aber das Zusammenspiel?

3.3.2 Architektur und Laufzeitumgebung

Am einfachsten erklären lässt sich die Funktionsweise von ASP.NET, wenn man die Architektur zunächst aus der Vogelperspektive betrachtet (siehe Abbildung 3.3).

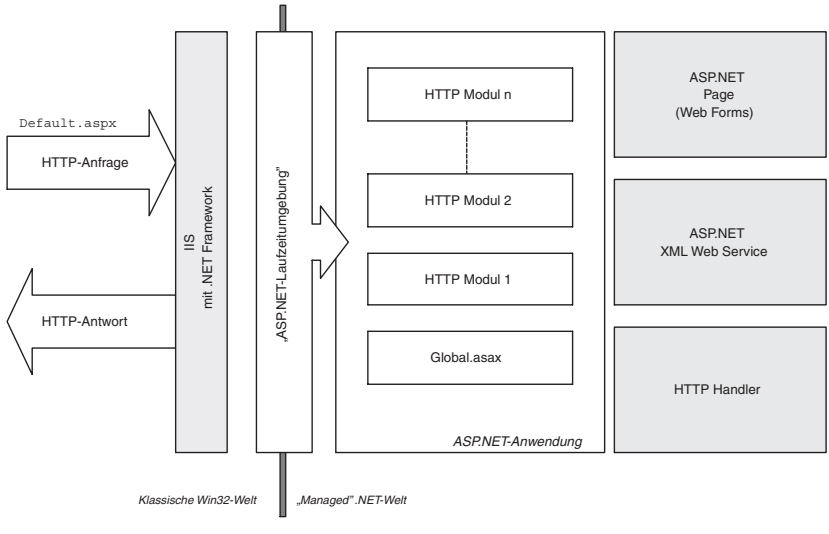


Abbildung 3.3: Verarbeitung einer HTTP-Anfrage in ASP.NET

Eine vom Web-Client eingehende HTTP-Anfrage wird üblicherweise vom IIS entgegengenommen. Ähnlich wie beim klassischen ASP sorgt eine ISAPI-Erweiterung und ein ISAPI-Filter für die korrekte Abarbeitung der weiteren Schritte. Diese Erweiterung stellt zusammen mit einem externen Prozess (mehr darüber weiter unten in diesem Abschnitt) die so genannte ASP.NET-Laufzeitumgebung dar. Die Laufzeitumgebung ist technisch gesehen ein Zwitter: die ISAPI DLL ist eine reine Win32-Implementierung und der externe Prozess ist eine Win32-Anwendung, welche aber die .NET CLR beherbergt. Über die CLR werden dann die eigentlichen ASP.NET-Aktionen ausgeführt. Also findet über diesen Prozess der Übergang von der klassischen Win32- in die verwaltete .NET-Welt statt. Diesen Fakt sollte man sich immer vor Augen halten, wenn es um die Entwicklung und das Debuggen von ASP.NET-Anwendungen geht.

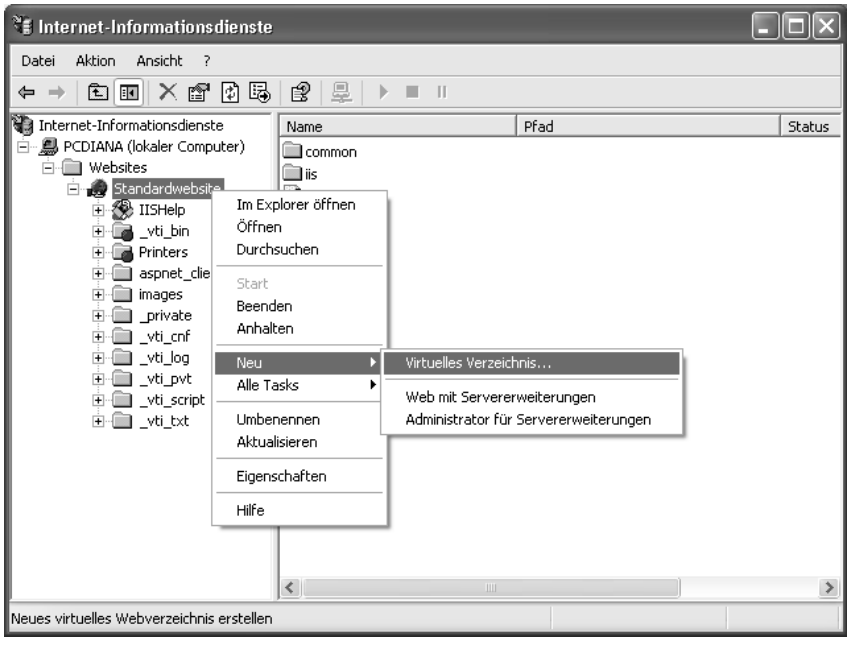


Abbildung 3.4: Erstellen eines virtuellen Verzeichnisses in der Internet-Informationsdienste-Anwendung

Innerhalb der Laufzeitumgebung werden nach Bedarf die einzelnen ASP.NET-Anwendungen geladen, gestartet und abgearbeitet. Eine Anwendung definiert sich im Umfeld von ASP.NET und IIS folgendermaßen: für jede ASP.NET-Anwendung muss ein virtuelles Verzeichnis innerhalb des IIS existieren, welches als IIS-Anwendung konfiguriert ist. Die Einrichtung kann zum einen über den Windows-Explorer (siehe Abbildung 3.5) zum anderen über die Verwaltungskonsole des IIS, die in der Anwendung INTERNET-INFORMATIONSDIENSTE implementiert ist (siehe Abbildung 3.4) erfolgen. Im ersten Fall wird die Erstellung der IIS-Anwendung automatisch vorgenommen, während man in der Verwaltungskonsole die erforderlichen Parameter selbst einstellen muss. In der klassischen ASP-Umgebung kann man

die Schutzeinstellungen einer Anwendung im Dialog für diese IIS-Anwendung konfigurieren. Es gibt die Werte *Niedrig*, *Mittel* und *Hoch* für den Punkt ANWENDUNGSSCHUTZ. *Niedrig* bedeutet, dass eine Anwendung innerhalb des IIS-Prozesses abläuft und somit keinerlei Schutz für den gesamten Prozess besteht: Wenn diese Anwendung fehlschlägt, wird der gesamte IIS lahm gelegt. Bei der Einstellung *Mittel* wird die Anwendung zusammen mit gleich konfigurierten Anwendungen in einem externen Prozess (eine COM+-Anwendung) ausgelagert. Mit dem Wert *Hoch* schließlich wird die IIS-Anwendung immer in einem eigenen Prozess gestartet, welches natürlich den größtmöglichen Schutz für die Anwendung und den eigentlichen Web-Server bedeutet. Die beiden letzten Konfigurationsmöglichkeiten werden über den *WAM* (*Web Application Manager*) abgewickelt, der hierfür mit der COM+-Laufzeitumgebung für das Anlegen von entsprechenden COM+-Anwendungen interagiert.

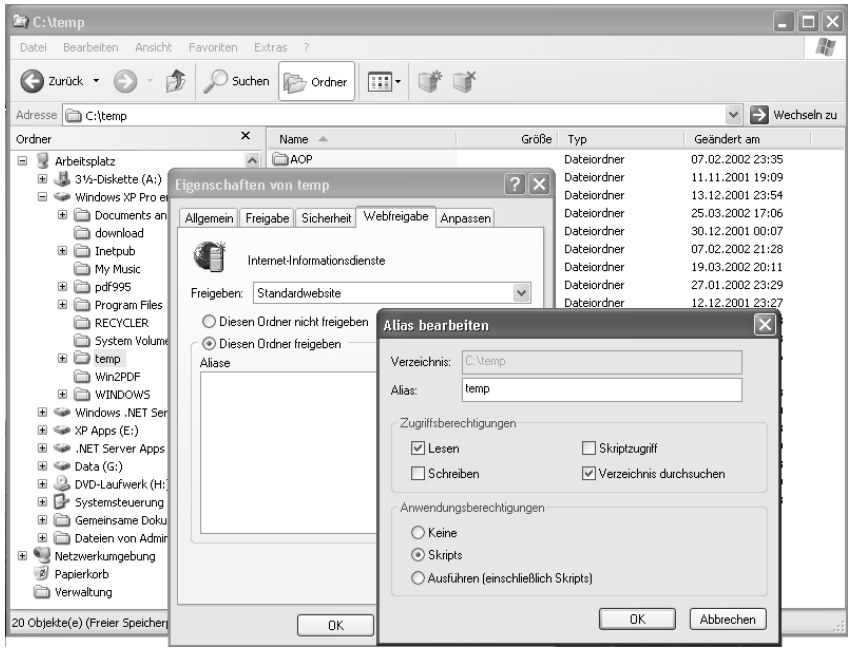
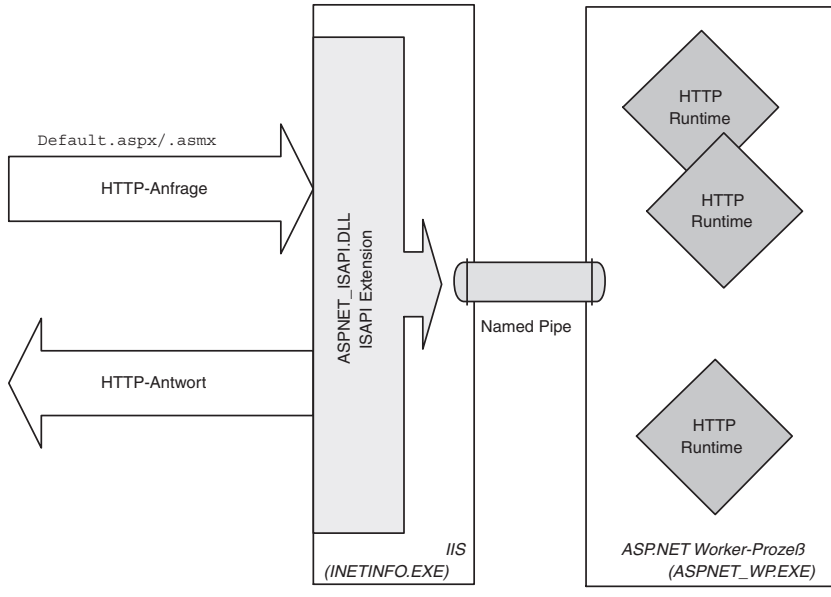


Abbildung 3.5: Erstellung eines virtuellen Verzeichnisses im Windows Explorer

Doch in ASP.NET sieht dies ganz anders aus. Bei der Installation von ASP.NET wird in der Konfigurationsdatenbank des IIS, die *Metabase*, die DLL *aspnet_isapi.dll* zur Eigenschaft *InProcessIsapiApps* hinzugefügt. Somit sind ASP.NET-Anwendungen automatisch als In-Prozess konfiguriert. Der Schutz erfolgt in ASP.NET über das Auslagern der eigentlichen Funktionalität in den externen Worker-Prozess *aspnet_wp.exe* (vgl. Abbildung 3.6). Zusätzlich bietet dieser Prozess eine weitere Sicherungsstufe. Jede einzelne ASP.NET Anwendung wird in einer eigenen *Application Domain* (*AppDomain*) innerhalb des Worker-Prozesses gestartet. Somit sind die einzelnen

Anwendungen innerhalb von .NET voneinander getrennt – wenn nun eine Anwendung abstürzt, dann wird nur deren AppDomain entfernt, alle anderen können unbehelligt weiter laufen. Und falls wirklich einmal der komplette Prozess hängen sollte, dann kann man diesen aus dem Speicher entfernen, ohne den IIS oder gar das Betriebssystem neu starten zu müssen.

Abbildung 3.6: ASP.NET-Architekturüberblick



Das Konzept der Application Domains wird in Kapitel 8 sehr genauer beschreiben. Stellen Sie sich für den Moment AppDomains als Mini-Prozesse vor, die einen gegenseitigen Schutz von .NET-Anwendungen innerhalb eines physikalischen Betriebssystem-Prozesses erlauben.

Die ASP.NET-Laufzeitumgebung (d.h. der Prozess *aspnet_wp.exe*) läuft nicht wie eine anonyme ASP-Applikation im IIS unter dem Windows-Account *IUSR_Rechnername*. Er läuft auch nicht, wie dies bei isolierten ASP-Anwendungen üblich ist, unter dem Account *IWAM_Rechnername*. Mit dem Erscheinen der Version 1.0 von ASP.NET wird der ASP.NET-Prozess mit der Identität des Kontos *ASPNET* ausgeführt. Dieses Konto hat nur sehr wenig Berechtigungen im System, vor allem im Dateisystem ist der Handlungsspielraum von ASPNET sehr stark eingeschränkt.

Dieser Umstand kann zu großen Problemen bei der Konfiguration für eine sichere Ausführung von ASP.NET-Applikationen führen. Details über diese Schwierigkeiten und wie man sie lösen kann werden in Kapitel 7 vorgestellt.

Wenn man ASP.NET übrigens auf einem Windows .NET Server installiert, ist das Verhalten wiederum anders: Der IIS 6.0 von Windows .NET hat ein komplett neues Prozessmodell und auch andere Account-Einstellungen.

Die Kommunikation der ISAPI DLL mit dem Worker-Prozess erfolgt über Windows *Named Pipes*, ein Mittel zur Inter-Prozess-Kommunikation unter Windows NT und höher. Als Parameter werden hauptsächlich Konfigurationseinstellungen übergeben, die sich vor allem auf Sicherheitsaspekte beziehen. Ist die Anfrage erst einmal im Worker-Prozess angekommen, dann wird für jeden einzelnen Prozess die ASP.NET Pipeline gestartet. Mehr Informationen hierüber finden Sie in Abschnitt 3.3.3.

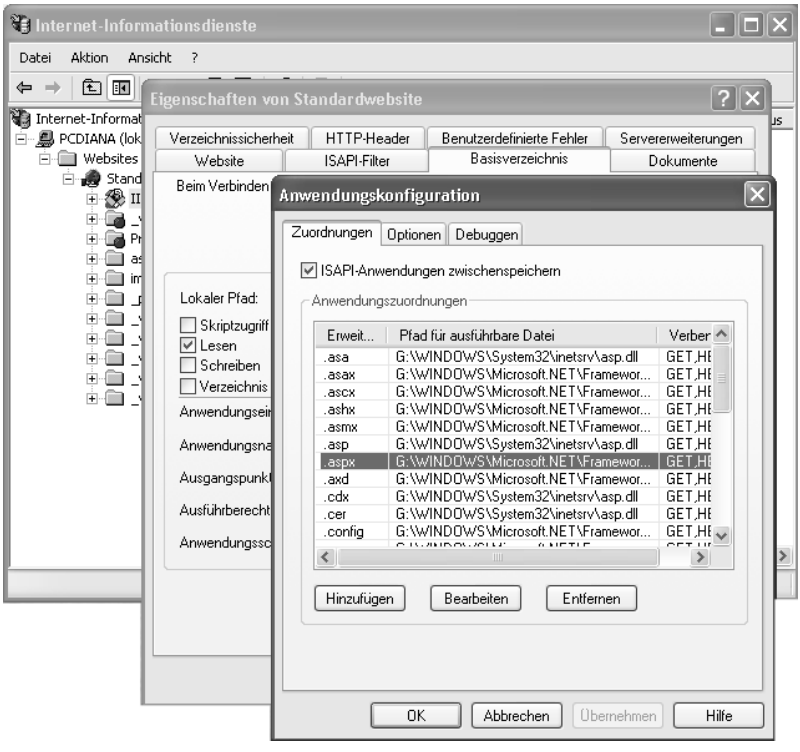


Abbildung 3.7: Dateiverknüpfungen unter ASP.NET

Damit das Zusammenspiel von IIS und ASP.NET-Laufzeitumgebung auch reibungslos funktioniert, müssen noch die entsprechenden Dateiendungen auf die ISAPI DLL verweisen. Auch diese Einstellungen kann man über die INTERNET-INFORMATIONSDIENSTE-Anwendung vornehmen, jedoch wird dies bereits bei der Installation von ASP.NET automatisch erledigt. Wie man in Abbildung 3.7 gut sehen kann, sind mehrere Endungen mit der *aspnet_isapi.dll* verknüpft. Tabelle 3.1 zeigt die wichtigsten Dateien im ASP.NET-Umfeld mit ihren Endungen auf.

Tab. 3.1: Dateiendungen in ASP.NET

Dateiendung	Beschreibung
aspx	ASP.NET-Seiten, mit Web Forms (siehe Abschnitt 3.4), äquivalent zu *.asp* im klassischen ASP.
asmx	ASP.NET Web Service-Datei (siehe Kapitel 5).
ashx	Generischer ASP.NET HTTP Handler.
ascx	Server-seitiges ASP.NET User Control.
config	ASP.NET-Konfigurationsdatei.
rem	Kommunikationsendpunkt für .NET Remoting (siehe Kapitel 8).
soap	Kommunikationsendpunkt für .NET Remoting (siehe Kapitel 8).

Die oben nicht aufgeführten Endungen, die man aber in der Verwaltungskonsole des IIS noch sehen kann, sind vornehmlich Endungen für Dateien mit Quelltext. Dateien für C#, Visual Basic .NET etc., die evtl. in einem virtuellen Verzeichnis stehen, sollen nicht einfach herunter geladen werden können. Über diese Zuordnung im IIS zusammen mit der Konfiguration der ASP.NET-Laufzeitumgebung kann dies erfolgreich verhindert werden.

Mit diesem technischen Überblick über die einzelnen Bestandteile und die grundlegende Funktionsweise von ASP.NET sind nun die Voraussetzungen geschaffen, etwas tiefer in die Abarbeitung einer HTTP-Anfrage in ASP.NET einzutauchen. Im Folgenden werden wir uns die HTTP Pipeline und den Kompilierungsvorgang näher ansehen, bevor wir dann endgültig die ersten Zeilen Code schreiben werden.

3.3.3 ASP.NET Pipeline

Mit der Einführung von ASP.NET werden die vor allem von C-Programmierern geschätzten ISAPI-Erweiterungen und -Filter langsam verdrängt. Dies bedeutet nicht, dass man ab heute keine ISAPI DLLs mehr programmieren kann oder darf, sondern dass mit ASP.NET ein neues, schlüssigeres Modell zur Abarbeitung einer kompletten Anfrage und der zugehörigen Antwort eingeführt wird: Die so genannte *ASP.NET oder HTTP Pipeline*. Bereits beim Überblick in Abbildung 3.3 kann man den internen Ablauf innerhalb einer ASP.NET-Anwendung grob erkennen. Bevor die eigentliche Seite abgearbeitet wird, wird unter Umständen noch eine Vielzahl von weiteren Schritten ausgeführt. Die Pipeline besteht prinzipiell aus einer Instanz der *HttpRuntime*-Klasse, einer beliebigen Anzahl von HTTP-Modulen und genau einem HTTP Handler Die *HttpRuntime* ist das Herzstück einer ASP.NET-Anwendung. Ein Modul kann man hierbei mit einem ISAPI-Filter vergleichen, wobei ein Handler eher einer ISAPI-Erweiterung gleicht. Ersteres inspiziert die eingehende Anfrage, kann sie bei Bedarf modifizieren oder auf Basis des Inhalts bestimmte Aktionen starten (z. B. eine maßgeschneiderte Authentifizierung gegen eine Benutzerdatenbank) und gibt sie dann an das nächste Modul in

der Pipeline weiter – sofern eines vorhanden ist. Am Ende der Kette steht schließlich ein Handler (siehe Abbildung 3.8). Dieser Handler muss die Schnittstelle *IHttpModule* implementieren und sorgt für die endgültige Verarbeitung der Anfrage. So gibt es beispielsweise einen Handler für *.aspx*-Seiten und einen Handler für *.asmx*-Web Service-Dateien (vgl. Kapitel 5). Jeder Entwickler kann sich darüber hinaus auch seine eigenen Handler erstellen, um neben der vom System zur Verfügung gestellten Funktionalität auch speziell zugeschnittenes Verhalten implementieren zu können.

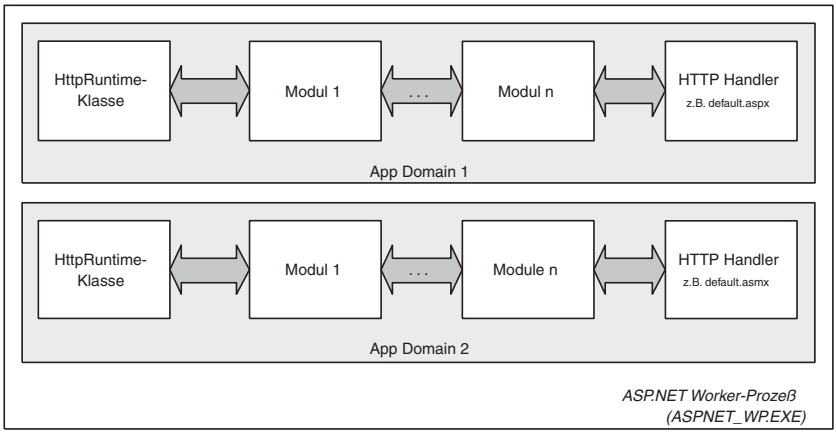


Abbildung 3.8: ASP.NET Pipeline im Worker-Prozess

Aufgrund des modularen Aufbaus von ASP.NET mit der HTTP Pipeline und dem Worker-Prozess kann man ASP.NET leicht auch in andere Web-Server integrieren. So könnte man für den Apache-Server ein Apache-Modul schreiben, welches die HTTP Pipeline/Runtime anspricht oder aber einfach den Worker-Prozess über CGI in einem Server-seitigen Skript ansprechen. Der Phantasie sind hier keine Grenzen gesetzt.

Zu erwähnen ist noch, dass jede ASP.NET-Anwendung ihre eigene Pipeline besitzt und diese innerhalb einer eigenen AppDomain im Worker-Prozess abläuft (siehe Abbildung 3.8). Jede ASP.NET-Anwendung verfügt über ein AppDomain-globales Objekt des Typs *HttpApplication*. Über dieses Objekt hat der Entwickler Zugriff auf immer wieder benötigte Eigenschaften wie *Request*, *Response* oder aber auf die HTTP-Header. Das Objekt instanziiert weiterhin die einzelnen Module, um die Abarbeitung der HTTP-Anfrage durchzuführen. Ein Programmierer kann *HttpApplication* erweitern, indem er in der Datei *global.asax* Einträge vornimmt. Diese Datei ist vergleichbar mit *global.asa* von ASP. Sämtlicher Code in *global.asax* wird automatisch in eine Unterklasse von *HttpApplication* kompiliert. Somit kann man das Verhalten der ASP.NET-Anwendung einfach durch Programmierung einer skriptähnlichen Datei manipulieren, ohne eigene Handler oder Module schreiben zu müssen. Typischerweise werden in der *global.asax* Ereignisse bearbeitet, die während des Lebenszyklus einer Anwendung auftreten kön-

nen. Ein Beispiel für eine *global.asax*-Datei steht in Listing 3.2. Hier wird beim Start der Anwendung ein spezielles HTML-Tag ausgegeben, um zu demonstrieren, dass gerade dieses Ereignis ausgelöst wurde. Zusätzlich wird auch noch das Ereignis `Application_BeginRequest` behandelt, um den Unterschied zwischen dem Starten einer Anwendung und dem Starten einer Anfrage innerhalb einer Anwendung zu verdeutlichen.

```
<%@ Import Namespace="System.Web" %>

<script language="C#" runat="server">
   void Application_OnStart(Object sender, EventArgs E)
   {
      HttpContext.Current.Response.Write
         ("<B>Anwendung startet ... </B><BR>");
   }

   void Application_BeginRequest
      (Object sender, EventArgs E)
   {
      Response.Write("<B>Global.asax: ");
      Response.Write("Anfrage startet ...<B><BR>");
   }
</script>
```

Listing 3.2: Exemplarische global.asax

Nach dieser eingehenden Betrachtung des Aufbaus von ASP.NET wollen wir im folgenden Abschnitt aufklären, wie die von uns geschriebenen Seiten kompiliert und dadurch schneller als von ASP gewohnt ausgeführt werden.

3.3.4 Kompilierte Ausführung

ASP war schon im Vergleich zu seinen Kontrahenten wie CGI schnell. Allerdings wurde der gesamte Code in einer ASP-Seite immer und immer wieder interpretiert. Abgesehen von binären COM-Komponenten in Form von DLLs wurde bei jeder Anfrage der Skript-Code aufs Neue mit einem Parser bearbeitet – eigentlich eine große Verschwendung hinsichtlich maximaler Leistungsfähigkeit. Deshalb geht ASP.NET hier einen anderen Weg.

Mit dem Auftreten von ASP.NET wird es keine echten Skriptsprachen mehr geben. Das bedeutet für viele Entwickler, dass sie Abschied von VBScript nehmen müssen. Lediglich eine neue Version von JScript überlebt in der .NET-Welt. Grund hierfür ist, dass alle Programmiersprachen nun auf Basis der CLR laufen und in MSIL-Code übersetzt werden müssen. Auf der anderen Seite bedeutet diese Entwicklung aber auch, dass man fortan *.aspx*-Seiten mit Visual Basic, C# oder C++ entwickeln kann. Eine *.aspx*-Seite bzw. -Anwendung wird also immer kompiliert ausgeführt, Quelltext wird nicht mehr bei jedem Aufruf analysiert. Die Funktionsweise soll in den folgenden Absätzen erläutert werden.

Da wir hier noch keine *.aspx*-Seite im Quelltext gesehen oder programmiert haben, machen wir einen kleinen Vorgriff. Das unten stehende Listing repräsentiert eine der einfachsten *.aspx*-Seiten, die man sich vorstellen kann.

```
<%@ Page Language="C#" %>
<%
Response.Write("Sehr einfache .aspx Seite.");
%>
```

Diese Seite ähnelt auf den ersten Blick sehr stark einer herkömmlichen ASP-Seite. Allerdings ist sie in C# geschrieben. In dieser Form kann die *.aspx*-Seite aber nicht ausgeführt werden, da in .NET schlussendlich jedes Stück ausführbarer Code in MSIL vorliegen muss. Zum Glück gibt es aber die ASP.NET-Laufzeitumgebung, welche sich um diese Angelegenheit kümmert. Der dynamische Kompilierungsvorgang wird grafisch in Abbildung 3.9 dargestellt.

Die ASP.NET-Laufzeitumgebung erhält beispielsweise eine Anfrage für die Seite *default.aspx*. Im Falle des vorliegenden Listings wird überprüft, ob für diese Seite bereits eine kompilierte .NET Assembly existiert. Wenn dies der Fall ist, dann befindet sich diese Assembly in einem speziellen Verzeichnis der ASP.NET-Installation: *<Installationsverzeichnis>\Temporary ASP.NET Files*. Hier finden sich verschiedene Unterverzeichnisse mit den temporären DLLs, die die gleichen Namen tragen wie die virtuellen Verzeichnisse für die jeweilige ASP.NET-Anwendung. Also z.B. *aspx\44f0f453\60305b04\9qxcl6dn.dll*. Und eben diese DLL ist die kompilierte Version unserer *.aspx*-Seite. Wenn die DLL aber nicht existiert oder aber der Zeitstempel der *.aspx*-Seite jünger ist als der Zeitstempel der Assembly, dann wird diese Assembly zur Laufzeit neu generiert. Die ASP.NET-Laufzeitumgebung erzeugt über eine spezielle Funktionalität des .NET-Frameworks (das so genannte *CodeDom*) eine Klasse, die von Page abgeleitet ist, und kompiliert diese Klasse in eine Assembly DLL. Die so genannte *Shadow Copy*-Funktionalität von ASP.NET sorgt wiederum dafür, dass eine *.aspx*-Datei geändert werden kann während gerade eine Anfrage von der alten Version der Assembly abgearbeitet wird. Nach Änderung der *.aspx*-Quelle wird bei der nächsten Anfrage eine neue Version der DLL kompiliert und die unter Umständen noch andauernde Abarbeitung der alten Anfrage kann ungestört zu Ende gebracht werden.

Die kompilierte Version unserer *.aspx*-Seite wollen wir uns einmal in einem Disassembler anschauen. Hierfür verwenden wir ILDASM, welches mit dem .NET Framework mitgeliefert wird (siehe Kapitel 2). Wie man in Abbildung 3.10 gut sehen kann, handelt es sich bei der temporären DLL um eine vollständige .NET Assembly. Besonders hervorzuheben ist die Tatsache, dass eine aus einer *.aspx*-Seite erzeugte Assembly immer von der Klasse *Page* abgeleitet ist. Dies bringt uns nun endlich zur Programmierung von *.aspx*-Seiten.

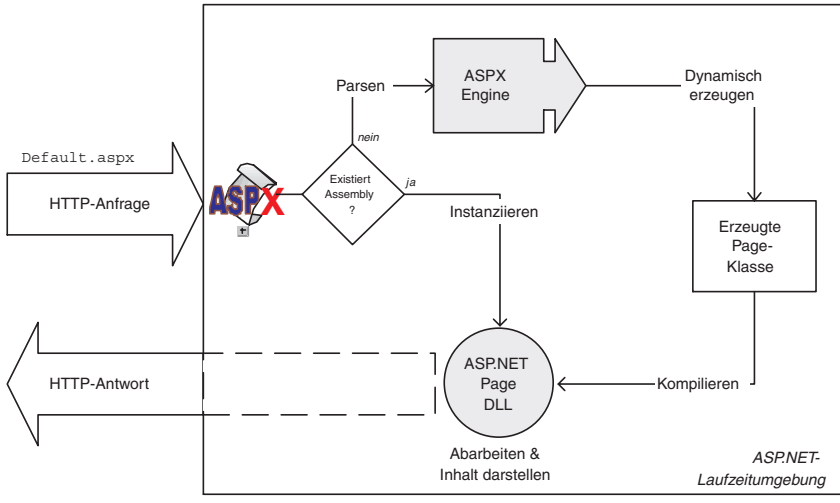


Abbildung 3.9: Dynamischer Kompilierungsvorgang in ASP.NET

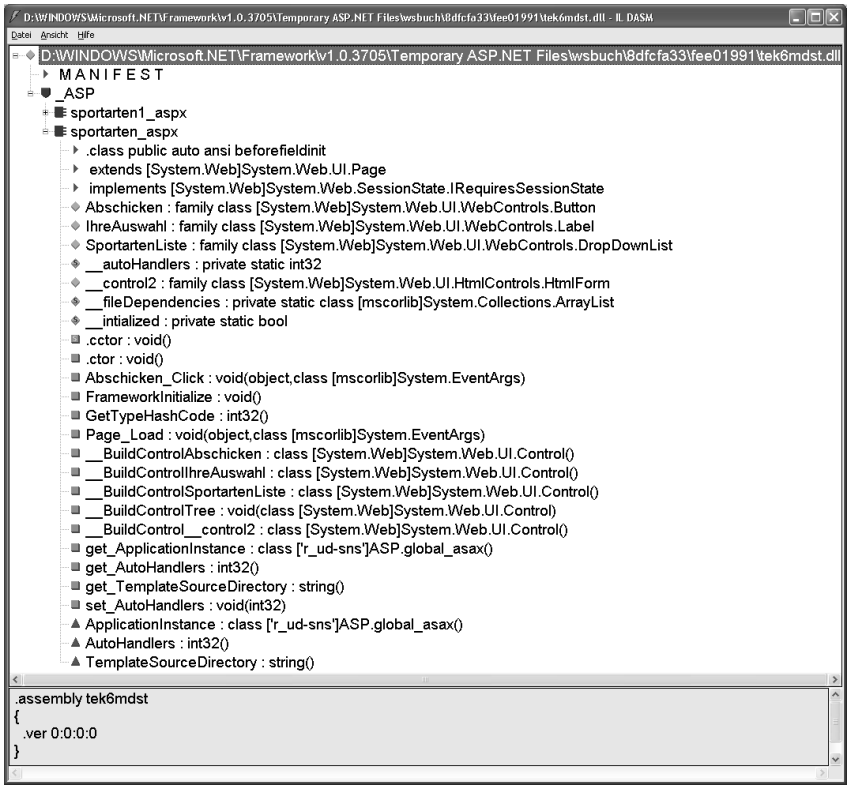


Abbildung 3.10: Kompilierte Assembly einer einfachen .aspx-Seite

3.4 ASP.NET Web Forms

Nachdem wir uns in den vorangegangenen Abschnitten hauptsächlich mit Trockenschwimmübungen hinsichtlich ASP.NET beschäftigt haben, wollen wir nun in die Programmierung von Web-basierten Anwendungen auf Basis von ASP.NET einsteigen. Nach Beendigung diese Abschnitts werden Sie in der Lage sein, erste *.aspx*-Anwendungen zu programmieren – und Sie werden sehen, dass es mehr Spaß macht und insbesondere produktiver ist als mit dem klassischen ASP. Willkommen in der Welt der ASP.NET Web Forms.

3.4.1 Einführung

ASP.NET ist auf der einen Seite eine Evolution von ASP, auf der anderen Seite aber auch eine ziemlich radikale Revolution. Evolution deshalb, weil ASP.NET in den größten Teilen abwärtskompatibel zu ASP ist. Dies bedeutet, dass es alle in ASP eingebauten Objekte auch in ASP.NET gibt. Man kann immer noch Spaghetti-Code erzeugen, indem man Skript-Code (der ja eigentlich kein Skript-Code mehr ist) zusammen mit der HTML-Darstellung mischt. Und man kann Code einfach von ASP nach ASP.NET portieren. Natürlich hat man dann noch keine der neuen und arbeitserleichternden Features von ASP.NET genutzt.

Revolution deshalb, weil ASP.NET außer der Kompilierung von Seiten (siehe vorhergehenden Abschnitt) ein zugleich neues und altes Programmiermodell mit sich bringt. Neu ist es für ASP- und andere Web-Entwickler. Alt für Visual Basic-, Visual C++- oder Java-Entwickler, die seit Jahren mit Benutzungsoberflächen zu tun haben. Die Rede ist von Ereignis-basierter Programmierung und der Verwendung von vorgefertigten Kontrollelementen. Neben diesen Neuerungen finden sich vor allem Ansätze zur Strukturierung von Code, zur besseren Verarbeitung von HTML-Formularen und zur Arbeit mit Datenbeständen.

Eine *.aspx*-Seite besteht prinzipiell aus einer oder mehreren Direktiven (siehe Abschnitt 3.4.3) und dem eigentlichen Seiteninhalt. Für eine *.aspx*-Seite ist etwa die *@Page*-Direktive als Einstiegspunkt vorgesehen. Wenn man nun folgenden Code in einer Datei mit der Endung *.aspx* in ein Verzeichnis einer ASP.NET-Anwendung kopiert, dann hat man die erste *.aspx*-Anwendung erstellt. Man kann auch einfach hergehen und eine existierende HTML-Seite in *.aspx* umbenennen – schon hat man eine *.aspx*-Seite.

```
<%@ Page Language="C#" %>
<%
Response.Write("Meine erste einfache ASP.NET Seite.");
%>
```

Listing 3.3: Erste einfache ASP.NET-Seite

3.4.2 Ereignis-basierte Programmierung

Eine der größten Neuerungen ist sicherlich die Ereignis-basierte Programmierung Server-seitiger Anwendungen. Anders als bei Visual Basic 6 üblich, kennt man in der klassischen HTML- und ASP-Programmierung keine Ereignisse (bis auf JavaScript-Code in der HTML-Seite). Im klassischen Ansatz werden die eingegebenen Werte in einem Formular schlicht über einen HTTP-GET oder -POST-Befehl an den Server gesendet. Dieser Server bzw. der ASP-Code muss dann für die korrekte Ab- und Verarbeitung der Werte sorgen. Dabei wäre es doch wesentlich intuitiver und einfacher zu programmieren, wenn man ähnlich wie in Visual Basic 6 für jede aktive und passive Aktion ein Ereignis hätte, welches ausgelöst wird, sobald eine Aktion gestartet wird. Zu aktiven Aktionen zähle ich hier das Drücken einer Schaltfläche, das Auswählen eines Wertes oder die Eingabe in ein Textfeld. Passive Aktionen wiederum sind beispielsweise das Laden einer Seite auf dem Server oder das Abarbeiten der unterschiedlichen Elemente einer Seite zur Generierung der HTML-Darstellung.

Wollen wir uns ein kleines Beispiel in ASP ansehen, in dem eine Liste in einer HTML-Seite aus einer Datenbank mithilfe des ADO Recordset-Objekts gefüllt wird. In dieser Seite kann man dann verschiedene Werte wählen und das Formular an den Server abschicken. Als Beispiel nehmen wir eine Umfrage unter Programmieren, welche Sportart sie in ihrer Freizeit betreiben.

```
<html>
<head>
<title>Umfrage</title>
</head>
<body>

<form method="POST" action="sportarten.asp">
   <h2>Umfrage: Ausgeübte Sportarten unter
   Programmierern</h2>
   <p>Welche Sportart betreiben Sie?<br>

   <select size="1" name="SportartenListe">
   <%
   Set oRS = Server.CreateObject("ADODB.Recordset")
   oRS.Open "SELECT * FROM Sportarten", &_
      "Driver={SQL Server};Server=(local); &_
         Database=WS_Buch;UID=BuchUser;PWD=ws_buch"

   While Not oRS.EOF
      Response.Write "<option value='" & oRS("ID") &_
         "'>" & oRS("Sportart") & "</option>" & vbCrLf
      oRS.MoveNext
   Wend
```

```
   oRS.Close
   Set oRS = Nothing
   %>
   </select>
   <p>
   <input type="submit" value="Abschicken" name="S">
   </p>
</form>
<%
If Len(Request.Form("SportartenListe")) > 0 Then
   Response.Write "Aha, Sie betreiben also " &_
      Request.Form("SportartenListe")
End If
%>
</body>
</html>
```

Listing 3.4: ASP-Seite zur Auswahl einer Option

Abbildung 3.11: Ausgabe der ASP-Seite zur Umfrage über Sportarten

Eine ähnliche ASP-Seite wird jeder schon einmal gesehen oder sogar programmiert haben. An diesem Beispiel können wir einige Unzulänglichkeiten von ASP gut ausmachen. Zum einen muss man den Code zum Füllen der Liste mitten in den HTML-Code hinein schreiben. Dies ist eines der größten Übel von ASP, welches auch immer wieder angeführt wurde, wenn

größere Projekte unübersichtlich zu werden drohten. Zum anderen ist es in ASP nicht sehr intuitiv, Daten eines Formulars an die gleiche Seite zu senden. Hier muss man extra Code schreiben, der überprüft, ob die Seite das erste Mal aufgerufen wurde oder bereits Daten übermittelt wurden. Nicht zuletzt sollten wir einen Blick auf die Ausgabe unserer ASP-Seite im Browser werfen – denn das ist eigentlich nicht die Antwort, die man erwartet.

Das ASP-Skript gibt uns als Ausgabe also den Wert von `option value` und nicht etwa die Zeichenkette, die in der Liste steht, zurück. Das ist aber nicht der Wert, den wir unserem Benutzer anzeigen wollen – um dies zu erreichen, müssten wir uns im Skript noch die Zuordnung der Listenelemente merken, um dann wirklich die jeweilige Sportart ausgeben zu können. Das ist sehr umständlich und unschön. Ein Nebeneffekt dieser Demonstration ist, dass der Wert der Liste nach Ausführung des Skripts immer wieder auf den Ursprungswert zurückgesetzt wird. Dies mag man als Schönheitsfehler abtun, jedoch kann in einer langen Liste das Merken des ausgewählten Wertes durchaus sinnvoll sein.

Sehen wir uns das obige Beispiel einmal unter ASP.NET an. Bitte beachten Sie, dass in diesem ASP.NET-Code bereits eine bestimmte Technik verwendet wird, um auf die Datenbank zuzugreifen. Zum jetzigen Zeitpunkt wollen wir dies einfach als gegeben hinnehmen. Eine genauere Erläuterung der Vorgänge kann man in Kapitel 4 über ADO.NET nachlesen.

```
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<html>
<head>
<script language="VB" runat="server">
   Sub Page_Load(source as Object, e As EventArgs)
      If Not Page.IsPostBack Then
         Dim oConn As SQLConnection
         Dim oCmd As SQLCommand
         Dim oDR As SQLDataReader

         oConn = New SQLConnection _
               ("server=localhost;uid=BuchUser;pwd=ws_buch;“ & _
               „database=WS_Buch")
         oConn.Open()

         oCmd = New SQLCommand _
               ("SELECT * FROM Sportarten", oConn)
         oDR = oCmd.ExecuteReader()

         SportartenListe.DataSource = oDR
         SportartenListe.DataBind()
      End If
   End Sub
```

```
   Sub Abschicken_Click(source As Object, _
      e As EventArgs)
      IhreAuswahl.Text = "Aha, Sie betreiben also " & _
         SportartenListe.SelectedItem.Text
   End Sub
</script>
</head>

<body>
<form method="post" runat="server">
   <h2>Umfrage: Ausgeübte Sportarten unter
   Programmierern</h2>
   <p>Welche Sportart betreiben Sie?<br>

   <asp:DropDownList id="SportartenListe"
        DataTextField="Sportart" DataValueField="ID"
        runat="server"/>
   <asp:Button id="Abschicken" Text="Abschicken"
        onClick="Abschicken_Click" runat="server"/>
   <br/><br/><br/><br/><br/><br/>

   <asp:Label id="IhreAuswahl" runat="server"/>
</form>
</body>
</html>
```

Listing 3.5: ASP.NET-Code für Umfrageseite

Sieht schon ein bisschen anders aus? Aber immer wieder der Hinweis: Wer schon einmal Visual Basic 6 oder Java Oberflächenentwicklung gemacht hat, wird sich hier wohl fühlen. Augenscheinlichstes Merkmal des Codes ist die Eigenschaft `runat="server"` einiger Tag-Elemente. Angefangen beim Skriptblock gleich zu Beginn des Quelltextes bis hin zu den einzelnen Tags für die Darstellung von HTML-Elementen, immer wieder ist diese Eigenschaft anzutreffen. Die Funktionsweise ist ganz einfach: Jedes Element, welches mit dieser Eigenschaft ausgezeichnet ist, wird auf dem Server durch ASP.NET abgearbeitet. Auch wenn es auf den ersten Blick wie ein Client-seitiges VBScript erscheinen mag, ist es doch Visual Basic .NET-Code, welcher speziell für die Auswertung auf dem Server geschrieben wird. Damit die Abarbeitung von Code auf dem Server genau kontrollierbar ist, stellt die ASP.NET-Laufzeitumgebung einige Ereignisse zur Verfügung, auf welche der Programmierer reagieren kann. In Abbildung 3.12 ist eine Übersicht der Ereignisse bei einem Seitenaufruf zu sehen. Als zweites Ereignis in der Ablauffolge kann man das *Page_Load*-Ereignis erkennen, welches auch in unserem Beispiel behandelt wird. Hier legt man üblicherweise Code ab, der beim Laden der Seite (Server-seitig) ausgeführt werden soll. In unserem Fall ist dies der Verbindungsaufbau zu der Datenbank und das Auslesen von dort gespeicherten Daten. Die zusätzliche `If`-Abfrage hat einen ganz beson-

deren Stellenwert auf den wir gleich zurückkommen werden. Die anderen Ereignisse werden wir nicht genauer besprechen, man kann als Übung mit ihnen experimentieren und sehen, wie sie sich auf die Generierung der Seite auswirken.

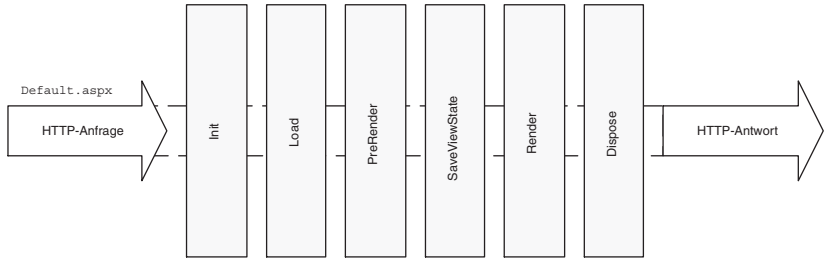


Abbildung 3.12: Reihenfolge der Ereignisse beim ersten Seitenaufruf

Für das Füllen eines Formulars aus einer Datenbank oder einer XML-Datei heraus muss in ASP die Ausleseaktion aus der Datenbank immer wieder ausgeführt werden. Wie kann man diesen Umstand in ASP.NET umgehen? Das Stichwort hierfür heißt *Postback*. Jede ASP.NET-Seite besitzt eine Eigenschaft *IsPostBack*, über die man erfahren kann, ob diese Seite das erste mal aufgerufen wird oder ob eine HTTP-POST-Aktion (z.B. aus einem HTML-Formular heraus) auf diese Seite erfolgt ist. Hintergrund ist folgender: In ASP ist es mitunter sehr schwierig, die Bearbeitung einer Seite und vor allem eines Formulars innerhalb der gleichen ASP-Seite vorzunehmen. In ASP.NET ist dies nun einfach durch Abfragen der *IsPostBack*-Eigenschaft möglich. In unserem Beispiel wird somit das Auslesen der Datenbank nur vorgenommen, wenn die Seite das erste Mal geladen wird. Doch wie soll dies funktionieren? Wie wir wissen, ist HTTP ein zustandsloses Protokoll. Wenn die Datenbank nur beim ersten Aufruf der Seite kontaktiert wird, sollten in allen darauf folgenden Aufrufen keine Datenzugriffe vorliegen. Wo ist der fehlende Stein im Mosaik? Er findet sich im *ViewState*. Eine ASP.NET-Seite und ein Server-seitiges Control (näheres hierzu in Abschnitt 3.5) besitzen einen ViewState, der dafür verantwortlich ist, sich Einstellungen oder Daten ‚zu merken'. So besitzt die im Beispiel verwendete *DropDownList* einen ViewState, der gesetzt wird nachdem die Bindung der Daten aus der Datenbank an das Server Control erfolgt ist. Dieser ViewState merkt sich zum einen die Daten, welche in der Liste angezeigt werden, und zum anderen, welcher dieser Werte zuletzt ausgewählt wurde. Somit hat man als Programmierer ein einfaches Mittel, den Zustand von HTML-Elementen abzufragen, ohne unnötige Zugriffe auf teure Ressourcen machen zu müssen.

Doch wie ist der ViewState realisiert? Am einfachsten zu verstehen ist dieses Konzept wenn man einen Blick auf den HTML-Quelltext wirft der zwischen Client und Server ausgetauscht wird (siehe Listing 3.6). Der Server bedient sich dieser Informationen im ViewState, um die auf dem Server gespeicherten Zustände von Controls zwischen verschiedenen Seitenaufrufen verfolgen zu können.

```
<html>
<head>
</head>
<body>
<form name="ctrl0" method="post" action="sportarten.aspx" id="ctrl0">
<input type="hidden" name="__VIEWSTATE"
value="dDwxNTY5OTcyNzI0O3Q8O2w8aTwyPjs+O2w8dDw7bDxpPDE+
Oz47bDxOPHQ8O3Q8aTw1PjtAPFZvbGxleWJhbGwgICAgICAgICAgICAgICAgICAgICAgIDtGdc
OfYmFsbCAgICAgICAgICAgICAgICAgICAgICAgICAgOOJpZXJOcmlua2VuICAgICAgICAgICAg
ICAgICAgICAgIDtQaXp6YWVzc2VuICAgICAgICAgICAgICAgICAgICA7TW9uaXRvcndlaXR3dX
JmICAgICAgICAgICAgICAgOz47QDwxOzI7MzsOOzU7Pj47Pjs7Pjs+Pjs+" />

   <h2>Umfrage: Ausgeübte Sportarten unter Programmierern</h2>
   <p>Welche Sportart betreiben Sie?<br>

   <select name="SportartenListe" id="SportartenListe">
   <option value="1">Volleyball                    </option>
   <option value="2">Fu&#223;ball                  </option>
   <option value="3">Biertrinken                   </option>
   <option value="4">Pizzaessen                    </option>
   <option value="5">Monitorweitwurf               </option>
</select>

   <input type="submit" name="Abschicken" value="Abschicken"
id="Abschicken" />

   <br/><br/><br/><br/><br/><br/>

   <span id="IhreAuswahl"></span>
</form>
</body>
</html>
```

Listing 3.6: Erzeugter HTML-Quelltext mit ViewState

Man sieht ein verstecktes HTML-Eingabefeld (`<input type="hidden" name="__VIEWSTATE" ...>`). Jedoch haben nicht wir dieses Feld programmiert, sondern ASP.NET hat es automatisch eingefügt und mit einem Wert belegt. Der Wert von `value` ist ein kodierter Wert, welcher den Zustand aller auf einer Seite befindlichen Controls widerspiegelt. Dieser Wert wird auf der Server-Seite automatisch in einem Speicherbereich gehalten und dort auch wieder ausgewertet. Dadurch verändert sich die Abarbeitung der Ereignisse auf dem Server wie in Abbildung 3.13 dargestellt. Mit dieser einfachen Methode wird dem Programmierer von *.aspx*-Seiten ein großer Teil der Arbeit bei der Erstellung der Logik abgenommen. Allerdings besteht auch die Gefahr, dass der ViewState bei vielen Controls und vielen Einstellungen sehr groß wird. Daher kann man den ViewState auch deaktivieren. Dies kann durch Setzen der *EnableViewState*-Eigenschaft für jedes Control oder für die gesamte Seite erfolgen.

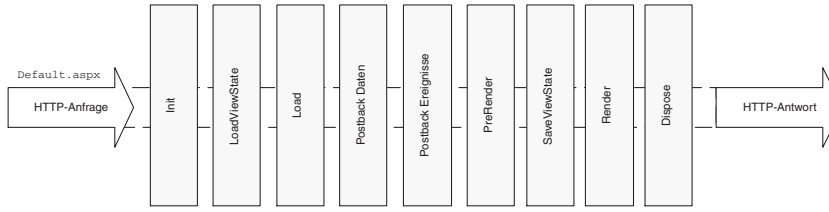


Abbildung 3.13: Reihenfolge der Ereignisse nach einem Postback

Schließlich ergibt sich ein Ergebnis unserer ASP.NET-Seite, welches dem Bildschirmfoto in Abbildung 3.14 gleicht. Egal welchen Wert man in der Liste auswertet, es wird nach der Abarbeitung auf dem Server immer der zuletzt ausgewählte Wert angezeigt. Mit diesem kleinen Beispiel wurde die grundlegende Funktionsweise von ASP.NET verdeutlicht und bereits ein paar spezifische Neuerungen vorgestellt.

Abbildung 3.14: Ergebnisseite der ASP.NET-Implementierung

3.4.3 Direktiven

Wenn man es genau nimmt, so ist unser Beispiel in Listing 3.5 keine richtige ASP.NET-Seite. Denn eine für ASP.NET bestimmte Datei sollte immer mit einer so genannten Direktive beginnen. Eine Direktive besteht aus einem @-Zeichen gepaart mit einem oder mehreren Schlagwörtern. Das ganze ist dann in einen Skriptblock mit <% und %> eingeschlossen. Damit unsere Seite auch offiziell eine ASP.NET-Seite wird, fügen wir ihr folgende Zeile gleich zu Beginn des Quelltextes hinzu:

```
<%@ Page Language="VB" %>
```

Wichtigste Eigenschaft ist hier die Wahl der Programmiersprache. Wenn diese Eigenschaft nicht gesetzt wird, geht ASP.NET automatisch von C# aus. Die *Page*-Direktive besitzt aber noch mehr Eigenschaften, welche vom ASP.NET-Parser und -Compiler verwendet werden können, hier jedoch nicht besprochen werden.

Außer der *Page*-Direktive stehen noch weitere Direktiven für den Gebrauch in einer Web Forms-Seite oder einem Benutzerelement (User Control, siehe Abschnitt 3.5.5) zur Verfügung. Tabelle 3.2 listet diese Direktiven auf und erklärt deren Bedeutung.

Direktive	Bedeutung
@Page	Definiert seitenspezifische Attribute, welche im ASP.NET-Parser und -Compiler verwendet werden (nur in Verbindung mit *.aspx*-Dateien möglich).
@Control	Definiert Kontrollelement-spezifische Attribute (nur in Verbindung mit *.ascx*-Dateien möglich).
@Import	Explizites Importieren eines Namensraums in die Seite oder in ein Control.
@Implements	Zeigt an, dass eine Seite oder ein Benutzerkontrollelement ein bestimmtes Interface implementiert.
@Register	Erzeugt einen Alias-Namen für Namensräume oder Klassennamen. Hierdurch wird eine mögliche Doppeldeutigkeit bei der Verwendung im Code verhindert.
@Assembly	Linkt eine spezifische Assembly explizit zu der Seite oder dem Control.
@OutputCache	Kontrolliert die Cache-Eigenschaft einer Seite oder eines Controls.
@Reference	Zeigt an, dass eine andere Seite oder ein anderes Control explizit zu dieser Seite oder diesem Control hinzu gelinkt werden soll.

Tab. 3.2: ASP.NET Web Forms-Direktiven

Interessant neben *Page* sind vor allem die Direktiven *Import* und *Assembly*. Mit diesen zwei Direktiven lassen sich die Eingabeparameter für den jeweiligen Sprachencompiler beeinflussen. Durch folgende Zeilen kann man den C#-Compiler mit externen Argumenten versorgen:

```
<%@ Page Language="C#" %>
<%@ Import namespace="eYesoft.WSBuch" %>
```

```
<%@ Import namespace="eYesoft.WSBuch.WebForms.
    Beispiele" %>
<%@ Assembly name="eYesoft_WSBuch" %>
// und hier dann der Code ...
```

Obiges Listing entspricht dem folgenden Compileraufruf:

```
csc.exe /r:eYesoft_WSBuch.dll main.cs
```

Die beiden Einträge unter Import würden in einem C#-Listing folgende Entsprechung haben:

```
using eYesoft.WSBuch;
using eYesoft.WSBuch.WebForms.Beispiele;
```

Obwohl Compiler und Parser von ASP.NET selbstständig nach abhängigen Assemblies suchen, kann man so über die entsprechende Direktive auch Assemblies verwenden, die in einem fremden Pfad liegen.

Die Einträge in der Liste der Eigenschaften einer Direktive wird durch ein Leerzeichen (*Space*) getrennt. Deshalb darf in der Wertzuweisung einer Eigenschaft kein Leerzeichen stehen! Folgendes ist also gültig: `<%@ Page Language="VB" enableViewState="True"%>` Wohingegen dieser Eintrag zu Problemen führt: `<%@ Page Language="VB" enableViewState = "True"%>`.

3.4.4 Inline und Code-Behind

Eines der großen Hindernisse für umfangreiche Code-Bibliotheken und Projekte in ASP war die Vermischung von HTML- und Skript-Code. Im bisher vorgestellten Beispiel für ASP.NET war dies nicht viel anders. Zwar erleichtert die Ereignis-basierte Programmierung in ASP.NET das Leben des Programmierers ungemein und sorgt auch für eine grundlegende Trennung von vermischtem HTML und Skript, jedoch ist die getrennte Entwicklung bzw. Pflege der Seiten dadurch noch nicht gewährleistet. ASP.NET führt für diese Problematik das so genannte *Code-Behind*-Konzept ein.

Im Gegensatz zur *Inline*-Technik, wie sie im obigen Beispiel verwendet wurde, wird beim Code-Behind die Implementierungslogik der Seite von vorne herein in einer separaten Datei untergebracht. Diese separate Datei kann anschließend einfach im Quelltext vorliegen oder aber schon durch einen Compiler vorkompiliert sein. Im ersten Fall erkennt ASP.NET diesen Umstand und erzeugt zunächst eine Assembly DLL für die Code-Behind-Datei. Anschließend wird die gesamte Seite dynamisch als Assembly generiert. Eine Code-Behind-Implementierung wird in der *Page*-Direktive verankert. Folgendes Beispiel deklariert eine Code-Behind-Datei für unser Beispielprogramm:

```
<%@ Page Language="vb" Src="WebForm1.aspx.vb"
Inherits="Sportarten.WebForm1" %>
```

Zusätzlich muss noch die Angabe erfolgen, welche Klasse aus der Code-Behind-Implementierung die *.aspx*-Seite implementiert. Eine Code-Behind-

Klasse muss nämlich immer von der Klasse *Page* aus dem Namensraum *System.Web.UI* abgeleitet sein. Listing 3.7 zeigt die *.aspx*-Seite, welche nun nur noch aus HTML-Code und Code für die Deklaration der Server-seitigen ASP.NET Controls besteht. Die zugehörige Implementierung in einer separaten Datei sieht dann wie in Listing 3.8 aufgezeigt aus.

Wenn in einer ASP.NET-Anwendung (beispielsweise in einer *.aspx*-Seite) ein Typ verwendet wird, welcher in einer anderen Assembly zu finden ist, diese aber nicht im Global Assembly Cache (GAC) installiert ist, dann muss sich diese Assembly standardmäßig in einem Unterverzeichnis *bin* befinden. So muss dies auch im Falle von vorkompilierten Code-Behind-Assemblies geschehen.

```
<%@ Page Language="vb" AutoEventWireup="false"
   Src="WebForm1.aspx.vb"
   Inherits="SrcWebForm1" %>
<HTML>
   <body >
      <form id="Form1" method="post" runat="server">
<P>
<asp:DropDownList id=DropDownList1
   DataTextField="Sportart" DataValueField="ID"
   runat="server"></asp:DropDownList>
<asp:Button id=Button1 runat="server"
   Text="Button"></asp:Button></P>
<P>
<asp:Label id=Label1
   runat="server">Label</asp:Label></P>
      </form>
   </body>
</HTML>
```

Listing 3.7: Entwurfsansicht einer .aspx-Seite mit Code-Behind

```
Imports System.Data
Imports System.Data.SqlClient

Public Class WebForm1
      Inherits System.Web.UI.Page

      Protected WithEvents DropDownList1 As
         System.Web.UI.WebControls.DropDownList
      Protected WithEvents Button1 As
         System.Web.UI.WebControls.Button
      Protected WithEvents Label1 As
         System.Web.UI.WebControls.Label

      Private Sub Page_Load(ByVal sender As
         System.Object, ByVal e As System.EventArgs)
```

```
        Handles MyBase.Load
        Dim oConn As SQLConnection
        Dim oCmd As SQLCommand
        Dim oDR As SQLDataReader

        oConn = New SQLConnection
           ("server=localhost;uid=BuchUser;
            pwd=ws_buch;database=WS_Buch")
        oConn.Open()

        oCmd = New SQLCommand
           ("SELECT * FROM Sportarten", oConn)
        oDR = oCmd.ExecuteReader()

       DropDownList1.DataSource = oDR
       DropDownList1.DataBind()

    End Sub

    Private Sub Button1_Click(ByVal sender As
       System.Object, ByVal e As System.EventArgs)
       Handles Button1.Click
       Label1.Text = "Aha, Sie betreiben also " &
          DropDownList1.SelectedItem.Text
  End Sub
End Class
```

Listing 3.8: Code-Behind-Implementierung der Seitenlogik

Durch diese Vorgehensweise kann man besser die Erstellung der HTML-Oberfläche von der Programmierung der Seitenlogik trennen. Natürlich wird eine vollständige Trennung nie erreicht werden können. Denn es ist unabdingbar, dass in der Entwurfsansicht Stellen markiert werden, an denen dann entsprechende Code-Fragmente oder Ereignisimplementierungen aus der Code-Behind-Klasse aufgerufen werden. Dies liegt einfach in der Natur von HTML.

Nachdem wir in diesem und in vorhergehenden Abschnitten schon des öfteren von den Server Controls in ASP.NET gesprochen haben, wollen wir nun einen genaueren Blick auf diese werfen.

3.5 Server Controls

Damit die Vorteile der Ereignis-basierten Programmierung durch ständiges Neuschreiben von bestimmten Code-Blöcken nicht wieder zunichte gemacht werden, haben sich die Erfinder von ASP.NET noch einmal bei der Programmierung von traditionellen grafischen Benutzungsoberflächen umgesehen. Ein Visual Basic-Programmierer kennt seit Jahren die Erstellung komplexer

Oberflächen einfach durch Zusammensetzung und Verwendung zahlreicher, vorgefertigter Kontrollelemente (*Controls*). Diese Controls liegen im Falle von Visual Basic in binärer Form vor und werden in der Entwicklungsumgebung auf eine dafür vorgesehene Fläche gezogen. Ihre Eigenschaften kann man einfach entweder programmatisch oder durch Setzen in einem Bereich der IDE festlegen. Und eben diese Funktionalität versuchten die ASP.NET-Entwickler zu kopieren. Mit Erfolg.

Die moderne Programmierung von ASP.NET-Web-Anwendungen basiert fast ausschließlich auf Nutzung bereits vorhandener Controls. Microsoft liefert mit der ASP.NET-Installation und mit Visual Studio .NET bereits eine Vielzahl solcher Controls mit, jedoch ist es auch möglich, eigene Controls zu erstellen und diese dann anderen Entwicklern zur Verfügung zu stellen. Ein Server Control erkennt in der Regel am eingehenden HTTP-Request, um welchen Client-Browser es sich handelt. Mit der entsprechenden internen Implementierung kann das Control dann je nach Browser-Version eine andere Darstellung generieren. Somit wird die Programmierung und Pflege von Cross-Browser-Lösungen vereinfacht. Eine Cross-Browser-Lösung ist beispielsweise dann notwendig, wenn sich in einem Unternehmen mehrere Zielplattformen wie Windows, Linux oder MacOS befinden. Auf diesen Plattformen laufen jeweils unterschiedliche Web-Browser mehrerer Hersteller in unterschiedlichen Versionen.

Wir haben bereits in Abschnitt 3.4.2 die Verwendung von Server Controls gesehen:

```
<asp:DropDownList id="SportartenListe"
   DataTextField="Sportart" DataValueField="ID"
   runat="server"/>
```

Dieses Beispiel ist eine Server Control-Implementierung für eine HTML-Auswahlliste. Allerdings ist dies nicht einfach die Umsetzung der ursprünglichen HTML-Liste in ASP.NET. Vielmehr kann diese spezielle Version beispielsweise direkt mit einer Datenquelle über das Setzen der entsprechenden Eigenschaften verknüpft werden. Dieses Control zählt zur Kategorie ASP.NET-*Web Controls*. Eine zweite Kategorie sind die einfacher gehaltenen *HMTL Controls*, die wir zuerst betrachten werden.

3.5.1 HTML Controls

HTML Controls können als eins-zu-eins Abbildung der jeweiligen HTML-Tags gesehen werden. Sie wurden u.a. eingeführt, um eine einheitliche Programmierweise in ASP.NET vorzufinden. In Tabelle 3.3 sehen Sie eine Auflistung aller verfügbaren HTML Controls, die mit der ASP.NET-Installation ausgeliefert werden. HTML Controls sind für die Gestaltung einfacher Oberflächen meist ausreichend. Wenn es aber um komplexere Abläufe und grafische Darstellungen geht, kommt man nicht um die Verwendung von ASP.NET-Web Controls herum. Allerdings ist der Umfang des erzeugten HTML-Codes bei HTML Controls wesentlich geringer als bei den mächtigeren Web Controls.

Tab. 3.3: Liste aller HTML Controls in ASP.NET

Funktion	Control	Beschreibung
HTML-Formulare	*HtmlForm*	Definiert ein HTML-Formular.
Text anzeigen und bearbeiten	*HtmlInputText*	Darstellung von Text, der von Benutzern zur Laufzeit bearbeitet werden kann. Auch als Variante für Passwörter einsetzbar.
Text anzeigen und bearbeiten	*HtmlTextArea*	Darstellung und Bearbeitung von großen Texten über mehrere Zeilen.
Kommando (führt zu einem HTTP-POST an den Server)	*HtmlButton*	Führt eine Aktion aus. Dieses Control kann beliebiges HTML enthalten und ist dadurch sehr flexibel. Es ist nicht mit allen Browsern kompatibel.
Kommando	*HtmlInputButton*	Führt eine Aktion aus. Ist mit allen Browsern kompatibel.
Kommando	*HtmlInputImage*	Ähnlich wie *HtmlInputButton*, stellt aber ein Bild dar.
Auswahl eines Wertes	*HtmlSelect*	Darstellung eine Auswahlliste.
Bilddarstellung	*HtmlImage*	Darstellung eines Bildes.
Halten von Daten	*HtmlInputHidden*	Manuelles Speichern von Zustandsdaten oder Extrainformationen im HTML-Formular.
Navigation	*HtmlAnchor*	Erzeugt einen Navigationslink.
Setzen von Werten	*HtmlInputCheckbox*	Erzeugt eine HTML-Checkbox für das Setzen von Optionen.
Setzen von Werten	*HtmlInputRadioButton*	Erzeugt einen HTML-Radio Button für die Auswahl eines bestimmten Wertes in einer Liste mit mehreren Optionen.
HTML-Tabellen	*HtmlTable*	Erzeugt eine Tabelle.

Funktion	Control	Beschreibung
HTML-Tabellen	*HtmlTableRow*	Erzeugt eine Reihe innerhalb einer Tabelle.
HTML-Tabellen	*HtmlTableCell*	Erzeugt eine Zelle in einer Tabellenreihe.
Dateiübertragung	*HtmlInputFile*	Ermöglicht das Hochladen von Dateien (muss vom Server unterstützt werden).
Andere	*HtmlGenericControl*	Generisches Control.

Tab. 3.3: Liste aller HTML Controls in ASP.NET (Forts.)

3.5.2 Web Controls

Anders als die HTML Controls können die Web Controls wesentlich komplexer gestaltet sein und auch komplexere Abläufe in sich vereinen. Oftmals sind Web Controls auch nur für spezielle Browser geeignet, die beispielsweise HTML 4.0, JavaScript 1.3 und *Cascading Stylesheets* (*CSS*) unterstützen. Sie sind gewissermaßen eine abstraktere Version von HTML Controls, da sie nicht wie diese zwingend eine HTML-Syntax verkörpern. Da Web Controls im Grunde nichts anderes sind als implementierte Klassen im .NET-Framework, verfügen sie auch über öffentliche Methoden und Eigenschaften. Üblicherweise werden diese Eigenschaften im Entwurfsmodus innerhalb eines Control-Tags gesetzt, wie in Listing 3.5 gezeigt. Allerdings kann ein Programmierer auch aus der Programmlogik (z.B. innerhalb einer Code-Behind-Seite) auf die Funktionalität des Controls zugreifen und diese beeinflussen. Eine Liste der verfügbaren Web Controls mit einer kurzen Beschreibung ihrer Funktion kann man in Tabelle 3.4 einsehen.

Funktion	Control	Beschreibung
Text anzeigen	*Label*	Darstellung von nicht-editierbarem Text.
Text editieren	*TextBox*	Darstellung von Text, der auch geändert werden kann.
Auswahl aus einer Liste	*DropDownList*	Darstellung einer Auswahlliste. Auswahl eines Wertes oder Eingabe eines neuen Wertes.
Auswahl aus einer Liste	*ListBox*	Darstellung einer Anzahl an Auswahlmöglichkeiten. Mehrfachauswahl möglich (optional).

Tab. 3.4: Liste aller Web Controls in ASP.NET

Tab 3.4: Liste aller Web Controls in ASP.NET (Forts.)

Funktion	Control	Beschreibung
Grafikdarstellung	*Image*	Darstellung eines Bildes.
Grafikdarstellung	*AdRotator*	Anzeige einer Bildsequenz (vorgegeben oder zufällig), z.B. für Werbebanner.
Auswahl von Werten	*CheckBox*	Darstellung einer Checkbox. Beinhaltet ein Label.
Auswahl von Werten	*CheckBoxList*	Erzeugt eine Gruppe von Checkboxen. Unterstützt Data Binding.
Auswahl von Werten	*RadioButton*	Erzeugt einen Radio Button.
Auswahl von Werten	*RadioButtonList*	Erzeugt eine Gruppe von Radio Buttons, innerhalb derer nur ein Button ausgewählt werden kann.
Datumsauswahl	*Calendar*	Darstellung eines grafischen Kalenders.
Kommandos (führt zu einem HTTP-POST an den Server)	*Button*	Ausführung einer Aktion.
Kommandos	*LinkButton*	Ähnlich wie Button, wird aber als Hyperlink dargestellt.
Kommandos	*ImageButton*	Ähnlich wie Button, aber Anzeige eines Bildes statt Text.
Navigation	*HyperLink*	Erzeugt einen Navigationslink.
HTML-Tabellen	*Table*	Erzeugt eine Tabelle.
HTML-Tabellen	*TableRow*	Erzeugt eine Reihe innerhalb einer Tabelle.
HTML-Tabellen	*TableCell*	Erzeugt eine Zelle in einer Tabellenreihe.
Gruppierung	*Panel*	Erzeugt einen randlosen Bereich innerhalb der Web Form. Dient als Container für andere Controls.

Funktion	Control	Beschreibung
Listendarstellung	*Repeater*	Darstellung von Informationen aus einem *DataSet* (siehe Kapitel 4). Automatisches Durchlaufen des *DataSet*.
Listendarstellung	*DataList*	Ähnlich wie *Repeater*. Bietet mehr Formatierungs- und Darstellungsoptionen. Kann auch zum Ändern von Werten verwendet werden.
Listendarstellung	*DataGrid*	Darstellung von Informationen in Tabellenform. Die Daten stammen üblicherweise aus einer Data Binding-Quelle. Ermöglicht Editieren und Sortieren.

Tab 3.4: Liste aller Web Controls in ASP.NET (Forts.)

Beispiele für Web Controls haben wir bereits weiter oben gesehen. In unserem Beispiel aus Listing 3.5 wurden zwei dieser Controls verwendet: DropDownList und Button. Web Controls werden anders als die HTML Controls über eine spezielle Syntax angesprochen. Bei ihnen wird immer ein *asp:* als Namensraumkennzeichner vorangestellt, damit keine Verwechslung oder Doppeldeutigkeit innerhalb einer Seite auftreten kann.

3.5.3 Validierung

Eines der meist benötigten Merkmale in einer Web-Anwendung ist das Überprüfen von Formularen. In ASP und ähnlichen Technologien muss man per Hand jedes einzelne Formularfeld anfassen und Code für die Auswertung und Überprüfung schreiben. In ASP.NET haben wir jedoch das Konzept der Server Controls und können diese für unsere Validierungsaufgaben etwas erweitern.

Eine spezielle Klasse von Web Controls sind die *Validation Controls*. Sie ermöglichen die automatische Überprüfung von Formularinhalten, ohne eine Zeile Logik zu implementieren. Will man eine Textbox auf einen bestimmten Wert hin überprüfen, dann verwendet man das *CompareValidator* Control. Ein Beispiel ist in Listing 3.9 zu sehen. Das Beispiel ist zwar kurz, aber aussagekräftig. Es zeigt die prinzipielle Vorgehensweise beim Einsatz von Validation Controls. Ein Validation Control wird immer genau jenem Web Control zugewiesen, dessen Werte es überprüfen soll. In diesem Fall wird ein *CompareValidator* Control einem *TextBox* Control zugewiesen. Durch

das Setzen der Eigenschaften im *CompareValidator* Control erreichen wir die beabsichtigte Überprüfung: Der Benutzer soll in das Textfeld einfach nur eine Zahl 0 oder größer 0 eingeben können. Stellen Sie sich jetzt einmal die Programmierung der zugehörigen Logik im klassischen ASP-Umfeld vor!

Eine komplette Auflistung der verfügbaren Controls mit Beschreibung ihrer Tätigkeit kann man in Tabelle 3.5 nachlesen.

```
<TABLE>
<TR>
<TD>
   <asp:Textbox id="txtAlter"
      runat="server"></asp:Textbox>
</TD>
<TD>
   <asp:CompareValidator id="myCompareFieldValidator"
      runat="server"
      ForeColor="Red"
      ControlToValidate="txtAlter"
      ValueToCompare=0
      Type="Integer"
      Operator="GreaterThanEqual"
      ErrorMessage="Bitte geben Sie eine Zahl 0
         oder größer 0 ein!">
   </asp:CompareValidator >
</TD>
</TR>
</TABLE>
```

Listing 3.9: Überprüfung eines Formularwerts mit einem Validator Control

Tab. 3.5: Liste aller Validation Controls in ASP.NET

Art der Validierung	Control	Beschreibung
Erwartete Eingabe	*RequiredFieldValidator*	Stellt sicher, dass der Benutzer einen Wert eingibt.
Vergleich mit einem Wert	*CompareValidator*	Vergleicht Benutzereingabe mit einem Referenzwert. Dies kann eine Konstante sein, ein Wert einer Eigenschaft oder ein Datum aus einer Datenbank.
Überprüfung eines Wertebereichs	*RangeValidator*	Stellt sicher, dass ein eingegebener Wert innerhalb eines spezifizierten Bereichs liegt.

