Verteilte Anwendungen

mit

Martin C. Pöpping und Thomas V. Wild

Inhaltsverzeichnis

1. Einführung in .NET

1.1 Verteilte Anwendungen mit .NET

.NET soll in Zukunft Microsofts Plattform für die Entwicklung von verteilten Anwendungen sein. Die Technologie die dafür verwendet werden soll sind Web Services.

Mit .NET sollen sich verteilte Softwarekomponenten bauen lassen, die untereinander kommunizieren und Daten gemeinsam nutzen können, ohne dabei fest miteinander gekoppelt zu sein. Die.NET Technologie soll den PC mit modernen Gadgets wie z. B. Mobiltelefone, PDAs und Handhelds verheiraten. Informationen sollen nicht nur auf einem Gerät verfügbar sein, sondern auch von anderen Geräten einsehbar und manipulierbar sein. Das Internet soll dabei das gemeinsam genutzte Kommunikationsmittel zwischen den einzelnen Geräten sein. .NET besteht aus einem umfangreichen Satz von Werkzeugen zur Softwareentwicklung für Internet, Windows-PCs und andere Geräte.

Sanjay Parthasarathy Vice President, Platform Strategy, Microsoft Corp. beschreibt die .NET- Strategie so:

- Everything is a Web service.
- The ability to aggregate and integrate Web services.
- The ability to deliver a simple and compelling experience to end users.

Jede verteilte Softwarekomponente soll eine Web Services Schnittstelle bereitstellen. Dadurch werden diese Softwarekomponenten plattformübergreifend und sprachenübergreifend verfügbar.

Diese Web Services müssen sich anschließend zusammenfügen lassen. Neue Services lassen sich aus vorhandenen entwickeln. Web Services kommunizieren untereinander um so neue verteilte Applikationen entstehen zu lassen

In der Endanwendung sollen diese verteilten Applikationen leicht zugänglich sein. Unter der Verwendung von herkömmlichen Geräten, wie z. B. PCs und Terminals aber auch durch mobile Geräte. Der Anwender soll zwischen Services und Endgeräten wechseln können.

1.2 Web Services

Web-Services sind einfach ausgedrückt maschinenlesbare Webseiten. Genauso wie ein Surfer eine Webseite im Internet aufruft, um sich Informationen anzeigen zu lassen, können Programme einen Web-Service aufrufen, um Daten abzufragen, die das Programm selbst weiterverarbeiten möchte.

Um Web-Services zu implementieren bedient man sich diverser Techniken.
Eine endgültige Standardisierung ist noch nicht erreicht. Die gebräuchlichste Methode, die auch von .NET Web Services verwendet wird, benutzt die Technologien SOAP, WSDL und UDDI.

1.3 SOAP

SOAP ist ähnlich wie XML-RPC. Mit SOAP kann man RPC über HTTP ausführen. SOAP unterscheidet sich von XML-RPC aber dadurch, dass man mit SOAP auch asynchrones Messaging betreiben kann und, dass SOAP auch komplexe Datenstrukturen in Form von XML-Schemas als Parameter und Rückgabewerte verwenden kann. Mittels eines POST-Request wird ein SOAP-Dokument, das RPC-Daten enthält an einen Server geschickt. Dabei muß der Content-Type immer „text/xml“ sein. Zusätzlich kann der optionale Parameter SOAPAction gesetzt werden, der dazu dient eingehende SOAP-Requests gegenüber einer Firewall zu identifizieren.

```
POST /soapworkshop/services/id/id.asp HTTP/1.1
Host: xxx.xxx.xxx.xxx
Content-Type: text/xml
Content-Length: nnn
SOAPAction:
"http://www.topxml.com/soapworkshop/GetSecretIdentity"

<?xml version="1.0"?>
<S:Envelope =
xmlns:S='http://schemas.xmlsoap.org/soap/envelope/'

S:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/
'>
    <S:Body>
        <vb:GetSecretIdentity

xmlns:vb='http://www.topxml.com/soapworkshop/'>
            <codename>Daredevil</codename>
        </vb:GetSecretIdentity>
    </S:Body>
</S:Envelope>
```

Root-Element eines SOAP-Dokuments ist immer „Envelope“. Envelope identifiziert das Dokument als SOAP-Nachricht inklusive Versionsnummer und spezifiziert Regeln für das Serialisieren von Daten.

Das Header-Element ist optional und kann verwendet werden, um Applikationsunabhängige Erweiterungen der Syntax zu definieren.

Im Body-Element werden Applikationsspezifische Informationen eingetragen.
Wie z. B. ein RPC-Aufruf „GetSecretIdentity“ mit dem Parameter „codename“, der den Inhalt „Daredevil“ hat.

Das HTTP-Response enthält die Rückgabewerte des RPC-Aufrufs. Per Konvention lauten die Namen der Elemente, die Antworten auf entsprechende Anfragen sind genauso wie die Anfrage plus den Suffix „Response“.
Auf unsere Anfrage „GetSecretIdentity“ erhalten wir also die Antwort in „GetSecretIdentityResponse“.

