

Kapitel 4

Grundkonzepte des .NET Framework 4.0

In diesem Kapitel:

Zwischensprache	108
Laufzeitumgebung	111
Programmiersprachen	112
Objektorientierung	118
.NET-Klassenbibliothek (FCL)	127
Softwarekomponentenkonzept	133
Verbreitung und Installation von .NET-Anwendungen	143
Weitere Fähigkeiten der Laufzeitumgebung	146
Interoperabilität	153
.NET auf 64-Bit-Systemen	155
Versionskompatibilität	159

Zwischensprache

Sowohl die Programmiersprache Java als auch das .NET Framework basieren auf dem Konzept *Write Once Run Anywhere* (WORA), d.h., eine einmal entwickelte und kompilierte Anwendung kann auf verschiedenen Betriebssystemen ablaufen. Das .NET Framework arbeitet – genau wie Java – mit einem Intermediationskonzept auf Basis einer Zwischensprache und einer virtuellen Maschine (WM).

Native Code versus Managed Code

Ein Compiler einer .NET-Hochsprache erzeugt in .NET keinen prozessorspezifischen Maschinencode, sondern einen plattformunabhängigen Zwischencode. Dieser Zwischencode wird *Microsoft Intermediate Language* (MSIL) oder – im ECMA- und ISO-Standard – *Common Intermediate Language* (CIL) genannt. Code in MSIL wird auch als *verwalteter Code* (*Managed Code*) bezeichnet. Das Kompilat ist auch bei Managed Code eine DLL-Datei oder eine EXE-Datei, wofür man als Oberbegriff den Begriff *Assembly* (in der deutschen Version zum Teil mit *Assemblierung* übersetzt) verwendet.

Im Gegensatz dazu wird prozessorspezifischer Maschinencode als *nicht-verwalteter* (*Unmanaged Code*) oder *Native Code* bezeichnet. Assemblys sind immer Managed Code. Es können darin aber Inseln von Native Code existieren.

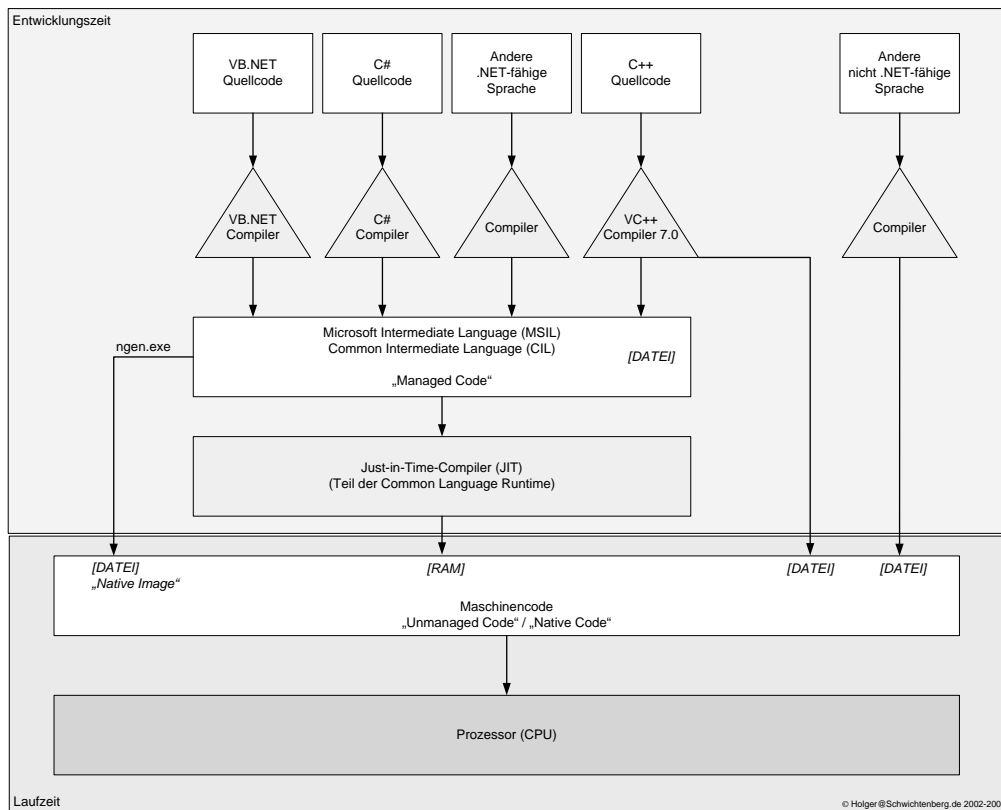


Abbildung 4.1 Zwischensprachkonzept im .NET Framework

Just-in-Time-Compiler

Erst zur Laufzeit wird der MSIL-Code dann in einen prozessorspezifischen Maschinencode (*Native Code*) umgewandelt. MSIL-Code wird nicht interpretiert, sondern von einem so genannten *Just-in-Time-Compiler* stückchenweise umgewandelt und dann ausgeführt. Dabei berücksichtigt der Just-in-Time-Compiler prozessorspezifische Optimierungsmöglichkeiten. Dadurch, dass nicht interpretiert, sondern vor der Ausführung kompiliert wird und der Just-in-Time-Compiler sehr schnell arbeitet, ist der Leistungsverlust durch die Intermediation sehr gering. Im Zweifel gibt es auch die Möglichkeit, das Ergebnis der Kompilierung von MSIL zu Maschinencode zu speichern und später auszuführen. Dies nennt man *Native Image*. Ein Native Image ist jedoch nicht mehr plattformunabhängig. Es ist nicht verboten, dass Sprach-Compiler wahlweise auch direkt Native Code erzeugen, der nicht unter der Kontrolle der .NET-Laufzeitumgebung abläuft.

Der *Just-in-Time-Compiler* (*JIT-Compiler* oder *JITter*) ist Teil der .NET-Laufzeitumgebung, die *Common Language Runtime* (CLR) genannt wird. Das Intermediationskonzept ist die Basis für die Plattformunabhängigkeit der Anwendungen. Managed Code kann auf jedem Betriebssystem ausgeführt werden, für das eine Implementierung der Common Language Runtime verfügbar ist.

Natürlich ist Managed Code langsamer als Native Code, wobei der Geschwindigkeitsunterschied sehr viel geringer ist, als man vermutet. Wenn jedoch die optimale Geschwindigkeit notwendig ist, besteht die Möglichkeit, eine Managed-Code-Datei einmalig in eine Native-Code-Datei umzuwandeln. Dazu dient das Werkzeug *ngen.exe*. Der Vorgang wird als *Pre-Jitting* bezeichnet. Das Resultat von *ngen.exe* nennt man ein *Native Image*. *Ngen.exe* wurde schon in .NET 2.0 stark vereinfacht (z.B. werden nun alle referenzierten Komponenten automatisch mitübersetzt).

Grundsätzlich kann zwar ein Native Image bereits während der Entwicklung erzeugt werden; aufgrund der plattformspezifischen Übersetzung bietet es sich jedoch an, das Native Image erst bei der Installation einer .NET-Anwendung zu erzeugen (*Install Time Compilation*). Das Native Image ist spezifisch für eine bestimmte Version der .NET-Laufzeitumgebung. Auch benötigen DOS- und NT-basierte Windows-Systeme verschiedene Native Images. Eine .NET-Anwendung, die in einem Native Image gespeichert ist, braucht ebenfalls die .NET-Laufzeitumgebung, um ausgeführt werden zu können.

Die CIL-Sprache

Das nachfolgende Listing zeigt ein »Hello World«-Beispiel in CIL-Code. Interessant dabei ist, dass Unter-routinen nicht – wie in vielen Programmiersprachen – über ihre Position in einer Liste von Funktionen (*vTable*-Verfahren) angesprungen werden, sondern dass der komplette Methodenname im Code verewigt ist. Damit ist CIL robust gegenüber Änderungen der Schnittstelle.

```
.method public static void main() cil managed
{
    .entrypoint
    .custom instance void [mscorlib]System.STAThreadAttribute::.ctor() = ( 01 00 00 00 )
    // Code size      11 (0xb)
    .maxstack 8
    IL_0000: ldstr      "Hello World aus der Anwendung !!"
    IL_0005: call      void [mscorlib]System.Console::WriteLine(string)
    IL_000a: ret
} // end of method Anwendung1::main
```

Listing 4.1 Beispiel für eine »Hello World«-Ausgabe in CIL

Decompiler

Der Nachteil des sehr »sprechenden« Verfahrens ist, dass bei der Dekompilierung von CIL-Code sehr gut lesbarer Hochsprachen-Programmcode entsteht. Das geistige Eigentum des Softwareentwicklers ist also nicht gut geschützt.

Microsoft selbst stellt im .NET Framework SDK einen Decompiler bereit, der CIL-Code anzeigt (*ildasm.exe*). Zwei Decompiler, die C#-Code aus CIL-Code erzeugen, erhalten Sie als Freeware im Internet (z.B. *.NET Reflector* [LROED01]).

Obfuskatoren

Gegenspieler der Klassen-Browser und Decompiler sind *Obfuskatoren*, die durch Ersetzungen das intellektuelle Eigentum von Software-Autoren besser schützen sollen. Die Methoden und Variablen erhalten kryptische Namen und erschweren es so dem Disassembler-Nutzer, ein Programm zu verstehen. Einige Obfuskatoren verschlüsseln zudem Zeichenkettenlitterale und bauen den Kontrollfluss so um, dass er kaum noch zu verstehen ist. Die Semantik bleibt dabei erhalten, jedoch werden die Assemblys verständlicherweise größer und langsamer. Es existieren zahlreiche Obfuskatoren für .NET (siehe [DOTNET02]).

