Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Studienarbeit

Eignung von .Net für Webservices

vorgelegt von
Rainer Schulz
am 24. März 2005

Studiengang Softwaretechnik

Betreuender Prüfer: Prof. Dr. Kai von Luck

Fachbereich Elektrotechnik und Informatik
Department of Electrical Engineering and Computer Science

Inhaltsverzeichnis

1. Einleitung

Eignung von .Net für Webservices

Webservices, ein Dienst der mit Hilfe von XML über das Internet Netzwerkprotokoll erbracht wird, zieht immer mehr Aufmerksamkeit auf sich. Da auch Microsoft in diesem Bereich sehr tätig ist und behauptet, dass Webservices mit .Net sehr leicht zu entwickeln sind, habe ich mich gefragt

Wie tauglich ist .NET für Webservices?

Web Services bieten, wie der Namen schon sagt, Dienste im Internet an, und dies nicht für menschliche Benutzer, sondern für Softwarekomponenten, die Informationen sammeln müssen. Um dies zu gewährleisten, arbeitet man überwiegend mit 4 Techniken und Protokollen die XML basierend sind. Dies sind SOAP, XML-RPC, WSDL und UDDI. Microsoft behauptet, dass Sie diese Techniken einfach und verständlich in ihrem Framework .NET oder spezieller ASP.NET integriert haben und es sehr einfach sein soll mit Hilfe von Visual Studio .Net solche Webservices zu erstellen. Ich werde mir im Laufe dieser Studienarbeit anschauen, wie man in Visual Studio .NET Webservice erstellt und verwaltet. Dazu werde ich mir exemplarisch spezielle Anwendungsfelder nehmen und versuchen diese in .NET zu realisieren. Dies könnten verteilte Transaktionen sein, also z.B. «liefere mir den günstigsten Mietwagen nach München, von 3 Anbietern und miete diese Wagen gleich» oder andere Dienste die man nutzen möchte. Anfangs werde ich mir das Framework anschauen und einen simplen Webservice erstellen. Auf diese versuche ich dann zu zugreifen. Wenn diese funktioniert kann man sich anschauen, wie dieser Service noch erweitert werden kann und wie tauglich dazu .Net ist. Zusätzlich halte ich während meiner Studienarbeit Rücksprache mit Lars Burfeindt, welcher sich Webservices im Bereich Java anschaut. Vielleicht kann man dann daraus Vergleiche ziehen und sehen, wo .NET oder Java ihre Stärken oder Schwächen in diesem Bereich haben.

Im Verlauf dieser Arbeit, werde ich im Kapitel 2 kurz darauf Eingehen, warum ich mich für dieses Thema entschieden habe und gebe im Kapitel 3 einen kurzen Überblick, welche Technologien in Webservices genutzt werden. Dort wird genauer erläutert, um was es sich bei SOAP, WSDL und UDDI handelt. Zusätzlich wird es noch einen Überblick der .NET Technologie geben, welche für diese Arbeit zu Grunde liegt.

Darauf Aufbauend hab ich im Kapitel 3 ein kleines Szenario aufgebaut, welches repräsentativ für einen gängigen WebService steht, der so oder in ähnlicher Form in der Internetwelt vorkommen könnte. Anhand dieses Beispielszenarios untersuche ich in Kapitel 4, wie sich dieser Service in .Net erstellen lässt und ob dieser so auch den Erwartungen und Problemen der realen Welt stellen kann. Aufgrund dieser Untersuchung werde ich ein kurzes Resueme in Kapitel 5 geben und einen kleinen Ausblick erwähnen, wo es evtl. noch genauere Untersuchungen bedarf oder gar noch Nachholbedarf besteht. Im Anhang wird es ein kurzes HowTo geben, wie man sehr schnell einen simplen WebService mit hilfe von Visual Studio .Net erstellt.

2. Motivation

Sicherlich hat jeder gemerkt, dass Microsoft in den letzten 3 Jahren im Bereich Internet grosse Sprünge gemacht hat. Im Moment ist Microsoft in vielen Bereichen des Internets Markführer und viele, wenn nicht alle Produkte sind Internetbasierend.

Das Thema «Webservice» spielt immer eine grössere Rolle im Internet, da fast alle Firmen dort vertreten sind und sich dadurch eine gute Möglichkeit bietet, seine Dienste über das Internet anzubieten. So wandelt sich die alte «B2C»-Anwendungsarchitektur (Business-to-Consumer) immer mehr zur «B2B»-Anewendungsarchitektur (Business-to-Business). Da es aber kein einheitlichen Protokoll in diesem Bereich gab, hat Microsoft 1997 versucht dieses zu Entwickeln und es entstand die erste Version von SOAP (Simple Object Access Protocol). Die gesamte .Net Architektur ist auf Webentwicklung spezialisiert und alte Technologien wurde auch der Web Service Entwicklung angepasst (ASP.NET).

Da ich in meiner Laufbahn an der HAW-Hamburg viel mit Java zu tun hatte, interessierte mich die .Net Technologie und der Umgang mit dem Entwicklungstool Visual Studio. Daher werde ich im Rahmen meiner Arbeit versuchen einen Exemplarischen Webdienst mit .Net zu entwickeln und diesen daraufhin Untersuchen, ob es wirklich so einfach ist, diesen Dienst zu Erstellen, Publizieren und Warten.

Im Rahmen eines Projektes von Herrn Prof. Luck, dem PartLive Projekt, bietet sich eine Möglichkeit, diese Technologie zu Untersuchen.

3. Grundlagen

Bevor ein Szenario aufgebaut wird und ein Webservice mit .Net erstellt wird, sollten vorher ein paar Grundlagen zum Thema Webservices und .Net behandelt werden.

3.1. .NET Framework

> «.NET Framework ist eine neue Computerplattform, die in der hochgradig verteilten Internetumgebung die Entwicklung von Anwendungen vereinfacht.» [1]

Das .Net Framework besteht grundlegend aus zwei Komponenten. Die Runtime Enviroment genannt*Commom Language Runtime (CLR)* und der .Net - Klassenbiliothek *Framework Class Library (FCL)* . Die FCL ist über der CLR angesiedelt und bietet Services an, die man benötigt um moderne Applikationen zu entwickeln (Siehe Abbildung 3.1).

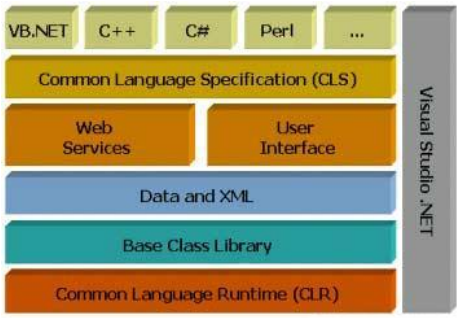

Abbildung 3.1.: .Net Framework

Zusätzlich ist es möglich in .NET in einer Programmiersprache seiner Wahl (oder die Wahl, die einem Microsoft vorgibt) zu Programmieren. Wie in Abbildung 3.1 zu sehen ist, werden zum jetzigen Stand schon viele Sprachen unterstützt (die gängigsten sind VB.Net und CSharp). Um diese verschiedenen Sprachen nutzen zu können, hat Microsoft die *Common Language Specification (CLS)* und das *Common Type System (CTS)* entwickelt.

Common Language Runtime (CLR)

Die CLR ist eine moderne Laufzeitumgebung, welches die Ausführung von User Code managed. Services anbietet, wie die Just in Time (JST) kompilierung, Speichermanagement, Fehlerbehandlung, Debugging und Security-, Permissionmanagement. In Abbildung 3.2 ist ein kurzer Überblick gegeben. Um mehr darüber zu erfahren verweise ich an die MSDN-Library [1] oder andere Fachliteratur [13].

Abbildung 3.2.: Common Language Runtime

Common Type System (CTS)

Die CTS definiert die Regeln, wie alle Datentypen deklariert, definiert und gemanaged werden, unabhängig der Programmiersprache. Dies bietet die Basis für Cross-Language Integration. Erst wenn alle Datentypen der CTS genügen, ist es möglich, verschidenen Programmiersprachen zu kombinieren.

Common Language Specification (CLS)

Da nicht alle Sprachen die gleichen Konstrukte aufweisen ist die Common Language Specification von Nöten. Beispie: Sprache A unterstützt unsigend types und Sprache B nicht, wie kann man jetzt aus B eine Methode von A aufrufen? Hier kommt die CLS zu tragen, sie definiert eine Untermenge der CTS, welche diese Probleme beheben soll. Jede verwaltete Sprache entscheidet, wie viel der CTS unterstützt wird. Eine Sprache, die alles der CTS Unterstützt, nennt man *CLS Consumer* . Eine Sprache, die die CTS erweitert nennt man *CLS Extender* . C# ist z.B. eine Sprache die sowohl Consumer und Extender ist. Jeder Sprache ist es frei , die CLS zu unterstützen oder gar darüber hinaus zu gehen. Doch erst durch diese Gemeinsamkeit ist es richtig möglich Sprachen zu kombinieren. Zur Zeit meiner letzten

Recherche gab es sechs Sprachen, die dem genügten (C#, VB.NET, JScript, Managed Extensions for C++, Microsoft IL, und J#). Aber dieses ändert sich, da Microsoft und externe Firmen versuchen, immer mehr Sprachen .Net tauglich zu machen.

Framework Class Library (FCL)

Der Anspruch der Entwickler ist im Laufe der Zeit, in der Windows immer weiter entwickelt worden ist, mehr und mehr gestiegen. Die API's wuchsen derart an, dass es fast unmöglich war, diese komplett zu Überblicken. Durch das Durchsetzten von modernen Paradigmen, wie Objektorientierung, Komponentensofware und Internet wurde die Programmierung in der Windows - Umgebung immer schwerer. Also wurde es Zeit für einen Neuanfang. Als Resultat wurde die Framework Class Library (FCL) entwickelt, die die meisten traditionellen Windows API's in eine gut organiesierte, objectorientierte Umgebung bringt. Die FCL bietet eine grosse Anzahl von Diensten, die für moderne Applicationen nötig sind. Darunter fallen z.B.

