.NET
-
Teleseminar Webservices

Seminararbeit
von

Dirk Mazur
aus
Heidelberg

vorgelegt am

Lehrstuhl Praktische Informatik IV
Prof. Dr. Wolfgang Effelsberg
Universität Mannheim

und am

Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB)
Prof. Dr. Hartmut Schmeck
Universität Karlsruhe

Juli 2004

Betreuer: Dipl. Inform. Matthias Bonn

Inhaltsverzeichnis

Abkürzungsverzeichnis

COM:	Component Object Model
CTS:	Common Type System
CLR:	Common Language Runtime
CIL:	Common Intermediate Language
ASP:	Active Server Pages
IIS:	Internet Information Services
HTTP:	Hypertext Transfer Protocol
SOAP:	Simple Object Access Protocol
HTML:	Hypertext Markup Language
IDL:	Interface Definition Language
WSDL	Web Service Description Language
XML:	Extensible Markup Language

Abbildungsverzeichnis

Codeverzeichnis

1. Einleitung

Diese Abhandlung ist die Ausarbeitung des Vortrages „Microsoft .NET“, der am 22. Juni 2004 im Teleseminar „Web Services“ am Lehrstuhl Praktische Informatik IV der Universität Mannheim gehalten wurde.

Im ersten Teil dieser Arbeit wird das .NET Framework im Allgemeinen kurz vorgestellt und der Begriff .NET genauer spezifiziert. Es wird eine Motivation gegeben warum ein neues Framework entwickelt werden musste und es werden die Probleme, welche bei den alten Lösungen noch nicht beseitigt worden sind, aufgezeigt. Eine genauere Betrachtung wichtiger Bestandteile eines .NET Programms wird im zweiten Teil dieser Arbeit vorgenommen, hier zeigen sich dann auch die Lösungsversuche der oben angesprochenen Probleme und Fehler. Der letzte Abschnitt befasst sich mit .NET im Zusammenhang mit Web Services, hier wird zum einen ein kleiner Beispiel-Web-Service vorgestellt und zum anderen auf den Lebenszyklus und Codierung von .NET Datentypen eingegangen.

2. Motivation

Das grundsätzliche Problem der Windows-Plattform war, dass man Programme, die für eine Plattform geschrieben worden sind nicht einfach miteinander koppeln konnte. Gründe hierfür sind die Differenzen in den Typsystemen und den Aufrufkonventionen, die Compiler der unterschiedlichen Sprachen hatten immer eine andere Vorstellung wie zum Beispiel eine Zeichenkette intern abgebildet wird.

Ein wenig wurden diese Probleme durch die Einführung von COM entschärft. Es war ein Schritt in die Richtung der Interoperabilität beliebiger Programmiersprachen. COM ist ein binärer Standard, der Vorgaben zum binären Aufbau von Objekten und einfachen Datentypen, der Speicherverwaltung und dem Aufruf von Methoden macht. Es gab aber immer noch Lücken in COM, die Komponenten mussten sich alle in der Registry anmelden, ein parallel Betrieb verschiedener Versionen war unmöglich, es gab nur eine beschränkte Objektorientierung und eine umständliche Objektlebenszyklusverwaltung mittels Referenzzählung.

Daher versuchte Microsoft diese Probleme mit ihrem neuen Framework zu lösen. Jim Miller einer der Architekten formulierte seine Motivation folgendermaßen: „Ich möchte nur zwei Dinge tun können – und das schon seit über 30 Jahren. Erstens Programme in einer Sprache schreiben, die mir gefällt, dabei aber Bibliotheken von anderen benutzen, die in anderen Sprachen entwickelt wurden. Zweitens Bibliotheken in einer Sprache meiner Wahl schreiben, die dann von anderen in ihren Sprachen genutzt werden“[1].

[1] Vgl. iX Special 1/03 S.6

3. Was ist .NET

Zu Beginn des Jahres 2002 brachte Microsoft nach mehrjähriger Forschung und Entwicklung die .NET-Technologie auf den Markt. .NET ist kein Betriebssystem im engeren Sinne und somit auch kein Nachfolger von Windows. Es handelt sich vielmehr um eine Schicht, die auf Windows (und später vielleicht auch auf anderen Betriebssystemen) aufsetzt und vor allem folgende zwei Dinge hinzufügt:

- Eine **Laufzeitumgebung**, welche automatische Speicherbereinigung (garbage colletion), Sicherheitsmechanismen, Versionierung und vor allem Interoperabilität zwischen verschiedenen Programmiersprachen bietet.
- Eine **objektorientierte Klassenbibliothek** mit umfangreichen Funktionen für graphische Benutzeroberflächen, Web-Oberflächen, Datenbankanschluss, Collection-Klassen, Threads, Reflection und vieles mehr. Sie ersetzt in vielen Fällen das bisherige Windows-API und geht weit darüber hinaus.