Microsoft liefert mit Visual Studio zusammen eine funktionsreduzierte Version des DOTFUSCATOR der Firma PreEmptive Solutions aus. Das Produkt Xenocode [XEN01] beherrscht darüber hinaus ein Verfahren, mit welchem die Metadaten einer Assembly so verfälscht werden, dass weder ILDASM noch der .Net Reflector die Assembly überhaupt öffnen können. Die Funktionalität bleibt aber erhalten. Auch der Salamander .Net Protector schützt komplett vor der Disassemblierung durch Übersetzung in ein anderes Format (Pseudo-Native-Code). Durch zusätzliche Verschlüsselung soll laut dem Hersteller Remotesoft jeder Decompiler chancenlos sein.

Secure Virtual Machine (SVM)

Secure Virtual Machine (SVM) ist die Bezeichnung für eine spezielle Virtual Machine (Ablaufumgebung) für .NET-Anwendungen, die im Gegensatz zu der Standard-VM der CLR die Dekompilierung stark erschwert.

Durch so genannte Permutationen wird für jeden Hersteller oder sogar jedes Produkt eine eigene SVM mit einem eigenen Befehlssatz erstellt. Die eigene SVM besitzt einen eigenen, undokumentierten Befehlssatz. Anwendungen werden nach der eigentlichen Kompilierung nach MSIL/CIL nochmals umgewandelt in die SVM-spezifische Zwischensprache. Die Anwendung kann danach nur noch mit der SVM ausgeführt werden, was bedeutet, dass die SVM mit ausgeliefert werden muss. Eine SVM hat eine Größe von rund 1 MB.

Zusätzlich zu dem eigenen Befehlssatz verwendet die SVM auch noch Verschlüsselung, um die Dekompilierung nochmals zu erschweren. Eine Dekompilierung einer SVM-geschützten Anwendung ist jedoch nicht komplett unmöglich.

Wichtig ist, dass der SVM-Schutz die Leistung (Ausführungsgeschwindigkeit) wesentlich reduziert. Daher sollte man niemals eine ganze Anwendung, sondern allenfalls besonders sensible und/oder innovative Bereiche einer Anwendung schützen.

Software Licencing and Protection Services (SLPS) ist ein Produkt (früher von Microsoft, jetzt eigenständig durch *InishTech*, siehe [SLPS01]) mit folgenden Leistungsmerkmalen:

- Schutz von .NET-Anwendungen vor Dekompilierung
- Produktaktivierung von .NET-Anwendungen (auch modulweise)

SLPS besteht aus folgenden Bausteinen:

- Dem Konzept der Secure Virtual Machines (SVM)
- Einem Werkzeug zur Umwandlung von MSIL-Code in SVM-IL-Code (SLP Code Protector)
- Einem Server bei Microsoft zur Produktaktivierung (darin ein Webportal für den Hersteller zur Definition von Produkten und Features und zur Vergabe von Produktschlüsseln)
- Einem Serversystem, das Kunden von Microsoft kaufen können, um selbst einen Server zur Produktaktivierung bereitzustellen

Laufzeitumgebung

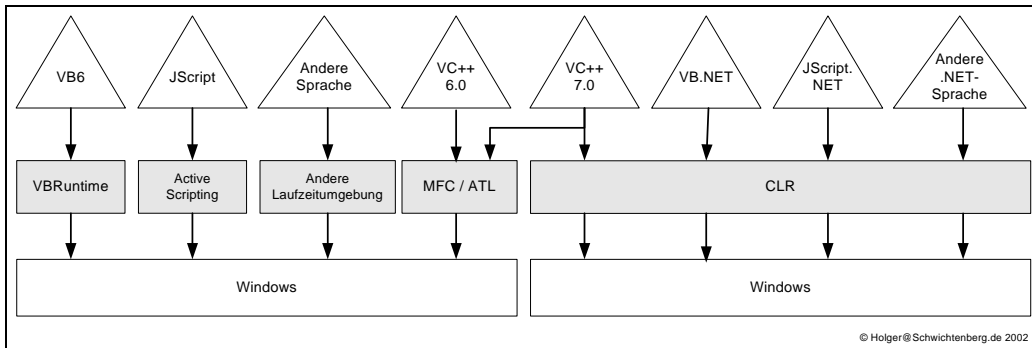
Die Ausführung einer .NET-Anwendung setzt eine Laufzeitumgebung, die Common Language Runtime (CLR), voraus.

Common Language Runtime (CLR)

Die CLR stellt den Just-in-Time-Compiler und zahlreiche andere Basisdienste bereit, die von allen .NET-fähigen Sprachen verwendet werden. Dazu gehören zum Beispiel

- eine automatische Speicherverwaltung durch einen Garbage Collector,
- ein System für eine Ausnahmebehandlung (Exception Handling),
- ein Sicherheitssystem, das die Anwender vor böartigem Code schützen kann,
- die Abgrenzung von Anwendungen durch Application Domains und
- die Interoperabilität mit Nicht-.NET-Anwendungen

Die CLR löst das Problem, dass bisher jede Programmiersprache ihre eigene Laufzeitumgebung benötigt hat und dass diese Laufzeitumgebungen sehr verschiedene Dienste bereitgestellt haben, was deutliche Unterschiede in der Programmierweise und im Programmierkomfort mit sich gebracht hat.

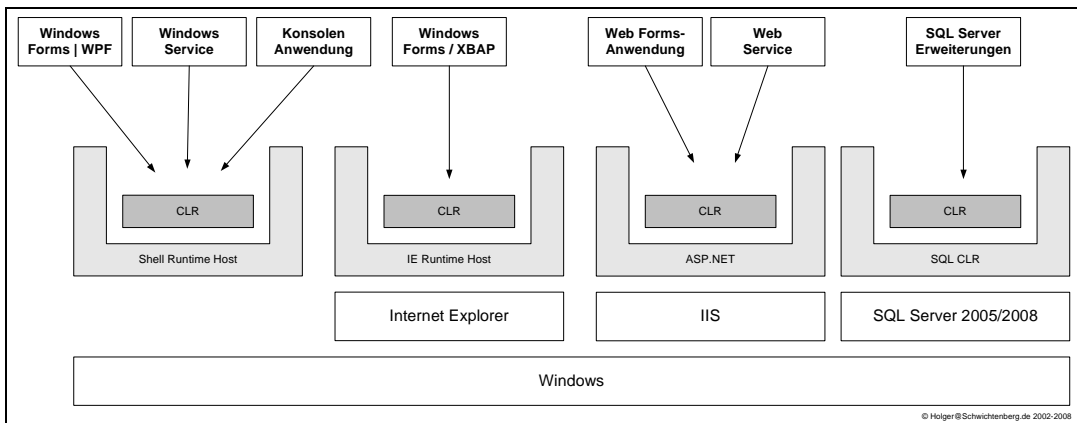


© Holger@Schwichtenberg.de 2002

Abbildung 4.2 Die CLR als einheitliche Laufzeitumgebung

.NET Runtime Host

Damit eine .NET-Anwendung ausgeführt werden kann, ist es notwendig, die CLR in den Speicher zu laden und die .NET-Anwendung zur Ausführung an die CLR zu übergeben. Dies ist die Aufgabe eines .NET Runtime Hosts (alias Application Domain Host). Für jeden Typ von .NET-Anwendungen muss es einen Runtime Host geben. Jeder Runtime Host besitzt am Anfang einen Teil Unmanaged Code, der *Stub* genannt wird. Dieser Stub kann nicht Managed Code sein, weil die CLR ja noch nicht geladen ist. Es ist die Aufgabe des Stub, die CLR zu laden. Dabei benutzt der Runtime Host das so genannte .NET Hosting API.



© Holger@Schwichtenberg.de 2002-2006

Abbildung 4.3 Anwendungstypen und Runtime Hosts

Programmiersprachen

Während die Programmiersprache Java in den 1990er Jahren mit dem Leitsatz »Eine Sprache für alle Plattformen« auf dem Markt angetreten ist, zielte Microsoft zunächst mehr auf »Eine Plattform für alle Sprachen« ab. Von Anfang an stand die Integration zahlreicher verschiedener Programmiersprachen im Mittelpunkt von .NET. Inzwischen weichen die beiden gegensätzlichen Positionen von Java und .NET auf: Es gibt .NET-Laufzeitumgebungen für andere Plattformen und es gibt Compiler für andere Programmiersprachen, die auch Java-Bytecode erzeugen können.

Insgesamt existieren mittlerweile über 70 verschiedene Programmiersprachen für .NET. Darunter sind nicht nur objektorientierte Sprachen wie C#, Java, C++, Visual Basic und Delphi, sondern auch funktionale Sprachen wie SML, Caml und Haskell sowie »alte Tanten« wie Fortran und Cobol vertreten. Es gibt sowohl kommerzielle als auch kostenlose Sprachen. Eine ständig aktualisierte Liste der .NET-Programmiersprachen finden Sie unter [DOTNET01].