Art der Validierung	Control	Beschreibung
Pattern Matching	*RegularExpressionValidator*	Vergleicht die Eingabe mit einem vorgegebenen Muster. Das Muster wird über einen regulären Ausdruck definiert.
Benutzerdefiniert	*CustomValidator*	Benutzer stellt eine eigene Logik zur Überprüfung der Eingaben zur Verfügung.

Tab 3.5: Liste aller Validation Controls in ASP.NET (Forts.)

Standardmäßig erfolgt die Überprüfung in einem Client-seitigen Skript, wenn der Browser diese Funktionalität unterstützt. Oft ist es aber der Fall, dass man die Client-seitige Überprüfung durch JavaScript nicht haben möchte. Dieses Verhalten kann man beeinflussen, indem man die Eigenschaft *EnableClientScript* des betroffenen Validation Controls auf *False* setzt. Dann wird nur noch auf der Server-Seite überprüft. Es sollte übrigens immer auf der Server-Seite überprüft werden, auch wenn bereits in einem Client-Skript die Werte validiert wurden. Denn mit Client-Skripts gab es in der Vergangenheit und wird es auch in Zukunft immer wieder Probleme hinsichtlich Sicherheitslöchern geben.

3.5.4 Data Binding

Das Arbeiten mit Daten aller Art ist in vielen Anwendungen das A und O. So auch in Web-basierten Anwendungen unter ASP.NET. Egal ob Arrays, Listen, Aufzählungen oder spezielle auf Datenbankinhalte abgestimmte Datenstrukturen: Daten und Werte werden innerhalb einer Anwendung ständig benötigt und werden in den unterschiedlichsten Gebieten bearbeitet. ASP.NET bietet speziell für diesen Bereich das *Data Binding*. Data Binding ermöglicht es, dass Datenbehälter, die einer gewissen Ausprägung genügen, praktisch an jedes in ASP.NET verfügbare Server Control gebunden werden können. Man braucht so keinen eigenen Code zur Verwaltung von Listen o.ä. zu schreiben, um diese Daten dann in einer Schleife beispielsweise einer Liste oder einem Textfeld zuzuordnen, wie dies noch in ASP der Fall war.

Für das Data Binding wird eine spezielle Syntax verwendet, die ein bisschen an ASP erinnert:

```
<%# Kunde.Name %>
```

Data Binding-Elemente können beliebig auf einer *.aspx*-Seite platziert werden und ihre Auswertung erfolgt, wenn die Funktion *Page.DataBind()* ausgeführt wird. Somit ähnelt die Syntax der von ASP (`<%= Kunde.Name %>`), allerdings mit dem Unterschied, dass in ASP diese Zuweisung sofort ausgewertet wird. Ein kleines Beispiel soll die Verwendung und die Vorteile von

Data Binding verdeutlichen. Unser Beispiel für die Umfrage von Sportarten unter Programmierern wollen wir dafür etwas abändern. Ähnlich wie in Abschnitt 3.4.2 verwenden wir eine DropDownList zur Darstellung der Optionen. Nur dieses Mal kommen die Werte nicht aus der Datenbank, sondern wir füllen direkt im Code ein *ArrayList*-Objekt (vgl. Listing 3.10). Nun ist es aber egal, ob diese Daten aus einer XML-Datei, einer Datenbank, von einem Web Service oder aus einer anderen Quelle stammen. Es muss lediglich dafür gesorgt werden, dass die eingehenden Daten in eine »bindbare« Form gebracht werden. Dies bedeutet, dass die *DataSource*-Eigenschaft eines Controls jedes Objekt annimmt, welches die Schnittstellen *IEnumerable* oder *ICollection* implementiert. Tabelle 3.6 zeigt eine Auswahl an Datentypen, die man für Data Binding an Web Controls verwenden kann. Die letzten fünf Einträge werden vornehmlich beim Einsatz mit ADO.NET – der Datenzugriffstechnologie in .NET (siehe Kapitel 4) – verwendet.

```
<html>
<head>
   <script language="C#" runat="server">
      void Page_Load(Object Sender, EventArgs E)
      {
         if (!Page.IsPostBack)
        {
            ArrayList sportarten = new ArrayList();
            sportarten.Add ("Volleyball");
            sportarten.Add ("Fußball");
            sportarten.Add ("Biertrinken");
            sportarten.Add ("Pizzaessen");
            sportarten.Add ("Monitorweitwurf");

            DropDown1.DataSource = sportarten;
            DropDown1.DataBind();
        }
      }

      void SubmitBtn_Click(Object sender, EventArgs e)
      {
         Label1.Text = "Aha, Sie betreiben also " +
            DropDown1.SelectedItem.Text;
      }
    </script>
</head>
<body>

   <h2>Umfrage: Ausgeübte Sportarten unter
   Programmierern (Data Binding)</h2>

   <form runat=server>
      <asp:DropDownList id="DropDown1" runat="server" />
      <asp:button Text="Submit"
```

```
         OnClick="SubmitBtn_Click" runat=server/>
      <p>
      <asp:Label id=Label1 font-name="Verdana"
         font-size="10pt" runat="server" />
   </form>
</body>
</html>
```

Listing 3.10: Data Binding mit einem ArrayList-Objekt

Mit der Data Binding-Technologie wird die Erstellung und vor allem die Pflege datengetriebener Web-Anwendungen sehr viel einfacher als mit dem klassischen ASP. Und sie ist zudem noch offen und erweiterbar.

Datentyp
ArrayList
HashTable
SortedList
Stack
StringCollection
DataView
DataTable
DataSet
SqlDataReader
OleDbDataReader

Tab. 3.6: Liste der bindbaren Datentypen (Auswahl)

3.5.5 User Controls

Nun ist es ein großer Fortschritt, dass ASP.NET die Server Controls eingeführt hat. Mit ihnen wird sich die Entwicklungszeit erheblich verkürzen. Doch was macht man, wenn es für die aktuellen Anforderungen gerade kein passendes Server Control gibt? Genau: man erstellt ein eigenes und kann dieses dann immer wieder verwenden. Wiederverwendung in Reinkultur. Auch hierfür ist in ASP.NET vorgesorgt. Die entsprechende Technologie heißt User Controls (auch als *Pagelets* bekannt).

Neben den User Controls gibt es auch noch die *Custom Controls*. Diese sind im Vergleich zu den User Controls wesentlich mächtigere Werkzeuge, benötigen aber auch deutlich mehr Implementierungsaufwand. Custom Controls werden so programmiert, dass der Entwickler eigenhändig im Code festlegt, welches Derivat von HTML ausgegeben werden soll. Zudem sind die Custom Controls im Vergleich zu den User Controls echte .NET Assemblies und liegen in Form von DLLs vor.

User Controls erkennt man leicht an der Dateiendung *.ascx*. Das *c* steht hierbei für *Control*. Vereinfacht gesprochen ist ein User Control ein Teil einer Web Form, der immer wieder in unterschiedlichen Kontexten verwendet werden muss. Daher wird genau dieser Teil in eine *.ascx*-Datei ausgelagert. Will man das Control dann in einer neuen Web Form verwenden, referenziert man dieses *.ascx*-Element und platziert an der geeigneten Stelle im Web Form die entsprechende Verankerung. Ein Beispiel soll diese Vorgehensweise verdeutlichen.

Ein typischer Anwendungsfall für die Erstellung eines User Controls ist die Programmierung eines Anmeldedialogs (Login). Ein Anmeldedialog besteht in den meisten Fällen aus Eingabefeldern für den Benutzernamen und für das Kennwort. Zusätzlich möchten wir aber noch die Hintergrundfarbe und die Schriftart des Dialogs durch Belegung von Parametern ändern können. Hierfür müssen wir in unserem Control öffentliche Eigenschaften definieren, die wir dann in einer beliebigen Web Form ändern können. Der Quelltext für das Login User Control steht in Listing 3.11. Das Layout des Controls wird über eine einfache HTML-Tabelle festgelegt. Durch das Setzen der Control-Eigenschaften werden automatisch die entsprechenden Eigenschaften in der Tabelle bzw. in den darin eingebetteten Web Controls gesetzt.

Als Beispiel für die Verwendung des Login Controls bauen wir das Control in unsere Umfrageseite ein. Der Code ist in Listing 3.12 einzusehen. Die wichtigste Neuerung steht hier gleich zu Beginn der Datei. Über die *@Register*-Direktive wird der Web Form mitgeteilt, dass im weiteren Verlauf ein externes User Control verwendet wird. *TagPrefix* zeigt an, welchen eindeutigen Namensraumbezeichner man dem Control geben möchte. *TagName* zeigt den Namen des Controls an und *Src* schließlich verweist auf die eigentliche Implementierung des Login-Controls in einer *.ascx*-Datei.

```
<script language="C#" runat="server">
   public String hgFarbe = "white";
   public String fnStil = "Arial";
   public String UserId;
   {
      get
      {
         return User.Text;
      }
      set
      {
         User.Text = value;
      }
   }
   public String Password
   {
      get
      {
         return Pass.Text;
      }
```

```
      set
      {
         Pass.Text = value;
      }
   }
</script>

<table style="background-color:<%= hgFarbe %>;font: 10pt
<%= fnStil %>;border-width:1;border-style:solid;border-color:black;"
cellspacing=15>
   <tr>
      <td><b>Benutzername: </b></td>
      <td><ASP:TextBox id="User" runat="server"/></td>
   </tr>
   <tr>
      <td><b>Passwort: </b></td>
      <td><ASP:TextBox id="Pass" TextMode="Password" runat="server"/>
      </td>
   </tr>
   <tr>
      <td></td>
      <td><ASP:Button Text="Anmelden" runat="server"/></td>
   </tr>
</table>
```

Listing 3.11: User Control für einen Anmeldedialog

```
<%@ Register TagPrefix="eYesoft" TagName="Login" Src="login.ascx" %>

<html>
<script language="C#" runat="server">
   void Page_Load(Object sender, EventArgs E)
   {
      if (Page.IsPostBack)
      {
         MyLabel.Text += "Der Benutzername lautet: " +
            MyLogin.UserId + "<br>";
         MyLabel.Text += "Das Passwort lautet: " +
            MyLogin.Password + "<br>";
      }
   }
</script>
<body style="font: 10pt verdana">
   <h2>Umfrage: Ausgeübte Sportarten unter
   Programmierern<h2>
   <h3>Login User Control</h3>
```

```
    <P>Bitte melden Sie sich für die Umfrage am
    System an.</P>
    <form runat="server">
       <eYesoft:Login id="login" UserId="Christian Weyer"
       Password="Testi" runat="server"/>
    </form>
    <asp:Label id="MyLabel" runat="server"/>
</body>
</html>
```

Listing 3.12: Verwendung des Login User Controls in einer Web Form

3.6 ASP.NET Web Forms mit Visual Studio .NET

»Wieso soll ich den weiten Weg zu Fuß gehen, wenn ich mit dem Auto schneller und bequemer hinkomme?« Bisher haben wir .NET und ASP.NET mit Notepad oder einem ähnlichen Editor programmiert. Das ist kein großes Problem, es funktioniert. Allerdings wird diese Art der Softwareentwicklung mit der Zeit mühselig. Vor allem wenn das Projekt immer mächtiger und umfangreicher wird. In diesem Abschnitt wollen wir uns die Erstellung einer Web Forms-Anwendung mit Visual Studio .NET ansehen. Als Beispiel nehmen wir wieder unsere Umfrageseite.

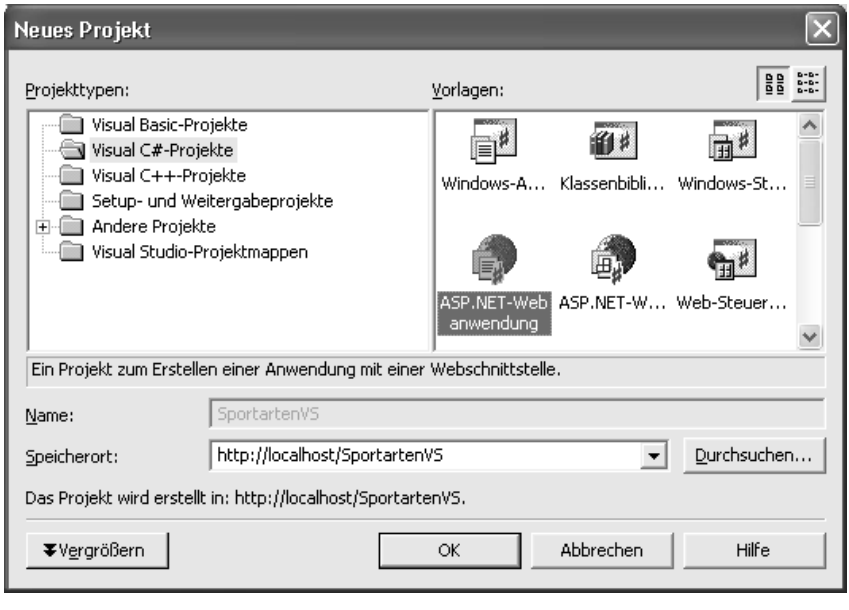


Abbildung 3.15: Auswahl eines Projektes für die Erstellung von Web Forms-Anwendungen im Visual Studio .NET

Über den Menüeintrag DATEI – NEU – PROJEKT gelangt man auf die Auswahlmaske für die verschiedenen Projekttypen (siehe Abbildung 3.15). In diesem Dialog wählen wir die Projektoption ASP.NET-WEBANWENDUNG. Hinter diesem Projekt verbirgt sich eine auf ASP.NET aufsetzende Web Forms-Anwendung. Damit der Projektassistent weiß, auf welchem Server und unter welchem virtuellen Verzeichnis die Anwendung angelegt werden soll, muss in dieser Maske ein gültiger URL zu einem IIS angegeben werden. Zusammen mit dem gewählten Projektnamen ergibt diese Eintragung den kompletten URL der Web Forms-Anwendung. Nach Abschluss dieses Dialogs erstellt Visual Studio automatisch eine IIS-Anwendung auf dem eingetragenen Web-Server (vgl. Abbildung 3.16). Dieser Vorgang kann einige Minuten dauern. Die Einstellungen für diese IIS-Anwendung kann man später in der INTERNET-INFORMATIONSDIENSTE-Anwendung überprüfen und ggf. seinen Anforderungen entsprechend anpassen. Nachdem diese Aktion abgeschlossen ist, erscheint die Visual Studio-Oberfläche. Besonderes Augenmerk wollen wir auf den PROJEKTMAPPEN-EXPLORER werfen (Abbildung 3.17). Hier sind sämtliche Dateien für das Projekt versammelt. Allerdings ist standardmäßig die sehr hilfreiche Option ALLE DATEIEN ANZEIGEN zur erweiterten Darstellung der Dateien deaktiviert. Zur Aktivierung klickt man auf die in Abbildung 3.17 markierte Schaltfläche. Dadurch werden jetzt alle relevanten Dateien angezeigt, die im Dateisystem existieren. Dies ist vor allem bei der Arbeit mit der Code-Behind-Implementierung von Visual Studio hilfreich.

Abbildung 3.16: Erzeugung einer IIS-Anwendung durch den Projektassistenten

Im Folgenden wollen wir einen Blick auf die im Solution Explorer aufgeführten Dateien werfen. Unter dem Ordner VERWEISE befinden sich alle .NET Assemblies, die zum Projekt hinzugelinkt werden. Diese Liste entspricht also den Einträgen in der Referenzenliste beim C#- oder Visual Basic .NET-Compiler (z. B. *csc.exe /r:System.dll /r:System.Data.dll WebForm1.aspx.cs*). Visual Studio hat beim Erstellen des Projekts automatisch eine Web Form-Seite hinzugefügt. Die Implementierung der Web Form erfolgt in den Dateien *WebForm1.aspx* und *WebForm1.aspx.cs*. *WebForm1.aspx* ist die Design-Implementierung (HTML- und Control-Tags) der Web Form, während *WebForm1.aspx.cs* die Code-Behind-Implementierung für die Logik beinhaltet. Weiterhin von Bedeutung ist die Datei *web.config*, über die Konfigurationseinstellungen für die Web-Anwendung erfolgen (Teile der Konfigurationsmöglichkeiten werden in Kapitel 5 und 7 besprochen). Sämtliche Informationen zu Autor, Firma,

Version und ähnliches über die kompilierte Assembly wird in der Datei *AssemblyInfo.cs* festgehalten. Eine wichtige Datei stellt die g*lobal.asax* dar (siehe Abschnitt 3.3.3). Hier werden anwendungs- und sitzungsspezifische Daten und Ereignisse verarbeitet. Die *.resx*-Dateien im Solution Explorer basieren übrigens auf XML und dienen der Verwaltung von Ressourcen – wie z. B. Bilder – durch Visual Studio.

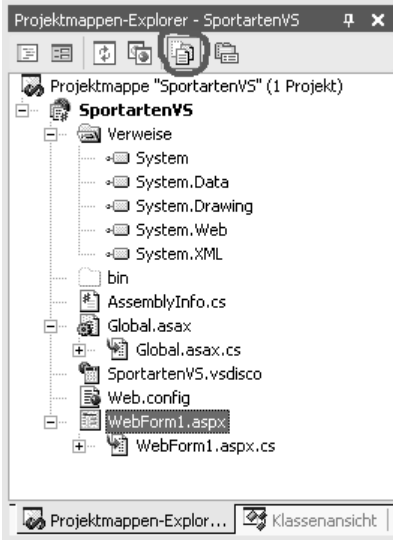


Abbildung 3.17: Projektmappen-Explorer eines Web Form-Projekts im Visual Studio .NET

Damit man nun die Web Form mit Leben füllen kann, muss man sich zunächst über das Entwicklungskonzept von Visual Studio .NET in diesem Kontext klar werden. Visual Studio trennt die Entwurfs- (oder *RAD*, *Rapid Application Development*) von der Implementierungsansicht. Bei Web-Anwendungen ist letztere sogar noch einmal aufgeteilt: Die Sicht für den HTML- bzw. Control-Tag-Quelltext und die Sicht für die Code-Behind-Implementierung. Standardmäßig wird die Entwurfsansicht dargestellt. In der linken unteren Ecke dieser Ansicht kann man in den HTML-Modus umschalten. Um zur Code-Behind-Implementierung zu gelangen, kann man entweder in der Entwurfs- oder der HTML-Ansicht den Eintrag CODE ANZEIGEN aus dem Kontextmenü auswählen (mit der rechten Maustaste in den Darstellungsbereich klicken).

Schreiten wir zum Aufbau unserer Web Form. Visual Studio stellt standardmäßig die Layout-Option *GridLayout* für den Entwurf einer Web Form ein. Mit dieser Einstellung kann man Elemente absolut auf der Seite positionieren. Während diese Option eine sehr hilfreiche Erleichterung in vielen Projekten sein wird, wollen wir hier der Einfachheit halber den Wert auf *FlowLayout* ändern. Dies entspricht der gewohnten Verhaltensweise von HTML. Diese Änderung wird auf Dokumentenebene im EIGENSCHAFTEN-Dialog vorgenommen, der entsprechende Eintrag heißt *pageLayout*. Für die

einzelnen Elemente unserer Beispielseite verwenden wir ausschließlich Web Controls. Zwar sind Web Controls für diesen Zweck teilweise etwas überdimensioniert, aber ihre Verwendung erleichtert das Verständnis für unser Beispiel ungemein. Ziehen Sie bitte ein *Label* aus der sich am linken Bildrand befindlichen TOOLBOX auf die Entwurfsansicht. Setzen Sie nun den entsprechenden Wert für *Text* (vgl. Abbildung 3.18). Anschließend betätigen Sie die Eingabetaste. Wiederholen Sie den Vorgang bitte für ein zweites Label.

Das eigentlich interessante Control in unserer Web-Anwendung ist aber die *DropDownList*. Ziehen Sie diese ebenfalls aus der Toolbox auf die Entwurfsansicht. Es erscheint der Text UNGEBUNDEN in dem Listenelement. Visual Studio .NET zeigt dadurch an, dass diese Liste noch nicht mit Werten gefüllt wurde. Wie wir in Abschnitt 3.5.4 gesehen haben, ist das Data Binding eine der großen Neuerungen in ASP.NET, und die DropDownList verwendet Data Binding. Nun haben wir zwei Möglichkeiten, die Liste zu füllen: Wir bestimmen zur Laufzeit ein Objekt welches wir über den EIGENSCHAFTEN-Dialog an das Control binden oder erzeugen dynamisch durch Code zur Laufzeit den Inhalt für die Listbox. Wir entscheiden uns hier für die zweite Variante und kopieren den hierfür nötigen Code in die *Page_Load* Methode der Code-Behind-Datei:

```
if (!Page.IsPostBack)
{
   ArrayList sportarten = new ArrayList();

   sportarten.Add ("Volleyball");
   sportarten.Add ("Fußball");
   sportarten.Add ("Biertrinken");
   sportarten.Add ("Pizzaessen");
   sportarten.Add ("Monitorweitwurf");

   DropDownList1.DataSource = sportarten;
   DropDownList1.DataBind();
}
```

Dieser Code ist bereits aus Abschnitt 3.5.4 bekannt. Er sorgt für die Bindung des ArrayList-Objekts an die DropDownList.

Zu guter Letzt müssen wir das Projekt noch kompilieren. Dies geschieht durch Auswahl des Menüpunkts ERSTELLEN – PROJEKTMAPPE ERSTELLEN. Wenn keine Fehler aufgetreten sind, können wir uns die Seite im Browser anschauen. Visual Studio besitzt einen integrierten Web-Browser, der auf dem Internet Explorer basiert. Durch Auswahl von IN BROWSER ANZEIGEN aus dem Kontextmenü (rechter Mausklick auf die Web Form-Datei, in diesem Fall *WebForm1.aspx*) gelangt man zu der gewünschten Seite.

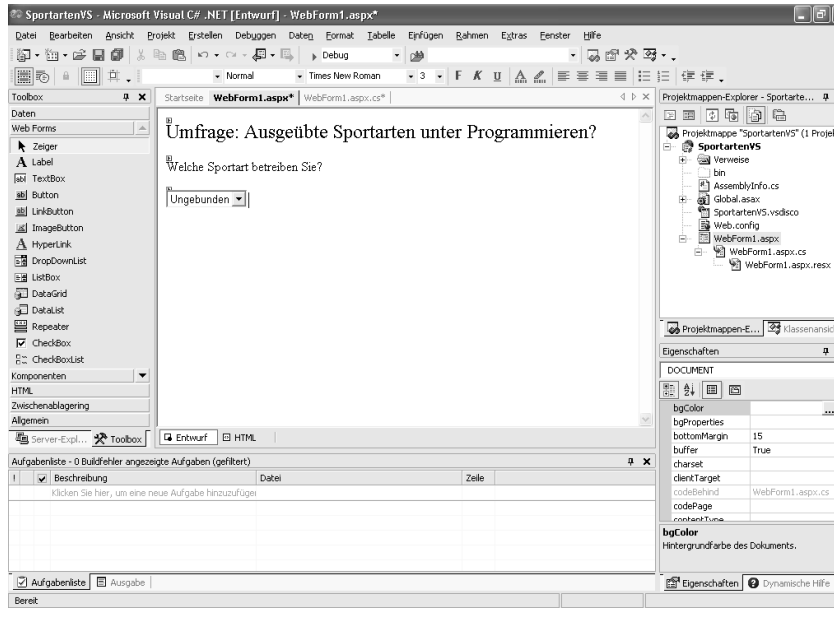


Abbildung 3.18: Entwurfsansicht der Web Form im Visual Studio .NET

Alles in allem ist die Erstellung von Web Forms-Anwendungen im Visual Studio viel einfacher und intuitiver als mit einem einfachen Texteditor. Vor allem die integrierte Hilfe und IntelliSense werden mit der Zeit zu unverzichtbaren Helfern werden. IntelliSense ermöglicht die Eingabe eines Typ- oder Klassenamens und bietet als Option die Liste aller verfügbaren Methoden, Eigenschaften und Ereignisse an. Durch diese Hilfestellung spart man viel Zeit bei der Programmierung.

3.7 Zusammenfassung

Mit ASP.NET erhält der Programmierer ein mächtiges Werkzeug an die Hand, um Web-basierte Applikation zu erstellen. ASP.NET baut zum einen auf die Stärken seiner Vorgängerversionen und reagiert zum anderen auf die Schwächen und Unzulänglichkeiten mit besseren und durchdachteren Lösungen. Durch die Einführung von Server Controls kann ein ASP.NET-Entwickler Web-Oberflächen mit der gleichen Herangehensweise entwickeln wie herkömmliche Desktop-GUI-Anwendungen. Die Verwendbarkeit und somit die Produktivität steigt merklich durch ASP.NET. ASP.NET Web-Anwendungen kann man mit einem einfachen Texteditor oder mit dem Visual Studio .NET erstellen. Letzteres ist vor allem dann vorzuziehen, wenn es sich um größere Projekte handelt.

Wir werden noch sehen, dass ASP.NET als Laufzeitumgebung nicht nur für Web-basierte grafische Oberfächen verwendet werden kann. Ein Großteil des Buchs handelt von den XML-basierten Web Services. ASP.NET ist eine mögliche Plattform für die Implementierung von Web Services.

4 Daten, Daten, Daten

ADO.NET und XML in .NET

4.1 Einführung

Nahezu jede Web-Anwendung arbeitet mit Daten. Diese Daten können unterschiedliche Ausprägungen haben. Die einen kommen aus einer SQL-Datenbank, die anderen von einem LDAP-Server, wieder andere von einem Mainframe und schließlich stammt noch eine Vielzahl von Daten aus XML-Dokumenten. Diese Vielfalt an Datenquellen und Datenformaten muss ein Programmierer in einer Anwendung unter einen Hut bringen. Musste man vor den Zeiten von Windows DNA noch viele Programmierschnittstellen zur Kommunikation mit unterschiedlichen Datenbehältern kennen, konnte man sich mit der Einführung der *Universal Data Access*-Strategie (UDA) hauptsächlich auf einige wenige konzentrieren.

UDA baut vornehmlich auf zwei Datenzugriffsschnittstellen: *OLEDB* und *ADO* (Active Data Objects). OLEDB ist die Basistechnologie, um über COM-Schnittstellen auf Datenbanken und andere Datenquellen zugreifen zu können. OLEDB arbeitet mit so genannten OLEDB Providern, die in etwa den ODBC-Treibern aus der herkömmlichen Datenbankprogrammierung entsprechen. OLEDB ist jedoch für die meisten Projekte – vor allem im Bereich der Web-Programmierung – zu komplex und zu umfangreich. Auch die Erstellung neuer OLEDB Provider ist ein sehr schwieriges und zähes Unterfangen. Zur Vereinfachung der Programmierung dient ADO als eine Abstraktionsschicht von OLEDB, um die Komplexität vor Visual Basic 6- oder ASP-Programmierern abzuschirmen. Viele Projekte arbeiten heute mit der ADO-Technologie. Vor allem das beliebte *Recordset*-Objekt wird in mehrfacher Hinsicht als die »eierlegende Wollmilchsau« verwendet.

Auch im Bereich XML gibt es für den COM-basierten Lösungsweg eine geeignete Technologie in Bezug auf den Datenzugriff: *MSXML*. MSXML ist eine COM-Komponente, die verschiedene Schnittstellen und Objekte für die Verarbeitung von XML bereitstellt. Auch MSXML ist sehr verbreitet im Projekteinsatz und hat sich in der Vergangenheit durch seine große Funktionalität bewährt. Ein Schwachpunkt von MSXML im Hinblick auf ADO ist allerdings deren singuläres Dasein. Sie sind nicht ineinander integriert. Die Integration einer Datenzugriffslösung über ADO mit einer XML-Komponente wie MSXML wäre jedoch ein ideales Gespann. So könnte man beispielsweise mit ADO bequem auf XML-Daten zugreifen und in MSXML die

XML-Repräsentation eines ADO-Objektes weiter verarbeiten. Es gibt zwar Mittel, um dies zu erreichen, diese sind aber nicht intuitiv und von Microsoft auch nicht so vorgesehen.

Mit dem Erscheinen von *ADO.NET* und den neuen XML-Klassen im .NET Framework wird eine Integration der beiden Technologien forciert, ohne sie in zu große gegenseitige Abhängigkeit zu stellen. Im ersten großen Abschnitt dieses Kapitels werde ich Ihnen die ADO.NET-Technologie und alle wichtigen Vorgehensweisen zur Arbeit mit einer Datenquelle vorstellen – der Schwerpunkt des Kapitels liegt daher auf diesem Thema. Der zweite Teil beschäftigt sich in kurzer und kompakter Weise mit den XML-Klassen in .NET, die man zur Erstellung und Bearbeitung von XML-Dokumenten benötigt.

Ich möchte ein paar Worte zur Vorgehensweise in diesem Kapitel verlieren. Es wird hauptsächlich auf die Lösung von Problemstellungen eingegangen, anstatt detailliert die diversen Technologien und Objekte und deren Einzelheiten zu beschreiben. Fragen wie »Wie verbinde ich mich zu einer Datenbank?«, »Wie rufe ich eine gespeicherte Prozedur mit Rückgabeparameter auf?« oder »Wie erzeuge ich ein XML-Dokument?« werden in der Vordergrund gestellt. Dies unterscheidet sich somit von der Vorgehensweise beispielsweise im Kapitel über ASP.NET. In diesem Kapitel erfolgte vor allem die Vorstellung der Architektur und der wichtigen Neuerungen von ASP.NET, da es sich hierbei um grundlegend neue und erklärungsbedürftige Dinge handelte. Die Datenbankanbindung und die Bearbeitung von XML-Dokumenten sind aber essentielle Bestandteile in nahezu jeder Anwendung, daher habe ich hier den oben beschriebenen Weg gewählt.

4.2 ADO.NET

Mit ADO.NET hat Microsoft nicht einfach eine neue Version des altbekannten ADO auf den Markt gebracht. Es handelt sich nicht um ADO 3.0, sondern um eine konsequente Neuentwicklung auf Basis der .NET-Plattform. Selbstverständlich wurde die Behebung von Fehlern und Problemen bei der Weiterentwicklung von ADO berücksichtigt und in ADO.NET entsprechend adressiert. An einigen Stellen werden wir dennoch sehen, dass ADO.NET in seiner Gesamtheit noch nicht so ausgereift ist, wie dies bei ADO der Fall ist. Man muss in gewissen Situationen tatsächlich noch auf das COM-basierte Vorgängermodell zurückgreifen.

Im Folgenden werde ich zunächst die grundlegende Architektur von ADO.NET aufzeigen. Der Rest dieses Abschnitts beschäftigt sich dann mit den Problemen und Vorgehensweisen zur praxisnahen Arbeit mit Datenquellen und vor allem Datenbanken.

4.2.1 Architektur

Vom allgemeinen Architekturaufbau (siehe Abbildung 4.1) unterscheiden sich ADO und ADO.NET auf den ersten Blick nur wenig. Eine Anwendung benutzt die ADO.NET-Programmierschnittstelle, um abstrahierend auf die Datenquelle zugreifen zu können. ADO.NET bedient sich so genannter *.NET Data Provider*, um den eigentlichen Zugriff z.B. auf Oracle-Datenbanken oder SQL Server zu tätigen. Einen Data Provider kann man sich als eine Art Treiber vorstellen. Dieser Treiber ist eine .NET Assembly, deren Klassen gewisse Schnittstellen implementieren müssen, damit sie als offiziell anerkannter .NET Data Provider genutzt werden können. Der Data Provider kapselt für ADO.NET – und somit auch für den Entwickler – den Zugriff auf die Datenquelle. Dieser Zugriff erfolgt üblicherweise mit dem nativen Protokoll der Datenquelle. In der Version 1.0 des .NET Frameworks werden zwei Data Provider mitgeliefert: der *SQL Server Data Provider* und der *OLEDB Data Provider* (siehe Tabelle 4.1).

Data Provider	Verwendung
SQL Server *System.Data.SqlClient*	Nativer Zugriff ausschließlich auf Microsoft SQL Server (ab Version 7.0).
OLEDB *System.DataOledb*	Zugriff auf OLEDB-Datenquellen. Verwendet vorhandene OLEDB Provider.

Tab. 4.1: Verfügbare Data Provider in ADO.NET

Wer ausschließlich den SQL Server als Datenbankmanagementsystem verwendet, wird den SQL Server Data Provider einsetzen. Dieser implementiert native Zugriffe auf den SQL Server in dessen eigenem Protokoll. Somit sind die Zugriffe sehr schnell und können speziell auf den SQL Server optimiert werden. Es sind keine zusätzlichen Schichten wie ODBC oder OLEDB im Spiel. Wer hingegen mit dem SQL Server 6.5 bzw. einer Datenbank von Oracle oder IBM arbeitet, muss auf den OLEDB Data Provider zurückgreifen. Dieser Provider bedient sich bereits existierender OLEDB Provider, um den Zugriff auf die Datenbank zu vollziehen. Somit dient der Data Provider als Schale (*wrapper*) für eine bereits vorhandene Datenzugriffstechnologie. Sobald sich .NET und ADO.NET etablieren, ist anzunehmen, dass immer mehr Datenbankanbieter oder Dritthersteller native Data Provider für nicht-SQL Server-Produkte anbieten.

In der aktuellen Version von ADO.NET können keine ODBC-Treiber verwendet werden. Der OLEDB Data Provider arbeitet zwar mit vorhandenen OLEDB Providern, aber nicht mit dem MSDASQL Provider. MSDASQL ermöglicht in einer nicht-.NET-Umgebung den Zugriff über OLEDB auf ODBC-basierte Datenbanken (z.B. *mySql*).

Um den Zugriff auf ODBC-Datenbanken zu erhalten, muss man den *ODBC .NET Data Provider* installieren. Dieser Data Provider implementiert intern das ODBC-API, über das die ODBC-Treiber angesprochen werden. Er kann über den Microsoft MSDN Server bezogen werden: `http://msdn.microsoft.com/downloads/default.asp?URL=/downloads/sample.asp?url=/MSDN-FILES/027/001/668/msdncompositedoc.xml`

Wer auf der Basis von XML-Daten mit dem SQL Server kommunizieren möchte, benötigt ebenfalls einen gesonderten Data Provider. Er heißt *SQLXML* und befindet sich auf dem MSDN-Server: `http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28001300`

Kurz vor Fertigstellung des Buchs hat Microsoft noch die Beta-Version eines Data Providers zur nativen Unterstützung von Oracle-Datenbanken veröffentlichet: .NET Data Provider for Oracle Beta 1. Dieser Provider stellt somit das Gegenstück zum SQL Server Data Provider dar. Er ist unter folgender Adresse zu finden: `http://microsoft.com/downloads/release.asp?ReleaseID=37805`

Die beiden mitgelieferten Data Provider implementieren jeweils unterschiedliche Klassen im Namensraum *System.Data*. Diesen Namensraum werden wir im Folgenden unter die Lupe nehmen.

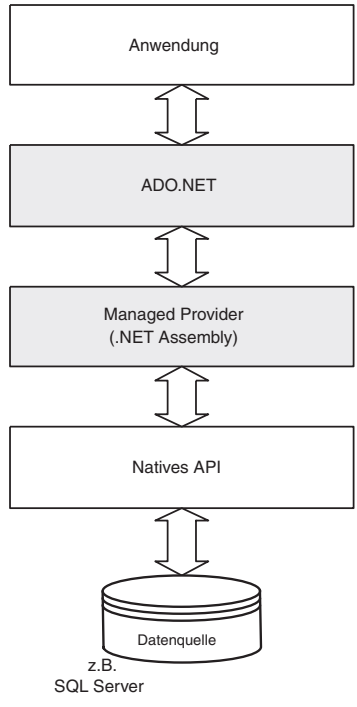


Abbildung 4.1: ADO.NET in der UDA-Strategie

4.2.2 Der Namensraum System.Data

Die Klassen und Typen der .NET Data Provider für den Zugriff auf Datenquellen über ADO.NET befinden sich alle im Namensraum *System.Data*. In diesem Abschnitt werde ich Ihnen einen Überblick über diesen Namensraum und die wichtigsten Klassen geben. Einige der Klassen werden dann noch im Verlauf des Kapitels genauer beleuchtet und mit Beispiel-Code illustriert.

Ich werde mich hier auf die Abhandlung der beiden wichtigsten Data Provider für den SQL Server und für OLEDB beschränken.

Außer den hier vorgestellten Klassen gibt es noch wesentlich mehr Klassen in diesem Namensraum. Einige von ihnen werden dann im Abschnitt über das ADO.NET *DataSet* abgehandelt (siehe Abschnitt 4.2.10). Das *DataSet* ist keinem der beiden folgenden Namensräume zugeordnet und bietet einzigartige Funktionalitäten für getrennte (*disconnected*) Szenarien in einer verteilten Anwendung.

System.Data.SqlClient

Wer bereits mit ADO gearbeitet und sich ein wenig in .NET eingearbeitet hat, wird mit den Klassen und Objekten des *SqlClient*- und auch des *OleDb*-Namensraums keine großen Schwierigkeiten haben. Einige der Klassen haben direkte Entsprechungen im ADO-Modell, wohingegen einige Klassen neue Konzepte verkörpern und realisieren. Die wichtigsten Klassen sind in Tabelle 4.2 aufgeführt. Sie werden in diesem Abschnitt kurz erläutert. Die folgenden Abschnitte zeigen die Klassen dann im Einsatz bei der Verwendung in einer Web-Anwendung.

Klasse	Beschreibung
SqlConnection	Repräsentiert eine Verbindung zu einem SQL Server.
SqlCommand	Repräsentiert eine Transact-SQL-Anweisung oder eine gespeicherte Prozedur (*stored procedure*), die auf dem SQL Server ausgeführt werden sollen. Die genaue Implementierung hängt von der eingesetzten Datenquelle ab.
SqlDataAdapter	Repräsentiert eine Menge von Kommandos und eine Verbindung für die Verwendung mit einem *DataSet*-Objekt (siehe Abschnitt 4.2.9).
SqlDataReader	Ermöglicht *Stream*-basiertes Lesen (nur nach vorn) von Daten aus einem SQL Server.
SqlError	Enthält Warnungen oder Fehler, die vom SQL Server zurückgegeben werden.
SqlParameter	Repräsentiert einen Parameter für ein *SqlCommand*-Objekt.
SqlTransaction	Repräsentiert eine Transact-SQL-Transaktion innerhalb des SQL Server.

Tab. 4.2: Wichtige Klassen im Namensraum System.Data.SqlClient

SqlConnection. Diese Klasse repräsentiert eine gültige und offene Verbindung zu einem SQL Server. Wenn sich dieser SQL Server im Netzwerk auf einem anderen Computer befindet, entspricht eine *SqlConnection* einer Netzwerkverbindung zu diesem Datenbank-Server. Der Verbindungsaufbau wird über so genannte *Connection Strings* (Verbindungszeichenketten) gesteuert. Mehr zu diesem Thema können Sie in Abschnitt 4.2.3 nachlesen.

SqlCommand. Die Klasse *SqlCommand* steht für eine *Transact-SQL*-Anweisung (*T-SQL*) oder eine gespeicherte Prozedur (*stored procedure*). T-SQL ist ein spezifischer SQL-Dialekt des SQL Servers und erlaubt weitreichende Manipulationen auf relationalen Datenbeständen. *SqlCommand* ermöglicht die Ausführung der Kommandos in unterschiedlichen Kontexten. Je nach Auswahl der Methode zum Ausführen des Kommandos wird ein einfacher Ergebniswert, eine Ergebnismenge oder aber kein Ergebnis zurückgegeben. Die genauen Abläufe mitsamt Beispiel-Code können Sie in den Abschnitten 4.2.4, 4.2.5 und 4.2.6 nachlesen.

SqlDataAdapter. Mit der Klasse *SqlDataAdapter* kann der Entwickler eine Menge von Datenkommandos zusammenfassen. Gemeinsam mit einer gültigen Verbindung zu einem SQL Server wird über den *DataAdapter* ein *DataSet* gefüllt. Über die Kommandos des *DataAdapters* wird das Verhalten des *DataSets* beim Erneuern, Einfügen oder Löschen von Daten gesteuert. Das *DataSet* und seine Funktionalität wird genauer in Abschnitt 4.2.10 beleuchtet.

SqlDataReader. Eine Instanz von *SqlDataReader* stellt ein Mittel zum Lesen einer Ergebnismenge dar. In einem *DataReader* werden die Reihen der Ergebnisrelation innerhalb eines *Streams* abgebildet. Der *Stream* kann nur in einer Richtung von Anfang bis zum Ende des Datenstroms durchlaufen werden. Mit einem solchen Modell wird eine höchstmögliche Geschwindigkeit gewährleistet. Dieses Objekt sollte verwendet werden, wenn man Daten mit hoher Geschwindigkeit aus einer Datenbank auslesen muss und diese gleich weiter verarbeitet. Ein *SqlDataReader*-Objekt kann nur über den Aufruf der entsprechenden Methode *ExecuteReader* auf einem *SqlCommand*-Objekt erzeugt werden.

SqlError. Tritt ein Fehler in der Abarbeitung innerhalb eines SQL Servers auf, so erzeugt der SQL Server .NET Data Provider eine Instanz dieses Typs. Ein *SqlError* wird in einem Programm ausgelesen, nachdem eine Ausnahme vom Typ *SqlExcpetion* aufgetreten ist.

SqlParameter. Ein *SqlDataCommand*-Objekt kann eine einfache SQL-Anweisung oder aber eine komplizierte SQL-Abfrage bzw. eine gespeicherte Prozedur mit mehreren Parametern repräsentieren. Die Klasse *SqlParameter* dient als Stellvertreter für Parameter in einem *SqlDataCommand*-Objekt. Mehr Informationen hierüber finden Sie in Abschnitt 4.2.7.

SqlTransaction. Über eine Instanz der Klasse *SqlTransaction* wird eine SQL-Server-interne Transaktion innerhalb von ADO.NET gesteuert. Diese Instanz wird erzeugt, indem der Programmierer die Methode *BeginTransaction* des

SqlConnection-Objekts aufruft. Über dieses neu erzeugte Objekt wird dann die Transaktion gesteuert. Ein Beispiel für die Verwendung von Transaktionen finden Sie in Abschnitt 4.2.9.

System.Data.Oledb

Die Klassen im Namensraum *Oledb* sind ähnlich ausgelegt wie jene im Namensraum *SqlClient*. Aus diesem Grund wird hier auf eine eingehende Besprechung verzichtet. Die relevanten Klassen können Sie in Tabelle 4.3 einsehen.

Klasse	Beschreibung
OleDbConnection	Repräsentiert eine Verbindung zu einer OLEDB-Datenquelle.
OleDbCommand	Repräsentiert eine SQL-Anweisung oder eine gespeicherte Prozedur, die auf einer Datenquelle ausgeführt werden sollen.
OleDbDataAdapter	Repräsentiert eine Menge von Kommandos und eine Verbindung für die Verwendung mit einem *DataSet*-Objekt (siehe Abschnitt 4.2.9).
OleDbDataReader	Ermöglicht *Stream*-basiertes Lesen (nur von vorne nach hinten) von Daten aus einer Datenquelle.
OleDbError	Enthält Warnungen oder Fehler, die von der Datenquelle zurückgegeben werden.
OleDbParameter	Repräsentiert einen Parameter für ein *OleDbCommand*-Objekt.
OleDbTransaction	Repräsentiert eine SQL-basierte Transaktion innerhalb einer Datenquelle. Die genaue Implementierung hängt von der eingesetzten Datenquelle ab.

Tab. 4.3: Wichtige Klassen im Namensraum System.Data.Oledb

Sämtliche Beispiele über ADO.NET werden für den Namensraum *System.Data.SqlClient* und somit für den SQL Server erstellt. Wo es aber interessant ist zu sehen, wie der Einsatz einer bestimmten Technik beispielsweise bei einer Access-Datenbank aussieht, wird das Beispiel auch für den *System.Data.Oledb* Namensraum vorgestellt.

Ansonsten kann man fast alle Beispiele relativ einfach von *SqlClient* nach *Oledb* portieren: Dies erfordert nur das Ersetzen des Namensraums und der Zeichenkette *SqlClient* durch *Oledb*. Bis auf einige Ausnahmen ist dies die notwendige Vorgehensweise.

4.2.3 Verbindung herstellen

Der erste Beispiel-Code in diesem Kapitel dient dem Öffnen einer Verbindung zu einer SQL Server-Datenbank. Der Quelltext aus Listing 4.1 ist Ihnen bereits aus Kapitel 3 über ASP.NET bekannt (hier in der C#-Version). Diese zwei Zeilen und der Einsatz der Klasse *SqlConnection* genügen, um sich mit der *WS_Buch*-Datenbank auf dem lokalen SQL Server zu verbinden.

```
SqlConnection oConn = new SqlConnection
("server=localhost;uid=sa;database=WS_Buch");
oConn.Open();
// Abarbeiten der gewünschten Aktionen auf der DB
// ...
```

Listing 4.1: Öffnen einer Datenbankverbindung über SqlConnection

Das einzige Argument für den Konstruktor von *SqlConnection* ist eine Zeichenkette. Diese Zeichenkette setzt sich aus mehreren Segmenten zusammen und wird als Connection String bezeichnet. Eine überladene Version des Konstruktors nimmt keine Argumente entgegen. Hier muss dann der Connection String später über eine Eigenschaft gesetzt werden. Damit die Verbindung tatsächlich geöffnet wird, muss der Programmierer noch die Methode *Open* aufrufen – erst dann wird die physikalische Verbindung aufgebaut.

Man könnte hier annehmen, dass die Verbindung zum SQL Server geschlossen wird, nachdem das *SqlConnection*-Objekt durch den Garbage Collector zerstört wird. Aber dies ist nicht der Fall! Die Verbindung muss immer explizit durch Aufruf der Methode *Close* geschlossen werden. Wird dies nicht getan, so bleiben alle vorhandenen Verbindungen geöffnet. Hat man beispielsweise die Anzahl der möglichen Verbindungen im SQL Server auf eine bestimmte Zahl limitiert, dann tritt die Abweisung einer Verbindungsabfrage wesentlich früher ein als gewollt.

Connection Strings. Das Konzept der Connection Strings gibt es auch unter ADO und OLEDB. Ein Connection String ist eine Konfigurationszeichenkette, welche dem zugrunde liegenden Provider mitteilt, mit welchen Parametern die Verbindung zu einer gewünschten Datenbank aufgebaut werden soll. In einem Connection String können mehrere Eigenschaften gesetzt werden, z.B. für den SQL Server. Die wichtigsten Eigenschaften sind in Tabelle 4.4 aufgelistet. Innerhalb des Connection Strings werden die jeweiligen Eigenschaften aus der Tabelle durch ein Semikolon getrennt.

Name	Standardwert	Beschreibung
Connection Timeout oder *Connect Timeout*	15	Zeitintervall, in dem versucht wird, die Verbindung zum SQL Server herzustellen. Nach Überschreiten des Wertes wird ein Fehler erzeugt.
Data Source oder *Server* oder *Address* oder *Addr* oder *Network Address*		Name oder Adresse des SQL Server, zu dem eine Verbindung aufgebaut werden soll.
Initial Catalog oder *Database*		Name der Datenbankinstanz.
Integrated Security oder *Trusted Connection*	false	Legt fest, ob eine sichere Verbindung zustande kommen soll. Gültige Werte sind *true*, *false* und *sspi* (entspricht *true*).
Password oder *Pwd*		Passwort des SQL Server Logins.
User ID oder *UID*		Benutzername des SQL Server Logins.

Tab. 4.4: Wichtige Eigenschaften für SQL Server Connection String

Die am meisten verwendeten Eigenschaften sind wohl *Server, UID, PWD* und *Database*. Alternativ wird zu *UID* und *PWD* oft die Eigenschaft *Trusted Connection* auf *true* gesetzt. Dann authentifiziert der SQL Server .NET Data Provider den aktuellen Benutzer beim SQL Server. Dies erhöht die Sicherheit, da keine expliziten Benutzerdaten über die Verbindung ausgetauscht werden.

Bei der Verwendung des OLEDB .NET Data Providers gibt es einen wichtigen Unterschied im Vergleich zum SQL Server Data Provider. Da der OLEDB Data Provider mit mehreren OLEDB Providern arbeiten kann, muss dessen Angabe im entsprechenden Connection String erfolgen. Folgende Connection Strings sind denkbar:

```
Provider=MSDAORA;Data Source=MyOracle8i;
   User ID=TestAccount;Password=TestPwd;
Provider=Microsoft.Jet.OLEDB.4.0;Data Source=
   C:\WebApps\db\WS_Buch.MDB;
```

Der Wert für `Provider` muss immer mit angegeben werden, damit der Data Provider weiß, welchen nativen OLEDB Provider er ansprechen muss. Das erste Beispiel öffnet eine Verbindung zu einer Oracle 8i-Datenbank über den Oracle OLEDB Provider von Microsoft. Das zweite Beispiel hingegen öffnet die Verbindung zu einer Access-Datenbank. Dies wird durch den Einsatz des OLEDB Jet-Treibers bewerkstelligt. Vor allem die Verwendung von Access in kleinen und mittleren Web-Projekten macht diesen Connection String sehr wertvoll.

Speichern Sie Ihre Connection Strings nicht wie unter ASP üblich in der Datei *global.asa*. Mit ASP.NET hat ein Konzept zur Konfiguration von .NET- und somit auch von ASP.NET- Anwendungen Einzug gehalten, das auf XML-Dateien basiert. So ist die *web.config* als Konfigurationsdatei einer ASP.NET-Anwendung der richtige Platz für die Verwaltung von Connection Strings. Ein exemplarischer Abschnitt in der *web.config* könnte folgendermaßen aussehen:

```
<configuration>
   <appSettings>
      <add key="connectionString"
      value="Provider=Microsoft.Jet.OLEDB.4.0;
      Data Source=C:\WebApps\db\WS_Buch.MDB" />
   </appSettings>
<system.web>
...
```

Im Abschnitt `appSettings` kann der Programmierer wichtige Konstanten und Werte ablegen, die dann zur Laufzeit im Code aus der *web.config*-Datei ausgelesen werden. So kann man den Wert für einen aktuellen Connection String mit diesem Aufruf auslesen:

```
string strConn =
ConfigurationSettings.AppSettings["connectionString"];
```

Connection Pooling. Für Web-Anwendungen mit vielen Benutzern und einer großen Last bei der Übertragung von Daten sind besondere Maßnahmen notwendig, wenn es um die Integration von Datenbanken geht. Hierbei hat sich das Prinzip des *Connection Poolings* für eine Verbesserung der Skalierbarkeit in den letzten Jahren bewährt. Beim Connection Pooling wird ein Vorrat an Datenbankverbindungen angelegt. Aus diesem Vorrat versorgt der .NET Data Provider die eingehenden Verbindungsabfragen mit bereits instanziierten Verbindungen. Die Identifizierung einer Verbindung erfolgt über die bereitgestellten Parameter im Connection String. Ist eine Verbindung mit den exakt gleichen Eigenschaftsparametern bereits im Pool, so wird diese aus dem Pool ausgelöst und verwendet. Nach Schließen der Verbindung wird sie wieder in den Vorrat zurückgestellt. Sie wird allerdings nur dann geschlossen, wenn die Methode *Close* explizit aufgerufen wird.

Connection Pooling ist in beiden .NET Data Providern standardmäßig aktiviert. Man kann das zugehörige Verhalten jedoch über das Setzen von neuen Eigenschaften im Connection String beeinflussen. Die Implementierung des OLEDB .NET Data Providers nutzt dabei den Connection Pooling-Mechanismus von OLEDB. Daher werden auch die exakt gleichen Parameter im Connection String verwendet, um beispielsweise das Pooling mitsamt Transaktionsverwaltung zu deaktivieren:

```
Provider=SQLOLEDB;OLE DB Services=-4; ...
```

Der SQL Server .NET Data Provider implementiert das Connection Pooling mithilfe von COM+. Der Data Provider sorgt dafür, dass das Connection Pooling über das von COM+ zur Verfügung gestellte Objekt-Pooling erledigt wird. Durch die Verwendung von neuen Parametern im Connection String lässt sich auch hier das Verhalten des Connection Poolings beeinflussen. Die wichtigsten Parameter hierfür können Sie in Tabelle 4.5 nachlesen. So lässt sich mit der folgenden Zeile die minimale Anzahl des Pools auf 5 und die maximale Zahl auf 50 einstellen:

```
Server=localhost;Database=WS_Buch; Min Pool Size=5;Max Pool Size=50;
```

Name	Standardwert	Beschreibung
Min Pool Size	0	Minimale Größe des Pools für Datenbankverbindungen.
Max Pool Size	100	Maximale Größe des Pools für Datenbankverbindungen.
Pooling	True	Zeigt an, ob Connection Pooling aktiviert ist.

Tab. 4.5: Wichtige Eigenschaften zur Steuerung des Connection Poolings

4.2.4 Lesen von Daten

Die wohl wichtigste und am häufigsten verwendete Datenbankaktion ist das Auslesen von Datensätzen aus einem bestehenden Datenvorrat. Diese Daten sind zumeist textuelle Informationen aus einer oder mehreren Tabellen, die im Programm angezeigt oder weiter verarbeitet werden. Zunächst muss eine Verbindung zur Datenquelle hergestellt werden. Dieser Vorgang wurde im vorhergehenden Abschnitt genau erläutert. Wenn eine Verbindung existiert, wird über ein spezielles Objekt ein Kommando an die Datenquelle geschickt. Das Ergebnis des Kommandos wird dann über ein weiteres Objekt der aufrufenden Anwendung zur Verfügung gestellt. Die beiden verantwortlichen Objekte in ADO.NET heißen *Command* und *DataReader* (um genau zu sein *SqlCommand/SqlDataReader* und *OledbCommand/OledbDataReader*). Es gibt zudem noch das *DataSet* zum Auslesen von Daten und zum Weiterverarbeiten von Kommandos. Das *DataSet* wird in Abschnitt 4.2.10 genauer beleuchtet.

Command

Das *SqlCommand*-Objekt repräsentiert ein Kommando, welches gegen eine SQL Server-Datenbank abgesetzt werden soll. Über dieses Objekt werden die SQL Server-eigenen T-SQL-Anweisungen formuliert, unter Umständen mit Parametern versehen und über den Aufruf einer entsprechenden Methode ausgeführt (siehe weiter unten in diesem Abschnitt). Folgendes Beispiel repräsentiert eine SQL-Abfrage, um alle Einträge in der Tabelle `Sportarten` auszulesen:

```
SqlConnection oConn = new SqlConnection(
   "server=localhost;uid=BuchUser;pwd=ws_buch;
   database=WS_Buch");
SqlCommand oCmd = new SqlCommand
   ("SELECT * FROM Sportarten", oConn);
oConn.Open();
```

Es bleibt hier zu erwähnen, dass mit obigem Beispiel das Kommando noch nicht ausgeführt, sondern lediglich vorbereitet wurde. Durch die Verwendung des Konstruktors wurde der Wert für die T-SQL-Anweisung bereits bei der Objekterzeugung gesetzt. Der Wert kann aber auch durch die Eigenschaft *CommandText* ausgelesen bzw. gesetzt werden. Dies gilt aber nur für reine SQL-Anweisungen. Abfragen über gespeicherte Prozeduren müssen unterschiedlich gehandhabt werden. Eine genaue Beschreibung des Vorgangs wird in Abschnitt 4.2.8 vorgestellt. Eng mit dieser Unterscheidung verbunden ist die Eigenschaft *CommandType*. Sie ist im Großen und Ganzen für die eindeutige Identifizierung des Typs des auszuführenden Kommandos verantwortlich. So erhält man mit dem folgenden kleinen Beispiel den aktuellen Wert der Eigenschaft *CommandType*. In unserem Fall wird dies der Wert *Text* sein, denn der Konstruktor hat erkannt, dass wir das betreffende *Command*-Objekt mit einer T-SQL-Anweisung initialisiert haben:

```
Response.Write oCmd.CommandType;
```

Das *Command*-Objekt besitzt eine Reihe von Methoden und Eigenschaften, von denen wir im Laufe des Kapitels noch einige kennenlernen werden.

Was uns jetzt noch fehlt, um wirklich mit den in der Datenbank schlummernden Informationen arbeiten zu können, ist die tatsächliche Ausführung des Kommandos und das Entgegennehmen des Ergebnisses. Die Ausführung wird über eine von vier Methoden initiiert. Diese sind in Tabelle 4.6 zu sehen.

Tab. 4.6: Methoden der SqlCommand-Klasse zum Ausführen eines Kommandos

Methode	Beschreibung
ExecuteNonQuery	Liefert keine Ergebnisreihen zurück. Wird für `INSERT`-, `UPDATE`-, `DELETE`-Kommandos verwendet
ExecuteScalar	Liefert einen einzigen Wert zurück
ExecuteReader	Liefert ein *SqlDataReader*-Objekt zurück. In diesem Objekt stehen die Ergebnisreihen der Abfrage.
ExecuteXmlReader	Liefert ein *XmlReader*-Objekt zurück. Speziell für XML-basierte Datenbankabfragen

Die Methode *ExecuteReader* und das *SqlDataReader*-Objekt werden noch im folgenden Abschnitt besprochen. Möchte man beispielsweise nur einen aggregierten Wert aus einer Datenbankabfrage weiter verarbeiten (z.B. die Anzahl der Ergebnisse der Abfrage), dann bietet sich der Einsatz von *ExecuteScalar* an. Folgendes Beispiel illustriert den Vorgang:

```
SqlCommand oCmd = new SqlCommand
   „SELECT COUNT(*) AS Summe FROM Sportarten“, oConn);
oConn.Open();
int count = (int) oCmd.ExecuteScalar();
```

Andererseits gibt es durchaus Situationen, in denen kein expliziter Wert zurückgeliefert werden muss. Dies ist vor allem bei Manipulationskommandos der Fall. Diese Art von Kommandos werden weiter unten in den entsprechenden Abschnitten über das Einfügen und Ändern von Daten erklärt.

DataReader

ASP-Programmierer kennen das ADO *Recordset* zum Halten und Verwalten von Datenbeständen. Das *Recordset* wird auch benutzt, um Daten aus einem Datenbehälter zu holen, sie auszulesen und dann wieder zu verwerfen. Ein ADO *Recordset* ist ein komplexes Objekt mit einer internen Speicherstruktur, die u.a. auch für ein von der Datenbank losgelöstes Szenario ausgelegt ist (d.h. dass das *Recordset* als generischer Datenbehälter verwendet und zwischen Methoden und sogar Anwendungen ausgetauscht werden kann, ohne eine Datenbankverbindung offen halten zu müssen). Eine solche Methode bedeutet einen relativ großen Aufwand für Daten, die nur einmal gelesen werden müssen und danach nicht mehr benötigt werden. Das *DataReader*-Objekt in ADO.NET verfolgt hier einen anderen Ansatz. *SqlDataReader* (für den SQL Server) ist ein *Stream*-orientiertes Objekt, um Daten aus einer Datenbankabfrage in eine Richtung lesen zu können. Dies bedeutet, dass man das *SqlDataReader*-Objekt auslesen kann, aber nicht mehr zurück an den Anfang springen kann. Diese Einschränkung dient einer besseren Speicherverwaltung und einem optimierten Geschwindigkeitsverhalten. Ein *DataReader* wird ausschließlich durch den Aufruf von *ExecuteReader* auf einem *Command*-Objekt erzeugt – und nicht durch den Aufruf eines Konstruktors. Wenn ein *DataReader* in Gebrauch ist, wird die zugrunde liegende Datenbankverbindung offen gehalten und ausschließlich für jenen *DataReader* reserviert. Dies bedeutet, dass mit diesem *Connection*-Objekt keine anderen Aktionen durchgeführt werden können, bis die *Close*-Methode auf dem *DataReader* aufgerufen wird.

Das Beispiel in Listing 4.2 zeigt den relevanten Teil einer ASP.NET-Seite für das Auslesen von Daten aus einer Tabelle und die Ausgabe in die ASP.NET-Seite. Im Listing wurde auch der Code zur Fehlerbehandlung zugunsten der besseren Übersichtlichkeit weggelassen. Das komplette Listing ist auf der Begleit-CD zu finden.

```
SqlDataReader oDR = null;
SqlConnection oConn = new SqlConnection
   ("server=localhost;uid=BuchUser;pwd=ws_buch;
   database=WS_Buch");
SqlCommand oCmd = new SqlCommand
   ("SELECT * FROM Benutzer", oConn);
oConn.Open();
oDR = oCmd.ExecuteReader
   (CommandBehavior.CloseConnection);
while (oDR.Read())
{
   Response.Write(oDR.GetInt32(0) +
      "   ");
   Response.Write(oDR.GetString(3) + ", " +
      oDR.GetString(2) + "   ");
   Response.Write(oDR.GetString(1) +
      "   ");

   if (oDR.IsDBNull(5))
      Response.Write("Nicht verfügbar<br>");
   else
      Response.Write
         (oDR.GetDateTime(5).ToLongDateString() +
            "<br>");
}

oDR.Close();
//oConn.Close();
```

Listing 4.2: Auslesen von Daten aus einer SQL Server-Datenbank mit dem SqlDataReader

Der Code in Listing 4.2 müsste in groben Zügen bereits vertraut wirken. Das *DataReader*-Objekt wird mit dem Parameter *CommandBehavior.CloseConnection* erzeugt. Er zeigt dem *DataReader* an, dass die zugrunde liegende Verbindung zur Datenbank geschlossen werden soll, wenn das *DataReader*-Objekt geschlossen wird. Dieser Parameter verhindert somit, dass durch ein fehlendes `oConn.Close()` Datenbankverbindungen unnötig geöffnet bleiben. Bevor man auf den *DataReader* zugreift, muss immer erst die Methode *Read* aufgerufen werden, damit der *DataReader* an der richtigen Stelle mit dem Lesen beginnt. Anschließend erfolgt das eigentliche Auslesen der Daten aus dem *DataReader*.

Für das Auslesen der Werte aus dem *DataReader* steht eine Reihe von Methoden zur Verfügung. Prinzipiell besitzt jeder unterstützte Datentyp mindestens eine Methode zum Auslesen. In Tabelle 4.7 werden lediglich die wichtigsten und am häufigsten verwendeten *Get*-Methoden aufgelistet.

Methode	Beschreibung
GetBoolean	Liefert einen Boolean-Wert (wahr/falsch) zurück.
GetBytes	Liefert ein Array von Bytes zurück (siehe Abschnitt 4.2.11).
GetDateTime	Liefert eine *DateTime*-Struktur zurück.
GetInt[16, 32, 64]	Liefert einen 16, 32 oder 64 Bit großen *Integer*-Wert zurück.
IsDBNull	Prüft das Feld auf einen NULL-Wert.
GetString	Liefert einen String zurück.

Tab. 4.7: Wichtigste Methoden zum Auslesen von Werten aus einem DataReader

Die vorgestellten Schritte sind die wichtigsten Vorgehensweisen beim Auslesen von vorwiegend textuellen Daten aus einer Datenquelle, wenn diese Daten nicht im Speicher gehalten werden müssen, sondern direkt weiter verarbeitet werden. Der Spezialfall der Behandlung von Binärdaten wird in Abschnitt 4.2.11 näher erläutert.

4.2.5 Einfügen von Daten

Bevor man aus einer Datenquelle Daten auslesen kann, muss man diese selbstverständlich mit Informationen füllen. Auch diese Aktion geschieht in ADO.NET über das *Command*-Objekt. Nur benötigen wir in diesem Fall keinen *DataReader*, da wir keine Daten aus der Datenbank abholen wollen. Das Beispiel in Listing 4.3 zeigt eine Einfügeoperation in die Tabelle `Benutzer`. Mit diesem einfachen SQL-Befehl lassen sich Daten über neue Benutzer in diese Tabelle einfügen.

```
SqlConnection oConn = new SqlConnection
   ("server=localhost;uid=BuchUser;pwd=ws_buch;
      database=WS_Buch");
string sSQL =
   "INSERT INTO Benutzer Values ('lmatt', 'Lothar',
   'Matthäus', 'testpw','" + DateTime.Now + "',NULL)";
SqlCommand oCmd = new SqlCommand(sSQL, oConn);

oConn.Open();
int iRes = oCmd.ExecuteNonQuery();
oConn.Close();
```

Listing 4.3: Einfügen von Daten in eine Datenbank

Im Beispiel wird der *Integer*-Variablen das Ergebnis von *ExecuteNonQuery* zugewiesen. Diese Zahl zeigt an, wie viele Datensätze von der Operation betroffen waren. Im Falle einer Einfügeoperation ist es die Anzahl der Einfü-

geoperationen, die in der SQL-Anweisung vorgegeben werden. Eine komfortablere Art des Einfügens von Daten ist die Verwendung von Parametern. Diese Vorgehensweise wird weiter unten in Abschnitt 4.2.7 besprochen.

Konnte man unter ADO mithilfe von *ADOX* Datenbanken und Tabellen mittels Programmcode erstellen, so existiert diese integrierte Lösung in ADO.NET nicht mehr. Bestimmte Aktionen (wie das Anlegen einer Tabelle im SQL Server) kann man über spezielle SQL/T-SQL-Anweisungen abwickeln. Es gibt aber keine Programmierschnittstelle, die unabhängig vom Datenbehälter agieren kann. In diesem Fall muss der Programmierer auf ADOX zugreifen, indem er die COM-Interoperabilitäts-Unterstützung in .NET verwendet.

4.2.6 Aktualisieren von Daten

Ähnlich wie das Einfügen verhält sich auch das Aktualisieren von Daten. In ADO.NET greift man auch hier auf das *Command*-Objekt zurück. Folgendes Beispiel (Listing 4.4) aktualisiert das Kennwort für den Benutzer `cweyer` in der Benutzertabelle. Einziger Unterschied zu Listing 4.3 ist das T-SQL-Kommando welches über das *Command*-Objekt ausgeführt wird.

```
SqlConnection oConn = new SqlConnection
   ("server=localhost;uid=BuchUser;pwd=ws_buch;
      database=WS_Buch");
string sSQL =
   "UPDATE Benutzer SET Passwort = 'NEUESpw'
   WHERE Benutzername = 'cweyer'";
SqlCommand oCmd = new SqlCommand(sSQL, oConn);

oConn.Open();
int iRes = oCmd.ExecuteNonQuery();
oConn.Close();
```

Listing 4.4: Aktualisieren von Daten in einer Datenbank

Die Aktionen für Einfügen und Aktualisieren unterscheiden sich bei dieser Vorgehensweise also nur durch das verwendete SQL-Kommando.

4.2.7 Arbeiten mit Parametern

Bei der Arbeit mit langen und teilweise unübersichtlichen SQL-Kommandos kann es schnell passieren, dass man nicht mehr genau weiß, welcher Parameter schon behandelt wurde oder ob das abschließende Hochkomma für eine Zeichenkette schon geschrieben wurde oder nicht. Abhilfe schafft in diesem Fall die Verwendung von *Parameter*-Objekten. Mit Parametern kann man bequem seine SQL-Anweisung zusammensetzen und auch leicht die Werte der einzelnen Parameter kontrollieren.

In Listing 4.5 sehen Sie das bereits bekannte Code-Beispiel, diesmal jedoch mit Verwendung von expliziten Parametern für das *Command*-Objekt.

```
SqlConnection oConn = new SqlConnection
   ("server=localhost;uid=BuchUser;pwd=ws_buch;
   database=WS_Buch");
string sSQL =
   "INSERT INTO Benutzer Values
   (@BenName, @Vorname, @Nachname, @Passwort, @Datum, @Bild)";
SqlCommand oCmd = new SqlCommand(sSQL, oConn);
string sBenName = "lmatt";
string sVorname = "Lothar";
string sNachname = "Matthäus";
string sPasswort = "testpw";
DateTime dtJetzt = DateTime.Now;

oCmd.Parameters.Add("@BenName", sBenName);
oCmd.Parameters.Add("@Vorname", sVorname);
oCmd.Parameters.Add("@Nachname", sNachname);
oCmd.Parameters.Add("@Passwort", sPasswort);
oCmd.Parameters.Add("@Datum", dtJetzt);
oCmd.Parameters.Add("@Bild", newSystem.Data.SqlTypes.SqlBinary());

oConn.Open();
int iRes = oCmd.ExecuteNonQuery();
```

Listing 4.5: Verwendung von Parametern in Datenbankzugriffen

Das *SqlCommand*-Objekt besitzt eine Kollektion von *Parameter*-Objekten. Diese Kollektion kann man entweder wie in Listing 4.5 direkt über die *Parameters*-Eigenschaft des *Command*-Objekts ansprechen. Alternativ kann man *Parameter* direkt über den Aufruf der *CreateParameter*-Methode zu einem *Command*-Objekt hinzufügen.

Mit dem OLEDB .NET Data Provider wird das Fragezeichen (*?*) als Platzhalter eingesetzt. Der SQL Server .NET Data Provider unterstützt keine Fragezeichen (*?*) als Platzhalter für Parameter. Hier müssen so genannte benannte Parameter (*named parameters)* zum Einsatz kommen:

```
SELECT * FROM Benutzer WHERE Benutzername = @BenName
```

Der Einsatz von Parametern ist vor allem bei der Verwendung von gespeicherten Prozeduren unabdingbar. Wie diese in ADO.NET verwendet werden zeigt der folgende Abschnitt.

4.2.8 Gespeicherte Prozeduren

In jeder ernsthaft eingesetzten Web-Anwendung wird man aufgrund einer besseren Skalierbarkeit auf textuelle SQL-Anweisungen zugunsten von gespeicherten Prozeduren verzichten. Gespeicherte Prozeduren (*Stored Procedures*) werden im Falle des SQL Server mit der erweiterten Syntax von Transact-SQL programmiert. Diese durchaus komplexen Programme werden in der Datenbank gehalten, dort kompiliert und auch dort ausgeführt. Dies bedeutet eine wesentlich bessere Leistung als das Arbeiten mit SQL-Anweisungen aus der Programmlogik heraus. Für die Arbeit mit gespeicherten Prozeduren muss man in ADO.NET einige Dinge beachten.