.NET ist aber mehr als das, es ist eine Softwareplattform, mit der programmiersprachenneutrale Software entwickelt werden kann. Die entwickelte Software ist hardwareneutral, arbeitet und funktioniert überall dort, wo .NET installiert ist. Als .NET wird neben dem .NET-System eine ganze Reihe von Softwareprodukten bezeichnet. Dieses umschließt den Bereich des Web-Services, wie z.B. Office.NET oder Passport, den Bereich der Anwendungen, wie der Enterprise Server oder das Visual Studio .NET[2] sowie einige Funktionen im Bereich der Dienste.
Was ist nun also .NET? Es ist ein aufeinander abgestimmtes Ensemble aus Betriebssystemkomponenten, Bibliotheken, Werkzeugen, Web-Services und Servern, das schwer in einem einzigen Satz zu definieren ist. Alles zusammen hat zum Ziel, die Programmierung von Windows- und Web-Anwendungen bequemer und einheitlicher zu gestalten. Eines ist jedoch klar: Die Softwareentwicklung unter .NET unterscheidet sich deutlich von der bisherigen Windows- und Web-Programmierung. Sie wird einfacher, eleganter und sicherer. Allerdings muss man sich dafür auch mit neuen APIs und neuen Konzepten vertraut machen.

[2] Vgl. [ViStud]

3.1. Das .NET-Framework

Das .NET-Framework bildet den Kern der .NET-Technologie. Es besteht aus einer Laufzeitumgebung und einer objektorientierten Klassenbibliothek, die alle Bereiche der Windows- und Web-Programmierung abdeckt. Dazu kommt noch die neue Programmiersprache C#, die auf .NET abgestimmt ist.

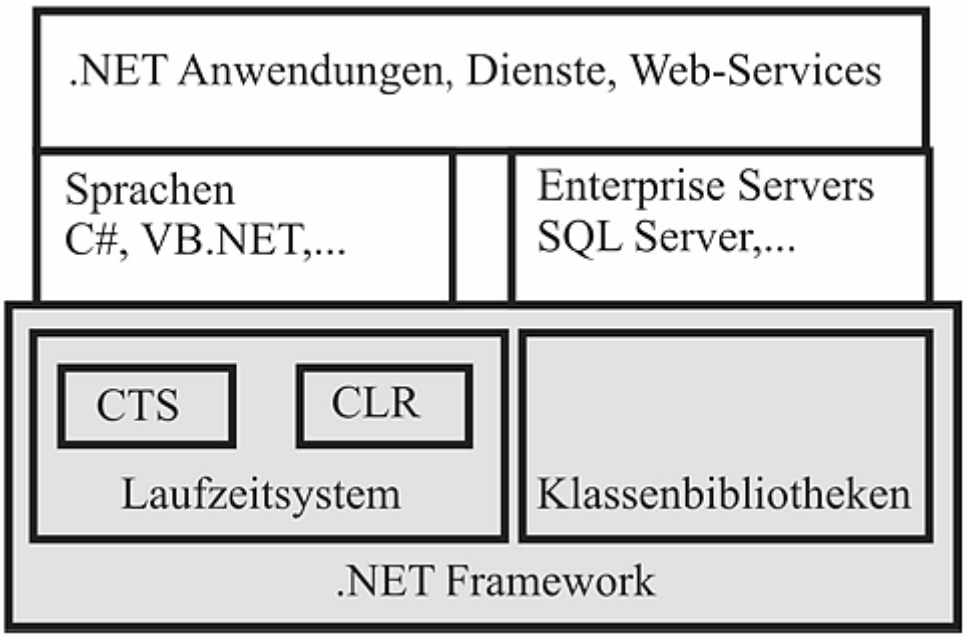

Abbildung 1: .NET Systemarchitektur

Die *Abbildung 1* zeigt den Zusammenhang der einzelnen Komponenten der .NET-Technologie. Das Laufzeitsystem ist nochmals unterteilt in das *Common Type System* CTS[3] und die *Common Language Runtime* CLR, diese Bestandteile und das .NET Framework selbst werden genauer in der Ausarbeitung von Dominik Schmieder beschrieben[4].

[3] Vgl. [MSDN]
[4] Vgl. [DoSchm]

4. Metadaten und Reflection

Ein .NET Programm und auch eine .NET Bibliothek bestehen nicht nur aus Maschinenbefehlen, der CIL[5], für die CLR, sondern es enthält auch immer *Metadaten* in Form von Tabellen. Die Zeilen dieser Tabellen enthalten ausführliche Beschreibungen der im *Modul* verwendeten oder definierten Typen und deren Komponenten.

Man kann drei Arten von Metadaten unterscheiden

- *Definitionstabellen* beschreiben Elemente, die in den Modulen selbst definiert werden
- *Referenztabellen* beschreiben Elemente, die im Modul nicht definiert, aber verwendet werden
- *Manifesttabellen*[6] bilden jenen Teil der Metadaten, der als Manifest bezeichnet wird. Darauf wird später noch eingegangen

Man kann auch sagen, dass die Metadaten eine Art Obermenge älterer Techniken zur Typbeschreibung sind, wie z.B. die Typbibliotheken von COM oder die die IDL-Dateien. Metadaten sind aber viel mächtiger als die alten Techniken, denn sie sind direkt mit dem Code verbunden, wie *Abbildung 2* zeigt und sind auch viel detaillierter. Sie sind immer in derselben Datei abgelegt wie der Code, direkt in dessen Modul eingebettet. Sie werden von jedem Compiler, der Code für die .NET-Plattform erzeugt, implizit miterzeugt und stimmen daher immer mit dem aktuellen Code überein.