Name	Basissprache	Hersteller	Website
Abstract State Machine Language (ASML)		Microsoft Research	http://research.microsoft.com/fse/asml/
Active Oberon for .Net	Oberon	ETH Zentrum Zürich	http://www.oberon.ethz.ch/lightning/
ASharp	Ada	United States Air Force Academy	http://www.usafa.af.mil/df/dfcs/bios/mcc_html/a_sharp.cfm
Chrome	Object Pascal	RemObjects	http://www.chromesville.com/page.asp?id={C5B896C5-5C61-4C1C-A617-136711C07F46}
Comega		Microsoft Research	http://research.microsoft.com/Comega/
CSharp (C#)	ECMA 334	Microsoft	http://www.microsoft.com
CULE.NET	Visual Objects/XBase	GOGETR Software Corporation	http://www.softwareperspectives.com/CULEPlace/
Delphi .NET	ANSI/ISO Pascal	Borland	http://www.borland.de
Delta Forth .NET	Forth	Valer Bocan	http://www.dataman.ro/dforth/
DotLisp	Lisp	Rich Hickey	http://sourceforge.net/projects/dotlisp
Dyalog.Net	APL (ISO 8485)	Dyalog Limited	http://bewwww.dyalog.com
Fortran for .NET	Fortran 95	Lahey Computer Systems, Inc.	http://www.lahey.com/dotnet.htm
FSharp (F#)	OCaml	Microsoft Research / Microsoft	http://msdn.microsoft.com/en-us/fsharp/default.aspx
FTN95 for Microsoft .NET	Fortran 77, Fortran 95, Fortran 2000	Salford Software Ltd.	http://www.ftn95.net
Gardens Point Component Pascal	Weiterentwicklung von Pascal, Modula-2 und Oberon-2	Queensland University of Technology	http://plas.fit.qut.edu.au/gpcp/.NET.aspx
Hugs98 for .NET	Haskell 98	Mark P. Jones, Alastair Reid, Yale Haskell Group, Oregon Graduate Institute of Science and Technology	http://galois.com/~sof/hugs98.net/
IronPython	Python	Microsoft	http://www.IronPython.com
ISE Eiffel Studio	Eiffel (ECMA TC39-TG4)	Eiffel Software	http://www.eiffel.com/
JScript .NET	ECMA 262, 290 und 327	Microsoft	http://www.microsoft.com
JSharp (J#)	Java 1.1.4	Microsoft	http://www.msdn.microsoft.com/downloads/default.asp?url=/downloads/sample.asp?url=/MSDN-FILES/027/001/973/msdncompositedoc.xml

Name	Basissprache	Hersteller	Website
Lua.NET	Lua	Roberto Ierusalimsky, Renato Cequeira, Fabio Mascarenhas	http://www.lua.inf.puc-rio.br/luanet/
Mercury.NET	Mercury	University of Melbourne	http://www.cs.mu.oz.au/mercury/dotnet.html
MixNet	Mixal	Community-Projekt	http://sourceforge.net/projects/mixnet/
Mondrian for .NET	Mondrian	Massey University	http://www.mondrian-script.org/
MonoLOGO	Berkeley LOGO / ObjectLOGO	Rachel Hestilow	http://monologo.sourceforge.net/
Multi-Target Pascal	Pascal	TMT Development	http://www.tmt.com/
Nemerle		University of Wroclaw	http://nemerle.org/Main_Page
NetCOBOL	Object-Oriented COBOL	Fujitsu	http://www.netcobol.com/
NetRuby	Ruby	Arton	http://www.geocities.co.jp/SiliconValley-PaloAlto/9251/ruby/nrb.html
PerlNet	Perl	ActiveState	http://aspn.activestate.com/ASPN/NET
PHP_Sharp	PHP	Community-Projekt	http://sourceforge.net/projects/php-sharp/
PSharp (P#)	Prolog	Jon Cook (University of Edinburgh)	http://homepages.inf.ed.ac.uk/jcook/
Python for .NET	Python	Mark Hammond	http://starship.python.net/crew/mhammond/dotnet/
Ruby .NET	Ruby	Ben Schroeder, John Pierce	http://www.saltpickle.com/rubydotnet
Scheme.NET	Scheme	Northwestern University	http://www.cs.indiana.edu/~jgrinbla/
Sharp Smalltalk (#Smalltalk)	Smalltalk	Refactory Inc	http://www.refactory.com/Software/SharpSmalltalk/
SmallScript.NET (S#.NET)	Smalltalk	SmallScript Corporation	http://www.smallscript.org/
Squeak .NET	Squeak	Ben Schroeder, John Pierce	http://www.saltpickle.com/squeakDotNet
Standard Meta Language (SML.NET)		Microsoft Research	http://www.research.microsoft.com/Projects/SML.NET/index.htm
Visual Basic (VB)	Basic	Microsoft	http://www.microsoft.com
Visual C++/CLI	C++	Microsoft	http://www.microsoft.com
Visual RPG for .NET	RPG/Caviar	ASNA	http://www.asna.com/pages/products_NET_AVR.aspx
Vulcan.NET	Visual Objects/XBase	GrafX Software	http://www.vulcandotnet.de

Tabelle 4.1 Programmiersprachen für das .NET Framework (Auswahl)

Sprachen von Microsoft

Microsoft selbst liefert aktuell folgende .NET-Sprachen:

- Visual Basic (Visual Basic .NET)
- C# (CSharp)
- F# (FSharp)
- JScript .NET (ein Derivat von JavaScript)
- C++/CLI und
- IronPython

Die Sprache J# (JSharp, ein Java-Derivat) wird seit .NET 2.0 nicht mehr weiterentwickelt.

Die Compiler der Sprachen Visual Basic, J#, C# und JScript .NET erzeugen immer Managed Code (alias MSIL-Anweisungen). Der Microsoft C++-Compiler (ab Version 7.0) erzeugt wahlweise Managed Code oder direkt Native Code. IronPython ist eine .NET-basierte Interpreter-Sprache.

	C#	Visual Basic (.NET)	C++/CLI	JScript .NET	J#	F#
.NET 1.0	1.0/7.0	7.0	7.0	7.0	1.0?	
.NET 1.1	1.1/7.1	7.1	7.1	7.1	1.1	
.NET 2.0	2.0 (2005)/8.0	8.0 (2005)	8.0 (2005)	8.0 (2005)	2.0	1.x
.NET 3.0	2.0 (2005)/8.0	8.0 (2005)	8.0 (2005)	8.0 (2005)	Nicht mehr aktualisiert	1.x
.NET 3.5	3.0 (2008)/3.5	9.0 (2008)	9.0 (2008)	9.0 (2008)	Nicht mehr aktualisiert	1.x
.NET 3.5 SP1	3.0 (2008)/3.5	9.0 (2008)	9.0 (2008)	9.0 (2008)	Nicht mehr aktualisiert	1.x
.NET 4.0	4.0 (2010)	10.0 (2010)	10.0 (2010)	10.0 (2010)	Nicht mehr aktualisiert	2.0

Tabelle 4.2 Versionsnummer der Microsoft .NET-Sprachen

Zur Erläuterung der Tabelle: Die Sprachen referenziert Microsoft neben Ihrer Versionsnummer oft auch über die Jahresbezeichnung, z.B. C# 4.0 ist gleich C# 2010.

Eine Besonderheit gibt es bei C#: Hier differiert vor .NET 4.0 die Versionszählung der Sprache von der Versionszählung des Compilers. Die erste Zahl in der C#-Spalte vor dem Schrägstrich ist die Versionsnummer der Sprachsyntax, die hintere Zahl die Versionsnummer, unter der sich der Compiler meldet. Bis einschließlich .NET 3.0 zählte Microsoft den C#-Compiler wie den Visual Basic-Compiler. Seit .NET 3.5 zählt Microsoft den C#-Compiler wie das .NET Framework.

TIPP

Da die .NET-Laufzeitumgebung bereits Kommandozeilen-Compiler für die Sprachen Visual Basic, C# und JScript .NET mitliefert und Compiler für F# und C++/CLI als kostenlose Erweiterungen verfügbar sind, ist grundsätzlich die Anschaffung von Visual Studio nicht zwingend nötig, um mit .NET zu programmieren. .NET-Programme können mit jedem beliebigen Texteditor geschrieben und mit dem in .NET Framework Redistributable vorhandenen Compilern kostenlos übersetzt werden. Visual Studio macht die Entwicklung von VB.NET-Anwendungen jedoch wesentlich komfortabler. Von Visual Studio gibt es seit 2005 auch kostenfreie Express-Varianten (siehe Kapitel 5 zu Visual Studio).

Das Korsett von CTS und CLS

Die Basis für diese babylonische Sprachenvielfalt ist das *Common Type System (CTS)*. Das CTS umfasst die minimalen Anforderungen an jede .NET-Sprache, damit sie überhaupt lauffähige Typen auf der CLR erzeugen kann. Alle .NET-Programmiersprachen müssen sich dem CTS unterwerfen.

Das .NET Framework will nicht nur erreichen, dass verschiedene Sprachen die gleiche Laufzeitumgebung und Klassenbibliothek nutzen können, sondern auch, dass ein Entwickler innerhalb einer einzigen Anwendung verschiedene Programmiersprachen verwenden kann (Cross-Language Interoperability). Dies erhöht die Wiederverwendbarkeit von Programmcode drastisch. Die gegenseitige Nutzung von Code in unterschiedlichen Programmiersprachen lässt sich dabei in zwei Untergebiete aufteilen:

- Gegenseitiger Aufruf von Unterrouinen (Cross-Language Calls)
- Gegenseitige Vererbung (Cross-Language Inheritance)

Die Zwischensprache MSIL und die Laufzeitumgebung CLR alleine reichen für diese hehren Ziele nicht aus, denn zwei Codeteile können nicht korrekt miteinander interagieren, wenn sie in zwei verschiedenen Sprachen entwickelt wurden, die ein unterschiedliches Verständnis davon haben, wie viele Bits ein Integer umfasst (16 oder 32) oder wie eine Zeichenkette im Speicher abzulegen ist (als 0 terminierte Byte-Folge oder mit der Angabe der Länge in den ersten Bytes).