- Grundlegende Funktionalitäten für Basistypen, Collections, Netzwerkdienste und I/O Operationen
- Unterstützung für Datenbankzugriffe, XML - Funktionalitäten
- Webbasierte Cliententwicklung (thin Client) mit Anbindung an rich Servern.
- Desktop Cliententwicklung mit Support der Windows GUI
- Entwicklung von SOAP basierenden XML WebServices

Die FCL umfasste mehr als 3.500 Klassen. [3]

3.2. Was ist ein Webservice ?

IBM verstand unter dem Begriff Webservice:

> «Web services are self-contained, modular applications that can be described, published, located, and invoked over a network, generally, the World-Wide Web.»

Im Grunde ist der Web Service nichts Neues. Es ist eine Verteilte Anwendung, die Remote über das Internet, Dienste anderer Programme aufruft. Vorher war es sehr aufwendig, eine solche verteilte Anwendung zu entwickeln. Man musste auf verschiedene Middelware - Ansätze (CORBA, RMI, COM usw.) zurückgreifen. Der Platformunabhängigkeit oder Sprachunabhängigkeit waren dadurch Grenzen gesetzt. Im Grunde ist der Web Service eine Art von Client/Server Middelware.

Das Problem einer Middelware ist die Vielfalt an Sprachen und Konzepten, um an die Schnittstellen eines Informationssystems zu gelangen. Wie oben angesprochen gibt es viele Middelware - Ansätze die eine komplexe Ablaufumgebung erforderten. Dazu kamen die verschiedenen Protokolle, die diese Ablaufumgebung mit sich brachte, siehe Bild 3.1.

Dadurch, dass das Internet immer weiter gewachsen ist, ergab sich eine Möglichkeit dieses zu nutzen und ein einheitliches Verfahren zu Entwickeln, welches jedem ermöglicht Dienste anderer Anbieter/Programme zu nutzen.

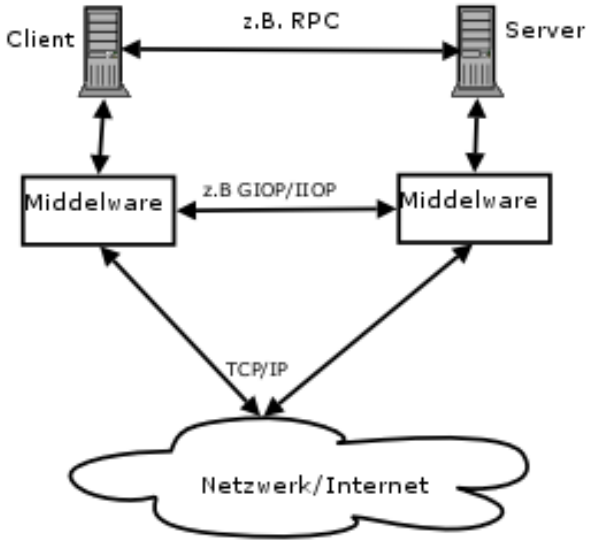

Abbildung 3.3.: Middelware einfach

Webdienste beruhen auf XML. Alle Protokolle setzten auf XML auf, so wie SOAP, WSDL, UDDI. Hierbei dient SOAP als Kommunikationsprotokoll, welches XML Dokumente austauscht und darüber hinaus auch die Möglichkeit von RPC (Remote Procedure Call) bietet. HTTP/SMTP/FDP dient hier als Transportprotokoll und UDDI dient zum Auffinden des Service. Eine Grundlegende Beschreibung des Service findet man dann unter WSDL (sozusagen die IDL des Webservice). Gehen wir nun kurz auf die Protokolle eines Web Service ein.

3.3. SOAP(Simple Object Access Protocol)

> «SOAP ist ein leichtgewichtiges Protokoll, dessen Zweck der Austausch strukturierter Information in einer dezentralisierten verteilten Umgebung ist.» [4]

SOAP ist sozusagen das Herzstück der Web Services. Es beschreibt auf Basis von XML den Austausch von Nachrichten. Da diese Protokoll komplett XML - Basierend ist wird es auch oft XML Protocol (XMLP) genannt. Der Vorteil ist, dass SOAP durch die Eigenschaft von XML komplett auf HTTP aufsetzten kann und somit leicht im Internet einsetzbar ist. Es ist auch möglich SOAP auf andere Transportprotokolle wie SMTP aufzusetzen.

SOAP liegt zurzeit in der Version 1.2 vor, die am 24.Juni 2003 vom WC3 (www.w3c.org/2000/xp/Group/)[14] standardisiert wurde. Die Arbeitsgruppe hierfür heisst *XML Protocol Working Group.*

3.4. Die SOAP Nachricht

Die SOAP Nachricht ist grundlegend in 3 Teile unterteilt. Dem *SOAP Envelope*, ein virtueller Umschlag. Dieser Umschlag umfasst dann 2 weiter Teile, dem optionalen *SOAP Header* und dem *SOAP Body*. Der Aufbau ist in Abbildung 3.2 abgebildet.

Abbildung 3.4.: SOAP Nachrichtenformat

SOAP Header

Im SOAP Header stehen Informationen, die nicht als Anwendungsdaten dienen. Es sind allgemein gesagt die Metainformationen der Nachricht. Im Header wird meist grundlegend angegeben, wie die Nachricht zu verarbeiten ist. Es wird nicht vorgeschrieben, wie genau dieser Header aussehen soll, doch es ist vorgeschrieben, dass bestimmte Attribute vorhanden sein müssen, wenn ein Header vorhanden ist. Zu diesen Attributen gehören *role, mustUnderstand* und *relay* .

Das role Attribut

Wenn das role Attribut vorhanden ist, kann es drei Werte annehmen. Grundlegend beschreibt diese Attribut, ob der Header für diesen Knoten relevant ist oder nicht.

Folgende Werte kann dieses Attribut annehmen:

- http://www.w3.org/2003/05/soap-envelope/role/none
- http://www.w3.org/2003/05/soap-envelope/role/next
- http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver

Es ist jedoch möglich weitere Rollen in der Anwendung zu definieren. Wie eine Anwendung eine bestimmte Rolle annehmen kann wird im SOAP - Standard nicht beschrieben, dies soll in der Anwendung selber geschehen.

Bei der Rolle *next* muss jeder Knoten, der die SOAP - Nachricht empfängt, den Header interpretieren. Ist die Rolle *ultimateReceiver* angegeben, ist nur der entgültige Empfänger dazu berechtigt. Die Angabe einer Rolle ist optional, d.H. bei keiner Angabe ist automatisch immer der letzte Empfänger gemeint.

Das mustUnderstand Attribut

Das mustUnderstand Attribut dient zur Verstärkung der Bedeutung eines Headers. Ist dieses Attribut vorhanden und auf «true» gesetzt und der Knoten anhand seiner Rolle berechtigt, den Header zu interpretieren, so muss er die Semantik des Headers komplett verstehen, bevor er die Nachricht weiterleitet.

Das relay Attribut

Das relay Attribut gibt schließlich an, was mit dem Headerblock geschehen soll, welcher eigentlich von einem SOAP - Knoten bearbeitet werden soll. Hier kann man angeben, ob er weitergeleitet wird oder nicht. Das hängt ganz von der Rolle des Kontens ab und ob das relay Attribut auf tru gesetzt wurde.

Mit diesen Eigenschaften des SOAP-Nachrichtenmodells, kann man die Verarbeitung sehr flexibel angeben. Tiefer werde ich jetzt nicht in SOAP einsteigen.

Daher kommen wir jetzt zum nächsten wichtigen Protokoll.

3.5. WSDL - Web Service Description Language

Die WSDL ist eine Beschreibungssprache, wie sie in jedem Middelwareansazt vorkommt. Im Grunde dienen diese Sprachen nur der Beschreibung der Schnittstellen. Man könnte in ihnen auch die Semantik beschreiben, doch muss man das ? Der Client möchte ja nur wissen, wie ein Dienst genutzt wird und welches Ergebnis dieser liefert, wie dies geschieht kann ihm egal sein. Es könnte auch Vorteile mit sich bringen, wenn man über die Semantik des Dienstes bescheid wüsste, z.B. im Sinne von Dienstfindung. Doch im Bereich von Web Services ist dies unnötig.

Was gehört zu einer Dienstbeschreibung ?

Im Grunde erst mal der Name (lokal also der Methodenname oder der Objektname) unter dem der Dienst bekannt ist, die Eingabeparameter, falls vorhanden und natürlich der Rückgabewert, falls vorhanden. Eine der bekanntesten Beschreibungssprachen ist wahrscheinlich die CORBA-IDL (Interface Definition Language). Die WSDL weicht aber ein wenig von der bekannt Form ab, da in ihr nicht nur eine Servicebeschreibung enthalten ist, sonder auch gleich eine Angabe über die Realisierung, welche bei den anderen Beschreibungssprachen nicht enthalten ist und teil der Anwendung war.

Die WSDL entstand aus einer Initiative von Microsoft, IBM und Ariba. Es gibt keinen offiziellen Standard wie bei SOAP, doch ist man sich über die wichtigen Teile einig. Die W3C hat ein sogenanntes *«Working Draft»*in der Version 1.2 herausgebracht. In diesem Paper kann man nachlesen, welche Anforderungen an die WSDL gestellt werden, wie z.B. eine XML Schema Unterstützung.

Aufbau der WSDL Datei

```
<definitions targetNamespace="xs:anyURI">
   <documentation />?
   [<import/> | <include/> ]*
   <types />?
   [<message /> | <interface /> | <binding/> | <service />]*
</definitions>
```

So wird die Struktur im Working Draft vom W3C angegeben.

Im <*definitions*> Bereich muss ein targetNamespace angegeben werden, mit welchem der Web Service identifiziert wird. Dazu kommen dann noch alle benutzten Namespaces der Bindungen, falls welche vorhanden sind. Diese Angabe ist unbedingt nötig, da ja mindestens die WSDL Bindung benutzt wird.

Beispiel:

```
<wsdl:definitions
    targetNamespace="http://www.der-web-service.de/dienst/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    smlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    .....>
```

Unterhalb des <*definitions*> Elements kann null oder ein <*documentation*> Element, null oder ein <*types*> und null oder mehrere Elemente von <*message*>, <*interface*>, <*binding*> *und* <*service*> enthalten sein. <*documentation*> wird aber häufig vernachlässigt.