Das Programmiermodell bei gespeicherten Prozeduren ist ähnlich wie bei reinen SQL-Anweisungen unter Verwendung von Parametern. Einziger, aber wichtiger Unterschied ist die explizite Angabe, dass es sich um eine gespeicherte Prozedur handelt. Dies geschieht über das Setzen der Eigenschaft *CommandType*. Folgendes Fragment setzt für ein gültiges *Command*-Objekt die *CommandType*-Eigenschaft für gespeicherte Prozeduren:

```
oCmd.CommandType = CommandType.StoredProcedure;
```

Einfügen von Daten

Für den weiteren Verlauf wollen wir eine einfache gespeicherte Prozedur für das Einfügen von Daten verwenden. Diese Prozedur dient dem Einfügen von neuen Benutzern in die Benutzertabelle. Der Code für die Prozedur `spBenutzerEinfuegen` ist in Listing 4.6 aufgeführt.

```
CREATE PROCEDURE spBenutzerEinfuegen
   @BenName varchar(10),
   @Vorname varchar(20),
   @Nachname varchar(20),
   @Passwort varchar(10)
AS
   INSERT INTO Benutzer Values
      (@BenName, @Vorname, @Nachname,
      @Passwort, NULL, NULL)
GO
```

Listing 4.6: Gespeicherte Prozedur zum Anlegen von Benutzern

Um nun Daten über diese gespeicherte Prozedur in unsere Datenbank einfügen zu können, müssen wir die Parameter in unserem Quelltext entsprechend füllen. Listing 4.7 zeigt wie man die gleiche Funktionalität wie in Listing 4.3 mittels gespeicherter Prozeduren erreicht.

```
SqlConnection oConn = new SqlConnection
   ("server=localhost;uid=BuchUser;pwd=ws_buch;
   database=WS_Buch");
string sSP = "spBenutzerEinfuegen";
SqlCommand oCmd = new SqlCommand(sSP, oConn);
```

```
oCmd.CommandType = CommandType.StoredProcedure;

string sBenName = "lmatt";
string sVorname = "Lothar";
string sNachname = "Matthäus";
string sPasswort = "testpw";

oCmd.Parameters.Add("@BenName", sBenName);
oCmd.Parameters.Add("@Vorname", sVorname);
oCmd.Parameters.Add("@Nachname", sNachname);
oCmd.Parameters.Add("@Passwort", sPasswort);

oConn.Open();
int iRes = oCmd.ExecuteNonQuery();
oConn.Close();
```

Listing 4.7: Einfügen von Daten über eine gespeicherte Prozedur

Auslesen von Daten

Die prinzipielle Vorgehensweise beim Auslesen von Daten über eine gespeicherte Prozedur ist der Vorgehensweise beim Einfügen von Daten ähnlich. In einem Leseszenario kommen jedoch zu den Eingabe- noch die Ausgabeparameter hinzu. Die Ausgabewerte können dann über einen *DataReader* (oder über ein *DataSet*, siehe Abschnitt 4.2.10) im Programm weiter verarbeitet werden. Für unser Beispiel wollen wir eine einfache Prozedur implementieren, die alle verfügbaren Informationen über einen bestimmten Benutzer zurückliefert. Listing 4.8 zeigt die gespeicherte Prozedur `spBenutzerAnzeigen`.

```
CREATE PROCEDURE spBenutzerAnzeigen
   @BenName varchar(10)
AS
   SELECT * FROM Benutzer WHERE Benutzername = @BenName
GO
```

Listing 4.8: Gespeicherte Prozedur zum Anzeigen eines Benutzers

In dieser Prozedur wird der im System registrierte Benutzername als Eingabeparameter übergeben. Als Ergebnis liefert die Prozedur sämtliche Spalten der Benutzertabelle für diesen Benutzer zurück. Der aufrufende Code steht in Listing 4.9.

```
SqlDataReader oDR = null;
SqlConnection oConn = new SqlConnection
   ("server=localhost;uid=BuchUser;pwd=ws_buch;
   database=WS_Buch");
string sSP = "spBenutzerAnzeigen";
SqlCommand oCmd = new SqlCommand(sSP, oConn);
oCmd.CommandType = CommandType.StoredProcedure;

string sBenName = "cweyer";
```

```
oCmd.Parameters.Add("@BenName", sBenName);
oConn.Open();
oDR = oCmd.ExecuteReader(
   CommandBehavior.CloseConnection);

while (oDR.Read())
{
   Response.Write(oDR.GetInt32(0) +
      "   ");
   Response.Write(oDR.GetString(3) + ", " +
      oDR.GetString(2) + "   ");
   Response.Write(oDR.GetString(1) +
      "   ");
   if (oDR.IsDBNull(5))
      Response.Write("Nicht verfügbar<br>");
   else
      Response.Write(oDR.GetDateTime(5).
         ToLongDateString() + "<br>");
}
oDR.Close();
oConn.Close();
```

Listing 4.9: Auslesen von Daten aus einer Datenbank über eine gespeicherte Prozedur

Ein großer Vorteil von gespeicherten Prozeduren ist die Möglichkeit, mehrere Ausgabeparameter (*Out-Parameter*) zusätzlich zu einem Rückgabeparameter zur Verfügung zu haben. Dies entspricht in etwa dem Programmierparadigma von Referenzwerten: Eine Funktion in einer Programmiersprache kann einen Rückgabeparameter besitzen, kann aber in der Parameterliste mehrere Werte für die Ausgabe setzen, wenn diese per Referenz übergeben werden. Somit ist auch die Variante möglich, einen Parameter in die Prozedur zu geben, der dann in der Prozedur verändert wird und vom aufrufenden Programm wieder gelesen werden kann (*InOut-Parameter*).

Ein Beispiel soll die Problematik verdeutlichen: In einer gespeicherten Prozedur zum Einfügen von Datensätzen in eine SQL Server-Datenbank soll als Rückgabeparameter die Anzahl der Zeilen in der Tabelle nach dem Einfügen gesetzt werden. Zusätzlich sind wir aber auch noch an dem Wert für das Auto-Feld in der Tabelle Benutzer interessiert – also setzen wir den Wert des Auto-Felds als Ausgabeparameter in der Prozedur. Dies erfordert kleine Änderungen an unserer Prozedur. Die neue Version (`spBenutzerEinfuegen2`) ist in Listing 4.10 zu sehen.

```
CREATE PROCEDURE spBenutzerEinfuegen2
   @BenName varchar(10),
   @Vorname varchar(20),
   @Nachname varchar(20),
   @Passwort varchar(10),
   @Autowert int OUT
```

```
AS
   INSERT INTO Benutzer Values
      (@BenName, @Vorname, @Nachname,
      @Passwort, NULL, NULL)
   SET @Autowert = @@Identity
   RETURN @@Rowcount
GO
```

Listing 4.10: Gespeicherte Prozedur zum Anlegen von Benutzern mit Rück- und Ausgabewert

Neu ist die Hinzunahme eines Out-Parameters in der Parameterliste der Prozedur. Ebenso neu ist das Setzen des Rückgabewerts (*RETURN …*) und des Ausgabewerts für den Autowert. In beiden Fällen werden interne Zähler des SQL Server verwendet: *@@Identity* liefert den Autowert der soeben eingefügten Zeile zurück, wohingegen *@@Rowcount* die Anzahl der von der Operation betroffenen Zeilen in der Tabelle beinhaltet. Listing 4.11 zeigt den relevanten Teil für den Zugriff auf die gespeicherte Prozedur aus Listing 4.10 (in diesem Beispiel ergeben die Werte für `iRes` und `iRows` das gleiche Ergebnis!).

```
oCmd.Parameters.Add("@BenName", sBenName);
oCmd.Parameters.Add("@Vorname", sVorname);
oCmd.Parameters.Add("@Nachname", sNachname);
oCmd.Parameters.Add("@Passwort", sPasswort);

SqlParameter oParam = oCmd.Parameters.Add(
   "@RowCount", SqlDbType.Int);
oParam.Direction = ParameterDirection.ReturnValue;
oParam = oCmd.Parameters.Add("
   @Autowert", SqlDbType.Int);
oParam.Direction = ParameterDirection.Output;

oConn.Open();
int iRes = oCmd.ExecuteNonQuery();
int iRows = (int)oCmd.Parameters["@RowCount"].Value;
int iId = (int)oCmd.Parameters["@Autowert"].Value;

Response.Write("Anzahl der betroffenen Zeilen
   in Tabelle: " + iRows + "<br>");
Response.Write("Autowert: " + iId);
oConn.Close();
```

Listing 4.11: Einfügen von Daten in eine Datenbank über eine gespeicherte Prozedur mit Aus- und Rückgabeparameter

In Listing 4.11 werden zusätzlich zu den bekannten Eingabeparametern noch zwei neue Parameter hinzugefügt. Das Richtungsverhalten eines Parameters ist standardmäßig *Input* (vgl Tabelle 4.8). Bei Ausgabe- oder Rückgabeparametern muss man zusätzlich noch das Richtungsverhalten explizit mit angeben. Dies erfolgt durch das Setzen einer Konstanten aus der *Parame-*

terDirection-Struktur. Zusätzlich müssen wir den Datentyp spezifizieren, den unsere beiden neuen Parameter besitzen sollen. Diese Angabe ist bei den Eingabeparametern nicht notwendig, da das *Parameter*-Objekt anhand des übergebenen Werts automatisch den Typ des Parameters erkennt. Wenn wir ganz sicher gehen wollten, dass auch bei den Eingabeparametern nur korrekte Typen angenommen werden, könnten wir als zusätzlichen Parameter bei der *Add*-Methode noch den Datentyp mit angeben. Nach Ausführung des Kommandos können wir dann über die *Parameters*-Kollektion auf die zurückgelieferten Werte aus der Prozedur zugreifen.

Tab. 4.8: Mögliche Werte für das Richtungsverhalten von Parametern bei Datenbankoperationen

Wert	Beschreibung
Input	Parameter ist ein Eingabeparameter.
InputOutput	Parameter kann sowohl als Ein- als auch als Ausgabeparameter verwendet werden.
Output	Parameter ist Ausgabeparameter.
ReturnValue	Parameter repräsentiert einen Rückgabewert (z. B. aus einer gespeicherten Prozedur).

Natürlich können gespeicherte Prozeduren sehr komplex werden. So können innerhalb einer Prozedur eine oder mehrere Tabellen aktualisiert werden. Anschließend kann eine Ergebnismenge an die aufrufende Funktion zurückgegeben werden. Somit fällt die Unterscheidung von Ein-, Aus- oder Rückgabeparameter nicht in eine Kategorisierung von Lesen oder Schreiben.

4.2.9 Transaktionen

Beim Schreiben und Aktualisieren von Daten können unter Umständen Probleme auftreten. Eine Netzwerkverbindung zwischen zwei Datenbanken kann unterbrochen werden, ein Datenbankserver kann abstürzen oder eine Festplatte im Datenbankserver versagt ihren Dienst. Man sollte also darauf bedacht sein, dass Daten entweder richtig in die Datenbank eingefügt werden oder gar nicht. Die Situation verschärft sich, wenn mehrere Tabellen oder sogar Datenbankmanagementsysteme im Spiel sind. Für diese Art von Operationen gibt es das Konzept der Transaktionen. Im Datenbankjargon ist eine Transaktion eine Menge von Datenbankoperationen, die zu einer logischen Einheit zusammengefasst sind. Treten keine Fehler auf, so werden alle Änderungen innerhalb der Transaktion persistent gemacht – treten jedoch Fehler auf, so wird der Zustand wieder hergestellt, der vor Abarbeitung der Transaktion aktuell war. Somit ist eine Integrität und Konsistenz des Datenbestandes gewährleistet. Auch in ADO.NET gibt es Möglichkeiten, Datenbankoperationen innerhalb von Transaktionen ablaufen zu lassen.

Im Falle von lokalen Transaktionen innerhalb einer Datenbank sind ADO.NET-Transaktionen völlig ausreichend. Wenn jedoch verteilte Transaktionen ein Thema in einer Anwendung sind, dann müssen diese Transaktionen von einem Transaktionskoordinator gesteuert werden. Im .NET Framework übernimmt diese Aufgabe die COM+-Laufzeitumgebung zusammen mit dem MS DTC. Ein Beispiel für transaktionsbasierte XML Web Services unter COM+ finden Sie in Kapitel 6.

Ein Programmierer kann Datenbanktransaktionen direkt im SQL Server in gespeicherten Prozeduren unterbringen. Allerdings hat die Transaktionsteuerung auf Ebene von ADO.NET den Vorteil, dass nicht in jeder einzelnen Prozedur eine Transaktionssyntax und -semantik implementiert sein muss. Außerdem kann man auf dieser Ebene den Aufruf von gespeicherten Prozeduren mit dem von einfachen SQL-Anweisungen mischen (z.B. für Testzwecke). Wenn man zudem den OLEDB .NET Data Provider verwendet, kann man Transaktionen allgemein formulieren, ohne jeweils die Sprache der zugrunde liegenden Datenbank beherrschen zu müssen.

In ADO.NET ist das *Transaction*-Objekt (*SqlTransaction* und *OledbTransaction*) für die Steuerung von lokalen Transaktionen zuständig. Ein *Transaction*-Objekt wird immer an eine gültige Verbindung gebunden – und nur für dieses *Connection*-Objekt ist das *Transaction*-Objekt verantwortlich. Erzeugt wird das *Transaction*-Objekt durch den Aufruf der Methode *BeginTransaction* auf dem *Connection*-Objekt. Von diesem Zeitpunkt an werden alle Aktionen zur Steuerung der Transaktion über das *Transaction*-Objekt ausgeführt. Die Methode *Commit* zeigt dem *Transaction*-Objekt an, dass während der Abarbeitung im Programm alles in Ordnung war und dass es versuchen kann, dem Datenbehälter mitzuteilen, die getätigten Änderungen dauerhaft zu machen. Das Gegenstück zu *Commit* ist die Methode *Rollback*, die beim Auftreten von Fehlern oder Ausnahmen angewendet wird. Sie teilt dem *Transaction*-Objekt mit, dass der Datenbehälter die Änderungen nicht endgültig durchführen soll und dass der ursprüngliche Datenbestand unangetastet bleiben soll. Den ausschlaggebenden Teil einer ASP.NET-Seite für die transaktionsbasierte Arbeit mit einer Datenbank können Sie in Listing 4.12 einsehen.

```
SqlConnection oConn = new SqlConnection
   ("server=localhost;uid=BuchUser;pwd=ws_buch;
   database=WS_Buch");
SqlTransaction oTx;
SqlCommand oCmd = new SqlCommand();

oConn.Open();
oTx = oConn.BeginTransaction();
oCmd.Connection = oConn;
oCmd.Transaction = oTx;

try
{
```

```
   oCmd.CommandText = "INSERT INTO Benutzer Values
      ('btritsch', 'Bernhard', 'Tritsch', 'ichdottore',
      '" + DateTime.Now + "', NULL)";
   oCmd.ExecuteNonQuery();
   //throw new Exception("Ausnahmefehler erzeugt!");
   oCmd.CommandText = "INSERT INTO Benutzer Values
      ('dweyer', 'Diana', 'Weyer', 'meinpw',
      '" + DateTime.Now + "', NULL)";
   oCmd.ExecuteNonQuery();
   oTx.Commit();
   Response.Write("Transaktion erfolgreich -
      Daten wurden eingefügt.");
}
catch(Exception e)
{
   oTx.Rollback();
   Response.Write(e.ToString() + "<br>");
   Response.Write("Transaktion fehlgeschlagen -
      keine Daten eingefügt!");
}
oConn.Close();
```

Listing 4.12: Einfügen von Daten innerhalb einer ADO.NET-Transaktion

Wenn man das Beispiel ausführt, wird im Normalfall alles gut gehen und die Daten werden in die Tabelle eingefügt. Man kann jedoch durch Entkommentieren der Zeile `//throw new Exception("Ausnahmefehler erzeugt!");` eine künstliche Ausnahme erzeugen und verfolgen, was in diesem Fall passiert: Die Transaktion wird zurückgesetzt, denn es ist ein Fehler in der Abarbeitung des Programms aufgetreten.

Die Ausführung der Kommandos innerhalb einer Transaktion sollte immer in einem *try/catch*-Block stattfinden. Wenn beispielsweise der aufgerufene .NET Data Provider eine Ausnahme erzeugt, dann wird diese aufgefangen. Innerhalb des Codes kann dann die Transaktion zurück gesetzt werden.

4.2.10 Das ADO.NET DataSet

Die bisher gezeigten Vorgehensweisen zur Arbeit mit Daten waren allesamt von der zugrunde liegenden Datenbank abhängig. Dies bedeutet, dass eine Verbindung zur Datenquelle zwingend notwendig ist, um Informationen aus einem *DataReader* auszulesen. Für viele Szenarien ist dieses Modell auch ausreichend bzw. aufgrund der guten Leistungsfähigkeit auch gewünscht. Jedoch gibt es in einer verteilten Anwendung oft die Anforderung, Daten und Informationen aus einer Datenquelle über einen längeren Zeitraum hinweg im Programm zu halten. In diesem Fall müsste es ein Objekt geben, wel-

ches als Cache für die aus der Datenbank stammenden Daten agieren könnte. Schließlich sollte dieses Objekt gänzlich von der Datenquelle unabhängig sein, damit man sich nicht zwingend zu einem Datenbehälter verbinden muss, um mit dem Cache-Objekt arbeiten zu können. Dieses Alleskönner-Objekt existiert in ADO.NET und heißt *DataSet*.

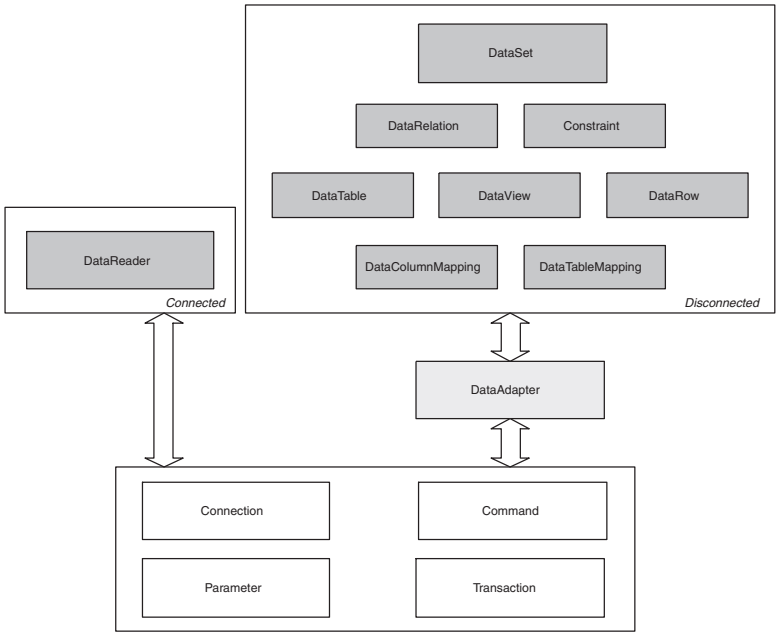


Abbildung 4.2: Klassenübersicht für connected und disconnected Szenarien

Das *DataSet* und seine verwandten Klassen (siehe Tabelle 4.9) werden im Gegensatz zum *DataReader* vornehmlich im so genannten *disconnected* Szenario eingesetzt (keine Verbindung zu einer Datenquelle, also als autonomes Objekt). Einen Überblick über die beteiligten Klassen in der ADO.NET-Welt kann man sich in Abbildung 4.2 verschaffen. Die Korrelation zwischen den wichtigsten Klassen und dem ADO.NET *DataSet* wird in Abbildung 4.3 nochmals genauer dargestellt. Bitte beachten Sie, dass die Thematik des *DataSets* hier nicht so detailliert besprochen werden kann, wie in der Abbildung angedeutet wird.

Name	Beschreibung
DataTable	Repräsentiert eine Tabelle im Speicher.
DataRelation	Beschreibt eine Vater-Kind-Beziehung zwischen zwei *DataTable*-Objekten.
Constraint	Repräsentiert eine Einschränkungsbedingung (*Constraint*), die auf eine oder mehrere *DataColumn*-Objekte angewendet werden kann.

Tab. 4.9: Wichtige mit dem DataSet verbundene Klassen

Tab. 4.9: Wichtige mit dem DataSet verbundene Klassen (Forts.)

Name	Beschreibung
DataView	Beschreibt eine bestimmte Ausprägung (*View*) einer *DataTable* für Sortieren, Filtern oder Suchen. Geeignet für Data Binding.
DataRow	Repräsentiert eine Zeile in einer *DataTable.*
DataColumn	Repräsentiert eine Spalte in einer *DataTable*.
DataAdapter	Beschreibt eine Menge von Kommandos und eine Verbindung zur Datenbank, um Daten in ein *DataSet* zu füllen und Änderungen an der Datenbank durchzuführen.

Daten lesen

Eine häufige Anwendung des *DataSets* ist die Verwendung als In-Memory-Datenbank (das Halten von Daten im Hauptspeicher ohne von einer physikalischen Datenbank abhängig zu sein). Da das *DataSet* eine Speicherrepräsentation von Daten ist und nicht mit einer Datenquelle verbunden ist, benötigt man ein weiteres Objekt welches bei Bedarf die Verbindung und Kontrolle zu einer Datenquelle übernimmt: das *DataAdapter*-Objekt. Die abstrakte Klasse *DataAdapter* muss von jedem .NET Data Provider implementiert werden. Wir werden in diesem Abschnitt den Fokus wieder auf den SQL Server Data Provider legen, deshalb verwenden wir die Klasse *SqlDataAdapter*. Der *DataAdapter* dient als Mediator-Objekt zwischen einem *DataSet* und einer Datenquelle. Über den *DataAdapter* kann man Tabellen in einem *DataSet* füllen. Außerdem setzt der *DataAdapter* Änderungen, die im *DataSet* vorgenommen wurden, so um, dass die Daten korrekt in der Datenquelle geändert werden – ohne dass das *DataSet* von der Quelle weiß. Der *DataAdapter* benötigt eine Verbindung über ein *Connection*-Objekt, um mit der Datenbank zu kommunizieren. Für die unterschiedlichen Kommandos zur Manipulation des Datenbestands besitzt die *DataAdapter*-Klasse vier Eigenschaften: *SelectCommand*, *InsertCommand*, *UpdateCommand* und *DeleteCommand*. Um nun also Daten auszulesen, damit sie in einem *DataSet* gespeichert werden können, muss zumindest die *SelectCommand*-Eigenschaft mit einer SQL-Anweisung belegt werden. Durch den Aufruf von *Fill* wird dann schließlich das *DataSet* mit den Daten aus der Datenbank über den *DataAdapter* gefüllt. Das Beispiel in Listing 4.13 soll diesen Sachverhalt illustrieren. Hier werden die Benutzernamen in der Tabelle `Benutzer` in einer ASP.NET-Seite ausgegeben.

```
SqlConnection oConn = new SqlConnection(
   "server=localhost;uid=BuchUser;pwd=ws_buch;
   database=WS_Buch");
SqlDataAdapter oDA = new SqlDataAdapter(
   "SELECT * FROM Benutzer", oConn);

oConn.Open();
DataSet oDS = new DataSet();
oDA.Fill(oDS,"Benutzer");
```

```
foreach (DataRow oDR in oDS.Tables["Benutzer"].Rows)
{
   Response.Write(oDR["Benutzername"].ToString() +
      "<br>");
}
oConn.Close();
```

Listing 4.13: Füllen eines DataSets über einen DataAdapter

Arbeiten mit dem DataSet

Wenn man ein *DataSet* im Speicher verfügbar hat, kann man mit ihm arbeiten als wäre es eine Datenbank im Hauptspeicher. Außer der Möglichkeit, ein *DataSet* über eine Datenquelle zu füllen, kann dies auch manuell erstellt und gefüllt werden. Folgende Zeile erzeugt ein neues *DataSet*-Objekt:

```
DataSet buchDS = new DataSet(„BuchPortal“);
```

Der Parameter für den Konstruktor gibt dem neuen *DataSet* einen eindeutigen Namen. Ebenso einfach kann man diesem neu erzeugten *DataSet* nun Tabellen hinzufügen:

```
DataTable sportartenTable =
   buchDS.Tables.Add(“Sportarten“);
```

Um ein komplett funktionsfähiges *DataSet* zu erhalten, müssen wir nur noch der Tabelle die entsprechenden Spalten (also das Tabellenschema) mitteilen. In den folgenden Zeilen fügen wir der Tabelle `Sportarten` zwei Spalten zu und deklarieren zusätzlich noch den Primärschlüssel für die Tabelle:

```
DataColumn pkCol = sportartenTable.Columns.Add(
   „ID“, typeof(int));
sportartenTable.Columns.Add(
   “Sportart”, typeof(string));
sportartenTable.PrimaryKey =
   new DataColumn[] {pkCol};
```

Mit diesen wenigen Zeilen haben wir eine voll funktionsfähige Datenbank im Hauptspeicher verfügbar, mit der wir nun unsere Einfüge-, Lösch- oder Aktualisierungsoperationen direkt auf diesem Objekt auch ohne SQL-Anweisungen durchführen können.

Mit folgenden Anweisungen erzeugen wir einen neuen Eintrag in der Tabelle Sportarten:

```
DataRow neuerEintrag =
   buchDS.Tables[„Sportarten“].NewRow;
neuerEintrag[„ID“] = 1;
neuerEintrag[„Sportart“] = „Extreme Buchschreibing“;
buchDS.Tables[„Sportarten“].Rows.Add(neuerEintrag);
```

Auch für den Fall, dass wir Daten in einer bereits existierenden Zeile ändern wollen, gibt es eine Lösung im *DataSet*. Dabei kann man auf die einzelnen Elemente in der Tabellen- und Spaltenkollektion (*Items*-Kollektion) entweder über die vergebenen Namen oder per Index zugreifen:

```
buchDS.Tables[„Sportarten“].Rows[0].Item[„Sportart“] =
   „Extreme .NET-Buchschreibing“;
```

Schlussendlich kann noch der Fall auftreten, dass wir Datensätze (sprich Zeilen) aus einer Tabelle im *DataSet* löschen wollen. Dies geschieht durch einen einzigen Aufruf:

```
buchDS.Tables[„Sportarten“].Rows[0].Delete()
```

Wir haben in diesem Abschnitt gesehen, dass es sich beim *DataSet* tatsächlich um eine Art In-Memory-Datenbank handelt, mit der man direkt auf Daten zugreifen und diese manipulieren kann.

Daten in der Datenbank aktualisieren

Man kann also auf die Tabellen im *DataSet* zugreifen oder einzelne Wert ändern bzw. neue Zeilen in eine *DataTable* einfügen. Wenn wir unser *DataSet* aus einer Datenbank heraus gefüllt haben, wollen wir die vorgenommenen Änderungen auch in der Datenbank persistent machen. Hierfür genügt in der Regel eine Zeile. Der zugehörige *DataAdapter* verfügt über die Methode *Update*, die gemäß dem spezifizierten *UpdateCommand* die Daten zurück in die Datenbank schreibt:

```
oDA.Update(buchDS);
```

Wenn kein *UpdateCommand* spezifiziert wurde, dann wird eine Ausnahme erzeugt. Man kann sich übrigens die Mühe sparen, die einzelnen Kommandos selbst zu bestimmen. Mithilfe der *SqlCommandBuilder*-Klasse können diese Anweisungen automatisch erzeugt werden. Ein Beispiel hierfür finden Sie in Abschnitt 4.2.11.

Die *DataSet*-Klasse besitzt die Methode *AcceptChanges*. Aufgrund des Namens dieser Methode könnte man annehmen, dass diese Methode dafür sorgt, die im *DataSet* vorgenommenen Änderungen zu akzeptieren und für die Datenbank aufzubereiten. Das Gegenteil ist jedoch der Fall: *AcceptChanges* akzeptiert die Änderungen am *DataSet* und setzt alle Flags für die Änderungen zurück. Somit hat ein Aufruf von *Update* keine Konsequenz, da die Änderungs-Flags nicht mehr existieren. Vielmehr ruft Update intern die Methode *AcceptChanges* auf, bevor die Daten in die Datenbank geschrieben werden.

Die Funktionalität des *DataSets* ist noch sehr viel facettenreicher als es hier dargestellt werden kann. Daher kann die obige Abhandlung nur eine Einführung in die Arbeit mit den Fähigkeiten des ADO.NET *DataSets* sein.

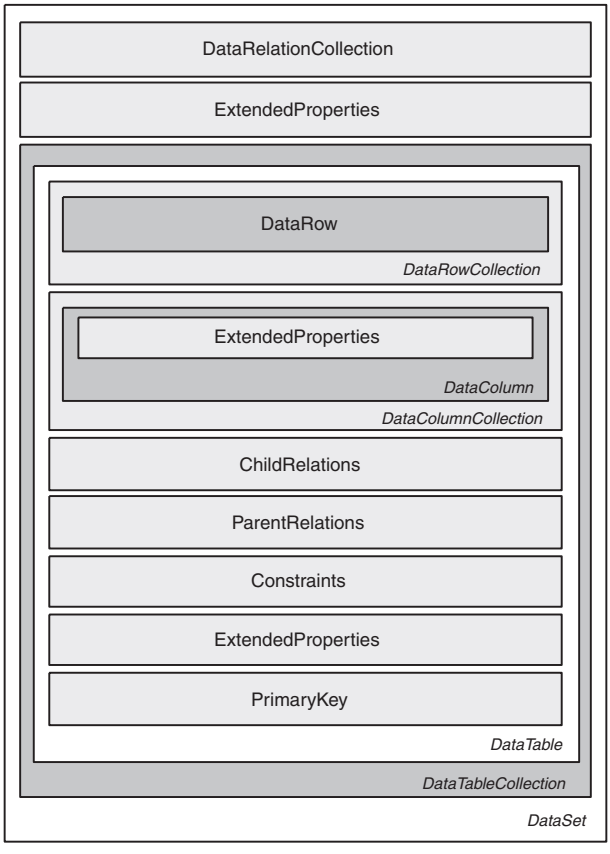


Abbildung 4.3: Der interne Aufbau des DataSet-Objekts

4.2.11 Binärdaten

Ein Spezialfall beim Arbeiten mit Datenbanken ist das Schreiben und Lesen von Binärdaten. Wenn man kleine Bilder im GIF- oder JPG-Format für eine Webseite in einer Tabelle abspeichern möchte, dann muss die Binärdarstellung der Bilder in ein speziell dafür vorgesehenes Feld eingefügt und auch daraus ausgelesen werden. Wenn man in der Benutzerverwaltung zusätzlich noch digitale Zertifikate – die eine sichere Kommunikation der Benutzer untereinander garantieren – mit aufnehmen will, dann besteht auch die Anforderung, diese binären Daten über ADO.NET zu schreiben und zu lesen. Für unsere Zwecke wollen wir unseren Benutzern ein Bild zuordnen.

Im Folgenden wird keine komplette Anwendung besprochen, mit der man Bilder in eine Datenbank speichern und wieder auslesen kann. Vielmehr werden Code-Fragmente bereitgestellt, mit denen man leicht eine solche Applikation realisieren kann.

Schreiben von Binärdaten

Der einfachste Weg, um Binärdaten zu handhaben, ist der Einsatz eines *DataSets*. Die Daten müssen zunächst aus einer Datei in den Speicher geladen werden. Hierfür verwenden wir ein *FileStream*-Objekt aus dem Namensraum *System.IO*. Der Inhalt dieses Objekts wird anschließend in ein Byte-Array gelesen. Dieses Byte-Array kann dann in ein entsprechend vorbereitetes *DataSet* eingefügt werden. Den Code zum Einfügen von Binärdaten in eine Tabelle finden Sie in Listing 4.14.

Über ein *DataAdapter*-Objekt erzeugen wir ein neues *DataSet*. Der *DataAdapter* enthält eine SQL-Anweisung zum Auslesen der Felder *ID* und *Bild*. In das Feld *Bild* soll zu einem späteren Zeitpunkt die Bildinformation in Form von Binärdaten eingefügt werden. Damit die am *DataSet* vorgenommenen Änderungen nach Aufruf der Methode *Update* automatisch richtig ausgeführt werden, erzeugen wir ein *SqlCommandBuilder*-Objekt. Dieses Objekt kann automatisch korrekte Kommandos für die Änderungen in einem *DataSet* generieren. Voraussetzung hierfür ist, dass ein gültiges *SelectCommand* für den zuständigen *DataAdapter* gesetzt wurde (was in unserem Beispiel im Konstruktor geschieht). Damit die Klasse *SqlCommandBuilder* ein Kommando zum Einfügen von Daten erzeugen kann, benötigt sie eindeutige Schlüsselinformationen über die Tabelle – daher wird über den *DataAdapter* auch das Feld *ID* ausgelesen (es dient als Primärschlüssel in der Tabelle `Benutzer`). Schließlich wird das *DataSet* über den *DataAdapter* gefüllt, das Feld *Bild* wird mit dem Byte-Array besetzt und die Daten werden über *Update* in die Datenbank geschrieben.

```
SqlConnection oConn = new SqlConnection(
   "server=localhost;uid=BuchUser;pwd=ws_buch;
   database=WS_Buch");
SqlDataAdapter oDA = new SqlDataAdapter(
   "SELECT ID, Bild FROM Benutzer WHERE Benutzername='"
   + sBenName + "'", oConn);
SqlCommandBuilder oCB = new SqlCommandBuilder(oDA);
DataSet oDS = new DataSet("BenutzerBild");

FileStream oFS = new FileStream(Server.MapPath(@".\" +
   sBenName + ".JPG"), FileMode.OpenOrCreate,
   FileAccess.Read);
byte[] oData= new byte[oFS.Length];
oFS.Read(oData, 0, System.Convert.ToInt32(oFS.Length));
oFS.Close();

oDA.Fill(oDS,"Benutzer");
oDS.Tables["Benutzer"].Rows[0]["Bild"] = oData;
oDA.Update(oDS, "Benutzer");

oConn.Close();
```

Listing 4.14: Einfügen von Binärdaten in eine Tabelle

Lesen von Binärdaten

Das Auslesen der Daten aus der Tabelle ist prinzipiell ebenso einfach wie das Einfügen. Über einen entsprechenden *DataAdapter* wird ein *DataSet* mit dem Inhalt aus der Datenbank gefüllt. Der Wert für das Bild wird aus dem Feld Bild des *DataSets* in ein Byte-Array kopiert. Von dort wird es über ein *FileStream*-Objekt in eine Datei auf der Festplatte geschrieben. Der beschriebene Vorgang ist als Quelltext in Listing 4.15 zu sehen.

```
SqlConnection oConn = new SqlConnection(
   "server=localhost;uid=BuchUser;pwd=ws_buch;
   database=WS_Buch");
SqlDataAdapter oDA = new SqlDataAdapter(
   "SELECT Bild FROM Benutzer WHERE Benutzername='"
   + sBenName + "'", oConn);
DataSet oDS = new DataSet("BenutzerBild");

oDA.Fill(oDS, "Benutzer");
byte[] oData= new byte[0];

oData =  (byte[])oDS.Tables["Benutzer"].Rows[0]["Bild"];
int iSize = oData.GetUpperBound(0);

FileStream oFS = new FileStream(Server.MapPath(@".\" +
   sBenName + "_DB.JPG"), FileMode.OpenOrCreate,
   FileAccess.Write);
oFS.Write(oData, 0, iSize);
oFS.Close();
oConn.Close()
```

Listing 4.15: Auslesen von Binärdaten aus einer Tabelle

4.3 XML in .NET

Die *eXtensible Markup Language* (*XML*) ist in aller Munde. Mit XML kann man strukturierte Dokumente erzeugen, die sich in den unterschiedlichsten Kontexten verwenden lassen. So kann man auf Basis eines XML-Dokuments aus den darin enthaltenen Daten eine HTML-Darstellung erzeugen, sie in einer Datenbank verwalten, sie in einer Anwendung direkt verwenden oder das Dokument mit anderen Geschäftspartnern austauschen. XML ist für viele Probleme die richtige Lösung – allerdings wird es in vielen Situationen auch überbewertet. In diesem Abschnitt werde ich Ihnen in kompakter Form die Techniken und Vorgehensweisen erläutern, mit denen man auf der .NET-Plattform mit XML arbeiten kann.

Eine kurze Einführung zu XML und verwandten Technologien vor allem im Bereich XML Web Services finden Sie in Kapitel 5.

4.3.1 Architektur

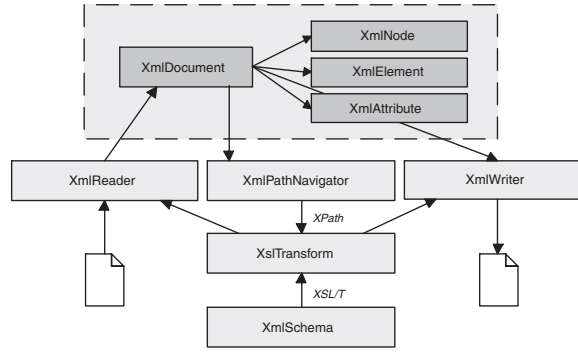
Die XML-Architektur im .NET Framework ist grundverschieden zur MSXML-Implementierung, wie sie aus der COM-Welt bekannt ist. Dies resultiert zum einen daraus, dass in der .NET-Umgebung eine klare Klassenstruktur und -hierarchie eingeführt wurde, und zum anderen daraus, dass man bei der neuen Lösung einen an öffentlichen Standards orientierten Weg eingeschlagen hat. Microsoft unterstützt die meisten der offiziellen XML-Standards des *World Wide Web Consortiums* (*W3C*). Einzig die Spezifikation für das *Simple API for XML* (*SAX*) hat man bewusst nicht implementiert, da mit der *XmlReader*/*XmlWriter*-Architektur eine ähnliche, aber wesentlich mächtigere und leistungsfähigere Lösung zur Verfügung steht. Eine Liste der unterstützten Standards und Empfehlungen steht in Tabelle 4.10.

Tab. 4.10: Unterstützte XML-Standards und -Empfehlungen in .NET

Standard	URL
XML 1.0	*http://www.w3.org/TR/1998/REC-xml-19980210*
XML Namespaces	*http://www.w3.org/TR/REC-xml-names/*
XSD Schemas	*http://www.w3.org/2001/XMLSchema*
XPath-Ausdrücke	*http://www.w3.org/TR/xpath/*
XSLT	*http://www.w3.org/TR/xslt/*
DOM Level 1 Core	*http://www.w3.org/TR/REC-DOM-Level-1/*
DOM Level 2 Core	*http://www.w3.org/TR/DOM-Level-2/*

Eine Übersicht über die allgemeine Architektur zur Arbeit mit XML in .NET können Sie aus Abbildung 4.4 ersehen. Die wichtigsten Klassen zum Lesen von XML-Dokumenten sind *XmlDocument* und *XmlReader* (bzw. deren Ableitungen). Ein Dokument schreiben bzw. verändern kann man auch über die Klasse *XmlDocument* oder mittels *XmlWriter* (bzw. den entsprechenden Ableitungen). *XmlDocument* unterstützt zusammen mit drei Helferklassen die *Document Object Model* (*DOM*)-Spezifikation des W3C. *XmlReader* und *XmlWriter* sind Microsoft-eigene Implementierungen, die für eine schnelle Bearbeitung von XML-Daten ins Leben gerufen wurden.

Abbildung 4.4: XML-Architektur in .NET



Einer der großen Vorteile der XML-Architektur in .NET im Vergleich zu MSXML ist der generische und erweiterbare Aufbau. Neben der Unterstützung öffentlicher Standards ist dieser Fakt dafür verantwortlich, dass XML auf einfache Weise zum integralen Bestandteil einer .NET-basierten Web-Anwendung werden kann. Durch die Verwendung von abstrakten Basisklassen wie *XmlReader* oder *XmlWriter* kann ein Programmierer seine eigene spezialisierte Implementierung der jeweiligen Funktionalität programmieren und anderen zur Verfügung stellen. Darüber hinausgehend bietet das XML-Rahmenwerk in .NET eine Architektur mit austauschbaren Komponenten an. Dies bedeutet, dass sich die auf die abstrakten Basisklassen stützenden Konzepte einfach durch andere bereits vorhandene oder neu implementierte Lösungen substituieren lassen.

4.3.2 Der Namensraum System.Xml

Sämtliche Klassen für den Zugriff auf XML-Daten befinden sich im .NET-Namensraum *System.Xml* (dieser wiederum befindet sich ausschließlich in der Assembly *System.Xml.dll*). Der Namensraum umfasst knapp 40 Klassen, die wir hier natürlich nicht alle besprechen können und wollen. Ich werde mich ähnlich wie bei ADO.NET auf die wichtigsten Klassen beschränken. Viele der Klassen sind auch einfach nur Helferklassen für die am häufigsten verwendeten Hauptklassen. Die Wichtigkeit dieser Klassen ergibt sich aus der Häufigkeit ihres Einsatzes in alltäglichen Problemstellungen. Die Liste dieser Klassen finden Sie in Tabelle 4.11.

Klasse	Beschreibung
XmlTextReader	Ermöglicht schnelles Lesen (nur nach vorn) von XML-Daten.
XmlNodeReader	Implementiert einen *XmlReader* über einer DOM-Struktur.
XmlValidatingReader	Ermöglicht die Validierung eines XML-Dokuments gegen eine DTD-, XDR- oder XML Schema-Beschreibung.
XmlTextWriter	Ermöglicht schnelles Schreiben (nur nach vorne) von XML-Daten.
XmlDocument	Implementiert das W3C DOM (Level 1 und 2).
XmlDataDocument	Eine Implementierung von *XmlDocument*, die mit einem *DataSet* verbunden werden kann.
XPathDocument	Dient als schneller Cache für die Bearbeitung von Dokumenten mit XSLT und XPath.
XPathNavigator	Stellt ein XPath-Datenmodell über einer Datenquelle zur Verfügung.
XslTransform	En XSLT-Prozessor für die Transformation von XML-Dokumenten.

Tab. 4.11: Wichtige Klassen im Namensraum System.Xml

XmlTextReader, XmlNodeReader, XmlValidatingReader. Die gemeinsame abstrakte Basisklasse für diese drei Implementierungen ist *XmlReader*. Diese Klassen ermöglichen ein schnelles Lesen von XML-Dokumenten in einem Datenstrom. Das Lesen ist nur in eine Richtung (vom Anfang bis zum Ende des Stroms) möglich. Dadurch wird so wenig Speicher wie möglich verbraucht. *XmlTextReader* ist die schnellste der drei Varianten. Mit dieser Klasse kann man wohlgeformtes XML lesen, aber keine Validierung der Daten vornehmen. Für die Validierung von Dokumenten wird in der Regel die Klasse *XmlValidatingReader* heran gezogen. Durch die Angabe einer *DTD* (*Document Type Definition*) oder eines XML-Schemas kann ein XML-Dokument auf syntaktische Korrektheit geprüft werden. Mithilfe von *XmlNodeReader* kann man schließlich einen Teilbaum einer XML-DOM-Darstellung lesen und verarbeiten.

XmlTextWriter. Die Basisklasse für *XmlTextWriter* ist *XmlWriter*. *XmlTextWriter* stellt einen Behälter zur Verfügung, mit dem man XML-Daten in einen Datenstrom schreiben kann. Diese Vorgehensweise impliziert das Schreiben der Daten in eine Richtung (vom Anfang bis zum Ende des Dokuments).

XmlDocument, XmlDataDocument. *XmlDocument* repräsentiert ein XML-Dokument in Form einer DOM-Darstellung. Über *XmlDocument* kann man durch ein XML-Dokument navigieren und dieses auch verändern. Die Klasse *XmlDataDocument* ist ein Spezialfall von *XmlDocument* und erlaubt, dass strukturierte Daten (XML) über ein ADO.NET *DataSet* gelesen und geschrieben werden können.

XPathDocument. Mittels *XPathDocument* kann ein Programmierer über die Angabe von XSLT- oder XPath-Anweisungen ein XML-Dokument auf schnelle Art und Weise bearbeiten bzw. durchsuchen. Zusammen mit *XslTransform* spielt es seine Vorteile gegenüber *XmlDocument* aus, denn es ist speziell für die beiden oben beschriebenen Aufgaben implementiert und optimiert worden.

XPathNavigator. Die Klasse *XPathNavigator* basiert auf dem XPath-Datenmodell und stellt ein Cursor-basiertes Konzept zum Lesen jeglicher Daten zur Verfügung. Einzige Voraussetzung ist, dass die zu bearbeitenden Daten über eine Klasse zugänglich sind, welche die Schnittstelle *IXPathNavigable* implementiert.

XslTransform. Mithilfe dieser Klasse können XML-Dokumente durch Anwendung eines XSLT-Stylesheets in eine andere Repräsentation umgewandelt werden. Ein typischer Anwendungsfall ist die Konvertierung von XML-Rohdaten in eine HTML-Darstellung.

In den folgenden Abschnitten werden die typischsten Anwendungsfälle für die Verarbeitung von XML-Daten anhand von praktischen Beispielen mit .NET-Technologien implementiert.

4.3.3 Lesen von XML-Dokumenten

Es gibt grundsätzlich zwei Arten in .NET XML-Dokumente zu lesen: Zum einen über das an W3C DOM anlehnende *XmlDocument* und zum anderen durch eine Implementierung von *XmlReader*. Der Einsatz von *XmlDocument* erfordert immer die Nutzung des Datenmodells für eine DOM-Baumdarstellung im Speicher. Wenn man auf schnelle Art und Weise XML-Daten parsen möchte, ist die Klasse *XmlTextReader* die beste Alternative.

Werfen wir zunächst einen Blick auf die Vorgehensweise, wie man mit *XmlTextReader* ein XML-Dokument einliest und weiter verarbeitet. Im Beispiel aus Listing 4.16 (zeigt nur Auszüge aus dem Gesamtlisting) wird die Datei *Adressbuch.xml* eingelesen. Anhand der XML-Darstellung wird im Code der jeweilige Typ eines Elements erfragt und entsprechend in einer ASP.NET-Seite ausgegeben.

```
XmlTextReader reader = null;
reader = new XmlTextReader(Server.MapPath(
   "Adressbuch.xml"));
Response.Write("<XMP>");
FormatXml(reader);
Response.Write("</XMP>");
reader.Close();

private void FormatXml(XmlReader reader)
{
   while(reader.Read())
   {
      switch(reader.NodeType)
      {
         case XmlNodeType.XmlDeclaration:
            Format(reader, "XmlDeclaration");
            break;
         case XmlNodeType.ProcessingInstruction:
            Format(reader, "ProcessingInstruction");
            break;
         case XmlNodeType.DocumentType:
            Format(reader, "DocumentType");
            break;
         case XmlNodeType.Comment:
            Format(reader, "Comment");
            break;
         case XmlNodeType.Element:
            Format(reader, "Element");
            break;
         case XmlNodeType.Text:
            Format(reader, "Text");
            break;
      }
```

```
    }
}

private void Format(XmlReader reader, String nodeType)
{
    for (int i=0; i < reader.Depth; i++)
    {
        Response.Write("\t");
    }
    Response.Write(nodeType + ": <" + reader.Name + ">" +
        reader.Value);
    if (reader.HasAttributes)
    {
        Response.Write(" Attribute:");
        for (int j=0; j < reader.AttributeCount; j++)
        {
            Response.Write(" [" + j + "] " + reader[j]);
        }
    }
    Response.Write("\n");
```

Listing 4.16: Lesen eines XML-Dokuments mit XmlTextReader

Eine der am meisten verwendeten Methoden in der Klasse *XmlTextReader* ist *Read*. Mit *Read* wird der nächste Knoten im XML-Strom gelesen. Über die Eigenschaft *NodeType* kann der Typ des gerade aktuellen Knotens erfragt werden. Diese unterschiedlichen Typen sind in der Aufzählung *XmlNodeType* festgehalten. In den meisten Anwendungsfällen wird eine schnelle Vorwärtssuche innerhalb eines XML-Stroms nach diesem Muster ausreichend sein.

Die Klasse *XmlTextReader* besitzt mehrere überladene Konstruktoren. Außer der Angabe einer Datei mit den XML-Daten kann man dem *XmlTextReader* beispielsweise auch einen URL oder ein *Stream*-Objekt übergeben. Für das Öffnen einer XML-Datei über einen URL schreibt man einfach:

```
reader = new XmlTextReader (http:\\myserver.com\Adressbuch.xml");
```

Bei Verwendung einer DOM-basierten Bearbeitung der XML-Daten kann man sich anhand des DOM-Objektmodells durch das Dokument navigieren. Das Beispiel aus Listing 4.17 erstellt eine ähnliche Darstellung wie das vorherige Beispiel. Allerdings wird in diesem Fall über das DOM die Baumstruktur des Dokuments ausgelesen und ausgegeben.

```
XmlTextReader reader = null;
reader = new XmlTextReader (Server.MapPath("
   Adressbuch.xml"));
XmlDocument myXmlDocument = new XmlDocument();
myXmlDocument.Load(reader);

Response.Write("<XMP>");
DisplayTree(myXmlDocument.DocumentElement);
Response.Write("</XMP>");

reader.Close();

private void DisplayTree(XmlNode node)
{
   if (node != null)
      Format(node);
   if (node.HasChildNodes)
   {
      node = node.FirstChild;
      while (node != null)
      {
         DisplayTree(node);
         node = node.NextSibling;
      }
   }
}

private void Format(XmlNode node)
{
   if (!node.HasChildNodes && node.Attributes==null)
   {
      Response.Write("\t" + node.Name + "<" +
         node.Value + ">\n");
   }
   else
   {
      Response.Write(node.Name);
      if (XmlNodeType.Element == node.NodeType)
      {
         XmlNamedNodeMap map = node.Attributes;
         foreach (XmlNode attrnode in map)
            Response.Write(" " + attrnode.Name + "<" +
               attrnode.Value + "> ");
      }
      Response.Write("\n");
   }
}
```

Listing 4.17: Lesen und Navigieren eines XML-Dokuments mit XmlDocument

Das *XmlDocument*-Objekt wird über einen *XmlTextReader* mit dem Dokumenteninhalt gefüllt. Von der Wurzel aus (Eigenschaft *DocumentElement*) wird dann über die rekursiv arbeitende Hilfsroutine `DisplayTree` die Struktur des Dokuments gelesen und entsprechend formatiert ausgegeben. Wenn ein Knoten weitere Kindknoten besitzt, dann wird die Routine rekursiv aufgerufen und so lange abgearbeitet, bis keine Kindknoten mehr existieren. Anhand der Baumstruktur kann man den Mehraufwand bzgl. Speicherbedarf bei Verwendung des DOM-Ansatzes erahnen.

4.3.4 Schreiben von XML-Dokumenten

Das Erzeugen von XML-Dokumenten kann ebenfalls über zwei Wege erfolgen: Unter Verwendung von *XmlTextWriter* oder über *XmlDocument*. Das Beispiel in Listing 4.18 zeigt das Erstellen eines Adressbucheintrags in eine XML-Datei. Über die Methode *WriteStartElement* wird ein XML-Tag eröffnet und über die symmetrische Methode *WriteEndElement* wird das gleiche Tag geschlossen. Elemente werden über die Methode *WriteElementXXX* geschrieben, wobei *XXX* durch verschiedene Zeichenketten ersetzt werden kann. Im Beispiel wird *WriteElementString* verwendet, um eine Zeichenfolge in das XML-Dokument zu schreiben. So gibt es für jedes XML-Fragment eine eigene Methode zum Schreiben von Werten. Es ist wichtig zu beachten, dass jedes öffnende Element auch wieder ein schließendes Element besitzt. Als Faustregel muss die Anzahl der öffnenden gleich der Anzahl der schließenden Elemente sein. Durch die Aufrufe von *Flush* und *Close* auf das *XmlTextWriter*-Objekt wird der Schreibvorgang abgeschlossen und der Inhalt der Puffers in die Datei geschrieben.

```
XmlTextWriter w = new XmlTextWriter(
   Server.MapPath("Adressbuch_XML.xml"), null);

w.WriteStartDocument();
w.WriteStartElement("AdressBuch");
w.WriteStartElement("Eintrag");
w.WriteElementString("Vorname", "Christian");
w.WriteElementString("Nachname", "Weyer");
w.WriteStartElement("Adresse");
w.WriteElementString("Strasse", "Buchallee");
w.WriteElementString("Hausnummer", "7");
w.WriteElementString("Ort", "Musterbach");
w.WriteElementString("PLZ", "97865");
w.WriteEndElement();
w.WriteStartElement("Telefon");
w.WriteStartElement("Festnetz");
w.WriteAttributeString("Vorwahl", "0234");
w.WriteAttributeString("Durchwahl", "98765");
w.WriteEndElement();
w.WriteStartElement("Mobil");
w.WriteAttributeString("Vorwahl", "0179");
```

```
w.WriteAttributeString("Durchwahl", "987654");
w.WriteEndElement();
w.WriteEndElement();
w.WriteEndElement();

w.Flush();
w.Close();
```

Listing 4.18: Erstellen eines XML-Dokuments mit XmlTextWriter

Ähnlich wie beim Lesen und Verarbeiten von XML-Daten kann man auch bei deren Erzeugung den DOM-Ansatz verwenden. Der Auszug aus einem Beispiel in Listing 4.19 zeigt die Verwendung von *XmlDocument*, um neue Knoten zu einem bereits existierenden Dokument hinzuzufügen. Da es sich bei DOM um eine Baumstruktur handelt, müssen im Code immer wieder neue Knoten hinzugefügt werden. So wird z.B. der Knoten `Adresse` dem Knoten `Eintrag` untergeordnet und hinzugefügt (über die Methode *AppendChild*). Einem Elementknoten kann man einen Wert zuweisen, indem man die Eigenschaft *InnerText* setzt. Nachdem man den internen DOM-Baum aufgebaut hat, muss man dieses XML-Fragment noch an den Wurzelknoten des ursprünglichen Dokuments mithilfe der folgenden Zeile anhängen:

```
XmlElement elRoot = xmlDoc.DocumentElement;
elRoot.AppendChild(elEintrag);
```

Abschließend wird das Ergebnis über die Methode *Save* von *XmlDocument* abgespeichert.

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.Load(Server.MapPath(inFileName));
XmlElement elEintrag = xmlDoc.CreateElement("Eintrag");

XmlElement elVorname = xmlDoc.CreateElement("Vorname");
elVorname.InnerText = "Pauline";
XmlElement elNachname = xmlDoc.CreateElement(
   "Nachname");
elNachname.InnerText = "Weyer";
elEintrag.AppendChild(elVorname);
elEintrag.AppendChild(elNachname);

XmlElement elAdresse = xmlDoc.CreateElement("Adresse");
XmlElement elStrasse = xmlDoc.CreateElement("Strasse");
elStrasse.InnerText = "Mustergasse";
XmlElement elHausnummer = xmlDoc.CreateElement(
   "Hausnummer");
elHausnummer.InnerText = "1";
XmlElement elOrt = xmlDoc.CreateElement("Ort");
elOrt.InnerText = "Nixdorf";
XmlElement elPLZ = xmlDoc.CreateElement("PLZ");
```

```
elPLZ.InnerText = "13579";
elAdresse.AppendChild(elStrasse);
elAdresse.AppendChild(elHausnummer);
elAdresse.AppendChild(elOrt);
elAdresse.AppendChild(elPLZ);

XmlElement elTelefon = xmlDoc.CreateElement("Telefon");
XmlElement elFestnetz = xmlDoc.CreateElement(
   "Festnetz");
elFestnetz.SetAttribute("Vorwahl", "01263");
elFestnetz.SetAttribute("Durchwahl", "12345");
XmlElement elHandy = xmlDoc.CreateElement("Handy");
elHandy.SetAttribute("Vorwahl", "0179");
elHandy.SetAttribute("Durchwahl", "987654");
elTelefon.AppendChild(elFestnetz);
elTelefon.AppendChild(elHandy);

elEintrag.AppendChild(elAdresse);
elEintrag.AppendChild(elTelefon);

XmlElement elRoot = xmlDoc.DocumentElement;
elRoot.AppendChild(elEintrag);
xmlDoc.Save(Server.MapPath(outFileName));
```

Listing 4.19: Erstellen eines XML-Dokuments mit XmlDocument

4.4 Zusammenfassung

Das große Thema Datenverwaltung kann in .NET und ASP.NET auf unterschiedliche Art und Weise angegangen werden. Egal, ob man mit herkömmlichen relationalen Datenbankmanagementsystemen oder mit strukturierter Information auf Basis von XML arbeitet, das .NET Framework hat eine Lösung für den Zugriff und zur Bearbeitung aller Daten parat.

ADO.NET hat im Vergleich zum altbewährten ADO merklich an Funktionalität gewonnen, hat aber in der aktuellen Version auch noch seine Kinderkrankheiten. Die Einführung des *DataSet*-Objekts als XML-basierter Datenbehälter auch für die Arbeit ohne Datenbank im Hintergrund wird mit Sicherheit von vielen sehr geschätzt werden.

Vor allem die breite Unterstützung von XML-Standards und die Integration mit ADO.NET macht die Arbeit mit den XML-Schnittstellen komfortabel und sichert außerdem die Investitionen für die Zukunft. Im weiteren Verlauf werden wir noch des Öfteren auf XML stoßen, vor allem wenn es um die Arbeit mit XML Web Services auf der .NET Plattform geht.

5 Öffentlicher Dienst

XML Web Services mit ASP.NET

5.1 Einführung

Der Titel dieses Buchs enthält den Begriff *XML Web Services*. In Kapitel 1 wurde bereits auf die neuen Möglichkeiten und das Paradigma von XML-basierten Web Services hingewiesen. In diesem Kapitel werde ich Ihnen nun sowohl die grundlegenden Techniken und Standards näher bringen als auch eine mögliche Implementierung von XML Web Services-Anwendungen mit ASP.NET vorstellen. Web Services stehen dabei in ihrer Entwicklung und mit ihrem Potenzial noch am Beginn eines langen Wegs. Dieser Weg besitzt viele Hindernisse und Falltüren, mit denen sich ein Entwickler zumindest momentan noch auseinandersetzen muss. Damit aber die Arbeit mit Web Services so einfach und effektiv wie möglich gestaltet werden kann, bedarf es eines gut ausgestatteten Frameworks, welches dem Programmierer die wichtigsten Aufgaben abnimmt. ASP.NET bietet mit seiner Web Services-Funktionalität ein solches Rahmenwerk.

5.2 Web Services-Überblick

Bevor wir die Vorgehensweise in ASP.NET zur Erstellung und Nutzung von XML Web Services erkunden, wollen wir den Blick auf eine kompakte Einführung in die Welt der Web Services werfen. Die Technologien und Standards hinter Web Services werden anhand einer allgemeinen Betrachtung des Sinns und Unsinns von dienstbasierten Angeboten im WWW beleuchtet. Wie wir sehen werden, ist es nicht immer sinnvoll, Web Services als neue Architektur für Anwendungen zugrunde zu legen. Es muss je nach Applikation und Rahmenbedingungen sorgfältig abgewogen werden.

5.2.1 Was sind Web Services und warum werden sie eingesetzt?

Wie bereits im ersten Kapitel angesprochen, bringt die Entwicklung des Internet und des WWW im Speziellen in den letzten Jahren auch eine notwendige Veränderung der Denkansätze in Bezug auf Software und Architekturen mit sich. Durch den Aufschwung des Internet und auch der Intranets ergeben sich zwar neue Chancen und Möglichkeiten, Geschäfte zu

machen und Probleme zu lösen, die man bisher nicht Angriff nehmen wollte oder konnte. Aber durch diese neuen Perspektiven wird man als Manager oder Entwickler auch sehr schnell mit den Grenzen aktueller Technik und Geschäftsmodelle konfrontiert. Prinzipiell lassen sich zusätzlich zu der in Kapitel 1 geschilderten Situation vier große Problembereiche ausmachen, die im Folgenden kurz beleuchtet werden (vgl. Abbildung 5.1).

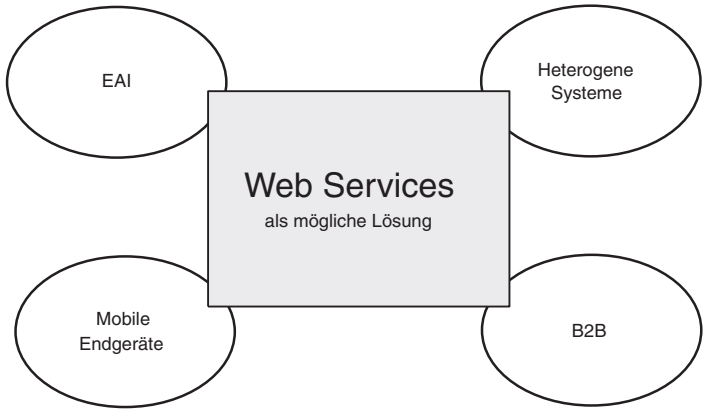


Abbildung 5.1: Web Services als mögliche Lösung für Problembereiche im Internet

Zu diesen Problemen zählt das Themengebiet der *Enterprise Application Integration* (*EAI*). Bei EAI handelt es sich um eine Vorgehensweise, bei der die unterschiedlichen Applikationen und Datenbestände in einem Unternehmen oder in einem Konzern in Teilen oder komplett über eine oder wenige Endanwendungen den Mitarbeitern im Unternehmen zugänglich gemacht werden sollen. Im Zuge der »Internetisierung« der Firmen wurde der Wunsch und die Notwendigkeit immer größer, auch Endkunden und Handelspartner mit in diese Infrastruktur einzubeziehen. Die Kunden sollten beispielsweise Zugriff auf einen bestimmten Datenbestand und eine Programmlogik über einen Standard-WWW-Browser erlangen. Die Zulieferer und Partner hingegen könnten durch standardisierte Schnittstellen und Protokolle in einem automatisierten Prozess in bestimmte Geschäftsszenarien integriert werden. Hier gibt es zwar heute bereits Lösungen, jedoch ohne große Akzeptanz bezüglich offen definierter Standards.

Eng verbunden mit der Situation des EAI ist der Wunsch nach einer homogenen Kommunikationsmöglichkeit zwischen Unternehmen im Bereich *Electronic Commerce* (*E-Commerce*). Dieser als *Business-to-Business* (*B2B*) bekannt gewordene Ansatz soll den Weg von der »Copy-und-Paste-Transaktion« hin zu einem automatisierten, nicht von Menschen initiierten oder kontrollierten Geschäftsvorgang realisieren. Auch hier bedarf es einheitlicher Schnittstellen, die bis dato zwar existieren, aber keine echte Interoperabilität zwischen heterogenen Lösungen zulassen.

Zu den beiden oben dargelegten Problemen auf Geschäftsprozessebene gesellen sich spätestens bei deren Implementierung die Unwägbarkeiten unterschiedlicher Plattformen und Systeme. In fast jeder Firma gibt es ent-

weder aus historischen oder aus politischen Gründen eine Vielzahl von eingesetzten Betriebssystem-, Entwicklungsplattform- und Protokollversionen. In dieser heterogenen Welt muss auf technischer Ebene eine Lösung zur Integration der Plattformen gefunden werden. Dies wird umso wichtiger, je mehr die Integration der Unternehmen untereinander über Firmen- und somit auch Firewallgrenzen hinweg stattfindet. Die vorliegenden Konzepte und Technologien für Middlewaresysteme sind hier nicht mehr ausreichend. Es muss ein alles umfassendes Konzept her.

Nicht zuletzt wird die Vision und zukünftige Realität der mobilen Informationsverarbeitung durch portable, intelligente Endgeräte die Gesamtsituation nicht verbessern. Durch immer mehr mobile *Devices* müssen immer mehr Daten und Anwendungen über die Firmengrenzen hinweg zur Verfügung gestellt werden. Abgesehen von den hardwareseitigen Voraussetzungen des Internetzugangs sollen neue Architekturkonzepte das Motto *»Information any time, any place on any device«* in den nächsten Jahren verwirklichen. Das transparente Einbeziehen der mobilen Benutzer in das Firmennetz wird über bestehende Technologien nur schwer erreichbar sein – also ist auch hier eine geeignete Lösung gefragt.

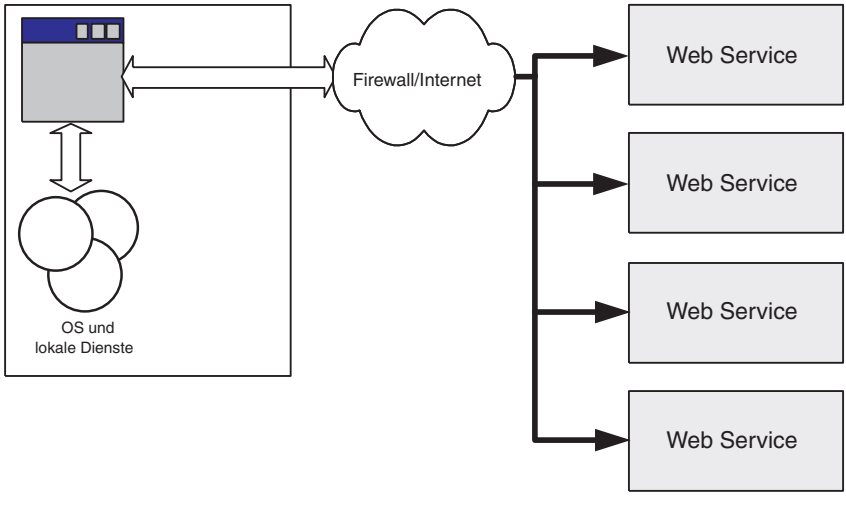


Abbildung 5.2: Web Services als »Komponenten für das WWW«

Eine mögliche Lösung für die geschilderten Probleme und Aufgabenstellungen könnte das Konzept der *Web Services* sein. Web Services lassen sich einfach als Softwarekomponenten definieren, die über Beschreibungsstandards und Standardprotokolle im Internet zugänglich sind. Die Metapher der Komponenten für das WWW passt hier vielleicht am besten (siehe Abbildung 5.2). Während heute eine Anwendung vornehmlich mit dem lokalen Betriebssystem und lokal (also im Intranet) verfügbaren Diensten und Komponenten arbeitet, wird sich in Zukunft vieles im Internet abspielen. Gewisse Dienstleistungen werden in Form von öffentlich zugänglichen Web Services realisiert. Dies hat z.B. den Vorteil, dass man die Dienste nicht mehr im Unternehmen

betreiben und pflegen oder die Softwarekomponente lokal installieren muss. Natürlich müssen für die Erfüllung dieser Vision zunächst die kommunikationstechnischen Grundlagen wie mobile Zugänge und kostengünstige Tarife geschaffen werden. Doch dies ist hoffentlich nur eine Frage der Zeit. Web Services bieten sich darüber hinaus aber auch gut für eine lose gekoppelte Infrastruktur innerhalb des Intranets an. Schließlich wurden sie zwar grundsätzlich für die Anwendung im Internet konzipiert, können aber genauso gut ihre Dienste in weit verteilten internationalen Firmennetzen zur Verfügung stellen.

Web Services werden oft auch als *XML Web Services* bezeichnet. Dies rührt daher, dass die ursprünglichen Ansätze und Standards alle auf der *eXtensible Markup Language* beruhen. Das Konzept der XML Web Services stützt sich auf XML als Beschreibungssprache für die zu übertragenden Daten und HTTP als Basiskommunikationsprotokoll.

Ich werde Ihnen in den nächsten Abschnitten die Grundlagen für XML Web Services vorstellen und erklären, wo die Vor- und Nachteile dieses Ansatzes liegen. Der Einsatz von Web Services ist hierbei nicht immer die beste Lösung für jede Problemstellung. Wenn man z.B. binäre Daten wie Bilder oder gar Videoströme übertragen muss, dann ist eine Implementierung auf Basis von XML Web Services nicht sinnvoll.

Wenn Sie bereits mit den Themen XML, SOAP und WSDL vertraut sind, können Sie diese theoretischen und teilweise etwas trockenen Abschnitte getrost überspringen. Ab Abschnitt 5.3 beschreibe ich dann, wie XML Web Services in Verbindung mit ASP.NET funktionieren.

In diesem Kapitel werde ich ausschließlich die XML-basierten Web Services beschreiben, da sich mit der ASP.NET-Technologie nur XML Web Services realisieren lassen. In Kapitel 8 wird dann mit .NET Remoting eine Architektur und eine Infrastruktur vorgestellt, mit der man auch Web Services implementieren kann, die nicht auf XML basieren.

5.2.2 XML und Co.

Die eXtensible Markup Language (XML) gilt in der IT-Landschaft als einer der wichtigsten Standards und Entwicklungen der letzten Jahre. Durch XML können Daten strukturiert und mit Bezeichnerelementen versehen werden. XML-Daten oder -Dokumente liegen in einer textuellen Form vor, welches die Verarbeitung auch durch den Menschen ermöglicht. XML wird vor allem für den Austausch von Daten und Dokumenten zwischen Plattformen, Applikationen und Organisationen eingesetzt. Ein Entwickler oder Designer kann ein XML-Dokument nach Belieben gestalten, hier gibt es keinerlei strukturelle Vorgaben. Ein Dokument kann gemäß der XML-Spezifikation aus mehreren syntaktischen Konstrukten bestehen. Diese Konstrukte sowie weiterführende Erweiterungen bzw. Ergänzungen zur XML-Spezifikation werden im Folgenden erklärt.

Elemente und Attribute

Ein XML-Dokument beginnt mit einer XML-Deklaration – dies ist allerdings nicht zwingend vorgeschrieben. Eine mögliche XML-Deklaration kann folgendermaßen aussehen:

```
<?xml version='1.0' encoding='UTF-8'?>
```

Der eigentliche Inhalt von XML-Dokumenten besteht aus so genannten Elementen. Diese Elemente definieren die grundlegende Struktur eines Dokuments. Ein Dokument hat genau ein Element höchster Stufe, welches Dokumentenelement genannt wird. Ein Element kann beliebig viele Kindelemente haben. Das folgende kleine Beispiel zeigt einen Auszug aus einem XML-Dokument, hierbei ist ein Element ohne und ein anderes mit Kindelementen zu sehen.

```
<Teilnehmer>Bill Gates</Teilnehmer>
<Teilnehmer>
   <Name>Weyer</Name>
   <Vorname>Christian</Vorname>
</Teilnehmer>
```

Wie man aus dem obigen Beispiel ersehen kann, werden Elemente innerhalb eines XML-Dokuments über die so genannten Kennzeichner (*Tags*) markiert. Ein Tag ist der Name für ein Element umschlossen mit den Symbolen < und >. Man unterscheidet prinzipiell zwischen einem Start- und einem Ende-Tag. Das Ende-Tag beginnt im Gegensatz zum Start-Tag mit der Zeichenfolge </. Ein Spezialfall für ein Tag ist das leere Element. Es kann in zwei Darstellungsarten verwendet werden. Das folgende Beispiel zeigt die beiden Möglichkeiten, ein leeres Element zu notieren:

```
<Teilnehmer></Teilnehmer>
<Teilnehmer/>
```

Zusätzlich zur Verwendung von Elementen lässt die XML-Spezifikation auch den Einsatz von Attributen zu. Attribute können verwendet werden, um Metadaten eines Elements zu beschreiben. Attribute sollen somit zusätzliche Informationen über das Element liefern, auf welches sie angewendet werden. Attribute werden durch Name/Wert-Paare dargestellt, wie folgendes Beispiel zeigt:

```
<Teilnehmer Nummer='7' Kategorie='SportartenUmfrage'>
   <Name>Weyer</Name>
   <Vorname>Christian</Vorname>
<Teilnehmer>
```

Es ist eine bekannte Fragestellung in der XML-basierten Modellierung, ob man nun für einen bestimmten Sachverhalt Elemente bzw. Elementhierarchien oder Attribute verwendet. Diese Frage lässt sich meistens nur abhängig vom jeweiligen Anwendungsfall beantworten.

Außer den vorgestellten Elementen und Attributen können in einem XML-Dokument noch mehrere andere Entitäten auftreten. Hierzu zählen Kommentare, Verarbeitungsanweisungen (*Processing Instructions*) und *CDATA*-Blöcke. Letztere sind beispielsweise notwendig, um Text mit gewissen speziellen Zeichen oder einer vorgegebenen Formatierung innerhalb eines XML-Elements einbetten zu können.

Namensräume

Da XML-Dokumente von beliebigen Personen erstellt werden können und diese Personen auch noch freie Hand bei der Benennung ihrer Elemente haben, kann es häufig zu einer Zweideutigkeit innerhalb eines Dokuments kommen. Man stelle sich eine Situation vor, in der ein XML-Dokument den Inhalt eines anderen, fremden XML-Dokuments umfasst. Dann kann es vorkommen, dass Autor A die gleichen Tag-Namen verwendet wie Autor B. Dieses Dilemma muss zugunsten der Eindeutigkeit von Tags verhindert werden. Hierfür existieren die XML-Namensräume (*XML Namespaces*).

XML-Namensräume werden durch die Angabe eines *URI* (*Uniform Resource Identifier*) festgelegt. Ein URI muss aber nicht auflösbar sein, sondern dient lediglich der Gewährleistung der Eindeutigkeit. Der Name des Namensraums und der lokale Elementname ergeben den so genannten qualifizierten Namen. Das folgende XML-Fragment zeigt unser Beispiel von oben mit qualifizierten und nicht-qualifizierten Elementen:

```
<cw:Teilnehmer xmlns:cw='urn:eyesoft-de:Teilnehmer'>
   <Name>Weyer</Name>
   <Vorname>Christian</Vorname>
</cw:Teilnehmer>
```

Ein Namensraum wird also über ein Präfix an einen Elementnamen gebunden. Im obigen Beispiel sind die Kindelemente von `Teilnehmer` nicht qualifiziert. Außer explizit benannten Namensräumen kann man auch noch einen standardmäßigen Namensraum für ein Element angeben. Mit der folgenden Anweisung wird dem Element `Teilnehmer` ein Standardnamensraum zugewiesen, der nicht mehr explizit durch ein Präfix angegeben werden muss.