Dazu ist es notwendig, dass sich die Programmiersprachen in ein noch viel engeres Korsett zwingen, als es das CTS vorgibt. Dieses Korsett heißt *Common Language Specification (CLS)*. Die CLS ist eine Untermenge des CTS, in der strengere Regeln herrschen. Es gibt sie wieder in zwei Größen:

- *CLS Consumer* sind .NET-fähige Programmiersprachen, die Cross-Language Calls unterstützen
- *CLS Extender* sind .NET-fähige Programmiersprachen, die zusätzlich Cross-Language Inheritance erlauben

Verboten gemäß CLS sind u.a.:

- Vorzeichenlose Datentypen, wie z. B. uint, ulong
- Bezeichner für Klassen, Methoden, Attribute usw., die sich nur durch eine unterschiedliche Groß- und Kleinschreibung unterscheiden
- Globale statische Methoden
- Schnittstellen mit statischen Methoden oder Feldern
- Klassen, die nicht von CLS-kompatiblen Klassen erben

- Arrays variabler Größe oder Arrays, die nicht bei Element 0 beginnen
- Überladen von Attributen und Ereignissen
- Zeiger (*Pointer*)

Dabei ist die CLS-Unterstützung für eine Programmiersprache kein »ganz-oder-gar-nicht«; eine Sprache darf durchaus Elemente enthalten, die über die CLS hinausgehen. Sprachen, die nachträglich auf das .NET Framework adaptiert wurden (z.B. C++ und Fortran), besitzen zahlreiche nicht CLS-kompatible Konstrukte. Aber auch Visual Basic und die speziell für das .NET Framework entwickelte Sprache C# verfügen über einige nicht CLS-kompatible Konstrukte, z.B. die nicht vorzeichenbehafteten Ganzzahlen `UInt16`, `UInt32`, `UInt64`. Auch die .NET-Klassenbibliothek enthält einige wenige nicht CLS-kompatible Typen.

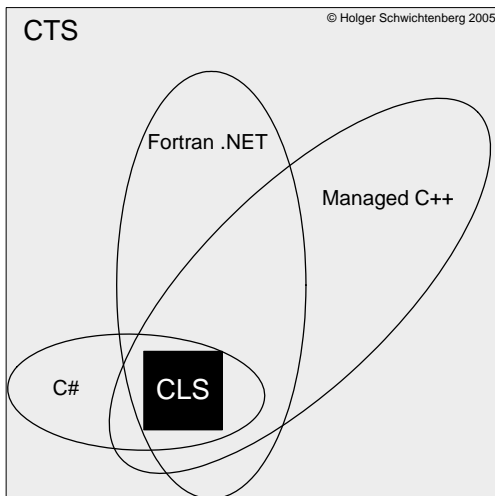


Abbildung 4.4 CTS versus CLS

Die Verwendung nicht CLS-kompatibler Konstrukte ist so lange unkritisch, wie diese Konstrukte nur in den Innereien von Softwarekomponenten verwendet werden. Alle von außen nutzbaren Typen (also in außerhalb der Assembly zugänglichen Klassen/Schnittstellen) sollten jedoch nur CLS-kompatible Konstrukte verwenden.

TIPP Ein Entwickler kann mit der Annotation `[Assembly: System.CLSCompliant(true)]` deklarieren, dass eine Softwarekomponente nur CLS-kompatible Typen veröffentlichen darf. Bei Missachten dieser Regel erzeugen die Compiler von Visual Basic und C# eine Warnung. Einzelne Klassen oder Klassenmitglieder kann man mit `[System.CLSCompliant(false)]` von der Prüfung ausnehmen.

CIL-Name	CLS-Typ	Name in der Klassenbibliothek	Beschreibung
bool	Ja	System.Boolean	True / false
char	Ja	System.Char	Einzelnes Unicode-Zeichen (16 Bit)
object	Ja	System.Object	Objekt
string	Ja	System.String	Unicode-Zeichenkette ▶

CIL-Name	CLS-Typ	Name in der Klassenbibliothek	Beschreibung
float32	Ja	System.Single	IEC 60559:1989 Fließkommazahl (32 Bit)
float64	Ja	System.Double	IEC 60559:1989 Fließkommazahl (64 Bit)
int8	Nein	System.SByte	Vorzeichenbehafteter 8-Bit-Integer-Wert
int16	Ja	System.Int16	Vorzeichenbehafteter 16-Bit-Integer-Wert
int32	Ja	System.Int32	Vorzeichenbehafteter 32-Bit-Integer-Wert
int64	Ja	System.Int64	Vorzeichenbehafteter 64-Bit-Integer-Wert
natural int	Ja	System.IntPtr	Zeiger
natural unsigned int	Nein	System.UIntPtr	Vorzeichenloser Zeiger
typedref	Nein	System.TypedReference	Zeiger mit Typ
unsigned int8	Ja	System.Byte	Vorzeichenloser 8-Bit-Integer-Wert
unsigned int16	Nein	System.UInt16	Vorzeichenloser 16-Bit-Integer-Wert
unsigned int32	Nein	System.UInt32	Vorzeichenloser 32-Bit-Integer-Wert
unsigned int64	Nein	System.UInt64	Vorzeichenloser 64-Bit-Integer-Wert

Tabelle 4.3 Eingebaute elementare Datentypen im .NET Framework

Objektorientierung

Ein hervorstechendes Merkmal des .NET Framework ist die durchgehende Objektorientierung. Jede Information, auch elementare Werte wie Zahlen, wird als Instanz von Klassen betrachtet. Aus Leistungsgründen gibt es intern dennoch eine unterschiedliche Behandlung von Referenztypen (die auf dem *Managed Heap* liegen) und Werttypen (die auf dem *Stack* liegen).

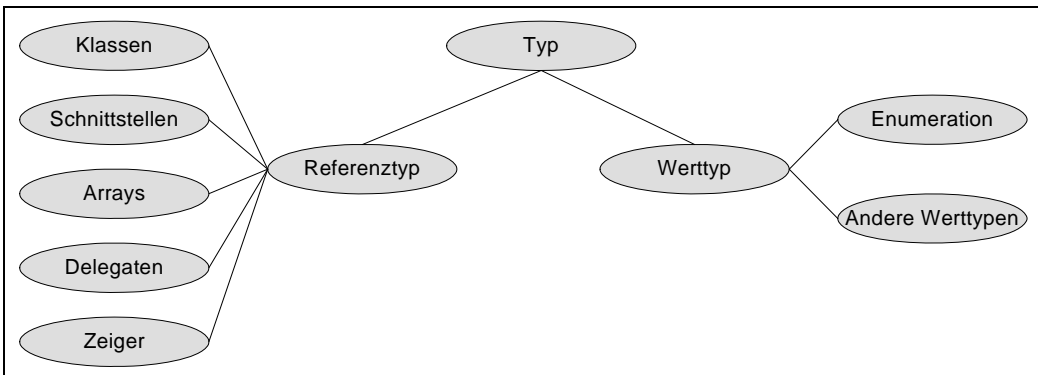


Abbildung 4.5 Typsystem im .NET Framework

Referenztypen

Referenztypen sind zu unterscheiden in Klassen, Schnittstellen, Arrays, Delegaten und Zeiger.

Klassen

Eine Klasse kann folgende Mitglieder (Members) besitzen:

- **Attribute (Felder und Eigenschaften)** Attribute sind die Datenmitglieder einer Klasse. In den Instanzen der Klasse repräsentiert der Inhalt der Attribute den Zustand des Objekts. Attribute besitzen einen bestimmten Typ und in den Instanzen einen Wert. Sie können konstant sein. Zu unterscheiden ist zwischen einfachen (direkten) Attributen ohne hinterlegten Programmcode (Felder, engl. *Fields*) und Attributen, die mit Getter- und Setter-Methoden realisiert sind (Eigenschaften, engl. *Properties*). Ein Indexer ist eine Property mit Parametern.

HINWEIS Leider hat Microsoft eine von der objektorientierten Lehre (vgl. für den deutschen Sprachraum [OES97, S. 157] und [ScWe04, S. 277] und für den englischen Sprachraum [OXF97, S. 243]) und von der Verwendung in vielen etablierten Programmiersprachen (z. B. Java, Delphi, C++) sowie Modellierungssprachen (vgl. UML) abweichende Verwendung des Begriffs *Attribut* in .NET. Microsoft benutzt den Begriff *Attribut* in .NET für Metadaten einer Klasse und nicht für die Daten einer Klasse. Dies führt in der Praxis zu Definitions- und Kommunikationsproblemen – zumal .NET den Anspruch einer universalen Architektur für alle Sprachen erhebt. Dieses Buch wird sich der ungünstigen Nomenklatur daher nicht anschließen und zwischen *Attributen* (den Datenmitgliedern) und *Annotationen* (Metadaten, siehe Abschnitt »Metadaten«) differenzieren.

- **Methoden** Methoden sind Operationen in Klassen, die innerhalb der Klasse oder von Nutzern aufgerufen werden können. Methoden können einen Rückgabewert liefern. Ein Konstruktor ist eine Methode, die beim Instanzieren aufgerufen wird. Echte Destruktoren, die beim Löschen eines Objekts aufgerufen werden, kennt das .NET Framework hingegen nicht. Der Aufruf des Destruktors ist im .NET Framework nicht deterministisch. Daher spricht man oft auch von Finalizern statt von Destruktoren.
- **Ereignisse** Klassen oder einzelne Objekte können Ereignisse auslösen, die von anderen abonniert werden können. Zu einem Ereignis kann es beliebig viele Abonnenten in beliebig vielen Objekten geben. In diesem Fall ruft das Objekt Unterrouinen in allen Abonnenten auf, wenn eine bestimmte Situation eintritt.

Vererbung

.NET unterstützt Einfachvererbung. Jede Klasse kann von höchstens von einer anderen Klasse erben. Mehrfachvererbung wird unterstützt. Wird keine Vererbung explizit definiert, erbt die Klasse implizit von der Klasse `System.Object`. Aus diesem Grund ist die Klasse `System.Object` die Wurzel jeder Vererbungshierarchie und alle Klassen in .NET besitzen die von `System.Object` definierten Mitglieder. Bei der Vererbung werden nur Attribute und Methoden vererbt. Konstruktoren und Ereignisse werden nicht vererbt.