Im <*types*> Element werden, wie der Name schon sagt, alle Datentypen definiert, die in einer Nachricht benutzt werden. Da es auch möglich ist, komplexe Datentypen zu verwenden, sollten auch diese im <*types*> Element definiert werden. Dazu bietet sich das XML - Schema an.

Da ein Web Service meist aus mehreren Operationen (Nachrichten) besteht (z.B. Anfrage: Rechne 3 + 2, Antwort: 5) müssen auch diese definiert werden. Dies geschieht im <*message*> Element. Dabei kann jede Message aus keinem oder mehreren <*part*> Element bestehen. In ihm können einfache Datentypen enthalten sein oder eben die komplexen Datentypen, welche man im <*types*> Element definiert hat. Im <*message*> Element kann man somit alle Nachrichten definieren, die über die im <*definitions*> Element eingebundenen Bindungen ablaufen, sei es SOAP oder wie im obigen Beispiel HTTP.

Kommen wir nun zum <*interface*> Element. Ein Web Service kann über mehrer Interface verfügen, die wiederum mehrer Operationen beinhalten. Eine Operation besteht meist aus

mehreren Nachrichten (siehe oben). Die Abfolge dieser Nachrichten wird über ein *pattern* Attribut festgelegt. Diese Pattern werden wiederum über eine URI identifiziert:

- *In-Only* (http://www.w3.org/2003/06/wsdl/in-only): Dieses Pattern hat nur einen input ohne eine Antwort vom Web Service
- *In-Out* (http://www.w3.org.2003/06/wsdl/in-out): Hier folgt auf den Input eine Reaktion in Form einer Nachricht oder Fehlermeldung
- *Request-Response* (http://www.w3.org/2003/06/wsdl/request-response): Genau wie In-Out, nur erfolgt die Antwort auf dem selben Kanal, auf dem die Input Nachrich kam
- *In-Multi-Out* (http://www.w3.org/2003/06/wsdl/in-multi-out): Dem Input folgt keine, eine oder mehrer Antworten oder Fehlermeldungen
- *Out-Only* (http://www.w3.org/2003/06/wsdl/out-only): Es gibt kein Eingabe sonder nur eine Ausgabenachricht
- *Out-In* (http://www.w3.org/2003/06/wsdl/out-in): Zuerst erfolgt die Ausgabenachricht, woraufhin eine Eingabenachricht folgt. Der Web Service kann so z.B. eine Eingabe anfordern
- *Out-Multi-In* (http://www.w3.org/2003/06/wsdl/out-multi-in): Wie Out-In, nur dass mehrere Eingabenachrichten/Fehlermeldungen folgen
- *Multicast-Solicit-Response*(http://www.w3.org/2003/06/wsdl/multicast-solicit-response): Es folgt auf eine Ausgabenachricht eine Eingabe und daraufhin evtl. wieder eine Ausgabenachricht

Bei WSDL bedeutet die Bindung genau wie bei SOAP, die Assoziierung eines Datenformats mit den Datentypen einer Nachricht. Also hier die Operationen eines Interfaces. Dazu dient das <*binding*> Element. In Ihm kann man spezifische Dinge angeben, wie z.B. die Nachricht der Operation übertragen wird, also HTTP oder andere Protokolle.

Als letztes wird über das <*service*> Element beschrieben, über welche Wege der Service ansprechbar ist. Also werden hier nur die Adressen angeben, da es evtl. möglich ist eine Bindung über mehrere Endpunkte zu erreichen, z.B. über SOAP, HTTP-Get usw.

Kommen wir nun zum finden eines Web Service. Dies geschieht über UDDI.

3.6. UDDI Universal Description, Deiscovery and Integration

Mit den vorherigen Protokollen könnte man jetzt eine Web Service aufbauen und benutzen. Dazu müssten sich die Partner aber vorher austauschen und beschreiben, wo ihre Dienste zu finden sind und wie man sie benutzt. Sieht man aber den Web Service als etwas Globales, so muss es eine Möglichkeit geben, seinen Dienst der Öffentlichkeit zugänglich zu machen. Genau dieses übernimmt UDDI.

> «UDDI wird von Dienstanbietern verwendet, um Services bekannt zu machen, und von Dienstnutzer, um eine den speziellen Anforderungen genügenden Service zu finden.» (Eberhart und Fischer, 2003: 299)

UDDI ist nichts anderes als ein Verzeichnisdienst, vergleichbar mit dem Telefonbuch. Es bring Objekte mit verschiedenen Eigenschaften in Verbindung. Z.B. der Name im Telefonbuch, der über eine Adresse (Telefonnummer) erreichbar ist.

Jeder Verzeichnissdienst hat im Grunde die gleichen Schnittstellen, die sich in Ihm wiederfinden. Dazu gehören Funktionen wie das *binden, rebind, delete (unbind), lookup und list* .

Auch UDDI wurde von Microsoft, IBM und Ariba im Jahre 200 ins Leben gerufen. Zur Zeit wird es von OASIS standardisiert.

UDDI bietet zwei Möglichkeiten an, wie man Web Services suchen kann. Einmal die Maschine-Maschine-Kommunikation, wo ein Web Service angeboten wird, auf denen laufende Anwendungen zugreifen könne um einen Dienst zu finden. Die zweite Möglichkeit ist die Mensch-Maschine-Kommunikation, dort kann der Programmierer über ein Webinterface selber nach Web Services suchen.

UDDI - Kategorien

Welche Informationen erhält man über UDDI ? Im Grunde gibt es 3 verschiedene Kategorien.

- Die *White Pages* Sie sind wie das Telefonbuch: Hier stehen nur die grundlegenden Informationen einer Firma, wie z.B. Telefonnummer, Faxnummer und Adresse
- Die *Yellow Pages* Sie sind vergleichbar mit den Gelben Seiten: D.H. nach Branchen sortiert. Üblicherweise sind Firmen in den White und Yellow Pages vertreten

- Die *Green Pages* Dies sind die Einträge, die für einen Web Service relevant sind. In ihr stehen die Technischen Details eines Web Services einer Firma. In ihr stehen also die URI des Service, die WSDL-Beschreibung und noch eine allgemeine Beschreibung.

Auf die Technischen Details möchte ich jetzt erst mal nicht weiter eingehen.

4. Szenario

4.1. Allgemein das Part Live Projekt

Kommen wir zur Vorstellung des Szenarios. An der HAW-Hamburg gibt es ein Forschungsprojekt, welches einen Ferienclub simuliert [9]. In diesem Ferienclub soll jeder Gast ein portables Gerät bekommen (PDA, Handheld usw.) welches er am Ende seines Aufenthalts wieder abgibt. Über dieses Gerät kann der Gast dann Kurse buchen, Bezahlungen tätigen oder er wird permanent informiert über Ereignisse, die Ihn interessieren könnten. Durch die Ausstattung des Ferienclubs mit Accespoints können sogar ortspezifische Informationen an den Gast weitergegeben werden. Wie z.B. «Hier findet in 3 Stunden ein Yoga - Kurs statt....wollen Sie mitmachen ?» Die Vorstellung einer Architektur kann man in der Studienarbeit von Lars Mählmann («Sichere Übertragung im WLan mit mobilen Endgeräten»,Absatz 4, Seite 19, Abbildung 4.2) sehen, ich beschäftige mich mit einer vereinfachten Variante, siehe Abbildung 4.1.

Im Rahmen dieses Projektes fallen viele Diplomarbeiten und Studienarbeiten an, welche sich mit Agentensysteme, Embedded Systeme, Verteilte Systeme, Handhelds usw. beschäftigen. So fiel auch meine Studienarbeit dort hinein. Da es grundlegend Möglich ist, auch in .NET einen Webserver von Grund auf zu Programmieren (wie es auch in JAVA der Fall ist) beschränkt sich die Arbeit auf die Umgebung Visual Studio .Net und .Net, welches ein Framework (ASP.NET) zur Verfügung stellt, die viele Dinge erleichtern soll. Und genau dieses wird untersucht.

4.2. Die Idee, Autovermietung

Die Idee ist, dass der Ferienclub dem Gast die Möglichkeit bietet, Leihwagen zu bekommen um Ausflüge zu tätigen. Diese Leihwagen sollen normal über eine Autovermietung beschafft werden, nur tritt der Ferienclub dem Gast gegenüber als Vermieter auf. Der Gast soll also über eine Webseite oder einem anderem Client die Möglichkeit bekommen, einen Wagen zu bestellen ohne diesen bei einer Vermietung ab zu holen. Alles wird vom Ferienclub geregelt. Und hier kommen Web Services zum Einsatz. Um ein aktuelles Angebot über Fahrzeuge

zu ermöglichen, greift der Ferienclub auf den Web Service einer Autovermietung zu, wo er erfahren kann, welche Fahrzeuge wann zur Verfügung stehen. Der Gast kann sich ein Fahrzeug auswählen und der Ferienclub bucht dann diesen Wagen über den Web Service.

Szenario

Als ersts wird ein einfacher Webservice entwickelt, welcher sich fiktive Daten einer imaginären Autovermietung besorgt und diese Daten dann an einen Webclient weiterleitet.

Über diesen Webclient kann der Urlauber sehen, ob sein Wunschauto verfügbar ist und diesen dann auch gleich mieten. Siehe Abbildung 4.1.

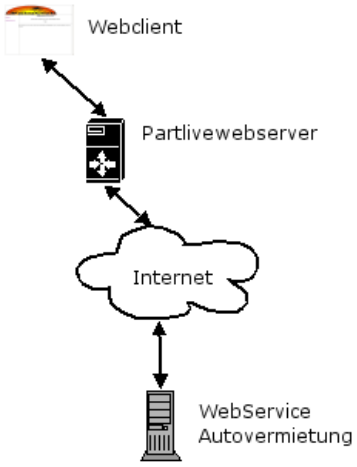

Abbildung 4.1.: Szenario 1

Somit ist es erst mal eine B2B Lösung, in der unser Ferienclub dem Kunden ein Webportal anbietet welches dann auf einen Webservice einer Autovermietung zugreift.