```
<Teilnehmer xmlns='urn:eyesoft-de:Teilnehmer'>
   <Name>Weyer</Name>
   <Vorname>Christian</Vorname>
</Teilnehmer>
```

Ein wesentlicher Unterschied zwischen der Deklaration eines Standardnamensraums und der expliziten Angabe von Namensräumen ist, dass die Kindelemente in einem an den Standardnamensraum gebundenen Element diesen Namensraum erben.

XML Schema

XML alleine reicht noch nicht aus, um einen reibungslosen Datenaustausch realisieren und garantieren zu können. Aufgrund der Wahlmöglichkeiten eines XML-Entwicklers bezüglich der Struktur, der Namen von Tags und der

erlaubten Werte von Elementen kann es viele unterschiedliche XML-Dokumentausprägungen geben, die aber inhaltlich das Gleiche beschreiben. Für die Sicherstellung einer syntaktischen und semantischen Korrektheit ist eine zusätzliche Technologie notwendig. In den ersten Jahren des Erfolgs von XML war für die Validierung von XML-Dokumenten die so genannte *DTD* zuständig. DTD steht für *Document Type Definition* und ist eine nicht auf XML-basierende Sprache – die zudem nicht erweiterbar ist - zur Überprüfung von XML-Dokumenten anhand einer vorgegebenen Struktur, die in einem DTD beschrieben wird.

Im Mai 2001 wurde DTD endgültig vom neuen, auf XML aufsetzenden Standard der *XML Schemas* abgelöst. Der Einsatz von XML Schema erlaubt folgende Überprüfungen:

- Die Struktur von Elementen und Attributen,
- Die Reihenfolge der Elemente,
- Die Datenwerte der Elemente und Attribute, abhängig von Wertbereichen, Aufzählungen und *Pattern Matching*,
- Die Eindeutigkeit der Werte.

Über diese Funktionalität hinaus kann man XML Schema auch als Vertragsmöglichkeit zwischen Programmen oder Unternehmen betrachten. Diesen Ansatz verfolgt Microsoft beispielsweise mit seiner BizTalk-Initiative.

Das unten stehende Beispiel-Schema kann dafür verwendet werden, zu überprüfen, ob ein XML-Dokument unseren Vorstellungen des Datentyps `Teilnehmer` entspricht. XML Schema definiert eigene Datentypen, die über alle verfügbaren Plattformen und Programmiersprachen hinweg ihre Entsprechung haben sollten. Dabei kann man auf die eingebauten, einfachen Datentypen verweisen, kann aber auch eigene, komplexe Typen erzeugen und diese in einem Schema festlegen. Der unten stehende Code definiert einen komplexen Typ für `Teilnehmer`.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="CW_Teilnehmer"
   targetNamespace="urn:eyesoft-de:Teilnehmer"
   xmlns:mstns="urn:eyesoft-de:Teilnehmer"
   xmlns="urn:eyesoft-de:Teilnehmer"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
   attributeFormDefault="qualified"
   elementFormDefault="qualified">
   <xs:element name="Teilnehmer" msdata:Prefix="cw">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="Name" form="unqualified"
               type="xs:string" minOccurs="1" />
            <xs:element name="Vorname"
```

```
            form="unqualified" type="xs:string"
            minOccurs="1" />
        </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Zu Beginn des Dokuments (übrigens oft mit der Dateiendung *.xsd* versehen) werden verschiedene Namensräume definiert. Das Herzstück ist allerdings der Abschnitt, in dem der komplexe Typ festgelegt wird. Anhand dieser XML-Beschreibung könnte man nun beispielsweise in Java oder in Perl eine Klasse erstellen, die den Anforderungen des XML Schemas genügt und könnte so eine plattformübergreifende Konsistenz der Typen gewährleisten (vorausgesetzt, die Umwandlung von XML + Schema in die Programmiersprache und vice versa ist korrekt implementiert).

Mit diesem Handwerkszeug aus dem XML-Umfeld können wir nun den nächsten Schritt wagen, und die Basis-Standards und -Technologien im XML Web Services-Umfeld betrachten.

5.2.3 SOAP

Stellen Sie sich vor, Sie suchen nach dem heiligen Gral, finden ihn aber nicht. Allerdings sind alle Mittel und Hilfestellungen für eine erfolgreiche Suche schon vorhanden, Sie müssen sie nur richtig und sinnvoll in Zusammenhang bringen und benutzen. So ähnlich war es bei der Entwicklung des *Simple Object Access Protocols* (*SOAP*).

SOAP ist der Basisstandard für die Kommunikation im Internet nach dem Muster der Web Services. Einfach gesprochen ist SOAP in seiner heute in der Industrie zu findenden Ausprägung ein spezieller XML-Dialekt, der über das HTTP-Protokoll zwischen den Kommunikationsparteien übertragen wird. Dieser Umstand hat sich in den letzten Monaten herauskristallisiert, ist aber nicht in Stein gemeißelt, da die ursprüngliche SOAP-Spezifikation wesentlich mehr Möglichkeiten offen lässt. Ich werde Ihnen in den folgenden Abschnitten die wichtigsten Eigenschaften von SOAP erklären und gleichzeitig auf mögliche Probleme und Fallstricke in einer Real-World-Implementierung hinweisen.

Die SOAP-Spezifikation definiert ein Rahmenwerk für die Übertragung von XML-Nachrichten über ein standardisiertes und anerkanntes Transportprotokoll. Eine SOAP-Nachricht befindet sich innerhalb des so genannten SOAP *Envelopes*, der gewissermaßen den Umschlag für unsere Nachricht repräsentiert. In der aktuellen Version 1.1 definiert die SOAP-Spezifikation die grundlegende Gestalt eines Envelopes wie in Abbildung 5.3 dargestellt.

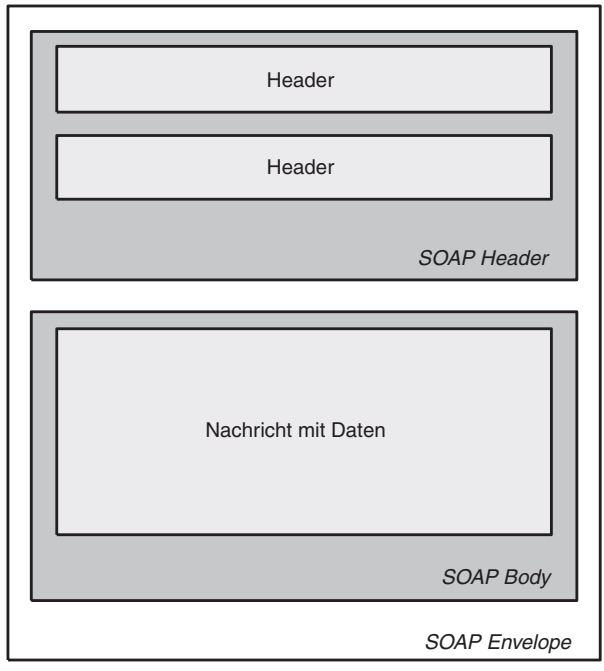


Abbildung 5.3: SOAP Envelope gemäß SOAP-Spezifikation V1.1

Envelope

Die eigentliche SOAP-Nachricht ist also der SOAP Envelope. Damit eine SOAP-Nachricht auch wirklich versendbar ist, benötigt man noch eine Bindung an ein Transportprotokoll, aber darauf kommen wir weiter unten zu sprechen. Der Envelope definiert den Rahmen für eine Nachricht und besteht aus zwei Subelementen: dem *Header*-Teil und dem SOAP *Body*. Die SOAP Header sind optional, wohingegen der SOAP Body immer in einem SOAP Envelope vorhanden sein muss. SOAP baut stark auf den Einsatz von XML-Namensräumen, um eine eindeutige Benennung der einzelnen Elemente innerhalb einer Nachricht gewährleisten zu können. Daher werden im allgemeinen Envelope-Abschnitt vor allem die verwendeten Namensräume festgelegt. Listing 5.1 zeigt einen unvollständigen SOAP Envelope (man beachte, dass das Element `Envelope` heißen muss).

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/
      soap/envelope/">
   [SOAP Header]
   [SOAP Body]
</SOAP-ENV:Envelope>
```

Listing 5.1: Unvollständiger SOAP Envelope

Schauen wir uns nun die beiden Hauptelemente im SOAP Envelope etwas genauer an.

Header

Die eigentlichen Nutzdaten in einem Envelope sind im SOAP Body enthalten. Es gibt aber eine Vielzahl von Situationen – vor allem in zukünftigen Szenarien – welche die Erweiterung des Envelopes um zusätzliche Daten und Informationen erfordern. Hierfür gibt es die so genannten SOAP Header. Ein Beispiel für die Verwendung von SOAP Headern ist die Übertragung von Benutzerdaten (Benutzername und Passwort) für eine SOAP-Anfrage. Weiterhin könnte man in einem SOAP Header beispielsweise eine Transaktions-ID übertragen, um eine eindeutige Zuordnung der Nachricht zu einer bestimmten Transaktion zu erreichen.

Das Beispiel in Listing 5.2 zeigt einen exemplarischen SOAP Header mit Benutzerdaten. Die Daten in diesem SOAP Header sind mit einem eigenen XML-Namensraum ausgestattet.

```
<SOAP-ENV:Header>
   <sec:credentials
      xmlns:sec="http://schemas.xmlsoap.org/ws/
      2002/01/sec">
      <sec:userName>CWeyer</sec:userName>
      <sec:userPassword>w3efr56z</sec:userPassword>
   </sec:credentials>
</SOAP-ENV:Header>
```

Listing 5.2: SOAP Header mit Benutzerdaten

Soll eine SOAP-fähige Applikation einen bestimmten SOAP Header unbedingt abarbeiten, dann kann man dem Header das Attribut *mustUnderstand* hinzufügen. Das obige Beispiel würde dann folgendermaßen aussehen:

```
<SOAP-ENV:Header>
   <sec:credentials
      xmlns:sec="http://schemas.xmlsoap.org/ws/
      2002/01/sec"
      mustUnderstand="1">
      <sec:userName>CWeyer</sec:userName>
      <sec:userPassword>w3efr56z</sec:userPassword>
   </sec:credentials>
</SOAP-ENV:Header>
```

Durch dieses zusätzliche Attribut wird der Empfänger der SOAP-Nachricht gewissermaßen gezwungen, mit diesem SOAP Header umgehen zu können. Wenn dies nicht der Fall ist, muss eine entsprechende Fehlermeldung erzeugt werden (siehe auch Abschnitt Fehlerbehandlung). Wenn das Attribut nicht vorhanden ist, dann hat dies die gleiche Bedeutung wie `mustUnderstand="0"`.

Body

Die eigentlichen Daten einer SOAP-Nachricht stehen im SOAP Body. Dieser ist im Gegensatz zu den SOAP Headern unbedingt vorgeschrieben und muss daher in jeder SOAP-Nachricht vorhanden sein. Wenn man SOAP als

plattformübergreifenden RPC-Mechanimus (Remote Procdure Call) ansieht, dann stehen im SOAP Body die serialisierten Informationen für den entfernten Methodenaufruf: Methodenname mitsamt Signatur. Serialisierung bedeutet, dass Objekte in ein Format überführt werden, das abgespeichert oder versendet werden kann. Wie diese Parameter codiert sind – also in XML dargestellt werden – wird unten im Abschnitt Datenkodierung genauer beschrieben. Der exemplarische SOAP Body in Listing 5.3 zeigt eine Anfrage an einen Web Service mit der Methode `GetUserID`. Diese Methode akzeptiert einen Eingabeparameter als Zeichenfolge für den Benutzernamen. Bitte beachten Sie, dass wir zu diesem Zeitpunkt noch keine Informationen haben, wie ein SOAP Body aussehen muss. Dies ist nicht die Aufgabe der SOAP-Spezifikation, denn hierfür benötigen wir zusätzliche Informationen, die nicht direkt in das Aufgabengebiet von SOAP fallen.

```
<SOAP-ENV:Body>
   <SOAP:GetUserID xmlns:SOAP=
      'http://eyesoft.de/webservices/'>
      <userName>CWeyer</userName>
   </SOAP:GetUserID>
</SOAP-ENV:Body>
```

Listing 5.3: SOAP Body einer SOAP-Anfrage

Üblicherweise wird SOAP in Web Services-Szenarien für Zwei-Wege-Kommunikation verwendet. Dies bedeutet, dass immer ein Nachrichtenpaar an einer SOAP-Kommunikation beteiligt ist: eine Nachricht für die Anfrage und eine weitere Nachricht für die Antwort vom SOAP-Endpunkt. Einen Vorschlag für die Handhabung eines solchen RPC-Musters mit SOAP ist ebenfalls in der Spezifikation festgehalten. Man sollte jedoch im Hinterkopf behalten, dass SOAP an sich für Ein-Wege-Nachrichten konzipiert wurde.

Wenn in einer an der SOAP-Kommunikation beteiligten Komponente ein Fehler auftritt, dann muss ein SOAP-Prozessor (also ein SOAP-fähiger Client oder Server) einen so genannten SOAP Fault erzeugen. Ein SOAP Fault ist der einzig vordefinierte SOAP Body im Umfeld von SOAP.

Was SOAP nicht ist

In den Monaten seit seinem erstmaligen Auftritt wurden in der Öffentlichkeit viele verwirrende und falsche Information über SOAP verbreitet. In diesem kurzen Absatz möchte ich mit den größten Irrtümern über SOAP aufräumen.

SOAP ist kein Ersatz für DCOM & Co. Verteilte Middleware-Architekturen wie DCOM oder CORBA sind komplexe Frameworks mit Fähigkeiten zur Auflösung von Objektreferenzen, zur Verwaltung der Lebenszeit von Objekten und Ressourcen in einer verteilten Umgebung (*distributed garbage collection*) oder aber zur Übergabe von Objekten per Referenz.

SOAP ist auch nicht dafür zuständig, wie das jeweilige Zielobjekt auf dem Server aktiviert und angesprochen wird. Dies bleibt einzig und allein dem Implementierer des SOAP-Stacks für die jeweilige Plattform und Technologie überlassen. SOAP ist einfach eine Konvention, wie man mit XML Daten und Informationen zwischen Kommunikationspartnern auf technischer Ebene über ein standardisiertes Transportprotokoll übertragen kann – und ist somit eher als Ersatz für DCOMs ORPC oder CORBAs IIOP zu sehen. Alles andere ist Gegenstand zukünftiger Standards und Frameworks.

Fehlerbehandlung

In jeder Anwendung treten Fehler oder Programmausnahmen auf. Speziell für diesen Fall definiert die SOAP-Spezifikation einen standardmäßigen SOAP Body namens SOAP Fault. Ein SOAP Fault enthält einen Fehler-Code und eine Fehlerbeschreibung. Diese beiden Elemente `faultcode` und `faultstring` sind zwingend vorgeschrieben. Zusätzlich kann ein Entwickler noch genauere Informationen über den jeweiligen Fehler mithilfe des `detail`-Element angeben. Dieses Element muss vorhanden sein, wenn sich der Fehler auf die Abarbeitung des SOAP Body-Elements bezieht – in jedem anderen Fall ist das Vorkommen des `detail`-Elements nicht vorgeschrieben. Das SOAP Fault-Beispiel in Listing 5.4 zeigt einen vollständigen SOAP Fault, der einen Ausnahmefehler in der Server-seitigen Abarbeitung der SOAP-Anfrage repräsentiert.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:Server</faultcode>
         <faultstring>Server Error</faultstring>
         <detail>
            <e:myfaultdetails
               xmlns:e="urn:eyesoft-de:SOAPFault">
               <message>
               Ausnahmefehler in XYZ aufgetreten!
               </message>
               <errorcode>4711</errorcode>
            </e:myfaultdetails>
         </detail>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 5.4: Fehlerübermittlung mittels SOAP Fault

Wie man im Beispiel sieht, kann das `detail`-Element beliebige zusätzliche Informationen über den aufgetretenen Fehler enthalten.

Detaillierte Beschreibungen von Fehlern bei der Abarbeitung von SOAP Headern dürfen übrigens nicht im `detail`-Element von SOAP Fault zurückgegeben werden, sondern müssen ebenfalls über SOAP Header propagiert werden.

Im Übrigen muss bei der optionalen HTTP-Bindung (siehe Abschnitt weiter unten) immer der Fehlerstatus 500 (interner Server-Fehler) im Falle eines SOAP Faults zurückgegeben werden.

Datenkodierung

Wie sehen meine Daten eigentlich aus? Damit Anwendungen in einer standardisierten Art und Weise über XML miteinander kommunizieren können, bedarf es Regeln, wie die Daten und Datentypen als XML verpackt werden müssen. Den Vorgang der Umwandlung von Daten in eine XML-Repräsentation nach festgelegten Regeln bezeichnet man auch als *Encoding*. Diese Regeln werden in der SOAP-Spezifikation angegeben.

Die Art und Weise wie Daten kodiert werden, kann über das Attribut `encodingStyle` für ein XML-Element innerhalb einer SOAP-Nachricht angegeben werden. Für die in der SOAP-Spezifikation (Abschnitt 5) festgelegte Kodierung muss dieses Attribut den Wert *http://schemas.xmlsoap.org/soap/encoding/* besitzen. Hinter diesem URL verbirgt sich im Übrigen ein XML Schema, welches die Kodierungs-Regeln nach Abschnitt 5 beschreibt. Dieses Schema kann dafür verwendet werden, SOAP-Nachrichten auf ihre Gültigkeit bezüglich der Kodierung zu überprüfen. Wenn dieses Attribut auf ein Element angewendet wird, dann gilt es für dieses Element und alle Nachkommen – bis unter Umständen ein neuer Wert für `encodingStyle` gesetzt wird.

SOAP unterstützt eine Reihe von Datentypen. Diese Typen sind die gleichen wie sie in der XML Schema (XSD) Spezifikation, Teil 2 definiert sind. Neben den einfachen Datentypen von XSD kann man auch zusammengesetzte (oder komplexe) Typen definieren und übertragen. Tabelle 5.1 zeigt die wichtigsten einfachen Typen für den Einsatz in SOAP.

XSD Datentyp	Beispiel
int	7
float	7,4
string	Dies ist eine Zeichenkette
boolean	true

Tab. 5.1: Einfache Datentypen in SOAP

Ein einfaches Beispiel für die Übertragung einer Zeichenfolge (String) kann man in Listing 5.3 sehen. Hier ist im XML-Dokument nicht angegeben, um welchen Datentyp es sich genau handelt. Dies lässt die Spezifikation offen.

Es ist nämlich optional möglich auch über einen XSD-Datentypkennzeichner (`xsi:type="Datentyp"`) den Typ des Elementwertes mit anzugeben.

Eine ernsthafte Anwendung kommt aber in der Regel nicht mit den primitiven Datentypen aus. Oft ist es der Fall, dass mehr oder weniger komplexe Objektmodelle einer Programmiersprache in SOAP serialisiert werden müssen, um eine Übertragung zum Kommunikationspartner zu erlauben. Daher bietet die SOAP-Spezifikation eine Festlegung der Codierung von komplexen oder zusammengesetzten Typen (*compound types*). Zusammengesetzte Typen können Arrays, Strukturen oder allgemeine komplexe Typen (entsprechen in etwa Klassen in einer Programmiersprache) sein. Die einzelnen Regeln für die Kodierung hier aufzuführen, würde den Rahmen des Kapitels leider sprengen. Vor allem im Bereich der Arrays gibt es viele unterschiedliche Typen, von denen jeder eigene Regeln befolgen muss. Um ein Gespür für eine SOAP-Nachricht mit einem komplexen Typ zu erhalten, können Sie in Listing 5.5 ein mit SOAP übertragenes Array von Objekten begutachten.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap=
   "http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soap:Body>
      <GetArrayListResponse xmlns="http://tempuri.org/">
         <GetArrayListResult>
            <anyType xsi:type="Address">
               <Street>Bat-Gasse 7</Street>
               <City>Gotham City</City>
               <ZIP>0815</ZIP>
               <Country>Batland</Country>
            </anyType>
            <anyType xsi:type="Address">
               <Street>Robin-Allee</Street>
               <City>Gotham City</City>
               <ZIP>0815</ZIP>
               <Country>Batland</Country>
            </anyType>
         </GetArrayListResult>
      </GetArrayListResponse>
   </soap:Body>
</soap:Envelope>
```

Listing 5.5: Array von Objekten als zusammengesetzter Typ in SOAP

Das Beispiel zeigt, wie die Daten im XML-Dokument codiert werden. Zusätzlich werden in diesem SOAP-Paket noch Werte für `Result` und `Response` angegeben. Diese führen uns direkt zum Thema entfernter Methodenaufruf (RPC) über SOAP gemäß dem Anfrage-Antwort-Muster.

Die SOAP-Spezifikation in der Version 1.1 ist sehr umfangreich und bietet einige Eigenheiten bei der Kodierung von Typen. Bei der Besprechung von ASP.NET XML Web Services werde ich an geeigneter Stelle auf die Beziehung zur Spezifikation des W3C eingehen.

Hier möchte ich noch erwähnen, dass es prinzipiell zwei Arten gibt, Daten und Typen in einem SOAP-XML-Dokument zu platzieren. Im Jargon der Spezifikation heißen diese beiden Alternativen *embedded* und *independent*. Im ersten Fall erscheint das Element als Kindelement eines unabhängigen Elements (wie z.B. ein Element innerhalb eines SOAP Headers oder des SOAP Bodys). Folgendes Beispiel soll dies verdeutlichen (das unabhängige Element muss übrigens per Definition den gleichen Namensraum besitzen wie das eigentliche Methodenelement):

```
<SOAP-ENV:Body>
   <SOAP:GetUserID xmlns:SOAP=
      'http://eyesoft.de/webservices/'>
      <userName href="#str0">CWeyer</userName>
   </SOAP:GetUserID>
   <SOAP:string xmlns:SOAP=
      'http://eyesoft.de/webservices/' id="str0">
      CWeyer
   </SOAP:string>
</SOAP-ENV:Body>
```

Im zweiten Fall erscheint das Element auf der höchsten Stufe als unmittelbares Kindelement von SOAP Header oder Body. Dies ist der Fall, den wir bis jetzt in unseren Beispielen verwendet haben:

```
<SOAP-ENV:Body>
   <SOAP:GetUserID xmlns:SOAP=
      'http://eyesoft.de/webservices/'>
      <userName>CWeyer</userName>
   </SOAP:GetUserID>
</SOAP-ENV:Body>
```

Eng mit diesen beiden Begriffen einher gehen zwei weitere Schlagwörter: *single reference* und *multi reference*. Nehmen wir die beiden obigen Beispiele, dann bedeutet single reference, dass es genau eine Referenz auf die Zeichenfolge `CWeyer` gibt. *Single reference*-Werte treten so gut wie immer als *embedded* Elemente auf. Dieser Ansatz entspricht also einer Parameterübergabe *by value* in einer Programmiersprache. *Multi reference* hingegen ermöglicht eine Darstellung gemäß dem Zeigerprinzip: Anstatt des eigentlichen Werts wird eine Referenz auf diesen Wert übertragen. Diese Referenz befindet sich auf der gleichen Ebene wie der SOAP Body. *Multi reference*-Werte treten somit immer als *independent* Elemente auf.

RPC-Kommunikation

Bis jetzt haben wir SOAP immer nur als Nachrichtenformat auf Basis von XML betrachtet. Wir wissen nun wie Daten in SOAP repräsentiert werden können und werden zu einem späteren Zeitpunkt auch sehen, wie .NET SOAP implementiert. Allerdings werden wir in einer echten Anwendung immer wieder auf die Problematik stoßen, dass die Kommunikation zwischen Komponenten so abläuft, dass auf eine Anfrage auch immer eine Antwort folgt. Dieses Kommunikationsmuster entspricht dem entfernten Methodenaufruf im klassischen RPC (Remote Procedure Call)-Umfeld. Für eine solchermaßen verteilte Kommunikation benötigen wir noch ein Transportprotokoll für unsere Daten. Diesen Teil werden wir anschließend besprechen und das Thema somit abrunden.

Die Kommunikation nach RPC-Muster ist in der Spezifikation als eine mögliche Art und Weise für den Datenaustausch zwischen Kommunikationspartnern vorgeschlagen, also nicht zwingend vorgeschrieben. RPC bedeutet in diesem Fall zum einen eine Kommunikation nach dem Muster Anfrage-Antwort (*request/response*) und zum anderen eine bestimmte Formatierung des SOAP Bodys. Der erste Fall ist eng gekoppelt mit dem zugrunde liegenden Transportmechanismus. Wichtig ist hier vor allem die Angabe eines URI für die Spezifizierung eines Zielobjekts bzw. Endpunkts. Der zweite Fall wird durch Regeln beschrieben, wie ein SOAP Body im Falle einer RPC-Kommunikation auszusehen hat. Wie man es vom herkömmlichen RPC (z.b. SUN RPC oder DCE RPC) her kennt, werden für einen erfolgreichen Aufruf ein Methodenname, eine Methodensignatur (optional), die Parameter für die Methode und im Falle von SOAP optionale Header-Informationen benötigt.

Der SOAP Body einer RPC-Kommunikation hat eine bestimmte Struktur. Eine Methodenanfrage wird als Struktur modelliert und dargestellt. Hier ist die Namensgebung der Struktur identisch mit dem Methodennamen (siehe Listing 5.6) und jeder Parameter (ob *[in]* oder *[in/out]*) wird als Kindelement dargestellt. Die Namensgebung dieser Elemente entspricht den Namen der Methodenparameter.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
   "http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m=
         "urn:de-eyesoft:Webservices">
         <symbol>EYS</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 5.6: SOAP Body einer Anfrage gemäß RPC-Muster

Ähnlich wie bei der Anfrage verhält es sich bei der Antwort. Zu erwähnen ist, dass der erste Wert der Parameterkollektion immer der Rückgabewert ist (andere Parameter sind dann ggf. »per Referenz« übergebene Werte). Die Namensgebung ist hier nicht ausschlaggebend. Allerdings hat sich für die Bezeichnung der Antwortstruktur eingebürgert, dass man dem Methodennamen das Suffix `Response` anhängt (siehe Listing 5.7). Das Element für den Rückgabewert wird oft als `Result` gekennzeichnet. Allerdings sind beide Konventionen nicht vorgeschrieben: es kann hier also zwischen unterschiedlichen SOAP-Implementierungen zu Interpretationsproblemen kommen. Wie man diese umgehen kann, werde ich Ihnen im Abschnitt über WSDL (Abschnitt 5.2.4) zeigen.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
   "http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
      <m:GetLastTradePriceResponse xmlns:m=
         "urn:de-eyesoft:Webservices">
         <Result>17.4</Result>
      </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 5.7: SOAP Body einer Antwort gemäß RPC-Muster

Mit diesen Informationen lässt sich eine vollständige RPC-basierte Kommunikation über SOAP realisieren. Allerdings gibt es – wie mehrfach angedeutet – noch viele Unzulänglichkeiten bzw. Mehrdeutigkeiten, die vor allem aus der SOAP-Spezifikation kommen. Abschließend benötigen wir nun noch einen Transportmechanismus zur tatsächlichen Übertragung unserer SOAP-XML-Daten.

Im Abschnitt über WSDL werden wir eine Alternative zur RPC-Darstellung eines SOAP Requests kennenlernen. Hier geht es vor allem um die Repräsentation der Anfrage oder der Antwort als XML-Dokument im Gegensatz zur Struktur-orientierten Herangehensweise beim RPC-Ansatz. Zusätzlich wird in diesem Abschnitt auch noch eine weitere Möglichkeit zur Kodierung der Daten angesprochen.

HTTP-Bindung

Das am meisten verwendete und am weitesten verbreitete Kommunikationsprotokoll im Internet ist HTTP (HyperText Transfer Protocol). HTTP wird in den meisten Fällen über den TCP-Port 80 übertragen. In diesem Fall sind die Firewalls von Unternehmen so konfiguriert, dass sie Kommunikation über diesen Port mit der Außenwelt zulassen. Da eines der großen Probleme traditioneller Middleware- und RPC-Ansätze die Durchdringung von

Firewallgrenzen ist, kommt die Kommunikation von SOAP über HTTP für Web Services-Szenarien gerade recht. Zudem ist HTTP sehr gut für die Implementierung von RPC-basierten SOAP-Kommunikationen geeignet.

Die HTTP-Bindung für SOAP schreibt einen *Content-Type* HTTP Header mit dem Wert *text/xml* vor. Zum Vergleich: gewöhnliche HTML-Seiten werden mit einem Content-Type *text/html* oder JPG-Bilder mit *image/jpeg* übertragen. Dieser HTTP Header soll dem verarbeitenden SOAP-Prozessor anzeigen, dass es sich hier um eine XML-Nachricht handelt, die er verarbeiten kann, und dass er überprüfen kann, ob es eine gültige SOAP-Nachricht ist.

Als Kommunikationsbasis verwendet die HTTP-Bindung die POST-Operation von HTTP. Dabei muss ein besonderer HTTP Header immer vorhanden sein (dies gilt für die Version 1.1 der Spezifikation – allerdings wird der Sinn dieses Headers für die nächste Version heftig diskutiert). Dieser Header heißt *SOAPAction*. SOAPAction soll die Absicht bzw. das Ziel der SOAP-Anfrage anzeigen. Der Wert des Headers kann dann beispielsweise von Firewalls ausgewertet werden. Diese können überprüfen, ob es ein gültiger URI ist bzw. mit einer separaten Datenbank vergleichen, ob der Zugriff auf diese Ressource über SOAP überhaupt erlaubt ist. Ein Beispiel für einen *SOAP-Action* HTTP Header sieht folgendermaßen aus:

```
SOAPAction: "http://www.eyesoft.de/webservices/buch/"
```

Doch auch hier gilt wieder: die Spezifikation lässt in diesem Bereich sehr viele Möglichkeiten offen, die durch unsauber implementierte SOAP Stacks zu Inkompatibilitäten führen können.

Eingebettet in eine HTTP-POST-Operation für die Anfrage und die Antwort sehen die oben aufgeführten SOAP Bodies aus wie in Listing 5.8 und Listing 5.9.

```
POST http:/www.eyesoft.de/webservices/buch/
   StockQuote HTTP/1.1
Content-Type: text/xml; charset="utf-8"
Content-Length: 306
SOAPAction: "http://www.eyesoft.de/webservices/buch/"

<SOAP-ENV:Envelope xmlns:SOAP-ENV=
   "http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m=
         "urn:de-eyesoft:Webservices">
         <symbol>EYS</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 5.8: HTTP Request für SOAP-Anfrage

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nn

<SOAP-ENV:Envelope xmlns:SOAP-ENV=
   "http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/"/>
   <SOAP-ENV:Body>
      <m:GetLastTradePriceResponse xmlns:m=
         "urn:de-eyesoft:Webservices">
         <Result>17.4</Result>
      </m:GetLastTradePriceResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Listing 5.9: HTTP Response für SOAP-Antwort

Viele werden schlucken, wenn sie lesen, dass man mit SOAP ganz einfach durch die Firewall dringen kann. Diese Tatsache ist aber nicht minder sicherheitsrelevant als ein HTML-Formular, welches über HTTP-POST an einen Server gesendet wird: auch hier können Daten oder Inhalte mitgeschickt werden, die Schaden oder Unheil auf dem Zielsserver anrichten können.

Das Thema Sicherheit bei Web Services im Umfeld von .NET behandeln wir in Kapitel 7.

Mir diesem Rüstzeug und einem Blick in die öffentliche SOAP-Spezifikation kann man sich nun einer SOAP-basierten Implementierung auf eigenen Füßen nähern. Allerdings hätten Sie nicht dieses Buch gekauft, wenn Sie jedwede SOAP-Kommunikation selbst und immer wieder neu implementieren müssten. Nachdem wir im nächsten Abschnitt über die Bereitstellung von Metadaten für unsere SOAP-Web Services sprechen, werde ich Ihnen im verbleibenden Rest des Kapitels vorführen, wie man mit ASP.NET XML Web Services-getriebene Architekturen realisieren kann.

5.2.4 WSDL

Im obigen Abschnitt über SOAP habe ich es mehrmals angesprochen: viele Dinge innerhalb von SOAP sind nicht zwingend vorgeschrieben (beispielsweise gewisse Attribute oder die Bezeichnung von Antwortelementen). Hinzu kommt dann noch das Problem, dass ein SOAP-Client nicht weiß, wie eine exakte SOAP-Anfrage für den jeweiligen Web Service auszusehen hat. Es gibt keinerlei Vorschrift in der SOAP-Spezifikation, wie ein SOAP Body auf jeden Fall gestaltet sein muss. Und wer teilt dem Client mit, welche Parameter in welcher Reihenfolge für die SOAP-Anfrage spezifiziert werden

müssen? Diese Problematik fällt nicht in den Bereich von SOAP bzw. der SOAP-Spezifikation. Für diese Metadaten gab es zunächst keine Lösung. Seit einiger Zeit hat sich aber ein Standard namens *WSDL* (*Web Services Description Language*) heraus kristallisiert, der mittlerweile auch von nahezu allen SOAP-Stack-Anbietern unterstützt wird.

WSDL ist ein XML-Dialekt, der vor allem dafür zuständig ist, die fehlenden und in vielen Situationen notwendigen Metadaten für Web Services zu beschreiben. WSDL ist dabei aber nicht allein an SOAP gebunden. Es gibt eine Vielzahl von vordefinierten Bindungen, zu denen auch SOAP 1.1 zählt. WSDL hält sich an eine abstrakte Beschreibung von Web Services. Neben diesen abstrakten Konstrukten wird aber in den meisten Fällen auch eine konkrete Implementierung mit zugehörigem Endpunkt (URL zum Web Service) angegeben.

WSDL verfolgt somit unterschiedliche Ziele:

- Erweiterbarkeit: neue Kodierungsvarianten und Transportmechanismen können definiert werden.
- Abstrakte Definitionen: Nachrichten und Dienste werden abstrakt beschrieben und können aber auf eine oder mehrere konkrete Implementierungen abgebildet werden.
- Wiederverwendung von Definitionen: vorhandene Definitionen von Endpunkten können für neue Definitionen wieder verwertet werden.

Im Folgenden werde ich Ihnen die wichtigsten Bestandteile einer WSDL-Beschreibung erläutern.

Überblick

Eine WSDL-Datei besteht aus mehreren XML-Elementen und entsprechenden Kindelementen. In Abbildung 5.4 können Sie die grafische Darstellung der einzelnen Elemente von WSDL sehen. Diese werden in den folgenden Abschnitten genauer erläutert. Für eine komplette Referenz der verwendeten WSDL-Beispiele werfen Sie bitte einen Blick auf Listing 5.10 (für den Fall *RPC*, siehe unten) und Listing 5.11 (für den Fall *Document*, siehe unten). Diese WSDL-Beschreibungen sind zwei unterschiedliche Darstellungen für einen Währungsumrechner-Web Service.

Dieser Web Service bietet zwei Methoden an: das Umrechnen von DM nach EUR und von EUR nach DM. Es sind zwei verschiedene WSDLs, weil es für eine SOAP-basierte Kommunikation in der Regel unterschiedliche Repräsentationen geben kann (vgl. Kasten).

Eine WSDL-Beschreibung beginnt immer mit dem Element `<definitions>`. Direkt in diesem Element werden die im Dokument verwendeten Namensräume als Attribute definiert. Sämtliche anderen, in der WSDL-Beschreibung vorkommenden Elemente sind dann Kinder dieses Wurzelelements.

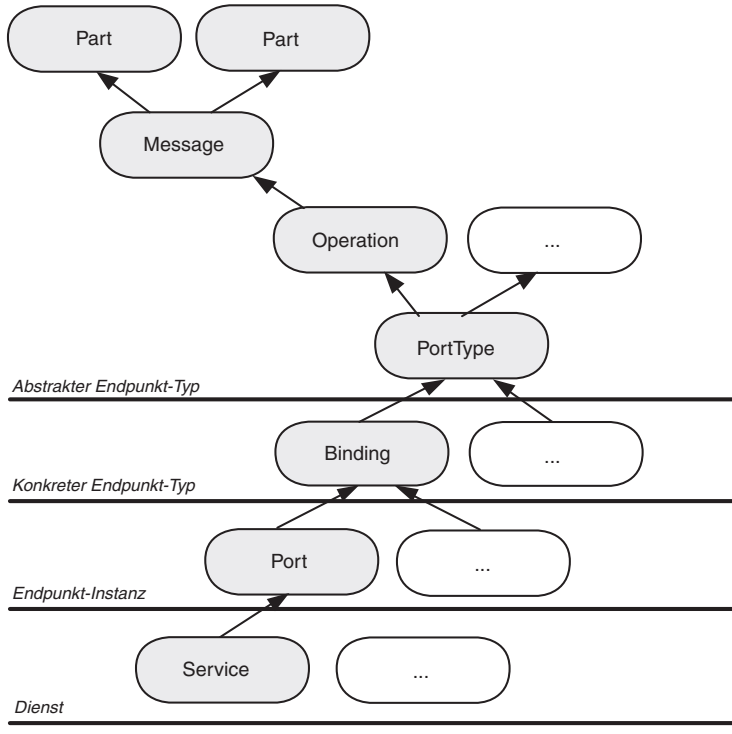


Abbildung 5.4: Überblick über die Elemente einer WSDL-Beschreibung

Types

Nicht in der Abbildung aufgeführt ist die Sektion für die Datentypen (*Types*). In der *Types*-Sektion werden die im WSDL verwendeten Datentypen definiert. Diese Datentypen sind komplexe oder zusammengesetzte Typen, die mithilfe von XML Schema-Datentypen erstellt werden.

Message

Der Bereich *Message* beschreibt die einzelnen Nachrichten in `message`-Elementen, die zwischen den Kommunikationspartnern ausgetauscht werden können. Dabei gilt für eine Kommunikation nach dem RPC-Muster, dass jede einzelne Nachricht für Anfrage und Antwort in diesem Bereich spezifiziert werden muss. Das Attribut *Part* einer *Message* spezifiziert die Parameter einer Nachricht.

Operation

Eine *Operation* definiert eine Kombination aus Eingabe-, Ausgabe- und Fehlernachrichten. Das zuständige Element <`operation`> besitzt weitere Kindelemente, welche die Richtung der Operation angeben: `input` oder `output`.

PortType

Ein *PortType* ist eine Sammlung von Operationen, die gemeinsam nach außen dargestellt werden. Man könnte das abstrakte Konstrukt des *PortTypes* also mit einer Schnittstelle vergleichen.

Zusammen mit den Elementen *Message* und *Operation* bildet der *PortType* die abstrakte Definiton des WSDL-Vertrags. Diese abstrakte Darstellung gilt es nun konkret abzubilden. In diesen Definitionen sind keine Informationen bzgl. des verwendeten Protokolls oder der Kodierung enthalten.

Binding

Die Bindung in einem WSDL gibt an, welches Protokoll und Format zur Übertragung der Nachrichten verwendet werden soll. In unserem Fall interessieren wir uns hauptsächlich für SOAP als Bindung. Es gibt allerdings auch vordefinierte Bindungen für HTTP GET, HTTP POST und MIME.

Im Falle der SOAP-Bindung gibt es zwei verschiedene Ausprägungen: zum einen *RPC* und zum anderen *Document* (diese werden über das Attribut `style` des Elements `<binding>` festgelegt. Zudem wird bei einer SOAP-Bindung auch noch angegeben, wie die Daten und Parameter serialisiert werden: gemäß den Regeln in Abschnitt 5 der SOAP-Spezifikation (*encoded* genannt) oder gemäß den Regeln der XML-Schema-Spezifikation (*literal* genannt). Vergleichen Sie hierzu bitte auch den Kasten »SOAP ist nicht SOAP«.

Die Beschreibungen im Abschnitt *Binding* stellen somit die konkrete Beschreibung eines Endpunkt-Typs im Gegensatz zu seinem abstrakten Widerpart im Abschnitt *PortType* dar. Hier werden nun Informationen zur Verwendung von Protokollen und Kodierungen angegeben.

Document bedeutet, dass XML-Dokumente im SOAP Body übertragen werden. Dies entspricht in etwa der Vorstellung des Kommunikationsprinzips beim Messaging: eine Partei schickt eine Nachricht zu einer anderen Partei, welche mit einer weiteren Nachricht antworten kann. In diesem Fall sind die Nachrichten eben XML-Dokumente innerhalb des SOAP Bodys.

RPC bedeutet, dass man die Daten in Form von Methodenaufrufen (z.B: für RPC) kodieren und versenden möchte. Dieses Format ist dann gemäß der SOAP-Spezifikation im SOAP Body zu realisieren (siehe auch Abschnitt 5.2.3).

Service und Port

Das `service`-Element schließlich beschreibt den eigentlichen Web Service in seiner Gesamtheit als Dienst aus der Sicht des Clients. Es definiert eine Sammlung von so genannten *Ports* (Element `<port>`). Jedes `service`-Element sollte einem `portType`-Element entsprechen und kann unterschiedliche Wege repräsentieren, auf welche Weise man auf den Web Service zugreifen kann.

So enthält das Kindelement <address> von <port> die Adresse des Endpunkts, welcher den Web Service tatsächlich implementiert.

Während ein *Port* also eine Instanz des Endpunkts darstellt, kann man die Informationen im Bereich *Service* als den eigentlichen Dienst bezeichnen.

Dokumentation

Eine nette Eigenschaft von WSDL ist, dass man zu den wichtigsten Elementen wie <operation> oder <service> eigene Dokumentationsinformationen angeben kann. Dies geschieht über das Element <documentation>. Diese Informationen können dann entsprechende SOAP- bzw. WSDL-Implementierungen beispielsweise in ihre Online-Hilfen integrieren.

SOAP ist nicht gleich SOAP

Es mag auf den ersten Blick etwas seltsam anmuten, aber SOAP ist wirklich nicht immer gleich SOAP. Und WSDL ist nicht immer WSDL. Aufgrund der in Abschnitt 5.2.3 und 5.2.4 beschriebenen Tatsachen kann es in einer realen Web Services-Landschaft vorkommen, dass ein SOAP-Client nicht mit einem SOAP-Server (also einem XML Web Service) zusammen arbeiten kann. Grund hierfür sind zum einen die unterschiedlichen Möglichkeiten, den SOAP Body zu formatieren: RPC oder Document. Zum anderen gibt es zusätzlich noch zwei Alternativen, wie die Datentypen serialisiert in der SOAP-Nachricht dargestellt werden: *encoded* oder *literal*. Mit diesen insgesamt vier Möglichkeiten wird das eigentliche Ziel von SOAP und WSDL – nämlich die Interoperabilität – momentan stark beschnitten. In der Praxis trifft man heute jedoch vornehmlich auf die Kombinationen RPC/encoded (aus der ursprünglichen SOAP-Spezifikation) und Document/literal (für eine XML-Dokumenten-Darstellung gemäß XSD).

Allerdings kümmerte sich zum Zeitpunkt der Erstellung dieses Buchs bereits ein Standardgremium darum, dass die Interoperabilität in Zukunft auch tatsächlich gewährleistet ist.

Anhand der oben beschriebenen XML-Darstellung von Metadaten über Web Services kann ein Client sich typsicher mit einem Web Service verbinden. Die Definition von WSDL befindet sich zur Zeit noch im Fluss, daher gibt es auch immer wieder Ungereimtheiten bei der Zusammenarbeit unterschiedlicher Implementierungen.

Doch keine Bange! Sie müssen WSDL nicht wirklich beherrschen. Es ist zwar gut, wenn Sie die grundlegende Funktionalität verstanden haben und mit den einzelnen Elementen etwas anfangen können. Aber Sie müssen hoffentlich nie ein WSDL selbst schreiben oder gar verarbeiten (der erste Fall wird für den Bereich Datenmodellierung sicherlich interessant werden. Dafür wird es aber entsprechend ausgestattete Werkzeuge geben). Für .NET brauchen Sie sich auf keinen Fall Sorgen zu machen. Sämtliche SOAP-Stacks in

.NET kümmern sich für Sie um die Generierung und Verarbeitung von WSDL. Allerdings können Sie über entsprechende APIs auch manuell eingreifen – wenn Sie möchten.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:soap=
   "http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://tempuri.org/"
   xmlns:s="http://www.w3.org/2001/XMLSchema"
   xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
   xmlns:tm=
      "http://microsoft.com/wsdl/mime/textMatching/"
   xmlns:mime=
      "http://schemas.xmlsoap.org/wsdl/mime/"
   xmlns:soapenc=
      "http://schemas.xmlsoap.org/soap/encoding/"
   targetNamespace="http://tempuri.org/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
   <types />
   <message name="DM2EURSoapIn">
      <part name="DMvalue" type="s:float" />
   </message>
   <message name="DM2EURSoapOut">
      <part name="DM2EURResult" type="s:float" />
   </message>
   <message name="EUR2DMSoapIn">
      <part name="EURvalue" type="s:float" />
   </message>
  <message name="EUR2DMSoapOut">
      <part name="EUR2DMResult" type="s:float" />
   </message>
   <portType name="WaehrungsRechnerSoap">
      <operation name="DM2EUR">
         <documentation>Rechnet D-Mark nach
         Euro um.</documentation>
         <input message="tns:DM2EURSoapIn" />
         <output message="tns:DM2EURSoapOut" />
      </operation>
      <operation name="EUR2DM">
         <documentation>Rechnet Euro nach
         D-Mark um.</documentation>
         <input message="tns:EUR2DMSoapIn" />
         <output message="tns:EUR2DMSoapOut" />
      </operation>
   </portType>
   <binding name="WaehrungsRechnerSoap" type=
      "tns:WaehrungsRechnerSoap">
      <soap:binding transport=
```

```
            "http://schemas.xmlsoap.org/soap/http"
            style="rpc" />
         <operation name="DM2EUR">
            <soap:operation soapAction=
               "http://tempuri.org/DM2EUR" style="rpc" />
            <input>
               <soap:body use="encoded" namespace=
                  "http://tempuri.org/" encodingStyle=
                  "http://schemas.xmlsoap.org/soap/
                   encoding/" />
            </input>
            <output>
               <soap:body use="encoded" namespace=
                  "http://tempuri.org/" encodingStyle=
                  "http://schemas.xmlsoap.org/soap/
                   encoding/" />
            </output>
         </operation>
         <operation name="EUR2DM">
            <soap:operation soapAction=
               "http://tempuri.org/EUR2DM" style="rpc" />
            <input>
               <soap:body use="encoded" namespace=
                  "http://tempuri.org/" encodingStyle=
                  "http://schemas.xmlsoap.org/soap/
                   encoding/" />
            </input>
            <output>
               <soap:body use="encoded" namespace=
                  "http://tempuri.org/" encodingStyle=
                  "http://schemas.xmlsoap.org/soap/
                   encoding/" />
            </output>
         </operation>
      </binding>
      <service name="WaehrungsRechner">
         <documentation>Ein einfacher Währungsumrechner-
         Web Service.</documentation>
         <port name="WaehrungsRechnerSoap" binding=
            "tns:WaehrungsRechnerSoap">
            <soap:address location=
               "http://localhost/wstest/
                WaehrungsRechner_RPC.asmx" />
         </port>
      </service>
</definitions>
```

Listing 5.10: WSDL-Beschreibung des Währungsrechners für RPC-Kommunikation

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:http=
   "http://schemas.xmlsoap.org/wsdl/http/" xmlns:soap=
   "http://schemas.xmlsoap.org/wsdl/soap/" xmlns:s=
   "http://www.w3.org/2001/XMLSchema" xmlns:s0=
   "http://tempuri.org/" xmlns:soapenc=
   "http://schemas.xmlsoap.org/soap/encoding/" xmlns:tm=
   "http://microsoft.com/wsdl/mime/
   textMatching/" xmlns:mime=
   "http://schemas.xmlsoap.org/wsdl/
   mime/" targetNamespace= "http://tempuri.org/"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
   <types>
      <s:schema elementFormDefault="qualified"
         targetNamespace="http://tempuri.org/">
         <s:element name="DM2EUR">
            <s:complexType>
               <s:sequence>
                  <s:element minOccurs="1" maxOccurs="1"
                   name="DMvalue" type="s:float" />
               </s:sequence>
            </s:complexType>
         </s:element>
         <s:element name="DM2EURResponse">
            <s:complexType>
               <s:sequence>
                  <s:element minOccurs="1" maxOccurs="1"
                   name="DM2EURResult" type="s:float" />
               </s:sequence>
            </s:complexType>
         </s:element>
         <s:element name="EUR2DM">
            <s:complexType>
               <s:sequence>
                  <s:element minOccurs="1" maxOccurs="1"
                   name="EURvalue" type="s:float" />
               </s:sequence>
            </s:complexType>
         </s:element>
         <s:element name="EUR2DMResponse">
            <s:complexType>
               <s:sequence>
                  <s:element minOccurs="1" maxOccurs="1"
                   name="EUR2DMResult" type="s:float" />
               </s:sequence>
            </s:complexType>
         </s:element>
```

```
   </s:schema>
</types>
<message name="DM2EURSoapIn">
   <part name="parameters" element="s0:DM2EUR" />
</message>
<message name="DM2EURSoapOut">
   <part name="parameters" element=
    "s0:DM2EURResponse" />
</message>
<message name="EUR2DMSoapIn">
   <part name="parameters" element="s0:EUR2DM" />
</message>
<message name="EUR2DMSoapOut">
   <part name="parameters" element=
    "s0:EUR2DMResponse" />
</message>
<portType name="WaehrungsRechnerSoap">
   <operation name="DM2EUR">
      <documentation>Rechnet D-Mark nach Euro
      um.</documentation>
      <input message="s0:DM2EURSoapIn" />
      <output message="s0:DM2EURSoapOut" />
   </operation>
   <operation name="EUR2DM">
      <documentation>Rechnet Euro nach D-Mark
      um.</documentation>
      <input message="s0:EUR2DMSoapIn" />
      <output message="s0:EUR2DMSoapOut" />
   </operation>
</portType>
<binding name="WaehrungsRechnerSoap" type=
   "s0:WaehrungsRechnerSoap">
   <soap:binding transport=
      "http://schemas.xmlsoap.org/soap/http"
      style="document" />
   <operation name="DM2EUR">
      <soap:operation soapAction=
         "http://tempuri.org/DM2EUR"
         style="document" />
      <input>

         <soap:body use="literal" />
      </input>
      <output>
         <soap:body use="literal" />
      </output>
   </operation>
```

```
      <operation name="EUR2DM">
         <soap:operation soapAction=
            "http://tempuri.org/EUR2DM"
            style="document" />
         <input>
            <soap:body use="literal" />
         </input>
         <output>
            <soap:body use="literal" />
         </output>
      </operation>
   </binding>
   <service name="WaehrungsRechner">
      <documentation>Ein einfacher Währungsumrechner-
      Web Service.</documentation>
      <port name="WaehrungsRechnerSoap"
       binding="s0:WaehrungsRechnerSoap">
         <soap:address location=
            "http://localhost/wstest/
             WaehrungsRechner_DOC.asmx" />
      </port>
   </service>
</definitions>
```

Listing 5.11: WSDL-Beschreibung des Währungsrechners für Document-Kommunikation

5.3 XML Web Services in ASP.NET

Nach all den theoretischen und zugegebenermaßen relativ trockenen Grundlagen kommen wir nun zur Erstellung und Nutzung von XML Web Services in ASP.NET. Dabei sind die im ersten Teil des Kapitels erlangten Kenntnisse durchaus von Vorteil – vor allem wenn mal etwas schief geht und nicht wie beabsichtigt funktioniert.

In .NET gibt es zwei SOAP- und Web Service-Implementierungen: ASP.NET und .NET Remoting. Während wir uns im Rest des Kapitels mit ASP.NET beschäftigen, können Sie in Kapitel 8 mehr über das Konzept und die Technologie von .NET Remoting nachlesen.

5.3.1 ASP.NET und Web Services

Die ASP.NET-Umgebung in .NET gibt uns die Möglichkeit, SOAP-basierte XML Web Services zu erstellen, zu betreiben und in Client-Anwendungen zu benutzen. In diesem Kapitel werden sowohl die Erstellung von Serverseitigen Web Services mit Werkzeugen aus dem .NET Framework SDK und mit dem Visual Studio .NET, als auch die Verwendung dieser Web Services

in Client-Awendungen vorgestellt. Die Applikationen werden sowohl mit der integrierten Entwicklungsumgebung als auch mit Kommandozeilenbasierten Programmen erstellt.

Wie bereits in Kapitel 3 dargestellt, bedient sich ASP.NET eines neuen Konzepts, der so genannten HTTP Pipeline. In dieser Pipeline können mehrere HTTP-Module abgearbeitet werden, bevor am Ende der Verarbeitungskette ein HTTP Handler für die eigentliche Umsetzung der Anfrage zuständig zeichnet. Damit ASP.NET nun Web Services auf Basis von XML und SOAP zur Verfügung stellen kann, implementiert es einen eigenen Handler für diese Zwecke. Dieser Handler ist implementiert in *System.Web.Services.Protocols.WebServiceHandler*. Er ist wie alle anderen Handler in der maschinenweiten Konfigurationsdatei *machine.config* registriert und er reagiert auf Dateien mit der Endung *.asmx*. Der Handler sorgt u.a. auch dafür, dass einfach durch das Anhängen von *?wsdl* an den URL aus einem vorliegenden Web Service automatisch eine entsprechende WSDL-Darstellung zur Laufzeit erzeugt wird. Wann immer Sie also einen auf ASP.NET aufsetzenden XML Web Service programmieren wollen, müssen Sie Ihren Code in eine Datei *dateiname.asmx* abspeichern.

Natürlich muss für die Verwendung von Web Services mit dem IIS diese Dateiendung noch in der Konfiguration des IIS festgehalten werden (vgl. auch Kapitel 3).

5.3.2 XML Web Services erstellen

Ein angenehmer Nebeneffekt von ASP.NETs Architektur und Konzept ist die Möglichkeit, sämtliche Applikationen für ASP.NET mit einem einfachen Editor wie Notepad zu erstellen. Für unsere Zwecke im Bereich Web Services wollen wir also Notepad verwenden, um eine .asmx-Datei für unseren ersten XML Web Service zu realisieren. Die erstellte .asmx-Datei müssen wir in ein virtuelles Verzeichnis im IIS abspeichern, damit die ASP.NET-Laufzeitumgebung korrekt damit umgehen kann (siehe auch Kapitel 3).

Ähnlich wie bei der Programmierung von ASP.NET Web Form-Anwendungen übernimmt die Laufzeitumgebung den Löwenanteil, um den Entwickler vor zu viel Arbeit und Details zu »schützen«. Die verwendeten Klassen bei der Web Service-Programmierung befinden sich im Namensraum *System.Web.Services*. Diesen wollen wir uns nun genauer betrachten.

Der Namensraum System.Web.Services

Der Namensraum besteht aus vier Klassen, die Sie in Tabelle 5.2 sehen können. Die Programmierung von ASP.NET Web Services beruht auf dem in .NET weit verbreiteten Prinzip der Attribute (siehe Kapitel 2). Daher sind die drei Klassen zur Implementierung der geeigneten Attribute auch die wichtigsten.

Tab. 5.2: Klassen im Namensraum System.Web.Services

Klasse	Beschreibung
WebMethodAttribute	Diese Klasse implementiert ein Attribut, welches – auf eine Methode angewendet – diese automatisch als eine in einen Web Service eingebettete Methode veröffentlicht.
WebService	Definiert die optionale Basisklasse für XML Web Services in ASP.NET. Erlaubt den direkten Zugriff auf den HTTP-Kontext (z. B. Request, Response, Session, …).
WebServiceAttribute	Diese Klasse implementiert ein Attribut, welches - auf eine Klasse angewendet – die Web Service-Beschreibung mit zusätzlichen Informationen versehen kann.
WebServiceBindingAttribute	Mit diesem Attribut kann die Zuordnung von Methoden an WSDL-Bindungen gesteuert werden.

In den kommenden Abschnitten werden wir drei dieser Klassen wieder sehen und in Aktion erleben, wenn wir unseren Web Service implementieren.

Klassen als XML Web Services veröffentlichen

Ein Vorteil von ASP.NET Web Services ist die Tatsache, dass Sie als Entwickler Ihre Klassen wie gewohnt modellieren und implementieren. Auf Basis dieser Klassen entscheiden Sie dann, welche Klassen und welche Methoden über eine Web Service-Schnittstelle von außen zugänglich sein sollen.

Damit ASP.NET überhaupt etwas mit der .asmx-Datei anfangen kann, bedarf es wieder einer speziellen Direktive, die am Beginn einer jeden Web Service-Datei stehen muss. Diese Direktive heißt *@WebService* und hat folgende Ausprägung:

```
<%@ WebService Language="C#" Class="WaehrungsRechner" %>
```

Die WebService-Direktive besitzt vier verschiedene Eigenschaften, deren Bedeutungen in Tabelle 5.3 nochmals zusammenfassend aufgeführt sind. Am wichtigsten sind hierbei die beiden Eigenschaften *Language*, die angibt, um welche Implementierungssprache es sich handelt, und *Class*, die angibt, welche Klasse in der Datei oder in der Code-Behind-Datei den Web Service implementiert.

Eigenschaft	Beschreibung
Language	Implementierungssprache des Web Services (C#, VB.NET oder JScript.NET). Wird durch die Angabe von Kürzeln C#, VB oder JS spezifiziert.
Class	Gibt die Klasse an, die einen Web Service implementiert.
Code-Behind	Wenn sich die Implementierung des Web Services nicht in der gleichen .asmx-Datei oder nicht in einer vorkompilierten Assembly im bin-Verzeichnis befindet, dann gibt diese Eigenschaft die Datei der Implementierung an.
Debug	Erzeugt Debug-Symbole für diesen Web Service, wenn der Wert auf *true* gesetzt wird.

Tab. 5.3: Eigenschaften der @WebService-Direktive

Vor allem für die Erstellung von Web Services in Testumgebungen ist die *Debug*-Eigenschaft sehr interessant. Wenn diese auf *true* gesetzt wird, dann erzeugt der ASP.NET-Compiler zusätzlich zu der ohnehin automatisch generierten Assembly noch Debug-Informationen, die dann in geeigneten Debuggern ausgewertet werden können.

Schauen wir uns nun unseren ersten kleinen ASP.NET XML Web Service an. Es handelt sich um einen Währungsumrechner für DM und Euro. Der Code steht in Listing 5.12 (dies ist eine vereinfachte Implementierung. Eine bessere Version befindet sich auf der CD). Erstes auffälliges Merkmal des Codes ist die Verwendung des Attributs *[WebService]*. Dieses Attribut ist allerdings optional und besitzt als Eigenschaften auch nur die Angabe einer textuellen Beschreibung und das Festlegen des gewünschten XML-Namensraums innerhalb des SOAP- und WSDL-Dokuments.

Die wesentlich interessantere und notwendige Neuerung in diesem Code ist das Attribut *[WebMethod]*. Wird dieses Attribut auf eine öffentliche Methode innerhalb der in der Direktive angegebenen Klasse angewendet, dann sorgt die ASP.NET-Laufzeitumgebung (in Verbindung mit der Klasse *XmlSerializer*) dafür, dass diese Methode über die Klasse per Web Service-Technologien ansprechbar ist. Standardmäßig veröffentlicht ASP.NET einen XML Web Service über SOAP, HTTP-GET und HTTP-POST (dies entspricht auch den vordefinierten Bindungen in WSDL). Die *XmlSerializer*-Instanz innerhalb von ASP.NET sorgt dann ohne Zutun des Programmierers für die Serialisierung und Deserialisierung der Parameter.

Das ist wirklich alles, was man wissen und können muss, um einfache Web Services (will heißen: mit einfachen Datentypen und für einfache Szenarien) in ASP.NET umsetzen zu können. An diesem Punkt ist keine Kenntnis von XML, SOAP oder gar WSDL notwendig. .NET und ASP.NET schirmen den Entwickler von diesen Technologien und Standards vollständig ab. Man kann allerdings auch manipulierend in die Erzeugung der SOAP-Pakete und der WSDL-Beschreibung eingreifen. Darauf kommen wir in Kapitel 6 zurück.

```
<%@ WebService Language="C#" Class="WaehrungsRechner" %>
using System.Web.Services;

[WebService(Description="Ein einfacher
 Währungsumrechner-Web Service.",
 Namespace="http://eyesoft.de/webservices/")]
public class WaehrungsRechner
{
   [WebMethod(Description="Rechnet D-Mark nach
    Euro um.")]
   public double DM2EUR (double DMvalue)
   {
      return DMvalue / 1.95583;
   }

   [WebMethod(Description="Rechnet Euro nach
    D-Mark um.")]
   public double EUR2DM (double EURvalue)
   {
      return EURvalue * 1.95583;
   }
}
```

Listing 5.12: Währungsumrechner als einfacher XML Web Service in ASP.NET

Ist der Web Service einmal geschrieben, dann möchte und muss man ihn testen und in Client-Applikationen einsetzen. Hierzu speichern wir unseren Code in der Datei *umrechner.asmx* in ein dem IIS bekannten Verzeichnis ab.

Testen mit der Hilfeseite

Für das Testen der XML Web Services hält ASP.NET ein unverzichtbares Feature bereit. Es erzeugt für eine korrekt implementierte .asmx-Datei automatisch eine HTML-Hilfeseite mit eingebauter Funktionalität zum rudimentären Testen der Web Service-Logik. Damit wir diese Seite genauer betrachten können, geben wir den URL zu unserem Währungsumrechner-Web Service in einen Browser ein: *http://localhost/wsbuch/K5/umrechner.asmx* (siehe Abbildung 5.5). Die Hilfeseite erzeugt beim Aufruf eine Liste aller in der Web Service-Klasse verfügbaren *[WebMethod]*-Methoden und listet diese in einer HTML-Seite auf – ohne dass wir als Programmierer jemals eine Zeile HMTL erzeugt haben. Im oberen Bereich der Seite befindet sich ein Link zu der zu diesem Web Service gehörenden WSDL-Beschreibung. Diese Beschreibung ist allerdings nicht statisch im Dateisystem abgelegt, sondern wird von der Laufzeitumgebung dynamisch erzeugt (siehe Abbildung 5.6). Dadurch wird jede Änderung am Quelltext, z.B. durch Abänderung eines Parameters, sofort im WSDL

reflektiert. Diese Eigenschaft von ASP.NET macht es dem Web Service-Entwickler ungemein einfach, seine Dienste interessierten Clients zur Verfügung zu stellen.

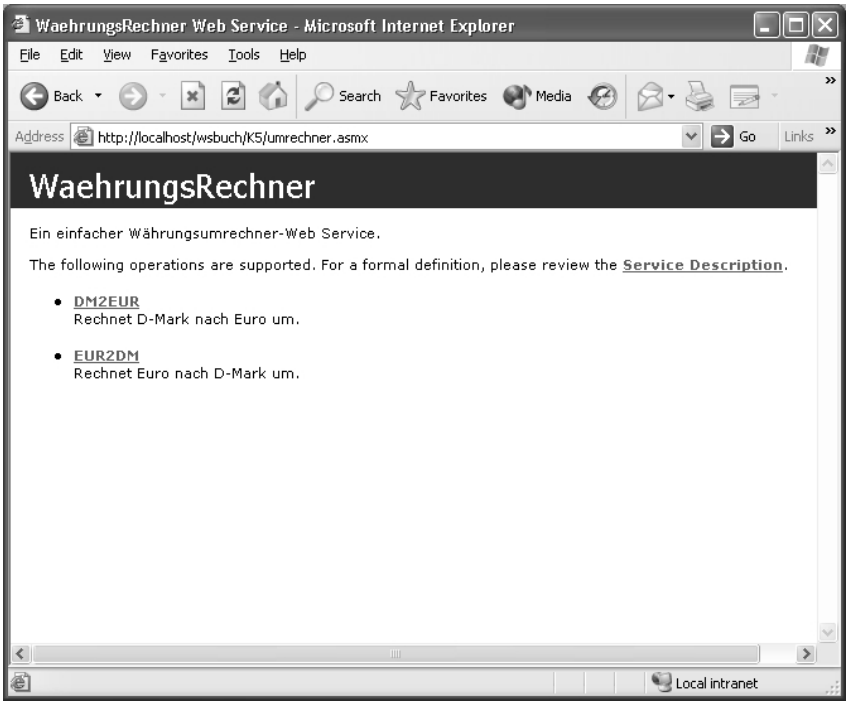


Abbildung 5.5: Automatisch erstellte Hilfeseite für ASP.NET XML Web Services

Die Hilfeseite für ASP.NET XML Web Services ist in der Datei *DefaultWsdlHelpGenerator.aspx* implementiert. Diese Datei befindet sich im Verzeichnis *<SYSTEM>\Microsoft.NET\Framework\<VERSION>\CONFIG*. Diese Seite kann man auch ändern, sollte dies aber mit Vorsicht tun. Ein besserer Ansatz ist es, die Definition der Datei in *machine.config* einfach auf die neue Datei umzulenken. Alternativ ist auch eine Verwendung von unterschiedlichen Hilfeseiten auf Basis von ASP.NET Web-Anwendungen durch eine Definition in *web.config* möglich.

Nachdem man auf eine der verfügbaren Operationen auf der Hilfeseite geklickt hat, erscheint eine Eingabemaske für die diversen Parameter, welche u.U. der Methode übergeben werden (vgl. Abbildung 5.7). Für unseren Umrechner geben wir beispielsweise den DM-Betrag an, der in Euro umgerechnet werden soll. Nach Abschicken der Testfunktion erscheint ein neues Fenster mit dem Ergebnis (Abbildung 5.8).

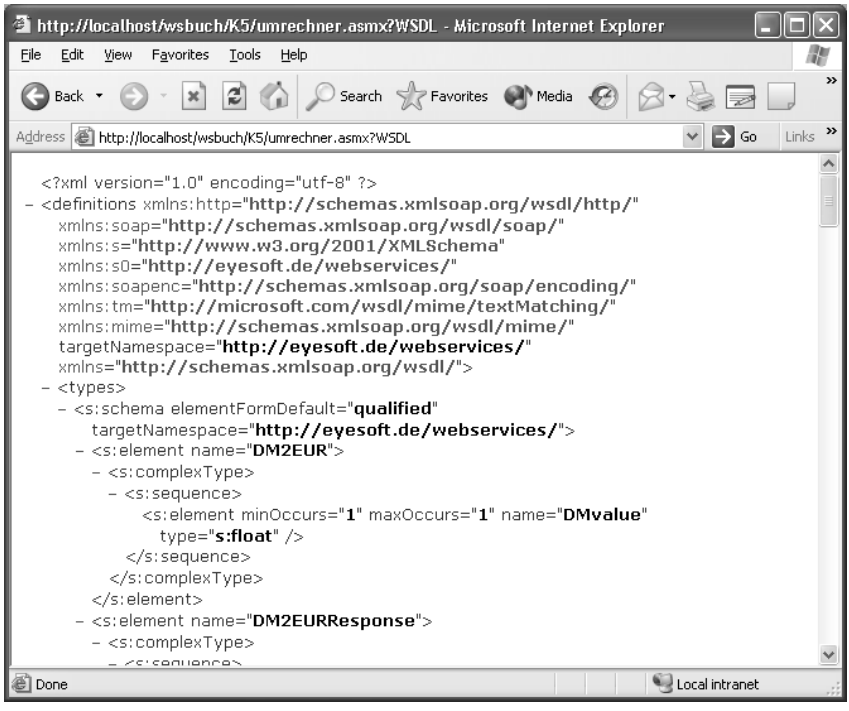


Abbildung 5.6: Automatisch erstellte WSDL-Beschreibung für ASP.NET Web Services

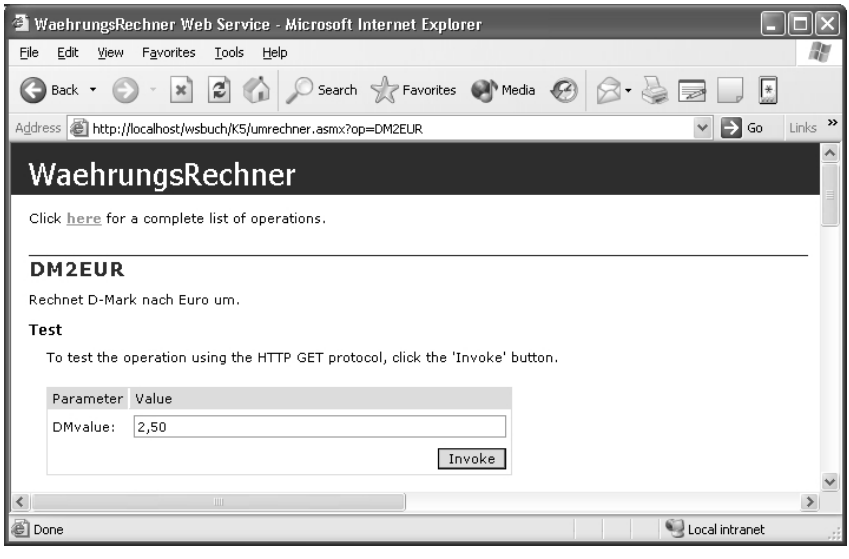


Abbildung 5.7: Testen der Web Services-Logik mit der Hilfeseite

Hier wird zur Kommunikation mit dem Web Service aber nicht etwa SOAP verwendet, sondern ein simpler HTTP-GET-Aufruf. Man kann also mit dieser Methode nicht überprüfen, ob die erzeugten und versendeten SOAP-

Pakete den eigenen oder den Vorstellungen anderer SOAP-Implementierer entsprechen. Denn alles, was hier hinter der Oberfläche passiert, ist die Übergabe der Parameter für die Methode als Anhang im URL:

```
http://localhost/wsbuch/K5/umrechner.asmx/DM2EUR?
   DMvalue=2,50
```

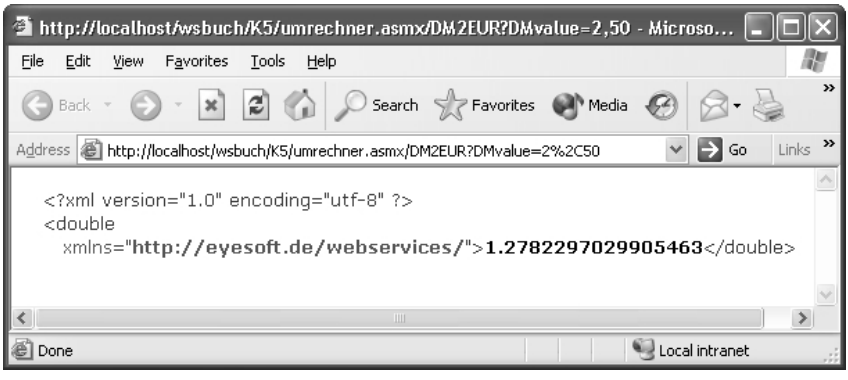


Abbildung 5.8: Aufruf des Web Services und Ausgabe der getesteten Methode

Wenn man also so die Logik des Web Services in den Grundzügen getestet hat, kann man dazu übergehen, eine Client-Anwendung dafür zu schreiben. Zuvor wollen wir uns aber noch ein paar wichtige Themen bei der Erstellung von ASP.NET Web Services anschauen.

XML-Serialisierung

Die automatische Umwandlung der CLR-Typen in eine XML-Repräsentation und wieder zurück wird im Falle von ASP.NET-basierten XML Web Services von der Klasse *XmlSerializer* (bzw. von Instanzen der Klasse) erledigt. *XmlSerializer* kann einfache Typen umwandeln und weiß auch mit zusammen gesetzten Strukturen oder eigenen Klassen umzugehen (siehe Tabelle 5.4).

Datentyp	Beschreibung und Beispiel
Einfache Typen	z.B. *String*, *Int32*, *Byte*, *Boolean*, *Int16*, *Int64*, *Single*, *Double*, *Decimal*, *DateTime* und *XmlQualifiedName*.
Aufzählungstypen	`Public enum Farbe { rot=1, blau=2 }.`
Felder von einfachen Typen und Aufzählungen	Felder (Arrays) wie z.B. *String[]* oder *Integer[]*.
Klassen und Strukturen	Klassen und Strukturen mit öffentlichen Felder oder Eigenschaften.
Felder von Klassen/Strukturen	*MyType[].*

Tab. 5.4: Unterstützte Datentypen in ASP.NET XML Web Services (über SOAP)

Tab. 5.4: Unterstützte Datentypen in ASP.NET XML Web Services (über SOAP) (Forts.)

Datentyp	Beschreibung und Beispiel
DataSet	ADO.NET *DataSet*. Ein DataSet ist ein Speicherabbild einer Ergebnisrelation (z. B. aus einer DB-Abfrage). Grob vergleichbar mit ADO *Recordset*. Siehe Kapitel 4.
Felder von *DataSet*	*DataSet[]*.
XmlNode	*XmlNode* ist eine In-Memory-Repräsentation eines XML-Fragments. Siehe Kapitel 4.
Felder von *XmlNode*	*XmlNode[]*.