Metadaten sind für alle Module gleich aufgebaut, dieses einheitliche Format kommt durch das CTS, daher können diese auch sprachübergreifend verwendet werden, wie das Beispiel [WebMethod] (siehe S.12) zeigt. Es ist ein Metaattribut und muss vor jeder Web-Methode angegeben werden, dies aber in jeder der von .NET unterstützen Programmiersprachen auf die gleiche Weise.

[5] Vgl. [MSDN1]
[6] Siehe. Abschnitt 5.2

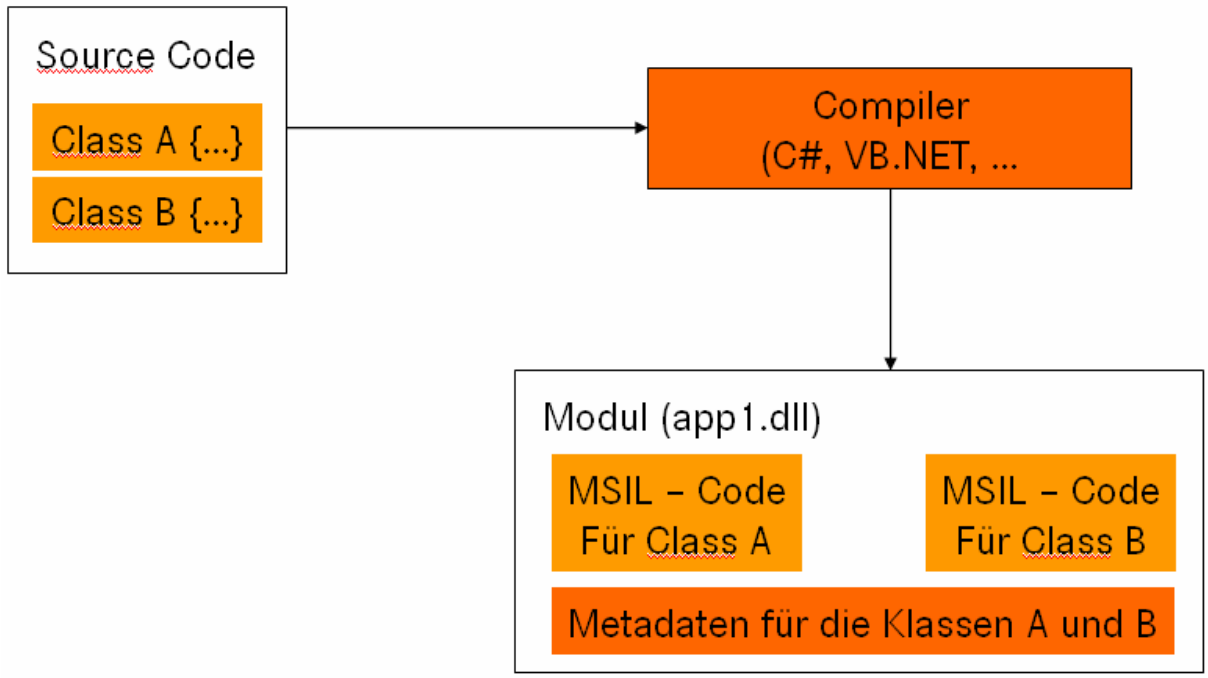

Abbildung 2: Enger Zusammenhang zwischen CIL und Metadaten

Metadaten können zur Laufzeit und auch während der Programmierung ausgelesen werden, dies nennt man dann Reflection. Reflection nutzt man z.B. bei IntelliSense, was sich hier auch gut anbietet, da sie immer mit dem Code übereinstimmen. IntelliSense ist die automatische Code-Vervollständigung, welche von vielen Entwicklungsumgebungen unterstützt wird und einem die Programmierung sehr erleichtert.

5. Assembly

5.1. Modul

Ein Modul ist ein Container für Typen, es enthält den CIL-Code der definierten Methoden und die Beschreibung der Typen (Metadaten). Der Grund warum man so selten mit Modulen zu tun hat, ist der, dass die CLR ein Modul alleine nicht verwenden kann. Ihr fehlen hier noch einige Meta-Informationen, die im *Manifest*[7] abgespeichert werden. Erweitert man ein Modul um ein Manifest, so wird es zum *Assembly*[8].

5.2. Manifest

In Manifest stehen alle Informationen, die nötig sind, um alle Definitionen eines Assemblies zu finden. Dieses wird auch immer beim Übersetzen des Quellcodes automatisch erzeugt. Im Manifest steht die Identität des Assemblies, welche ein String aus Name + Version + Ländercode ist. Ebenso darin enthalten sind die Informationen über alle Module die in diesem Assembly vorhanden sind und nicht zu vergessen die Informationen über die von außen zugreifbaren Methoden und Module, welche in der Exported Type Tabelle abgelegt werden.

