.NET Framework

Konzepte von Betriebssystem-Komponenten

Wilhelm Haas

Wilhelm.Haas@informatik.stud.uni-erlangen.de
11. Januar 2005

1 Einleitung

Im Jahre 1998 lies Microsoft verlauten, dass sie mit der Entwicklung einer neuen Version des Component Object Models (COM)[1] begonnen haben. Sehr schnell wurde festgestellt, dass man sich weit von der COM Entwicklung bewegen würde und somit ein neuer Name erfunden werden müsste. Die Entwicklung wurde zunächst unter dem Namen „COM Version 3.0“ getrieben, später als „Next Generation Windows Service“. Im Juli 2000 wurde die alpha-Version unter dem heute weit verbreiteten Namen „.NET Framework“ herausgegeben.

Microsoft betrachtet das .NET Framework als Entwicklungsplattform für das Internet, das gleichzeitig eine neue Programmierschnittstelle für Anwendungen (API) unter dem Betriebssystem Windows bereitstellt. Dabei wurden auch herkömmliche Entwicklungen betrachtet und ausgewertet. Die Vorteile dieser sind mit in die Entwicklung eingeflossen.

2 .NET Framework

Ziel des .NET-Frameworks ist die Lösung der aktuellen Probleme der Entwicklerwelt. Viele Programmiersprachen mit Speziellen und unterschiedlichen Bibliotheken. Kaum portable Programme da viele Programmiersprachen nur betriebssystemabhängige Programme erstellen.

Lösung ist das .NET Framework. Es stellt im großen Umfang einheitliche Bibliotheken und Problemlösungen zur Verfügung mit einer einheitlichen Schnittstelle zur Verwendung der Bibliotheken. Ausserdem frei wählbare Programmierspachen mit denen sich betriebssystemabhängige oder betriebssystemunabhängige Programme erstellen lassen.

2.1 Struktur

Die Abbildung 1 zeigt die logische Struktur des .NET Frameworks. Gemäß der „Common Language Specification“ werden Quellcodes der .NET-Sprachfamilien in die „Microsoft Intermediat Language (MSIL-Code)“ übersetzt. Die Programmiersprachen werden von zahlreichen, für die Entwicklung von Webdiensten,

[1] Näheres zu COM siehe **http://de.wikipedia.org/wiki/Component_Object_Model**

Web Forms und Windows Forms vorgesehenen Klassenbibliotheken unterstützt. Das Grundgerüst für diese Bibliotheken und Funktionen auf niederer Ebene stellen dann die Basisklassenbibliotheken zur Verfügung (z.B. Ein- und Ausgabe, Strings, Hashset, usw.). Die wichtigste aller Schichten ist die „Common Language Runtime“, die die Ausführung der Programme übernimmt.

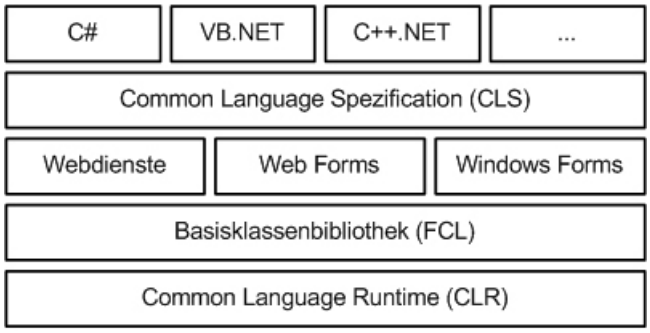

Abbildung 1: Struktur der .NET Plattform

2.2 Freie Wahl der Sprache

Aus wirtschaftlicher und programmiertechnischer Sicht verfolgt das .NET-Framework die Idee der „freien Wahl des passenden Werkzeugs“, d. h. es ist jedem Programmierer selbst überlassen in welcher Sprache er programmieren möchte. Es werden aber auch nicht nur mehrere Sprachen unterstützt. Diese unterschiedlichen Sprachen können auch gleichzeitig verwendet werden. So ist es zum Beispiel möglich eine Unterklasse einer VB.NET-Klasse aus C# ab zu leiten und die sich daraus ergebende Klasse anschließend im C++.NET-Code zu benutzen.

Zur Zeit existieren über 20 Programmiersprachen für das .NET-Framework und es werden immer mehr.