Der Urlauber sollte die Möglichkeit haben, eine Kategorie und ein Wunschtermin auszuwählen, also z.B. Cabrios am 05.5.200. Er bekommt dann eine Info, ob in dieser Kategorie noch Autos frei sind. Möchte er dann ein Auto bestellen, so soll er die Möglickeit haben diesen direkt zu buchen. Da dies ein exemplarischer Dienst ist, verzichte ich auf die Anmeldung und Authentifizierung des Urlaubers. Wir gehen davon aus, dass dies schon über das Webportal des Ferienclubs geschehen ist. Auch die Sicherheitsaspekte des Webservice vernachlässige ich hier mal, da dies nicht Bestandteil meiner Arbeit ist.

Als Grundlage wurden Webseiten der gängigen Autovermietungen wie Sixt, Avis und Hertz angeschaut.(Abbildung 4.2 - 4.4) angesehen.

Alle Vermietungen bieten

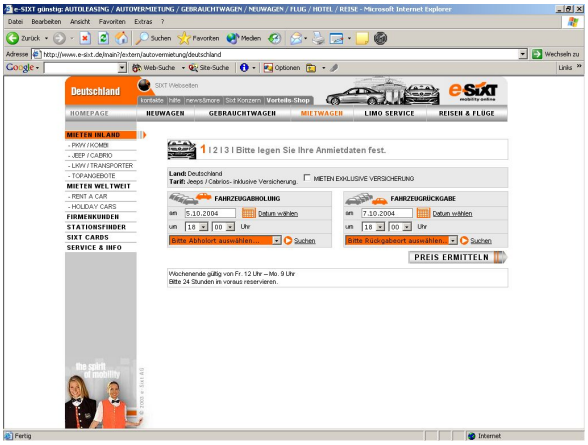

Abbildung 4.2.: www.sixt.de

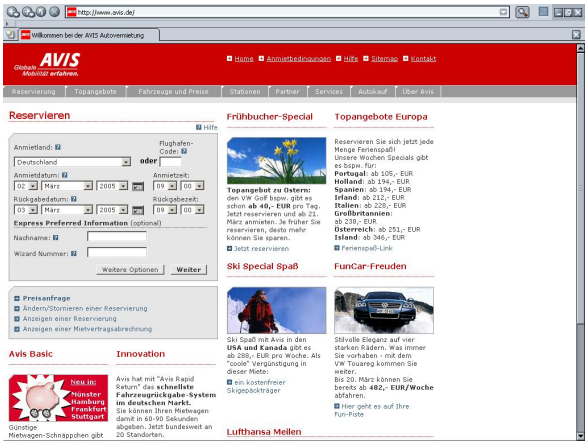

Abbildung 4.3.: www.avis.de

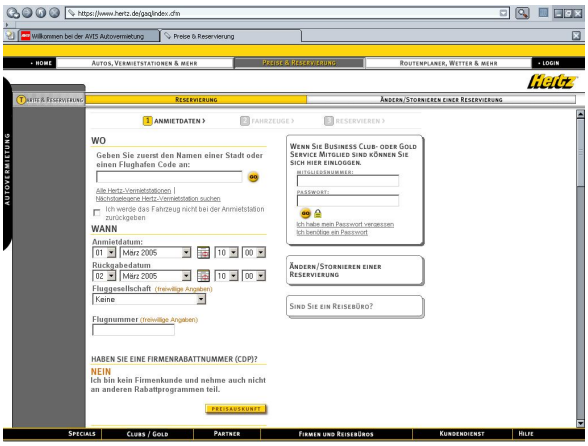

Abbildung 4.4.: www.hertz.de

- Auswahl des Abhol- und Rückgabeort
- Auswahl des Abhol- und Rückgabedatum
- Auswahl eine Kategorie/Fahrzeugtyp (die aber bei jeder Autovermietung individuell Bezeichnet wird) z.B. Sixt gibt Kategorien wie «Jeep/Cabrio» oder «PKW/Kombi» an. Bei Hertz sieht es eher nach einer Staffelung des Preises aus, woraus sich dann ein Fahrzeugtyp ergibt.
- Errechnung des Preises und entgültiges Buchen.

Da wir im Ferienclub sind, lasse ich den Abhol- und Rückgabeort wegfallen.

4.3. Entwicklungsumgebung

Da die Eignung von .Net für Webservices untersucht werden soll, wurde entschieden, die ReferenzIDE *Visual Studio .Net 2003* zu benutzen. Visual Studio .Net 2003 bietet ein Rundumsorglospaket zur Entwicklung von Windows basierten Anwendungen. Es bietet für die Entwicklung von Oberflächen (Html-Seiten, Windows Forms) eine einfache Drag and Drop Lösung an. Trotzdem ist es möglich, diese auch per Hand selber zu setzten und zu entwickeln. Für alle spezifischen Anwendungen in der Windows Welt, bietet diese Umgebung Wizards an, um diese schneller und einfacher zu erstellen. So kann man bei Projektbeginn schon wählen, ob es eine Webanwendung, Dosanwendung usw. wird. Die IDE bereitet dann alle spezifischen Dinge vor. Desweiteren bietet diese Umgebung alle möglichen Features, die ein Entwickler so braucht, wie Codhighlightning, automatische Klammersetzung, Teamunterstützung für Versionsmanager usw. Die Entscheidung fiel auf Visual Studio, da diese als Referenzide angepriesen wird und dort die grösste Funktionsvielfalt und Unterstützung für .Net gegeben ist. Natürlich ist es jedem freigestellt, eine andere Entwicklungsumgebung zu wählen, die z.B. C# unterstützt. Zu beachten sind auch die hohen Liezenkosten für diese Version. Die Enterprise Developer Edition z.B. kostet als Upgrade US $1.079. Alls Vollversion sogar US $ 1.799 [1].

4.4. ASP .NET

> «ASP.NET provides a programming model, and infrastructure, to make creating scalable, secure and stable applications faster, and easier than with previous Web technologies.» [1]

ASP .NET liegt derzeit in der Version 2.0 vor. Es ist das Framework von Microsoft, welches alle Web spezifischen Pakete beinhaltet. Es bietet eine grosse Klassenbibliothek und ist gegenüber dem alten ASP, was eher eine Sprache wie JSP war, ein komplettes Framework geworden, welches Klassen für Web Services, Html Server Controls, XML, Mobile ASP usw. anbietet, um die Entwicklung von Webanwendungen zu erleichtern.

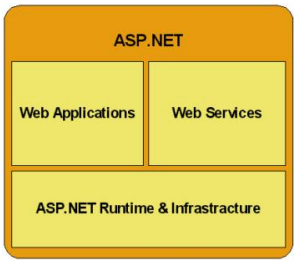

Abbildung 4.5.: ASP .NET Überblick

Es bietet die Unterstüzung verschiedener Sprachen, auf die man zurückgreifen kann, wie z.B. VB.NET, C#, und JScript.NET.

5. Analyse

5.1. Erwartung

Die Erwartung eines Entwicklers können unterschiedlicher nicht sein. Die einen habe bereits Erfahrungen in Web Services sammeln können (evtl. im Bereich JAVA), kennen sich somit in der Materie aus. Andere Entwickler haben evtl. viel Erfahrung in Middelwaresystemen wie Corba oder der Gleichen und möchten einfach eine Alternative kennen lernen. Meistens wird es aber in der heutigen Wirtschaft so sein, dass man meist überrascht wird und sich in ein Thema, welches man vorher nie gesehen hat, einarbeiten muss, da dieses von einem Kunden verlangt wird. Ich zähle mich zu letzteren, um auch die angepriesene Einfachheit, die Microsoft verspricht zu testen. Die Erwartung ist derart, dass man ohne tiefgehende Erfahrungen schnell ein passables Ergebnis erreicht und wenn möglich sogar einen guten produktiven Einsatz abliefern kann. Dies gilt es zu Untersuchen.

5.2. Ablauf

Einen grundlegenden Web Service in .Net zu erstellen ist relativ simple (siehe Anhang How-To). Um einen grundlegenden Ablauf zu generieren muss eine gewisse Verallgemeinerung her. Wie in Kapitel 4.2 angesprochen, ist der Ablauf, um ein Auto zu mieten, bei allen Autovermietungen relativ gleich. Der Unterschied zwischen den Autovermietungen besteht meist nur in der Namensgebung der Kategorien von Fahrzeugen (falls eine überhaupt vorhanden ist). Daher hat der Beispiel WebService eine Methode *getKategories()* welches es ermöglicht, die Vorhandenen Kategorien aus zu lesen. Beim Aufrufen der Seite wird diese Methode von unserem Client aufgerufen, so dass eine Wahl des Angebotes zur Verfügung steht. Weitergehend wird dann anhand der Kategorie das Auto ausgewählt, welches gewünscht wird. Mit den gelieferten Daten und dem Wunschtermin kann dann über den Service festgestellt werden, ob dieses Auto zur Verfügung steht oder nicht und was es dann kostet. Dann braucht man nur noch buchen.(Abbildung 5.1)

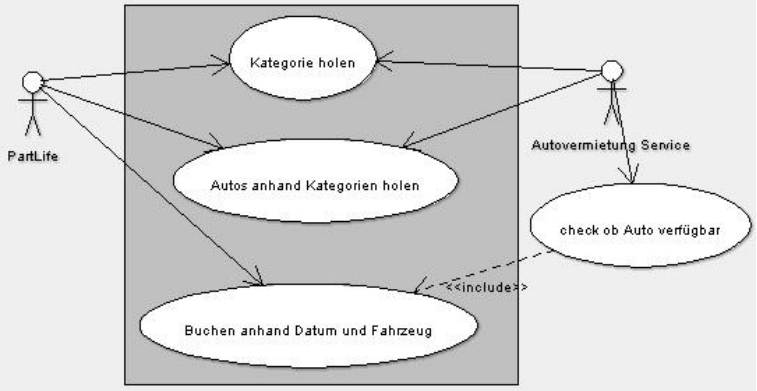

Abbildung 5.1.: Grober Ablauf

5.3. Implementierung

Visual Studio bietet eine einfache Möglichkeit, eine Methode als Servicemethode zur Verfügung zu stellen. Man deklariert diese einfach als *[WebMethod]* .

```
[WebMethod]
public string[] getKategories(){
```

Diese Methode z.B. liefert ein einfaches String-Array zurück, welches alle Verfügbaren Kategorien beinhaltet. Somit muss die Anwendung, die diesen Service nutzt, keine eigenen Kategorien definieren. Allein mit der Kennzeichnung *[WebMethod]* erstellt die .Net Umgebung im Hintergrund die korrespondierende WSDL Datei und XML - Mappings (siehe Anhang B).