Schnittstellen

Eine Schnittstelle ist eine Beschreibung von Attributen, Methoden und Ereignissen, die im Unterschied zu einer Klasse keinerlei Verhalten für die Methoden besitzt. Eine Schnittstelle ist nur die Hülle ohne einen Kern.

Eine Schnittstelle kann nicht instanziiert, sondern nur im Rahmen einer Klasse verwendet werden. Dort muss die Schnittstelle implementiert werden, genauer gesagt, dort müssen *alle* Attribute und Ereignisse deklariert und alle Methoden deklariert und implementiert werden.

Während das .NET Framework nur die einfache Implementierungsvererbung unterstützt, gibt es Mehrfachvererbung für Schnittstellen, d.h., eine Klasse kann optional eine oder mehrere Schnittstellen implementieren. Eine Schnittstelle kann auch von mehreren anderen Schnittstellen erben (interface IMultiFunc : IFax, IScanner, IDrucker). Eine .NET-Klasse ist im Gegensatz zu einer COM-Klasse nicht verpflichtet, eine bestimmte Schnittstelle explizit zu implementieren.

HINWEIS Im Gegensatz zu COM kann sich eine Schnittstelle weiterentwickeln, d.h., der Entwickler darf Mitglieder hinzufügen. Er bricht den Schnittstellenvertrag nicht, solange er nur hinzufügt. Der Schnittstellenvertrag würde nur gebrochen, wenn alte Mitglieder entfernt bzw. die Parameteranzahl oder Datentypen geändert würden.

Arrays

Arrays im .NET Framework besitzen folgende Eigenschaften:

- Sie sind abgeleitet von System.Array
- Die Arrays haben eine oder mehrere Dimensionen
- Dimension und Größe des Arrays müssen nicht durch die Deklaration vorgegeben werden, sondern werden bei der Instanziierung angegeben
- Nach der Instanziierung ist das Arrays in seiner Dimension und Größe nicht mehr veränderbar

Delegaten (Funktionszeiger)

Delegaten (engl. *Delegates*) sind typsichere Zeiger auf Funktionen. Durch Delegaten kann der aufzurufende Code variabel gehalten werden. Sie kommen insbesondere zum Einsatz für die Ereignisbehandlung und für asynchrone Methodenaufrufe. Ein Delegat kann auf mehrere Funktionen zeigen (*Multicast Delegate*). Beim Aufruf des Delegaten werden alle an den Delegaten gebundenen Funktionen aufgerufen. Delegaten sind das technische Instrument zur Realisierung des Ereignissystems in .NET.

Zeiger

Das .NET Framework unterstützt auch Zeiger und die zugehörige Zeigerarithmetik. Allerdings führen Zeiger ein Schattendasein, weil sie dem Verwaltungsprinzip der CLR widersprechen. Zeiger werden in Managed C++ und C# (in einem speziellen Modus) unterstützt, nicht aber in Visual Basic .NET.

Eingebettete Typen (Geschachtelte Typen)

Eine Typdefinition kann andere Typdefinitionen enthalten, so genannte eingebettete (geschachtelte) Typen (engl. *Nested Type*); z.B. kann eine Klassendefinition eine andere Klassendefinition enthalten oder eine Schnittstellendefinition. Diese eingebetteten Typen werden über den Namen des übergeordneten Typs adressiert (z.B. Klasse.Klasse).

Wertetypen (Strukturen)

Grundsätzlich sind alle Typen im .NET Framework Klassen, das .NET Framework ist also komplett objektorientiert, weil auch einfache Datentypen wie Zahlen als Objekte aufgefasst werden, auf denen man Methoden ausführen kann. So sind z. B. `5.ToString()` und `#8/1/1972#.ToLongDateString()`¹ gültige Ausdrücke. Klassen sind üblicherweise Referenztypen, d.h., im Stack wird ein Zeiger auf einen Speicherplatz im Heap vorgehalten.

Für einfache Datentypen ist diese Zwischenstufe jedoch sehr ineffizient. Microsoft hat daher im .NET Framework auch *Wertetypen* (alias *Strukturen*) vorgesehen, deren Inhalt direkt auf dem Stack abgelegt werden kann.

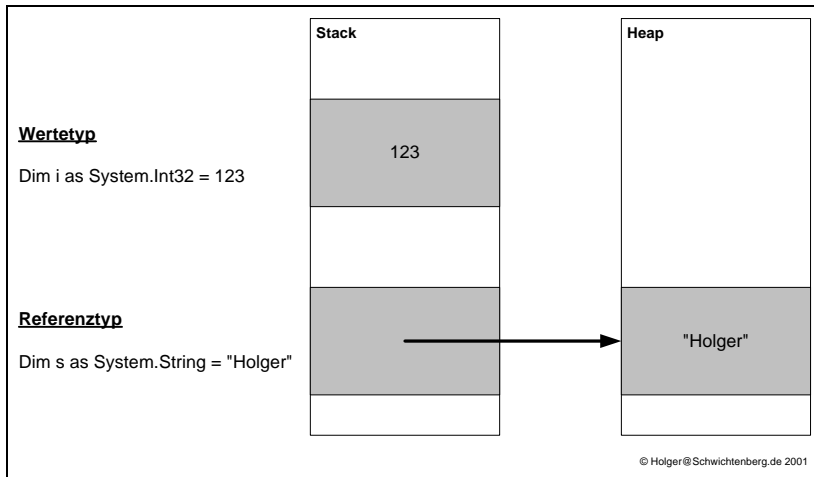


Abbildung 4.6 Werttyp versus Referenztyp im Hauptspeicher

Auch Werttypen sind als Klassen implementiert und können daher die gleichen Mitglieder wie Klassen besitzen. Ihre Besonderheit besteht jedoch darin, dass sie von `System.ValueType` erben.

Vergleich zwischen Werttyp und Referenztyp

Tabelle 4.4 zeigt die Unterschiede zwischen Werttyp und Referenztyp. Besonders zu erwähnen ist noch die Klasse `System.String`. Diese Klasse gehört zwar zu den Referenztypen, verhält sich aber beim Kopieren wie ein Werttyp.

	Referenztyp	Werttyp
Speicherort der Werte	Heap	Stack
Basisklasse	Direktes oder indirektes Erben von <code>System.Object</code>	Direktes oder indirektes Erben von <code>System.ValueType</code>
Setzen auf <i>Null</i>	Ja	Ja (ab .NET 2.0 mit der generischen Struktur <code>Nullable</code> – siehe Abschnitt »Wertelose Wertetypen (Nullable Value Types)«)
Vererbung an andere Klasse	Ja	Nein ▶

¹ Nur VB! – C# kennt keine Datumskonstanten im Code.

	Referenztyp	Werttyp
Abonnement von Ereignissen	Ja	Nein
Instanziierung	Pflicht	Optional, Instanziierung führt zu Initialisierung
Vergleich	Referenzvergleich	Wertvergleich
Kopie	Referenzkopie (flache Wertkopie optional mit <code>MemberwiseClone()</code> , tiefe Kopie muss selbst entwickelt werden)	Wertkopie

Tabelle 4.4 Werttyp versus Referenztyp

Boxing

Ein Werttyp kann explizit als ein Referenztyp behandelt werden. Dazu muss der Werttyp in ein Objekt verpackt werden. Dieser Vorgang wird als *Boxing* bezeichnet. Der gegensätzliche Vorgang heißt *Unboxing*.

Enumerationen

Ein Aufzählungstyp (*Enumeration*) ist ein spezieller Werttyp, der eine Liste von Konstanten darstellt. Jede einzelne symbolische Konstante der Liste repräsentiert einen Ganzzahlwert (Byte, Int16, Int32, Int64). Zeichenketten oder andere Datentypen sind nicht erlaubt.

Typnamen und Namensräume (Namespaces)

Typen werden in .NET nicht mehr wie in COM durch GUIDs, sondern durch Zeichenketten eindeutig benannt. Diese Zeichenketten sind hierarchische Namen, die aus einem *Namensraum* (engl. *Namespace*) und einem Typnamen bestehen. Ein Namensraum kann aus mehreren Hierarchieebenen bestehen. Zur Bildung eines voll qualifizierten .NET-Typnamens werden sowohl Namensraum und Typname als auch die Ebenen innerhalb eines Namensraums durch Punkte getrennt. Über alle Namensräume hinweg kann der Typname mehrfach vorkommen, vergleichbar mit gleichnamigen Dateien in verschiedenen Ordnern in einem Dateisystem.

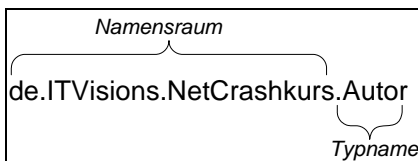


Abbildung 4.7 Beispiel für einen voll qualifizierten .NET-Typnamen

Softwarekomponenten versus Namensräume

Eine einzelne .NET-Softwarekomponente kann beliebig viele Namensräume umfassen und ein Namensraum kann sich über beliebig viele Softwarekomponenten erstrecken. Die Auswahl der Typen, die zu einem Namensraum gehören, sollte nach logischen oder funktionellen Prinzipien erfolgen. Im Gegensatz dazu sollte die Zusammenfassung von Typen zu einer Softwarekomponente gemäß den Bedürfnissen zur Verbreitung der Klassen (Deployment) erfolgen.