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Wed, 31 Jan 2001 07:21:19 GMT
MessageType: CallResponse
Content-Length: nnn
Content-Type: text/xml
Expires: Wed, 31 Jan 2001 07:21:20 GMT
Cache-control: private

<?xml version="1.0"?>
<Env:Envelope
xmlns:Env="http://schemas.xmlsoap.org/soap/envelope/"

Env:encodingStyle="http://schemas.xmlsoap.org/soap/encodin
g/"
 xmlns:vb='http://www.topxml.com/soapworkshop/">
    <Env:Body>
        <vb:GetSecretIdentityResponse>
            <id>Matt Murdock</id>
        </vb:GetSecretIdentityResponse>
    </Env:Body>
</Env:Envelope>
```

1.4 WSDL

WSDL steht für Webservices Description Language.
WSDL basiert ebenfalls auf XML und beschreibt die Schnittstelle die ein Webservice anbietet. Aus der Sicht eines CORBA oder COM Programmierers entspricht WSDL der IDL. Die Beschreibung eines Dienstes durch WSDL stellt z. B. sicher, dass sich auch bisher unbekannte Web Services in eine bestehende Anwendung einbauen lassen.

Zu den Informationen, die ein WSDL Dokument beschreibt, gehören z. B.:

- Der Name des Services, einschließlich der URN
- Den Ort an dem auf den Service zugegriffen werden kann (normalerweise eine HTTP-URL-Adresse)
- Die aufrufbaren Methoden
- Die Argumenttypen und die Typen der Rückgabewerte für jede Methode.

Aus einem WSDL Dokument kann dann automatisch ein Client erzeugt werden, der mit dem beschriebenen Web Service interagieren kann.

1.5 UDDI

Da UDDI als ein Thema von einer anderen Gruppe bearbeitet wird, beschreibe ich hier nur kurz, wie UDDI von .NET verwendet wird. UDDI ist eine zentrale Instanz, die das Registrieren von Web Services ermöglicht. An UDDI kann man anschließend Suchanfragen stellen, um bestimmte Web Services zu finden.

2. Die Technik von .NET

2.1 Offener Standard?

Microsoft wirbt für .NET als einen „offenen Standard". Kritiker dagegen stehen dieser Aussage skeptisch gegenüber. So ist zwar die CLI (Common Language Infrastructure) und C# von der ECMA [3] als Standard veröffentlicht, diese kümmert sich allerdings nicht um eventuelle Patentfragen. Eine Standardisierung durch die ECMA hat zur Folge, dass dieser Standard öffentlich wird und von jedem implementiert werden darf. Allerdings könnte Microsoft Teile seines Konzeptes trotzdem Patentieren lassen und somit Lizenzgebühren von Drittanbietern, die den Standard implementieren, verlangen. Die ECMA stellt hier allerdings die Forderung, dass diese Lizenzgebühren zu akzeptablen Konditionen erhoben werden.
Den für .NET wichtigen .NET-Framework hat Microsoft allerdings nicht bei der ECMA als öffentlichen Standard verabschieden lassen. Drittanbieter können deshalb nicht auf Teile des .NET-Frameworks zugreifen und haben somit einen entscheidenden Nachteil [2]. Die Aussage von Microsoft: „.NET ist ein offener Standard" ist somit eigentlich nur eine Halbwahrheit und mit äußerster Vorsicht zu genießen.

2.2 Standardisierung der WSDL

Ähnlich handelt Microsoft auch bei der WSDL. Die WSDL soll die Fähigkeiten der Web Services mit XML beschreiben. Diese ist noch nicht endgültig standardisiert (Stand März 2002) sondern nur als Draft erhältlich. Die Standardiserung wurde der W3C im März 2001 vorgelegt. Konkurrenten von Microsoft ist es daher nicht möglich frühzeitig Produkte auf den Markt zu bringen, da Microsoft die WSDL noch jeder Zeit in wichtigen Punkten abändern könnte. Es ergibt sich somit für Microsoft Konkurrenten ein starker Wettbewerbsnachteil [2]. Nach diesem Muster geht Microsoft auch mit den .NET myServices-Spezifikationen vor, die bis heute noch nicht in einem endgültigen Standard verabschiedet sind.

2.3 Technik von .NET

.NET Software setzt ähnlich wie Java auf einer virtuellen Maschine auf. Die so genannte Common Language Runtime (CLR). Die CLR ist Microsofts Implementierung der Common Language Infrastructure (CLI). .NET-Programme werden ähnlich wie bei Java in einen Zwischencode übersetzt. Dieser Zwischencode wird als Microsoft Intermediate Language (MSIL) bezeichnet. Später kann dann dieser Zwischencode zur Laufzeit oder bereits früher in Maschinencode übersetzt werden. Neu ist, dass der Entwickler nicht auf eine Programmiersprche beschränkt ist. .NET-Programme lassen sich in verschiedenen Programmiersprachen schreiben. Zur Zeit werden die folgenden Sprachen unterstützt:

- C++
- Visual Basic
- C#

In internen nicht offiziellen Projekten wurden des weiteren .NET Varianten von **COBOL**, **Eiffel** und **Pascal** bereitgestellt.
Außerdem bereits verfügbar im Beta-Stadium Microsofts proprietäre Version von Java: **J#.** Mit J# lassen sich keine Programme schreiben, die auf irgendeiner JVM laufen, sondern J# Programme sind nur auf der .NET Platform lauffähig. J# ist in etwa auf dem technischen Stand von Java in der Version 1.1.4.

Die bedeutendste Programmiersprache dürfte allerdings C# sein. C# ist die Programmiersprache, die von Microsoft speziell zur Entwicklung von .NET-Programmen entwickelt wurde. Bei den anderen unterstützten Programmiersprachen ist eine Übersetzung durch einen .NET-Compiler nicht ohne weiteres möglich beziehungsweise anzuraten. Damit eine Interoperabilität unter den Programmen möglich ist, die in unterschiedlichen Programmiersprachen geschrieben worden sind, ist es notwendig, dass die Programme der sogenannten Common Language Specification (CLS) entsprechen. So können Entwickler in unterschiedlichen Programmiersprachen arbeiten und trotzdem den erstellten Code eines Anderen verwenden. CLS konform bedeutet z. B. bezüglich C++, die Programme dürfen keine Mehrfachvererbung verwenden. VB Programme müssen unter Verwendung .NET spezifischer Spracherweiterungen objektorientiert gemacht werden.

2.4 Common Language Specification

Nur wenn die Programme der CLS entsprechen ist eine sprachenübergreifende Interoperabilität möglich. Die CLS beschreibt eine Menge fundamentaler Spracheigenschaften und Regeln, wie diese benutzt werden müssen.

Die CLS ermöglicht, dass:

- Datentypen von anderen Datentypen erben, Instanzen als Parameter an ein Objekt eines anderen Datentyps übergeben werden können, Objekte Methoden anderer Objekte aufrufen, unabhängig davon in welcher Sprache die jeweilige Klasse implementiert ist. Das heißt z. B. eine C# Klasse kann von einer VB-„Klasse“ erben.

- Debugger und ähnliche Tools brauchen nur eine Umgebung zu verstehen. Die MSIL (Microsoft Intermediate Language) und die Metadaten für die CLR (die Common Language Runtime)

- Ein einheitliches Exception-Handling. Programmcode, geschrieben in einer Sprache kann eine Exception werfen, die von einem anderen Programmcode, der in einer anderen Sprache geschrieben wurde, gefangen und verstanden werden kann.

Damit Programmcode sprachenübergreifend verwendet werden kann ist es notwendig, dass die öffentliche Schnittstelle CLS konform ist.
Das heißt im Einzelnen:

- Öffentliche Klassen müssen CLS konform sein
- Öffentliche Instanzvariablen öffentlicher Klassen und Instanzvariablen, die in abgeleiteten Klassen sichtbar sind müssen CLS konform sein.
- Parameter und Rückgabewerte öffentlicher Methoden von öffentlichen Klassen sowie Parameter und Rückgabewerte solcher Methoden, die in abgeleiteten Klassen sichtbar sind müssen CLS konform sein.

Alles was nicht öffentlich, und auch nicht in abgeleiteten Klassen sichtbar ist darf die CLS Regeln verletzen ohne dabei die CLS Konformität der ganzen Klasse zu gefährden.

Assemblies, Module, Typen, und Instanzvariablen können vom Programmierer explizit als CLS konform deklariert werden. Dazu wird das „CLSCompliantAttribute“ verwendet. Eine Assembly, die nicht explizit als CLS konform deklariert wurde wird als nicht CLS konform betrachtet. Module, Typen und Instanzvariablen, die nicht explizit deklariert wurden erhalten den gleichen Wert, den die Assembly hat, der sie angehören.

Zum Beispiel die folgenden Datentypen sind CLS konform:
Byte (unsigned), Int16, Int32, Int64, Single, Double, Boolean, Char (Unicode 16 Bit), Decimal, IntPtr, String.

Für eine CLS konforme Klasse gilt z. B., dass sie von einer CLS konformen Klasse abgeleitet sein muss. System.Object ist CLS konform.

Der C# Compiler erzeugt Fehlermeldungen, wenn eine Klasse nicht CLS konform ist.