5.3. Assembly

.NET unterstützt komponentenorientierte Softwareentwicklung. Die Komponenten heißen *Assemblies* und sind die kleinsten Programmbausteine, die separat ausgeliefert werden können. Ein Assembly ist ein Container für Module, d.h. ein Modul gehört immer zu mindestens einem Assembly. Assemblies sind eine logische Einheit, damit ist gemeint, dass sie im Verzeichnis auf der Festplatte nicht als solches erkannt werden können. Um festzustellen, welche Dateien zu einem Assembly gehören, muss man dessen Metadaten auswerten, das *Manifest*. Wie diese angeordnet sind zeigt *Abbildung 3*. Man sieht hier auch, dass es zwei Arten von Assemblies gibt, das single-file Assembly und das multi-file Assembly. Der große Vorteil eines multi-file Assemblies besteht in verteilten Anwendungen, die über das Internet oder das Intranet geladen werden müssen. Falls man beim Design des Programms schon weiß, dass nicht immer alle Teile des Programms verwendet werden müssen, kann man dieses in einzel-

[7] Siehe Abschnitt 5.2
[8] Siehe Abschnitt 5.3

ne Module aufteilen. Der Benutzer muss sich dann immer nur die Teile auf seinen Rechner laden, die er auch wirklich braucht, er hat darauf aber keinen Einfluss, da dies alles automatisch geschieht. Somit wird aber die Bandbreite des Benutzers nicht überlastet, da die Dateien auch bei Bedarf nachgeladen werden können. Ein weiterer Vorteil ist, dass nicht das komplette Programm beim Start in den Hauptspeicher geladen werden muss, sonder auch hier immer nur der Teil im Hauptspeicher ist, welcher auch wirklich benötigt wird. *Abbildung 3* zeigt zum einen die Unterscheidung dieser 2 Arten und auch welche Dateien noch alle zu einem Assembly gehören können.

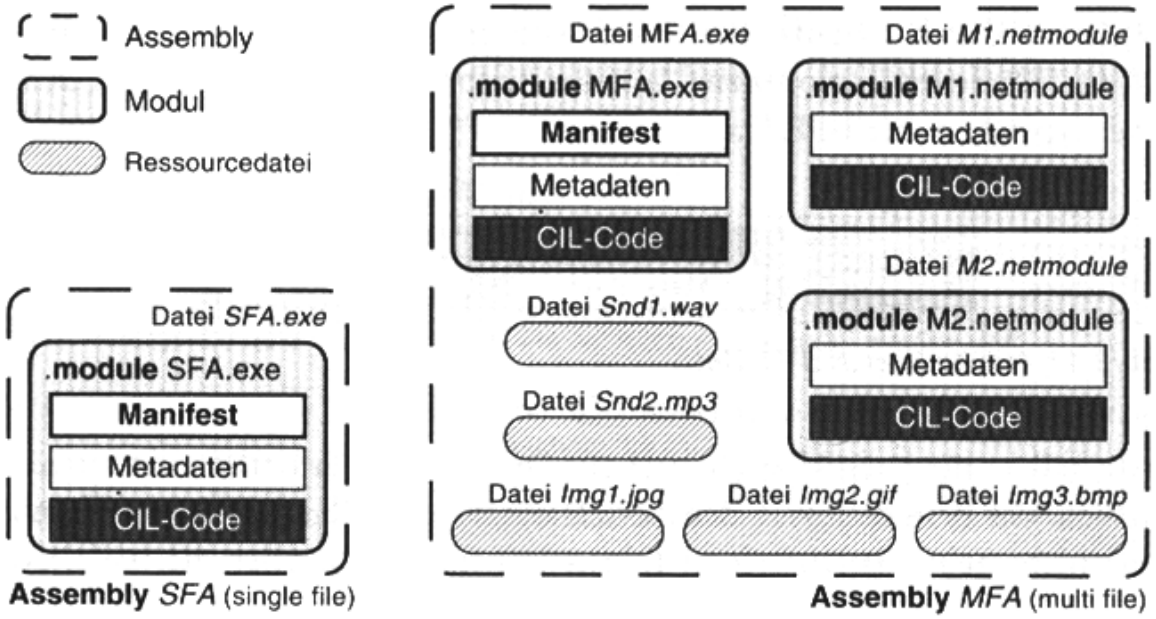

Abbildung 3: Unterschied single-file und multi-file Assembly

Ein Assembly hat verschiedene Aufgaben:

- Auslieferung
- Kapselung (Sichtbarkeit und Zugriffsrechte)
- Versionierung

Wie schon beschrieben sind in einem Assembly alle wichtigen Informationen, Typen und Daten gespeichert, es ist daher die kleinste alleine auszuführende Einheit. Jede Anwendung wird in der Form eines Assemblies ausgeliefert. Ein Assembly regelt auch die Sichtbarkeit und die Zugriffsrechte. Man kann genau beschreiben welche Teile des Betriebssystems ein Assembly benutzen darf, d.h. mein Assembly ist aus Sicherheitsgründen zum Beispiel nicht in der Lage Dateien über das Netzwerk zu verschicken. Es ist aber auch dafür zuständig, inwieweit andere Module auf die eigenen zugreifen dürfen, welche als *public* und welche als *priva-*

te gelten. Ein wichtiger Punkt und gleichzeitig eine große Neuerung ist die Versionierung der Assemblies. Man hat sich hier zwei Ansätze überlegt um verschiedene Versionen eines Programms auf einem Rechner verwalten zu können. Dies basiert auf der Unterscheidung zweier Arten von Assemblies:

- Private Assembly: Diese sind nur innerhalb eines Applikationsverzeichnisses bekannt, Verschiedene Versionen liegen in unterschiedlichen Verzeichnissen und können sich daher nicht in die Quere kommen.
- Öffentliche Assembly (shared Assembly): Sie stehen in einem systemweiten Repository, dem so genannten *Global Assembly Cache* GAC, und müssen durch einen *starken Namen* (strong Name) identifiziert werden. Verschiedene Versionen von Assemblies, die in gleichnamigen Dateien enthalten sind, können nun trotzdem vom System unterschieden werden und so nebeneinander existieren.

Die Identifikation der öffentlichen Assemblies erfolgt über den starken Namen. Dieser ist für jede Version des Programms eindeutig. Er besteht aus der Identität, verknüpft mit einem Public Key Token. Die Versionsprüfung erfolgt durch die CLR.
Diese neue Art der Versionierung ist eine große Verbesserung gegenüber anderen Techniken wie COM. Ein großes Problem bei alten Anwendungen war das parallele Ausführen verschiedener Versionen einer Anwendung. Dieses ist vielen als die DLL-Hölle bekannt. Neuere Versionen eines Programms haben Teile anderer Programme überschrieben und diese damit laufunfähig gemacht. Durch die globale Verwaltung auf einem System - durch den GAC – und die starken Namen ist dieses Problem größtenteils beseitigt worden. Jede Anwendung gibt nun genau an, welche Version einer Komponente sie braucht.
Es ist aber ein anderes Problem aufgetreten. Es kann nun sein, dass zwei Versionen eines Programms auf dem gleichen System laufen und sich nicht überschreiben oder löschen. Sie sind aber von der Struktur her sehr ähnlich, bedingt durch den gemeinsamen Ursprung. Daher kann es sein, dass sie die gleichen temporären Dateien verwenden. Sie stören sich nun also, falls sie zur gleichen Zeit aufgerufen werden, da sie die gleichen Dateien beschreiben wollen.

5.4. Sprachübergreifende Vererbung und Fehlerbehandlung

Durch das Konzept der Modularisierung und durch die CIL ist es möglich Anwendung in mehr als einer Programmiersprache zu schreiben. In .NET ist eine sprachübergreifende Vererbung möglich. Die Komponenten der in verschiedenen Sprachen implementierten Anwendung können durch das CTS und das einheitliche Design ohne Probleme miteinander kom-

munizieren. Es ist sogar möglich in Programmteilen aufgetretene Fehler in anderen abzufangen, auch wenn diese in einer anderen Sprache geschrieben worden sind.

6. Web Services mit .NET

Im zweiten Teil der Arbeit geht es nun um .NET im Zusammenhang mit Web Services. Das .NET Framework beinhaltet auch wieder Microsofts Webentwicklungsarchitektur ASP[9], diese wurde neu überarbeitet und heißt nun ASP.NET. Anwendungen in ASP.NET[10] laufen auch unter der CLR und können in jeder .NET Sprache programmiert werden. Die Besonderheit von ASP.NET ist die native Unterstützung von Web Services. Da Web Services zustandslos sind, bietet ASP.NET hier einiges an Hilfe, was später aufgezeigt wird.

6.1. Aufbau eines Web Services

Ein Web Service unter .NET braucht zum einen ein eindeutiges virtuelles Verzeichnis, welches vom IIS[11] verwaltet wird. Des Weiteren benötigen Web Services noch eine Infrastruktur die folgendes beinhaltet:

- Ein Anwendungs- oder Übertragungsprotokoll wie HTTP GET und HTTP POST oder SOAP
- Eine Beschreibungsmöglichkeit des Web Services, damit jeder Client weiß welche Funktionen der Web Service bereitstellt und wie diese genutzt werden können
- Eine Suchfunktion, um als Client über das Vorhandensein eines Web Services bescheid zu wissen

Diese Dinge stellt .NET in seinem Web Service Namespace bereit.

- *System.Web.Services:* Hier ist eine minimal vollständige Menge aller benötigten Typen zur Erstellung eines Web Services
- System.Web.Services.Description: Die Typen in diesem Namespace ermöglichen die Kommunikation mit WSDL[12] aus einem Programm heraus
- System.Web.Services.Discovery: Dieser Namespace stellt Typen zu Verfügung mit dessen Hilfe und einer DISCO-Datei man Web Services, die auf einem bestimmten Rechner installiert sind, finden kann.
- System.Web.Services.Protocols: Hier werden Typen definiert, die die Anwendungs- oder Übertragungsprotokolle darstellen.