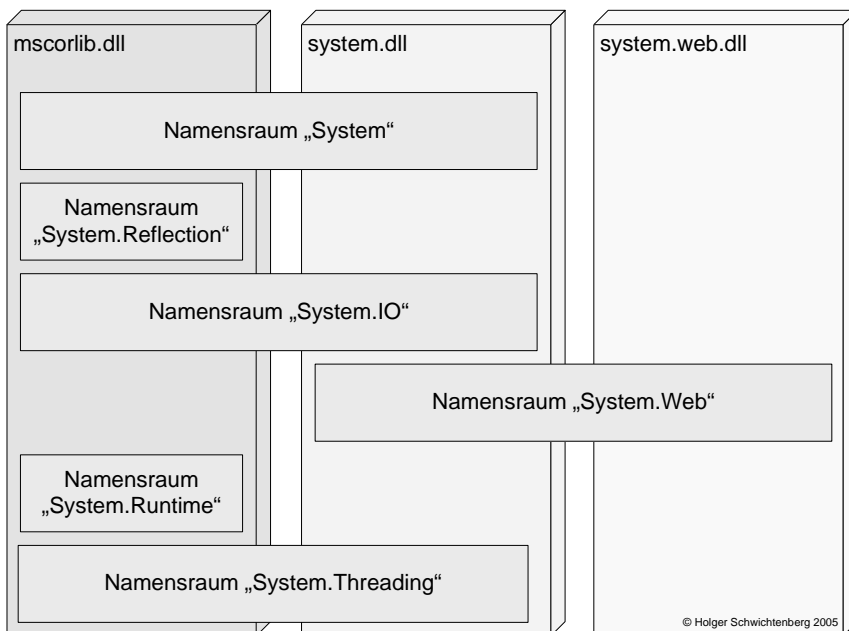


Abbildung 4.8 Namensräume versus Softwarekomponenten am Beispiel ausgewählter Teile der .NET-Klassenbibliothek

Im .NET Framework können beliebig viele Namensraumhierarchien parallel existieren. Es gibt keinen gemeinsamen Wurzelnamensraum und keine zentrale Registrierung der Namensräume. Die .NET Framework Class Library (FCL) besitzt zwei Wurzelnamensräume, System und Microsoft.

Da kein globales Verzeichnis aller Namensräume auf einem System existiert, gibt es nicht wie in COM eine einfache Möglichkeit, alle auf einem System vorhandenen .NET-Klassen aufzulisten. Möglich wäre aber die Suche nach .dll- bzw. .exe-Dateien im Dateisystem und eine Einzelprüfung dieser Dateien daraufhin, ob sie .NET-Typen enthalten.

Vergabe der Namensraumbezeichner

Da keine zentrale Stelle existiert, die die Namensraumbezeichner vergibt, besteht natürlich grundsätzlich die Gefahr, dass zwei Softwareentwickler gleiche Typnamen festlegen. Im CLI-Standard (Teil 5, D.1.5) ist daher vorgesehen, dass der Namensraum mit dem Firmennamen beginnt. Noch eindeutiger wird der Name jedoch, wenn man anstelle des Firmennamens den Internet-Domännennamen verwendet, z.B. de.itvisions.wwwings.autor statt itvisions.wwwings.autor.

Diese Konvention schützt natürlich nicht vor mutwilligen Doppelbenennungen. Für .NET-Anwendungen und -Softwarekomponenten ist deshalb vorgesehen, dass diese digital signiert werden können.

Vergabe der Typnamen

Auch für die Namensgebung von Typen in der .NET-Klassenbibliothek gibt es Regeln, die im CLI-Standard manifestiert sind. Die Namen für Klassen, Schnittstellen und Attribute sollen Substantive sein. Die Namen für Methoden und Ereignisse sollen Verben sein.

Für die Groß-/Kleinschreibung gilt grundsätzlich *PascalCasing*, d. h., ein Bezeichner beginnt grundsätzlich mit einem Großbuchstaben und jedes weitere Wort innerhalb des Bezeichners beginnt ebenfalls wieder mit einem Großbuchstaben. Ausnahmen gibt es lediglich für Abkürzungen, die nur aus zwei Buchstaben bestehen. Diese dürfen komplett in Großbuchstaben geschrieben werden (z. B. UI und IO). Alle anderen Abkürzungen werden entgegen ihrer normalen Schreibweise in Groß-/Kleinschreibung geschrieben (z. B. Xml, Xsd und W3c). Attribute, die geschützt (Schlüsselwort *Protected*) sind, und die Namen von Parametern sollen in *camelCasing* (Bezeichner beginnt mit einem Kleinbuchstaben, aber jedes weitere Wort innerhalb des Bezeichners beginnt mit einem Großbuchstaben) geschrieben werden.

Einige Programmiersprachen (wie beispielsweise C#) erlauben, dass sich zwei Bezeichner nur hinsichtlich der Groß- und Kleinschreibung unterscheiden können. Es wäre in C# also gültig zu definieren:

```
public class Autor
{
    public string Name;
    public string name;
}
```

Jedoch ist diese Vorgehensweise nicht CLS-konform, weil eine andere, nicht zwischen Groß- und Kleinschreibung unterscheidende (case-sensitive) Sprache diese beiden Attribute nicht unterscheiden könnte. Ein Client in Visual Basic würde nur das erste Mitglied *Name* sehen; das zweite *name* bliebe verdeckt. CLS-konform ist jedoch folgende Deklaration, weil in diesem Fall das zweite Attribut nicht nach außen angeboten wird:

```
public class Autor
{
    public string Name;
    private string name;
}
```

Neuheiten im Typkonzept seit .NET 2.0

Zentrale Neuerungen, die erstmals mit .NET 2.0 im Typkonzept eingeführt wurden, sind

- generische Klassen,
- wertelose Werttypen (*Nullable Value Types*) und
- partielle Typen

HINWEIS Tatsächlich sind die generischen Klassen das einzige neue Typkonstrukt, das zu einer Veränderung der CLR selbst geführt hat. Wertelose Werttypen werden nämlich durch den Einsatz eines generischen Typs implementiert und die partiellen Typen werden bereits auf Compiler-Ebene behandelt.

In .NET 3.0/3.5 gibt es keine Neuerungen im Typkonzept.

Generische Klassen (Generics)

Generische Klassen (engl. *Generics*, zum Teil mit *Generika* übersetzt) erlauben die Parametrisierung einer Klassendefinition, sodass ein oder mehrere Typen, die die Klasse verarbeiten soll, nicht zur Entwicklungszeit der Klasse, sondern erst bei der Nutzung der Klasse vorgegeben werden können. Der Nutzer einer Klasse kann also bei der Deklaration die Datentypen vorgeben, welche die Klasse verarbeiten soll. Bei den zusätzlichen Parametern spricht man von *Typparametern*, die von dem Nutzer der Klasse durch Typargumente gefüllt werden. Daraus entsteht ein so genannter *konstruierter Typ* (*Constructed Type*). C# verwendet zur Festlegung der *Typargumente* spitze Klammern und Visual Basic das Schlüsselwort *Of*. Durch so genannte *Einschränkungen für Typparameter* (*Generic Constraints*) können die Typargumente eingeschränkt werden.

HINWEIS Grundsätzlich sind generische Klassen dem Konzept von Templates in C++ sehr ähnlich. C++-Templates sind aber ein mächtigeres Konzept als generische Klassen. Ziel der Generics ist es, eine Klasse zu implementieren, aber bei der Nutzung zu konkretisieren.

Ein wichtiges Einsatzgebiet für Generics sind typisierte Objektmengen. Während die .NET-Klassenbibliothek bisher nur untypisierte Objektmengen angeboten hat, ermöglichen ab .NET 2.0 die in den neuen Namensräumen `System.Collections.Generic` und `System.Collections.ObjectModel` hinterlegten Klassen die einfache Definition von typisierten Objektmengen. Generics können auch auf andere .NET-Typen wie Strukturen, Schnittstellen und Delegaten sowie auf Methoden (Generische Methoden) angewendet werden.

Eine vollständige Unterstützung für Generics liefert Microsoft nur für Visual Basic 2008/2010, C# 2008/2010 und C++/CLI. J# 2008 (hier gibt es keine Version 2010) und JScript .NET 2008/2010 können generische Typen nutzen, aber nicht selbst erzeugen.

Details zur Nutzung und Erstellung von generischen Typen erfahren Sie in den Kapiteln zu Visual Basic und C#.

Wertelose Werttypen (Nullable Value Types)

Während Referenztypen bereits in .NET 1.x den Zustand `null` als Repräsentanz des Zustands nicht *vorhanden/nicht gesetzt* annehmen konnten, war dies für Werttypen nicht vorgesehen. Ab .NET 2.0 existiert ein Hilfskonstrukt, um auch Werttypen den Wert `null` zuweisen zu können.

In .NET (ab Version 2.0) ist ein auf `null` setzbarer Werttyp eine generische Struktur (`System.Nullable`), die aus dem eigentlichen Wert (`Value`) und einem Hilfs-Flag `HasValue` (Typ `boolean`) besteht, das anzeigt, ob der Wert des Typs `Null` ist.

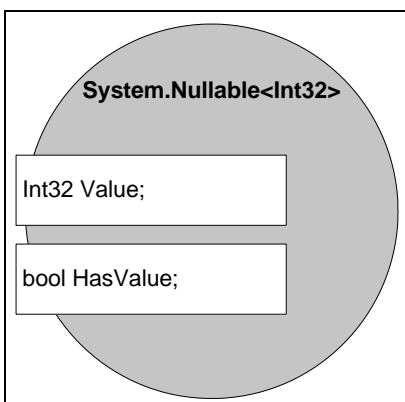


Abbildung 4.9 Realisierung von wertelosen Werttypen durch die generische Struktur `System.Nullable`

Integration in die Programmiersprachen

In C# (ab Version 2005) und Visual Basic (ab Version 2008) existiert ein besonderes Sprachkonstrukt (Anhängen eines Fragezeichens an den Typ bei der Deklaration), das die Schreibweise verkürzt. Weiterhin wird hier durch eine zusätzliche Sonderbehandlung des Compilers sichergestellt, dass der Entwickler auch direkt über den Variablennamen auf den Wert zugreifen kann – ohne den umständlichen Weg über das `Value`-Attribut. Außerdem gibt es einen neuen Operator ab C# 2005: das doppelte Fragezeichen. `??` liefert den Wert des vorangestellten Ausdrucks, wenn dieser nicht Null ist. Wenn der Wert Null ist, wird der Wert des nachfolgenden Ausdrucks übergeben.