2.3 Common Language Specification (CLS)

Eines der Ziele der .NET-Strategie ist dem Programmierer die freie Wahl zu geben mit welcher Programmiersprache dieser seine Applikationen entwickeln möchte. So kann dieser die Vorteile der Vererbung, des Polymorphismus, ... nutzen. Das Problem ist nur, dass viele Programmiersprachen nicht gleich aufgebaut sind. So kann es sein, dass eine Sprache ein Feature bereit stellt das sich mit dem in einer anderen Programmiersprache total unterscheidet. Wenn man sich z.B. C++.NET und VB.NET betrachtet so fällt sofort in der Syntax auf, dass C++.NET zwischen Groß- und Kleinschreibung unterscheidet, was unter VB.NET keine Rolle spielt. Um aber die bijektive Abbildung

$$f : C++.NET \rightarrow VB.NET \qquad bzw. \qquad f : PS1 \rightarrow PS2$$

zu gewährleisten enthält das .NET-Framework die CLS. Als Zwischensprache der Abbildungen wird das „Microsoft Intermediat Language (MSIL)“ verwendet.

Die Common Language Specification ist ein Regelwerk für Compiler, das festlegt, wie die Umsetzung von sprachspezifischen Konzepten in die MSIL erfolgen muss. Mit dieser Spezifikation ist es jetzt für jeden möglich Compiler für die .NET-Umgebung zu schreiben.

Der Kern der CLS ist die „Common Type Specification (CTS)“. Jede Programmiersprache besitzt eigene Datenstrukturen. Die Menge dieser Datenstrukturen bilden das Typsystem, zu der sowohl einfache wie auch komplexe Typen gehören. Die Vielzahl an Typsystemen verschiedener Programmiersprachen hat zur Folge, dass es bei der Entwicklung von Software in unterschiedlichen Programmiersprachen zu Komplikationen kommt. Die einzelnen Komponenten können aufgrund verschiedener Typsysteme und Aufrufkonventionen nicht ohne Weiteres interoperieren. So muss z.B. die Äquivalenz einer in C++ geschriebenen Klasse mit einer in VB geschriebenen gewährleistet werden. Es muss also eine Einigung geschaffen werden wie Datentypen abgelegt werden. Die Einigung schafft Microsoft mit der Entwicklung des CTS. Dadurch wird sichergestellt, dass Variablentypen in einer Programmiersprache die gleiche Semantik besitzen wie in einer anderen .NET Programmiersprache.

2.4 Microsoft Intermediate Language (MSIL)

Jedes in einer .NET-Sprache geschriebene Programm wird durch das Compilieren in eine Zwischensprache übersetzt, die MSIL (wird auch als Common Intermediate Language CIL bezeichnet). Diese Zwischensprache ist eine streng typisierte, prozessorunabhängige objektorientierte Assembler-Sprache speziel für das .NET-Framework entwickelt. Wie jede moderne Sprache kennt diese alle Konzepte einer modernen Programmiersprache, wie z.B. Vererbung, Polymorphismus, Exception-Handling und vieles andere.

Die Übersetzung erfolgt nach den Regeln der „Common Language Specification“. Sie ist der Schlüssel für die Interoperabilität der .NET-Sprachen.

Theoretisch ist es möglich die Programme auch in der MSIL zu programmieren, wie dies auch mit der Assembler-Sprache möglich ist. Dies ist aber in den meisten Situationen nicht erforderlich.

2.5 Assemblies

Das .NET-Framework ist nicht nur objektorientiert, sondern auch komponentenorientiert. Im Mittelpunkt des Komponentenkonzepts stehen die sogenannten Assemblies.

Eine Assembly ist ein logischer Verbund von einer oder mehreren EXE- oder DLL-Dateien, die den Programmcode und Resourcen enthält. Eine DLL-Assembly ist immer eine wiederverwendbare Softwarekomponente, die von einer anderen Assembly genutzt werden kann. Eine EXE-Assembly lässt sich mit der „Common Language Runtime“ als eine unabhängige Anwendung starten. Eine EXE-Assembly kann aber auch Dienste für andere bereitstellen.

Eine Assembly kann aus mehreren Teilen bestehen (Abbildung 2):

- Der **CLR Header** enthält Informationen für die „Common Language Runtime“

- Das **Manifest** beschreibt die Assembly. Diese enthält die Version der Assembly, Liste der dazugehörigen Assemblies, externe Assemblies, Resourcen, externe Resourcen, Zugriffsrechte.
- Der **MSIL-Code** ist der Code, der von der „Common Language Runtime“ ausgeführt wird.
- Die **Metadaten** enthalten weitere Details zu dem MSIL-Code, wie z. B. Typdefinitionen, Informationen über die Version.
- **Resourcen** sind z. B. Bilder, Texte, Datenbanken.