```
[C#]

using System;

// Assembly marked as compliant.

[assembly: CLSCompliantAttribute(true)]

// Class marked as compliant.

[CLSCompliantAttribute(true)]

public class MyCompliantClass {

   // ChangeValue exposes UInt32, which is not in CLS.

   // A compile-time error results.

   public void ChangeValue(UInt32 value){ }

   public static void Main( ) {

   int i = 2;

   Console.WriteLine(i);

   }

}
```

Dieser Code zeigt bei der Übersetzung folgenden Fehler in der Methode „ChangeValue“ an:

```
error: CS3001: Argument type 'uint' is not CLS-compliant.
```

Wenn man explizit für die Methode „ChangeValue“ das CLSCompliantAttribute auf false setzt ist eine Übersetzung möglich.

2.5 Common Type System (CTS)

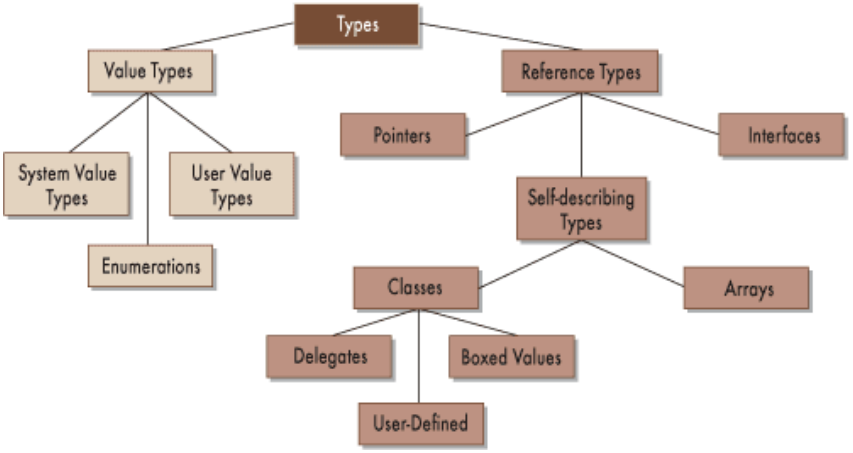

Abb.1: Common Type System

Das CTS unterscheidet grob zwischen "value types" und "reference types". Zu der Kategorie der value types zählen primitive Datentypen, Aufzählungstypen (enum) und benutzerdefinierte Werttypen (struct). Value Types werden auf dem Stack verwaltet. Zu den Reference Types zählen Felder, Klassen, Delegates (typisierte Funktionszeiger) und Zeiger. Die Reference Types werden auf dem Heap gespeichert. Im CTS sind alle Datentypen auch gleichzeitig Objekte. Die Typbezeichner in C# (int, short, long, float, string, object) sind lediglich Aliases für die entsprechenden Typen im CTS.

- Short System.Int16
- Int System.Int32
- Long System.Int64
- Float System.Single
- String System.String
- Object System.Object

Eine Unterscheidung in Objekte und sonstige Datentypen wie in Java oder C++ gibt es nicht. Value Types sind von der Klasse System.ValueType abgeleitet. Value Types können auch per Referenz übergeben werden. Dazu wird zu einem Objekt eines Value Types ein semantisch identisches Referenzobjekt erzeugt. Dieser Vorgang wird „Boxing" genannt. Geboxte Objekte können auch wieder zurückverwandelt werden (Unboxing).

Ein Datentyp kann von maximal einem anderen Datentyp erben. Aber ein Datentyp kann beliebig viele Interfaces implementieren. Alle Datentypen sind von System.Object abgeleitet.

Benutzerdefinierte Werttypen (Value Types) lassen sich effizient verwenden, wenn es sich um relativ kleine Datentypen handelt. Zum Beispiel die Darstellung von komplexen Zahlen.
Benutzerdefinierte Werttypen lassen sich auch per Wertübergabe an Methoden übergeben. Außerdem ist mit Benutzerdefinierten Werttypen nicht die Menge an Aufwand für die Speicherung verbunden, wie für Instanzen von Klassen. Für selbstdefinierte Werttypen wird automatisch eine geboxte Version bereitgestellt. Werttypen können Attribute, Properties, Events, statische und nicht-statische Methoden haben. Value Types sind stets versiegelt (sealed), das heißt, man kann keinen weiteren Datentyp von Value Types ableiten.

Delegates sind ähnlich wie Funktionszeiger in C++. Delegates sind immer typisiert und können sowohl statische als auch Instanzmethoden referenzieren. Dabei muss aber der Typ der referenzierten Methode mit dem Typ des Delegates kompatibel sein. Delegates werden hauptsächlich zur Ereignisbehandlung (event handling) und für Callback-Methoden verwendet. Alle Delegates erben von System.Delegate und verfügen über eine Aufrufliste der Methoden, die aufgerufen werden, wenn das Delegate Objekt aufgerufen wird.
Der Rückgabewert eines Delegate Objekts ist nicht definiert, wenn das Delegate einen Rückgabetyp hat und die Aufrufliste mehr als eine Funktion beinhaltet.

Arrays sind ähnlich wie in Java. Die Operationen die auf einem Array möglich sind, sind:
Obere Grenze abfragen (Länge des Array)
Die Dimension des Arrays abfragen
Die Anzahl der im Array gespeicherter Werte abfragen.

Es gibt drei unterschiedliche Zeigertypen:

- Managed Pointers
- Unmanaged Pointers