Natürlich würde eine solche Methode auch schon funktionieren, doch für den produktiven Einsatz reicht der Standard meist nicht aus. Daher gibt es noch verschiedene Eigenschaften, die dem Attribut *[WebMethod]* mitgegeben werden kann. Darunter fallen z.B.

- BufferResponse = soll dien Antwort an den Client gepuffert werden ?
- CachDuarion = Zeit in Sekunden, die eine Antwort vom Webserver im Speicher gecacht wird.
- EnableSession = Angabe, ob der ASP.NET-Sitzungsdienst für die Implementierung der Methode verfügbar sein wird
- MessageName = Name der Methode, die vom Webdienst verwendet wird. Hiermit werden dann auch die WSDL Präfix der Elemente message, input und output ändern
- TransacitonOption = Angabe, der Transaktionsunterstützung für diese Methode.

ASP.Net bietet auch die Möglichkeit auf Klassenebene ein *[WebService]* Attribut an zu geben. Diesem Attribut kann man auch Eigenschaften zuweisen, welche im WSDL - Dokument des Services wiedergegeben werden:

- Description = Gibt das Element *description* unter dem Element *service* im generierten WSDL-Dokument an.
- Name = Name des *service* Elements im WSDL-Dokument, sowie das Präfix für die Namen der Element *protType, binding* und *sort* .
- Namespace = Hier wird der Zielnamespace für das WSDL-Dokument definiert. Sollte immer angegeben werden, damit der Service auffindbar wird. Der default Namespace ist *tempuri.org* .

Wenn man jetzt eine bessere Dokumentation dieser Methode oder des WebDienstes haben möchte, kann man dieses so realisieren:

```
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;

namespace PartLiveAutoService
{
/// <summary>
/// Zusammenfassung für Service1.
/// </summary>

[WebService(Description="Ein Testwebservice für Partlive.")]
public class PartLiveAuto1 : System.Web.Services.WebService
{....

[WebMethod(Description="Liefert die verfügbaren Autokategorien.")]
public string[] getKategories(){
....
}
```

Serialisierung

Da ich dem *[WebService]*Attribut noch keinen Namespace zugewiesen habe, wird derzeit der Defaultnamespace *http://tempuri.org/* verwendet. Dies ist für den Produktiven Einsatz nicht zu Empfehlen, sogar sehr schlecht. Doch wenn man in Visual Studio Entwickelt, kann

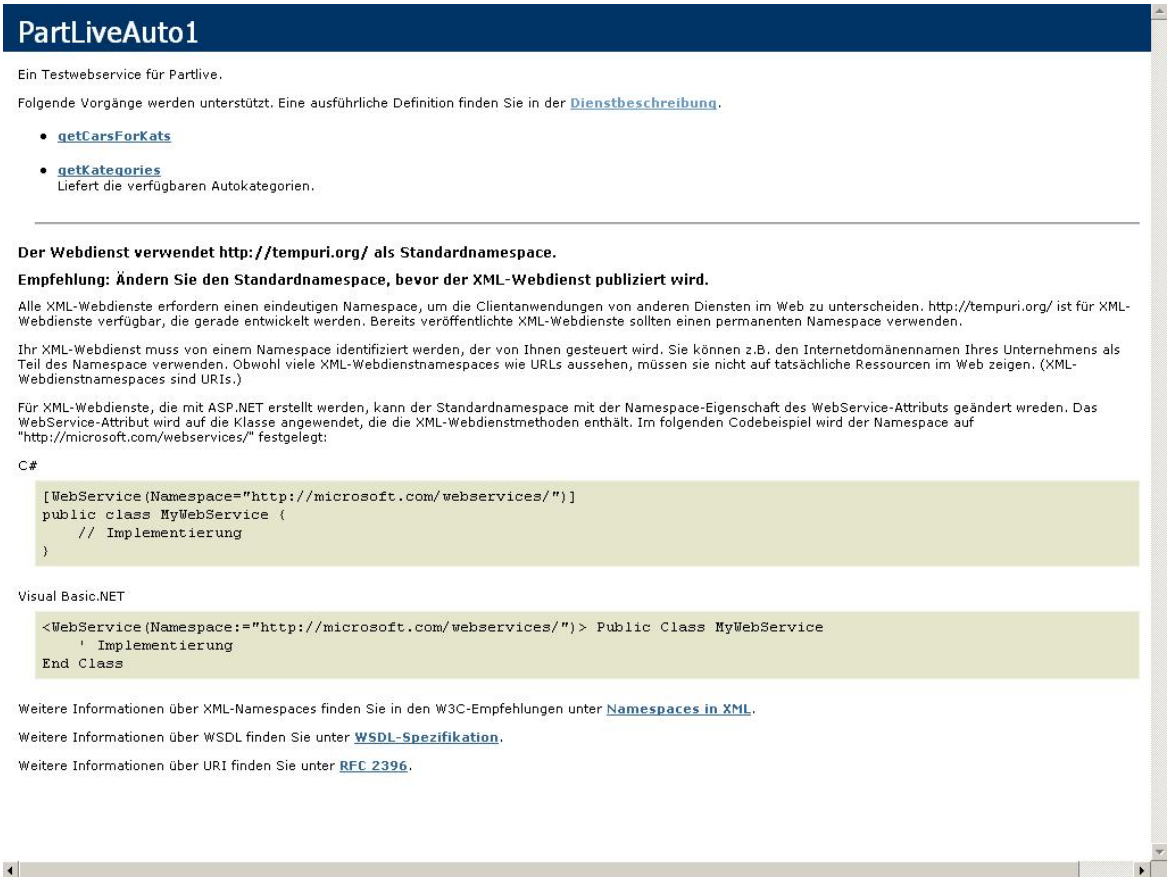

Abbildung 5.2.: Testen des Services

man so sehr gut den Webdienst testen, denn durch diesen Defaultnamespace öffnet Visual Studio beim Ausführen des Service eine selbst generierte html - Seite (Abbildung 5.1), die einen diskret darauf aufmerksam macht, dass man doch den Namespace ändern solle und einem die Möglichkeit bietet, seinen Webservice zu testen. Hier ist es auch möglich, den ausgelieferten XML-Code anzuschauen.

```
<?xml version="1.0" encoding="utf-8" ?>
 <ArrayOfString xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://tempuri.org/">
  <string>Jeep/Cabrio</string>
  <string>PKW/Kombi</string>
  </ArrayOfString>
```

Wie man in Abbildung 5.2 sieht, steht unter der Methode *getKategories()* die Beschreibung, die dem *[WebMethod]* Attribut gegeben wurde. Dieses findet sich auch im WSDL - Dokument wieder, zusammen mit der Service Beschreibung.

Um den Namespace zu ändern, damit der Service eindeutig zu identifizieren ist und alle Typen, die innerhalb des Services definiert werden über den Namespace erreichbar sind, muss man dem *[WebService]* Attribut den Wunschnamespace übergeben.

```
[WebService(Description="Ein Testwebservice für Partlive.",
Namespace="www.autoservice.de")]
public class PartLiveAuto1 : System.Web.Services.WebService{
...}
```

Es ist auch möglich, den Webservice nicht von *System.Web.Services.WebServices* ableiten zu lassen. Dann muss man aber auf die Standardobjekte wie :

- *HttpContext*
- *HttpServerUtility*
- *SessionState*

direkt zugegriffen werden, da diese nicht mehr über die Basisklasse definiert sind. Das ist erst mal mehr Arbeit, aber dadurch ist es auch möglich, eine komplexere Objektstruktur des Services zu erstellen. Auf die Erstellung einer solchen komplexen Struktur habe ich erstmal verzichtet, da dies zwar für den produktiven Einsatz empfehlenswerter ist, aber für die Testzwecke ist dies erst mal unrelevant und ich leite weiter von *System.Web.Services.WebServices* ab. Diese Objekte werden beim Client benötigt um sich die Proxyklasse zu erstellen.

Objekt übergeben

Möchte man jetzt nicht nur simple Typen übergeben, sondern Objekte, so kann man dies recht einfach realisieren:

```
public class Car{
public String type;
public double price;

public Car(){
```

```
type = "";
price = 0.0;

}

public Car(String strType, double dPrice){

this.type = strType;
this.price = dPrice;
}
}

[WebService(Description="Ein Testwebservice für Partlive.",Namespace="w
public class PartLiveAuto1 : System.Web.Services.WebService
{

[WebMethod]
public Car getCar()
{
return new Car("Mercedes SLK",25.0);
}

}
```

Diese Klasse wird auch im WSDL wiedergegeben.

```
- <s:complexType name="Car">
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="1"
   name="type" type="s:string" />
  <s:element minOccurs="1" maxOccurs="1"
   name="price" type="s:double" />
  </s:sequence>
  </s:complexType>
```

Doch was ist, wenn man Objektnamen und Eigenschaften hat, die man dem Client gegenüber verbergen möchte [11]. Um dies zu realisieren muss man die Klasse, hier CAR, XML - Attribute mit übergeben, damit der XML-Serialisierer diese auch in der WSDL - Datei ändert. So ist es möglich, die Proxyklassen des Clients besser auf zu bauen.

```
using System.Xml;
using System.Xml.Serialization;


public class Car{

[XmlElement("Auttotyp")]
public String type;

[XmlElement("Tagespreis")]
public double price;
..
```

In der WSDL-Datei, erscheint jetzt nicht mehr, wie oben gesehen *name= «price»* sonder *name= «Tagespreis»* .