Die Unterstützung für wertelose Wertetypen war in Visual Basic 2005 wesentlich schlechter. Seit Visual Basic 2008 ist die Unterstützung jedoch äquivalent zu C#.

Integration in die Klassenbibliothek

Wertelose Wertetypen könnten an einigen Stellen in der .NET-Klassenbibliothek hilfreiche Dienste leisten, insbesondere beim Zugriff auf Datenbanken, wo es häufig Null-Werte gibt. Leider sind die wertelosen Wertetypen weder in ADO.NET noch in anderen Teilen der Klassenbibliothek ordentlich integriert, sodass Null-Werte fast immer explizit geprüft und behandelt werden müssen.

Partielle Klassen

Die dritte wichtige Neuerung im Typkonzept in .NET 2.0 waren partielle Klassen, mit denen der Entwickler den Programmcode einer Klasse auf mehrere einzelne Klassendefinitionen aufteilen kann. Dabei können die partiellen Klassendefinitionen auch in verschiedenen Dateien existieren. Partielle Klassen erlauben, dass verschiedene Entwickler an einer Klasse arbeiten können bzw. dass ein Teil einer Klasse automatisch durch ein CASE-Tool generiert wird, während andere Teile per Hand codiert werden.

Microsoft setzt partielle Klassen in Webforms und Windows Forms ein. Während in ASP.NET 1.x Webforms durch Vererbung in eine ASPX-Klasse und eine Hintergrundcode-Klasse getrennt wurden, ist dies in ASP.NET 2.0/3.5 durch partielle Klassen besser gelöst. Auch der automatisch generierte Programmcode (*Windows Form Designer Generated Code*) einer Windows Form wird von Visual Studio 2005/2008 in eine separate Klasse (*Formularname.Designer.cs*) ausgelagert.

Visual Studio unterstützt für partielle Klassen IntelliSense, d. h., der Entwickler sieht alle Mitglieder unabhängig davon, in welcher Codedatei sie implementiert sind.

Dynamic Language Runtime (DLR) ab .NET 4.0

Die Dynamic Language Runtime (DLR) ist eine Erweiterung der Common Language Runtime (CLR), die Dienste für dynamisch typisierte Sprachen auf Basis von .NET anbietet. Beispiele für Sprache, die auf der DLR basieren, sind *IronPython* [IP01] und *IronRuby* [IR01]. Vor .NET 4.0 war die DLR ein Zusatz, seit .NET 4.0 ist sie fest in das .NET Framework integriert. Auch Visual Basic (spätes Binden mit `Typ object`) und C# (Schlüsselwort `dynamic`) verwenden nun diese Erweiterung. Auch die Klassenbibliothek bietet jetzt Klassen, die die DLR nutzen (Klassen `DynamicObject` und `ExpandoObject`).

.NET-Klassenbibliothek (FCL)

Ein weiterer Aspekt, der die Programmierung in verschiedenen Programmiersprachen bislang höchst unterschiedlich gemacht hat, waren die verschiedenen Funktions- bzw. Klassenbibliotheken. Die .NET-Klassenbibliothek – *.NET Framework Class Library (FCL)* – ist eine sehr umfangreiche Klassenbibliothek, die von allen .NET-Sprachen aus genutzt werden kann. Selbst wenn es in verschiedenen .NET-Sprachen noch alternative Möglichkeiten für die Ausführung unterschiedlicher Systemfunktionen (z.B. für den Dateisystemzugriff) gibt, sollten die Klassen der FCL genutzt werden. Dies vermindert den Lern- und Umstellungsaufwand beim Wechsel auf eine andere Sprache enorm. Die FCL ist implementiert als eine Reihe von DLLs in Managed Code.

HINWEIS Grundsätzlich kann eine .NET-Programmiersprache noch eine eigenständige Klassenbibliothek besitzen. J# verfügt über eine eigene Bibliothek mit Namen *vslib*, um die Quellcodekompatibilität zur Sprache Java herzustellen. Mit Visual Basic wird eine kleine Bibliothek mitgeliefert (*Microsoft.VisualBasic.dll*), die zwingend für jede mit Visual Basic .NET entwickelte Anwendung ist (seit Visual Basic 2008 kann sie durch eine eigene Implementierung ersetzt werden, siehe Kapitel 5).

Umfang der Klassenbibliothek

Die folgende Grafik stellt den Umfang der .NET-Klassenbibliothek in verschiedenen .NET-Versionen dar. Gezählt wurden nur öffentliche Klassen (keine privaten Klassen und Schnittstellen) und nur die Klassen unterhalb des Wurzelnamensraums »System«.

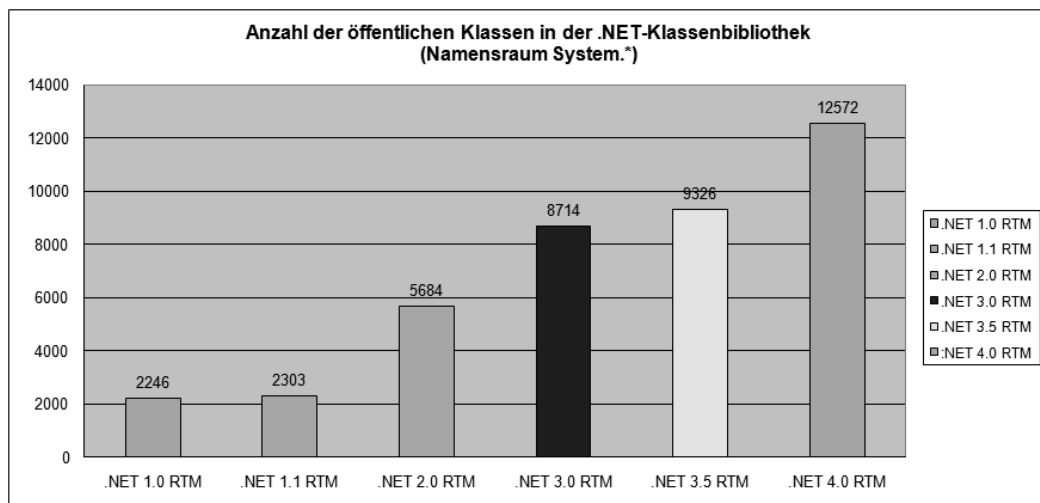


Abbildung 4.10 Umfang der .NET-Klassenbibliothek

Implementierung der FCL

Die .NET-Klassenbibliothek ist in weiten Teilen lediglich ein Wrapper für Funktionen aus dem Win32-API oder bestehenden COM-Komponenten. In Redmond existiert die Vision, dieses Verhältnis irgendwann umzukehren und .NET-Komponenten zur primären Schnittstelle für Windows zu machen. Bisher gab es davon aber nur *WinFX* im Rahmen der ursprünglichen Ankündigung von Windows Vista (siehe Kapitel 1).

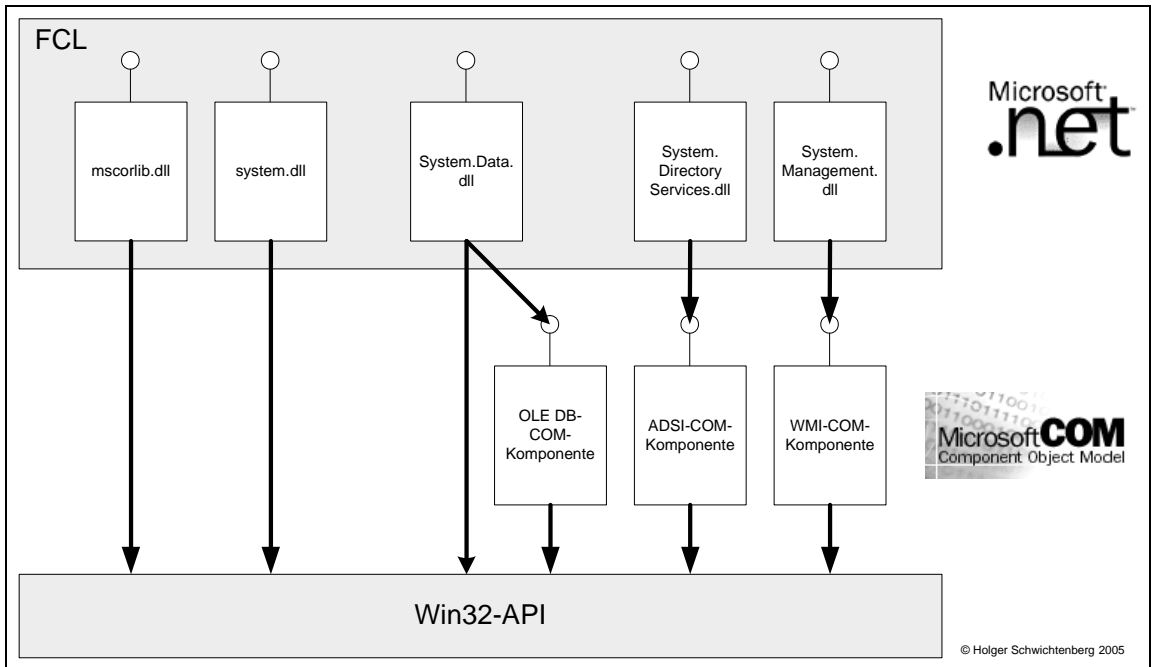


Abbildung 4.11 Implementierung der FCL

Namensräume

Um die Übersichtlichkeit zu gewährleisten, sind die FCL-Klassen in 312 Namensräume (Namespaces) eingeteilt. Ein Beispiel für einen voll qualifizierten FCL-Klassennamen ist `System.Collections.ArrayList`.