XmlSerializer kann allerdings keine privaten Klassenfelder serialisieren und verfolgt alles in allem ein sehr stark auf die XSD-Datentypen ausgelegtes Modell, um eben eine größtmögliche Kompatibilität mit anderen SOAP-Implementierungen zu erzielen. Die Ausprägung des endgültigen Serialisierungsergebnisses lässt sich über Attribute steuern. An dieser Stelle sei vor allem auf die Klassen *System.Web.Services.Protocols.SoapRpcServiceAttribute* und *System.Web.Services.Protocols.SoapDocumentServiceAttribute* hingewiesen. Wie im Abschnitt über SOAP und WSDL bereits angedeutet, gibt es mehrere mögliche Darstellungen ein und derselben SOAP-Kommunikation. Mit diesen beiden Attributen kann der Entwickler nun bestimmen, ob das erzeugte WSDL (und somit auch automatisch das erwartete bzw. erzeugte SOAP-Format) nach dem RPC- oder dem Document-Muster gestaltet werden soll. Als Voreinstellung erzeugt ein ASP.NET-Web Service eine Dokumentendarstellung mit literaler Parameterdarstellung (was dem Angeben des Attributs *[SoapDocumentService]* ohne gesetzte Eigenschaften entspricht). Durch Auszeichnen einer Web Service-Klasse mit *[SoapRpcService]* erzeugt der *XmlSerializer* eine RPC-orientierte Darstellung mit encodierter Parameterrepräsentation. Über die Eigenschaft *Use* kann man in beiden Fällen noch die Parameterformatierung im SOAP Envelope gesondert festlegen: *Literal* oder *Encoded*, die jeweils Konstanten der Struktur *SoapBindingUse* sind (zu finden im Namensraum *System.Web.Services.Description*).

Neben diesen beiden wichtigen Attributen auf Klassenebene gibt es zusätzlich noch zwei weitere Attribute auf Methodenebene, die eine ähnliche Funktionalität bieten. Eine Auflistung der Attribute zur Steuerung der XML-Ausprägung des SOAP Envelopes ist in Tabelle 5.5 zu ersehen.

Attribut	Bedeutung
SoapRpcService	RPC-orientierte Kommunikation mit *encoded* Parameterformatierung (steuerbar über die Eigenschaft *Use*).
SoapDocumentService	Dokumenten-orientierte Kommunikation mit *literal* Parameterformatierung (steuerbar über die Eigenschaft *Use*).
SoapRpcMethod	RPC-orientierte Kommunikation mit *encoded* Parameterformatierung (steuerbar über die Eigenschaft *Use*). Bezieht sich nur auf die jeweilige Methode.
SoapDocumentMethod	Dokumenten-orientierte Kommunikation mit *literal* Parameterformatierung (steuerbar über die Eigenschaft *Use*). Bezieht sich nur auf die jeweilige Methode

Tab. 5.5: Web Service-Attribute zur Bestimmung der SOAP Envelope-Formatierung

Fehlerbehandlung

Wie wir bereits in Abschnitt 5.2.3 gesehen haben, werden Fehlerfälle (egal ob auf der Client- oder auf der Server-Seite) im Falle von SOAP im so genannten SOAP Fault-Element kodiert und übertragen. Doch wie sieht dies bei ASP.NET-basierten XML Web Services aus?

Um diese Frage beantworten zu können, erstellen wir einen Web Service, der eine Ausnahme erzeugt. Der entsprechende Quelltext ist in Listing 5.13 zu finden.

```
<% @WebService Language="C#" Class="Fehler"%>
using System.Web.Services;

[WebService(Namespace="http://eyesoft.de/webservices/")]
public class Fehler
{
   [WebMethod]
   public int ErzeugeFehler()
   {
      throw new System.Exception("Fehler in
         SOAP-Server aufgetreten!");
   }
}
```

Listing 5.13: XML Web Service, der einen Fehler erzeugt

Die ASP.NET-Laufzeitumgebung sorgt hierbei dafür, dass die CLR-Ausnahme in eine SOAP-Darstellung umgewandelt wird. Der *XmlSerializer* veranlasst die Serialisierung dieses Fehlers in ein entsprechendes SOAP Fault-Element in der SOAP-Antwort (vgl. Listing 5.14). Wie man leicht sehen

kann, wird die im Web Service-Code erzeugte Ausnahme vom Typ *System.Exception* in eine Ausnahme vom Typ *System.Web.Services.Protocols.SoapException* eingebettet. Dies sollte man unbedingt beachten, wenn man eine Fehlerbehandlung und –überprüfung auf der Client-Seite (in .NET-Anwendungen) implementieren möchte.

```
HTTP/1.1 500 Internal Server Error.
Server: Microsoft-IIS/5.1
Date: Mon, 25 Feb 2002 15:16:53 GMT
Cache-Control: private
Content-Type: text/xml; charset=utf-8
Content-Length: 511

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
      <soap:Fault>
         <faultcode>soap:Server</faultcode>
            <faultstring>
               System.Web.Services.Protocols.
               SoapException: Server was unable to
               process request. ---&gt;
               System.Exception: Fehler in SOAP-Server
               aufgetreten!
               at Fehler.ErzeugeFehler()
               - End of inner exception stack trace -
            </faultstring>
         <detail />
      </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

Listing 5.14: Automatisch generierter SOAP Fault

5.3.3 XML Web Services benutzen

Das Erstellen von Web Services ist natürlich nur das halbe Leben. Irgendwann muss man für seine Web Services eine oder mehrere Client-Anwendungen erstellen. Alternativ hierzu hat man einen guten und wertvollen Web Service im Internet gefunden und möchte dafür einen Client mit grafischer Benutzungsoberfläche programmieren. Dieser Abschnitt widmet sich der Erstellung von Web Service-Clients durch manuelle und automatisierte Vorgehensweisen.

Ich sehe was, was du nicht siehst

Nachdem wir im ersten, theoretischen Teil teilweise sehr genau auf die Zusammensetzung von Anfragen an Web Services und deren Antworten in Ausprägung von SOAP-Nachrichten eingegangen sind, werde ich Ihnen jetzt kurz eine einfache Vorgehensweise vorstellen, wie man sich die tatsächlichen Nachrichten in einer Web Service-Anwendung anschauen kann.

Mithilfe von Werkzeugen wie *tcpTrace* (siehe Abbildung 5.10) kann ein Entwickler »unter die Haube« der SOAP-Kommunikation blicken. tcpTrace agiert dabei als Logging-Werkzeug, welches eingehende SOAP-Nachrichten in einem Fenster und auf Wunsch auch in eine Datei auf die Festplatte ausgibt. Technisch gesehen spielt tcpTrace für den Web Service-Client den eigentlichen Server. Der Client verbindet sich an den Port, auf dem tcpTrace hört (diesen Port kann man einstellen). Durch eine weitere Konfigurationseinstellung in tcpTrace wird – nachdem die Nachricht festgehalten wurde – das SOAP-Paket an den tatsächlichen SOAP-Server weiter geleitet. Abbildung 5.9 zeigt ein Beispiel, in dem tcpTrace auf Port 8085 hört und die Anfrage an den lokalen IIS auf Port 80 weiter gibt. Abbildung 5.10 zeigt tcpTrace in Aktion mit einer SOAP-Anfrage vom Client an den Server im oberen Fenster auf der rechten Seite und einer SOAP-Antwort vom Server an den Client im unteren Fenster (in der Abbildung wird der Währungsumrechner-Web Service angesprochen). Die Einstellung für den Endpunkt-URL kann man für den Web Service-Proxy nach dessen Erstellung mit folgender Anweisung bewirken:

```
proxyInstanz.Url = "http://localhost:8085/
   <URI zum Web Service>";
```

Zusätzlich zu tcpTrace gibt es noch die Schwesterapplikation *proxyTrace*. proxyTrace ist im Gegensatz zu tcpTrace ein HTTP-Proxy, der einfach über eine entsprechende Eigenschaft in der Web Service-Proxy-Klasse (siehe Abschnitt Proxy-basierter SOAP-Aufruf) wie im folgenden Beispiel aktiviert wird:

```
proxyInstanz.Proxy = new System.Net.WebProxy(
   http://<proxyservername>:<portnummer>/, true);
```

Der Wert *true* für den zweiten Parameter zeigt an, dass auch lokale Anfragen über den Proxy-Server gehen sollen. Mit dieser Eigenschaft lässt sich übrigens jede Web Service-Client-Anwendung über einen HTTP-Proxy betreiben.

tcpTrace und proxyTrace sind online zu finden unter *http://www.pocketsoap.com/tcptrace/* – oder auf der Begleit-CD.

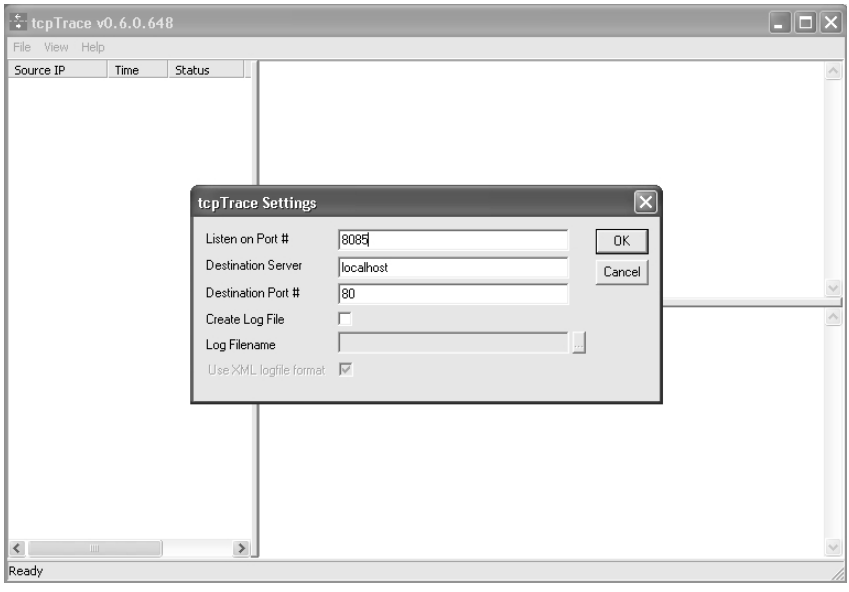


Abbildung 5.9: Konfigurationsmöglichkeiten für tcpTrace

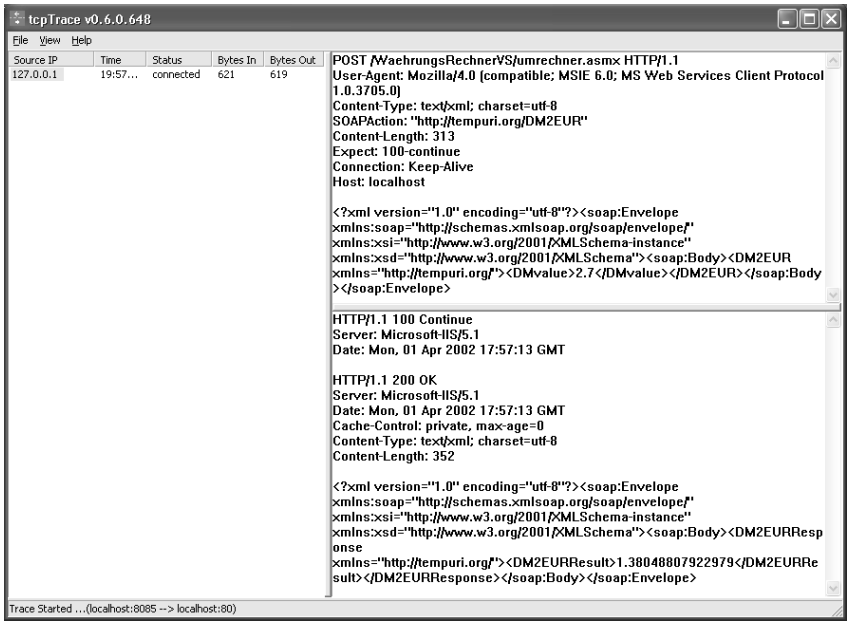


Abbildung 5.10: tcpTrace in Aktion

Manueller Aufruf über HTTP

Da es sich bei der Kommunikation mit XML Web Services im Grunde um eine HTTP-Kommunikation handelt, kann man eine Client-Anwendung relativ leicht programmieren. Alles, was man dafür benötigt, ist eine XML- und eine HTTP-Bibliothek. Mit der XML-Bibliothek lässt sich der SOAP

Envelope erzeugen und wieder parsen. Mit dem HTTP-Pendant werden die notwendigen HTTP Header gesetzt, um schließlich eine HTTP-POST-Operation an den Endpunkt zu schicken. Außerdem kann man zum Testen auch ganz einfache Clients programmieren, die sich, ähnlich wie die ASP.NET Hilfeseite für XML Web Services, per HTTP-GET-Operation zum Web Service verbinden.

Zur Veranschaulichung möchte ich hier die relevanten Teile einer kleinen Beispielanwendung (zu finden auf der CD) aufführen, die einen manuellen Aufruf von XML Web Services durch die Verwendung von Klassen aus der .NET Base Class Library ermöglicht. Die beiden tragenden Klassen für die HTTP-Funktionalität sind *System.Net.HttpWebRequest* für eine HTTP-Anfrage und *System.Net.HttpWebResponse* für eine HTTP-Antwort. Der SOAP Envelope wird intern in einer Helferklasse aufgebaut – ebenso wie die für den HTTP-Verkehr notwendigen Initialisierungen. Als Eingabeparameter muss ein Benutzer den *SOAPAction* HTTP Header und eine serialisierte Instanz der Methodenanfrage zur Verfügung stellen, und natürlich den Endpunkt-URL des Web Services. Als Ergebnis erhält man das Rückgabeelement aus der SOAP-Antwort.

Dieser manuelle Ansatz erfordert, dass man die in seiner Anwendung verwendeten Datentypen manuell in XML umwandelt und ebenso die SOAP-Antwort wieder parst und entsprechend den Variablen für die Rückgabewerte zuweist. Dies ist ein ungemein großer, unangemessener und vor allem fehleranfälliger Aufwand. Eine automatisierte Lösung sollte hier eine bessere Alternative sein.

Die wichtigsten Methoden der Helferklasse zur manuellen Kommunikation mit Web Services können Sie in Listing 5.15 sehen.

```
public class ManuelleWebServices
{
   private HttpWebRequest objHTTPReq;
   private HttpWebResponse objHTTPRes;

   public ManuelleWebServices(string url)
   {
      objHTTPReq =(HttpWebRequest)
         WebRequest.CreateDefault
         (new System.Uri(url));
   }

   public void setHeader(string name,
      string headerValue)
   {
      objHTTPReq.Headers.Add(name, headerValue);
   }

   public void send(string Body)
```

```
    {
        StreamWriter objStream;
        objHTTPReq.ContentType =
            "text/xml; charset=utf-8";
        objHTTPReq.Method = "POST";
        objStream = new
            StreamWriter(objHTTPReq.GetRequestStream(),
                Encoding.Default);
        objStream.Write(buildEnvelope(Body));
        objStream.Close();
    }

    private string buildEnvelope(string Body)
    {
        return "<?xml version='1.0' encoding=" +
            "'UTF-8'?><Envelope xmlns=" +
            "'http://schemas.xmlsoap.org/soap/" +
            "envelope/'>" +
            "<Body>" + Body + "</Body></Envelope>";
    }

    public string getResponse()
    {
        XmlDocument objXML = new XmlDocument();
        objHTTPRes = (HttpWebResponse)
            objHTTPReq.GetResponse();
        objXML.Load(objHTTPRes.GetResponseStream());

        return objXML.DocumentElement.FirstChild.
            InnerXml.ToString();
    }
}
```

Listing 5.15: Manuelle Kommunikation mit XML Web Services mithilfe der BCL

Proxy-basierter SOAP-Aufruf

Das .NET Framework SDK bietet einen automatisierten Ansatz zur Kommunikation mit XML Web Services, welcher auf dem Entwurfsmuster des Proxys basiert. Ein Proxy-Objekt ist ein lokales Stellvertreterobjekt, das im Namen eines anderen, meist entfernten Objekts agiert und die gleiche Funktionalität (im Sinne von Funktionen bzw. Methoden) zur Verfügung stellt wie das eigentliche Objekt. Das Schaubild in Abbildung 5.11 soll dieses Szenario verdeutlichen.

Dabei ist die Kommunikation nicht wie in der Abbildung angedeutet nur auf XML Web Services beschränkt, die mit ASP.NET implementiert sind. Man kann auch mit anderen XML Web Services kommunizieren – einzige Voraussetzung ist das Vorhandensein einer adäquaten WSDL-Beschreibung des

Web Services. Der Client-seitige Web Service-Proxy wird nämlich anhand der in der WSDL-Datei befindlichen Metadatendarstellung zur Entwurfszeit des Clients erzeugt. Dies geschieht im Rahmen des SDKs mit der Kommandozeilenanwendung *wsdl.exe*.

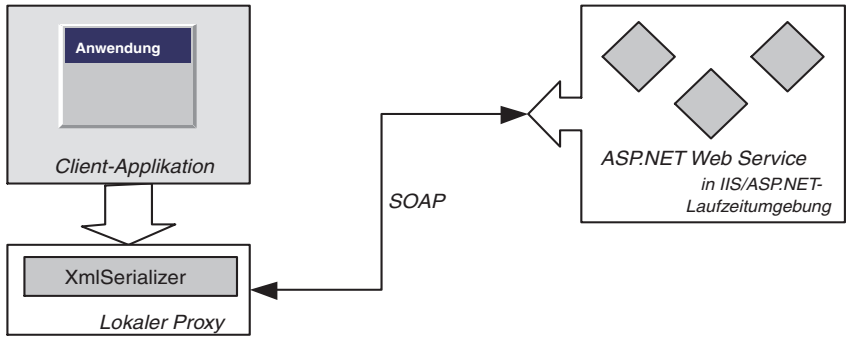


Abbildung 5.11: Proxy-basierte Kommunikation mit einem XML Web Service

Wsdl.exe kann Client-seitigen Proxy-Code erzeugen, der dann entweder in eine Assembly-DLL kompiliert oder als Quelltext in die Client-Anwendung integriert wird. Wsdl.exe besitzt eine Reihe von Optionen, z. B. um Benutzerinformationen für einen vorhandenen Proxy-Server anzugeben. Das Programm *wsdl.exe* öffnet eine gewöhnliche Netzwerkverbindung zum Server, auf dem die WSDL-Beschreibung liegt. Die wichtigsten Optionen sind in Tabelle 5.6 zusammengefasst. Die grundlegende Syntax für den Aufruf von wsdl.exe sieht folgendermaßen aus:

```
wsdl.exe [Optionen] {URL | Pfad für WSDL-Beschreibung}
```

Option	Beschreibung
`/l[anguage]:Sprache`	Gibt die Implementierungssprache an, in der der von *wsdl.exe* erzeugte Proxy generiert werden soll.
`/n[amespace]:Namensraum`	Gibt den CLR-Namensraum an, in dem die erzeugte Proxy-Klasse stehen soll.
`/o[ut]:Dateiname`	Gibt den Dateinamen der zu erstellenden Proxy-Klasse an.

Tab. 5.6: Die wichtigsten Optionen von wsdl.exe

Wir wollen nun eine Proxy-Klasse für unseren Währungsumrechner-Web Service erstellen. Hierzu führen wir in der Kommandozeile folgenden Befehl aus:

```
wsdl.exe http://localhost/wsbuch/K5/umrechner.asmx
   /l:CSharp /o:waehrungsrechner.cs
   /n:eYesoft.Web.Services.Client
```

Nachdem *wsdl.exe* seine Arbeit erledigt hat, befindet sich eine Datei namens *waehrungrechner.cs* im aktuellen Verzeichnis. Nun wollen wir uns etwas dem erzeugten C#-Code widmen (die relevanten Teile stehen in Listing 5.16). Die Proxy-Klasse ist von der Klasse *System.Web.Services.Protocols.SoapHttpClientProtocol* abgeleitet. Diese Klasse ist wiederum abgeleitet von *HttpWebRequest*, welche für die HTTP-basierte Kommunikation zuständig ist. In Verbindung mit der Basisklasse *SoapHttpClientProtocol* ist in unserer Proxy-Klasse die Eigenschaft *Url* sehr wichtig. Sie wird im Konstruktor gesetzt und verweist auf den eigentlichen Web Service-Endpunkt.

Des Weiteren ist der erzeugte Quelltext mit vielen Attributen ausgezeichnet. Diese Attribute sind fast ausschließlich für die Steuerung der XML-Serialisierung zuständig. Die meisten dieser Attribute sind bereits aus vorangegangenen Abschnitten in diesem Kapitel bekannt. Sie dienen dazu, dem auf der Client-Seite zuständigen *XmlSerializer* anzudeuten, dass die Umsetzung der CLR-Typen in XML-Daten nach einem Dokumenten-orientierten Ansatz erfolgen soll.

Erwähnenswert sind noch die Methoden für den asynchronen Aufruf des Web Services. Üblicherweise wird ein Web Service synchron und somit blockierend angesprochen. Der erzeugte Proxy stellt aber zusätzliche Methoden zur Verfügung, die es erlauben, eine einfache asynchrone Kommunikation mit dem Web Service zu realisieren, sodass der aufrufende Thread nicht blockiert wird. Ein Beispiel für diese etwas fortgeschrittene Methode wird uns in Kapitel 6 begegnen.

```
[System.Web.Services.WebServiceBindingAttribute
 (Name="WaehrungsRechnerSoap",
  Namespace="http://eyesoft.de/webservices/")]
public class WaehrungsRechner :
   System.Web.Services.Protocols.SoapHttpClientProtocol
{
   public WaehrungsRechner()
   {
      this.Url =
         "http://localhost/wsbuch/K5/umrechner.asmx";
   }

   [System.Web.Services.Protocols.
    SoapDocumentMethodAttribute(
    "http://eyesoft.de/webservices/DM2EUR",
    RequestNamespace="http://eyesoft.de/webservices/",
    ResponseNamespace="http://eyesoft.de/webservices/",
    Use=System.Web.Services.Description.SoapBindingUse.
    Literal,
    ParameterStyle=System.Web.Services.Protocols.
    SoapParameterStyle.Wrapped)]
```

```
public System.Double DM2EUR(System.Single DMvalue)
{
   object[] results = this.Invoke("DM2EUR",
      new object[]{DMvalue});
   return ((System.Double)(results[0]));
}

public System.IAsyncResult BeginDM2EUR(
   System.Single DMvalue, System.AsyncCallback
   callback, object asyncState)
{
   return this.BeginInvoke("DM2EUR",
      new object[] {DMvalue}, callback, asyncState);
}

public System.Double EndDM2EUR(
   System.IAsyncResult asyncResult)
{
   object[] results = this.EndInvoke(asyncResult);
   return ((System.Double)(results[0]));
}

[System.Web.Services.Protocols.
 SoapDocumentMethodAttribute(
 "http://eyesoft.de/webservices/EUR2DM",
 RequestNamespace="http://eyesoft.de/webservices/",
 ResponseNamespace="http://eyesoft.de/webservices/",
 Use=System.Web.Services.Description.SoapBindingUse.
 Literal,
 ParameterStyle=System.Web.Services.Protocols.
 SoapParameterStyle.Wrapped)]
public System.Double EUR2DM(System.Single EURvalue)
{
   object[] results = this.Invoke("EUR2DM",
      new object[] {EURvalue});
   return ((System.Double)(results[0]));
}

public System.IAsyncResult BeginEUR2DM(
   System.Single EURvalue, System.AsyncCallback
   callback, object asyncState)
{
   return this.BeginInvoke("EUR2DM",
      new object[] {EURvalue}, callback, asyncState);
}
```

```
    public System.Double EndEUR2DM(
        System.IAsyncResult asyncResult)
    {
        object[] results = this.EndInvoke(asyncResult);
        return ((System.Double)(results[0]));
    }
}
```

Listing 5.16: Erzeugter Web Service Proxy-Code

Damit ein Client-Programm nun mit diesem Proxy arbeiten kann, muss es lediglich eine Instanz der Proxy-Klasse erzeugen und die verfügbaren Methoden aufrufen. Ein Ausschnitt aus einer simplen Konsolenanwendung (ein Beispiel befindet sich auf der Begleit-CD) für die Benutzung unseres Währungsumrechners sieht folgendermaßen aus:

```
using eYesoft.Web.Services.Client;

WaehrungsRechner waeWebService = new WaehrungsRechner();
Console.WriteLine(waeWebService.DM2EUR(2.50F));
```

Dies ist tatsächlich alles, was ein Client-Programmierer tun muss, um mit einem Programm auf einen XML Web Service (mit vorhandener WSDL-Beschreibung) zuzugreifen. Somit lässt sich zusammenfassend der Lebenszyklus eines ASP.NET XML Web Services mitsamt Client-seitigem Aufruf wie in Abbildung 5.12 darstellen.

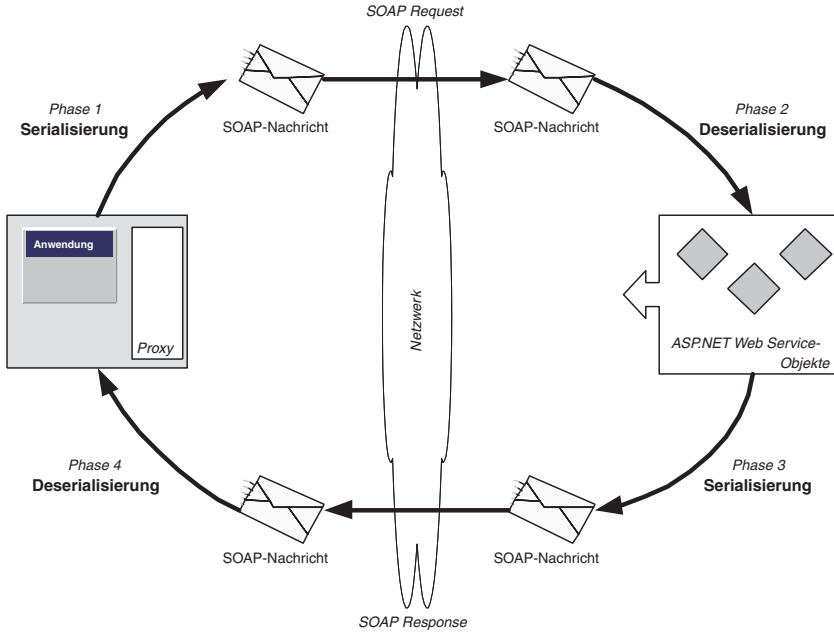


Abbildung 5.12: Lebenszyklus eines ASP.NET XML Web Services

Mit der Option */server* erzeugt *wsdl.exe* eine Proxy-Klasse auf Basis einer WSDL-Beschreibung, die als Grundlage für eine Server-seitige Web Service-Implementierung dienen kann. Da WSDL im Grunde nichts anderes als ein Vertrag oder eine Schnittstellenbeschreibung (u.a.) ist, kann es dafür verwendet werden, dass auf Basis dieser Beschreibung eine Gerüst-Klasse für eine konkrete Implementierung in ASP.NET erstellt wird.

Mit der in diesem Kapitel vorgestellten, einfachen Vorgehensweise lassen sich sowohl XML Web Services als auch Client-Anwendungen für XML Web Services auf Basis von ASP.NET und dem *XmlSerializer* sehr einfach implementieren. Man kann dies prinzipiell alles mit Notepad und der Kommandozeile erledigen. Allerdings wünscht man sich bei umfangreicheren Projekten schnell die Unterstützung durch eine integrierte Entwicklungsumgebung.

Wenn sich XML Web Services als das im Internet verfügbare Pendant zu lokalen Softwarekomponenten durchsetzen soll, dann bedarf es einer ausgeklügelten und gut durchdachten Infrastruktur zum Suchen und Finden von Web Services. Diesem Thema nimmt sich u.a. die *UDDI*-Technologie (*Universal Description, Discovery, and Integration*) an. Im Rahmen dieses Buches kann ich leider nicht auf die diversen Ansätze zum Auffinden von XML Web Services mit und ohne UDDI eingehen.

5.4 XML Web Services-Anwendungen mit Visual Studio .NET

Am Ende dieses Kapitels wollen wir den weiter oben vorgestellten XML Web Service zur Umrechnung von DM und Euro mit dem Visual Studio .NET programmieren. Hinzu kommt dann noch eine Windows-Anwendung, die über das Proxy-Muster mit dem Web Service kommuniziert.

5.4.1 Erstellen eines XML Web Services

Ähnlich wie bei der Entwicklung von ASP.NET Web Forms-Anwendungen bietet uns das Visual Studio auch für den Bereich XML Web Services eine Projektvorlage. Durch Auswahl von DATEI – NEU – PROJEKT gelangt man in den Auswahlbildschirm für die existierenden Vorlagen (siehe Abbildung 5.13). Wir wollen uns für ein ASP.NET WEB SERVICE-PROJEKT in C# entscheiden. In dieser Maske wird ausgewählt, auf welchem Web-Server (IIS) das Projekt angelegt wird und wie die Web-Anwendung heißen soll (den gleichen Namen bekommt dann auch das Projekt im Visual Studio). Durch Klicken auf die Schaltfläche OK wird dann - ähnlich wie bei ASP.NET Web Forms in Kapitel 3 - eine IIS-Anwendung erzeugt (siehe Abbildung 5.14).

Zusätzlich werden noch die Projektdatei und einige weitere Dateien angelegt, die entweder unbedingt benötigt werden oder aber dem Entwickler das Leben leichter machen.

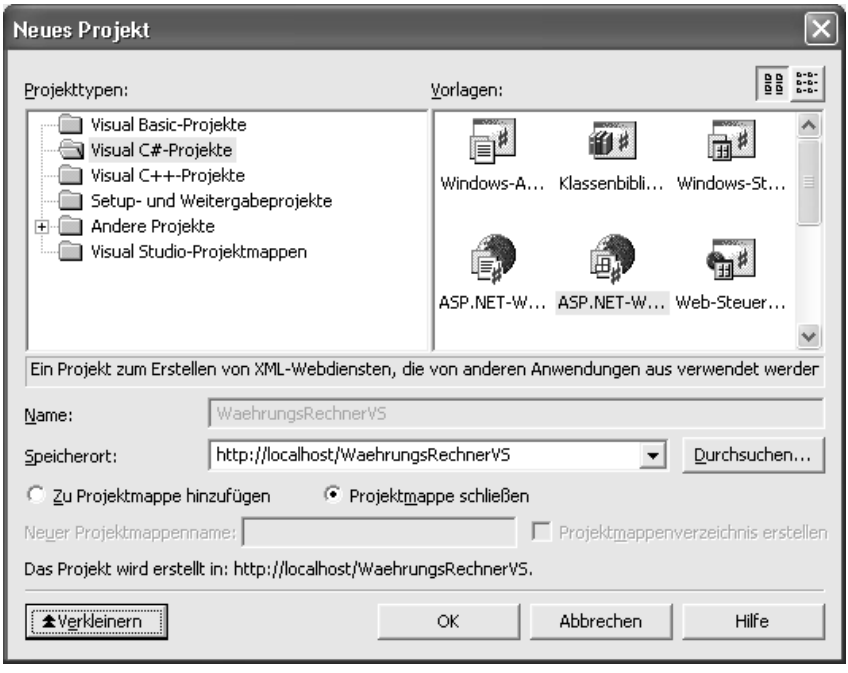


Abbildung 5.13: Anlegen eines ASP.NET XML Web Service-Projektes im Visual Studio .NET

Abbildung 5.14: Anlegen einer IIS-Anwendung für den XML Web Service durch Visual Studio .NET

Nachdem der Vorgang der automatischen Erstellung der Projektdateien und das Anlegen der notwendigen Web-Anwendung im IIS abgeschlossen ist, präsentiert sich Visual Studio .NET mit einer Oberfläche ähnlich wie in Abbildung 5.15. Der Entwickler bekommt zunächst keinen Quelltext, sondern die Entwurfsansicht präsentiert. Hier könnte er nun – wieder ähnlich wie bei Web Forms-Applikationen – Kontrollelemente zur Entwurfszeit auf die Oberfläche ziehen und Einstellungen z.B. für eine Datenbankverbin-

dung vornehmen. Im rechten Teil der Abbildung befinden sich die automatisch erzeugten Dateien für das Projekt. Die wichtigste Datei ist *Service1.asmx* (zusammen mit der Code-Behind-Implementierung des Web Services *Service1.asmc.cs*). Bitte nennen Sie zunächst die Datei in *umrechner.asmx* um.

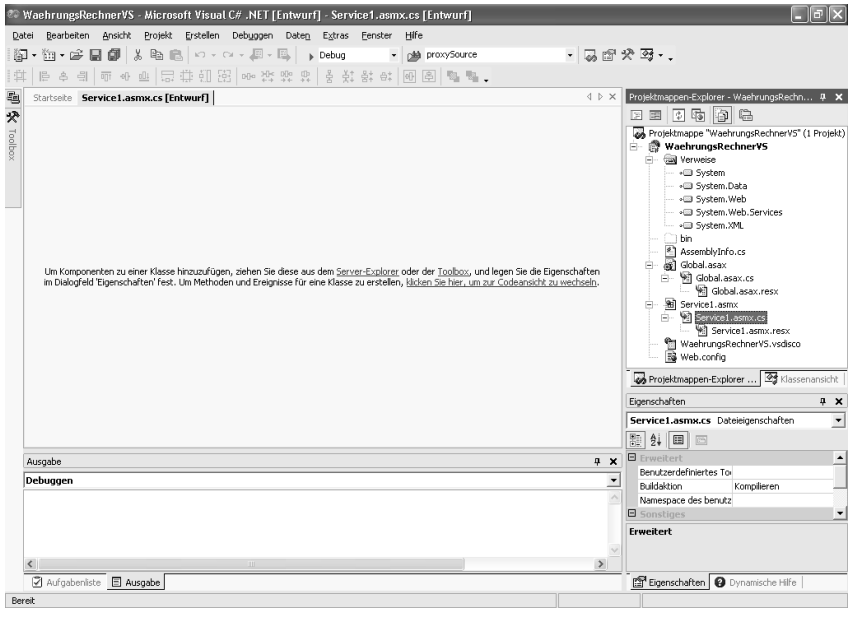


Abbildung 5.15: ASP.NET Web Service-Projekt im Visual Studio .NET

Um nun zur Quelltextansicht zu gelangen, klicken Sie bitte in der Entwurfsansicht auf die entsprechend markierte Stelle (alternativ könnten Sie auch mit der rechten Maustaste auf die Datei *umrechner.asmx* zeigen und per Kontextmenü den Menüpunkt CODE ANZEIGEN auswählen). Die Code-Ansicht ist der aus Abbildung 5.16 ähnlich. Wie wir erkennen können, erstellt Visual Studio einen komplett funktionsfähigen ASP.NET Web Service für uns. Wenn wir bei der auskommentierten *Hello World*-Methode noch die Kommentarzeichen entfernen, dann können wir nach der Erstellung (Kompilierung) des Projekts diesen einfachen Web Service testen. Doch wir wollen diesen Schritt hier auslassen und gleich die notwendigen Änderungen für den Währungsumrechner vornehmen.

Bitte ändern Sie den Quelltext in *umrechner.asmx.cs* so ab, dass er dem Code in Abbildung 5.17 gleicht. Dies ist im Grunde der gleiche Code wie wir ihn bereits weiter oben im Kapitel für unseren Web Service verwendet haben, der mit dem einfachen Texteditor programmiert wurde.

Um das Projekt nun zu kompilieren – wir wissen ja noch aus Kapitel 3, dass Visual Studio .NET immer den Code-Behind-Ansatz mit vorkompilierten Assemblies verwendet – wählen wir aus dem Menü ERSTELLEN den Menüpunkt WAEHRUNGSRECHNERVS ERSTELLEN.

Abbildung 5.16: Quelltextvorgabe von Visual Studio .NET

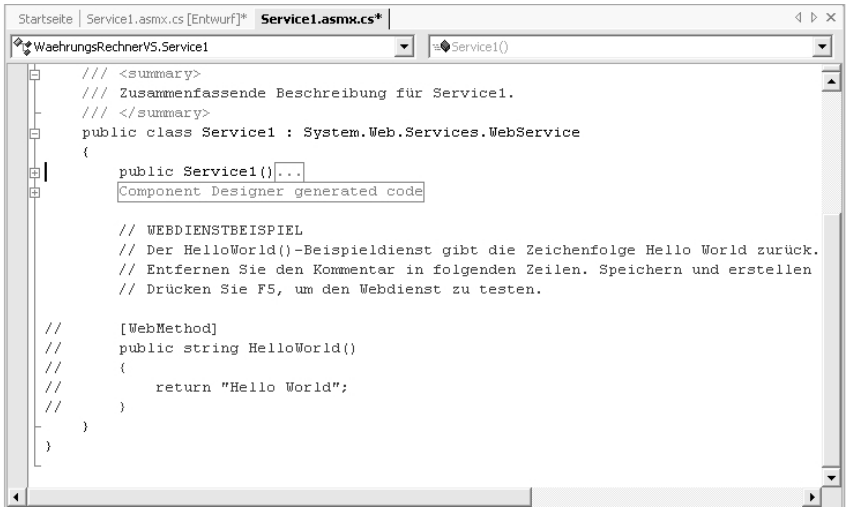
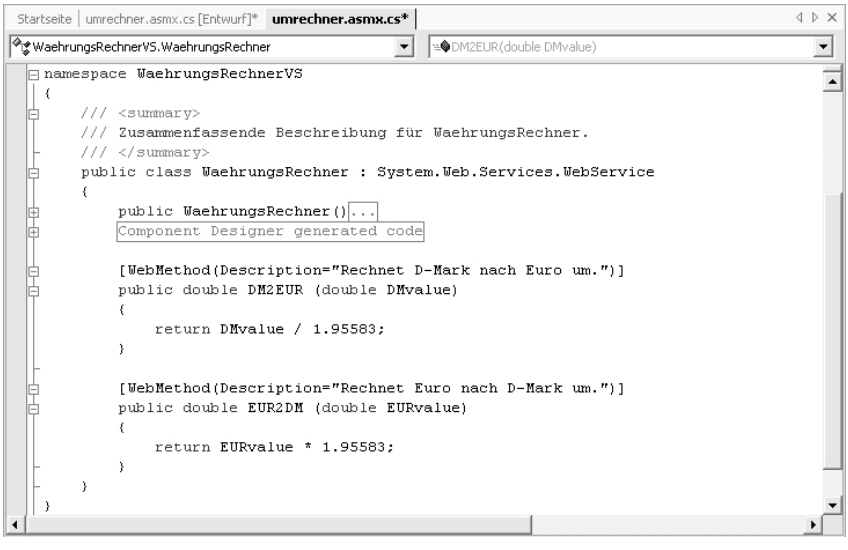


Abbildung 5.17: Geänderter Quelltext für den Währungsumrechner-Web Service im Visual Studio .NET



Alternativ zur Menüauswahl kann ein Projekt bzw. die gesamte Projektmappe auch einfach per Tastenkombination kompiliert werden: Strg + Umschalt + B.

Wir können unseren XML Web Service nun testen, indem wir in den Debug-Modus schalten oder einfach die *umrechner.asmx*-Seite aufrufen. Der erste Fall wird über die Taste F5 erreicht und ermöglicht das Setzen von Unterbrechungspunkten im Quelltext, um die Funktionalität im Debugger von Visual Studio .NET auf Herz und Nieren zu überprüfen. Im zweiten Fall

wählen Sie mit der rechten Maustaste durch Zeigen auf die Datei *umrechner.asmx* aus dem Kontextmenü den Punkt IN BROWSER ANZEIGEN. Dann wird der auf dem Internet Explorer basierende Web-Browser gestartet und es erscheint die bereits bekannte Hilfeseite für ASP.NET XML Web Services. Hier können Sie nun den Web Service testen, wie dies bereits weiter oben erläutert wurde.

Nach nur wenigen Minuten haben wir einen komplett funktionsfähigen ASP.NET XML Web Service mit dem Visual Studio .NET erstellt und getestet. Gehen wir nun an die nächste Aufgabe und schreiben eine kleine Client-Anwendung für unseren Web Service.

5.4.2 Erstellen einer Windows-Client-Anwendung

Wir werden im Folgenden eine Windows Forms-Anwendung erstellen, die unseren Währungsumrechner-Web Service als externe Komponente verwenden soll. Hierfür wählen wir wieder den Menüpunkt DATEI – NEU – PROJEKT aus. Wählen Sie nun für die Sprache C# die Option WINDOWS-ANWENDUNG aus. Stellen Sie hierbei aber sicher, dass die Option ZU PROJEKTMAPPE HINZUFÜGEN aktiviert ist. Diese Option wird das neue Projekt in unsere bereits erstellte Projektmappe einfügen, sodass wir den Web Service und den Client innerhalb einer Mappe vorliegen haben (vgl. Abbildung 5.18).

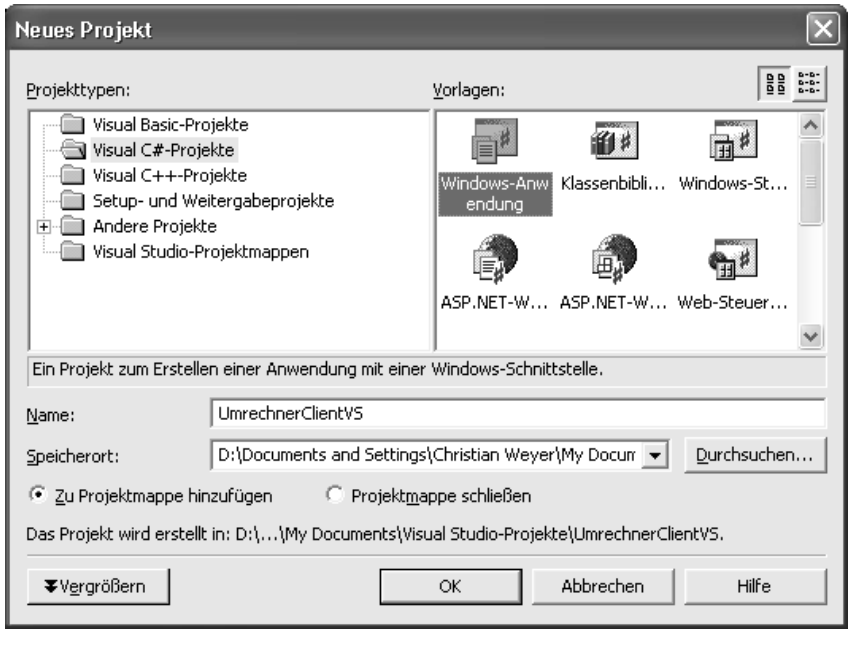


Abbildung 5.18: Projektvorlage für den Windows Forms-Client in Visual Studio .NET

Nach Erstellung des Projekts erscheint ein Windows-Formular in der Entwurfsansicht. Bitte verwenden Sie die TOOLBOX am linken Rand der Entwicklungsumgebung, um eine ähnliche Oberfläche zu erstellen, wie sie in Abbildung 5.19 zu sehen ist. Die Oberfläche ist sehr einfach gehalten. Sie bietet eine Eingabemöglichkeit für den Geldbetrag in der Ausgangswährung und zwei Schaltflächen für die Umrechnung des Betrags. Das Ergebnis wird dann in dem Formular angezeigt.

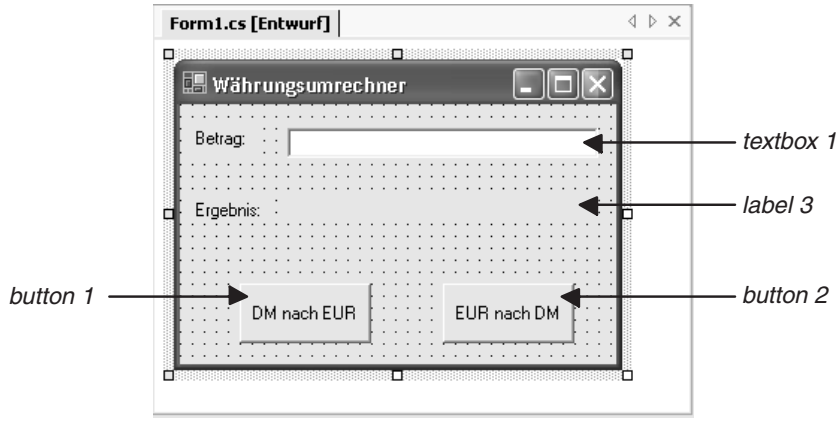


Abbildung 5.19: Oberfläche der Client-Anwendung in der Entwurfsansicht im Visual Studio .NET

Ich werde an dieser Stelle nicht weiter auf die detaillierte Erstellung von Windows Forms-Applikationen eingehen.

Wir müssen nun unsere Anwendung noch mit Leben füllen. Zunächst müssen wir der Anwendung mitteilen, dass wir eine externe Ressource, nämlich einen XML Web Service, verwenden möchten. Visual Studio .NET bietet hierfür eine ähnliche Funktionalität wie *wsdl.exe* in der Kommandozeile. Wählen Sie mit der rechten Maustaste durch Zeigen auf das Client-Projekt den Punkt WEBVERWEIS HINZUFÜGEN aus dem Kontextmenü aus (siehe Abbildung 5.20). Es erscheint ein Dialog zur Eingabe des URLs für den Endpunkt oder die WSDL-Beschreibung des gewünschten Web Services (siehe Abbildung 5.21). Eigentlich benötigt dieser Dialog eine WSDL-Beschreibung, um die geforderte Proxy-Klasse erstellen zu können. Im Falle einer .asmx-Datei kann die Logik selbst feststellen, dass es sich hier um einen ASP.NET Web Service handelt und gelangt durch Anhängen von *?wsdl* zu der automatisch erzeugten WSDL-Darstellung des Web Services.

Im Dialog kann man sich zusätzlich auch das WSDL durch Klicken auf VERTRAG ANZEIGEN anschauen. Nachdem man auf die Schaltfläche VERWEIS HINZUFÜGEN geklickt hat, erstellt Visual Studio .NET, ähnlich wie *wsdl.exe*, eine Proxy-Klasse aus der WSDL-Beschreibung.

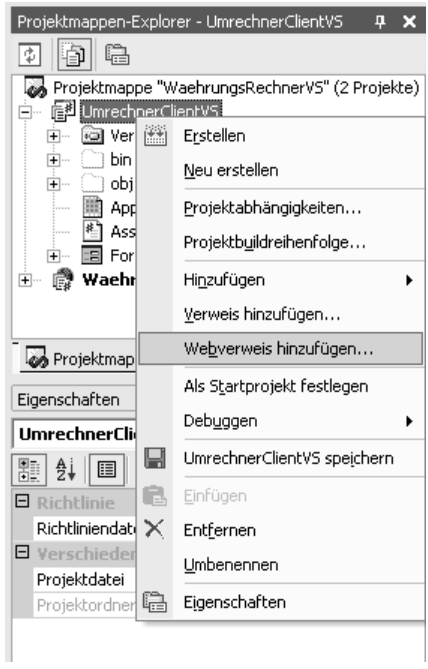


Abbildung 5.20: Aufruf des Dialogs zur Erstellung eines Webverweises im Visual Studio .NET

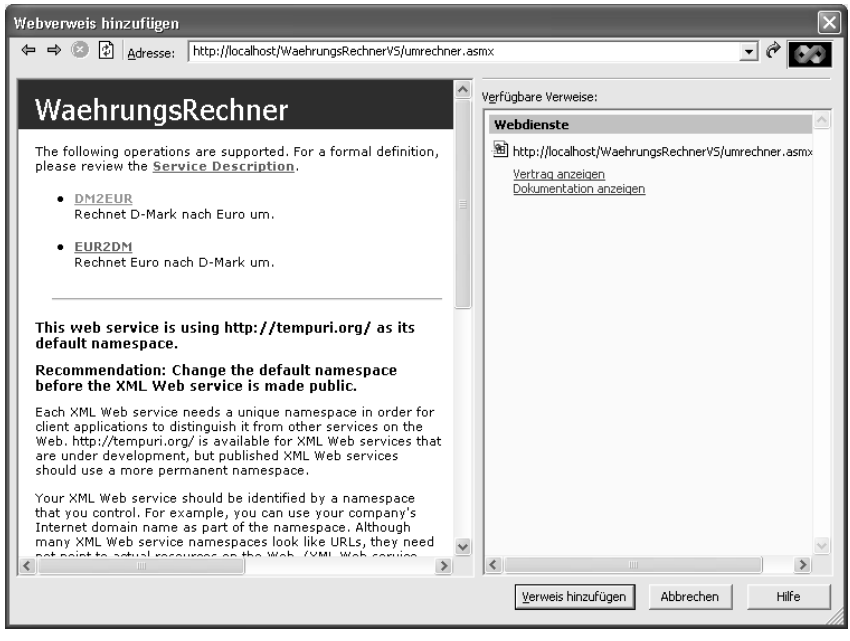


Abbildung 5.21: Dialog für das Hinzufügen eines Webverweises auf einen XML Web Service im Visual Studio .NET

Die relevante Datei ist hier *Reference.cs* (vgl. Abbildung 5.22). Nachdem wir nun eine Proxy-Klasse für den Web Service erzeugt haben, können wir die Anwendung fit machen für den Aufruf des Währungsumrechners.

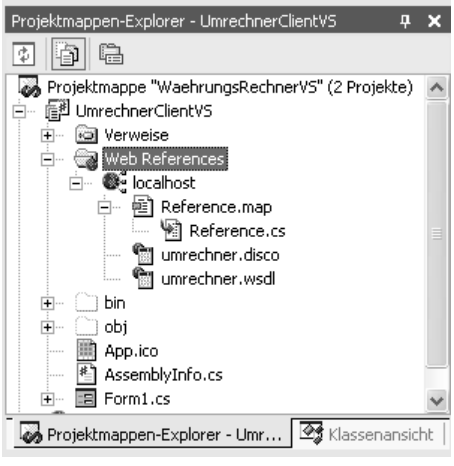


Abbildung 5.22: Erstellte Proxy-Klassen-Dateien für die Nutzung des XML Web Services im Visual Studio .NET

Hierfür doppelklicken Sie bitte auf die linke Schaltfläche. Sie gelangen nun in die Quelltextansicht. Durch den Doppelklick auf die Schaltfläche hat Visual Studio ein Event für die Schaltfläche für Sie angelegt, welches wir nun noch mit unserem Code füllen müssen. Bevor Sie aber hier starten, fügen Sie bitte noch folgende Zeile oben in der Datei ein:

```
using UmrechnerClientVS.localhost;
```

Mit dieser Anweisung importieren wir den Namensraum, in dem die Proxy-Klasse steht. Der Webverweis steht standardmäßig im Namensraum *Projektname.Servername*. Diese Zeile erleichtert uns die Schreib- und Programmierarbeit. In das Ereignis für unsere erste Schaltfläche fügen Sie bitte folgenden Quelltext ein:

```
private void button1_Click(object sender,
   System.EventArgs e)
{
   WaehrungsRechner waeRechner = new
      WaehrungsRechner();
   label3.Text =
      (waeRechner.DM2EUR(Convert.ToDouble(
         textBox1.Text))).ToString();
}
```

Hier erzeugen wir eine Instanz des Proxy-Objekts, rufen die Methode zum Konvertieren von DM nach Euro auf und weisen den Rückgabewert einem Text-Label zu.

Das Gleiche wiederholen Sie bitte für die zweite Schaltfläche. Zunächst wechseln Sie in die Entwurfsansicht, führen einen Doppelklick aus und fügen folgenden Code ein:

```
private void button2_Click(object sender,
   System.EventArgs e)
{
   WaehrungsRechner waeRechner = new
      WaehrungsRechner();
   label3.Text =
      (waeRechner.EUR2DM(Convert.ToDouble(
         textBox1.Text))).ToString();
}
```

Damit ist unsere Client-Anwendung für den Währungsumrechner fertig. Wir müssen das Projekt nur noch kompilieren und testen. Zum Erstellen wählen Sie entweder den entsprechenden Menüpunkt (siehe oben) oder Sie drücken einfach F5 zum Starten des Debug-Vorgangs. Nun können Sie unsere Web Service-Client-Anwendung testen (siehe auch Abbildung 5.23).

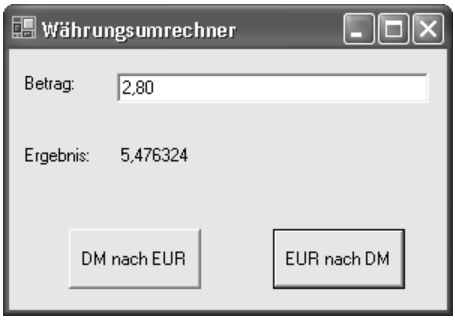


Abbildung 5.23: Fertige Client-Anwendung mit Beispieleingabe im Visual Studio .NET

Zum Schluss möchte ich noch anmerken, dass man die gleiche Vorgehensweise auch bei einer ASP.NET Web Forms-Applikation anwenden kann. Dann agiert die ASP.NET-Anwendung als Client für den ASP.NET Web Service. Der Benutzer greift dann über den Web-Browser auf die Web Forms-Seiten zu. Zu beachten sind bei dieser Lösung einige Dinge, z. B. im Bereich Sicherheit oder Sitzungsverwaltung. Doch mehr dazu in den kommenden Kapiteln.

Die hier gezeigte Vorgehensweise zur Instanziierung des Proxy-Objekts und die anschließende Nutzung ist nicht optimal. Normalerweise würde man ein für das Formular globales Proxy-Objekt anlegen, auf welches dann innerhalb des Formulars zugegriffen werden kann. Die oben dargestellte Lösung wurde aufgrund der besseren Übersichtlichkeit gewählt.

Es besteht auch die Möglichkeit, Web Service-Client-Anwendungen zu programmieren, die komplett im Browser ablaufen. Damit ist nicht eine ASP.NET-Lösung, sondern eine auf JavaScript und aktiven Elementen basierende Realisierung gemeint. Von Microsoft gibt es hierfür das so genannte *Internet Explorer (IE) WebService Behavior*. Hierbei handelt es sich um eine DHTML-Vorlage (*.htc*-Datei) für den IE, welche die skriptgesteuerte Kommunikation mit XML Web Services ermöglicht. Zu beziehen ist das IE WebService Behavior über MSDN: *http://msdn.microsoft.com/workshop/author/webservice/webservice.htc*

5.5 Zusammenfassung

SOAP, WSDL und auch XML sind teilweise sehr trockene und staubige Themen, deren Kenntnis aber bei der Entwicklung und dem Betrieb von XML Web Services durchaus von Nutzen sein können. Aufgrund des relativ geringen Alters der beiden Haupttechnologien SOAP und WSDL gibt es noch viele Ungereimtheiten – vor allem im Bereich Interoperabilität.

Alles in allem ist ASP.NET eine mächtige Laufzeitumgebung, auch für Web Services. Zusammen mit dem *XmlSerializer* werden automatisch die Typen zwischen der .NET CLR- und der SOAP-XSD-Welt ausgetauscht. Bei Bedarf kann ein Entwickler aber auch manuell eingreifen. Durch die Proxy-basierte Client-Programmierung bleibt dem Entwickler viel Arbeit und Mühe erspart – somit wird seine Produktivität gesteigert und er kann sich auf die wesentlichen Dinge konzentrieren.

6 Fast wie im richtigen Leben

ASP.NET XML Web Services im Einsatz

6.1 Einführung

Microsoft propagiert mit seiner .NET-Initiative vor allem den XML-basierten Ansatz zur plattformübergreifenden Kommunikation zwischen Anwendungen.

Nachdem ich Ihnen in Kapitel 5 die notwendigen technischen Grundlagen für XML Web Services und deren prinzipielle Umsetzung in ASP.NET vorgestellt habe, wollen wir in diesem Kapitel nun verschiedene Szenarien beleuchten, die beim täglichen Einsatz von Web Services von großer Bedeutung sein können. Denn XML Web Services sind nicht immer und für jedes Problem die beste Lösung. Oftmals fehlt es noch an echten Standards oder das grundlegende Konzept der Web Services ist einfach ungeeignet für den Einsatz in speziellen Umgebungen.

Auf den folgenden Seiten werde ich Ihnen Lösungsansätze und Implementierungen vorstellen, wie man immer wieder auftretende Muster von Anwendungen auf Basis von ASP.NET XML Web Services realisieren kann. Angefangen von einer asynchronen Kommunikation der Client-Anwendungen mit den Web Services, über die Nutzung von existierenden COM-Komponenten und der COM+-Dienstinfrastruktur, bis hin zur Verwaltung großer Datenmengen in XML Web Services.

6.2 Asynchrone Web Service-Kommunikation

Die Vorgehensweise für den Aufruf von Web Services über einen Proxy haben Sie bereits im vorangegangenen Kapitel kennengelernt. Hierfür wird eine Proxy-Klasse aus einer WSDL-Beschreibung heraus generiert und eine Instanz dieser Klasse in der Client-Anwendung erzeugt. Der Client greift dann über den Proxy auf die Web Service-Funktionalität zu, als wäre es ein lokales Objekt.

Diese Kommunikation erfolgt in synchroner Art und Weise. Dies bedeutet, dass der folgende Aufruf einer Web Service-Methode das aufrufende Programm blockiert:

```
WaehrungsRechner waeWebService = new WaehrungsRechner();
Console.WriteLine(waeWebService.DM2EUR(2,50));
```

Der Client ist in diesem Fall solange blockiert, bis der Aufruf des Web Services wieder zurückkehrt und das Ergebnis liefert. Somit entspricht diese Art der Programmierung dem oberen Teil in Abbildung 6.1. Vor allem bei Anwendungen mit einer grafischen Benutzungsoberfläche ist es sehr ärgerlich, wenn die gesamte Applikation einfriert, nur weil eine etwas länger dauernde Aktion ausgeführt werden muss. Da ein Web Service-Aufruf durchaus einige Sekunden dauern kann, ist diese Tatsache umso unerfreulicher. Es sollte also eine adäquate Lösung für solche Problemsituationen gefunden werden.

Ein möglicher Ansatz ist es, im Client einen neuen Thread zu starten, der sich um die Kommunikation mit dem Web Service kümmert. Während dieser Thread die Anfrage bearbeitet, kann die Anwendung normal weiterlaufen und der Benutzer kann zusätzliche Eingaben vornehmen. Allerdings ist die Programmierung mit Threads nicht immer trivial. In unserem Fall müssten wir uns von dem neu angelegten Thread benachrichtigen lassen, sobald der Web Service-Aufruf beendet ist – und das Thema der Thread-Synchronisation bzw. Benachrichtigung eines Threads aus einem anderen heraus ist mitunter aufwendig zu implementierten. Doch das .NET Framework bietet uns hier eine Lösung an. Sie werden diese Lösung vielleicht schon beim Durchschauen des Proxy-Codes in Kapitel 5 entdeckt haben. Der Vollständigkeit halber sind die relevanten Teile des Quelltextes noch einmal in Listing 6.1 abgebildet. Der Code zeigt zwei Methoden: *BeginDEM2EUR* und *EndDM2EUR*. Diese beiden Methoden werden zusätzlich zu der »normalen« Web Service-Methode *DM2EUR* in der Proxy-Klasse erzeugt.

```
public System.IAsyncResult BeginDM2EUR(
   System.Single DMvalue, System.AsyncCallback
   callback, object asyncState)
{
   return this.BeginInvoke("DM2EUR",
      new object[] {DMvalue}, callback, asyncState);
}

public System.Double EndDM2EUR(
   System.IAsyncResult asyncResult)
{
   object[] results = this.EndInvoke(asyncResult);
   return ((System.Double)(results[0]));
}
```

Listing 6.1: Methoden für die asynchrone Kommunikation im erzeugten Proxy-Code

Schauen wir uns diese beiden Methoden einmal genauer an. Die Methode *BeginXXX* nimmt drei Parameter entgegen: den eigentlichen Parameterwert für den Geldbetrag, einen Parameter vom Typ *AsynchCallback* und einen *object*-Typ. Auf diese Parameter werde ich gleich noch eingehen. Die Implementierung der Methode ruft einfach die Methode *BeginInvoke* auf. Allgemein muss *BeginXXX* alle Eingabe- und Referenzparameter einer Web

Service-Methode entgegennehmen. Der Rückgabewert ist vom Typ *IAsynch-Result*. Dieser Typ kapselt die Informationen über eine asynchrone Aktion und wird benötigt, wenn die eigentliche Verarbeitung des Ergebnisses stattfindet. Die Abarbeitung der Web Service-Anfrage wird intern an einen aus dem Thread-Pool der CLR bezogenen Thread delegiert. Sämtliche notwendigen Schritte werden für den Entwickler in der *BeginXXX*-Methode gekapselt. Der untere Teil in Abbildung 6.1 stellt diese Art der Kommunikation bildlich dar.

Das eigentliche Ergebnis des Aufrufs an den Web Service erhält man nun durch die Methode *EndXXX*. Außer dem Rückgabewert ist diese Methode auch noch für eventuell vorhandene Referenzparameter der Web Service-Methode zuständig. Denn diese werden ja im Web Service gesetzt und sind somit eine Art zusätzlicher Rückgabewert.

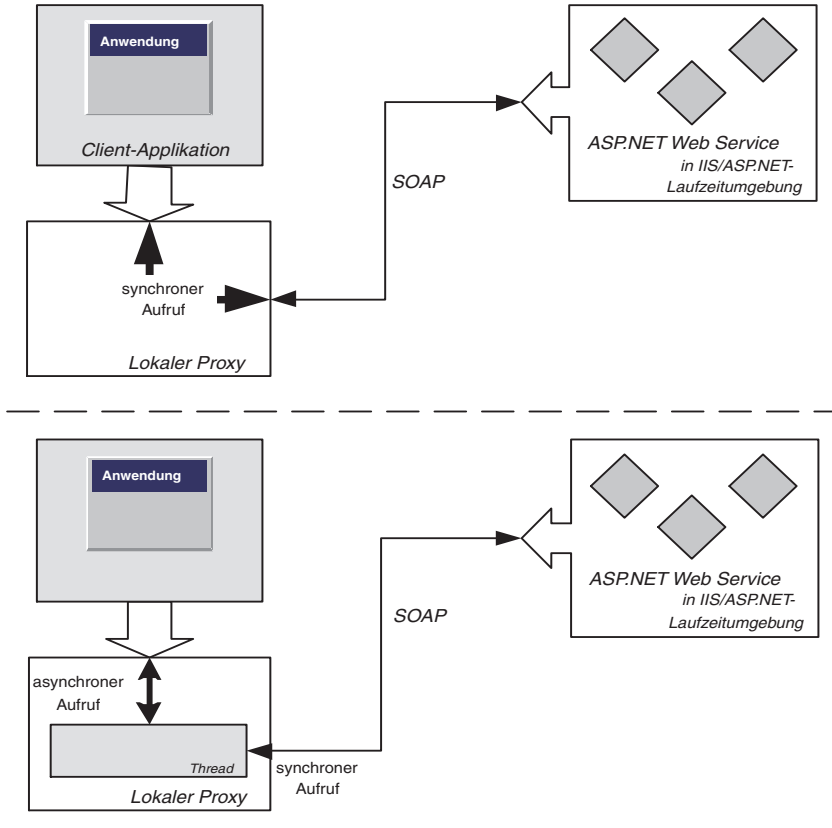


Abbildung 6.1: Synchroner und asynchroner Aufruf eines XML Web Service

Um nun tatsächlich ein Ergebnis von einem asynchronen Aufruf zu erhalten, gibt es mehrere Möglichkeiten, von denen ich eine hier verwenden möchte: die Verwendung von so genannten Rückruf-Delegates (*callback delegates*).

Damit die Erklärung der Vorgehens- und Funktionsweise an einem Beispiel vereinfacht werden kann, verwenden wir hier das Windows Forms-Projekt für den Währungsumrechner-Client aus Kapitel 5. Hierfür wandeln wir die Funktion zum Aufruf der Konvertierung von DM nach Euro in einen asynchronen Aufruf um. Das Ergebnis steht in Listing 6.2.

```
private void button1_Click(object sender,
   System.EventArgs e)
{
   WaehrungsRechner waeRechner =
      new WaehrungsRechner();
AsyncCallback asCb = new
   AsyncCallback(DM2EURCallback);

waeRechner.BeginDM2EUR(Convert.ToDouble(textBox1.Text),
   asCb, waeRechner);
}

protected void DM2EURCallback(System.IAsyncResult ar)
{
   WaehrungsRechner waeRechner =
      (WaehrungsRechner)ar.AsyncState;

   double resultValue = waeRechner.EndDM2EUR(ar);
   label3.Text = resultValue.ToString();
}
```

Listing 6.2: Asynchroner Aufruf einer Web Service-Methode im Client

Die Vorgehensweise ist so, dass die Abarbeitung des Web Service-Aufrufs über ein Objekt vom Typ *AsyncCallback* gesteuert wird. Dem Konstruktor für dieses Objekt geben wir einen Verweis auf unsere Rückrufmethode *DM2EURCallback* mit. Den eigentlichen Aufruf starten wir dann durch *waeRechner.BeginDM2EUR*. Als Argumente übergeben wir den Eingabewert für DM und zwei weitere Parameter. Der zweite von diesen ist das vorher erzeugte Delegate-Objekt vom Typ *AsyncCallback* für die Rückrufmethode. Zusätzlich übergeben wir noch die Instanz unseres Web Service-Proxys, damit wir innerhalb der Rückrufmethode Zugriff auf diese Instanz haben. Innerhalb dieser Methode extrahieren wir die übergebene Proxy-Instanz aus dem Parameter `ar` durch Auslesen der Eigenschaft *AsyncState*. Diese Eigenschaft ruft ein benutzerdefiniertes Objekt ab, das einen asynchronen Vorgang beschreibt oder Informationen darüber enthält – in unserem Fall eben das Proxy-Objekt. Einmal an die Proxy-Instanz gekommen, können wir nun innerhalb der Rückrufmethode *EndDM2EUR* aufrufen und in unserem Fall den Rückgabewert dem Label auf dem Formular zuweisen.

Dies ist die einfachste Methode, eine asynchrone Kommunikation zwischen einem .NET Client und einem XML Web Service herzustellen. In unserem Beispiel mit dem Währungsumrechner stellt sich der Effekt des asynchronen

Aufrufs nicht so spektakulär dar, da keine großen Datenmengen angefordert und übertragen werden. Stellt man sich jedoch einen Web Service vor, der beispielsweise Bilddaten (siehe Abschnitt 6.3.4) oder große Datenblöcke aus einer Datenbank übertragen muss, dann spielt diese Methode schnell ihre Vorteile aus.

6.3 XML-Serialisierung unter der Haube

Im .NET Framework gibt es zwei Möglichkeiten, Objekte zu serialisieren – also in eine persistente Darstellung zu überführen. Zum einen die binäre und zum anderen die XML-basierte Serialisierung. Ein Serialiserungsszenario besteht in der Regel aus zwei Vorgängen: der Serialisierung und der Deserialisierung, bei der eine serialisierte Instanz innerhalb eines Datenstroms (*Stream*) wieder in ein Objekt umgewandelt wird. Wir werden uns in diesem Abschnitt näher mit der XML-Serialisierung auseinander setzen, da sie unabdingbar für das Funktionieren von XML Web Services in ASP.NET ist. Außerdem kann man den Serialisierungsprozess von außen beeinflussen, was im Umgang mit anderen Web Services-Plattformen oft eine große Hilfe ist.

6.3.1 Überblick

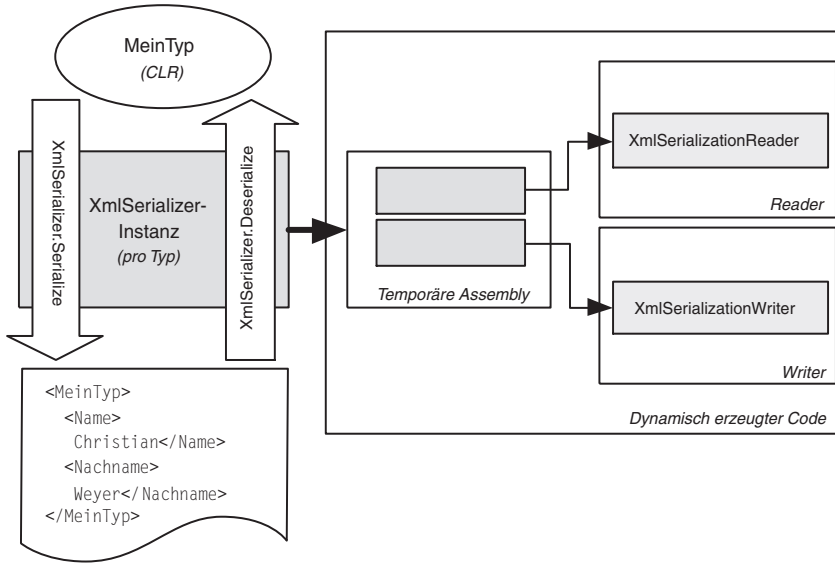
Bei der XML-Serialisierung gibt es zwei unterschiedliche Ansätze: zum einen die Serialisierung mit dem *SoapFormatter* (Namensraum *System.Runtime.Serialization*) und zum anderen mit dem *XmlSerializer* (Namensraum *System.Xml.Serialization*). Schon an den unterschiedlichen Namensräumen kann man den Hauptunterschied erahnen. Der *SoapFormatter* wird in der .NET Remoting-Architektur (siehe Kapitel 8) eingesetzt und ist für die Serialisierung aller CLR-Typen gedacht. Der *XmlSerializer* hat dagegen das XML Schema-Datenmodell im Fokus und versucht eine größtmögliche Kompatibilität mit anderen Implementierungen und Plattformen zu erreichen. Dieser Umstand hat beispielsweise zur Folge, dass *SoapFormatter* XML-Darstellungen erzeugt, die möglicherweise von keiner anderen SOAP-Implementierung interpretiert werden können. Auf der Gegenseite kann mit dem *XmlSerializer* nicht jeder CLR-Typ direkt umgewandelt werden.

Der binäre Ansatz zur Serialisierung geschieht im Übrigen über den *BinaryFormatter*, der wie der *SoapFormatter* im Namensraum *System.Runtime.Serialization* zu finden ist. Wir wollen uns nun die Architektur und Arbeitsweise des *XmlSerializer* zu Gemüte führen, nachdem in Kapitel 5 bereits ein grober Überblick über die Serialisierung in ASP.NET XML Web Services gegeben wurde.

6.3.2 Architektur von XmlSerializer

Im Gegensatz zur binären Serialisierung werden bei der XML-Serialisierung mit dem *XmlSerializer* nur öffentliche Eigenschaften und Felder verarbeitet. Die Typintegrität und weitere Informationen wie Assembly-Daten werden nicht beibehalten. Diese Art der Serialisierung bietet sich vor allem dann an, wenn Sie Daten bereitstellen oder nutzen möchten, ohne dazu die Anwendung, die mit diesen Daten arbeitet, einzuschränken. Dieses Konzept wird von der Klasse *XmlSerializer* implementiert. Sie ist die zentrale Klasse und über sie wird sämtliche Serialisierung abgehandelt. Durch verfügbare Attribute im gleichen Namensraum lässt sich der *XmlSerializer* zudem noch zur Laufzeit für die eigenen Wünsche und Anforderungen konfigurieren (siehe Abschnitt 6.3.3).

Abbildung 6.2: Architekturüberblick von XmlSerializer



Die beiden wichtigsten Methoden in der Klasse *XmlSerializer* sind *Serialize* und *Deserialize*. Diese beiden Methoden sorgen dafür, dass ein CLR-Typ in eine XML-Repräsentation konform zu XSD überführt wird, respektive wieder zurückgewandelt wird. Dabei serialisiert *XmlSerializer* nur Klassen mit einem Standardkonstruktor (also ein Konstruktor ohne Parameter) und kann auch keine Methoden serialisieren. Durch die alleinige Serialisierung von öffentlichen Feldern wird durch diese Aktion gewissermaßen eine Kopie des aktuellen Klassenobjekts in einer XML-Darstellung erzeugt. Die Serialisierung findet über ein *Stream*-Objekt statt, das entweder im Speicher vorhanden ist oder auf eine Datei im Dateisystem zeigt. Pro verwendetem CLR-Typ wird eine *XmlSerializer*-Instanz benötigt – daher wird normalerweise dem Konstruktor von *XmlSerializer* auch ein Parameter des Typs *Type* übergeben. Diese Instanz erzeugt dann beim Anlegen dynamisch eine temporäre Assembly (nur wenn noch keine Assembly für diesen Typ im Cache

vorliegt), in der zwei Klassen vom Typ *XmlSerializationReader* und *XmlSerializationWriter* referenziert werden (vgl. Abbildung 6.2). Diese enthalten schnellen Code zur Umwandlung der Typen in XML und umgekehrt, und werden in einem Laufzeitcache abgelegt, auf den auch andere *XmlSerializer*-Objekte zugreifen können.

Zur Veranschaulichung der Arbeitsweise von *XmlSerializer* werden wir uns ein kleines Beispiel ansehen, in dem ein einfacher komplexer Typ serialisiert wird. Das zugehörige Code-Beispiel ist in Listing 6.3 zu sehen.