SOAP - Codierungsstil

Es ist zwar für diesen Testwebservice relativ unwichtig. Trotzdem soll noch erwähnt werden, dass .NET also SOAP - Stil immer *document* angiebt. Doch wenn man komplexere Services erstellen möchte, die rein auf *RPC* beruhen, muss auch hier eine Anpassung vorgenommen werden. Dies erreicht man dadurch, dass man dem [WebService] Attribut auf Klassenebene noch ein ein Soap Attribut anhängt: *[SoapRpcService]* . Nun werden alle Methoden dieser Klasse mit dem RPC-Stil codiert. Möchte man nun einzelne Methoden nicht im RPC-Stil haben, so kann man auf Methodenebene das Attribut *[SoapDocumentService]* angeben. Im *[SoapDocumentService]* Attribut kann man noch drei Eigenschaften definieren, von dem das *[SoapRpcService]* nur die Eigenschaft *RoutinStyle* unterstüzt. [5]

- *ParameterStyle* = Gibt an, ob Parameter im Text der SOAP-Nachricht in ein einzelnes Element eingehüllt sind
- *RoutingStyle* = Gibt an, ob Http-Header SoapAction ausgefüllt oder frei gelassen werden soll
- *Use* = Gibt an, ob der Standard für den Codierungsstil Literal oder Encoded ist

Den *[SoapDocumentMethod] und [SoapRpcMethod]* Attributen kann man dann auch noch explezite Eigenschaften angeben. *[SoapRpcMehtod]* unterstüzt dabei alle Attribute mit Ausnahme von *ParameterStyle und Use* .

- *Action* = Gibt den URI an, im SoapAction-Http-Header platziert wird.

- *Binding* = Ordnet einer Webmethode eine bestimte Bindung zu, die vom Attribut WebServiceBinding angegeben wird.
- *OneWay* = Gibt an, ob der Client eine Antwort in Verbindung mit der Anfrage erhält.
- *ParameterStyle* = Gibt an, ob die Parameter im Text der SOAP-Nachricht in ein einzelnes Element eingehüllt werden.
- *RequestElementName* = Gibt den Namen des Anfrageelements innerhalb des Textes der SOAP-Nachricht an.
- *RquestNamespace* = Gibt den Namespace-URI an, der die Anfrageelementdefinition enthält.
- *ResponseElementName* = Gibt den Namen des Antwortelements innerhalb des Textes der SOAP-Nachricht an.
- *ResponseNamespace* = Gibt den Namespace-URI an, der die Antwortelementdefinition enthält.
- *Use* = Gibt an, ob der Codierungsstil der Nachricht Literal oder Encoded ist.

[5]

6. Fazit

6.1. Tauglichkeit von .Net

Generell bietet .Net alles, was man benötigt, um einen WebService zu erstellen. Jedoch sollte man nicht denken, dass alles mit dem Attribut *[WebMethod]* erledigt ist. Als Entwickler ohne Vorahnung der Materie, bedarf es doch einer gewissen Einarbeitungszeit. Solang man sich in dem WebService auf primitive Type beschränkt, ist die ganze Kommunikation relativ unkritisch und ein Zugriff von einer anderen Plattform, wie Java, stellt auch weniger Probleme dar. Doch sobald man versucht, ganze Objekte hin und her zu schicken, wird der Konfigurationsaufwand und die Probleme schon erheblich. Jedes Objekt wird über XML- also ComplexType abgebildet und dieses macht es schon schwer beim Empfänger, dieses zu richtig zu interpretieren. In einer reinen .NET Umgebung, d.H. Service und Kunde des Services sind .Net Programme, ist es etwas einfacher, mit Objekten zu arbeiten. Doch wer stellt einen Service zur Verfügung der nicht von jedem genutzt werden kann ? [11]

Microsoft hat für die Entwicklung eine Plattform zur Verfügung gestellt, die es extrem erleichtert, WebServices zu erstellen. Jedoch sind fundierte Kenntnisse des Entwicklers doch noch erforderlich. In Kundenprojekten, die unerwartet aufkommen, muss doch eine gewisse Einarbeitungszeit investiert werden. Gerade der Bereich Plattformunabhängigkeit, durch das Protokoll XML und SOAP bedarf spezieller Betrachtung, da Microsoft doch als Default die .NET Plattform anbietet. Marketingtechnisch ist dies gut zu verstehen, doch gerade WebServices, sind gerade dafür ausgelegt, Plattformunabhängig zu sein. Um dies zu erreichen ist doch mehr Aufwand zu betreiben.

MSDN Library

Die von Microsoft angebotene MSDN Library ist eine riesen Sammlung von Tutorials, Dokumentationen und technischen Artikeln. Es ist vielleicht die grösste Dokumentation einer Plattform die ich persönlich gesehen habe. Doch gerade die Grösse ist auch das Problem, es ist sehr schwer genau die Information auf Anhieb zu bekommen, die man benötigt, zusätzlich ist es sehr unübersichtlich und es kommt oft vor, dass man einen Artikel nicht wieder findet. Manchmal fehlte mir auch die Tiefe in den Artikeln, denn ab und zu möchte man als

Entwickler auch mal hinter die Fassade schauen und sich nicht blind auf etwas gegebenes verlassen. Der Umfang und die Grösse ist dennoch Herausragend (ca. 2GB nach der Installation).

6.2. Visual Studio

Visual Studio ist die erste Wahl bei der Entwicklung von .Net Anwendungen. Im Vergleich zu älteren Versionen hat Microsoft viel getan und die gesamte Entwicklung Transparenter gestaltet. Die Probleme, die es früher gab, dass man den Code hinter den Forms nicht sehen konnte sind ausgemerzt. Man kann sehr schnell Oberflächen erstellen und sich gleich den Code anschauen um Feintunings vor zu nehmen. Der Erstellt Code ist übersichtlich und gut lesbar. (Man denke an die unübersichtlichen html Seiten von FrontPage). Doch wo auch Licht ist, ist auch Schatten. Die Dokumentationen sind sehr unübersichtlich und verwirrend. Man kann schon mal Stunden damit zubringen, in der MSD Library zu stöbern um ein Thema oder einen Tipp zu finden. Wenn man diesen dann nicht gleich in seinen Favoriten speichert, ist es oft sehr mühsam , diesen wieder zu finden. Zusätzlich ist mir aufgefallen, dass sehr viel nur an der Oberfläche erklärt wird und wenn man hinter die Kulissen schauen möchte, was Visual Studio alles macht, hält sich Microsoft sehr bedeckt. Vielleicht wollen Sie sich nicht in die Karten schauen lassen, doch muss man sich oft auf bestimmte Dinge verlassen und weiss meist nicht, wie etwas funktioniert, es funktioniert einfach. Sehr schlecht war die Möglichkeit Dateien in sein Projekt hinzu zu fügen. Der Browser war dermassen langsam, dass ich schon mal 2 - 3 Minuten warten kann, bis ich durch die Verzeichnisse gebrowst bin um die Datei aus zu wählen. Dies könnte durch schneller Rechner evtl. nicht auftreten, da Visual Studio schon ein Ressourcenfresser ist (komplette Installation > 2 GB). Ich empfehle mind. 1 GB Arbeitsspeicher.

6.3. JAVA vs. .NET

Da ich aus der Java Welt komme, stellt sich auch die Frage, wie der Umstieg auf das .Net Framework zu meistern ist. Grundlegend gibt es bei der Wahl der Sprache keine Probleme, da gerade die Sprache C# sehr Java nahe ist und der Umstieg schnell und einfach zu meistern ist. Die kleineren Abweichungen sind schnell zu erlernen. Doch als Entwickler sollte es generell keine Probleme geben, sich in eine neue Sprache ein zu arbeiten. Wo etwas mehr Zeit investiert werden muss, ist die Benutzung der Objektbibliothek, die sich schon stark unterscheidet. Die Erfahrung hat gezeigt, dass es mehr Zeit gekostet hat, sich in das Thema WebServices ein zu arbeiten, als dieses dann auch in C# um zu setzten. Generell kann man

alle Erfahrungen in Objektorientierung sofort anwenden und kam schnell zum erwünschten Erfolg. Für eine Firma, die einen Wechsel von Java nach .Net plant, ist der Aufwand der Entwickler recht gering gehalten. Etwas Zeit für die Bibliothekenkenntnis muss investiert werden, doch dann kann schon produktiv gearbeitet werden. Generell muss aber noch einmal erwähnt werden, dass gerade die Lizenzkosten und die Preise für die Referenzide doch sehr hoch sind. Falls man schon in älteren Microsofttechnologien entwickelt, können die Kosten des Umstieges auf .NET auch höher sein als erwartet. Denn Entwickler, die es gewohnt waren in z.B. Visual Basic zu programmieren, wird zwar ein einfacher Umstieg geboten, mit der Sprache Visual Basic .Net, doch dieses bezieht sich nur auf die syntaktische Ebene. Zu beachten sind dann noch die Kosten für evtl. anfallenden Schulungen, die benötigt werden, den Entwickler in die geänderten Bibliotheken ein zu führen oder sogar die Objektorientierung nahe zu bringen.

6.4. Ausblick

Plattformübergreifend

Das Entwickeln und Testen in einer reinen .NET Umgebung hat gute Ergebnisse geliefert. Dennoch wäre es zu Untersuchen, wie es sich Verhält mit einer fremden Plattform (Java) auf einen in .Net entwickelten Service zu zugreifen.

UDDI und .NET als Client

Kurzzeitig habe ich versucht, einen bestehenden Service einer Autovermietung über UDDI zu finden. Dies führte leider zu keinem Ergebnis, da ich in sämtlichen UDDI Verzeichnissen von Microsoft, IBM usw. keinen Service gefunden habe, den man als Testumgebung hätte nutzen können. Interessant wäre es, genau dieses mal zu Untersuchen und zu sehen, wie man Services von anderen Firmen nutzt und in seinem in .Net entwickelten Programm integriert.

.Net auf anderen Systemen

Lars Mählmann untersucht in seiner Diplomarbeit, in wie weit WebServices in dem Mono Projekt [12] vorangeschritten ist und ob eine Entwicklung von .Net auf Linux Systemen gut

funktioniert. Mono ist eine plattformunabhängige .Net Implementierung. Für genauere Informationen besuchen Sie doch die MONO Seite [12] oder die AIS-Seite der HAW-Hamburg, wo auch ein paar Abhandlungen dazu zu finden sind [2].