- Unmanaged Function Pointers

Die letzten beiden Zeigertypen stammen aus C++ und sind nicht CLS konform. Programmiersprachen, die solche Zeiger nicht haben, werden auch nicht auf solche Zeiger zugreifen können. Managed Pointers referenzieren einen „managed block of memory“ vom Heap der Common Language Runtime. Managed Pointers werden für Methodenargumente erzeugt, die per Referenz übergeben werden. Für managed Pointers gibt es zwei typsichere CLS konforme Operationen. Einen Wert von einem managed Pointer lesen und einen Wert schreiben. Es gibt noch drei weitere byte-basierte Operationen auf managed Pointers, die aber nicht CLS konform sind. Einen Integerwert auf einen managed Pointer addieren oder subtrahieren. Und einen managed Pointer von einem anderen abziehen.

2.6 Die Common Language Runtime

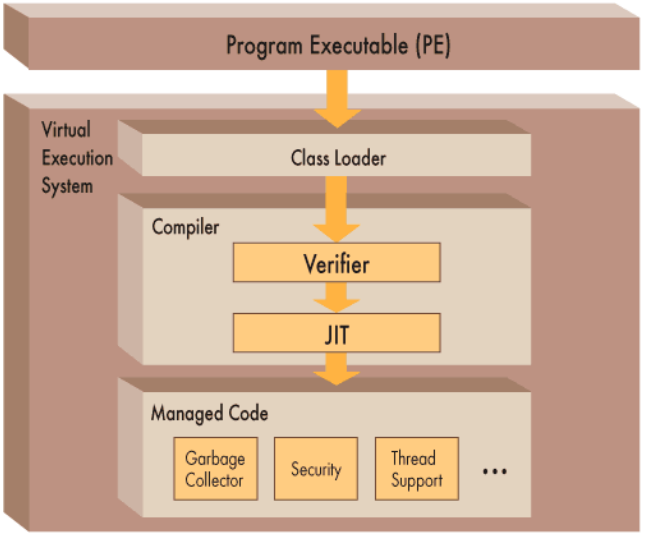

Abb. 2: Common Language Runtime 1

Die Common Language Runtime basiert auf einem Virtual Execution System (VES). Hierbei handelt es sich um eine abstrakte Stackmaschine, die ausschließlich Anweisungen des Zwischencodes MSIL ausführt. Programmcode, der von der CLR ausgeführt wird, wird als „managed Code" bezeichnet.

Der in der VES integrierte JIT-Compiler übersetzt den MSIL-Code in nativen Code der ausführenden Hardware.

Die CLR benutzt Metadaten, die im kompilierten MSIL-Code enthalten sind, um verschiedene Dienste für den managed Code bereitzustellen. Die Metadaten enthalten Informationen über Typen, Member und Referenzen. Die CLR benutzte diese Metadaten, um Klassen zu lokalisieren und zu laden, um Instanzen im Speicher zu allozieren, Methodenaufrufe aufzulösen, nativen Code zu generieren und Sicherheit zu gewährleisten.

Jede Methode wird bevor sie das erstemal aufgerufen wird vom JIT-Compiler in Maschinencode übersetzt. Weitere Ausführungen dieser Methode können dann direkt den vorliegenden Maschinencode verwenden.

Install-time code generation erzeugt nativen Code wenn die Assembly installiert wird, unter Berücksichtigung, welche anderen Assemblies bereits installiert sind.

Die CLR ermöglicht auch Code aufzurufen, der nicht von der CLR, sondern außerhalb der Laufzeitumgebung ausgeführt wird. Code, der außerhalb der CLR ausgeführt wird, wird als „unmanaged Code" bezeichnet. Das können z. B. sein:

- Active X interfaces
- COM Komponenten
- Win32 API Funktionen

Um COM Komponenten in .NET zu integrieren stellte Microsoft ein Mapping bereit (COM Interop). Anderer nativer Code wird mit Hilfe des sogenannten „Platform Invoke" (P/Invoke) integriert.

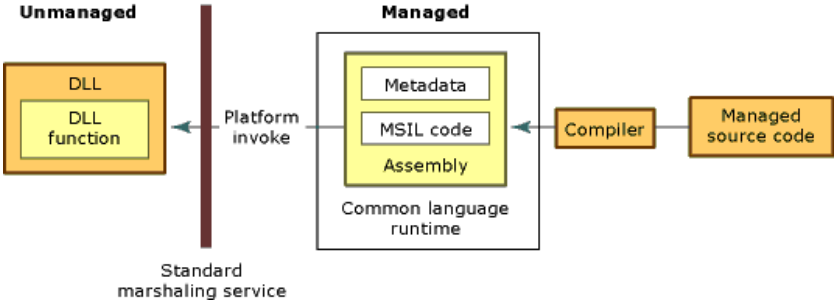

Abb. 3: Platform Invoke 1

Der Verifizierer prüft während der Kompilierung in nativen Code, ob der Code typsicher ist. Das heißt, es wird nur auf Speicherbereiche zugegriffen für die der Code berechtigt ist.

Die CLR verfügt auch über einen Garbage Collector. Der Garbage Collector ist dafür zuständig alle nicht persistenten Programmdaten zu verwalten (managed data). Der Programmierer kann Daten auch manuell allozieren und deallozieren (unmanaged data).

3. Die Programmiersprache C#

3.1 Einleitung

C# ist die Standardsprache von .NET, die mit .NET neu eingeführt wurde und verdient deswegen eine genaue Betrachtung. Wie später gezeigt wird, hat sie viele Gemeinsamkeiten zu Java und C/C++.
Es soll allerdings nur ein Überblick über die Funktionsweise und die Besonderheiten dieser neuen Sprache gezeigt werden. Tutorials zum Erlernen der Sprache gibt es mittlerweile viele im Netz [1].

3.2 Syntax

Das folgende Beispiel zeigt den typischen Aufbau eines C# Programms. Klassen- und Methodenrümpfe sind identisch mit denen aus Java (Zeile 8), allerdings ist es gebräuchlich in C# Methodennamen ebenfalls groß zu schreiben. Weitere Einzelheiten sind in den jeweiligen Kommentaren zu finden.

Beispiel:

```
1: //verwenden des Namespaces “System”.
2: using System;
3:
4: //definieren eines Namespaces, vergleichbar mit Paketen in Java
5: namespace MyNameSpace
6: {
7:         //Klasse HalloVPSKurs
8:         public class HalloVPSKurs
9:         {
10:        /*
11:        Mainmethode. Im Gegensatz zu Java kann eine C# auch mehrere Main-
12:        Methoden haben. Beim kompilieren wird angegeben wo der Compiler
13:        einsetzen soll.
14:        */
15:                public static void Main(string[] args)
16:                {
17:                      /*
18:                      Methode System.Console.WriteLine, vergleichbar mit
19:                      System.out.println aus Java
20:                      */
21:                      Console.WriteLine(“Hallo {0}! Test Nr. {1}”,“VPS-
                         Kurs”, 2);
22:                }
23:        }
24:}
```

Ausgabe:

```
Hallo VPS-Kurs! Test Nr. 2
```

3.3 Kommandozeilen-Tools

3.3.1 CSC

CSC ist der C#-Compiler (**C**-**S**harp-**C**ompiler) zum Übersetzen von C#-Programmen und ist vergleichbar mit diversen anderen Kommandozeilen-Compilern wie cc oder javac.
Um beispielsweise eine Datei „HalloVPSKurs.cs“ als Windows-Anwendung „HalloVPSKurs.exe“ zu kompilieren wird csc mit folgenden Parametern aufgerufen:

```
csc /target:winexe /out:HalloVPSKurs.exe HalloVPSKurs.cs
```

Mit /target: legt man den Anwendungstyp des Programms fest. /out:Dateiname bezeichnet den Dateinamen des fertig kompilierten Programms. Als letztes Argument verlangt der csc dann den Dateinamen des zu kompilierenden Quellcodes.

3.3.2 ILDASM

ILDASM ist der Intermediate Language Disassembler. Mit diesem Tool ist es möglich IL-Code zu betrachten. Aufgerufen wird das Tool standardmäßig mit:

```
ildasm HalloVPSKurs.exe
```

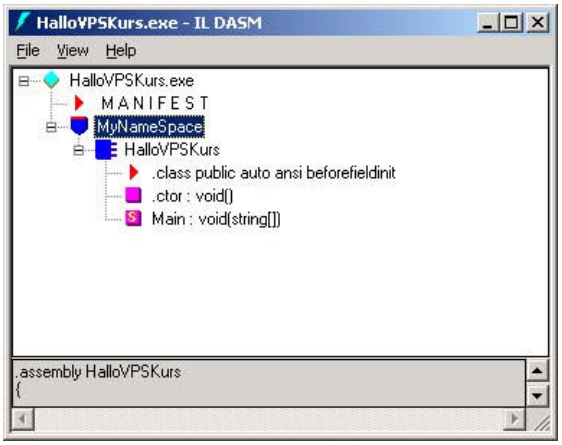

Abb. 4: Windows-Version von ildasm

Ausgabe als Textfile:

```
___[MOD] HalloVPSKurs.exe
   |      M A N I F E S T
   |___[NSP] MyNameSpace
   |   |___[CLS] HalloVPSKurs
   |   |   |     .class public auto ansi beforefieldinit
   |   |   |___[MET] .ctor : void()
   |   |   |___[STM] Main : void(string[])
   |   |
   |
```

3.3.3 ILASM

ILASM ist der Intermediate Language Assembler. Der Standard-Aufruf erfolgt durch:

```
ilasm fileToAssemble.il
```

3.3.4 DUMPBIN

Dumpbin ermöglicht das Betrachten eines kompilierten Programms im PE-Format:

```
dumpbin HalloVPSKurs.exe
```

Ausgabe:

```
Microsoft (R) COFF/PE Dumper Version 7.00.9254
Copyright (C) Microsoft Corporation.  All rights reserved.


Dump of file HalloVPSKurs.exe

File Type: EXECUTABLE IMAGE

  Summary

        2000 .reloc
        2000 .rsrc
        2000 .text
```

3.3.5 AL

Um Assemblies zu erstellen oder zu modifizieren verwendet man den Assembly-Linker AL. Als zusätzlichen Parameter verlangt AL eine „Link-Resource“:

```
al /linkresource:eineLinkResource HalloVPSKurs.exe
```

3.3.6 GACUTIL

GACUTIL wird verwendet um die erstellten Shared Assemblies im globalen Cache bereit zu stellen:

```
gacutil –i myLibrary.il
```

3.3.7 SN

SN (Shared Name Utility) ist ein Tool das Schlüsselpaare erzeugt um die erstellten Shared Assemblies zu signieren:

```
sn –k myKey.key
```

3.4 Assemblies und Reflexion

Assemblies sind die logischen Einheiten, die die .NET-Funktionen bereitstellen, sodass mit verschiedenen Programmiersprachen darauf zugegriffen werden kann.
Jedes Assembly enthält eine Manifest-Datei. Diese Datei enthält Sicherheitsinformationen, Referenzen auf importierte Assemblies sowie exportierte Ressourcen und Assemblies. Das Manifest selbst befindet sich entweder global im Assembly oder in dem jeweiligen Modul. Module implementieren den Code für die darin bereitgestellten Typen (vgl.: Abb.5)

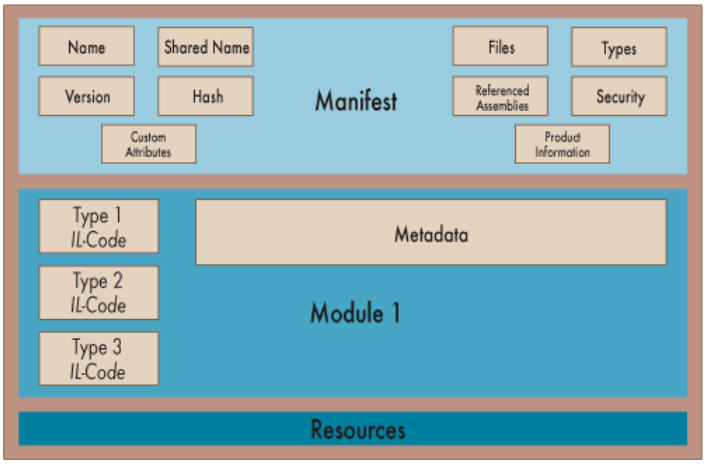

Abb. 5: Struktur eines Assemblies bestehend aus Manifest, Module und Resources

Beim Laden des Assemblies wird ein Runtime Host aktiviert, welche die Kontrolle über die Kompilierung und Ausführung übernimmt. Die Metadaten hierzu sind explizit im Assembly vorhanden. Mittels eines Reflexionsmechanismus kann man alle Informationen über Typen dynamisch abfragen. Dies geschieht durch die ausführlichen Metainformationen, die jedem Modul in einem Assembly beigefügt sind. Durch diese Kombination von Manifest, Metainformationen und Reflexionsmechanismen ist für .NET keine zentrale Registrierung der Assemblies erforderlich, wie es bei dem COM-Modell der Fall ist.
Man unterscheidet zwischen private und shared Assemblies. Private Assemblies werden nur von der lokalen Anwendung unterstützt und liegen daher im lokalen Verzeichnisbaum der Anwendung. Shared Assemblies werden in einem globalen Verzeichnis, mittels verschiedener Werkzeuge, installiert und sind von allen Anwendungen verwendbar. Um Assemblies global zur Verfügung zu stellen werden sie mit dem öffentlichen Schlüssel des Entwicklers signiert, um Namenskollisionen mit anderen Assemblies zu vermeiden.

3.5 Gemeinsamkeiten und Unterschiede zu C/C++ und Java

Wie schon aus dem Syntax zu erkennen ist, hat C# viele Eigenschaften der C-Familie und aus Java übernommen, auch wenn Java in keinem Satz der 376-seitigen C# Language Specification [5] erwähnt wird. So wird sowohl in C# als auch in Java ein Zwischencode generiert. Die Klassenhierarchie ist ähnlich modular aufgebaut, basierend auf einer root-Klasse „Object". Im Gegensatz zu C++ ist bei C# keine Mehrfachvererbung möglich, sondern es werden, wie bei Java, Interfaces verwendet. C# verzichtet wie Java auf globale Variablen. Variabeln müssen immer einer bestimmten Klasse zugehörig sein und initialisiert werden. Innere Klassen gibt es ebenfalls in C# sowie eine zu Java vergleichbares Exception Handling mit try-catch-finally Konstrukten. Der benutzte Speicher wird von einem GarbageCollector automatisch wieder frei geräumt, kann aber auch manuell freigegeben werden.
Neben den Schlüsselwörtern public, protected und private gibt es die Schlüsselwörter internal und protected internal. Internal bezieht sich hierbei auf die Assemblies. Eine als internal definierte Methode kann beispielsweise nur innerhalb des eigenen Assemblies aufgerufen werden.
Neben den Gemeinsamkeiten zwischen C# und Java gibt es auch einige Unterschiede.
Es wird nicht auf die in der C-Familie verwendeten Pointer verzichtet. Diese müssen allerdings als unsafed-Code [Kap. 2.5] markiert werden. Es können Operatoren überladen werden und es gibt mehrere Typen aus der C-Familie, welche in Java nicht vorhanden sind: Primitives, Structs, Enums, Unsigned Types, Decimal sowie rechteckige Arrays. Auch kann man in C# die fehleranfällige GoTo-Anweisung verwenden.
Man erkennt also, dass C# eine Mischung aus Java und C++ ist, wobei viele sinnvolle Techniken aus Java verwendet wurden, die fehleranfälligen Techniken wie GoTo und Pointer aus der C-Familie allerdings ebenfalls implementiert wurden. C# überlässt es dem Programmierer also selbst seinen Code sicher oder eventuell fehleranfällig zu erstellen wobei Java versucht unsicheren Code von vorne herein auszuschließen indem auf bestimmte Techniken verzichtet wurde.
Letzter Unterschied, der noch erwähnt werden sollte, ist der aus C übernommene Präprozessor, der auch eine bedingte Übersetzung ermöglicht.

4. Anwendungen mit .NET am Beispiel von .NET myServices

4.1 Grundidee

.NET myServices soll diverse Dienste und Services zu einer neuartigen Kommunikationsplattform verschmelzen. So sollen Adressbücher und Kalender, wie sie beispielsweise auf diversen Webseiten online angeboten werden, auf Handy und PDA übertragbar sein und somit eine gemeinsame Kommunikationsplattform für alle Kommunikationsdienste bieten. Hier kommt die Technologie von .NET ins Spiel. .NET-Anwendungen, wie Terminplaner für PDAs und Handys, können somit direkt miteinander kommunizieren.
Ein klassisches Anwendungsszenario ist folgendes [2]:
Eine Opernliebhaberin sucht über .NET myServices eine passende Opernvorstellung, bestellt direkt die Karten und bezahlt elektronisch über .NET Passport (siehe 3.2 .NET Passport). Anschließend wird der Termin direkt bei ihr, und eventuell bei Begleitpersonen, in den Terminkalender eingetragen. In ferner Zukunft könnte das System dann noch die Anfahrtszeiten und eine passende Reiseroute berechnen, sowie eventuell Bahn-Tickets bestellen. Würde die Vorstellung ausfallen, wird der Termin automatisch aus dem Terminkalender ausgetragen und ein passender neuer Termin mit den Kalendern der Begleitpersonen abgestimmt.
Technisch wird dies realisiert, indem der Terminkalender des Anwenders auf einem Server von Microsoft gespeichert wird. Dieser Schritt sollte allerdings verantwortungsbewussten Anwendern zu denken geben.