Ein Verbund besteht aus mindestens einer oder mehreren Assemblies, die MSIL-Code enthalten, wobei mindestens eine der Dateien eine DLL oder EXE ist. Optional können auch Ressource-Datein ein Teil der Assembly sein, diese können aber auch direkt in die Assembly eingebettet werden. Welche Dateien zum Verbund gehören, wird durch ein Manifest bestimmt. Der CLR Header enthält Informationen zum Laden des Runtime Hosts. MSIL-Dateien, die kein Manifest enthalten, werden Modul genannt.

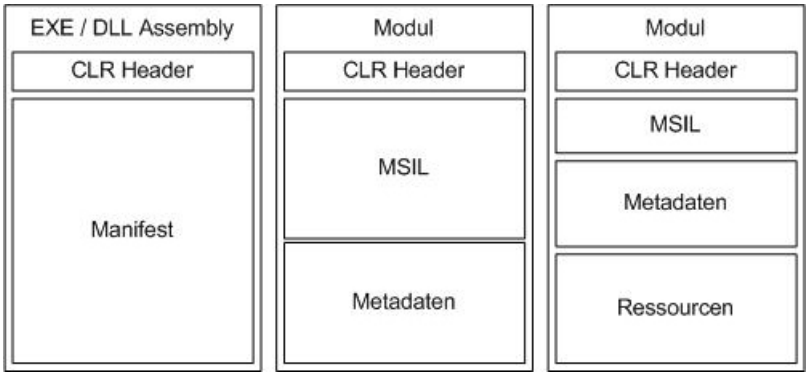

Abbildung 2: Assembly-Verbund

2.6 Framework Class Library (FCL)

Die Klassenbibliothek des .NET Framework bietet mit einer Reihe von Klassen, Schnittstellen und Werttypen Zugriff auf die Systemfunktionen, die die Grundlage für jede .NET-Anwendung bilden (Datenverwaltung, XML-Manipulationen, Windows Forms, Web Forms, Web Services, ...). Diese Bibliotheken ermöglichen eine sehr schnelle und einfache Entwicklung von Anwendungen.

Die Basis für die FCL[2] bildet die Basic Class Library[3] (BCL). Dazu zählen Klassen z. B. für die Ein- und Ausgabe, Threading, Netzwerkkommunikation, String-Manipulation. Jede Klasse der FCL ist eine Unterklasse der Basisklasse **System.Object**.

Mit der FCL stehen jeder Sprache der .NET-Familie die gleichen Möglichkeiten zur Verfügung. In der Version 1.0 des .NET-Frameworks sind 2246 öffentliche

[2]Liste der FCL: **http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/cpref_start.asp**
[3]Auflistung der BCLs: **http://www.gotdotnet.com/team/clr/bcl/default.aspx**

Klassen enthalten. Diese liegen klar in Namensräumen strukturiert. Die Wurzel der FCL Namensräume ist der Namensraum **System**.

2.7 Common Language Runtime (CLR)

Der Kern des .NET-Frameworks ist die „Common Language Runtime“[4]. Sie verwaltet und führt den MSIL-Code aus. Die CLR ist die Basis der .NET-Architektur, vergleichbar mit der Java Virtual Machine, wird auch als Virtual Execution System (VES) bezeichnet.

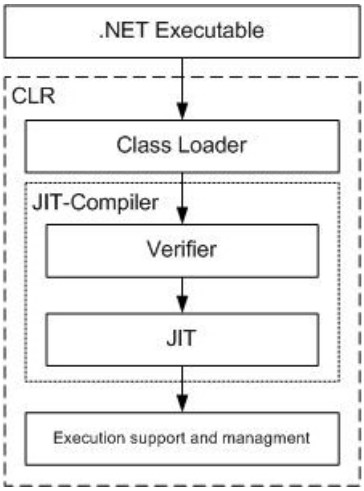

Abbildung 3: CLR's Virtual Execution System

Die Hauptbestandteile der CLR sind der „Class loader“, „Verifier“, „JIT-Compiler“ und „Execution support and managment“ der z. B. folgende Dienste bereit stellt: security management, garbage collection, exception management.