A. Anhang

A.1. How To

Um einen schnellen Hello World WebService zu erstellen, ist in Visual Studio nicht viel erforderlich. Vorraussetzung dafür ist, dass der IIS (Internet Information Service), also der Webserver von Microsoft richtig Installiert und Konfiguriert wurde.

Als erstes erstellt man ein neues Projekt und wählt als Projekttyp den *ASP.NET Webdienst* aus und als Projektsprache C Sharp oder Visual Basic .Net. (Abbildung 7.1).

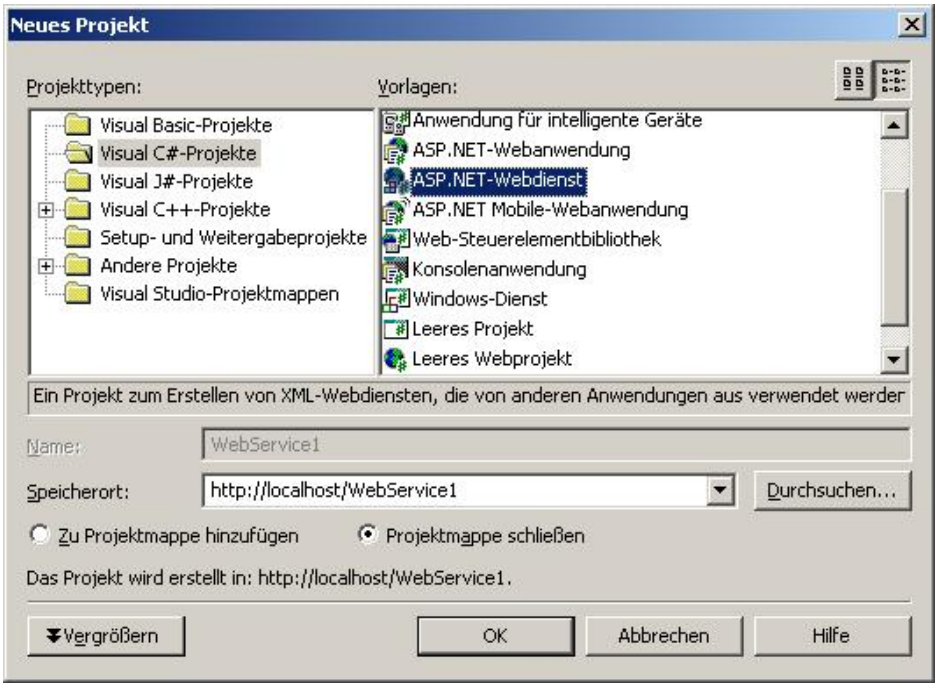

Abbildung A.1.: Auswahl

Man sollte dem Service noch einen sprechenden Namen geben. Dies macht man dort, wo der Speicherort angegeben wird. Z.B. HelloWorldService. Einfach die Namen im Code anpassen (leider macht das Visual Studio dies nicht automatisch). Visual Studio gene-

riert schon vorab eine HelloWorld Methode. Diese muss man einfach Auskommentieren und schon kann man den Dienst Testen (Abbildung 7.2).

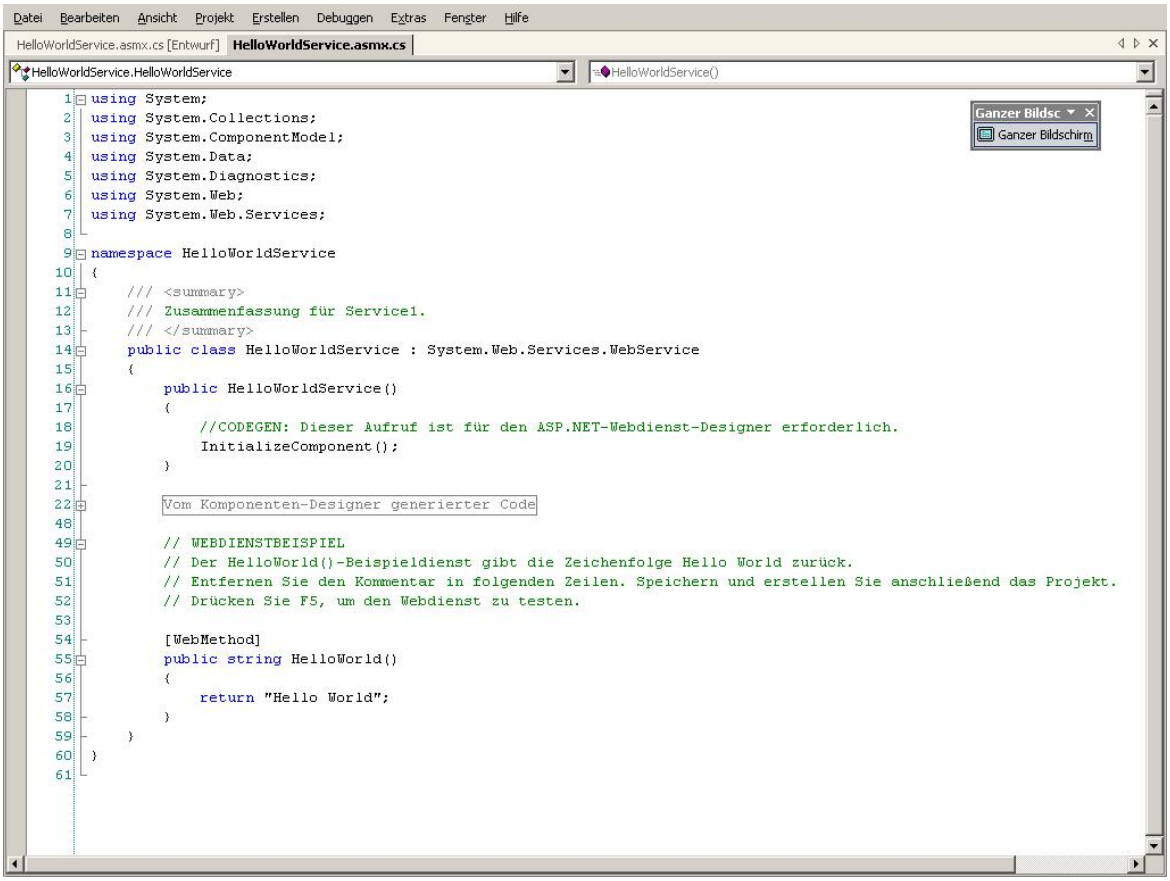

Abbildung A.2.: HelloWorld

Das Ergebnis beim Auffrufen sieht so aus (Abbildung 7.3). Schon kann man loslegen, den Service weiter auf zu bauen.

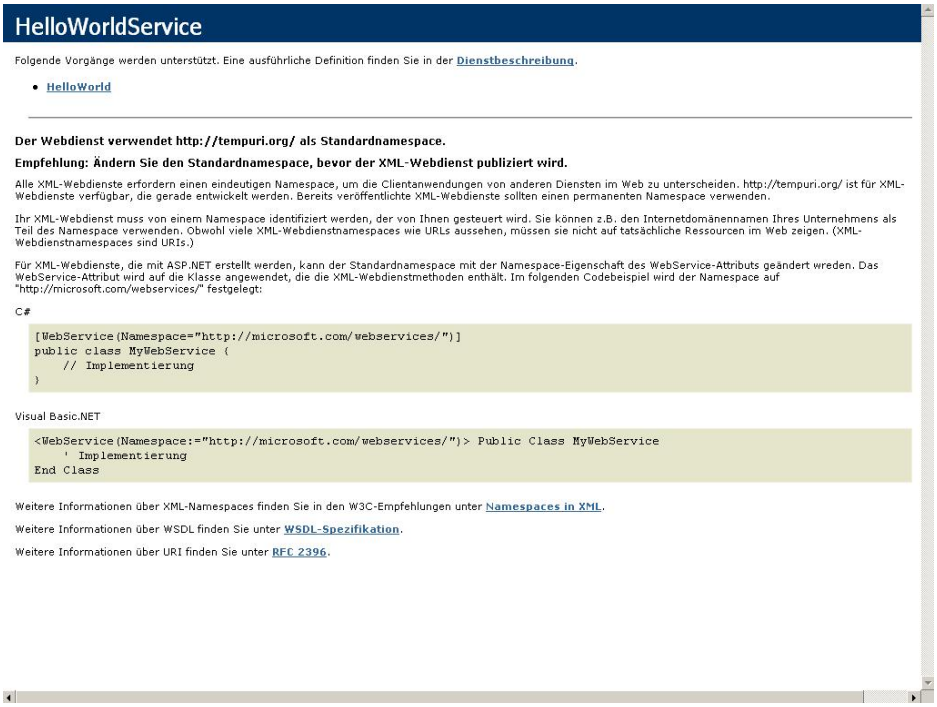

Abbildung A.3.: Aufruf HelloWorld

B. Codebeispiele

B.1. PartLive WebService

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml;
using System.Xml.Serialization;

namespace PartLiveAutoService
{
[WebService
(Description="Ein Testwebservice für Partlive." ,
Namespace="www.autoservice.de")]
public class PartLiveAuto1 : System.Web.Services.WebService
{

public PartLiveAuto1()
{
//CODEGEN: Dieser Aufruf ist für den ASP.NET-Webdienst-
// Designer erforderlich.
InitializeComponent();

}

#region Vom Komponenten-Designer generierter Code
```

```
//Erforderlich für den Webdienst-Designer
private IContainer components = null;

/// <summary>
/// Erforderliche Methode für die Designerunterstützung.
/// Der Inhalt der Methode darf nicht mit dem Code-Editor
/// geändert werden.
/// </summary>
private void InitializeComponent()
{
}

/// <summary>
/// Die verwendeten Ressourcen bereinigen.
/// </summary>
protected override void Dispose( bool disposing )
{
if(disposing && components != null)
{
components.Dispose();
}
base.Dispose(disposing);
}

#endregion

[WebMethod(Description="Liefert die verfügbaren Autokategorien.")]
public string[] getKategories()
{

// Hier würde normalerweise ein Datenbankzugriff stattfinden
string[] strAry = {"Jeep/Cabrio","PKW/Kombi"};
return strAry;
}