```
using System;
using System.Xml;
using System.Xml.Serialization;
using System.IO;

class SerTest
{
   static void Main(string[] args)
   {
      MeinTyp myObject = new MeinTyp();
      myObject.Vorname = "Christian";
      myObject.Name = "Weyer";
      myObject.Alter = 27;
      myObject.Geschlecht = "m";

      XmlSerializer mySerializer = new
         XmlSerializer(typeof(MeinTyp));

      StreamWriter myWriter = new
         StreamWriter("SerTyp.xml");
      mySerializer.Serialize(myWriter, myObject);
   }
}

public class MeinTyp
{
   public string Name;
   public string Vorname;
   public int Alter;
   private string geschlecht;

   public string Geschlecht
   {
      get
      {
         return geschlecht;
      }
```

```
      set
      {
         geschlecht = value;
      }
   }
}
```

Listing 6.3: Manuelle Serialisierung mit dem XmlSerializer

Im Beispiel wird ein eigener Typ `MeinTyp` mit drei öffentlichen Feldern, einem privaten Feld und dazu gehöriger Eigenschaft definiert. In der Hauptroutine der einfachen Kommandozeilenanwendung wird ein Objekt von `MeinTyp` erzeugt und die zugehörigen Werte gesetzt. Anschließend wird mit dem *XmlSerializer* über ein *Stream*-Objekt die erzeugte XML-Darstellung in eine Datei geschrieben. Dieses XML-Dokument ist in Listing 6.4 dargestellt. Man kann gut erkennen, wie das XML konform mit XSD ist und somit beispielsweise leicht in eine SOAP-basierte Kommunikation integriert werden könnte.

Dies ist also die prinzipielle Vorgehensweise bei der XML-Serialisierung. Die XML Web Service-Implementierung in ASP.NET bedient sich nun des *XmlSerializers*. Dabei spielt sich für den Web Services-Entwickler alles hinter der ASP.NET-Fassade ab. Aber er hat die Möglichkeit, durch die Anwendung von bestimmten Attributen in den Serialisierungsprozess bei XML Web Services einzugreifen.

```
<?xml version="1.0" encoding="utf-8"?>
<MeinTyp xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Name>Weyer</Name>
   <Vorname>Christian</Vorname>
   <Alter>27</Alter>
   <Geschlecht>m</Geschlecht>
</MeinTyp>
```

Listing 6.4: Durch XmlSerializer generierte XML-Darstellung von MeinTyp

6.3.3 Beeinflussung des Serialisierungsvorgangs

Wie kann ich nun Einfluss auf die erzeugten XML-SOAP-Nachrichten nehmen? In vielen Fällen möchte ich Elemente, Attribute oder Parameter anders benennen als der *XmlSerializer* das macht – oder aber den XML-Namesraum für bestimmte Elemente ändern. Das .NET Framework hält hier eine komfortable Konfigurationsmöglichkeit des *XmlSerializers* durch die Verwendung von Attributen bereit. Da eine SOAP-Nachricht entweder in literaler oder encodierter Form erstellt werden kann (vgl. Kapitel 5), gibt es unterschiedliche Attribute für diese Darstellungsformen.

Alle Attribute, die mit *Soap* beginnen, sind für die Manipulation eines encodierten XML-Datenstroms verantwortlich. Alle Attribute mit *Xml* hingegen können einen literalen XML-Datenstrom verändern. Für eine komplette Auflistung der jeweils verfügbaren Attribute aus dem Namensraum *System.Xml.Serialization* verweise ich Sie auf Tabelle 6.1 und Tabelle 6.2. Ich werde Ihnen einige wichtige Attribute im Einsatz vorführen und zeigen, wie sie sich auf die SOAP-Nachricht auswirken. Eine vollständige Behandlung aller Attribute kann an dieser Stelle leider nicht gewährleistet werden.

Tab. 6.1: Attribute zur Steuerung von literalen SOAP-Nachrichten

Attribut	Ziel	Beschreibung
XmlAnyAttribute	Öffentliches Feld, Eigenschaft, Parameter oder Rückgabewert, wodurch ein Array von *XmlAttribute*-Objekten zurückgegeben wird.	Beim Deserialisieren wird das Array mit *XmlAttribute*-Objekten aufgefüllt, die für alle im Schema unbekannten XML-Attribute stehen.
XmlAnyElement	Öffentliches Feld, Eigenschaft, Parameter oder Rückgabewert, wodurch ein Array von *XmlElement*-Objekten zurückgegeben wird.	Beim Deserialisieren wird das Array mit *XmlElement*-Objekten aufgefüllt, die für alle im Schema unbekannten XML-Elemente stehen.
XmlArray	Öffentliches Feld, Eigenschaft, Parameter oder Rückgabewert, wodurch ein Array von komplexen Objekten zurückgegeben wird.	Die Felder des Arrays werden als Felder eines XML-Arrays generiert.
XmlArrayItem	Öffentliches Feld, Eigenschaft, Parameter oder Rückgabewert, wodurch ein Array von komplexen Objekten zurückgegeben wird.	Die abgeleiteten Typen, die in ein Array eingefügt werden können. Wird i.d.R. im Zusammenhang mit einem *XmlArrayAttribute*-Objekt angewendet.
XmlAttribute	Öffentliches Feld, Eigenschaft, Parameter oder Rückgabewert.	Legt fest, dass das Feld als XML-Attribut serialisiert wird.
XmlChoiceIdentifier	Öffentliches Feld, Eigenschaft, Parameter oder Rückgabewert.	Das Feld kann durch Verwenden einer Aufzählung eindeutig bestimmt werden.
XmlElement	Öffentliches Feld, Eigenschaft, Parameter oder Rückgabewert.	Das Feld oder die Eigenschaft wird als XML-Element serialisiert.

Tab. 6.1: Attribute zur Steuerung von literalen SOAP-Nachrichten (Forts.)

Attribut	Ziel	Beschreibung
XmlEnum	Öffentliches Feld, das ein Aufzählungsbezeichner ist.	Der Name eines Aufzählungsfeldes.
XmlIgnore	Öffentliche Eigenschaften und Felder.	Die Eigenschaft oder das Feld wird beim Serialisieren ignoriert.
XmlInclude	Öffentliche abgeleitete Klassendeklarationen und Rückgabewerte öffentlicher Methoden (wird in WSDL-Dokumente eingebettet).	Die Klasse wird beim Generieren von Schemas eingeschlossen (und daher bei der Serialisierung erkannt).
XmlRoot	Deklarationen öffentlicher Klassen.	Die Klasse stellt das Stammelement des XML-Dokuments dar. Mit diesem Attribut können Sie den XML-Namensraum und Elementnamen genauer angeben.
XmlText	Öffentliche Eigenschaften und Felder.	Die Eigenschaft oder das Feld soll als XML-Text serialisiert werden.
XmlType	Deklarationen öffentlicher Klassen.	Der Name und XML-Namensraum des XML-Typs.

Werfen wir zunächst einen Blick auf den Beispiel-Web Service aus Listing 6.5. Außer der Web Service-Klasse ist eine weitere Klasse für den Typ `Person` definiert. In der Web Service-Klasse existieren drei Methoden, welche die Verwendung verschiedener Attribute demonstrieren sollen.

Die Methode `literalerTyp` ist eine Web Service-Methode, die in literaler Form kodiert wird (der Einsatz des Attributs *SoapDocumentMethod* wäre hier nicht unbedingt notwendig, da dies die Voreinstellung ist). Die Methode gibt eine Instanz vom Typ `Person` zurück. In der Klasse werden für das öffentliche Feld `Name` zwei Attribute angegeben: *SoapElement* und *XmlElement*. Diese beiden Attribute sorgen durch das Setzen ihrer Eigenschaft *ElementName* dafür, dass das Feld Name als Feld `Nachname1` bzw. `Nachname2` in das WSDL eingebettet und in der SOAP-Nachricht erwartet wird – je nachdem, ob der Typ in einer literalen oder einer encodierten Web Service-Methode verwendet wird. Hier wird also `Nachname2` verwendet.

Analog verhält sich die zweite Methode `encodierterTyp`. Hier wird das Element als `Nachname1` serialisiert.

Die dritte Methode schließlich zeigt, wie die Serialisierung eines Rückgabewerts beeinflusst werden kann. Durch Verwendung von `[return: …]` im Falle von C# kann ein Attribut auf den Rückgabeparameter dieser Methode angewendet werden. In diesem Beispiel wird neben dem Namen auch der XML-Namensraum neu definiert.

Wenn Sie sich den ASP.NET XML Web Service mit der Hilfeseite anschauen, können Sie in den angegebenen SOAP-Beispielen die Unterschiede genau erkunden. Den relevanten Ausschnitt des automatisch erzeugten WSDL für die Inkludierung der unterschiedlichen Typen nach Anwendung der Attribute kann man in Abbildung 6.3 sehen.

Abbildung 6.3: Im WSDL enthaltener komplexer Typ Person (für literales und encodiertes SOAP)

```
<%@ WebService Language="C#" Class="SoapSer" %>

using System;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Serialization;

public class SoapSer
{
   [WebMethod]
   [SoapDocumentMethod]
   public Person literalerTyp()
   {
      Person meinTyp = new Person();
      meinTyp.Name = "Weyer";
      meinTyp.Vorname = "Christian";

      return meinTyp;
   }

   [WebMethod]
   [SoapRpcMethod]
   public Person encodierterTyp()
   {
      Person meinTyp = new Person();
      meinTyp.Name = "Weyer";
      meinTyp.Vorname = "Christian";

      return meinTyp;
   }

   [WebMethod]
   [return: XmlElement(Namespace =
    "http://eyesoft.de/webservices/buch/",
    ElementName = "Autor")]
   public Person rueckgabeTyp()
   {
      Person meinTyp = new Person();
      meinTyp.Name = "Weyer";
      meinTyp.Vorname = "Christian";

      return meinTyp;
   }
}
```

```
public class Person
{
   [SoapElement(ElementName = "Nachname1")]
   [XmlElement(ElementName = "Nachname2")]
   public string Name;
   public string Vorname;
   public int Alter;
   private string geschlecht;

   public string Geschlecht
   {
      get
      {
         return geschlecht;
      }
      set
      {
         geschlecht = value;
      }
   }
}
```

Listing 6.5: ASP.NET Web Service mit Attributen zum Beeinflussen von WSDL und SOAP

Attribut	Ziel	Beschreibung
SoapAttribute	Öffentliches Feld, Eigenschaft, Parameter oder Rückgabewert.	Das Feld einer Klasse wird als XML-Attribut serialisiert.
SoapElement	Öffentliches Feld, Eigenschaft, Parameter oder Rückgabewert.	Das Feld einer Klasse wird als XML-Element serialisiert.
SoapEnum	Öffentliches Feld, das ein Aufzählungsbezeichner ist.	Der Feldname eines Aufzählungsfeldes.
SoapIgnore	Öffentliche Eigenschaften und Felder.	Die Eigenschaft oder das Feld wird beim Serialisieren ignoriert.
SoapInclude	Öffentliche abgeleitete Klassendeklarationen und öffentliche Methoden (für WSDL-Dokumente).	Der Typ wird beim Generieren von Schemas eingeschlossen (und daher bei der Serialisierung erkannt).
SoapType	Deklarationen öffentlicher Klassen.	

Tab. 6.2: Attribute zur Steuerung von encodierten SOAP-Nachrichten

Wir haben in diesem Abschnitt gesehen, wie man in den Prozess der automatischen Serialisierung durch *XmlSerializer* eingreifen kann und sowohl die WSDL-Beschreibung als auch die tatsächlich versendeten SOAP-Pakete manipulieren kann.

Man kann die Beeinflussung der XML-Erzeugung sogar so weit treiben, dass man das komplette Verfahren zur XML-Serialisierung von Objekten als SOAP-Meldungen überschreiben kann. Der Schlüssel hierfür ist die Klasse *SoapAttributeOverrides*. Wer noch tiefer in die Infrastruktur eingreifen möchte, kann durch Implementierung der Schnittstelle *IXmlSerializable* sogar einen vollständig neuen Serialisierungsmechanismus programmieren.

6.3.4 Binäre Daten

Ein heißes Thema ist die Übertragung von großen Daten oder gar binären Daten mit XML-basierten Web Services. Da viele Personen SOAP als Ersatz für beispielsweise DCOM sehen und der Meinung sind, dass man mit SOAP (und somit XML) auch binäre Datenblöcke übertragen können muss, widme ich diesem Thema hier einen Abschnitt. Dieses Thema passt auch gut zur Serialisierung von Daten, da die binären Daten ja irgendwie in eine XML-konforme Darstellung überführt werden müssen – wenn man den herkömmlichen SOAP-Lösungsweg gehen möchte.

Der einfachste aber auch am wenigsten sinnvolle Ansatz ist die Einbettung der Binärdaten in den SOAP Envelope. Sei es im SOAP Body oder in einem SOAP Header. ASP.NET bzw. *XmlSerializer* stellt hierfür sogar eine automatisierte Lösung zur Verfügung. Wenn in einer Web Service-Methodensignatur ein Array von Bytes (*byte[]*) vorkommt, dann wird dieses *byte[]* von *XmlSerializer* in eine Base64-Darstellung umgewandelt. Diese Base64-Darstellung ist eine Zeichenfolge, die dann als Parameter in den SOAP Envelope kodiert wird. Der umgekehrte Vorgang geschieht bei der Deserialisierung. Die Base64-Zeichenkette wird in ein *byte[]* umgewandelt und kann dann beispielsweise in einer Datei abgespeichert werden.

Ein Beispiel soll diesen Vorgang illustrieren. Der Web Service ist diesmal in Visual Basic .NET geschrieben und in Listing 6.6 zu sehen. In der Methode `holeBild` wird ein von der Festplatte eingelesenes JPG-Bild in einen *MemoryStream* geladen. Über die Methode *GetBuffer* des Datenstroms wird ein *byte[]* zurück gegeben, in dem sich die Bilddaten befinden. Der *XmlSerializer* erkennt nun den Typ des Rückgabeparameters und serialisiert das *byte[]* in eine Base64-Zeichenkette.

```
<%@ WebService Language="VB" Class=" BildDienst" %>
Imports System.Web.Services
Imports System.Drawing
Imports System.Drawing.Imaging
Imports System.IO
```

```
<WebService(Namespace:=
   "http://eyesoft.de/webservices/")> _
Public Class BildDienst
   Inherits WebService

   <WebMethod()> _
   Public Function holeBild() As Byte()
      Dim MyImg As Image
      MyImg = New Bitmap(Server.MapPath(".\") &
         "Test1.jpg")
      Dim MemStr As New MemoryStream()
      MyImg.Save(MemStr, ImageFormat.Jpeg)

      Return MemStr.GetBuffer()
   End Function
End Class
```

Listing 6.6: Übertragung von Bilddaten mit einem ASP.NET XML Web Service

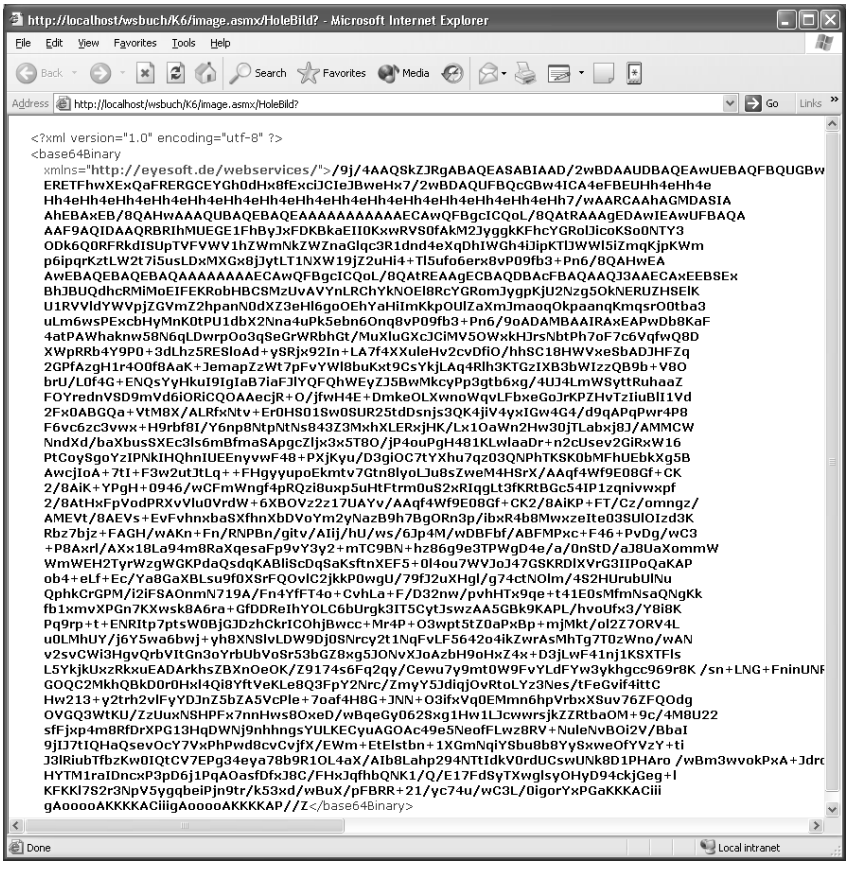


Abbildung 6.4: Base64-Darstellung eines 3KB großen JPG-Bildes

Das Ergebnis des Aufrufs der Web Service-Methode für ein ca. 3 KB großes Bild kann man in Abbildung 6.4 sehen. Wie man sieht, ist der Base64-Algorithmus nicht sehr effizient. Das verwendete Testbild ist nun wirklich nicht groß. Man kann sich also vorstellen, welche Datenmenge hier erzeugt wird, wenn etwa PDF-Dokumente oder BMP-Dateien als *byte[]* versendet werden sollen. Dieser Ansatz ist also in einer Real-World-Anwendung nicht zu gebrauchen.

Alternative Lösungen gibt es im .NET Framework zum Zeitpunkt des Verfassens dieses Buchs leider noch nicht. In der Welt von SOAP und XML Web Services gibt es zwei Standards bzw. Vorschläge, die aber für die Version 1.0 von .NET nicht implementiert wurden.

Der eine Ansatz heißt *SOAP Messages with Attachments* (abgekürzt *SwA*) und der andere heißt *Direct Internet Message Encapsulation* (*DIME*). SwA geht den Weg über das Versenden von SOAP-Nachrichten innerhalb einer umschließenden *MIME*-Nachricht, genauer gesagt einer *MIME-Multi-Part-Message*. Eine solche Nachricht kann mehrere Abschnitte besitzen, die durch einen Kennzeichner getrennt werden. Bei SwA wird nun der SOAP Envelope in einen MIME-Abschnitt und die zu übertragenden Dokumente als Binärdaten als Anhang (*attachment*) in andere Abschnitte gepackt. Somit ist die eigentliche Nachricht keine zulässige SOAP-Nachricht mehr. Es wird also eine Erweiterung des SOAP-Stacks benötigt, und dieser existiert wie gesagt für .NET noch nicht. Ein Beispiel für eine Nachricht gemäß der W3C-Spezifikation für SwA mitsamt Bindung an HTTP als Transportprotokoll sehen Sie in Listing 6.7.

```
POST /webservices/MIMEdemo HTTP/1.1
Host: www.eyesoft.de
Content-Type: Multipart/Related; boundary=MIME_boundary;
   type=text/xml;start="<part001.xml@eyesoft.de>"
Content-Length: XXX
SOAPAction: http://www.eyesoft.de/webservices/MIMEdemo
Content-Description: Dies ist eine optionale
   Beschreibung

--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: part001.xml@eyesoft.de

<?xml version='1.0' ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV=
   "http://schemas.xmlsoap.org/soap/envelope/">
   <SOAP-ENV:Body>
      <test:uebertrageDateien xmlns:test=
         "http://schemas.eyesoft.de/webservices/
          MIMEdemo">
         <dokument1 href=
```

```
          "cid:cpart001.tiff@eyesoft.de"/>
         <dokument2 href=
          "cid:cpart001.jpeg@eyesoft.de"/>
      </test:uebertrageDateien>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: image/tiff
Content-Transfer-Encoding: base64
Content-ID: cpart001.tiff@eyesoft.de

... Base64-codiertes Bild im TIFF-Format ...
--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: part001.jpeg@eyesoft.de

... Rohdaten eines JPG-Bildes ...
--MIME_boundary--
```

Listing 6.7: Beispiel einer Nachricht im SOAP with Attachments-Format (MIME)

DIME geht einen anderen Ansatz als SwA. DIME wurde von Microsoft entwickelt und als Standardisierungsvorschlag bei öffentlichen Gremien eingereicht. DIME ist zunächst unabhängig von SOAP und definiert ein neues Format zur Übertragung von mehreren binären Datensätzen innerhalb eines einzelnen Nachrichtenpakets. Aber es gibt auch einen Vorschlag, welcher den Einsatz von SOAP mit DIME beschreibt. Auf Basis dieses Vorschlags werden momentan bei Microsoft Implementierungen für .NET und das COM-basierte *SOAP Toolkit* erstellt. Eine DIME-Nachricht kann einen oder mehrere Datensätze enthalten, die alle einen Header-Abschnitt und einen Teil für die Nutzdaten haben. Der interne Aufbau eines Datensatzes in DIME sieht so aus, dass im Header diverse Informationen über Flags und Länge der Daten festgehalten sind. Diese Daten werden als Bitfeld kodiert. Im Body stehen dann die eigentlichen Daten des zu übertragenden Dokuments. Zusammen mit der Einbettung von SOAP in DIME ergibt sich die Beispielnachricht aus Listing 6.8.

```
_____________________________________________
1 0 0 0000000000000
010 0000000101001
00000000000000000000000000010110110
http://schemas.xmlsoap.org/soap/envelope/
<envelope>
   <body>
      <ConvertImage>
         <ConversionType>JPEGtoGIF</ConversionType>
```

```
        <image href="Image1" />
      </ConvertImage>
   </body>
</envelope>

--------------------------------------------------
0 0 1 0000000000110
001 0000000001010
00000000000000001111111111111111
Image1
image/jpeg
   <64K Binärdaten>
--------------------------------------------------
0 1 0 0000000000000
000 0000000000000
00000000000000000000011000111110000
   <12784 Bytes Binärdaten>
--------------------------------------------------
```

Listing 6.8: Beispielnachricht im DIME-Format

DIME wird sich vermutlich gegenüber MIME bzw. SwA durchsetzen. Allerdings wird es noch einiges an Zeit brauchen, bis eine breite Unterstützung für die Übertragung von Binärdaten mittels Web Services vorhanden sein wird.

6.4 COM/COM+ und XML Web Services

Viele Entwickler und verantwortliche Personen werden sich fragen, ob die bisher programmierten und teilweise auf COM-Basis entstandenen Applikationen und Komponenten überhaupt noch lauffähig sind beim Einsatz von .NET. Schließlich hat eine Großzahl an Unternehmen ihre Programmierer in COM, VB6, ATL o.ä. ausgebildet, um Komponenten-basierte Anwendungen mit dem Component Object Model zu erstellen. Nicht zuletzt das Erscheinen des MTS und von COM+ hat viele Beteiligte davon überzeugt, durch die Akzeptanz eines neuen Programmiermodells und einer neuen Herangehensweise, auch gut skalierbare Intranet- und Internet-Architekturen auf Basis von Windows gestalten zu können. All diese Mühe und die hohen Investitionen kann man nicht einfach unter den Tisch fallen lassen.

Microsoft hat an diesen Umstand gedacht und bietet eine Interoperabilitätsschicht an, über die die COM-Laufzeitumgebung und die NET CLR miteinander kooperieren können. In diesem Abschnitt stelle ich Ihnen vor, wie man COM-Komponenten von .NET aus anspricht und in .NET erstellte Assemblies von COM-basierten Anwendungen aus verwenden kann. Zudem gehe ich auf diverse Dienste der COM+-Infrastruktur ein und wie man Sie in .NET und ASP.NET Web Services benutzen kann. Vor allem das Thema Transaktionen wird hierbei beleuchtet.

6.4.1 .NET und COM

Schauen wir uns zunächst die Architektur an, mit der es möglich wird, zwischen .NET- und COM-Anwendungen zu kommunizieren. Abbildung 6.5 zeigt die beiden grundlegenden Konzepte des *RCW* (*Runtime Callable Wrapper*) und *CCW* (*COM Callable Wrapper*). Die Funktion des RCW ist es, COM-Objekte von der .NET CLR aus ansprechen zu können. Der RCW fungiert dabei als eine Art Proxy, der die Aufrufe übersetzt. Ähnlich arbeitet auch der CCW. Er sorgt dafür, dass .NET-Objekte von COM-Anwendungen aus aufrufbar und nutzbar sind. Beide Objekte werden dynamisch zur Laufzeit erzeugt und es existiert immer genau ein solches Proxy-Objekt pro COM- oder .NET-Objekt, egal wie viele Referenzen auf die Objekte vorhanden sind. Der CCW wird durch COM und der RCW durch die CLR instanziiert.

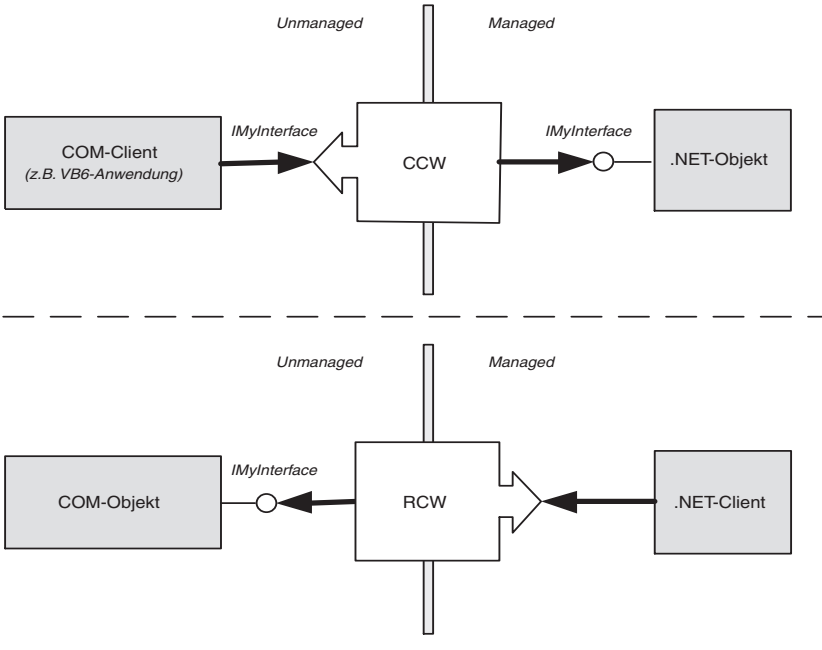


Abbildung 6.5: Interoperabilität zwischen COM und .NET über RCW und CCW

Doch wie kommen die jeweils fremden Laufzeitumgebungen an die Daten und Metadaten der anderen Welt, die sie benötigen, um Objekte zu lokalisieren und zu erzeugen? Hierfür ist ein Schritt notwendig, den der Entwickler übernehmen muss: Die Erstellung von Metadaten anhand von Metadaten. Was bedeutet dies? Die Antwort wird in den beiden folgenden Teilabschnitten gegeben.

Nutzung von COM-Objekten in .NET

Damit die CLR genügend Metadaten hat, um mit Typen innerhalb einer Assembly umgehen zu können, muss für vorhandene COM-Komponenten eine Schale (*wrapper*) erzeugt werden. Dieser Wrapper ist dann eine her-

kömmliche Assembly (in Form einer DLL), die von der CLR aus angesprochen werden kann. Die Informationen innerhalb dieser Wrapper-Assembly werden auf Basis der Typbibliothek (*Type Library*) einer COM-Komponente erstellt. Eine Typbibliothek liegt entweder in separierter Form in einer *.tlb*- oder einer *.olb*-Datei vor. Doch häufig ist die Typbibliothek in die Komponente (in Form einer DLL, einer *.ocx*-Datei oder in Gestalt von EXEs) integriert.

Die Generierung dieses Wrappers kann zum einen mit der Kommandozeilenanwendung *tlbimp.exe* oder mit dem Visual Studio .NET erfolgen. Als Beispiel betrachten wir die Typbibliothek von *Microsoft Word*. Word ist ja aus vielen COM-Schnittstellen und -Objekten zusammengesetzt und wir werden einen kleinen Web Service schreiben, der sich die Funktionalität von Word zunutze macht. Mit der folgenden Befehlszeile wird aus der Typbibliothek *MSWORD9.OLB* (für Word 2000) eine .NET Assembly erzeugt:

```
tlbimp.exe MSWORD9.OLB
```

Das Kommando erzeugt eine Assembly namens *Word.dll* (in der Tat werden drei Assemblies erzeugt, aber nur *Word.dll* ist für uns von Bedeutung). Diese Assembly kann man sich im ILDASM anschauen, um sicher zu gehen, dass die Konvertierung erfolgreich war (vgl. Abbildung 6.6).

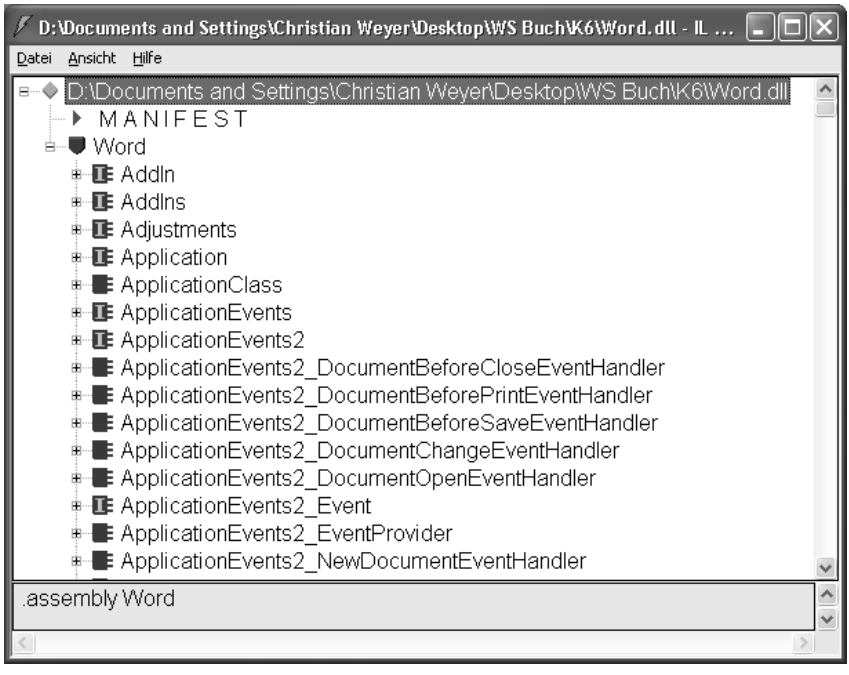


Abbildung 6.6: Erzeugte .NET-Assembly für eine COM-Komponente im ILDASM

Mit diesem Schritt sind die wichtigsten Vorbereitungen getroffen. Man kann nun eine Anwendung schreiben, die diese Assembly verwendet. Wir werden einen Web Service programmieren, der die Rechtschreibüberprüfung von Word verwendet. Das Beispiel liegt in Visual Basic .NET vor (siehe Listing 6.9).

Der Web Service kontaktiert Word über die erzeugte Assembly. Durch Metadaten in dieser Assembly generiert die CLR einen RCW, der wiederum die eigentliche COM-Komponente lädt und die angeforderten Objekte instanziiert. Die Objekt- und Schnittstellendefinition im Quelltext sind mit denen aus der originalen Typbibliothek identisch (außer es mussten Umbenennungen vorgenommen werden). Das Ergebnis des Aufrufs gibt der Web Service dann als komplexen Typ zurück.

```
<%@ WebService Language="VB"
   Class="RechtschreibPruefer" %>
Imports System.Web.Services

Public Class RechtschreibPruefer
   <WebMethod()> _
   Public Function holeRechtschreibVorschlaege
      (ByVal wordToCheck As String) As
       SpellingSuggestions
      Dim spellChecker As Word._Application =
         New Word.Application()
      Dim document As Word.Document =
         CType(spellChecker.Documents.Add,
            Word.Document)
      Dim spellSuggestions As Word.SpellingSuggestions
      Dim spellSuggestion As Word.SpellingSuggestion
      Dim strSpelling As String

      spellSuggestions =
         spellChecker.GetSpellingSuggestions(
         wordToCheck)

      Dim count As Long = spellSuggestions.Count
      Dim ssStruct As SpellingSuggestions
      Dim i As Integer
      Dim intCount As Integer = CInt(count)

      If count > 0 Then
         i = 0
         ssStruct.SpellingWord = wordToCheck
         ReDim ssStruct.SpellingSuggestionList(
            intCount - 1)

         For Each spellSuggestion In spellSuggestions
            ssStruct.SpellingSuggestionList(i) =
```

```
                spellSuggestion.Name
            i = i + 1
         Next
      End If

      spellChecker.Quit()

      Return ssStruct
   End Function
End Class

Public Structure SpellingSuggestions
   Dim SpellingWord As String
   Dim SpellingSuggestionList As String()
End Structure
```

Listing 6.9: ASP.NET Web Service verwendet ein COM-Objekt

Man sollte nicht vergessen, dass es sich hier um einen ASP.NET Web Service und somit um eine gewöhnliche ASP.NET Web-Anwendung handelt. Daher muss sich die Assembly für Word im Unterverzeichnis *bin* befinden, damit die Anwendung funktionieren kann. Außerdem muss man noch die Sicherheitseinstellungen beachten, denn die ASP.NET-Laufzeitumgebung läuft unter einem eigenen, unterprivilegierten Benutzerkonto (siehe hierzu auch Kapitel 5 und Kapitel 7). Der Account ASPNET muss Zugriff auf die Wrapper-Assembly und auf die COM-Komponente haben.

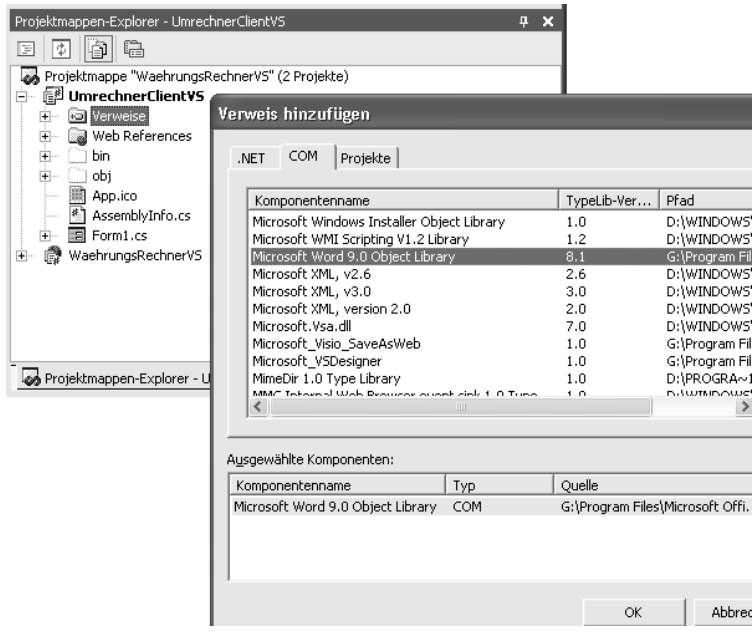


Abbildung 6.7: Erstellen eines Verweises auf eine COM-Komponente im Visual Studio .NET

Der Vollständigkeit halber ist in Abbildung 6.7 aufgezeigt, wie man in Visual Studio .NET eine Wrapper-Assembly erzeugen kann. Man fügt einfach durch Zeigen auf den Projektnamen per Kontextmenü und dem Befehl VERWEIS HINZUFÜGEN eine Referenz auf ein COM-Objekt ein. Visual Studio .NET erledigt dann die notwendigen Schritte und präsentiert innerhalb des Projektmappen-Explorers eine Referenz auf die erzeugte .NET Assembly. Dies funktioniert in jedem Projekt.

Nutzung von .NET-Objekten in COM

Wir wollen nun den Weg beleuchten, aus einer COM-Anwendung heraus auf eine .NET Assembly zuzugreifen. Ähnlich wie im vorhergehenden Abschnitt gilt auch hier, dass die COM-Laufzeitumgebung Metadaten benötigt, um die Komponente zu laden und die Objekte zu erzeugen. Diese Metadaten müssen in Form einer Typbibliothek vorliegen, wenn es um die Integration von .NET-Anwendungen geht.

Der Weg von der .NET Assembly hin zu einer COM-Typbibliothek geht über das Werkzeug *tlbexp.exe*. Für die Registrierung der notwendigen Informationen in der Registry ist die Anwendung *regasm.exe* zuständig. Wir verwenden als Beispiel eine in C# geschriebene Demoklasse, die uns einen Willkommensgruß aus der .NET-Welt sendet:

```
public interface IComTest
{
   string SagHallo();
}
public class COMTest : IComTest
{
   public string SagHallo()
   {
      return "Hallo aus der .NET-Welt :-)";
   }
}
```

Wichtig ist in diesem Beispiel, dass wir explizit eine Schnittstelle (*interface*) für unsere nach COM zu exportierende Klasse definieren. Die Klasse wiederum implementiert dann diese Schnittstelle. An dieser Stelle wird also der schnittstellenorientierten Vorgehensweise in COM Rechnung getragen.

Diese Klasse kompilieren wir zunächst in eine Assembly DLL:

```
csc.exe /t:library comtest.cs
```

Mit dem folgenden Aufruf wird dann aus der Assembly eine Typbibliothek generiert:

```
tlbexp.exe comtest.dll
```

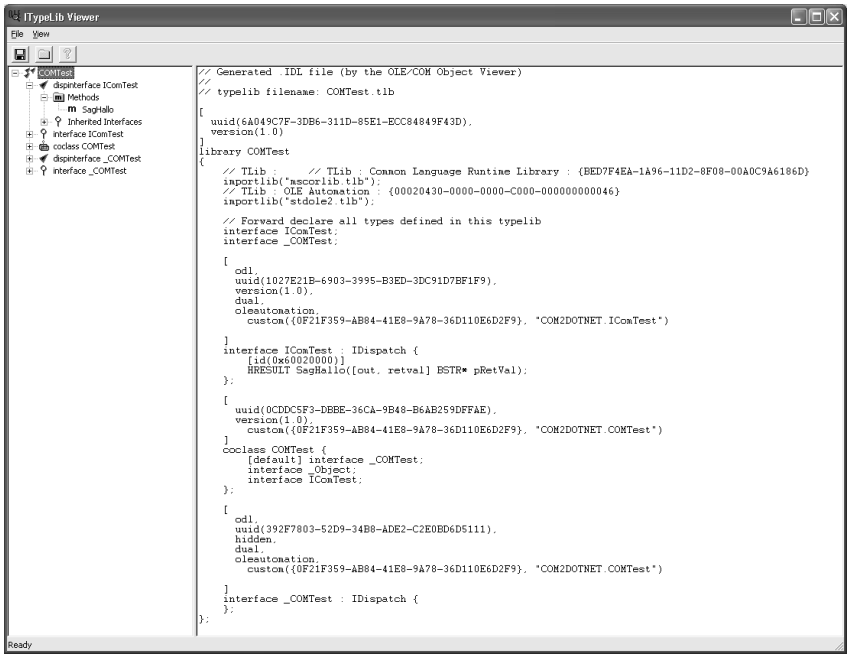
Das Ergebnis des Kommandos lautet *comtest.tlb* und stellt die Typbibliothek für den Einsatz mit COM dar. Wir müssen nun noch die relevanten Metadaten für COM im System mit folgendem Befehl in der Registry verankern (Registrierung deshalb, weil wir ja über die COM-Infrastruktur darauf zugreifen wollen):

```
regasm.exe comtest.dll
```

Optional kann man auch nur regasm.exe verwenden. Mit der Option */tlb:Dateiname* lässt sich auch damit eine Typbibliothek erzeugen und gleichzeitig registrieren.

Mit dem in Visual Studio 6.0 verfügbaren Werkzeug *OleView* schauen wir uns nun die aus einer .NET Assembly erzeugte COM-Typbibliothek an (Abbildung 6.8).

Abbildung 6.8: Aus .NET Assembly erzeugte COM-Typbibliothek in OleView



Damit wir das Beispiel auch in Aktion sehen, schreiben wir eine einfache Client-Anwendung in VBScript, welche dann mit dem *WSH* (*Windows Scripting Host*) ausgeführt wird. Allerdings muss unsere .NET Assembly dafür im GAC stehen. Also müssen wir für diese Assembly einen starken Namen erzeugen und sie im GAC registrieren (siehe Kapitel 2). Der Quelltext für das WSH-Skript steht in Listing 6.10.

```
Dim oDOTNET
Set oDOTNET = WScript.CreateObject("COM2DOTNET.COMTest")
WScript.Echo oDOTNET.SagHallo()
```

Listing 6.10: Einfaches WSH-Skript in VBScript zum Aufruf einer gewrappten .NET Assembly

Wir haben gesehen, dass die Nutzung von COM-Objekten in einer .NET-Anwendung ebenso einfach ist wie die Verwendung von .NET-Objekten in einer COM-Client-Anwendung. Den Löwenanteil der Arbeit übernehmen die jeweiligen Laufzeitumgebungen in Verbindung mit Hilfswerkzeugen aus dem .NET Framework SDK.

6.4.2 .NET und COM+-Dienste

Ein wichtiger Bestandteil der Windows DNA-Strategie von Microsoft ist die Erstellung skalierbarer, Komponenten-basierter Applikationen unter Windows. Für dieses Ziel hat man vor einigen Jahren das Konzept des MTS weiter entwickelt und mit dem Erscheinen von Windows 2000 das so genannte COM+ auf den Markt gebracht. Eine kurze Übersicht in COM+ befindet sich in Kapitel 1. Nun sollen aber mit .NET nicht auf einmal alle entwickelten Komponenten überflüssig geworden sein. Vor allem von den Diensten, die COM+ dem Entwickler zur Seite stellt, kann man auch unter .NET regen Gebrauch machen: Objekt-Pooling, automatische Transaktionen oder Rollen-basierte Sicherheit. Wohl erst mit der nächsten Version von .NET werden die kontextbasierten Dienste nativ als Implementierung auf Basis der CLR zur Verfügung stehen. In der Zwischenzeit kann man die Funktionalität von COM+ 1.0 und COM+ 1.5 in Windows XP und Windows .NET Server auch innerhalb des .NET Frameworks nutzen.

Namensraum System.EnterpriseServices

Die wichtigsten verfügbaren Dienste in COM+ können Sie nochmals in Tabelle 6.3 nachlesen. Bitte beachten Sie dabei, ob der jeweilige Dienst bereits unter Windows 2000 oder erst ab Windows XP bzw. .NET Server zur Verfügung steht. In .NET ist der Namensraum *System.EnterpriseServices* für die Zusammenarbeit mit COM+ zuständig. Die wichtigste Klasse heißt *ServicedComponent* und stellt die Basisklasse aller Klassen dar, die COM+-Dienste verwenden wollen.

Tab. 6.3: Die wichtigsten verfügbaren COM+-Dienste

Dienst	Beschreibung	Seit OS-Version
Automatische Transaktionsverarbeitung	Deklarative, verteilte Transaktionsverarbeitung unter Aufsicht des *DTC* (*Distributed Transaction Coordinator*).	Windows 2000
BYOT (*Bring Your Own Transaction*)	Festlegen der bevorzugten Transaktionsmethode. BYOT ermöglicht eine Form der Transaktionsvererbung.	Windows 2000
Kompensierende Ressourcen-Manager (Compensating Resource Managers, CRM)	Wenden die Unteilbarkeit und Dauerhaftigkeit auf Nicht-Transaktionsressourcen an.	Windows 2000
Just-In-Time-Aktivierung	Aktiviert ein Objekt über einen Methodenaufruf und deaktiviert es, wenn die Methode beendet ist.	Windows 2000
Lose verknüpfte Ereignisse	Verwaltet Objekt-basierte Ereignisse.	Windows 2000
Objekterstellung	Übergibt eine Zeichenfolge an die Instanz einer Klasse, wenn diese erstellt wird.	Windows 2000
Objekt-Pooling	Stellt einen Pool von gebrauchsfertigen Objekten bereit.	Windows 2000
Private Komponenten	Verhindert den prozessexternen Aufruf einer Komponente.	Windows XP
Komponenten in einer Warteschlange	Stellt asynchrones Message Queuing bereit.	Windows 2000
Rollen-basierte Sicherheit	Verwendet Rollen-basierte Sicherheitsberechtigungen.	Windows 2000
SOAP-Dienste	Veröffentlicht Komponenten als XML Web Services.	Windows XP
Synchronisierung	Verwaltet Parallelität der Zugriffe.	Windows 2000

Als Beispiel werden wir eine Klasse vom Typ *ServicedComponent* erstellen, die Objekt-Pooling und die Fähigkeit zur Übergabe einer Konfigurationszeichenkette von COM+ verwendet. Vor allem Objekt-Pooling ist für XML Web Services-Szenarien sehr interessant, da hierdurch Objektinstanzen innerhalb eines vorkonfigurierten Pools gehalten werden können. Bei Anforderung

eines Objekts wird dieses nicht neu erzeugt, sondern aus dem Pool genommen, was sich in den meisten Fällen in einer höheren Skalierbarkeit der Komponenten-basierten Applikation auswirkt.

Wie in vielen anderen Bereichen innerhalb von .NET auch, wird hier vornehmlich mit Attributen gearbeitet. Will ein Entwickler eine Klasse oder eine Methode mit gewissen COM+-Diensten ausstatten, wendet er einfach das entsprechende Attribut darauf an und setzt die notwendigen Eigenschaften. Sämtliche Attribute befinden sich in *System.EnterpriseServices*. Schauen wir uns die C#-Klasse an, die auf die COM+-Dienste zugreift (siehe Listing 6.11). Im Beispiel wird auf die Benutzerdatenbank aus Kapitel 4 zugegriffen. Allgemein lässt sich sagen, dass viele der von COM+ angebotenen Dienste vor allem in Verbindung mit Datenbankaktionen sinnvoll einzusetzen sind. Zum einen die automatische Transaktionsverarbeitung bei mehreren, mitunter verteilten Datenquellen und zum anderen das Objekt-Pooling, das vor allem beim Auslesen großer Datenmengen von großem Vorteil sein kann. Das Themengebiet der Transaktionen behandeln wir im übernächsten Abschnitt.

Damit für diese Klasse (bzw. für die noch zu erstellende COM+-Applikation, in der diese Klasse dann installiert wird) die gewünschten Dienste auch verwendet werden können, kommen die beiden Attribute *ObjectPooling* und *ConstructionEnabled* zum Einsatz. Über das Setzen von Eigenschaften werden beispielsweise die Größe des Pools und der initiale Wert der Konstruktorzeichenkette festgelegt. Über die geschützte Methode *Construct* wird aus der COM+-Laufzeitumgebung die Zeichenkette übernommen. Diese Zeichenkette kann nämlich in der installierten COM+-Anwendung überschrieben und somit für jede einzelne Installation der Applikation neu gesetzt werden. Im Falle des Objekt-Poolings wird die Untergrenze auf zwei und die Obergrenze des zu erstellenden Pools auf fünf gesetzt. Dies bedeutet, dass zum Zeitpunkt der ersten Anfrage zur Erstellung eines Objekts dieses Typs die COM+-Laufzeitumgebung zwei Instanzen dieser Klasse erzeugt und in den Pool stellt. Werden mehrere Objekte angefordert, dann wird der Pool bis maximal fünf Instanzen aufgestockt. Alle weiteren Anfragen werden in eine Warteschlange gestellt und müssen warten bis ein Objekt wieder freigegeben wird (maximale Zeitdauer des Wartens bis ein Fehler zurückgegeben wird: *CreationTimeout*).

```
using System;
using System.Data;
using System.EnterpriseServices;
using System.Data.SqlClient;

[ConstructionEnabled(true, Default=
 "uid=BuchUser;pwd=ws_buch;database=WS_Buch;
 server=localhost")]
[ObjectPooling(Enabled=true, MinPoolSize=2,
 MaxPoolSize=10, CreationTimeout=5000)]
public class COMplusDemo : ServicedComponent
{
```

```
    private string connString;

    protected override void Construct(string s)
    {
       connString = s;
    }

    public DataSet holeBenutzer()
    {
       DataSet dsBenutzer;
       string strSQL;
       SqlDataAdapter cmdBenutzer;
       SqlConnection conn;
       conn = GetConnection();

       try
       {
          conn.Open();
          dsBenutzer = new DataSet();

          strSQL = "SELECT * FROM Benutzer";
          cmdBenutzer = new SqlDataAdapter(strSQL,
            (SqlConnection)conn);
          cmdBenutzer.Fill(dsBenutzer, "Benutzer");
       }
       catch (Exception ex)
       {
          throw ex;
       }
       finally
       {
          if (conn.State == ConnectionState.Open)
             conn.Close();
       }

       return dsBenutzer;
    }

    private SqlConnection GetConnection()
    {
       SqlConnection conn = new
          SqlConnection(connString);

       return conn;
    }
}
```

Listing 6.11: Verwendung von COM+-Diensten in einer C#-Klasse

Diese Klasse lässt sich nun in einen XML Web Service integrieren. Die übliche Vorgehensweise sieht so aus, dass die eigentliche Geschäftslogik in einer oder mehreren Klassen unabhängig vom XML Web Service implementiert wird. Anschließend erzeugt man einen XML Web Service, der dann als Schale um die COM+-basierte Geschäftslogik agiert. Der XML Web Service für die obige Klasse steht in Listing 6.12. Er hat eine einzige Methode, die sich der Methode `holeBenutzer` aus der Klasse `COMplusDemo` bedient und ein ADO.NET *DataSet* mit den Daten über in die Datenbank eingetragenen Benutzer zurückgibt.

```
<% @WebService Class="COMplusWS" Language="C#" %>
using System;
using System.Data;
using System.Web.Services;

public class COMplusWS
{
   [WebMethod]
   public DataSet holeBenutzer()
   {
      return new COMplusDemo().holeBenutzer();
   }
}
```

Listing 6.12: XML Web Service Wrapper für COM+-Klasse

Allerdings ist unser Web Service so noch nicht lauffähig: es fehlen noch wichtige Informationen, um die COM+-Klasse im System zu registrieren.

Konfiguration und Registrierung

Damit eine Klasse innerhalb der COM+-Laufzeitumgebung ablaufen und die von COM+ angebotenen Dienste verwenden kann, muss eine COM+-Anwendung erstellt und die Klasse darin registriert werden. Die Registrierung kann ein Entwickler oder Administrator manuell vornehmen oder die CLR diese Aktion ausführen lassen.

Hierfür müssen wir unsere COM+-Klasse zunächst noch mit wichtigen Daten ausstatten. Zu diesen Informationen gehören der Name der zu erstellenden COM+-Applikation, die Aktivierungsart (*Server* oder *Library*) und eine Beschreibung. Zusätzlich muss jede Assembly, die in COM+ registriert werden soll, einen starken Namen haben. Also müssen wir auch diese Information in unserer Klasse festhalten. Hierfür verwenden wir das *assembly*-Attribut (Namensraum *System.Reflection*) und schreiben die relevanten Codezeilen direkt in die Datei mit unserer C#-Klasse (siehe Listing 6.13).

```
[assembly: AssemblyKeyFile(@"complus.snk")]
[assembly: ApplicationName("ComplusDemoWS")]
[assembly: ApplicationActivation
   (ActivationOption.Library)]
```

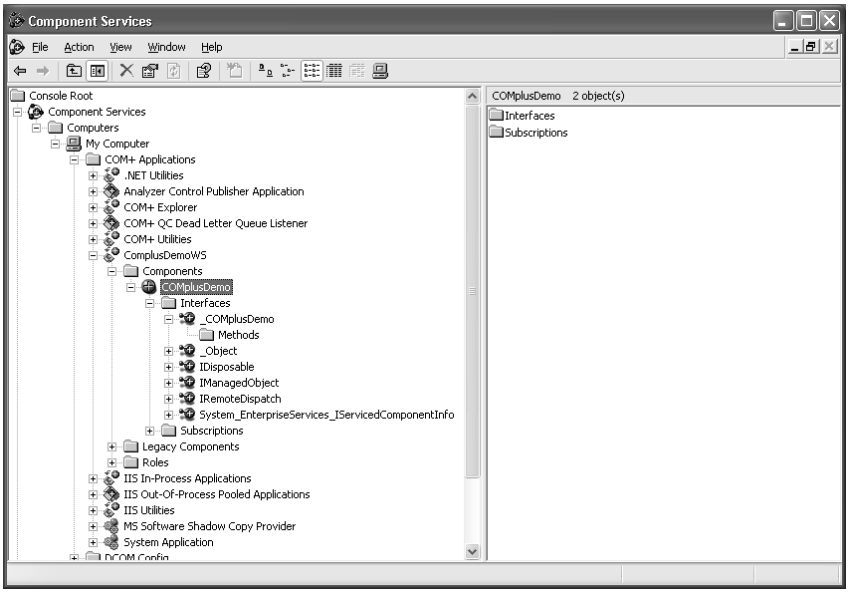
Listing 6.13: Metadaten für die Erstellung einer COM+-Anwendung und der Registrierung der Assembly

Manuell. Die manuelle Registrierung erfolgt mithilfe der Anwendung REGSVCS.EXE wie in folgender Kommandozeile:

```
regsvcs.exe COMPlus.dll
```

Anschließend kann man in der KOMPONENTENDIENSTE-Anwendung das Anlegen der neuen COM+-Applikation mit den im Quelltext angegebenen Informationen überprüfen (siehe Abbildung 6.9).

Abbildung 6.9: Registrierte .NET Assembly im COM+-Explorer



Beim Erstellen einer COM+-Anwendung mit der Aktivierungsoption *Server* müssen die Assembly und alle Assemblies, von denen sie abhängig ist, vor der Verwendung mithilfe des Dienstprogramms *gacutil.exe* in den GAC registriert werden. Andernfalls wird durch die Anwendung eine Ausnahme ausgelöst. Für *Library*-Anwendungen ist dies nicht erforderlich. Hier muss sich die Assembly im gleichen Verzeichnis befinden, wie die Client-Anwendung – und diese ist in unserem Fall von XML Web Services ASP.NET. Also muss die Assembly im *bin*-Verzeichnis liegen.

Automatisch. Die .NET-Laufzeitumgebung bietet einen sehr komfortablen Mechanismus zur dynamischen Registrierung einer COM+-Anwendung. Dynamisch bedeutet in diesem Fall, dass die COM+-Anwendung nicht zur Entwurfszeit der .NET-Applikation registriert werden muss, sondern zur Laufzeit bei der allerersten Anforderung durch einen Client erstellt wird. Die notwendigen Daten werden in den COM+-Katalog eingetragen und über den COM+-Explorer kann diese dynamische Registrierung überprüft werden. Die notwendigen Daten für die Erstellung einer COM+-Anwendung müssen sich wie bei der manuellen Registrierung in der Assembly befinden.

Damit die dynamische Registrierung einer COM+-Applikation zur Laufzeit auch funktioniert, muss das ausführende Benutzerkonto in der Rolle *Administrator* der systemeigenen COM+-Applikation *Systemanwendungen* innerhalb der COM+-Laufzeitumgebung bekannt sein. Hierfür muss der entsprechende Account aber auch in der Gruppe *Administratoren* des Betriebssystems sein – und dies ist beim Konto ASPNET nicht der Fall.

Dies hat zur Folge, dass standardmäßig die dynamische Registrierung von COM+-Anwendungen aus ASP.NET heraus nicht funktioniert! Einzige Abhilfe ist das Hinzufügen von ASPNET in die Gruppe *Administratoren* und in die Rolle *Administrator* von *Systemanwendungen* (oder es wird ein Administrator-Benutzerkonto für die ASP.NET-Laufzeitumgebung konfiguriert). Doch damit werden dem ASPNET-Account weitreichende Berechtigungen gegeben, was nicht unbedingt wünschenswert ist.

6.4.3 Web Services-Transaktionen

Das Thema Transaktionen und vor allem verteilte Transaktionen ist spätestens seit dem großen Erfolg von COM+ und *Enterprise Java Beans* (*EJB*) in aller Munde, wenn es um Unternehmensanwendungen geht. Diese Technologien übernehmen als Container die Verwaltung und Kontrolle von Transaktionen über mehrere Datenquellen und mehrere Computer. Das zugehörige Konzept funktioniert im Intranet sehr gut, im Internet lässt es sich allerdings nicht umsetzen, da die beteiligten Transaktionsmonitore spezielle Protokolle verwenden, die Probleme mit der Firewall haben. Ein zweiter Hinderungsgrund ist das verwendete Zwei-Phasen-Commit-Protokoll, das eine zentrale Instanz erforderlich macht, die schlussendlich über den Ausgang der Transaktion entscheidet. Und diese Annahme kann man im unsicheren und unzuverlässigen Internet nicht machen. Des weiteren sind typische Transaktionen im Internet zwischen Handelspartnern lang andauernde Transaktionen. Somit sind die klassischen Transaktionskonzepte hier nicht einsetzbar, da sie eine Sperrung und Blockierung der beteiligten Ressourcen über die gesamte Dauer einer Transaktion veranlassen.

Im Bereich ASP.NET-basierte XML Web Services gibt es die Möglichkeit, die COM+-Laufzeitumgebung für verteilte Transaktionen zu verwenden. Hier wird aber keine Transaktionskoordination über mehrere Web Services hinweg ermöglicht, sondern der Web Service wird als im Internet veröffentlichter Zugangspunkt für eine innerhalb eines Intranets oder eines abgeschlossenen Bereichs laufende verteilte COM+-Transaktion verstanden. Man könnte hier gewissermaßen von einer lokalen verteilten Transaktion sprechen.

Es gibt zur Entstehungszeit dieses Buchs einige Anstrengungen, auch über XML Web Services Transaktionen zu ermöglichen. Hierzu zählt das *Business Transaction Protocol* (*BTP*) der OASIS-Organisation und einer interne Initiative von Microsoft im Rahmen der *Global XML Web Services Architecture* (*GXA*). Allerdings wird es noch einige Zeit in Anspruch nehmen, bis fertige Standards und vor allem interoperierende Implementierungen in diesem Bereich verfügbar sind.

COM+-Transaktionen

Bei der Steuerung von verteilten Transaktionen in COM+ geht man ähnlich vor wie weiter oben bereits beschrieben wurde. Der Namensraum *System.EnterpriseServices* ist für die gesamte Steuerung aus Sicht einer .NET-Anwendung zuständig. Prinzipiell gibt es zwei Herangehensweisen: Zum einen durch die Verwendung der Eigenschaft *TransactionOption* des Attributs *WebMethod*. Zum anderen durch den oben verwendeten Weg eines Wrappers. Ich werde im Folgenden die erste Alternative wählen, da der zweite Lösungsweg im vorhergegangenen Abschnitt bereits vorgestellt wurde (die Vorgehensweise ist wie oben, nur ist zusätzlich die Verwendung des Attributs *Transaction* erforderlich).

In Listing 6.14 können Sie eine neue Methode unseres Web Services sehen, die einen Datensatz in die Benutzerdatenbank einfügen kann. Durch die Angabe der Eigenschaft *TransactionOption* im Attribut *WebMethod* deuten wir an, wie sich dieser Web Service bezüglich der Beteiligung an einer durch COM+ gesteuerten Transaktion verhalten soll. Die möglichen Werte für diese Eigenschaft sehen Sie in Tabelle 6.4. Die Transaktionen über das Attribut *WebMethod* werden zwar auch über COM+ gesteuert, es muss aber nicht explizit eine COM+-Anwendung dafür angelegt werden (weder manuell, noch dynamisch).

Wert der Eigenschaft	Beschreibung
Disabled	Deaktiviert die Steuerung automatischer Transaktionen für die Web Service-Methode. Ein Objekt mit diesem Attributwert kann den DTC (Distributed Transaction Coordinator) ohne Umweg über COM+ direkt zur Transaktionsunterstützung auffordern.
NotSupported	Gibt an, dass die Web Service-Methode nicht innerhalb des Gültigkeitsbereichs von Transaktionen ausgeführt wird. Beim Verarbeiten einer Anforderung wird der Objektkontext ohne eine Transaktion erstellt, unabhängig davon, ob eine Transaktion aktiv ist.
Supported	Gibt an, dass die Web Service-Methode im Gültigkeitsbereich von Transaktionen ausgeführt wird, aber keine neue Transaktion ausführt.
Required	Gibt an, dass für die Web Service-Methode eine Transaktion erforderlich ist. Da Web Services nur als Stammobjekt Teil einer Transaktion sein können, wird eine neue Transaktion für die Web Service-Methode erstellt.
RequiresNew	Gibt an, dass für die Web Service-Methode eine neue Transaktion erforderlich ist. Bei der Verarbeitung einer Anforderung wird der Web Service in einer neuen Transaktion erstellt.

Tab. 6.4: Mögliche Werte für TransactionOption

Standardmäßig wird die Transaktion mit dem Wert *AutoComplete* belegt. Dies bedeutet, dass man nicht explizit abstimmen muss, ob eine Transaktion erfolgreich verlaufen ist oder nicht. Das Attribut *System.EnterpriseServices.AutoCompleteAttribute* veranlasst ein Objekt, welches an einer Transaktion beteiligt ist, bei fehlerfreiem Beenden der Methode für einen Abschluss der Transaktion zu stimmen (*commit*). Wenn der Methodenaufruf eine Ausnahme auslöst, wird die Transaktion abgebrochen (*abort*). Für Web Service-Methoden wird dieses Attribut aber nicht benötigt, denn es ist von vornherein eingestellt.

Will ein Entwickler dennoch den Ausgang einer Transaktion steuern, dann bedient er sich der Helferklasse *ContextUtil*. Über die beiden wichtigsten Methoden *SetComplete* und *SetAbort* der Klasse *ContextUtil* lässt sich explizit entweder für das erfolgreiche Beenden (*commit*) oder das Zurücksetzen der Transaktion (*abort* bzw. *rollback*) stimmen.

```
[WebMethod(TransactionOption=
 TransactionOption.RequiresNew)]
public bool schreibeBenutzer(string benName,
   string vorName, string nachName, string passWort)
{
```

```
    SqlConnection oConn = new SqlConnection(
        "server=localhost;uid=BuchUser;pwd=ws_buch;
        database=WS_Buch");
    string sSQL = "INSERT INTO Benutzer Values ('" +
        benName + "', '" + vorName + "', '" + nachName +
        "', '" + passWort + "','" + DateTime.Now + "',
        null)";
    SqlCommand oCmd = new SqlCommand(sSQL, oConn);

    try
    {
        oConn.Open();
        int iRes = oCmd.ExecuteNonQuery();

        return true;
    }
    catch(Exception e)
    {
        return false;
    }
    finally
    {
        if (oConn.State == ConnectionState.Open)
            oConn.Close();
    }
}
```

Listing 6.14: Web Service-Methode zum transaktionsbasierten Schreiben in eine Datenbank

COM+-basierte Transaktionen sollten wirklich nur dann verwendet werden, wenn mehrere und vor allem verteilt verfügbare Datenquellen beteiligt sind. Für eine einzelne Datenquelle, die zudem meistens lokal ansprechbar ist, lohnt der Mehraufwand nicht, der bei der Erzeugung und der Koordination eines COM+-Objekts entsteht. In diesem Fall sollte ein Entwickler lokale ADO.NET-Transaktionen einsetzen (siehe Kapitel 4).

Im Beispiel in diesem Abschnitt wird der Einfachheit halber jedoch nur eine Datenquelle verwendet.

6.5 Zustand und Geschwindigkeit

Das Paradigma der XML Web Services lässt sich, wie bereits an anderen Stellen angemerkt, nicht für jedes Szenario gut und sinnvoll einsetzen. Doch eine beliebte Vorgehensweise ist es, der Öffentlichkeit große Datenmengen, die sich nur sehr selten ändern, über eine Web Service-Schnittstelle zur Verfügung zu stellen.

Eine andere Anforderung an Web-basierte Applikationen und somit auch an Web Services ist das Speichern von benutzer- oder anwendungsspezifischen Daten zwischen mehreren Aufrufen der Anwendung. Diesen beiden Themengebieten werden wir uns nun widmen.

6.5.1 Zustandsverwaltung

Herkömmliche Web-Anwendungen verwenden HTTP als Kommunikationsprotokoll. So tun dies auch ASP.NET-basierte XML Web Services (und auch viele andere Web Services-Implementierungen). Doch es liegt in der Natur von HTTP, dass es zustandslos ist. Dies bedeutet, dass der Server nach dem Zyklus der Anfrage-Antwort-Kommunikation nichts mehr vom Client weiß. Diese Tatsache ist aber in vielen Anwendungsfällen unzureichend. Daher gibt es in jeder Web Server-Technologie, egal ob CGI, ASP oder ähnliches, einen Mechanismus zum Verwalten von so genannten Sitzungen (*Sessions*).

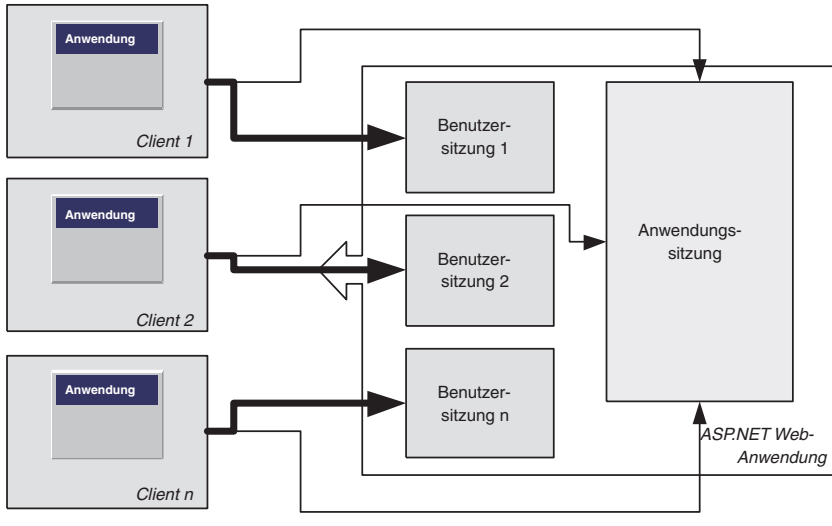


Abbildung 6.10: Benutzer- und Anwendungssitzung in einer Web-Anwendung

In ASP.NET unterscheidet man grundsätzlich zwischen Sitzungen mit Anwendungsgültigkeit und Sitzungen mit Benutzergültigkeit. Abbildung 6.10 soll diesen Unterschied verdeutlichen. Eine Anwendungssitzung ist innerhalb einer ASP.NET-Anwendung gültig und jeder Client, also Benutzer,

hat Zugriff auf die gleichen Daten. Eine Benutzersitzung hingegen ist für jeden einzelnen Benutzer einmalig vorhanden. In einem speziellen Speicherbereich können benutzerspezifische Daten gehalten werden.

Anwendungssitzung

In ASP.NET steht dem Programmier das Objekt *HttpApplicationState* für den Zugriff auf und die Verwaltung der Anwendungssitzung zur Seite. Zugriff auf die aktuelle *Application*-Instanz erhält man über das Abrufen der entsprechenden Eigenschaft des *HttpContext*-Objekts, welches Zugriff auf alle HTTP-spezifischen Informationen über eine einzelne HTTP-Anforderung bietet. Am einfachsten erhält man in einem ASP.NET Web Service Zugriff auf den HTTP-Kontext, indem man die Web Service-Klasse von der Klasse *System.Web.Service.WebService* ableitet. Somit ist der Quelltext in Listing 6.15 eine einfache Variante, um einen anwendungsweiten Zähler zu implementieren. Das Beispiel liegt in Visual Basic .NET vor. Durch Ableiten von der Klasse *WebService* kann man einfach über das *Application*-Objekt auf den Anwendungsspeicher zugreifen. Der Zugriff erfolgt sowohl lesend als auch schreibend über den Abruf eines Namens in einer *Collection*.

```
<%@ WebService Language="VB" Class="AppStateDemo" %>
Imports System
Imports System.Web.Services

Public Class AppStateDemo
   Inherits System.Web.Services.WebService

   <WebMethod()> _
   Public Function hochZaehlen(ByVal strName As String)
      As String
      Application("Counter") = Application("Counter")
         + 1

      Return "Hallo " & strName & " - wir stehen bei " &
         Application("Counter").ToString()
   End Function
End Class
```

Listing 6.15: Anwendungssitzung in einem ASP.NET XML Web Service

Benutzersitzung

Einen etwas anderen Weg muss man gehen, um Benutzersitzungen in Web Services verwenden zu können. Benutzersitzungen sind standardmäßig deaktiviert und müssen erst durch das Setzen einer Eigenschaft von *WebMethod* aktiviert werden. Diese Eigenschaft trägt die Bezeichnung *EnableState* und kann die Werte *true* oder *false* annehmen (*false* ist der vorbelegte Wert). Der Demo-Web Service in Listing 6.16 zeigt die Verwendung sowohl der Eigenschaft *EnableState* als auch der *Session*-Eigenschaft des HTTP-Kontexts im Einsatz.

Wenn man nun die Web Service-Testseite mit einem Browser aufruft und jeweils den Web Service für die Anwendungs- und die Benutzersitzung startet, wird sich der Zähler wie erwartet bei 1 startend erhöhen. Startet man aber beispielsweise einen neuen Browser (Netscape zusätzlich zum IE) und ruft dort auch wieder diese beiden Testseiten auf, dann sieht man, dass der Web Service für die Anwendungssitzung dort weiterzählt, wo er im anderen Browser aufgehört hat. Der Web Service für die Benutzersitzung aber fängt von vorne an zu zählen, was für einen eigenen Speicherbereich des neuen Clients auf Seiten von ASP.NET spricht.

```
<%@ WebService Language="VB" Class="SessionDemo" %>
Imports System
Imports System.Web.Services

Public Class SessionDemo
   Inherits System.Web.Services.WebService

   <WebMethod(EnableSession:=True)> _
   Public Function hochZaehlen(ByVal strName As String)
      As String
      Session("Counter") = Session("Counter") + 1

      Return "Hallo " & strName & " - wir stehen bei " &
         Session("Counter").ToString()
   End Function
End Class
```

Listing 6.16: Benutzersitzung in einem ASP.NET XML Web Service

Konfiguration. Die Benutzersitzung in ASP.NET kann vielseitig konfiguriert werden. Im Gegensatz zum klassischen ASP kann in ASP.NET die Sitzungsverwaltung in einen externen Prozess verlagert werden. Außerdem ist sie normalerweise auf Basis von HTTP Cookies implementiert, so auch in ASP und in ASP.NET. Allerdings kann man in ASP.NET durch eine einfache Konfigurationsanweisung angeben, dass die Sitzungen anstatt über Cookies über in den URL eingesetzte Sitzungs-IDs verwaltet werden. Für die Implementierung der Sitzungsfunktionalität ist nicht der IIS, sondern ein eigenes HTTP-Modul in der ASP.NET Pipeline zuständig.

Schauen wir uns zunächst die verschiedenen Möglichkeiten zur Haltung der Sitzungsdaten an. Standardmäßig werden die Sitzungsinformationen im Speicher der ASP.NET-Laufzeitumgebung gehalten. Dies hat zur Folge, dass, wenn der *aspnet_wp.exe*-Prozess – aus welchem Grund auch immer - beendet wird, auch alle Sitzungsdaten verloren gehen. Optional kann man die Daten in einen externen Windows-Dienst auslagern. Dieser Dienst ist als *aspnet_state.exe* implementiert und muss erst im System aktiviert werden (in der Windows-Diensteansicht erscheint er als ASP.NET STATE SERVICE). Alternativ kann man die Daten auch in einer SQL-Server-Datenbank speichern. Dies

hat den Vorteil, dass man so sehr einfach eine zentrale Datenbank für alle Sitzungsinformationen in einer Web Server-Farm mit mehreren beteiligten Web Servern erzeugen kann.

Wenn eine Client-Anwendung oder eine Plattform für Clients (z.B. mobile Endgeräte) keine HTTP-Cookies unterstützen, dann kann man ASP.NET so konfigurieren, dass die auf dem Server in einer Tabelle gehaltene Sitzungs-ID in den URL eingebaut wird. Ein URL für eine Web-Anwendung mit Sitzungsunterstützung sieht dann etwa so aus:

```
http://localhost/wsbuch/(ccdlk2555qhpmojsekuuu555)/K3/
   sportarten.aspx
```

Allerdings ist diese Technik beim Einsatz von Web Services nicht verwendbar. Denn ein Web Service-Client benötigt einen eindeutigen, statischen URL, um sich an einen Web Service zu binden.

Sämtliche Einstellungen nimmt man entweder in der Datei *machine.config* oder für jede Anwendung separat in der Datei *web.config* vor. Das allgemeine Format für die Konfiguration der Sitzungsverwaltung kann man in Listing 6.17 sehen.