Class Loader

Der „Class Loader“ lädt die für die Ausführung benötigten Klassen. Die Information, welche Klassen geladen werden müssen, holt sich der „Class Loader“ aus dem Manifest der Assembly. Ist die zuladende Klasse gefunden worden erweitert der „Class Loader“ die Methoden der Klasse. Es fügt zu jeder Methode einen Stub hinzu der z. B. den Status der JIT-Compilation enthält. Wenn die geladene Klasse auf eine Andere referenziert, so wird probiert auch diese zu laden. Schließlich werden die statischen Variablen initialisiert und eine Instanz der Klasse erzeugt.

Verifier

Der Verifier ist ein Schutzmechanismus der für Typsicherheit sorgt. Script und interpretierende Sprachen sind sehr nachsichtig beim Umgang mit den Typen. So gibt es Sprachen die erlauben Code zu schreiben ohne jemals eine Variable explizit zu deklarieren. Diese Flexibilität kann zu einem Code führen der

[4]Siehe Buch: **O'Reilly .NET Framework Essentials, 2nd Edition**

fehleranfällig und schwer zu warten ist. Im Gegensatz dazu verlangen die zu compilierenden Sprachen eine explizite Definition mit der Annahme das die Typen korrekt und der Code nicht schädlich ist. So ist es möglich in C in ein Array mehr Daten zu schreiben als ursprünglich vorgesehen war ⇒ memory overwriting mit verherenden Folgen.

Die Lösung des Problems ist eine Gewährleistung der Typsicherheit, das ein fundamentales Konzept für die Code-Verifizierung in .NET ist. Zur Laufzeit überprüft der Verifier ob der Code typsicher ist. Der Nachteil dieses Konzepts ist der Performance-Verlust durch die ständige Überprüfung auf Typsicherheit. Deshalb ist es möglich durch den Erwerb eines Zertifikats die Verifikation abzustellen.

JIT - Just in Time Compiler

Die Hauptrolle der .NET-Plattform spielt der JIT. Dieser konvertiert den MSIL-Code in den nativen Code, so daß dieser auf der Maschine ausgeführt werden kann. Zur Performance-Steigerung kompiliert der JIT nur die Methoden die gebraucht werden. Diese werden aber nur einmal konvertiert und dann im Arbeitsspeicher gelassen. Der JIT erkennt an einem Flag in dem Stub, der von dem „Class Loader“ für jede Methode erstellt wird, ob die Methode schon übersetzt worden ist.

Ein Vorteil des JIT-Compilers ist, dass Code dynamisch compiliert wird und somit für jede Zielmaschine optimiert wird. Darüberhinaus ist man Betriebssystemunabhängig und auch Hardwareunabhängig.

Will man sich die teuere Übersetzungszeit zur Laufzeit sparen so kann man die Assemblies mit dem „Ngen (Native Image Generator) Tool“ sofort in nativen Code übersetzen. Je nach Anwendung ist eines der beiden Verfahren zu bevorzugen.

3 Zukunft und Gegenwart

Die Kinderkrankheiten des .NET-Frameworks Version 1.0 sind zum größten Teil beseitigt worden. Heute ist das .NET-Frameworks in Version 2.0 erhältlich.

Die Entwicklung ist in der Wirtschaft schon weit verbreitet deshalb ist es jedem angehenden Entwickler in der freien Wirtschaft empfehlenswert sich mit dem .NET-Framework zu befassen.

Das Framework wird mit dem neuen Betriebssystem Windows Longhorn mit ausgeliefert.

4 Literatur

1. **Dot Net Framework**
   Link: http://www.dotnetframework.de
2. **Microsoft .NET Einführung**
   Link: http://www.tfh-berlin.de/~godberse/wiese/ia2_200312/DotNet.pdf
3. **O’Reilly .NET Framework Essentials, 2nd Edition**
   Autor: Thuan Thai, Hoang Q. Lain
   ISBN: 0-596-00302-1

4. **Introducing Microsoft .NET, 2nd Edition**
   Autor: David S. Platt
   ISBN: 0-735-61571-3

5. **Die .NET Philosophie**
   Autor: verlag moderne industrie Buch AG & Co. KG, Bonn
   Link: http://www.mitp.de/imperia/md/content/vmi/1311/1311_kap01.pdf

6. **MSDN .NET Framework Technologie Information**
   Autor: Microsoft Development Network
   Link: http://msdn.microsoft.com/netframework/technologyinfo/default.aspx