[9] Vgl. [MSDN2]
[10] Vgl. [MSDN2]
[11] Vgl. [IIS]
[12] Vgl. [WSDL]

6.2. Erstellen eines einfachen Webdienstes

Um einen einfachen Web Service in .NET zu programmieren braucht man nicht viel. Es reicht eine Datei mit der Endung .asmx anzulegen und diese dann mit dem Texteditor zu bearbeiten. Hier ist nun der Quellcode eines einfachen Web Services dargestellt, dieser implementiert nur eine Methode, diese liefert die Summe von zwei Integer Zahlen zurück, die man vorher übergeben hat. Bei der Erstellung eines einfachen Web Services gibt es nur einige wenige Dinge auf die man achten muss. Als erste Zeile muss immer das Attribut mit der Programmiersprache gesetzt werden und die Klasse, die den Web Service beinhaltet angegeben werden. Es gibt hier die Möglichkeit den Code in eine andere Datei auszulagern, indem man das Attribut „Code behind= MyCode.asmx.csh“ (csh gibt hier die Programmiersprache an) setzt und dort dann die Datei angibt in dem dieser steht. MyCode.asmx.csh steht hier natürlich für einen beliebigen Dateinamen. Dies kommt aus dem Bereich der ASPs, da es hier sinnvoll war, den Code von der Darstellungsbeschreibung zu trennen. Wichtig ist natürlich auch der Namespace System.Web.Services, welchen man importieren sollte. Um eine Methode nach außen hin sichtbar und auch zugreifbar zu machen, muss man nur das Attribute [WebMethod] voranstellen. Hier ist auch noch die Möglichkeit gegeben andere Attribute für die Methode zu setzten, sei es den Namen den sie im Web Service haben soll, oder falls man nur eine Beschreibung hinzufügen möchte, wie in diesem Beispiel *Code 1*. Die Datei muss nur noch in ein Applikationsverzeichnis auf dem IIS kopiert werden und ist dann unter einem Browser aufrufbar. Beim ersten Aufruf wird diese dann kompiliert, was zu einer kleinen Verzögerung führen kann. Sie ist dann aber in Form von CIL vorhanden und daher auch ohne Verzögerung aufrufbar.

```
<%@ WebService Language="c#"  Class="Service1" %>
using System;
using System.Web.Services;

[WebService(Description="Mein toller WS")]
public class Service1 : WebService
{
      [WebMethod(Description="Addiert 2 zahlen")]
      public int Add(int a, int b)
      {
            return a + b;
      }
}
```

Code 1: Ein kleiner Web Service

.NET generiert zu jedem Web Service eine kleine HTML - Seite um diesen auch testen zu können. Man kann sich diese über einen Browser anzeigen lassen. Da Web Services aber eigentlich zur Kommunikation auf Programmebene gedacht sind, ist dies auch wirklich nur zu Testzwecken geeignet.

Zum Erzeugen eines Clients bedient man sich am Besten der Hilfe eine IDE, wie zum Beispiel dem Visual Studio. .NET liefert zwar auch Befehlszeilentools zur Erzeugung von Proxy Klassen aus WSDL Dateien mit, dieses ist aber lange nicht so komfortabel wie die Entwicklung mit Visual Studio .NET[13]. Als erstes muss man einen Webverweis zu seinem Programm hinzufügen, welcher dann automatisch die Proxyklasse erzeugt. *Abbildung 4* zeigt, wie man dies in Visual Studio machen kann. Man hat nun die Möglichkeit in seiner Client Anwendung einfach ein Objekt des Web Services zu erstellen und so auf dessen Funktionalität zuzugreifen. Jetzt gilt es nur noch die eigentliche Logik der Clientanwendung zu implementieren. Ein Ausschnitt davon ist im *Code 2* zu sehen.