```
<sessionState
   mode=
      "Off|Inproc|StateServer|SQLServer"
   cookieless="true|false"
   timeout="Anzahl der Minuten"
   stateConnectionString="tcpip=server:port"
   sqlConnectionString="SQL Connection String" />
```

Listing 6.17: Konfiguration der Sitzungsverwaltung in ASP.NET

Die entsprechende Erklärung der einzelnen Werte finden Sie in Tabelle 6.5.

Tab. 6.5: Werte für die Konfiguration der Sitzungsverwaltung in ASP.NET

Eigenschaft	Beschreibung
mode	*Off*: Gibt an, dass der Sitzungsstatus nicht aktiviert ist. *Inproc*: Gibt an, dass der Sitzungsstatus lokal gespeichert wird. *StateServer:* Gibt an, dass der Sitzungsstatus in einem externen Prozess gespeichert wird. *SQLServer*: Gibt an, dass der Sitzungsstatus in einer SQL Server-Datenbank gespeichert wird.
cookieless	Gibt an, ob Sitzungen ohne Cookies zum Identifizieren von Clientsitzungen verwendet werden sollen (in Verbindung mit XML Web Services nicht möglich).

Eigenschaft	Beschreibung
timeout	Gibt die mögliche Leerlaufdauer einer Sitzung in Minuten an, bevor die Sitzung abgebrochen wird. Der Standardwert ist 20.
stateConnectionString	Gibt den Servernamen und den Port an, auf dem der Sitzungsstatus in einem externen Prozess gespeichert wird. Beispiel: `"tcpip=127.0.0.1:42424"`. Dieser Wert ist erforderlich, wenn für *mode* der Wert *StateServer* festgelegt ist.
sqlConnectionString	Gibt die Verbindungszeichenfolge für einen SQL Server an. Beispiel: `"data source=127.0.0.1;uid=BuchUser; pwd=ws_buch"`. Dieser Wert ist erforderlich, wenn für *mode* der Wert *SQLServer* festgelegt ist.

Tab. 6.5: Werte für die Konfiguration der Sitzungsverwaltung in ASP.NET (Forts.)

Damit man den SQL Server als Datenquelle für ASP.NET-Sitzungsdaten verwenden kann, muss erst eine entsprechende Datenbank mitsamt Tabellen eingerichtet werden. Dies erfolgt durch die Ausführung des SQL-Skripts *InstallSqlState.sql,* das sich im .NET-Installationsverzeichnis befindet.

Eigene Typen. Eine der großen Verbesserungen, die ASP.NET in der Sitzungsverwaltung im Vergleich zum klassischen ASP mitbringt, ist das Speichern von eigenen Typen und komplexen Strukturen. Das Abspeichern erfolgt mithilfe der Serialisierungstechnik von *System.Runtime.Serialization* (vgl. Abschnitt 6.3). Somit wird an dieser Stelle ein in das .NET Framework eingebautes Feature verwendet, das sich einfach nutzen lässt. Wenn wir einen komplexen Typ namens `Person` definieren und eine Instanz davon im *Session*-Objekt ablegen wollen, geschieht das sehr einfach wie in Listing 6.18 gezeigt.

```
<%@ WebService Language="C#" Class="ComplexSession" %>
using System;
using System.Web.Services;

public class ComplexSession : WebService
{
   [WebMethod(EnableSession=true)]
   public void speicherePerson()
   {
      Person meinTyp = new Person();
      meinTyp.Name = "Weyer";
      meinTyp.Vorname = "Christian";

      Session["Person"] = meinTyp;
```

```
    }
}

public class Person
{
    public string Name;
    public string Vorname;
    public int Alter;
}
```

Listing 6.18: Speichern von komplexen Typen in einer Benutzersitzung

Client-Anwendungen. Einer der Nachteile von Cookie-basierten Sitzungen ist die Tatsache, dass die Client-Anwendungen mit Cookies umgehen können müssen. Im Falle des WWW-Browsers mit der ASP.NET-Testseite für XML Web Services ist dies nicht weiter tragisch. WWW-Browser können sowieso von sich aus mit Cookies umgehen. Aber was ist mit meiner ASP.NET- oder Windows-basierten Client-Applikation?

Für den Fall von .NET-Clients muss die Unterstützung für Cookies im erzeugten XML Web Service-Proxy aktiviert werden – egal ob es sich um eine ASP.NET Web-Anwendung oder um einen Windows Forms-Client handelt. Dies geschieht über eine einzige Zeile:

```
meineProxyInstanz.CookieContainer =
    new System.Net.CookieContainer();
```

Mit dieser Anweisung wird ein so genannter *Cookie Container* erzeugt, der aber nur für die Lebensdauer dieser Proxy-Instanz gültig ist und das Auslesen bzw. Setzen der Cookie-Werte im entsprechenden HTTP Header übernimmt.

6.5.2 Caching

Damit man große oder sich selten verändernde Daten nicht immer wieder neu berechnen oder aus einer Datenbank holen muss, kann man das Caching-Verfahren einsetzen. Bei diesem Verfahren werden Daten für einen bestimmten, meist konfigurierbaren Zeitraum im Speicher einer Anwendung gehalten und die Anfragen einer anderen Anwendung bezüglich dieser Daten werden aus diesem Cache-Speicher bedient. Für ASP.NET Web Services gibt es hier zwei grundlegende Möglichkeiten.

Die eine Möglichkeit ist die Verwendung der *CacheDuration*-Eigenschaft des *WebMethod*-Attributs. Diese Eigenschaft ermöglicht die Aktivierung des Ausgabe-Cachings. Beim Ausgabe-Caching wird der Web Service einmal abgearbeitet und dann innerhalb des vorgegebenen Zeitintervalls nicht mehr (Angabe erfolgt in Sekunden). Dies bedeutet, dass der Web Service bei 50 Anfragen innerhalb des Intervalls nicht mehr 50 Mal, sondern nur ein einziges Mal abgearbeitet wird. Alle restlichen 49 Anfragen werden aus dem

Cache bedient. Dieser Effekt lässt sich schön an einem kleinen Web Service zeigen, der die aktuelle Uhrzeit ausgeben soll. In unserem Fall des aktivierten Ausgabe-Cachings wird 20 Sekunden lang immer die Uhrzeit zurückgegeben, die aktuell war, als der Web Service zum ersten Mal abgearbeitet wurde (siehe Listing 6.19).

```
<%@ WebService Language="C#" Class="AusgabeCache" %>
using System.Web.Services;

public class AusgabeCache
{
   [WebMethod(CacheDuration=20)]
   public string gibUhrzeit()
   {
      return System.DateTime.Now.ToString();
   }
}
```

Listing 6.19: Ausgabe-Caching in ASP.NET XML Web Service

Eine andere Variante ist das Daten-Caching. Hier werden die Daten in einer Anwendung oder einem Web Service in einem speziellen Speicherbereich gehalten. Dies ist vor allem für Datenbankanwendungen sehr interessant, bei denen große *DataSets* zurückgegeben werden müssen. Im Rahmen von ASP.NET wird das Daten-Caching durch die Klasse *System.Web.Caching.Cache* ermöglicht. Man hat über den HTTP-Kontext direkten Zugang zu einer Instanz dieser Klasse. Das Beispiel in Listing 6.20 zeigt einen Web Service, der Daten aus der Benutzerdatenbank ausliest und diese dann in das Cache-Objekt schreibt. Für eine Invalidierung des Caches muss der Programmierer allerdings selbst sorgen.

```
<% @WebService Class="DatenCaching" Language="C#" %>
using System;
using System.Data;
using System.Data.SqlClient;
using System.Web.Services;

public class DatenCaching : WebService
{
   [WebMethod]
   public DataSet holeBenutzer()
   {
      DataSet dsBenutzer;

      if (Context.Cache["BenutzerDS"] == null)
      {
         string strSQL;
         SqlDataAdapter cmdBenutzer;
         SqlConnection conn = new SqlConnection
```

```
            ("server=localhost;uid=BuchUser;
            pwd=ws_buch;database=WS_Buch");

         try
         {
            conn.Open();
            dsBenutzer = new DataSet();
            strSQL = "SELECT * FROM Benutzer";
            cmdBenutzer = new SqlDataAdapter(strSQL,
               (SqlConnection)conn);
            cmdBenutzer.Fill(dsBenutzer, "Benutzer");

            Context.Cache["BenutzerDS"] = dsBenutzer;
         }
         catch (Exception ex)
         {
            throw ex;
         }
         finally
         {
            if (conn.State == ConnectionState.Open)
               conn.Close();
         }
      }
      else
         dsBenutzer = (DataSet)Context.Cache[
            "BenutzerDS"];

      return dsBenutzer;
   }
}
```

Listing 6.20: Daten-Caching in ASP.NET XML Web Service

Das *DataSet* wird anfänglich aus der Datenbank gefüllt, denn der entsprechende Cache-Eintrag existiert zu diesem Zeitpunkt noch nicht. Wenn der Cache-Eintrag aber erst einmal vorhanden ist, dann wird das zurückzugebende *DataSet* aus dem Cache gefüllt.

6.6 Arbeiten mit SOAP Headern

Wie ich bereits in Kapitel 5 angedeutet habe, ist SOAP von seinem Aufbau her sehr modular und erweiterbar gestaltet. Ebenso verhält es sich mit dem .NET Framework und ASP.NET im Besonderen. Zum Abschluss dieses Kapitels möchte ich Ihnen zeigen, wie Sie mit SOAP Headern ihren XML Web Service erweitern können.

Die SOAP-Spezifikation enthält einen Abschnitt über SOAP Header. SOAP Header sind optionaler Bestandteil des SOAP Envelopes. Sie können dafür verwendet werden, Daten und Informationen in einer SOAP-Nachricht zu übertragen, die orthogonal zu den eigentlichen Anwendungsnutzdaten stehen. Beispiele hierfür sind das Propagieren von Transaktions-IDs oder das Übermitteln von Benutzerdaten für jede SOAP-Anfrage. Anhand des letzten Beispiels möchte ich Ihnen die Verwendung von SOAP Headern im Rahmen von ASP.NET XML Web Services demonstrieren.

Ein ideales Paar in ASP.NET XML Web Services sind SOAP Header in Verbindung mit *SOAP Extensions*. Eine SOAP Extension ist eine Implementierung der Klasse *System.Web.Services.Protocols.SoapExtension* und kann dafür verwendet werden, den serialisierten XML-Strom zu verschiedenen Zeitpunkten der Abarbeitung zu beeinflussen. Es ist somit möglich, vor und nach der Serialisierung bzw. Deserialisierung sowohl auf der Client- als auch auf der Web Service-Seite eigene Aktionen und Algorithmen zu implementieren. SOAP Extensions sind eine fortgeschrittene Technik, die hier nicht näher beleuchtet wird.

Zuständig für die Umsetzung von SOAP Headern ist die Klasse *SoapHeader* in Verbindung mit der Klasse *SoapHeaderAttribute*, beide im Namensraum *System.Web.Services.Protocols*. Die allgemeine Vorgehensweise sieht vor, dass für jeden geplanten SOAP Header eine eigene Klasse existieren muss, welche von *SoapHeader* abgeleitet ist. Über die Verwendung des Attributs *SoapHeader* wird dann für jede einzelne Web Service-Methode angezeigt, dass sie diesen SOAP Header verarbeiten kann. Dieses Attribut kann übrigens auch für eine Client-seitige Implementierung etwa in einem Web Service-Proxy gesetzt werden, da die Kommunikation ja bidirektional verlaufen kann. Damit der SOAP Header entsprechend bearbeitet werden kann, muss eine Web Service-globale Instanz der *SoapHeader*-Klasse definiert werden. Der Demo-Web Service in Listing 6.21 zeigt das prinzipielle Vorgehen auf Seiten des Web Services.

```
<%@ WebService Language="C#" Class="SoapHeaderDemo" %>
using System.Web.Services;
using System.Web.Services.Protocols;

public class MeinHeader : SoapHeader
{
   public string benutzerName;
   public string passWort;
}

public class  SoapHeaderDemo
{
   public MeinHeader benutzerDaten;
```

```
   [WebMethod]
   [SoapHeader("benutzerDaten")]
   public string zeigeBenutzerName()
   {
      return benutzerDaten.benutzerName;
   }
}
```

Listing 6.21: Verwendung von SOAP Headern in ASP.NET XML Web Services

Nun müssen wir noch einen XML Web Service-Client erstellen, der die Daten des SOAP Headers setzt und an den Web Service sendet. Da die ASP.NET-Laufzeitumgebung bei der dynamischen Generierung des WSDLs auch die SOAP Header mit in die WSDL-Beschreibung integriert, genügt die bisher eingesetzte Vorgehensweise: Erstellung eines Proxys mit *wsdl.exe* (oder der WEBVERWEIS HINZUFÜGEN-Funktionalität von Visual Studio .NET). Die relevanten Teile des Proxy-Codes sehen Sie in Listing 6.22. Vor allem das Attribut *SoapHeader* vor der Proxy-Methode `zeigeBenutzerNamen` ist für uns von Interesse. Dies zeigt nämlich an, dass diese Web Service-Methode mit einem SOAP Header arbeitet.

```
[System.Web.Services.WebServiceBindingAttribute(
   Name="SoapHeaderDemoSoap",
   Namespace="http://tempuri.org/")]
public class SoapHeaderDemo :
   System.Web.Services.Protocols.SoapHttpClientProtocol
{
   public MeinHeader MeinHeaderValue;

   public SoapHeaderDemo()
   {
      this.Url =
         "http://localhost/wsbuch/K6/SoapHeader.asmx";
   }

[System.Web.Services.Protocols.SoapHeaderAttribute(
   "MeinHeaderValue")]
[System.Web.Services.Protocols.
   SoapDocumentMethodAttribute("
   http://tempuri.org/zeigeBenutzerNamen",
   RequestNamespace="http://tempuri.org/",
   ResponseNamespace="http://tempuri.org/",
   Use=System.Web.Services.Description.
    SoapBindingUse.Literal,
   ParameterStyle=System.Web.Services.
    Protocols.SoapParameterStyle.Wrapped)]
public string zeigeBenutzerNamen()
   {
```

```
      object[] results = this.Invoke(
         "zeigeBenutzerNamen", new object[0]);
      return ((string)(results[0]));
   }
}

[System.Xml.Serialization.XmlTypeAttribute(
   Namespace="http://tempuri.org/")]
[System.Xml.Serialization.XmlRootAttribute(
   Namespace="http://tempuri.org/", IsNullable=false)]
public class MeinHeader : SoapHeader
{
   public string benutzerName;
   public string passWort;
}
```

Listing 6.22: Relevante Teile des erzeugten Proxy-Codes für die Verarbeitung von SOAP Headern

Damit wir auch wirklich testen können, ob unser Web Service mit SOAP Headern richtig arbeitet, erstellen wir eine kleine Konsolentestanwendung (siehe Listing 6.23). Wir erzeugen jeweils eine Instanz der bereits bekannten Proxy-Klasse und der Header-Klasse. Die Felder der Header-Instanz füllen wir mit Werten und weisen diese Header-Instanz der Web Service Proxy-Instanz zu. Abschließend rufen wir zur Kontrolle dann die Methode auf, die uns den vorher im SOAP Header übermittelten Wert des Benutzernamens ausgibt.

```
using System;

class SoapHeaderClient
{
   static void Main(string[] args)
   {
      SoapHeaderDemo soapDemo = new SoapHeaderDemo();
      MeinHeader meinHeader = new MeinHeader();

      meinHeader.benutzerName = "CW";
      meinHeader.passWort = "testi";
      soapDemo.MeinHeaderValue = meinHeader;

      Console.WriteLine(soapDemo.zeigeBenutzerNamen());
   }
}
```

Listing 6.23: Client-Anwendung mit SOAP Header-Unterstützung

Durch das Setzen der Eigenschaft *Direction* des *SoapHeader*-Attributs lässt sich zusätzlich noch die Richtung des SOAP Headers bestimmen. Dies entscheidet, ob die Daten des Headers vom Client zum Web Service, vom Web Service zum Client oder in beide Richtungen gehen sollen.

Wenn man in seinem Web Service oder seinem Web Service-Client einen Header einbauen möchte, der unbedingt vom jeweiligen Kommunikationspartner verstanden werden muss, dann kann man dies über die Eigenschaft *Required* erledigen. Folgendes Fragment teilt der Laufzeitumgebung mit, dass der SOAP Header unbedingt verstanden werden muss (vgl. auch Kapitel 5):

```
[WebMethod]
[SoapHeader("MeinPflichtHeader", Required=true)]
…
```

6.7 Zusammenfassung

XML Web Services sind sehr viel mehr als nur SOAP und WSDL. Man kann mit XML-basierten Web Services sehr viele Szenarien implementieren, gelangt aber auch schnell an ihre Grenzen – wie z.B. beim Verarbeiten von binären Daten. In diesem Kapitel habe ich Ihnen einige weiterführende Techniken und Ansätze vorgestellt, die man in einer täglichen Implementierung auf Basis von ASP.NET anwenden kann.

Vor allem die Weiterverwendung von bereits existierenden COM-Komponenten auch in Bezug auf die Dienste von COM+ ist eine große Erleichterung für jeden .NET-Programmierer. Ein wichtiges weiterführendes Thema wurde in diesem Kapitel allerdings noch nicht behandelt: Die Absicherung von XML Web Services und die sichere Kommunikation von Client-Anwendungen mit Web Services. Diesem Thema widmen wir uns in Kapitel 7.

7 Sicher ist sicher

Sicherheit in .NET und XML Web Services

7.1 Einführung

Kaum ein anderes Thema wird innerhalb von Unternehmen intensiver und heißer diskutiert als Sicherheit. Was nicht heißt, dass Sicherheit immer und überall an erster Stelle steht, im Gegenteil: Sie wird oft vernachlässigt. Wenn aber erst einmal Konzept und Infrastruktur für Sicherheit festgelegt und implementiert sind, dann wird sehr peinlich darauf geachtet, dass die zugehörigen Richtlinien auch eingehalten werden.

7.1.1 Sicherheit als Notwendigkeit

Zu oben genannten Richtlinien zählen u.a. auch der Einsatz von Firewalls. Diese sollen das Intranet des Unternehmens schützen und nur bestimmten, kontrollierbaren Datenaustausch ermöglichen. Die Administratoren der Firewalls achten sehr genau darauf, dass wirklich nur die wichtigsten und benötigten Ports in der Firewall offen gelegt sind. Und dann kommt das XML Web Services-Konzept mit SOAP als Basistechnologie und untergräbt die Firewall-Grenzen. SOAP ermöglicht die Kommunikation über HTTP und erlaubt somit die Durchdringung von Firewalls über den Standard-Port 80. Bei diesem Gedanken bekommen viele Netzwerkverwalter Bauchschmerzen.

Es sind also einige Maßnahmen erforderlich, um eine Web Service-basierte Anwendung gegen ungewollte Nutzung zu schützen. Zusätzlich erhält man mit .NET noch die Möglichkeit, fertige Code-Stücke in Form von Assemblies aus dem Intranet oder dem Internet auf seinen eigenen Computer zu laden und dort auszuführen (vgl. Kapitel 2). Dieses Konzept ist zwar sehr bequem wenn es um die Verwendung von jeweils aktuellen Versionen geht, bringt allerdings auch große Sicherheitsrisiken mit sich.

Alle diese Probleme, die mit Internet-Anwendungen allgemein und XML Web Services- und .NET-Applikationen im Speziellen verbunden sind, werden in diesem Kapitel betrachtet, um mögliche Lösungen für eine sichere Betriebsumgebung zu finden.

7.1.2 Allgemeine Konzepte

Wenn man von Sicherheit spricht, dann gleicht dies oft einer Unterhaltung über Autos: Es gibt viele verschiedene Typen, verschiedene Umsetzungen und nicht jedes Auto eignet sich für alle Einsatzszenarien. Ähnlich ist es mit dem Thema Sicherheit. In diesem Abschnitt werde ich kurz auf grundlegende Prinzipien und Begriffe der aktuellen Sicherheitsdiskussion eingehen.

Es gibt allgemein gesprochen vier Säulen einer modernen Sicherheitspolitik:

- Datenschutz/Verschlüsselung,
- Authentifizierung,
- Autorisierung,
- Unleugbarkeit des Ursprungs.

Diese Säulen stehen auf unterschiedlichen Ebenen. Man muss nicht jedes der vier Prinzipien implementieren, um eine sichere Anwendung zu erhalten. Es kommt ganz auf das Einsatzgebiet an.

Datenschutz / Verschlüsselung

Unter Datenschutz verstehe ich in diesem Fall die Tatsache, dass übertragene Datenmengen nicht einfach von einem im Internet surfenden Benutzer betrachtet und verwendet werden können. Im Fall einer SOAP-basierten Kommunikation wird XML im Klartext über HTTP versendet. Datenschutz bedeutet hier, dass diese Daten verschlüsselt werden, damit kein Dritter unbefugt diese Daten (in einem vertretbaren Zeitaufwand) lesen kann. Zur Verschlüsselung von Daten wird ein Schlüssel benötigt. Hier gibt es zwei Prinzipien, wie man Schlüssel verwendet: Symmetrische und asymmetrische Verfahren.

Bei symmetrischen Verfahren (wie z.B. *Rijndael*) wird zum Verschlüsseln und zum Entschlüsseln der gleiche Schlüssel verwendet. Der Vorteil dieses Ansatzes ist, dass die Schlüssel relativ kurz sein können (im Vergleich zur asymmetrischen Methode). Der große Nachteil ist die potentiell unsichere Methode, wie man die Schlüssel sicher austauscht, so dass alle beteiligten Parteien die Daten ver- und entschlüsseln können.

Bei den asymmetrischen Verfahren (beispielsweise *RSA*) werden immer Schlüsselpaare eingesetzt: ein privater und ein öffentlicher Schlüssel. Der private Schlüssel ist nur dem Benutzer bekannt, während der öffentliche Schlüssel über entsprechende Mechanismen und Infrastrukturen (den so genannten *Public Key Infrastructures*, *PKI*) der Öffentlichkeit bekannt gemacht wird. Der private Schlüssel wird dann zum Verschlüsseln verwendet. Diese Daten können mit dem öffentlichen Schlüssel wieder entschlüsselt werden. Zusätzlich kann man mit dem privaten Schlüssel eine Nachricht signieren – dies fällt allerdings mehr unter den Punkt Unleugbarkeit. In der Gegenrichtung kann ein Kommunikationspartner Daten mit dem öffentlichen Schlüssel eines

Benutzers verschlüsseln, die dieser dann nur mit dem privaten Schlüssel entschlüsseln kann. Alles in allem stehen hinter diesen Konzepten und Algorithmen komplexe mathematische Gebilde, die es so weit zu beweisen gilt, dass das Brechen eines Schlüssels oder einer Verschlüsselung in einem vertretbaren Aufwand nicht möglich ist.

Speziell für den Bereich der Web-basierten Anwendungen – und XML Web Services sind eine besondere Art davon – gibt es eine weit verbreitete Lösung für die Verschlüsselung des Datenverkehrs. *Secure Sockets Layer* (*SSL*) ist ein Protokoll, das anhand von Zertifikaten eine verschlüsselte und vertrauenswürdige Kommunikation z. B. über HTTP zulässt.

Authentifizierung

Eine andere Ebene der Sicherheit ist die Authentifizierung. Darunter versteht man die Überprüfung, ob ein Benutzer dem System bekannt ist. Wenn er bekannt ist, müssen die zur Verfügung gestellten Daten (meistens Benutzername und Kennwort) korrekt sein. Ist dies der Fall, ist der Benutzer erfolgreich authentifiziert. Alternative Methoden zur Überprüfung der Identität sind beispielsweise biometrische Verfahren oder Smart Cards.

Die Authentifizierung ist die einfachste Variante, einer vorgegebenen Gruppe bzw. Menge an registrierten Benutzern Zugang zu einem System zu verschaffen. Sie ist auch in Web Service-Anwendungen von Bedeutung. Wenn man einen Web Service anbietet, der Daten aufgrund von Benutzerinformationen oder nur gegen Bezahlung liefert, dann ist die Authentifizierung dieser Benutzer unabdinglich.

Autorisierung

Bei der Autorisierung handelt es sich, vereinfacht gesagt, um eine Zugangskontrolle. Es wird überprüft, ob ein Benutzer Zugang zu gewissen Ressourcen besitzt.

Autorisierung ist in Windows-Systemen mit *Access Control Lists* (*ACLs*, Zugriffskontrolllisten) implementiert. Jede Ressource hat eine ACL, die anzeigt, welcher Benutzer welche Aktion auf der Ressource ausführen darf. In einer Web-Anwendung ist Autorisierung ungleich wichtiger, da ja Aktionen auf einem dem Benutzer fremden System ausgeführt werden.

Unleugbarkeit

Der Punkt der Unleugbarkeit mag auf den ersten Blick etwas seltsam anmuten. Hiermit ist vor allem die Unleugbarkeit des Ursprungs bzw. der Herkunft einer Nachricht gemeint. Gleichwohl verbindet man damit auch die Tatsache, dass nicht vorgegeben werden kann, die Nachricht stamme aus einem anderen Ursprung.

Alle anderen Konzepte wie Verschlüsselung, Authentifizierung und Autorisierung greifen nicht effektiv, wenn ein potenzieller Angreifer in der Lage ist, eine Nachricht im Namen eines anderen Benutzers abzusenden und die

genannten Mechanismen auszuhebeln. Mithilfe von digitalen Zertifikaten, die Daten über den Benutzer beinhalten, ist eine Überprüfung des Ursprungs der Nachricht möglich. Zusätzlich lässt sich anhand einer Signatur auch überprüfen, ob die Nachricht auf dem Kommunikationsweg verändert wurde.

In den folgenden Abschnitten werde ich Ihnen die wichtigsten Sicherheitslösungen in Bereichen der .NET-Entwicklung und vor allem für die Programmierung und den Betrieb sicherer ASP.NET- und XML Web Services-Applikationen näher bringen.

7.2 Sicherheitskonzepte in .NET

Mit dem .NET Framework hält auch eine neue Gefahr Einzug in die Softwareentwicklung. Diese Gefahr ist jedoch nicht .NET-spezifisch, sondern rührt von der Natur des Internets und der Art und Weise, wie zukünftig Software erstellt und verteilt wird. Wenn nämlich Code von einem fremden Server heruntergeladen wird, dann muss ich diesem Code vertrauen. Dieses Vertrauen muss aber auch auf technischer Ebene abgesichert sein. Andererseits muss ich Code vertrauen, der meine Funktionalität in Anspruch nehmen möchte. In .NET heißt das Konzept zur Absicherung von Code Code-Zugriffssicherheit (*Code Access Security*).

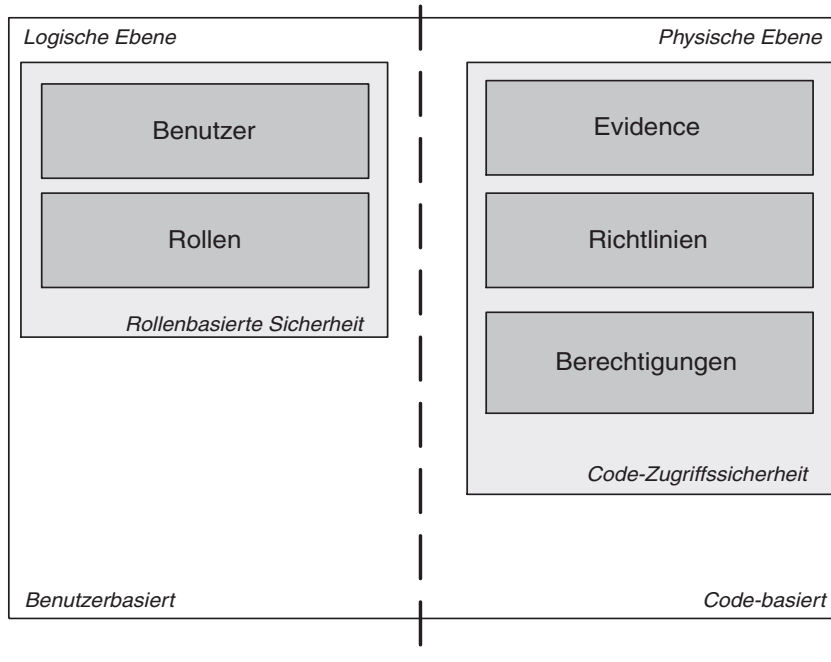


Abbildung 7.1: Zwei grundlegende Sicherheitsmodelle im .NET Framework

Während die Code-Zugriffssicherheit auf Ebene des Codes aktiv wird, agiert das zweite Konzept der Rollen-basierten Sicherheit auf der Ebene des Benutzers. Man kann diese beiden Konzepte also in eine logische und eine physische Sichtweise unterteilen (vgl. Abbildung 7.1).

7.2.1 Code-Zugriffssicherheit

Mit der Code-Zugriffssicherheit führt Microsoft erstmals eine neue Sicherheitsschicht in ein Softwaresystem ein. Alle verfügbaren Sicherheitsimplementierungen drehen sich um die Benutzer bzw. um Rollen, denen Benutzer angehören. Mit dem Ansatz der Code-Zugriffssicherheit geht Microsoft mit .NET den Schritt hin zur Code-basierten Überprüfung von Rechten und Berechtigungen und trägt damit der sich verändernden Internet-Landschaft Rechnung. Allgemein gesagt geht es bei der Code-Zugriffssicherheit um das Vertrauen, das Code – vor allem fremdem Code – ausgesprochen wird. Mit der Code-Zugriffssicherheit wird ermöglicht, dass Code unter einer Benutzeridentität in theoretisch unendlich vielen Vertrauensstufen ausgeführt werden kann. So kann Code aus dem Internet restriktiver behandelt werden als Code auf der lokalen Festplatte – selbst wenn der aktive Benutzer des Betriebssystems vollen Zugriff auf alle Systemressourcen hat.

Evidence

Die Code-Zugriffssicherheit erteilt Berechtigungen an Assemblies auf Basis der *Evidence* einer Assembly – Evidence im Sinne von Nachweis oder Beweismittel. Sie verwendet dafür den Ort, von dem der Code stammt, und weitere Informationen über die tatsächliche Identität des Codes, um herauszufinden, auf welche Ressourcen der Code Zugriff hat. Diese Informationen nennt man die Evidence des Codes bzw. der Assembly. Das bedeutet also, dass die Beweislast auf Seiten der Assembly liegt, damit die CLR sichergehen kann, dass der Code auch vertrauenswürdig ist.

Anhand dieser Evidence einer Assembly entscheidet das Code-Zugriffssicherheitssystem in der CLR, welche Ressourcen benutzt werden dürfen, indem es die Evidence auf eine Menge von Berechtigungen (*permissions*) abbildet. Diese Abbildung wird durch eine administrative Tätigkeit in so genannten Sicherheitsrichtlinien (*security policies*) festgelegt.

Das .NET Framework enthält standardmäßig verschiedene Klassen, die für Evidence stehen. Um genauer zu verstehen, was Evidence ist und wie sie aussieht, wollen wir uns die Liste der Klassen anschauen und anhand eines Beispiels die Evidence einer Assembly betrachten:

- *Zone*: Konzept der Zonen, wie sie auch im Internet Explorer vorzufinden sind.
- *URL*: URL oder Ort im Dateisystem zur Identifikation einer Ressource.
- *Hash*: Der Hash-Wert einer Assembly.

- *Strong Name*: Der Strong Name einer Assembly.
- *Site*: Der Server, von dem der Code kommt. *URL* ist wesentlich spezifischer als *Site*.
- *Application Directory*: Das Verzeichnis, aus dem der Code geladen wurde.
- *Publisher Certificate*: Das Zertifikat (digitale Signatur) einer Assembly.

Diese Liste enthält Klassen, die für die Evidence einer Assembly verwendet werden können. Allerdings kann ein Entwickler auch eigene Evidence-Klassen erstellen. Als Beispiel nehmen wir die Assembly *mscorlib.dll*. Der Quelltext in Listing 7.1 sorgt dafür, dass diese Assembly geladen wird und gibt die Daten über ihre Evidence in der Kommandozeile aus (siehe Listing 7.2).

```
using System;
using System.Collections;
using System.Reflection;
using System.Security.Policy;

class MainClass
{
   static void Main(string[] args)
   {
      Type myType = Type.GetType("System.String");
      Assembly ass = Assembly.GetAssembly(myType);
      Evidence ev = ass.Evidence;
      IEnumerator enumEv = ev.GetEnumerator();

      while(enumEv.MoveNext())
         Console.WriteLine(enumEv.Current);
   }
}
```

Listing 7.1: Ermitteln der Evidence einer Assembly

Wenn Sie dieses Beispiel ausführen, wird die Assembly *mscorlib.dll* inspiziert und die Evidence-Informationen in die Kommandozeile geschrieben. Es handelt sich hier um eine der wichtigsten Assemblies im .NET Framework, daher ist sie sehr umfangreich und auch ihr Hash-Wert ist sehr groß. Für die beispielhafte Ausgabe in Listing 7.2 habe ich den Hash-Wert nicht angegeben, da dieses Buch sonst ca. 50 Seiten länger geworden wäre …

```
<System.Security.Policy.Zone version="1">
   <Zone>MyComputer</Zone>
</System.Security.Policy.Zone>

<System.Security.Policy.Url version="1">
```

```
   <Url>file://D:/windows/microsoft.net/
      framework/v1.0.3705/mscorlib.dll</Url>
</System.Security.Policy.Url>

<StrongName version="1"
   Key="00000000000000000400000000000000"
   Name="mscorlib"
   Version="1.0.3300.0"/>

<System.Security.Policy.Hash version="1">
   <RawData>... Rohdaten ...</RawData>
</System.Security.Policy.Hash>
```

Listing 7.2: Evidence der Assembly mscorlib.dll

Berechtigungen

Eine Berechtigung ist ein essentieller Bestandteil eines Sicherheitssystems. In .NET ist eine Berechtigung die Basis für das gesamte Sicherheitskonzept. Eine Berechtigung wird in .NET durch die Schnittstelle *IPermission* repräsentiert. Die grundlegende Struktur für alle Berechtigungen im Bereich Code-Zugriffssicherheit definiert die Klasse *CodeAccessPermission* im Namensraum *System.Security.* Diese Klasse implementiert die zuvor genannte Schnittstelle.

Die Berechtigungen in der Code-Zugriffssicherheit werden also dazu genutzt, Ressourcen, Geräte oder Operationen vor nicht autorisierter Verwendung zu schützen. Anders gesagt stellen sie das Recht dar, auf etwas zuzugreifen oder etwas auszuführen. Die Berechtigungen werden innerhalb des so genannten *Stack Walks* überprüft (siehe nächster Abschnitt). Durch diese Vorgehensweise stellt das Sicherheitssystem in der CLR sicher, dass die Berechtigungen nicht nur im aktuellen Kontext der Assembly, sondern in der gesamten Umgebung, in welcher der Code ausgeführt wird, vorhanden sind. Somit wird das Betriebsystem bzw. das ausführende System, der eigene Code und der aufgerufene Code geschützt.

Stack Walk

Damit die CLR bzw. das Sicherheitssystem bestimmen kann, ob Code über die erforderliche Berechtigung zum Zugriff auf eine Ressource oder zum Ausführen einer Operation verfügt, durchläuft es den Aufruf-Stack und vergleicht die erteilten Berechtigungen der einzelnen Aufrufer mit der geforderten Berechtigung. Wenn ein Aufrufer im Aufruf-Stack nicht über die geforderte Berechtigung verfügt, wird eine Ausnahme ausgelöst, was zur Verweigerung des Zugriffs führt.

Der Stack Walk soll nun verhindern, dass wenig vertrauenswürdiger Code hoch vertrauenswürdigen Code aufruft und dadurch unzulässige Operationen ausführt. Abbildung 7.2 veranschaulicht den Stack Walk, der entsteht, wenn eine Methode in Assembly A4 fordert, dass ihre Aufrufer über

die Berechtigung E verfügen. In der Abbildung stehen die Buchstaben E für Berechtigung im Sinne von Erlaubnis und B für Berechtigung im Sinne von Berechtigungsregeln.

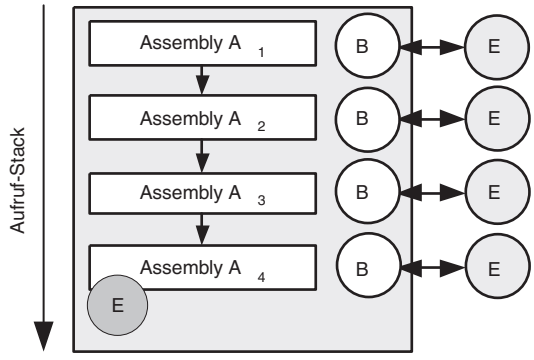


Abbildung 7.2: Sicherheitsüberprüfung im Aufruf-Stack

Sicherheitsrichtlinien

Durch die Aufstellung und Beachtung von Sicherheitsrichtlinien wird eine Abbildung zwischen der Evidence einer Assembly und der Menge an Berechtigungen für diese Assembly hergestellt. Diese Funktionalität wird vom *Security Manager* bereit gestellt, der in der Klasse *System.Security.SecurityManager* implementiert ist. Es gibt vier vordefinierte Ebenen für Sicherheitsrichtlinien, die wie folgt aussehen:

- *Enterprise Policy Level*: Richtlinie für unternehmensweiten Einsatz,
- *Machine Policy Level*: Richtlinie für alle Benutzer eines Computers,
- *User Policy Level*: Richtlinie für den aktiven (angemeldeten) Benutzer,
- *Application Domain Policy Level:* Richtlinie für eine Application Domain.

Die ersten drei Ebenen können durch einen Administrator konfiguriert werden, während die Ebene für Application Domain Policy programmatisch durch den jeweiligen CLR Host manipulierbar ist.

Damit der Security Manager die Menge der Berechtigungen für eine Assembly aufgrund der Sicherheitsrichtlinie bestimmen kann, beginnt er mit der Untersuchung der obersten Ebene (Enterprise). Ich möchte an dieser Stelle das Sicherheitsrichtlinienkonzept der Code-Zugriffssicherheit mit einer mathematischen Funktion vergleichen: In die Funktion werden zwei Eingabewerte, nämlich die Evidence und die administrativ festgelegten Sicherheitsrichtlinien, eingegeben. Als Ergebnis liefert die Funktion die Menge der Berechtigungen für eine Assembly. Daher erhält der Security Manager für jede Ebene eine Menge an Berechtigungen als Ergebnis. Aber nur die Schnittmenge aller Ergebnisse ist die gültige Menge an Berechtigungen. Dies bedeutet also, dass alle Ebenen eine gewisse Berechtigung für die Assembly erteilen müssen, bevor die Assembly diese Berechtigung dann schlussendlich auch erhält.

Die Sicherheitsrichtlinien werden mithilfe einer Hierarchie von so genannten Code-Gruppen konfiguriert. Es können mehrere Code-Gruppen pro Ebene existieren. Standardmäßig existieren Gruppen, die ihrerseits wieder Code-Gruppen als Kindelemente haben. Somit wird ein Baum von Gruppen für eine Ebene aufgespannt. Jede Code-Gruppe hat eine Mitgliedschaftsbedingung (*membership condition*) und eine Berechtigungsmenge (*permission set*). Wenn die Evidence einer Assembly mit der Mitgliedschaftsbedingung übereinstimmt, dann wird die entsprechende Berechtigungsmenge der Assembly für diese Ebene zugesagt. Die Standardinstallation von .NET kommt mit einer vorkonfigurierten Anzahl von Berechtigungsmengen.

Die Verwaltung von Code-Zugriffssicherheit und der Richtlinien inkl. Berechtigungsmengen ist entweder mit der Kommandozeilenanwendung *caspol.exe* oder mit der Anwendung MICROSOFT .NET FRAMEWORK-KONFIGURATION (*mscorcfg.msc*) möglich. Bei *mscorcfg.msc* handelt es sich um ein Plug-In für die Microsoft Management Console (MMC) (vgl. Abbildung 7.3).

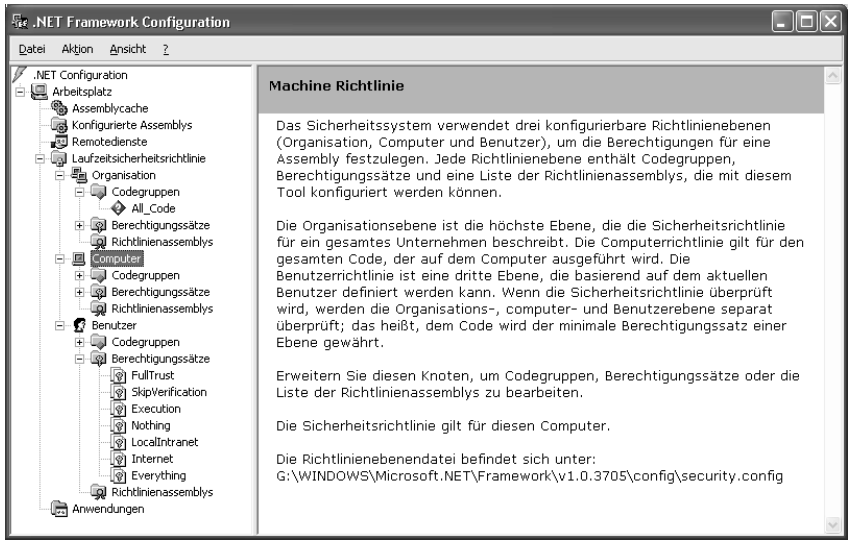


Abbildung 7.3: Konfiguration der .NET-Sicherheit über die MMC

Die Programmierung mit der .NET-Sicherheit ist auf zwei Arten möglich. Zum einen kann man über den deklarativen Ansatz mittels Attributen programmieren, andererseits kann man mit der imperativen Vorgehensweise über Instanzen der *Permission*-Klasse (und deren Ableitungen) Methoden aufrufen, wie folgende Zeilen zeigen:

```
// deklarative Anforderung einer Berechtigung
[PrincipalPermission(SecurityAction.Demand,
   Name=@"PCDIANA\cw")]
```

```
// imperative Anforderung einer Berechtigung
FileIOPermission ioPerm = new FileIOPermission(
   FileIOPermissionAccess.Write, file);
p.Demand();
```

Einer der Vorteile der deklarativen Programmierung liegt darin, dass im Gegensatz zur imperativen Programmierung alle Sicherheitsaktionen ausgeführt werden können. Allerdings können Werkzeuge über die Reflection-Funktionalität von .NET diese Metadaten auslesen und verarbeiten. Die Vorzüge der imperativen Programmierung liegen in der höheren Flexibilität, da sie auch in einem bestimmten Pfad innerhalb des Codes angewendet werden kann, während die Programmierung über die Attribute immer nur auf Assembly-, Klassen- oder Methodenebene greift.

7.2.2 Rollen-basierte Sicherheit

Nachdem wir im vorangegangenen Abschnitt hauptsächlich auf den ausgeführten Code geblickt haben, wenden wir uns nun den Benutzern und den Rollen zu, in die Benutzer schlüpfen können. Das .NET Framework bringt eine Abstraktion für Benutzer ins Spiel, denn ein Benutzer kann sowohl eine menschliche Person sein, die gerade am System angemeldet ist, als auch eine imaginäre Entität, in deren Namen eine Aktion im System ausgeführt wird (z.B. das Konto ASPNET für die ASP.NET-Laufzeitumgebung). Dieses abstrakte Konstrukt in .NET heißt *Identity*. Ähnlich verhält es sich mit Rollen, denen ein Benutzer angehören kann. Das Konstrukt für eine Rolle nennt sich *Principal*. Somit stellt Principal den Sicherheitskontext dar, in dem Code ausgeführt wird – vergleichbar mit einer Gruppe in Windows-Systemen (vgl. Abbildung 7.4).

Klassen, welche die Identität eines Benutzers repräsentieren, implementieren die Schnittstelle *IIdentity*. Das Framework stellt eine allgemeine Klasse zur Verfügung, die eine Standardimplementierung dieser Schnittstelle bietet: *GenericIdentity*. Darüber hinaus gibt es noch die Klasse *WindowsIdentity*, die einen Windows-Benutzer darstellt. Zusätzlich kann ein Entwickler – wie im .NET Framework auch an anderen Stellen üblich – seine eigene Implementierung bereitstellen, indem er wie bereits oben erwähnt die Schnittstelle *IIdentity* erfüllt.

Für die Darstellung eines Principals stehen dann entsprechend die Klassen *GenericPrincipal* und *WindowsPrincipal* zur Verfügung.

Die Rollen-basierte Sicherheit, und insbesondere das Principal-Objekt, wird im .NET Framework vor allem bei ASP.NET angewendet. Denn in einer Web-basierten Umgebung ist es von großer Bedeutung, unterschiedliche Benutzer authentifizieren und autorisieren zu können.

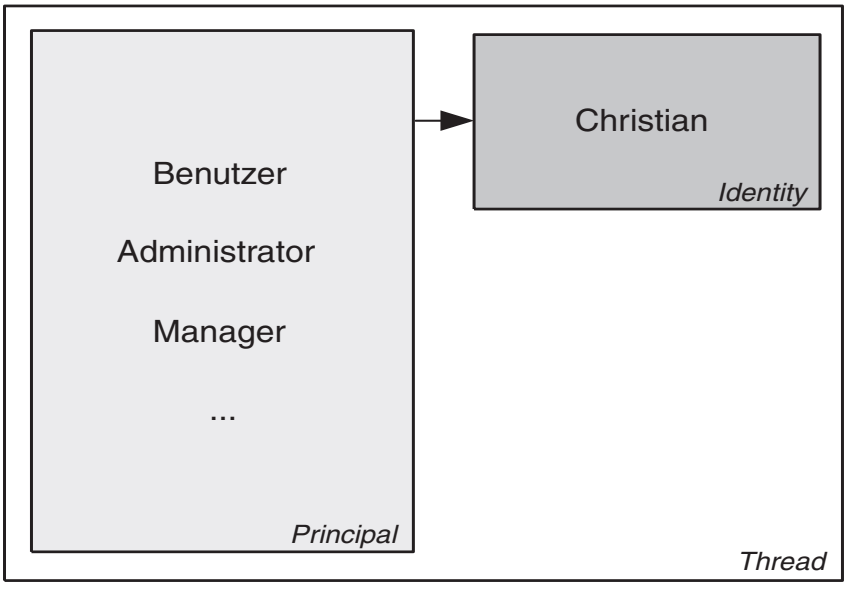


Abbildung 7.4: Struktur der Rollen-basierten Sicherheit

7.3 Web-Sicherheit

Wenn wir eine Anwendung im Internet einer Vielzahl von potenziellen Benutzern bereitstellen, ist dies zunächst unter Sicherheitsaspekten unkritisch. Ein Benutzer surft mehr oder weniger anonym durch das WWW und verwendet die Funktionalität unserer Seite. Interessant wird es zu dem Zeitpunkt, an dem wir uns entschließen, nur noch registrierten Benutzern Zugang zu unserem System zu gewähren. Wir müssen also dafür sorgen, dass sich die Benutzer anmelden und entsprechend ihren Berechtigungen Inhalte und Funktionalität der Seite in Anspruch nehmen können. Zudem sollte die Verbindung der Leitung bei der Übertragung von sensiblen Daten zusätzlich gesichert sein. In diesem Abschnitt werde ich Ihnen einige Techniken an die Hand geben, wie man mit ASP.NET und dem IIS sichere Web-Anwendungen und XML Web Services realisieren kann.

7.3.1 IIS-Sicherheit

Ich habe in Kapitel 3 erklärt, wie die ASP.NET-Architektur aufgebaut ist und funktioniert. An dieser Stelle habe ich auch erwähnt, dass man prinzipiell ASP.NET auch ohne den IIS betreiben könnte. Dies gilt jedoch nicht mehr, wenn man sich ASP.NET im Kontext Sicherheit anschaut. In diesem Fall ist ASP.NET sehr eng mit dem IIS verbunden und kooperiert mit ihm in vielen Bereichen.

Es folgt eine Liste der verfügbaren Schutzmechanismen des IIS:

- Verschlüsselung des Kommunikationskanals durch Unterstützung von *Secure Sockets Layer* (SSL),
- Beschränkung des Zugriffs auf Ressourcen auf Basis von IP-Adressen oder Domänenamen,
- Veröffentlichung von Inhalten über virtuelle Verzeichnisse, damit Benutzer keine Dateien vom Server abziehen können,
- Zuweisung eines Zugriff-Tokens für jede Anfrage. Somit kann das Betriebsystem über Zugriffskontrolllisten (*ACLs*) prüfen, ob der jeweilige Benutzer berechtigt ist, die Ressource zu verwenden. Dieses Token ist abhängig von der Einstellung der Authentifizierungsart.

In diesem Abschnitt werden wir uns hauptsächlich mit dem letzten der vier Punkte auseinander setzen. Denn die Überprüfung mittels ACLs erfolgt vollständig aufgrund der Identität, mit der eine Anfrage ausgeführt wird. Im Falle von ASP.NET Windows-Authentifizierung (siehe weiter unten für Details) arbeitet die ASP.NET-Laufzeitumgebung sehr eng mit dem IIS zusammen, um festzustellen, wer die Anfrage getätigt hat.

IIS bietet dem Administrator verschiedene Möglichkeiten, Benutzer am System zu authentifizieren. Über die Anwendung INTERNET-INFORMATIONSDIENSTE lassen sich diese Mechanismen konfigurieren (vgl. Abbildung 7.5):

- Anonymer Zugriff: Ein Benutzer der IIS-Anwendung benötigt kein eigenes Konto und muss sich nicht anmelden. Stattdessen wird der Systemeigene Account *IUSR_Rechnername* verwendet. Unter diesem Benutzerkonto wird die Anfrage dann ausgeführt. Dieses Konto ist frei wählbar und konfigurierbar.
- Authentifizierter Zugriff mittels *Basic Authentication:* Bei diesem Verfahren muss der Benutzer seinen Benutzernamen und ein Kennwort am Browser eingeben. Dies ist eine Standardfunktionalität eines jeden Browsers und basiert auf einer Erweiterung von HTTP. Die Benutzerdaten werden zwar mit Base64 kodiert, sind aber nicht verschlüsselt und gehen somit als Klartext über die Leitung.
- Authentifizierter Zugriff mittels *Digest Authentication*: Die Art ist ähnlich wie Basic Authentication. Allerdings wird ein MD5 Hashwert der Benutzerdaten übertragen, was das Verfahren sicherer macht. Digest Authentication erfordert den Zugriff auf das Active Directory.
- Authentifizierter Zugriff mittels integrierter Windows-Authentifizierung: Die Login-Informationen des aktuellen Windows-Benutzers werden bei dieser Variante in Form eines Hashwerts an den IIS geschickt. Hierfür muss der Client oder Browser das Windows-Sicherheitsprotokoll *NTLM* unterstützen.
- Authentifizierter Zugriff über Client-seitige SSL-Zertifikate: Hier erfolgt die Authentifizierung über ein SSL-Zertifikat, das auf dem Computer des Benutzers installiert ist.

Dies bedeutet also, dass je nach Authentifizierungsart des IIS die physikalischenVerzeichnisse und Dateien im Windows-Dateisystem mit entsprechende Berechtigungen für die jeweiligen Benutzer ausgestattet sein müssen. Der IIS verwendet dann das Benutzerkonto, um Aktionen im Namen dieses Benutzers auszuführen.

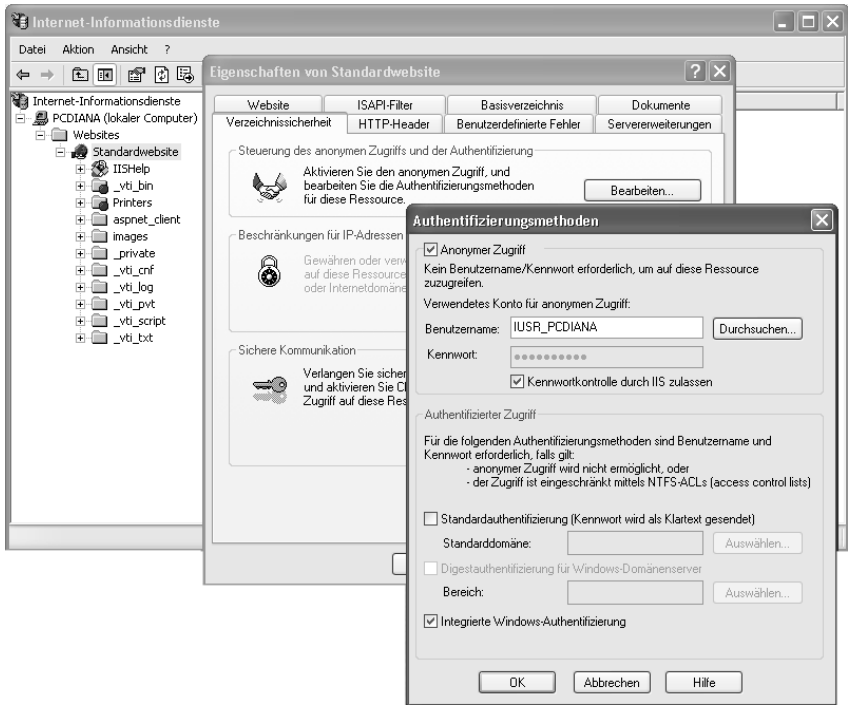


Abbildung 7.5: Authentifizierungseinstellungen im IIS

7.3.2 ASP.NET-Sicherheit

Der Einsatz von ASP.NET im Zusammenhang mit dem IIS wirft nun einige neue Punkte auf. Der IIS (als Prozess *inetinfo.exe*) läuft für gewöhnlich unter dem SYSTEM-Benutzerkonto. Dieses Konto hat sehr viele Rechte. Die ASP.NET-Laufzeitumgebung hingegen wird unter dem Konto ASPNET ausgeführt, welches, wie in Kapitel 3 schon erwähnt, nur geringe Privilegien im System besitzt. Dieses Konto kann auch umkonfiguriert werden. In der Datei *machine.config*, die sich im Installationsordner von .NET befindet, ist unter dem Eintrag `<system.web>` das ausführende Benutzerkonto im Abschnitt `<processModel>` festgehalten:

```
userName="machine" password="AutoGenerate"
```

Der Wert `machine` bedeutet, dass das im System verankerte Konto ASPNET verwendet werden soll. Alternativ kann an dieser Stelle auch der Wert `system` stehen. Dann wird der Prozess mit der Identität von SYSTEM (also wie der

IIS) ausgeführt. In beiden Fällen wird der Wert für das Kennwort automatisch vom System erzeugt. Natürlich kann man auch ein eigenes Konto einrichten und es hier eintragen.

Wenn nun der IIS eine Anfrage an den externen ASP.NET-Prozess *aspnet_wp.exe* weiterreicht, dann geschieht dies mit der Identität des im IIS authentifizierten Benutzers. Allerdings wird standardmäßig diese Anfrage von ASP.NET immer unter dem Konto ASPNET abgearbeitet. Man kann die Laufzeitumgebung jedoch auch dazu veranlassen, die Anfragen im Namen des vom IIS weiter gereichten Benutzers zu bedienen. Dieses Vorgehen heißt *Impersonation* und ist standardmäßig abgeschaltet. Auch dieser Eintrag wird in der Datei *machine.config* unter dem Knoten `<system.web>` vorgenommen. Dies hier ist der vorgegebene Eintrag:

```
<identity impersonate="false" userName="" password="" />
```

Durch den Wert `true` aktiviert man die Impersonation. Man kann auch einen ganz bestimmten Benutzer »impersonieren«, der dann immer hierfür verwendet wird.

Durch diese zusätzliche Architektur, die ASP.NET mit sich bringt, ist es also notwendig, den Zugriff auf Ressourcen über ACLs für den jeweiligen IIS-Benutzer und das Benutzerkonto für ASP.NET frei zu geben.

Autorisierung

Bevor ich auf die unterschiedlichen Authentifizierungsarten in ASP.NET zu sprechen komme, muss ich noch ein paar Worte über die Autorisierungsstrategien verlieren. Ich erläutere die Autorisierung vor der Authentifizierung, da die beiden unmittelbar zusammenhängen und eine sinnvoll konfigurierte ASP.NET zur Authentifizierung ohne Autorisierungsangaben nicht viel Sinn macht.

Außer der bereits erwähnten Kontrolle der Zugriffe für Benutzer über die ACLs, gibt es in ASP.NET die so genannte URL-Autorisierung. Mit der URL-Autorisierung lässt sich mithilfe von Einträgen in eine Konfigurationsdatei regeln, wer Zugriff auf die ASP.NET-Anwendung hat. Diese Einträge können entweder global in der Datei *machine.config* oder pro Anwendung in der Datei *web.config* verzeichnet werden. Das entsprechende Element heißt `<authorization>` und befindet sich innerhalb des Elements `<system.web>`. Das Beispiel in Listing 7.3 zeigt eine mögliche Ausprägung des Elements für die Zugriffskontrolle einer ASP.NET-Anwendung.

Es gibt zwei Tags, die anzeigen, ob Zugriff auf die Ressourcen erteilt (`allow`) oder verweigert (`deny`) werden sollen. Mögliche Attribute dieser Elemente sind `users`, `roles` und `verbs`. Über das Attribut `users` wird ASP.NET angezeigt, welche Benutzer Zugriff erhalten. Es kann ein einzelner Benutzername oder eine durch Kommas getrennte Liste von mehreren Benutzern sein. Ähnlich verhält es sich mit dem Attribut `roles`. Hier wird spezifiziert, welche Rollen Zugriff auf die Ressourcen erhalten. Ein Administrator oder Entwickler kann zusätzlich über das Attribut `verbs` auch noch verbieten, dass

Benutzer über bestimmte HTTP-Methoden auf die ASP.NET-Applikationen zugreifen. So wird im Beispiel verboten, dass alle Benutzer über eine HTTP-GET-Operation zugreifen können.

Die Abarbeitung der Konfigurationseinstellungen erfolgt von oben nach unten. Dies bedeutet, dass im Beispiel der Benutzer `cw` und die Rolle `WebServiceUsers` Zugriff erhalten (im Falle von Windows-Authentifizierung entsprechen diese Angaben einem Windows-Benutzer und einer Windows-Gruppe). Alle anderen Benutzer werden abgelehnt. Allerdings darf keiner der berechtigten Benutzer eine HTTP-GET-Anfrage absetzten.

```
<authorization>
   <allow users="PCJOHANNA\cw"/>
   <allow roles="PCJOHANNA\WebServiceUsers"/>
   <deny verbs="GET" users="*">
   <deny users="*"
</authorization>
```

Listing 7.3: Mögliche Konfiguration der Autorisierung in einer ASP.NET-Anwendung

Es lassen sich auch mithilfe des `<location>`-Tags nur bestimmte Bereiche oder Dateien gegen nicht autorisierten Zugriff schützen. Folgender Eintrag bewirkt, dass nur der Zugriff auf die Datei *AdminService.asmx* einer Autorisierung unterliegt und nur der Benutzer `Administrator` Zugriff erhält:

```
<location path="AdminService.asmx">
   <system.web>
      <authorization>
         <allow users="PCJOHANNA\Administrator"/>
         <deny users="*"/>
      </authorization>
   </system.web>
</location>
```

Authentifizierung

In ASP.NET stehen nun unterschiedliche Varianten zur Verfügung, um Benutzer mit der Laufzeitumgebung zu authentifizieren. Dabei kann ASP.NET auf die vorhandenen Mechanismen des IIS zurückgreifen, muss aber nicht. Folgende Liste zeigt die Möglichkeiten der Authentifizierung:

- Windows: In diesem Fall wird die Authentifizierungsfunktionalität des IIS verwendet.
- Forms: Die Authentifizierung erfolgt über HTML-Formulare mithilfe von HTTP-Cookies. Im Falle von XML Web Services erfolgt eine etwas abgeänderte Abarbeitung.

- Passport: Das öffentlich verfügbare *Passport*-System von Microsoft wird zur Authentifizierung der Benutzer herangezogen.
- Eigene: Ein Programmierer kann auch ein eigenes Authentifizierungsschema implementieren und in die ASP.NET-Umgebung integrieren.

Welche dieser Optionen nun verwendet wird, bestimmt ein Eintrag in der bekannten Datei *machine.config* (alternativ kann auch eine Einstellung für die jeweilige Anwendung in einer Datei *web.config* vorgenommen werden). Unter dem Knoten <system.web> kann man den Eintrag mit folgenden Werten ändern:

```
<authentication mode="[Windows|Forms|Passport|None]">
…
```

Je nach Art der Authentifizierung kann das Element `authentication` noch weitere Unterelemente haben. Es muss hier klargestellt werden, dass die Authentifizierung überhaupt erst dann in Aktion tritt, wenn Ressourcen – sprich eine ASP.NET-Anwendung – gegen Zugriff geschützt wurden, also Autorisierung verlangt wird. Dies bedeutet, dass das `authentication`-Element in den meisten Fällen zusammen mit dem `authorization`-Element verwendet wird, da das Verlangen nach Autorisierung die Authentifizierung anstößt.

Im Folgenden werde ich Ihnen die beiden wichtigsten Optionen `Windows` und `Forms` vorstellen und anhand von Beispielen ihren Einsatz in einer Web-Anwendung zeigen.

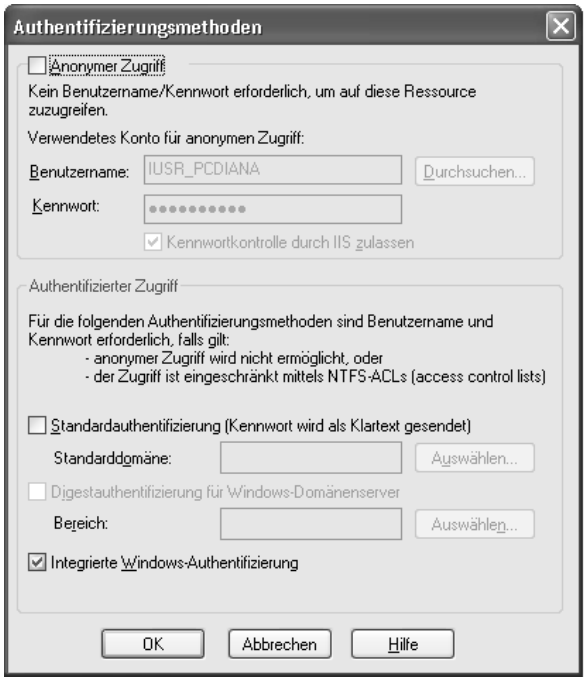


Abbildung 7.6: Aktivieren der integrierten Windows-Authentifizierung

Windows. Bei der Windows-Authentifizierung in ASP.NET ist zunächst der IIS für die korrekte Authentifizierung des Benutzers zuständig. Die Benutzerdaten werden dann bei erfolgreicher Authentifizierung an den ASP.NET-Prozess weitergeleitet. Um die Art der Authentifizierung des IIS festzulegen, müssen die entsprechenden Einstellungen im Internet-Dienste-Manager vorgenommen werden. Die Einstellung in Abbildung 7.6 sorgt dafür, dass nur Benutzer authentifiziert werden, die auch ein Benutzerkonto unter Windows besitzen. Die Vergabe der Zugriffsrechte für diese Benutzer auf die Ressourcen erfolgt dann über die ACLs.

Forms. Die Forms-basierte Authentifizierung ist ein interessantes Merkmal von ASP.NET. Während ein Programmierer mit den klassischen ASP-Login-Mechanismen vollständig neu und zu Fuß entwickeln musste, ist in ASP.NET nun diese auf HTML-Formularen beruhende Möglichkeit vorhanden.

Damit die Forms-basierte Authentifizierung aktiviert wird, muss im `authentication`-Element der Wert für `mode` auf `Forms` gesetzt werden (vgl. Listing 7.4). Das Unterelement `forms` besitzt fünf mögliche Attribute. Die Attribute `protection`, `timeout` und `path` sind dabei optional.

```
<authentication mode="Forms">
   <forms name=".WSBUCH" loginUrl="eYesoft_login.aspx"
      protection="All" timeout="30" path="/">
      <credentials passwordFormat="SHA1">
         <user name="UserName" password="password"/>
      </credentials>
   </forms>
</authentication>
```

Listing 7.4: Konfiguration von Forms-basierter Authentifizierung

Der Wert für `name` ist der Name des Cookies, welches im HTTP-Verkehr verwendet wird. Denn die Forms-basierte Authentifizierung arbeitet ausschließlich mit Cookies, wobei dieses Verhalten nicht geändert werden kann (wie beispielsweise bei Sessions). Das Attribut `loginUrl` zeigt an, auf welche ASP.NET-Seite umgeleitet werden soll, wenn der anfragende Benutzer noch nicht authentifiziert ist.

ASP.NET stellt mit der Forms-basierten Authentifizierung also ein in den größten Teilen automatisiertes Framework bereit. Alles, was ein Programmierer bereitstellen muss, ist die Seite, welche die Anmeldeinformationen entgegennimmt und den Code für die eigentliche Authentifizierung des Benutzers gegen eine Datenquelle. Die Daten können aus einer Datenbank, aus dem Active Directory, aus einer externen XML-Datenquelle oder direkt aus der Konfigurationsdatei stammen. Denn wie in Listing 7.4 gezeigt, kann über das Unterelement `credentials` eine Liste an Benutzern mitsamt Kennwörtern angegeben werden.

Der exakte Ablauf der Authentifizierung ist in Abbildung 7.7 zu sehen. Ein Benutzer fordert eine Seite an, die geschützt ist (*default.aspx*). Da der Benutzer noch nicht authentifiziert ist (er hat die Seite noch nicht besucht und sich angemeldet, daher existiert noch kein Cookie), wird er an die in der Konfigurationsdatei angegeben Login-Seite (*eYesoft_login.aspx*) umgeleitet. Auf dieser Seite werden die erforderlichen Anmeldedaten über ein HTML-Formular gesammelt und über eine HTTP-POST-Anweisung an die *.aspx*-Seite geschickt. In der Logik zur tatsächlichen Ermittlung, ob die Daten mit den Daten eines registrierten Benutzers übereinstimmen, wird nach erfolgreicher Authentifizierung über eine spezielle Methode *RedirectFromLoginPage* der Klasse *FormsAuthentication* das Authentifizierungs-Cookie mit einem verschlüsselten Sitzungs-Ticket gesetzt. Schließlich wird der Benutzer an die ursprünglich angeforderte Seite weitergeleitet. Somit ist der Benutzer für diese ASP.NET-Anwendung authentifiziert und das Cookie ist so lange gültig, wie im Wert für das Attribut `timeout` in Minuten angegeben wurde.

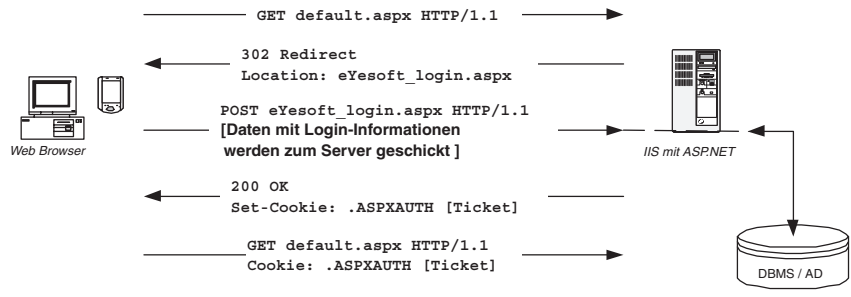


Abbildung 7.7: Forms-Authentifizierung in ASP.NET

7.4 Web Services-Sicherheit

Die bisher vorgestellten Konzepte und Vorgehensweisen zur Absicherung von ASP.NET-Anwendungen waren hauptsächlich auf WebForms-Applikationen (*.aspx*-Seiten) bezogen. Da XML Web Services aber auch innerhalb von ASP.NET ausgeführt werden, gelten die meisten Prinzipien auch für Web Service-Anwendungen. In diesem Abschnitt werde ich Ihnen vorstellen, wie man Web Services in ASP.NET sichern kann und zeige dabei auch, was man nicht von *.aspx*-Seiten übernehmen kann. Schließlich werfen wir noch einen Blick auf die Client-Anwendung, die unseren mit Sicherheitseigenschaften versehenen Web Service benutzen wollen.

7.4.1 Windows-Authentifizierung mit Web Services

Man kann die in ASP.NET eingebauten Authentifizierungsfunktionalitäten auch für XML Web Services einsetzen. Vor allem in Intranet-Umgebungen ist die Variante mit der Windows-Authentifizierung sehr interessant. Für den hier verwendeten Web Service werfen wir einen Blick auf die integrierte Windows-Authentifizierung. Bitten beachten Sie, dass das virtuelle Verzeichnis hierfür die korrekten Einstellungen haben muss.

Web Service

Der Web Service in Listing 7.5 ist nicht nur für Windows-Authentifizierung geeignet. Da die ASP.NET-Authentifizierung automatisch im Hintergrund abläuft, wird lediglich über den Eintrag in der Konfigurationsdatei bestimmt, wie sich das System verhalten soll. Die Methode `PrintUserInfo` gibt den Benutzernamen und die verwendete Authentifizierungsart zurück. Wenn die Anwendung im IIS entsprechend konfiguriert ist, dann erscheint für dieses Beispiel (siehe Listing 7.6) der Wert des gerade angemeldeten Windows-Benutzers und `NTLM` für die Art der Authentifizierung.

Auf einigen Systemen kann es vorkommen, dass anstatt `NTLM` der Wert `Negotiate` für die Art der Authentifizierung angegeben wird. Negotiate erlaubt einer Anwendung die Verwendung von weiteren, unter Umständen besser geeigneten Protokollen falls sie auf dem System vorhanden sind. Zur Zeit unterstützt Negotiate NTLM und Kerberos.

```
<%@ WebService Language="C#" Class="WinAuth" %>
using System.Web.Services;

public class WinAuth
{
   [WebMethod]
   public string PrintUserInfo()
   {
      return "Aktueller Benutzer: " +
         Context.User.Identity.Name +
         ". Verwendete Authentifizierungsart: '" +
         Context.User.Identity.AuthenticationType  +
         "'.";
   }
}
```

Listing 7.5: Windows-Authentifizierung mit XML Web Services

```
<configuration>
   <system.web>
      <authentication mode="Windows" />
      <authorization>
         <deny users="?" />
      </authorization>
   </system.web>
</configuration>
```

Listing 7.6: web-config-Datei für Windows-basierte Authentifizierung in XML Web Services

Client

Die hier vorgestellte Client-Anwendung ist eine ASP.NET-Seite, die sich über einen mittels *wsdl.exe* erzeugten Web Service-Proxy mit dem Web Service verbindet. Damit der Proxy die Benutzerdaten auch an den Web Service übermittelt, muss die Eigenschaft *Credentials* des Proxy-Objekts gesetzt werden. Dies kann zum einen durch Zuweisung von *System.Net.CredentialCache.DefaultCredentials* passieren. Dann werden die Benutzerinformationen des gerade angemeldeten Benutzers verwendet und übertragen. Zum anderen kann aber auch explizit ein anderer Benutzer angemeldet werden, indem ein neues Objekt vom Typ *NetworkCredential* erzeugt und zugewiesen wird.

Als Ergebnis erhalten wir hier den Benutzernamen ASPNET als aktuellen Benutzer, denn die *.aspx*-Seite wird unter diesem Konto ausgeführt und sie ist es, die den Web Service aufruft.