[WebMethod]
public string[] getCarsForKats(string kat)
{
string[] cars = new string[3];
```

```
switch(kat)
{
case "Jeep/Cabrio":
cars[0] = "Mercedes SLK";
cars[1] = "BMW 323i Cabrio";
cars[2] = "Opel Tigra Twintop";
break;
case "PKW/Kombi":
cars[0] = "BMW 323i";
cars[1] = "Opel Astra";
cars[2] = "VW Golf";
break;
}
return cars;
}

private bool carIsFree(string car, System.DateTime date)
{
return false;
}

[WebMethod]
public Car getCar()
{
return new Car("Mercedes SLK",25.0);
}

}

public class Car{

[XmlElement("Auttotyp")]
public String type;

[XmlElement("Tagespreis")]
public double price;

public Car(){
```

```
type = "";
price = 0.0;

}

public Car(String strType, double dPrice){

this.type = strType;
this.price = dPrice;
}
}
}
```

B.2. PartLive Webclient

Da diese Anwendung mehrere Html-Seiten mit Frames beinhaltet, ist hier nur der Teil, wo eine Methode vom WebService aufgerufen wird abgebildet:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.HtmlControls;

namespace PartLive
{
/// <summary>
/// Zusammenfassung für auto.
/// </summary>
public class auto : System.Web.UI.Page
{
protected System.Web.UI.WebControls.Label lblKategorien;
protected System.Web.UI.WebControls.Label Label2;
```

```
protected System.Web.UI.WebControls.Label Label1;
protected System.Web.UI.WebControls.Label lblFahrzeugAb;
protected System.Web.UI.WebControls.Panel panHolen;
protected System.Web.UI.WebControls.TextBox txtDatumAm;
protected System.Web.UI.WebControls.TextBox txtUhrVon;
protected System.Web.UI.WebControls.Label lblZurueck;
protected System.Web.UI.WebControls.Panel panBringen;
protected System.Web.UI.WebControls.TextBox txtUhrZurück;
protected System.Web.UI.WebControls.Label lblUhrZurück;
protected System.Web.UI.WebControls.TextBox txtDatumZurück;
protected System.Web.UI.WebControls.Label lblAmZurück;
protected System.Web.UI.WebControls.ListBox lstKat;

private void Page_Load(object sender, System.EventArgs e)
{
this.disableKomponents();

test.PartLiveAuto1 myAuto =
new PartLive.testTheFuk.PartLiveAuto1();

string[] kats = myAuto.getKategories();
ArrayList myList = new ArrayList();
for (int i=0; i < kats.Length; i++){
lstKat.Items.Add(kats[i]);
}

}

#region Vom Web Form-Designer generierter Code
override protected void OnInit(EventArgs e)
{
//
// CODEGEN: Dieser Aufruf ist für den ASP.NET Web
// Form-Designer erforderlich.
//
InitializeComponent();
base.OnInit(e);
}

/// <summary>
```

```
/// Erforderliche Methode für die Designerunterstützung.
/// Der Inhalt der Methode darf nicht mit
/// dem Code-Editor geändert werden.
/// </summary>
private void InitializeComponent()
{
this.lstKat.SelectedIndexChanged += new
 System.EventHandler(this.ListBox1_SelectedIndexChanged);
this.txtDatumAm.TextChanged += new
System.EventHandler(this.txtDatumAm_TextChanged);
this.txtUhrVon.TextChanged += new System.EventHandler(
this.txtUhrVon_TextChanged);
this.Load += new System.EventHandler(this.Page_Load);

}
#endregion

private void ListBox1_SelectedIndexChanged(
object sender, System.EventArgs e)
{
this.enableKomponents();
}
private void disableKomponents(){
panHolen.Visible = false;
panBringen.Visible = false;
}

private void enableKomponents(){
panHolen.Visible = true;
panBringen.Visible = true;
}

private void txtUhrVon_TextChanged(object sender, System.EventArgs e)
{

}

private void txtDatumAm_TextChanged(object sender, System.EventArgs e)
{
```

```
}

}
}
```

B.3. XML

WSDL des Services

```
  <?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="www.autoservice.de"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:tm="htt
targetNamespace="www.autoservice.de"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
 <s:schema elementFormDefault="qualified"
  targetNamespace="www.autoservice.de">
 <s:element name="getKategories">
  <s:complexType />
  </s:element>
 <s:element name="getKategoriesResponse">
 <s:complexType>
 <s:sequence>
 <s:element minOccurs="0" maxOccurs="1"
  name="getKategoriesResult" type="s0:ArrayOfString" />
 </s:sequence>
 </s:complexType>
 </s:element>
 <s:complexType name="ArrayOfString">
 <s:sequence>
 <s:element minOccurs="0" maxOccurs="unbounded"
  name="string" nillable="true" type="s:string" />
 </s:sequence>
 </s:complexType>
```

```
<s:element name="getCarsForKats">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1"
 name="kat" type="s:string" />
</s:sequence>
</s:complexType>
</s:element>
<s:element name="getCarsForKatsResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1"
 name="getCarsForKatsResult" type="s0:ArrayOfString" />
</s:sequence>
</s:complexType>
</s:element>
<s:element name="getCar">
<s:complexType />
</s:element>
<s:element name="getCarResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1"
 name="getCarResult" type="s0:Car" />
</s:sequence>
</s:complexType>
</s:element>
<s:complexType name="Car">
<s:sequence>
<s:element minOccurs="0" maxOccurs="1"
 name="Auttotyp" type="s:string" />
<s:element minOccurs="1" maxOccurs="1"
 name="Tagespreis" type="s:double" />
</s:sequence>
</s:complexType>
</s:schema>
</types>
<message name="getKategoriesSoapIn">
<part name="parameters" element="s0:getKategories" />
</message>
```

```
 <message name="getKategoriesSoapOut">
 <part name="parameters" element="s0:getKategoriesResponse" />
 </message>
 <message name="getCarsForKatsSoapIn">
 <part name="parameters" element="s0:getCarsForKats" />
 </message>
 <message name="getCarsForKatsSoapOut">
 <part name="parameters" element="s0:getCarsForKatsResponse" />
 </message>
 <message name="getCarSoapIn">
 <part name="parameters" element="s0:getCar" />
 </message>
 <message name="getCarSoapOut">
 <part name="parameters" element="s0:getCarResponse" />
 </message>
 <portType name="PartLiveAuto1Soap">
 <operation name="getKategories">
 <documentation>Liefert die verfügbaren
  Autokategorien.</documentation>
<input message="s0:getKategoriesSoapIn" />
 <output message="s0:getKategoriesSoapOut" />
 </operation>
 <operation name="getCarsForKats">
 <input message="s0:getCarsForKatsSoapIn" />
 <output message="s0:getCarsForKatsSoapOut" />
 </operation>
 <operation name="getCar">
 <input message="s0:getCarSoapIn" />
 <output message="s0:getCarSoapOut" />
 </operation>
 </portType>
 <binding name="PartLiveAuto1Soap" type="s0:PartLiveAuto1Soap">
 <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
  style="document" />
 <operation name="getKategories">
 <soap:operation soapAction="www.autoservice.de/getKategories"
  style="document" />
 <input>
 <soap:body use="literal" />
 </input>
```

```
<output>
<soap:body use="literal" />
</output>
</operation>
<operation name="getCarsForKats">
<soap:operation soapAction="www.autoservice.de/getCarsForKats"
 style="document" />
<input>
<soap:body use="literal" />
</input>
<output>
<soap:body use="literal" />
</output>
</operation>
<operation name="getCar">
<soap:operation soapAction="www.autoservice.de/getCar"
 style="document" />
<input>
<soap:body use="literal" />
</input>
<output>
<soap:body use="literal" />
</output>
</operation>
</binding>
<service name="PartLiveAuto1">
<documentation>Ein Testwebservice für Partlive.</documentation>
<port name="PartLiveAuto1Soap" binding="s0:PartLiveAuto1Soap">
<soap:address location="
 http://localhost/PartLiveAutoService/PartLiveAuto1.asmx" />
 </port>
</service>
</definitions>
```

Literaturverzeichnis

[1] MSDN Library 2003:*MSDN Library Visual Studio .Net 2003*- http://msdn.microsoft.com- Last Online: 26.02.2005

[2] AIS - Sminare an der HAW-Hamburg - http://users.informatik.haw-hamburg.de/~ais/ - Last Online : 10.12.2004

[3] Drayton Peter, Albahari Ben, Neward Ted :*C# In a Nutshell* - O'Reilly & Associates, Inc, USA, März 2002 -ISBN-0-596-00181-9- http://www.oreilly.com

[4] Eberhart, Andreas/Fischer, Stefan: *Web Services Grundlagen und praktische Umsetzung mit J2EE und .NET* - Hanser Fachbuchverlag,Oktober 2003. -ISBN 3-44622-530-7-

[5] Short, Scott: *Webdienste mit dem .Net Framework entwickeln* - Microsoft Press, 2002. -ISBN 3-86063-644-8-

[6] Webseite Avis Autovermietung - Last Online: 28.02.2005 http://www.avis.de

[7] Webseite Sixt Autovermietung - Last Online: 28.02.2005 http://www.sixt.de

[8] Webseite Hertz Autovermietung - Last Online: 28.02.2005 http://www.hertz.de

[9] Webseite des UbiCom-Labs an der HAW-Hamburg - Last Online: 28.02.2005 http://users.informatik.haw-hamburg.de/~ubicomp/

[10] Officiella Microsoft ASP .NET Seite - Last Online 12.03.2005 http://www.asp.net

[11] DotNetPro *DotNetPro Das Provi-Magazin für Entwickler*. Ausgabe Juli/August 2003 - 18.Jahrgang. Seite 16 - 21. http://www.dotnetpro.de

[12] dotNet auf Linux *Mono Projekt* http://www.mono-project.com/Main_Page

[13] MacDonald, Matthew. *.Net Distributed Applications*. Integration XML Web Services and .NET Remoting. Microsoft Press. 2003. ISBN 0-7356-1933-6

[14] W3C World Wide Web consortium www.w3c.org/

Abbildungsverzeichnis