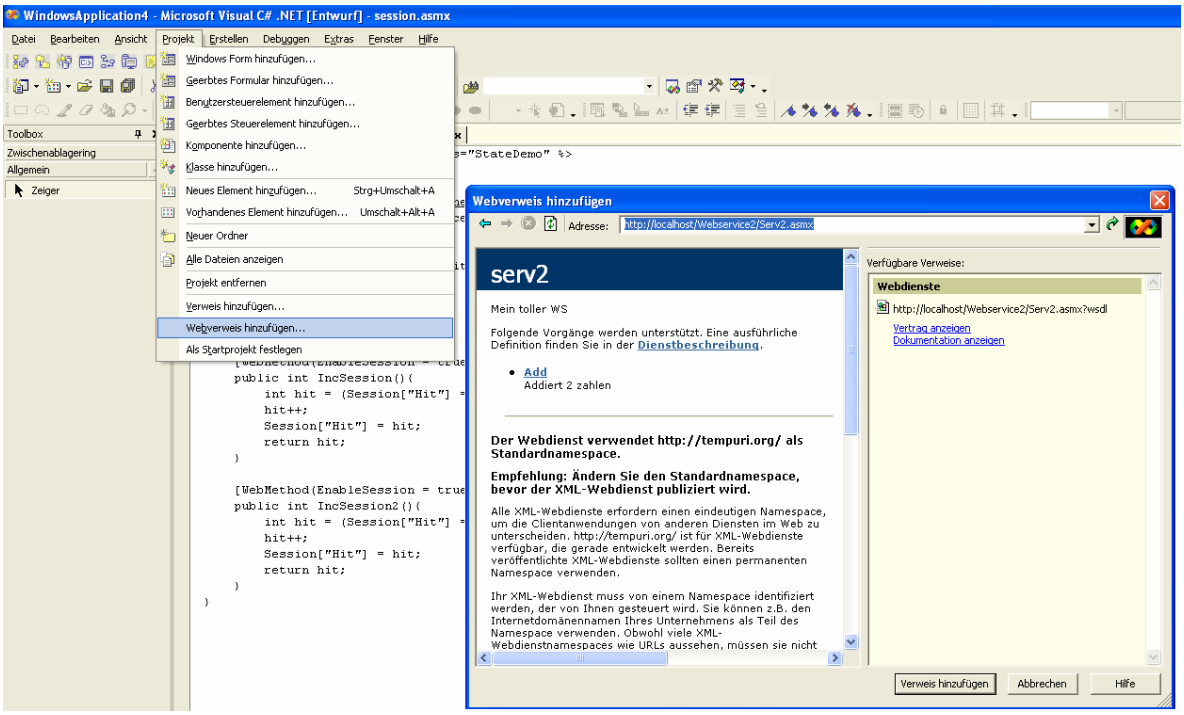

Abbildung 4: Einfügen eines Webverweises

Code 2 zeigt aber auch, dass es zwei Arten gibt, wie man eine WebMethod aufrufen kann. Entweder synchron oder asynchron. Nachdem man seine Instanz des Web Services erzeugt hat, kann man beim synchronen Aufruf auf diese Methoden wie auf lokale Methoden zugrei-

[13] Vgl. [ViStud]

fen. Da die Kommunikation aber die Schwachstelle ist, kann es natürlich passieren, dass diese abbricht oder sehr lange dauert, daher gibt es auch die Möglichkeit die Methoden asynchron aufzurufen. Der Vorteil liegt hier auf der Hand. Der Programmfluss wird nicht durch eine lange Verzögerung oder gar eine Unterbrechung beeinflusst, es kommt daher nicht zu einem Stoppen oder Hängen des Programms.

```
//proxy-Klasse deklarieren und instanziieren
localhost.Service1 myService = new localhost.Service1();

private void button1_Click(object sender, System.EventArgs e)
{
      //synchroner WS-Aufruf, Ergebnis speichern
      int result = myService.Add(47, 11);
      textBox1.Text = result.ToString();

      //asynchroner Aufruf
      AsyncCallback cb = new AsyncCallback (this.addIsComplete);
      IAsyncResult res = myService.BeginAdd (8, 15, cb, null);
}

//Methode für asynchronen Rückruf
private void addIsComplete(IAsyncResult res)
{
      int sum = myService.EndAdd (res);
      textBox2.Text = sum.ToString();
}
```

Code 2: Web Methoden Aufruf

6.3. Lebenszyklus von Webdiensten

Da ASP.NET Web Services als zustandslose Objekte realisiert, die zum Methodenaufruf erzeugt und nach dem Beenden der Methode wieder zerstört werden, stellt sich die Frage, wie man Daten über mehrere Aufrufe aufbewahren möchte. Hierzu muss man die Daten im aktuellen Zustand der Sitzung (session) oder der Anwendung (application) speichern. Dies geschieht mit dem Zugriff auf die Properties Session und Application. Auf Application hat jedes Web Service Objekt zugriff, um auch das Session Property verwenden zu können muss man im Attribut [WebMethod] das Property EnableSession setzen.

Code 3 soll den Zugriff auf diese Properties verdeutlichen. Die Methode IncApplication greift auf das Application-Property zu und erhöht einen Zähler. Dies geschieht bei jedem Aufruf des

Web Services, egal von welchem Rechner aus. IncSession dagegen greift auf das Session-Property zu und erhöht einen anderen Zähler. Der Benutzer wird hier anhand von Cookies identifiziert. Bei jedem erneuten Aufruf bekommt er immer wieder seinen alten Wert um eins erhöht zurückgeliefert.

```
<%@ Webservice Language="C#" Class="StateDemo" %>

using System.Web.Services;
[WebService(Namespace="http://dotnet.jku.at/StateDemo/")]
public class StateDemo : WebService {
      [WebMethod()]
      public int IncApplication(){
            int hit = (Application["Hit"]==null)?0:(int)Application["Hit"];
            hit++;
            Application["Hit"] = hit;
            return hit;
      }

      [WebMethod(EnableSession = true)]
      public int IncSession(){
            int hit = (Session["Hit"] == null)? 0 : (int)Session["Hit"];
            hit++;
            Session["Hit"] = hit;
            return hit;
      }
}
```

Code 3: Lebenszyklus von Web Services

6.4. SOAP Codierung von .NET Datentypen

Normalerweise werden die Datentypen nach der SOAP-Spezifikation 1.1 SOAP[14] serialisiert und deserialisiert. Man hat in .NET nun aber die Möglichkeit auf die Codierung über .NET-Attribute Einfluss zu nehmen. Eine Auswahl aus diesen Attributen ist:

- SoapAttribute: Mit diesem Attribut wird ausgedrückt, dass die Daten nicht als Unterelemente, sondern als XML-Attribute ihres Typs codiert werden sollen. Dies ist bei großen Datenmengen effizienter als die Codierung als Unterelement. Ein Beispiel gibt *Code 4.*