```
<script language="C#" runat="server">
public void Page_Load(Object sender, EventArgs e)
{
   WinAuth oWS = new WinAuth();
   //oWS.Credentials = new
      NetworkCredential("CWeyer", "pw");
   oWS.Credentials = System.Net.CredentialCache.
      DefaultCredentials;
   lblResult.Text = oWS.PrintUserInfo();
}
</script>
```

Listing 7.7: Relevanter Code eines Clients für Windows-basierte Authentifizierung mit XML Web Services

7.4.2 Forms-Authentifizierung mit Web Services

Im Gegensatz zur Windows-Authentifizierung, lässt sich die Forms-basierte Authentifizierung nicht so ohne weiteres mit Web Services einsetzen.

Web Service

Wie wir weiter oben gesehen haben, schaut ASP.NET auf ein gültiges Authentifizierungs-Cookie. Wenn dieses nicht vorhanden ist, wird auf die Login-Seite umgeleitet. Diese Tatsache stellt für eine Web Service-basierte Anwendung natürlich ein Problem dar, denn hier wird eine SOAP-Nachricht erwartet und keine grafische Oberfläche zur Interaktion mit dem menschlichen Benutzer. Somit lässt sich dieser Automatismus für XML Web Services also nicht übernehmen. Es gibt aber dennoch die Möglichkeit, die Cookie-basierte Lösung zu verwenden. Dieser Ansatz ist dann jedoch nicht mehr so komfortabel wie die Konfiguration über einen Eintrag in einer Datei, sondern erfordert die Verwendung eines API-Aufrufs.

Der Beispiel-Web Service in Listing 7.8 zeigt wie man das Forms-Authentication-API verwenden kann. Außerdem wird hier demonstriert, wie man mit ASP.NET und vor allem XML Web Services das Rollen-basierte Sicherheitskonzept von .NET verwenden kann.

Bevor ein Benutzer die Methode `PrintUserInfo` aufrufen kann, muss er sich zunächst über die Methode `Login` anmelden. Innerhalb von `PrintUserInfo` wird nämlich mittels des API-Aufrufs *Context.User.Identity.IsAuthenticated* ermittelt, ob der Client bereits authentifiziert ist. Hier kommt die *Identity*-Eigenschaft des ASP.NET-Kontexts zum Einsatz, die im Falle eines authentifizierten Benutzers sämtliche Informationen über diesen bereithält. Falls der Benutzer nicht bekannt ist, wird einfach eine entsprechende Meldung zurückgeben. In einer Real-World-Anwendung wäre es in diesem Fall angebrachter, eine aussagekräftige Ausnahme zu erzeugen, die dann im Client behandelt werden kann. Falls der Benutzer authentifiziert ist, werden Informationen über seine Identität und die verwendete Authentifizierungsart als Ergebnis zurückgegeben. Allerdings spielt die Authentifizierungsart in unserem Beispiel keine Rolle, da wir nicht Windows-Authentifizierung, sondern Forms-basierte Authentifizierung verwenden. Jedoch muss die IIS-Anwendung anonymen Zugriff auf das virtuelle Verzeichnis erlauben.

```
<%@ WebService Language="C#" Class="FormsAuth" %>
using System.Web.Services;
using System.Web.Security;

public class FormsAuth : WebService
{
   [WebMethod]
   public string PrintUserInfo()
   {
      if (Context.User.Identity.IsAuthenticated == true)
         return "Aktueller Benutzer: " +
         Context.User.Identity.Name +
         ". Verwendete Authentifizierungsart: '" +
         Context.User.Identity.AuthenticationType  +
         "'.";
      else
         return "Sie müssen sich zuerst anmelden!";
         // alternativ: Ausnahme erzeugen
   }

   [WebMethod]
   public bool Login(string sUsername, string sPwd)
   {
      if (sUsername == "CW" && sPwd == "nichtWIRKLICH")
      {
         FormsAuthentication.SetAuthCookie(
            sUsername, true);
```

```
            return true;
        }
        else
            return false;
    }

    [WebMethod]
    public bool Logout()
    {
        if (Context.User.Identity.IsAuthenticated == true)
        {
            FormsAuthentication.SignOut();

            return true;
        }
        else
            return false;
    }
}
```

Listing 7.8: Forms-Authentifizierung mit XML Web Services

In der `Login`-Methode wird der Einfachheit halber schlicht auf einen fest verdrahteten Benutzer überprüft. Wenn der übermittelte Benutzername und das Kennwort übereinstimmen, dann wird über die Methode *FormsAuthentication.SetAuthCookie* das Authentifizierungs-Cookie gesetzt – von nun an ist dieser Benutzer authentifiziert. Analog sorgt die Methode `Logout` dafür, dass das Cookie über den Aufruf von *FormsAuthentication.SignOut* wieder explizit gelöscht wird.

Wichtig ist der Blick auf die korrespondierende *web.config*-Datei (siehe Listing 7.9). Der obere Teil mit der Angabe der Authentifizierungsart kommt einem noch vertraut vor. Zu beachten ist hier, dass der Wert für die Login-Seite keine *.aspx*-Datei, sondern die Web Service-Datei selbst ist. Allerdings ist das Element für die Autorisierung deaktiviert bzw. nicht vorhanden (hier zur Demonstration auskommentiert). Dies ist zwingend notwendig, weil ansonsten die automatische Umleitung einer jeden, nicht authentifizierten Anfrage an die Web Service-Seite zurückgeleitet wird. Die Autorisierung kann in diesem Szenario also nicht auf die einfache Art und Weise eingesetzt werden, sondern müsste von Hand im Code abgehandelt werden.

```
<configuration>
   <system.web>
      <authentication mode="Forms">
         <forms name=".WSDemo"
            loginUrl="FormsAuthBasic/FormsAuthWS.asmx"
            protection="All" timeout="60" path="/" />
      </authentication>
      <!--
```

```
      <authorization>
         <deny users="?" />
      </authorization>
      -->
   </system.web>
</configuration>
```

Listing 7.9: web-config-Datei für Forms-basierte Authentifizierung in XML Web Services

Es sei an dieser Stelle noch einmal darauf hingewiesen, dass die Verwendung von Cookies in Verbindung mit XML Web Services problematisch sein kann, da nicht alle Web Service-Plattformen – und vor allem Clients – Cookies unterstützen. Außerdem ist dieses Verfahren an HTTP als Kommunikationsprotokoll gebunden.

Client

Schauen wir uns einen Client für den oben beschriebenen Web Service an. Bei diesem Client können wir nicht wie bei der Windows-Authentifizierung die *Credentials*-Eigenschaft verwenden. Wir müssen mit expliziten Aufrufen der Web Service-Methoden Login und Logout dafür sorgen, dass das HTTP-Cookie für die Authentifizierung gesetzt und wieder gelöscht wird. Allerdings kann ein Web Service-Proxy nicht standardmäßig mit Cookies umgehen. Hierfür muss man den so genannten Cookie Container mit einer neuen Instanz belegen. Dieser Container ist dann für die Lebenszeit des Proxy-Objekts gültig und speichert die notwendigen Cookie-Daten. Das Ergebnis des in eine *.aspx*-Seite eingebetteten Code-Fragments aus Listing 7.10 ist dann der Benutzer `CW` mit der Authentifizierungsart `Forms`.

```
<script language="C#" runat="server">
public void Page_Load(Object sender, EventArgs e)
{
   FormsAuth oWS = new FormsAuth();
   oWS.CookieContainer =  new CookieContainer();
   oWS.Login("CW", "nichtWIRKLICH");

   lblResult.Text = oWS.PrintUserInfo();
}
</script>
```

Listing 7.10: Relevanter Code eines Clients für Forms-basierte Authentifizierung mit XML Web Services

Da die Benutzerdaten im Klartext in einer SOAP-Nachricht übertragen werden, sollte man wie auch bei der herkömmlichen Forms-Authentifizierung und bei einer Basic Authentication den Kommunikationskanal mit SSL absichern.

Der Einsatz von SSL in XML Web Services ist im Prinzip keine große Magie. Voraussetzung hierfür ist, dass der IIS ein gültiges SSL-Zertifikat für die jeweilige IIS-Website besitzt und entsprechend konfiguriert ist. Der Web Service-Client muss dann lediglich darauf achten, dass der Web Service auch tatsächlich über den richtigen Weg mittels eines HTTPS-URL angesprochen wird. Eine genaue Beschreibung dieses Vorgangs findet man unter *http://support.microsoft.com/default.aspx?scid=KB;EN-US;q307267&ID=KB;EN-US;q307267*.

7.5 Zusammenfassung

XML Web Services sind eine neue und nützliche Technologie, um über Unternehmens- und Firewall-Grenzen hinweg Applikationen miteinander kommunizieren zu lassen. Allerdings bedeuten Web Services und vor allem die Basis SOAP ein großes Sicherheitsrisiko. Somit kommt zu den ohnehin schon wichtigen Punkten der Authentifizierung und Autorisierung sowie der Verschlüsselung des Kommunikationskanals bei herkömmlichen Web-Anwendungen auch noch das Themengebiet der Absicherung von SOAP-basierten Web Services auf den Entwickler und Administrator zu.

Die .NET-Umgebung hat hier vor allem im Bereich ASP.NET einiges zu bieten. Die Sicherheitskonzepte können zum einen auf denen des IIS aufbauen oder können andererseits komplett in Eigenregie implementiert werden. Zusätzlich bringt das .NET Framework eine zusätzliche Ebene an Sicherheit mit der Code-Zugriffsicherheit ins Spiel. Der Fokus liegt hierbei auf Seiten des ausführenden Codes anstatt auf dem ausführenden Benutzer oder einer seiner Rollen – mit diesem Konzept wird somit der sich verändernden Situation im Internet mit immer mehr mobilem Code Rechnung getragen.

8 Ob nah, ob fern

Verteilte Anwendungen mit .NET Remoting

8.1 Einführung

Am Ende des Buchs möchte ich Ihnen noch eine sehr interessante und mächtige neue Technologie im Rahmen von .NET vorstellen. Es handelt sich um *.NET Remoting*. Wenn man Remoting in einem Satz beschreiben müsste, könnte dies folgendermaßen aussehen: Remoting ist eine anpassbare und erweiterbare Architektur und Infrastruktur für verteilte Objektkommunikation im Intranet und im Internet. Diese Beschreibung erinnert in ihren Grundzügen an die von DCOM. Aber eben nur im Grundgedanken, denn DCOM ist nicht erweiterbar und nur mit Abstrichen anpassbar. Allerdings gibt es ein paar unscharfe Aussagen in der Definition. Was beispielsweise soll verteilt genau heißen? Was ist mit erweiterbar gemeint? Und warum überhaupt Remoting? Diese und noch weiterführende Fragen werde ich in diesem Kapitel beantworten und Ihnen die Remoting-Technologie vorstellen.

8.1.1 Warum etwas Neues?

Im Windows-Umfeld ist *Distributed COM* (DCOM) als das am weitesten verbreitete Protokoll und als Architektur zur Programmierung und zum Betrieb von verteilten Anwendungen im Einsatz. DCOM wird oft auch als COM mit einem langen Kabel bezeichnet. Diese Umschreibung soll auf höchster Ebene verdeutlichen, dass prinzipiell COM hinter DCOM steht und das Programmiermodell sich in den wichtigen Grundzügen nicht unterscheidet. Natürlich steckt der Teufel im Detail. So ist z.B. die Kommunikation über das von DCOM verwendetet DCE-RPC über Port 135 nur sehr schwierig mit Firmen-Firewalls zu vereinbaren. Zwar legte Microsoft mit den so genannten *COM Internet Services* (*CIS*) eine auf HTTP basierende Variante nach. Allerdings ist diese Lösung nicht vollständig in die DCOM-Umgebung eingearbeitet und stellt bei genauerem Hinsehen auch eine Mogelpackung dar: Nur der erste, initiale Kontakt mit dem Server-Objekt wird über HTTP-Port 80 abgewickelt. Ein weiteres Problem bei der Entwicklung mit DCOM sind Sicherheitshürden. Da DCOM sehr eng mit dem Konzept der Windows-Domänen, -Gruppen und -Benutzer verbunden ist, stellt sich eine funktionierende Konfiguration der Berechtigungen bei der Verteilung bzw. Installation einer Anwendung oftmals als schier unüberwindbares Problem dar. Diese beiden Probleme sind nur ein Ausschnitt der Unzulänglichkeiten DCOMs, die sich in den Jahren seit seiner

Geburt 1996 heraus kristallisiert haben. Der wahrscheinlich größte Nachteil aber ist, dass DCOM – ebenso wie COM als Komponentenmodell – niemals mit dem Gedanken des Internet im Kopf entworfen und entwickelt wurde. Die Konzeption von DCOM erfolgte ausschließlich für eine abgeschlossene Windows-Landschaft. Und genau hier gilt es, mit den Kriterien für eine neue Architektur anzusetzen.

8.1.2 Neue Anforderungen

Die Idee und die Implementierung von .NET besteht ja nicht erst seitdem die Öffentlichkeit im Sommer 2000 auf .NET (damals noch *NGWS*) aufmerksam wurde. Mit dem Konzept der XML-basierten Web Services trägt Microsoft hierbei der allgemeinen Entwicklung hin zu plattformübergreifenden und integrierenden Lösungen Rechnung. Über HTTP werden XML-Datenpakete zwischen den Kommunikationspartnern ausgetauscht. XML Web Services sind aber nicht immer die beste und die geeignetste Lösung für alle Probleme. Was im Mosaik noch fehlt, ist eine Technologie vergleichbar mit DCOM, allerdings auf Stand der neuesten Anforderungen hinsichtlich verteilter Kommunikation im Internet und mit einem erweiterbaren Grundkonzept.

Damit wir einen kompakten Überblick über notwendige Eigenschaften dieser neuen Technologie erhalten, stellen wir im Folgenden eine Wunschliste für die Fähigkeiten von .NET Remoting auf:

- Unterstützung mehrerer Protokolle,
- Unterstützung mehrerer Datenformate,
- Verwaltung der Lebenszeit eines Objekts durch den Client oder Server,
- Übergabe der Objektdaten per Referenz oder in Kopie,
- Möglichkeit, Ereignisse und Rückrufe implementieren,
- Allgemeine Erweiterbarkeit gemäß dem Kontextprinzip (aufbauend auf dem Kontextprinzip aus COM+),
- Interoperabilität mit anderen Plattformen.

Und da diese Liste nicht exklusiv aus meiner Feder stammt, sondern von vielen Personen auf der ganzen Welt in ähnlicher Weise formuliert wird, sind diese Punkte auch alle in .NET Remoting vorhanden. Remoting bietet darüber hinaus sogar noch viel mehr, allerdings ist es auch sehr komplex, weshalb ich in diesem Kapitel nur auf die wichtigsten Eigenschaften und Fakten eingehen möchte. Diese Informationen reichen aus, um auf Remoting aufsetzende, verteilte Applikationen mit .NET zu erstellen.

8.2 Erweiterte Grundlagen

Die notwendigen Grundlagen von .NET wurden in Kapitel 2 vorgestellt. In diesem Kapitel über Remoting müssen nun einige Konzepte aufgefrischt bzw. vertieft und erweitert werden. Der Grund liegt darin, dass Remoting sehr tief in .NET und in der CLR verankert ist und sich auf einige wenige, aber durchaus wichtige Tatsachen stützt.

8.2.1 Applications Domains

Bei den Grundlagen zu .NET in Kapitel 2 habe ich das Konzept der *Application Domains* (Anwendungsdomänen) nicht erwähnt. An dieser Stelle werde ich dies nun nachholen, da Application Domains die Basis für die Ausführung einer .NET-Applikation im Speicher darstellen. Wie in Kapitel 3 über die ASP.NET-Laufzeitumgebung bereits erwähnt, erfolgt vor allem die Trennung von Daten innerhalb eines Prozesses bei .NET durch Application Domains. Ich werde Ihnen in den folgenden Zeilen nahe bringen, was Application Domains sind und warum man sie braucht.

Vereinfacht gesagt, kann man Application Domains als Mini-Prozesse innerhalb eines Prozesses ansehen. Der eigentliche Prozess ist ein dem Betriebssystem zugehöriges Konstrukt, wohingegen eine Application Domain (*AppDomain*) ein .NET-Konstrukt ist. An dieser Stelle sehen wir auch schon den ersten Grund, warum AppDomains überhaupt eingeführt wurden. Sie sollen eine Abstraktion vom physikalischen Prozess darstellen, denn die CLR und die .NET-Umgebung kann und soll ja auch auf unterschiedliche Prozessor- und Betriebssystemversionen abgebildet werden. Bei diesem Vorhaben ist eine Implementierung der logisch aufgehängten AppDomains einfacher zu realisieren, als sich auf die plattformspezifischen Prozesse zu konzentrieren. Abbildung 8.1 zeigt das prinzipielle Bild, das sich bei der Verwendung von AppDomains in Bezug auf die physikalischen Prozesse ergibt.

Weil AppDomains im Grunde genommen die physikalischen Prozesse in der .NET-Welt außen vor lassen sollen, gelten die meisten Vorzüge – aber auch Nachteile – von Prozessen auch für die AppDomains. Ein Programmierer verwendet AppDomains, um Anwendungen voreinander zu schützen. Durch den Einsatz von AppDomains erreicht er somit eine Isolation der Anwendungen, sie können sich gegenseitig nicht direkt beeinflussen. Wenn eine Anwendung in AppDomain 1 abstürzt, dann ist AppDomain 2 davon nicht betroffen. Dies bedeutet auch, dass eine Anwendung in AppDomain 1 nicht direkt auf eine Anwendung in AppDomain 2 zugreifen kann. Ein kleines Beispiel soll diese Tatsache verdeutlichen. Die einfache Anwendung aus Listing 8.1 wird standardmäßig in einer eigenen AppDomain gestartet. Wenn man die Anwendung nach der Kompilierung in der Kommandozeile startet, wird eine eigene AppDomain angelegt. Die Anwendung in Listing 8.2 versucht nun, die Anwendung aus Listing 8.1 per Code zu starten. Dies geschieht über eine Instanz von *AppDomain* (im Namensraum *System*). Die Anwendung wird in die neu erstellte AppDomain geladen und versucht,

eine Instanz des Typs `ClassOne` aus der Anwendung aus Listing 8.1 zu erstellen. Dies scheitert allerdings mit einer Programmausnahme. Diese Ausnahme sagt, dass die Klasse nicht mit einem speziellen Attribut versehen ist. Was dies genau auf sich hat, werden wir weiter unten sehen.

Der Zugriff aus einer AppDomain auf eine andere AppDomain ist also per se nicht erlaubt. Allerdings gibt es Mechanismen, um dies zu bewerkstelligen.

```
using System;

class ClassOne
{
   static void Main(string[] args)
   {
      Console.WriteLine("Anwendung in AppDomain1
      gestartet!");
   }
}
```

Listing 8.1: Anwendung in AppDomain 1

```
using System;
using System.Reflection;
using System.Runtime.Remoting;

class ClassTwo
{
   static void Main(string[] args)
   {
      AppDomain domain = AppDomain.CurrentDomain;
      AppDomain newDomain =
         AppDomain.CreateDomain("WSBuchDomain");
      newDomain.ExecuteAssembly("appdomains1.exe",
         null,args);
      ObjectHandle o =
         newDomain.CreateInstance("appdomains1",
         "ClassOne");
      o.Unwrap();
   }
}
```

Listing 8.2: Anwendung in AppDomain 2, welche auf Typ aus AppDomain 1 zugreifen möchte

Objekte innerhalb einer AppDomain können direkt aufeinander zugreifen. Wie jedoch oben gesehen funktioniert dies nicht, wenn das Zielobjekt in einer anderen AppDomain liegt. Da AppDomains regelrechte Grenzen einer Anwendung definieren, müssen diese Grenzen mit einem Hilfsmittel überschritten werden. Dieses Hilfsmittel heißt *Marshaling*. Marshaling ist der Vorgang, bei dem im Rahmen eines Zugriffs auf eine AppDomain- oder pro-

zessfremde Ressource die zu übermittelnden Daten so verpackt werden, dass sie über diese Grenze verschickt werden können. Auf der Gegenseite werden die Daten dann entsprechend ausgepackt. Das gleiche Spiel wiederholt sich dann für etwaige Rückgabewerte (vgl. Abbildung 8.1). Beim Vorgang des Marshalings werden die zu übertragenden Daten in eine Nachricht umgewandelt. Diese Nachricht ist eine Instanz eines Typs, der die Schnittstelle *IMessage* implementiert. Ein Programmierer muss hier aber nicht selbst Hand anlegen. Das .NET Framework hält mit der Remoting-Technologie ein sehr mächtiges Werkzeug bereit, welches die meisten Marshaling-Aufgaben automatisch übernimmt. Immer dann, wenn Marshaling notwendig ist, tritt .NET CLR Remoting in Aktion.

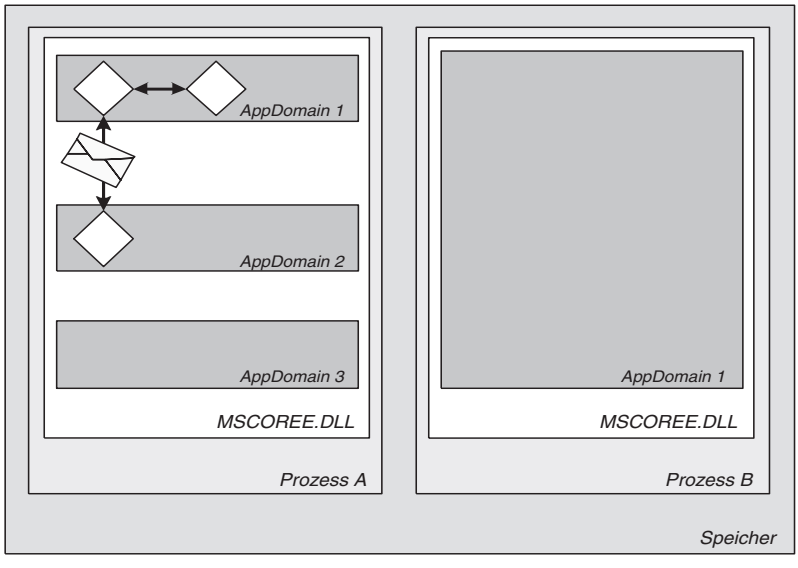


Abbildung 8.1: Application Domains und Prozesse

Marshaling ist keine Neuerfindung von .NET. Dieses Konzept existiert schon seit vielen Jahren und ist immer dann gefragt, wenn es um die Überschreitung von Prozessgrenzen geht – oder um die Überschreitung von Grenzen abgeschlossener Konstrukte, wo eine Verpackung und Bearbeitung der Aufrufdaten erfolgen muss. Ein Beispiel hierfür sind die Apartments in COM/DCOM.

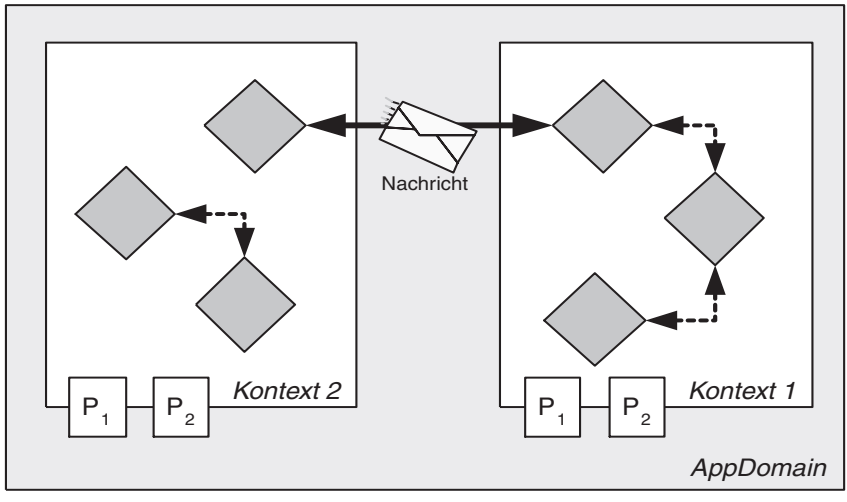
8.2.2 Kontext-basierte Programmierung

Ein weiteres, wichtiges Konzept in .NET ist der Kontext bzw. die Kontext-basierte Programmierung. Neben der Benutzung von AppDomains kann eine solche AppDomain in mehrere Kontexte unterteilt werden. Das Prinzip des Kontexts wurde dem Windows-Programmierer zuerst mit dem MTS und mit COM+ vorgestellt. Dort war der Kontext oder das Kontextobjekt dafür zuständig, einem Objekt innerhalb der COM+-Laufzeitumgebung

durch einfache Konfiguration von Eigenschaften zusätzliche Dienste zur Laufzeit anzubieten. Beispiele für solche Dienste sind automatische, verteilte Transaktionen oder Objekt-Pooling. In .NET wurde die Grundidee des Kontexts aus COM+ aufgenommen, weiter verfeinert und mit in die CLR eingebaut. Man darf aber den COM+-Kontext nicht mit dem .NET CLR-Kontext verwechseln oder gar gleich setzen.

Die primäre Aufgabe eines Kontexts ist es, funktional zusammengehörige Objekte zu gruppieren. Ein Kontext besitzt dabei eine Menge von Eigenschaften (beispielsweise P1 und P2), die eine Umgebung für die darin befindlichen Objekte definieren. Kontexte werden während der Aktivierung für Objekte erstellt und ein Kontext kann mehrere Objekte beinhalten (siehe Abbildung 8.2).

Abbildung 8.2: Kontext-Konzept in .NET



Die Kommunikation innerhalb einer AppDomain erfolgt – wie oben beschrieben - direkt und ohne Umwege. Über AppDomain-Grenzen hinweg erfolgt Marshaling. Marshaling ist aber auch dann notwendig, wenn ein Objekt aus Kontext 1 auf ein Objekt aus Kontext 2 zugreifen möchte – selbst wenn diese Objekte innerhalb der gleichen AppDomain existieren. Auch hier wird über Nachrichten kommuniziert, die von der .NET CLR Remoting-Infrastruktur in den meisten Fällen automatisch erstellt und verarbeitet werden.

8.2.3 Marshaling

Es gibt zwei Formen des Marshalings: Übergabe per Wert als Kopie oder Übergabe per Referenz. Die erste Variante wird im englischen als *Marshal By Value* (*MBV*) bezeichnet, während die zweite Ausprägung als *Marshal By Reference* (*MBR*) bekannt ist.

MBV ist die Standardeinstellung beim Marshaling. MBV bedeutet, dass bei der Übergabe eines Parameters an eine Methode eines Objekts in einer fremden AppDomain oder beim Erhalt eines solchen Rückgabewerts immer eine Kopie der Daten erstellt und verschickt wird. Damit eine Klasse per MBV verschickt werden kann, muss sie mit dem Attribut *Serializable* versehen werden. Somit lässt sich unsere Demoanwendung aus Abschnitt 8.2.1 leicht zum Laufen bringen, indem wir die Klasse `ClassOne` durch Auszeichnung mit dem entsprechenden Attribut MBV-fähig machen. Mit diesem Attribut wird der CLR und der Remoting-Infrastruktur angezeigt, dass diese Klasse serialisiert werden soll und eine Kopie der Instanz verschickt werden kann. Die Daten werden einfach zwischen den AppDomains im Speicher kopiert. Das Ergebnis gleicht dann dem Code in Listing 8.3.

```
using System;

[Serializable]
class ClassOne
{
   static void Main(string[] args)
   {
      Console.WriteLine("Anwendung in AppDomain1
      gestartet!");
   }
}
```

Listing 8.3: Serialisierbare Klasse für MBV

Will man als Programmierer selbst genau bestimmen, welche Daten einer Klasse serialisiert werden sollen, dann kann diese Klasse die Schnittstelle *ISerializable* implementieren.

MBR ist die anspruchsvollere Variante des Marshalings (zumindest aus Sicht der CLR). Da bei MBR eine Referenz zwischen den AppDomains ausgetauscht wird, muss dafür gesorgt werden, dass die entsprechenden Änderungen an den Daten des Objekts auch immer wieder zurück transferiert werden. Bei MBR tritt ein Proxy-Objekt in Aktion, dessen Aktivität in Abschnitt 8.3.1 etwas genauer beleuchtet wird. MBR ist das typische Szenario für Remoting, da es in den meisten Fällen wesentlich besser geeignet ist als MBV. Wenn beispielsweise die Größe eines Objekts eine kritische Grenze überschreitet oder einfach aus Geschwindigkeitsgründen eine Erstellung einer Kopie nicht in Frage kommt, dann wird MBR eingesetzt. Wir werden weiter unten in Abschnitt 8.4.1 noch die diversen Aktivierungsmodelle sehen, die uns die Remoting-Infrastruktur zur Erstellung und zur Arbeit mit entfernten Objekten anbietet.

In diesem Zusammenhang möchte ich auch auf die Objektverweise hinweisen. Objektverweise (Instanzen der Klasse *ObjRef*) spielen eine wichtige Rolle bei Remoting und Marshaling. Sie enthalten Metadaten, die den Typ des Objekts beschreiben, das »gemarshalt« wird (einschließlich der Klassen-

hierarchie). Außerdem enthalten sie einen eindeutigen Bezeichner für das betreffende Objekt, den so genannten URI, und Informationen zum Erreichen der AppDomain, in der sich das Objekt befindet. Objektverweise werden von einer AppDomain in eine andere übergeben, um ein MBR-Objekt in der AppDomain der Client-Anwendung darzustellen.

8.3 Remoting-Architektur

Die Remoting-Infrastruktur wird in .NET dann in Anspruch genommen, wenn es sich bei einem Objektzugriff nicht um einen lokalen, sondern einen entfernten Zugriff handelt. Wie wir in den beiden vorherigen Abschnitten gesehen haben, gibt es bestimmte Regeln, ob ein Objekt nun lokal oder entfernt ist. Diese Regeln werde ich hier noch mal zusammenfassen, da sie essentiell für das Verständnis von Remoting sind.

Alle Objekte innerhalb einer AppDomain sind lokal. Alle anderen Objekte sind entfernt – auch jene Objekte, die im gleichen Betriebssystem-Prozess liegen. Zusätzlich sind kontextgebundene Objekte lokal, wenn sie im gleichen Kontext liegen, ansonsten sind auch sie entfernt. Für all diese Fälle gilt nun, dass bei einem lokalen Zugriff, der Zugriff direkt ohne Hilfsmittel stattfindet. Bei einem entfernten Aufruf tritt immer Marshaling in Aktion. Und Marshaling wird von der Remoting-Funktionalität in der CLR übernommen. Dabei spielt der *Proxy* eine zentrale Rolle.

8.3.1 Proxy und Dispatcher

Der Proxy ist dafür zuständig, auf der Client-Seite wie das eigentliche Remoting-Objekt zu agieren. Er hat prinzipiell die gleiche Ausprägung wie das eigentliche Objekt, besitzt aber keine Implementierung der Methoden, sondern kümmert sich um das Marshaling, damit die Daten zum entfernten Objekt gelangen (und wieder zurück). Die weitere Kommunikation bis hin zum Remoting-Objekt durchläuft dann mehrere Schichten.

Abbildung 8.3 zeigt die allgemeine Architektur von .NET Remoting. Eine Client-Anwendung kommuniziert mit einem Proxy-Objekt, das als lokaler Stellvertreter für das entfernte Objekt gilt. Über so genannte *Formatter* und *Channels* werden die zu übermittelnden Daten zum Objekt transportiert. Auf der Server-Seite (in diesem Fall bedeutet Server das Server-seitige Objekt in einer anderen AppDomain oder einem anderen Kontext) hat der Proxy seine Entsprechung im *Dispatcher*-Objekt, welches dann zuständig zeichnet, die Daten in adäquater Form an das Objekt weiter zu reichen. In den folgenden Abschnitten werde ich die einzelnen Bestandteile näher beschreiben und Beispiele geben, wie man damit arbeitet.

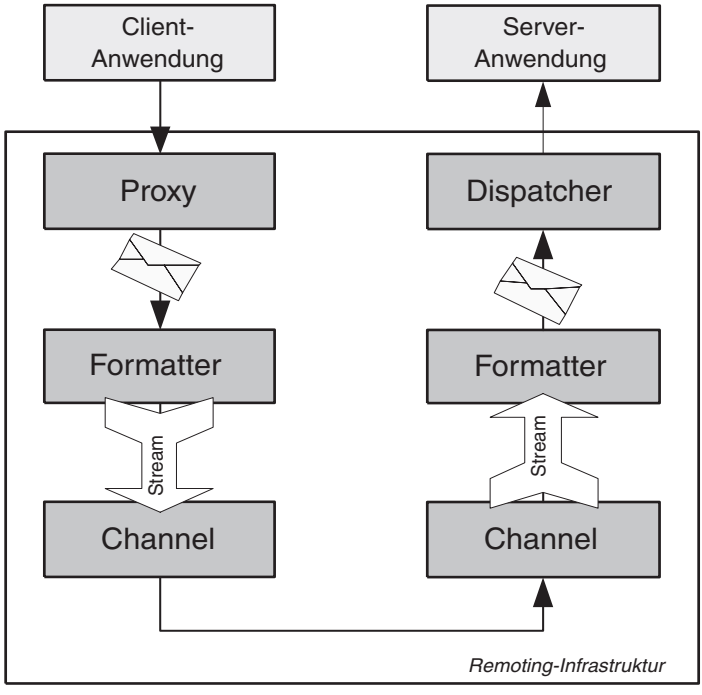


Abbildung 8.3: Überblick über die Remoting-Architektur

Wenn man sich den Proxy einmal genauer anschaut, dann fällt auf, dass die interne Implementierung des Proxy aus zwei Proxies besteht: Der *Transparent Proxy* und der *Real Proxy*. Was für die meisten Client-Anwendungen wie ein Proxy aussieht, ist eigentlich eine Instanz der *TransparentProxy*-Klasse. Der Transparent Proxy verlässt sich auf ein Hilfsobjekt vom Typ *RealProxy*, um den Großteil der Arbeit und der Verwaltungsaufgaben zu erledigen. Der Transparent Proxy erledigt nur die Vorarbeit des Abfangens von Aufrufen, wenn ein Client eine Methode für ihn aufruft. Der Real Proxy leitet die erstellte Nachricht über den konfigurierten Channel weiter (wie die Nachricht tatsächlich erzeugt wird, ist von zwei Anwendungsfällen anhängig und interessiert an dieser Stelle nicht). Innerhalb der Architektur kann nur der Real Proxy erweitert werden. Der Transparent Proxy hingegen ist ein internes Konstrukt, das nicht ersetzt oder erweitert werden kann.

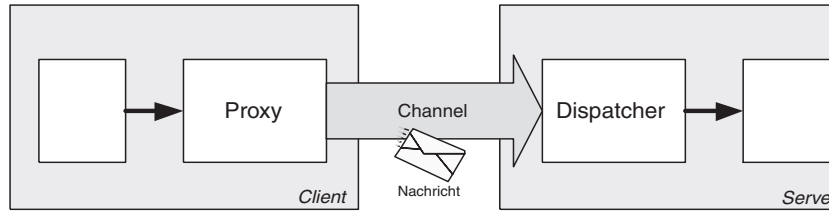
Innerhalb einer AppDomain wird von der Remoting-Laufzeitumgebung dafür gesorgt, dass entsprechende Optimierungen bzgl. der Effizienz vorgenommen werden.

8.3.2 Channels

Channels (Kanäle) in .NET sind dafür verantwortlich, dass Nachrichten zwischen zwei Endpunkten transportiert werden (siehe Abbildung 8.4). Dabei ist ein Channel immer an eine bestimmte Protokollimplementierung gebunden. In der Version 1.0 von .NET sind Channels für die Protokolle HTTP und TCP verfügbar (im Namensraum *System.Runtime.Remoting.Channels*). Jedoch lassen sich aufgrund der sehr offenen und gut erweiterbaren Architektur von Remoting leicht eigene Channels z. B. für FTP, SMTP oder Named Pipes implementieren. Diese eigenen Channels können dann in die Infrastruktur durch Einträge in Konfigurationsdateien eingebunden werden.

Ein Channel verschickt aber eigentlich keine Nachricht im Sinne von .NET. Eine Nachricht ist eine Ausprägung von *IMessage*, allerdings wird über den Channel ein Stream-Objekt versendet, welches dann auf der gegenüber liegenden Seite wieder in ein Nachrichtenobjekt umgewandelt wird.

Abbildung 8.4: Kommunikation über Channels



Ein Channel kann Nachrichten empfangen oder sie versenden. Die meisten Channels bieten jedoch beide Funktionalitäten an. Hierfür müssen die Schnittstellen *IChannelReceiver* und *IChannelSender* implementiert werden. Die eingebauten Channels *TcpChannel* und *HttpChannel* implementieren beide Schnittstellen.

Damit in einer Remoting-Umgebung ein Objekt von einem entfernten Client überhaupt angesprochen und benutzt werden kann, muss es einen Prozess geben, der dieses Objekt beherbergt – den so genannten Host (mehr Informationen können sie in Abschnitt 8.5 nachlesen). Im Falle von XML Web Services mit ASP.NET ist dies die ASP.NET-Laufzeitumgebung. Für Remoting kann man beispielsweise eine normale Kommandozeilenanwendung verwenden. Der Code in Listing 8.4 zeigt, wie ein Host die Kommunikationskanäle (Channels) für TCP und HTTP seinen interessierten Clients über die Ports 8085 und 8086 anbietet. Durch diese wenigen Zeilen ermöglicht ein Programmierer, dass seine Objekte entweder über TCP oder HTTP angesprochen werden können.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Http;
```

```
public class SimpleServer
{
   public static int Main(string [] args)
   {
      TcpChannel chanT = new TcpChannel(8085);
      ChannelServices.RegisterChannel(chanT);
      HttpChannel chanH = new HttpChannel(8086);
      ChannelServices.RegisterChannel(chanH);

      // hier wird dann das eigentliche Objekt erzeugt
      // und der Remoting-Infrastruktur bekannt gemacht
      // (siehe Abschnitt 8.4).

      System.Console.WriteLine("<Enter> beendet ...");
      System.Console.ReadLine();

      return 0;
   }
}
```

Listing 8.4: Bereitstellen von Channels in Remoting-Servern

8.3.3 Formatter

Damit die Nachrichten in ein Format gebracht werden, das über Channels versendbar ist, gibt es in der Remoting-Infrastruktur die *Formatter*. Formatter serialisieren Objekte oder Objektverweise (welche ihrerseits wieder spezielle Objekte sind) in ein spezielles Format. Nach der Umwandlung durch den Formatter liegt ein Datenstrom (*Stream*) vor, der dann zum Kommunikationsendpunkt über den ausgewählten Channel geschickt wird (vgl. Abbildung 8.5).

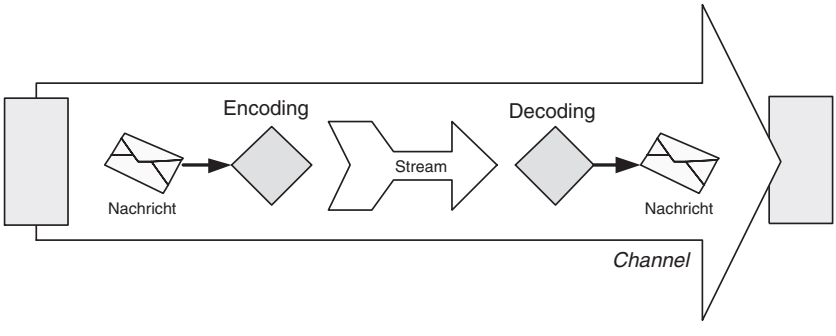


Abbildung 8.5: Formatter wandeln Nachrichten

Formatter-Objekte werden dynamisch von der Channel-Implementierung verwendet. Dies sieht in der Umsetzung so aus, dass ein Channel einen Formatter zur Laufzeit anfordert, um eine Nachricht umzuwandeln. Das Zusammenspiel von Formatter und Channel kann dabei sehr fein konfiguriert werden.

Standardmäßig sind in der Version 1.0 des .NET Frameworks zwei Formatter verfügbar. Der eine Formatter ist in der Klasse *SoapFomatter* (Namensraum *System.Runtime.Serialization.Formatters*) implementiert und serialisiert Nachrichten in SOAP V1.1-konforme XML-Datenpakete. Der zweite Formatter ist in der Klasse *BinaryFormatter* realisiert und bietet eine proprietäre, binäre Darstellung der Daten zum Versenden an. Der SOAP Formatter wird standardmäßig mit dem HTTP Channel verwendet – dies entspricht somit in etwa der Analogie der XML Web Services: SOAP-Datenpakete werden über HTTP verschickt. Der Binary Formatter hingegen wird standardmäßig mit dem TCP Channel eingesetzt. Diese Kombination ist vor allem für die Kommunikation im Intranet gedacht, wo es auf höchstmögliche Geschwindigkeit und kleine Datenpakete ankommt. Allerdings ist die Kombination aus Formatter und Channel beliebig mischbar. Ein einfacher Eintrag in einer Konfigurationsdatei zeigt der Remoting-Laufzeitumgebung an, dass dann z. B. ein SOAP Formatter mit dem TCP Channel in Aktion treten soll. Mehr Informationen und ein Beispiel dazu finden Sie in Abschnitt 8.5.

Fast selbstverständlich ist die Tatsache, dass auch eigene Formatter implementiert werden können. So kann man z. B. einen Formatter für die Übertragung eines eigenen Datenformats oder für die Übertragung der Daten gemäß einem offenen Standard, der noch nicht in einer Implementierung vorhanden ist (z. B. zukünftige Versionen von SOAP), erstellen. Eigene Formatter müssen prinzipiell die Schnittstelle *IRemotingFormatter* implementieren.

Diese gesamte Architektur von .NET Remoting ist mehrschichtig angelegt. Ein Entwickler kann bei Bedarf in jeder Schicht seine eigene Implementierung »einpflanzen« und somit Einfluss auf die diversen Aktionen nehmen. Die Konstrukte, mit denen die Remoting-Infrastruktur erweitert und angepasst werden kann, nennen sich *Sinks*. Es gibt *Message Sinks* (implementieren *IMessageSink*), *Channel Sinks* (implementieren *IClientChannelSink* respektive *IServerChannelSink*). Damit mehrere Sinks hintereinander in Aktion treten können, erlaubt Remoting so genannte *Sink Chains*. Die tatsächliche Verwendung der jeweiligen Sinks werden dann durch Einträge in eine Konfigurationsdatei festgelegt.

8.4 Arbeiten mit entfernten Objekten

Bisher haben wir hauptsächlich Theorie und Konzepte aus der .NET Remoting-Welt gesehen. Abgesehen von dem unvollständigen Code-Beispiel in Listing 8.4 haben Sie noch keinen Quelltext zu Gesicht bekommen. Dies soll sich in diesem und in den folgenden Abschnitten ändern. Wir werden einen Blick auf die Arbeit mit Objekten auf der Server-Seite werfen. In .NET Remoting gibt es prinzipiell zwei grundlegende Modelle, wie Objekte aktiviert werden, wie diese dann von Client-Anwendungen aus angesprochen werden und sich zueinander verhalten.

8.4.1 Aktivierungsmodelle

Zusätzlich zu dem aus der XML Web Services-Umgebung bekannten Aktivierungsmodell existiert bei Remoting in .NET noch ein weiteres Modell. Diese beiden Modelle werden im Folgenden kurz erläutert:

- *Server Activated Objects* (*SAO*): SAOs sind Objekte, die sich ähnlich verhalten wie XML Web Service-Objekte in ASP.NET. Sie werden vom Server (dem Host) aktiviert und wieder zerstört. Es gibt zwar zwei unterschiedliche Varianten, aber es ist immer der Server für die Aktivierung verantwortlich.
- *Client Activated Objects* (*CAO*): CAOs sind Objekte mit echten Objektreferenzen, die der Client hält. Es erhält jeder Client sein eigenes Objekt. Dies entspricht grundlegend dem Aktivierungsmodell vom COM/DCOM.

Im Folgenden werde ich die beiden Modelle genauer vorstellen und mit Beispielen illustrieren.

Server Activated Objects

SAOs sind, wie oben schon erwähnt, Objekte, deren Lebensdauer direkt vom Server, d.h. dem Host-Prozess, gesteuert wird. SAOs werden auch *Well Known Objects* (*WKO*) genannt. Die AppDomain des Servers instanziiert diese Objekte nur dann, wenn eine Client-Anwendung auch tatsächlich einen Methodenaufruf für das Objekt durchführt. Wird vom Client nur die Erzeugung des Objekts durch den Aufruf von *new* oder durch Verwendung von *Activator.GetObject()* angefordert (siehe Abschnitt 8.7), so wird das Objekt nicht durch den Server erstellt. Der Grund für diese Art der Implementierung liegt darin, dass damit für die Erstellung der Instanz keine eigene Verbindung zum Server aufgebaut werden muss. Wenn ein Client eine Instanz eines SAO-Typs anfordert, wird in der AppDomain des Clients nur ein Proxy erstellt.

Zu beachten ist hierbei, dass nur Standardkonstruktoren für SAO-Typen zulässig sind. Um einen Typ zu veröffentlichen, dessen Instanzen durch Konstruktoren mit Argumenten erzeugt werden, kann man das CAO-Modell verwenden oder die betreffende Instanz dynamisch veröffentlichen.

Bei den SAOs gibt es noch eine Unterscheidung zwischen zwei Aktivierungsmodi:

- *Singleton*,
- *SingleCall*

Singleton-Typen verfügen nie über mehrere Instanzen gleichzeitig. Wenn eine Instanz vorhanden ist, werden alle Client-Anfragen über diese Instanz abgewickelt – dieses Vorgehen entspricht somit dem bekannten Entwurfsmuster des Singleton in der objektorientierten Programmierung. Wenn noch keine Instanz vorhanden ist, erstellt der Server eine Instanz. Allerdings ist mit jedem Singleton-Typ auch eine Standardlebensdauer verbunden. Durch diese Einstellung erhalten Clients nicht immer eine Referenz auf die gleiche Instanz – obwohl nie mehr als eine Instanz des Typs auf dem Server existiert. Diese Lebensdauer kann man aber im Code auf unendlich setzen.

Das Beispiel in Listing 8.5 zeigt, wie ein Objekt als SAO Singleton in einem Remoting-Server publiziert wird. Es ist der gleiche Code, wie weiter oben bereits in Abschnitt 8.3.2 in Listing 8.4 gesehen. Nur, dass hier nun der Code für die Aktivierung und Bereitstellung des Objekts mit eingebaut ist. An dieser Stelle möchte ich nochmals darauf hinweisen, dass ein Objekt immer einen Host-Prozess benötigt. In Listing 8.5 sind diese beiden Implementierungen in einer Datei und einem Listing vereint. Normalerweise wird man diese Teile getrennt voneinander programmieren und pflegen.

Die wichtige Zeile im Listing ist jene mit dem Aufruf von *RegisterWellKnownServiceTypes*. Diese statische Methode aus der Klasse *RemotingConfiguration* ist für die Registrierung eines SAO-Typs mit der Remoting-Infrastruktur zuständig. Als Parameter werden der Typ, der Kommunikationsendpunkt und eine Konstante übergeben. Der Typ `HalloServer` wird normalerweise aus der Assembly *HalloServer.dll* geholt – in unserem Fall allerdings nicht, da der Typ ja in der gleichen Assembly implementiert ist. Der Parameter der Methode *GetType* ist eine Zeichenkette, die den Typnamen und den Assembly-Namen repräsentiert. Alternativ kann man in C# auch einfach den *typeof*-Operator einsetzen.

Der Kommunikationsendpunkt ist die URI, unter der man das Objekt ansprechen kann. In unserem Fall wäre dies für den Zugriff über den TCP Channel:

```
tcp://<servername>:8085/SagHallo/
```

Am wichtigsten in dem Aufruf ist aber die Konstante aus der *WellKnownObjectMode*-Aufzählung. Hier wird der Wert *Singleton* verwendet, der dafür sorgt, dass unser `HalloServer`-Objekt als SAO Singleton behandelt wird.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Http;
```

```
public class SimpleServer
{
   public static int Main(string [] args)
   {
      TcpChannel chanT = new TcpChannel(8085);
      ChannelServices.RegisterChannel(chanT);
      HttpChannel chanH = new HttpChannel(8086);
      ChannelServices.RegisterChannel(chanH);

      RemotingConfiguration.
         RegisterWellKnownServiceType
            (Type.GetType("HalloServer, HalloServer"),
            "SagHallo", WellKnownObjectMode.Singleton);

      System.Console.WriteLine("<Enter> beendet ...");
      System.Console.ReadLine();

      return 0;
   }
}

public class HalloServer : MarshalByRefObject
{
   public HalloServer()
   {
      Console.WriteLine("HalloServer-Objekt erzeugt.");
   }

   public String SagHallo(String name)
   {
      return "Hallo " + name;
   }
}
```

Listing 8.5: Veröffentlichung eines SAO Singleton-Objekts

SingleCall-Typen hingegen haben immer genau eine Instanz pro Client-Anfrage. Der nächste Methodenaufruf wird selbst dann über eine andere Instanz abgewickelt, wenn die vorherige Instanz vom System noch nicht aufgeräumt wurde. Dies entspricht dem statuslosen Zugriffskonzept über HTTP, bei dem Instanzen auf dem Server angelegt und gleich wieder zerstört werden.

In Listing 8.6 sehen Sie ein Beispiel für die Veröffentlichung eines Objekts als SAO SingleCall (Sie sehen nur den relevanten Teil, der Rest ist mit Listing 8.5 identisch). Der einzige Unterschied zum Code aus Listing 8.5 ist die Verwendung der Konstanten *SingleCall* aus der *WellKnownObjectMode*-Aufzählung.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Http;

public class SimpleServer
{
   public static int Main(string [] args)
   {
      TcpChannel chanT = new TcpChannel(8085);
      ChannelServices.RegisterChannel(chanT);
      HttpChannel chanH = new HttpChannel(8086);
      ChannelServices.RegisterChannel(chanH);

      RemotingConfiguration.
         RegisterWellKnownServiceType
            (Type.GetType("HalloServer, HalloServer"),
            "SagHallo", WellKnownObjectMode.SingleCall);

      System.Console.WriteLine("<Enter> beendet ...");
      System.Console.ReadLine();

      return 0;
   }
}
```

Listing 8.6: Veröffentlichung eines SAO SingleCall-Objekts

Ein SAO-Typ, der über einen HTTP Channel mit einem SOAP Formatter veröffentlicht wird, kann automatisch eine WSDL-Beschreibung bereitstellen. Denn bei dieser Kombination aus Channel und Formatter handelt es sich technisch gesehen um einen XML Web Service, ähnlich wie bei ASP.NET-basierten XML Web Services. Genau wie bei den Pendants in ASP.NET erhält man auch hier das WSDL durch einfaches Anhängen der Zeichenfolge *?wsdl* an den URL des Web Services. Für unser oben gezeigtes Beispiel sieht das wie folgt aus:

```
http://<servername>:8086/SagHallo?wsdl
```

Client Activated Objects

CAO-Typen sind im Gegensatz zu SAO-Typen echte zustandsbehaftete Objekte. Während bei einem SAO Singleton der Zustand für alle Client-Anwendungen geteilt wird und beim SAO SingleCall überhaupt keine Daten zwischen den Aufrufen gehalten werden, handelt es sich bei CAO-Typen um Objekte, die für den jeweiligen Client Daten zwischen den unterschiedlichen Aufrufen halten können (Zustand).

Die .NET Remoting-Infrastruktur bietet das Konzept des Leasings. Dies bedeutet, dass SAO Singleton-Typen und CAO-Typen nur für eine bestimmte Zeit leben. Jedes dieser Objekte hat eine bestimmte Lebensdauer. Diese ist von der Kombination aus *Leases*, einem *Lease-Manager* sowie einigen *Sponsoren* abhängig und wird von diesen gesteuert. Die Lebensdauer eines Objekts ist hier die gesamte Zeit, die das Objekt im Speicher aktiv ist. Diese Zeit kann durch Implementierung der Methode *MarshalByRefObject.InitializeLifetimeService* auch überschrieben werden. Der folgende Aufruf legt dann eine unendliche Lebensdauer fest:

```
public override Object InitializeLifetimeService()
{
   return null;
}
```

Ein genauere Betrachtung von Leasing und Sponsoren kann im Rahmen dieses Kapitels allerdings nicht vorgenommen werden.

Ein Beispiel für einen CAO-Typ sehen Sie in Listing 8.7. Im Beispiel wird demonstriert, dass CAO-Typen tatsächlich zustandsbehaftet sind: Es wird ein interner Zähler geführt, der bei jedem Aufruf erhöht wird. Er soll darstellen, dass Daten zwischen mehreren Aufrufen gehalten werden. Im Konstruktor des Objekts wird der Zähler initialisiert. Zusätzlich wird noch ein Instanzenzähler geführt, der anzeigt, wieviele unterschiedliche Instanzen des Objekts erstellt wurden. Im Destruktor des Objekts wird dieser Instanzenzähler entsprechend erniedrigt.

Für die Veröffentlichung als CAO-Typ ist die Methode *RemotingConfiguration.RegisterActivatedServiceType* verantwortlich, der lediglich der Typ übergeben wird.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class CAOServer
{
   public static void Main()
   {
      ChannelServices.RegisterChannel(new
         TcpChannel(8085));
      RemotingConfiguration.
         RegisterActivatedServiceType(
         typeof(CAOObject));

      Console.WriteLine("<Enter> beendet.");
      Console.ReadLine();
```

```
    }
}

public class CAOObject : MarshalByRefObject
{
   static int _zaehler;
   static int _instanz;

    public CAOObject()
    {
       _zaehler = 0;
       _instanz++;
       Console.WriteLine(this.GetType().Name +
          wurde erzeugt. Dies ist Instanz {0}.",
          _instanz);
   }

   ~CAOObject()
   {
      Console.WriteLine("Instanz {0} wurde zerstört.",
         _instanz);
      _instanz--;
   }

   public int hochZaehlen()
   {
      Console.WriteLine("Instanz {0}: hochZaehlen:
         {1}.", _instanz, _zaehler++);

      return _zaehler;
   }
}
```

Listing 8.7: Veröffentlichung eines CAO-Objekts

Neben der Veröffentlichung von neuen Objektinstanzen mit den oben vorgestellten Methoden ist auch das Publizieren bereits existierender Objekte möglich. Über die Methode *RemotingServices.Marshal* lässt sich eine existierende Instanz einfach der Remoting-Infrastruktur hinzufügen:

```
HttpChannel channel = new HttpChannel(8086);
ChannelServices.RegisterChannel(channel);
MeinObjekt obj = new MeinObjekt ();
RemotingServices.Marshal(obj,"meinObjekt.soap");
```

8.5 Konfiguration

Die bisher gezeigten Beispiele hatten sämtliche Informationen zur Registrierung der Objekte, zur Veröffentlichung der Channels und dem Einsatz der Formatter fest in den Quelltext integriert. Doch .NET wäre nicht .NET, wenn es hierfür nicht eine viel elegantere und mächtigere Variante gäbe. Mithilfe von Konfigurationsdateien lassen sich alle Einstellungen aus dem eigentlichen Code in ein XML-Dokument auslagern. Jede Anwendung in .NET kann eine eigene Konfigurationsdatei besitzen, in der sie anwendungsspezifische Daten speichert oder vom System vorgesehene Werte setzt. Damit können bestimmte Bestandteile der CLR und des Frameworks entsprechend konfiguriert werden.

Für die .NET Remoting-Funktionalität gibt es eine sehr umfangreiche und teils auch sehr komplexe Syntax für die XML-basierte Konfiguration. Ich möchte hier die für den täglichen Einsatz von Remoting wichtigsten Bestandteile zeigen. Die Einträge müssen unterhalb des Elements <configuration><system.runtime.remoting> vorhanden sein. Im Folgenden werde ich Ihnen Beispiele zeigen, welche die gleiche Funktionalität implementieren wie die bisher gezeigten Code-Beispiele und demonstrieren, wie man Channels und Formatter mischen kann.

Die Konfigurationsdatei in Listing 8.8 sorgt dafür, dass ein SAO SingleCall-Typ über einen TCP Channel auf Port 8085 und einen HTTP Channel auf Port 8086 veröffentlicht wird. Den entsprechenden Code für den Server-Host können Sie in Listing 8.9 sehen. Durch den Aufruf von *RemotingConfiguration.Configure* mit dem Namen der Konfigurationsdatei als Argument wird die Infrastruktur mit den notwendigen Werten versorgt. Dieser Ansatz ist somit sichtlich einfacher und flexibler. Anstatt den Host jedesmal neu kompilieren zu müssen, werden einfach die Werte in der XML-Datei verändert. Die Änderungen werden dann beim Start des Server-Hosts neu eingelesen.

Die Konfigurationsdatei für ein SAO Singleton sieht dann analog aus.

```
<configuration>
   <system.runtime.remoting>
      <application name="MyRemotingDemo">
         <service>
            <wellknown mode="SingleCall"
               type="WSBuchServer, Server"
               objectUri="WSBuchServer.rem" />
         </service>
         <channels>
            <channel port="8086" ref="tcp">
            </channel>
            <channel port="8088" ref="http">
            </channel>
         </channels>
```

```
      </application>
   </system.runtime.remoting>
</configuration>
```

Listing 8.8: Server-seitige Konfigurationsdatei für ein SAO SingleCall-Objekt

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Http;

public class WSBuchHost
{
   public static int Main(String[] args)
   {
      RemotingConfiguration.Configure(args[0]);

      System.Console.Write("<Enter> beendet.");
      System.Console.ReadLine();

      return 0;
   }
}
```

Listing 8.9: Server-Host mit Konfigurationsdatei

Etwas anders sieht die Konfigurationsdatei für CAO-Typen aus. Die Schablone in Listing 8.10 zeigt, dass anstatt eines `wellknown`-Elements ein `activated`-Element zum Einsatz kommt. Ansonsten kann man die bereits kennengelernten Abschnitte mit dem gewohnten Werten belegen.

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <activated type="ClientActivatedType,
               RemoteType"/>
            />
         </service>
      </application>
   </system.runtime.remoting>
</configuration>
```

Listing 8.10: Schablone für die Verwendung von CAO-Typen in Server-seitigen Konfigurationsdateien

Damit ein Entwickler frei entscheiden kann, welchen Channel er mit welchem Formatter zusammen verwendet, kann er diese Wahl in der Konfigurationsdatei festlegen. Diese Wahl wird allerdings zumeist auf der Client-Seite getroffen, da der Server von Haus aus alle Modi unterstützt – wenn er nicht entsprechend einschränkend konfiguriert ist.

Für den aufrufenden Client gibt es natürlich ebenfalls die Möglichkeit, Konfigurationsdateien zu verwenden. Das Beispiel in Listing 8.11 zeigt, wie für einen Client festgelegt wird, an welches Objekt über welchen Channel mit welchem Formatter er sich verbinden soll. In diesem Fall wird der SOAP Formatter zusammen mit dem TCP Channel gewählt. Eine weitere Alternative, abgesehen von den Standardkombinationen SOAP Formatter/HTTP Channel und Binary Formatter und TCP Channel, ist das Paar Binary Formatter und HTTP Channel. Für die Konstellation müssen einfach die Werte `http` für `channel` und `binary` für `formatter` gesetzt werden.

```
<configuration>
   <system.runtime.remoting>
      <application>
         <client>
            <wellknown type="WSBuchServer, Server"
               url="tcp://localhost:8086/MyRemotingDemo/
               WSBuchServer.rem" />
            <!-- <activated type="WSBuchServer, Server"
            /> -->
         </client>
         <channels>
            <channel ref="tcp">
               <clientProviders>
                  <formatter ref="soap" />
               </clientProviders>
            </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Listing 8.11: Client-seitige Konfigurationsdatei

Wir haben in diesem Abschnitt gesehen, dass die XML-basierte Konfiguration ein sehr mächtiges Instrument für den Remoting-Programmierer und auch für einen Administrator ist. Denn letzterer kann die Anwendungen an seine Bedürfnisse und Richtlinien anpassen, ohne zwingend mit dem Entwickler Rücksprache halten zu müssen.

8.6 Hosting

Außer in verwalteten Konsolenanwendungen können Remoting-Objekte in jeder verwalteten Anwendung gehostet werden. Dies umfasst auch Windows Dienste und Windows Forms-Programme. Zusätzlich können Objekte auch noch im IIS gehostet werden. Dies hat den Vorteil, die Sicherheitsfunktionalitäten des IIS (siehe Kapitel 7) in Anspruch nehmen zu können. Diese Option möchte ich Ihnen hier näher bringen.

Prinzipiell muss immer eine ASP.NET-Anwendung vorhanden sein (vgl. Kapitel 3). Darüber hinaus müssen alle Assemblies in einem Unterverzeichnis namens *bin* liegen. Damit nun überhaupt ein Hosting im IIS stattfinden kann, muss eine entsprechende Konfigurationsdatei erstellt werden. Wie es für Web-Anwendungen üblich ist, muss diese Datei *web.config* heißen. Ein Beispiel für eine solche Datei sehen Sie in Listing 8.12. Hier ist wichtig, dass kein anderer Port verwendet werden darf, als der, unter dem der IIS im System läuft. Außerdem kann die Anwendung in diesem Zusammenhang keinen Namen haben, da die Web-Anwendung im IIS bereits angelegt wurde und einen Namen besitzt.

```
<configuration>
   <system.runtime.remoting>
      <application>
         <service>
            <wellknown mode="SingleCall"
               type="WSBuchServer, Server"
               objectUri="WSBuchServer.rem" />
         </service>
         <channels>
            <channel ref="http">
            </channel>
         </channels>
      </application>
   </system.runtime.remoting>
</configuration>
```

Listing 8.12: Konfigurationsdatei für eine im IIS gehostete Remoting-Anwendung

Mit dieser Konstellation kann nun ein Client sich über HTTP auf den Port des IIS (Standard ist 80) mit dem Remoting-Objekt verbinden. Für unser Beispiel geschieht dies mit folgendem URL:

```
http://<servername>/<webanwendung>/WSBuchServer.rem
```

Der Code für das Objekt ist der gleiche wie in Listing 8.6 – allerdings nur der Teil für die Klasse `HalloServer`. Für den IIS muss der Quelltext für die zu hostenden Objekte in eine Assembly DLL kompiliert werden, die dann im Verzeichnis *bin* platziert wird.

Es gibt noch eine dritte Variante, entfernte Objekte zu hosten: über COM+. Da die Klasse *ServicedComponent* (siehe auch Kapitel 6) von *MarshalByRefObject* abgeleitet ist, können Clients sich über Remoting zu COM+-Anwendungen verbinden.

8.7 Client-Anwendungen

Im bisherigen Verlauf des Kapitel haben wir uns hauptsächlich mit den Server-Objekten und -Diensten beschäftigt. Am Ende des Kapitel werfen wir nun einen Blick auf Remoting-Clients, die mit entfernten Objekten kommunizieren wollen.

Ein .NET-Client kann sich über mehrere Wege mit einem Remoting-Objekt in Verbindung setzen. Diese Möglichkeiten sind:

- Programmatische Registrierung des Typs und Aufruf mit *new,*
- Verwendung der *Activator*-Klasse,
- Verwendung von Konfigurationsdateien und Aufruf mit *new.*

Der komplizierteste und umständlichste Weg ist die programmatische Konfiguration des Typs und der anschließende Aufruf mit *new.* Das Beispiel in Listing 8.13 zeigt, wie ein Client einen SAO-Typ über diesen Weg anspricht. Es muss eine Instanz des Typs *WellKnownClientTypeEntry* angelegt werden. Als Parameter nimmt der Konstruktor den Typ des entfernten Objekts und den URL des Servers entgegen. Wenn dieser Eintrag erfolgt ist, wird das *WellKnownClientTypeEntry*-Objekt über *RemotingConfiguration.RegisterWellKnownClientType* mit der Remoting-Infrastruktur registriert. Anschließend kann man das Objekt einfach durch den *new*-Operator instanziieren.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

public class Client
{
   public static int Main(string [] args)
   {
      TcpChannel chan = new TcpChannel();
      ChannelServices.RegisterChannel(chan);

      WellKnownClientTypeEntry entry = new
         WellKnownClientTypeEntry(
            typeof(HalloServer),
            "tcp://localhost:8085/SagHallo");
      RemotingConfiguration.
```

```
            RegisterWellKnownClientType(entry);

        HalloServer obj = new HalloServer();
        Console.WriteLine(obj.SagHallo("Bernhard"));

        return 0;
    }
}
```

Listing 8.13: Programmatische Registrierung und Aufruf durch new auf der Client-Seite.

Für einen CAO-Typ sieht die Prozedur ganz ähnlich aus. Nur wird anstatt der Methode *RegisterWellKnownClientType* die Methode *RemotingConfiguration. RegisterActivatedClientType(typeof(HalloServer), »tcp://localhost:8085/SagHallo«)* verwendet. Die Parameter für den Konstruktor sind dabei identisch. Zu beachten ist in diesem Fall, dass keine Registrierung mit der Infrastruktur notwendig ist.

Wesentlich einfacher im Vergleich ist der Einsatz der Klasse *Activator*. Anstatt den *new*-Operator zu verwenden, kann man über *Activator* Objektinstanzen erzeugen. Tatsächlich verwendet *new* unter der Haube diese Klasse auch zur Erzeugung lokaler Objekte. Um SAO-Typen über diese Klasse anzusprechen, benutzt man die statische Methode *GetObject*. Folgender Aufruf sorgt dann innerhalb einer Zeile dafür, dass der gleiche Effekt erreicht wird wie beim Aufruf in Listing 8.13:

```
HalloServer obj = (HalloServer)Activator.GetObject(
typeof(HalloServer), "tcp://localhost:8085/SagHallo");
```

Welche Methode man nun einsetzt, ist zum einen Geschmackssache und zum anderen eine Stilfrage. Über *new* wird suggeriert, dass der Typ sofort bei diesem Aufruf instanziiert wird. Da dies ja wie beschrieben nicht der Fall ist, ist der Aufruf über Activator.GetObject an dieser Stelle intuitiver.

Im Unterschied zu den SAO-Typen kann bei CAO-Typen nicht *GetObject* angewendet werden. Da die Objektinstanz ja tatsächlich auf Client-Anfrage hin erzeugt wird, muss in diesem Szenario mit der Methode *CreateInstance* aus der Klasse *Activator* gearbeitet werden. Diese Methode hat auch den Vorteil, dass man – im Gegensatz zu SAOs – auch nicht-standardmäßige Konstruktoren verwenden kann. *CreateInstance* hat eine lange Liste von Überladungen. Ich zeige Ihnen hier eine wichtige Variante. Diese Variante nimmt drei Argumente entgegen:

```
HalloServer obj = (HalloServer)Activator.CreateInstance
(typeof)HalloServer), constr, attr);
```

Das erste Argument ist der bekannte Typ des zu erzeugenden Objekts. Das zweite Argument ist ein Array von *object*-Typen und repräsentiert die Parameter für den Konstruktor unseres Zielobjekts. Das letzte Argument schließlich ist ebenfalls eine Array von *object*-Typen und versorgt die Aktivierung mit einer Reihe von Aktivierungsattributen. In unserem Fall ist dies lediglich der Endpunkt-URL des Objekts.

Der von mir vorgeschlagene Weg, wie ein Client auf entfernte Objekte zugreifen sollte, geht allerdings über Konfigurationsdateien. Wir haben in Abschnitt 8.5 bereits gesehen, wie mächtig diese XML-Dokumente sind. Und auch für die Client-Anwendungen liefern sie wertvolle Dienste.

Doch die heikle Frage ist eigentlich, wie der Client an die notwendigen Metadaten kommt. Der einfachste Fall ist, wenn der Entwickler von Server und Client eine Person ist. Dann kann der Client-Anwendung die Assembly des Servers zur Verfügung gestellt werden. Mit dieser Assembly kann der Client dann kompiliert werden. Doch dieses Vorgehen ist nicht immer gewünscht, da die Assembly ja normalerweise die gesamte Implementierung enthält. Ein Ausweg ist in diesem Fall, eine für Client und Server gemeinsame Assembly mit den Metadaten auf Schnittstellenbasis zu erstellen und diese zu benutzen.

Der einfachere Weg ist in jedem Fall, eine Metadaten-Assembly durch ein Kommandozeilenwerkzeug erstellen zu lassen. Dieses Werkzeug heißt *soapsuds.exe*. Für einen Remoting-Dienst, der über HTTP und mit dem SOAP Formatter ansprechbar ist, kann folgende Zeile unter Miteinbeziehung der WSDL-Beschreibung des Dienstes eine Metadaten-Assembly für den Client erstellen:

```
soapsuds.exe -url:http://<servername>/object.soap?wsdl -nowp
- oa:<zielname>.dll
```

Mit diesem Aufruf wird die WSDL-Beschreibung des Dienstes geholt und die Assembly mit den Namen <zielname> auf der Client-Seite abgespeichert. Der Parameter –nowp zeigt an, dass ein transparenter Proxy generiert wird, der alle Methoden des entfernten Objekts offen legt. Im Gegensatz hierzu ist auch ein so genannter Wrapper-Proxy möglich, der automatisch die Channels lädt, alle Methoden des Objekts offen legt und den Zugriff auf Channel-Eigenschaften ermöglicht. In unserem Falle müssen diese Informationen dann beispielsweise in einer Konfigurationsdatei vorhanden sein.

Mit dem gleichen Werkzeug kann man auch eine Metadaten-Assembly für einen auf TCP und Binary Formatter basierenden Remoting-Dienst erstellen. Der Name des Werkzeugs suggeriert zwar die ausschließliche Unterstützung von XML Web Services in Remoting, aber dies stimmt nur zum Teil. Über den -url-Parameter ist nur die Angabe des HTTP Channels möglich. Allerdings kann man mithilfe der Original-Assembly auch mit *soapsuds* eine Metadaten-Assembly für diese Art von Diensten erstellen: Der folgende Befehl erstellt eine Assembly mit Namen <zielname> auf Basis der Original-Assembly <originalassembly_ohneerweiterung>:

```
soapsuds.exe -ia:<originalassembly_ohneerweitertung>
-nowp - oa:<zielname>.dll
```

Der wichtige Punkt bei diesem Befehl ist das Weglassen der Dateiendung für die Original-Assembly. Mit den so erstellten Assemblies kann man dann den Client kompilieren und er kann sich auf die Metadaten in diesen Assemblies stützen.

Da mit .NET Remoting über den HTTP Channel und den SOAP Formatter auch XML Web Services realisierbar sind, können SOAP-fähige Clients von anderen Plattformen auf diese Dienste zugreifen. Doch hier ist Vorsicht geboten. Remoting legt den Fokus auf die Unterstützung des CLR-Typsystems, während ASP.NET XML Web Services und XML Web Services allgemein den Fokus auf das XSD-Typmodell legen. Wenn nun in einem Remoting-Dienst beispielsweise ein Objekt vom Typ *ArrayList* zurückgegeben wird, dann wird dies in der WSDL-Beschreibung zwar reflektiert, kann aber u.U. von anderen SOAP-Clients nicht verarbeitet werden (es wird ein Namensraum referenziert, der anderen Clients nicht bekannt ist).

Allgemein lässt sich festhalten, dass der SOAP Formatter die gesamte SOAP V1.1-Spezifikation unterstützt, aber auch noch viel mehr CLR-Datentypen zulässt – im Gegensatz zum `XmlSerializer` (siehe Kapitel 6).

8.8 Zusammenfassung

Neben der Möglichkeit, mit ASP.NET XML Web Services zu programmieren, bietet .NET Remoting eine wesentlich reichhaltigere und mächtigere Umgebung, um verteilte Anwendungen auf Basis des .NET Frameworks zu realisieren. Mit seiner erweiterbaren und generischen Natur lässt sich die Remoting-Infrastruktur den jeweiligen Bedürfnissen des Entwicklers sehr gut anpassen. Nicht zuletzt die Eigenschaft, dass auch mit Remoting XML Web Service-Anwendungen erstellt und Client-Anwendungen zur Verfügung gestellt werden können, macht Remoting sehr wertvoll.

In diesem Kapitel habe ich Ihnen die notwendigen Grundlagen für die meisten Einsatzgebiete von Remoting vorgestellt. Doch aufgrund der erwähnten Erweiterbarkeit war dies nur ein Kratzen an der Oberfläche. Für eine ausführliche Beschreibung aller Möglichkeiten innerhalb der Architektur benötigt man den Platz eines neuen Buchs.

Stichwortverzeichnis