.NET myServices bietet allerdings noch einiges mehr. Anwender können MS-Server als ausgelagerte Festplatte nutzen, wichtige Dateien, sowie Profile, Favoriten, Dodo-Listen und E-Mails auf dem Server speichern und hat diese somit auf jedem Rechner zur Verfügung.
Zentraler Dienst von .NET myServices ist .NET Passport [Kap. 4.2], andere Dienste sind: myProfile, myContacts, myLocations, myAlerts, myPresence, myInbox, myCalendar, myDocuments, myApplicationSettings, myFavoriteWebsites, myWallet, myDevices, myLists und myCategories.

4.2 .NET Passport

Passport ist eine Sammlung von Diensten die auf „Single Sign-in“ basieren. Dies bedeutet, dass der User sich nur einmal anmelden muss und dann beliebig viele Seiten besuchen kann, die Passport implementiert haben. Der Benutzer hinterlegt an einer zentralen Stelle im Internet seine Zugangsdaten auf die dann die angeschlossenen Dienste zugreifen können. Der zentrale Server von .NET Passport steht in diesem Fall bei Microsoft. Bei .NET Passport wird zur Registrierung nur eine gültige Email-Adresse sowie ein Passwort verlangt. Kunden von Hotmail erhalten schon automatisch einen Passport-Zugang. Alle anderen Angaben wie Name und Adresse bis hin zur Bankverbindung können optimal angegeben werden. Die an Passport angeschlossenen Webseiten sind verschiedenen Sicherheitsstufen zugeordnet. Niedrigste Stufe hierbei ist der Standard-Sign-in. Die Betreiber solcher Webseiten nutzen Passport beispielsweise, um den User ein eigenes Layout der Seite bzw. einen personalisierten Zugriff zu ermöglichen. Sie stellen nur eine Anfrage nach der Personal-User-ID (PUID) an den Passport-Server. Online-Versandhäuser sind beispielsweise einer hohen Sicherheitsstufe zugeteilt, da diese vertrauliche Daten wie Bankverbindungen abfragen.
Neben dem bereits erwähnten Standard-Sign-in gibt es noch andere Anmeldeverfahren. Wenn der Benutzer schon bei einem Dienst, der auf Passport zugreift, angemeldet ist, so kann er sich mittels „stiller Authentisierung“ anmelden. Hier muss der Nutzer nur auf

einen auf der Seite befindlichen Anmeldelink klicken und wird dann mittels einem gespeicherten Cookie authentifiziert. Er sich also nicht noch einmal durch eingeben der Email-Adresse und des Passwortes authentifizieren.
Eine andere Methode ist das „inline-Sign-in". Hier kann der Anbieter ein Modul auf seiner Seite implementieren, das ihm ermöglicht das Layout des Anmeldeprozesses selbst zu gestalten. Geht es um besonders vertrauenswürdige Daten, kann man den „Secure Channel Sign-in" verwenden. Mit dieser Methode werden alle Daten über eine SSL-Verbindung übertragen. Standardmäßig werden die Daten verschlüsselt mit Triple-DES versendet.
Um das Knacken von Passwörtern zu verhindern, gibt es den „Strong Credential Sign-in". Dieser erfordert nach der Standard-Anmeldung einen 4-stelligen Sicherheitscode, der nach 5 Falscheingaben gesperrt wird.
Hauptgrund für die Einführung war wohl der Passport-Dienst „Express Purchase", bei dem in einem „Wallet" auf dem Microsoft-Server Kreditkartennummer und Lieferadressen gespeichert werden. Diese vereinfachte Möglichkeit des Kaufens soll dazu führen, dass sich Kunden schneller für einen Kauf entscheiden, da sie nur „einen Klick vom Kauf entfernt" sind.

4.3 Lizenzkosten für Drittanbieter

Zwar wird .NET myServices für Endkunden kostenlos sein [2], Drittanbieter werden allerdings zur Kasse gebeten. Von diesen verlangt Microsoft 10.000$ pro Jahr sowie 1.500$ pro Kompatibilitätscheck, welcher in bestimmten Zeitabständen fällig wird.

5. .NET für Linux: „ Mono"

Mono ist ein Open Source Projekt, das eine Linux-basierte Version der .NET Plattform entwickelt. Mono ist hierbei keine spezielle Abkürzung, sondern ist das spanische Wort für Affe.
Ziel ist es Linux-Entwicklern eine Umgebung zu schaffen um plattformübergreifende .NET-Anwendungen zu schreiben. Es werden allerdings nur Technologien implementiert, die von Microsoft über die ECMA standardisiert wurden.
Zu beachten ist, dass Mono nur die Technologie von .NET implementiert und nicht die ganzen Dienste wie Passport etc.
Mono beinhaltet zur Zeit folgende Komponenten:

1. eine Common Language Infrastructure (CLI) Virtual Machine, welche einen Class Loader, einen Just-In-Time Compiler und eine Garbage-Collector-Laufzeitumgebung beinhaltet
2. eine Klassenbibliothek die mit jeder Sprache, welche die CLI implementiert, funktioniert
3. einen C#-Compiler.

Es soll versucht werden die herkömmlichen C# Kommandozeilentools [vgl. 2.3] für Linux in C# zu implementieren sowie die .NET-Framework Klassenbibliotheken. Außerdem wird geplant die von der ECMA standardisierten Klassenbibliotheken und diverse andere Compiler, die auf der CLR basieren, zu implementieren [4].

6. Quellenangaben:

[1] iX 12/2001: „C#-und .NET-Tutorial: Teil1“;
http://www.heise.de/ix/artikel/2001/12/122; 6/8/2002
[2] c´t 04/2002: „Das Microsoft-Internet“; http://www.heise.de/ct/02/04/086/; 6/8/2002
[3] ECMA: http://www.ecma.ch
[4] The Mono Project: FAQ; http://www.go-mono.net/faq.html; 6/2/2002
[5] The C# Language Spezification, V. 0.28, 5/7/2002
[6] http://www.dotnetexperts.com/, 6/6/2002
[7] http://www.mircosoft.com/net/
[8] http://www.topxml.com/soapworkshop/articles/intro/default.asp
[9] http://www.pocketsoap.com/weblog/stories/2002/01/13/whatsWsdl.html
[10] http://www.w3c.org/TR/wsdl
[11] Java & XML, Brett McLaughlin, O´Reilly, 2002