[14] Vgl. [SOAP]

- SoapElement: Dieses Attribut gibt an, dass die Daten als XML-Element und nicht als Attribute codiert werden sollen, dies ist aber auch die Standardeinstellung. Man hat hier nun aber auch zusätzlich die Möglichkeit auch auf die Datentypen Einfluss zu nehmen wie auch schon beim obigen Element. Ein Beispiel gibt
- *Code* 5.
- SoapIgnoreAttribute: Dieses kann auf öffentliche Felder und Properties angewendet werden und gibt an, dass diese nicht serialisiert werden sollen
- SoapIncludeAttribute: Dies kann auf Web Service Klassen und öffentliche Web Service Methoden angewendet werden. Damit wird angegeben, dass ein bestimmter Typ in die Beschreibung des Web Services mit aufgenommen werden soll, zum Beispiel eine Unterklasse B einer im Web Service verwendeten Klasse A. Damit können Objekte, welche vom statischen Typ A, aber vom dynamischen Typ B sind serialisiert werden.

```
Public struct TimeDesc {
[SoapAttribute] public string TimeLong;
[SoapAttribute] public string TimeShort;
[SoapAttribute(DataType=„nonNegativInteger“, AttributeName=„ZoneID“)] pub-
lic string TimeZone;
}
Wird in Soap codiert als:
<types:TimeDesc id=„id1“ xsi:type=„types:TimeDesc“
types:TimeLong=„10:00:25“ TimeShort=„10:00“ types:ZoneID=„1“ />
```

Code 4: Codierung von Datentypen I

```
Public struct TimeDesc {
public string TimeLong;
public string TimeShort;
[SoapElement(DataType=„nonNegativInteger“, ElementName=„ZoneID“)] public
string TimeZone;
}
Wird in SOAP codiert als
<types:TimeDesc id=„id1“ xsi:type“types:TimeDesc“>
      <TimeLong xsi:type=„xsd:string“> 10:00.25</TimeLong>
      <TimeShort xsi:type=„xsd:string“> 10:00 </TimeShort>
      <ZoneID xsi:type=„xsd:nonNegativeInteger“>1</ZoneID>
</types:TimeDesc>
```

Code 5: Codierung von Datentypen II

7. Fazit und Ausblick

Das .NET-Framework ist ein guter Ansatz, um die Probleme, die im Abschnitt Motivation angesprochen worden sind, zu lösen. Es ist auf jeden Fall im Bezug auf die Interoperabilität von Programmiersprachen und auf die Interoperabilität verschiedener Plattformen zukunftsweisend. Die Lauffähigkeit auf anderen Systemen wie UNIX oder LINUX sollen mit dem Projekt von Mono[15] gelöst werden. Hier wird das .NET-Framework auf diese Plattformen portiert, was aber leider noch nicht ganz abgeschlossen ist. Für die Softwareentwicklung unter Windows ist .NET auf jeden Fall die Zukunft.

Man hat gesehen, wie .NET mit den Problemen der Versionierung und der Objektorientierung sprachenübergreifend umgeht. Das Deployment von Anwendungen ist deutlich einfacher geworden, es ist jetzt nur noch ein Kopieren der Dateien in ein bestimmtes Verzeichnis notwendig, die Eintragungen in die Registry und die damit verbundenen Probleme entfallen. Die Webprogrammierung, sowie die durchgängige Unterstützung von XML und Web Services machen .NET für zukünftige Anwendungen sehr interessant. Der Wunsch von Jim Miller ist auch in Erfüllung gegangen, es ist nun für jeden möglich in seiner favorisierten Programmiersprache zu schreiben und trotzdem auf eine große allgemein verfügbare Klassenbibliothek zugreifen zu können. Genauso kann jeder die Programme eines anderen in seine Anwendung einbinden, egal in welcher Programmiersprache diese geschrieben worden sind.

[15] Vgl. [Mono]

Literaturverzeichnis

[AnTor] Andrew Troelsen (2002): C# und die .NET-Plattform

[iX] iX Special 1/03: Programmieren mit .NET

[WoBe] Wolfgang Beer / Dietrich Birngruber / Hanspeter Mössenböck / Albrecht Wöß (2002): Die .Net Technologie – Grundlagen und Anwendungsprogrammierung

[Mono] Mono Project
www.go-mono.org

[COM] Component Object Model
http://www.microsoft.com/com/

[MSDN1] Microsoft .NET Framework Developer Center
http://msdn.microsoft.com/netframework/default.aspx

[MSDN2] Microsoft ASP.NET Developer Center
http://msdn.microsoft.com/asp.net/

[DoSchm] Dominik Schmieder (2004): Tele-Seminar Web Services - Microsoft .NET

[IIS] Microsoft Internet Information Services
http://www.microsoft.com/windowsserver2003/iis/default.mspx

[WSDL] Web Service Description Language
http://www.w3.org/TR/wsd

[SOAP] Simple Object Access Protocol
http://www.w3.org/TR/soap/

[ViStud] Visual Studio .NET
http://msdn.microsoft.com/vstudio/