Die nachfolgende Tabelle gibt einen Überblick über die wichtigsten FCL-Namensräume.

Namensraum	Eingeführt in .NET... (Keine Angabe = .NET 1.0)	Beschreibung
Microsoft.CSharp		C#-Compiler
Microsoft.JScript		JScript .NET-Compiler
Microsoft.VisualBasic		Visual Basic-Compiler
Microsoft.Win32		Zugriff auf die Registrierungsdatenbank und Systemereignisse
System		Datentypen, Kommandozeilenfenster, Speicherverwaltung, Application Domain-Verwaltung
System.ServiceProcess		Erstellung von und Kontrolle über Windows-Systemdienste
System.Activities	4.0	Computergestützte Arbeitsabläufe, Neuimplementierung der Windows Workflow Foundation ab .NET 4.0 ▶

Namensraum	Eingeführt in .NET... (Keine Angabe = .NET 1.0)	Beschreibung
System.AddIn	3.5	Erstellen von Add-Ins (Managed Add-In-Framework)
System.CodeDom		Zugriff auf Quellcode
System.CodeDom.Compiler		Erstellen und Kompilieren von Code
System.Collections		Untypisierte Objektmengen
System.Collections.Concurrent	4.0	Thread-sichere Objektmengen
System.Collections.Generic	2.0	Generische Objektmengen
System.Collections.Specialized		Objektmengen mit strikter Typenbindung
System.ComponentModel		Unterstützung für Komponenten, die in einen Komponentencontainer aufgenommen werden können
System.ComponentModel.Composition	4.0	Neues Add-In Modell (Managed Extensibility Framework – MEF)
System.ComponentModel.Design		Unterstützung für Komponenten, die zur Entwurfszeit in einem WYSIWYG-Designer aufgenommen werden können
System.ComponentModel.Design.Serialization		Unterstützung von Komponentenserialisierung
System.Configuration		Zugriff auf Assembly-Konfigurationsdaten und globale Konfigurationsdaten
System.Configuration.Assemblies		Zugriff auf Assembly-Konfigurationsdaten
System.Configuration.Install		Erstellung von Installationsprogrammen
System.Data		Zugriff auf Datenquellen aller Art (ADO.NET)
System.Data.Common		Allgemeine Unterstützung von .NET-Datenprovidern
System.Data.Linq	3.5	LINQ to SQL (Datenbankzugriff)
System.Data.Linq.Mapping	3.5	Objektrelationales Mapping (ORM)
System.Data.OleDb		ADO.NET Data Provider für OLEDB
System.Data.OracleClient		ADO.NET Data Provider für Oracle
System.Data.SqlClient		ADO.NET Data Provider für Microsoft SQL Server
System.Data.SqlTypes		Datentypen des Microsoft SQL Server
System.Deployment	2.0	Click-Once-Deployment für Windows Forms- und Konsolenanwendungen
System.Device.Location	4.0	Klassen zur Unterstützung ortsabhängiger Dienste
System.Diagnostics		Debugging, Tracing, Systemprozesse, Ereignisprotokoll, Performance-Counter
System.Diagnostics.Contracts	4.0	Code Contracts
System.Diagnostics.Eventing	3.5	Ereignisprotokolle von Windows Vista und Windows Server 2008
System.Diagnostics.SymbolStore		Unterstützung von Debug-Symbolen
System.DirectoryServices		Wrapper für Active Directory Service Interface (ADSI) zum Zugriff auf Verzeichnisdienste aller Art
System.DirectoryServices.AccountManagement	3.5	Verwaltung von Benutzern, Gruppen und Computerkonten ►

Namensraum	Eingeführt in .NET... (Keine Angabe = .NET 1.0)	Beschreibung
System.DirectoryServices.ActiveDirectory	2.0	Active Directory-Programmierung
System.DirectoryServices.Protocols	2.0	Unterstützung der Directory Services Markup Language (DSML) für den XML-basierten Zugriff auf LDAP-Verzeichnisdienste
System.Drawing		Funktionen des Windows Graphics Device Interface (GDI+)
System.Drawing.Design		Grafikunterstützung zur Entwurfszeit
System.Drawing.Drawing2D		Unterstützung für zweidimensionale Grafiken
System.Drawing.Imaging		Unterstützung für Bitmap-Bearbeitung
System.Drawing.Printing		Druckdienste
System.Drawing.Text		Texte in Grafiken
System.Dynamic	4.0	Dynamische Typisierung (DLR)
System.EnterpriseServices		Zugriff auf COM+-Dienste
System.Globalization		Zugriff auf Ländereinstellungen (Kalender, Datumsformate, Zahlenformate)
System.IO		Dateisystem- und Dateizugriff
System.IO.Compression	2.0	Datenkomprimierung
System.IO.IsolatedStorage		Unterstützung isolierter Speicher
System.IO.MemoryMappedFiles	4.0	Inter-Prozess-Kommunikation mit Memory Mapped Files
System.IO.Packages	3.0	Zusammenfassen von Daten zu einem ZIP-Paket
System.IO.Pipes	3.5	Benannte und unbenannte Pipes
System.IO.Ports	2.0	Zugriff auf die seriellen Ports des Computers
System.Linq	3.5	Language Integrated Query (LINQ) – allgemeiner Teil
System.Linq.Expressions	3.5	Ausdrucksbäume
System.Management		Netz- und Systemmanagement mit der Windows Management Instrumentation (WMI)
System.Management.Instrumentation		Unterstützung bei der Entwicklung von WMI-Providern
System.Management.Instrumentation	3.5	Erstellen von WMI-Providern
System.Media	2.0	Abspielen von Audio-Daten
System.Messaging		Unterstützung des Microsoft Message Queue Service (MSMQ)
System.Net		Zugriff auf Netzwerkprotokolle (TCP, UDP, HTTP, DNS etc.)
System.Net.Mail	2.0	E-Mail-Versand
System.Net.NetworkInformation	2.0	Netzwerkverfügbarkeit, statische Daten aus dem TCP/IP-Protokoll-Stack
System.Net.PeerToPeer	3.5	Peer-To-Peer-Kommunikation
System.Net.Sockets		Zugriff auf die Socket-Schnittstelle
System.Numerics	4.0	Große und komplexe Zahlen ►

Namensraum	Eingeführt in .NET... (Keine Angabe = .NET 1.0)	Beschreibung
System.Reflection		Zugriff auf Metadaten von .NET-Komponenten
System.Reflection.Emit		Ausgabe von Metadaten, Erzeugen von PE-Dateien in MSIL-Code
System.Resources		Unterstützung kulturabhängiger Ressourcen
System.Runtime.Caching	4.0	Zwischenspeicherung von Werten (Caching)
System.Runtime.CompilerServices		Unterstützung für Compiler-Bau
System.Runtime.InteropServices		Interoperabilität zu COM-Komponenten
System.Runtime.Remoting		Nutzung für den objektorientierten Fernaufruf mit .NET Remoting
System.Runtime.Remoting.Activation		Aktivierung von entfernten Objekten
System.Runtime.Remoting.Channels		Allgemeine Channel-Unterstützung
System.Runtime.Remoting.Channels.Http		.NET Remoting über HTTP
System.Runtime.Remoting.Channels.Ipc	2.0	.NET Remoting über IPC
System.Runtime.Remoting.Channels.Tcp		.NET Remoting über TCP
System.Runtime.Remoting.Lifetime		Verwalten der Lebensdauer von entfernten Objekten
System.Runtime.Remoting.Proxies		Steuerung von .NET Remoting-Proxies
System.Runtime.Serialization		Serialisierung von .NET-Objekten
System.Runtime.Serialization.Formatters		Formatierung für die Serialisierung
System.Runtime.Serialization.Formatters.Binary		Binäre Serialisierung
System.Runtime.Serialization.Formatters.Soap		SOAP-Serialisierung
System.Security		Sicherheitseinstellungen für Komponenten und Objekte
System.Security.AccessControl	2.0	Beeinflussung der Zugriffsrechtliste von Dateien, Ordnern, Registrierungsdatenbank, Active Directory etc. in Windows
System.Security.Cryptography		Kryptografieunterstützung
System.Security.Cryptography.X509Certificates		Zugriff auf X509v3-Zertifikate
System.Security.Cryptography.Xml		Signieren von XML-Objekten
System.Security.Permissions		Berechtigungssätze für Code Access Security (CAS)
System.Security.Policy		Code-Gruppen für Code Access Security (CAS)
System.Security.Principal		Zugriff auf den Sicherheitskontext des Windows-Systems (angemeldeter Benutzer)
System.ServiceModel	3.0	Verteilte Systeme/Windows Communication Foundation (WCF)
System.ServiceModel.Routing	4.0	Routing in WCF
System.ServiceModel.Syndication	3.5	Unterstützung für RSS und ATOM
System.Text		Zeichenkettenfunktionen, Textcodierung ►

Namensraum	Eingeführt in .NET... (Keine Angabe = .NET 1.0)	Beschreibung
System.Text.RegularExpressions		Reguläre Ausdrücke
System.Threading		Unterstützung von Multi-Threading-Programmierung
System.Threading.Tasks	4.0	Multi-Threading-Programmierung mit der Task Parallel Library (TPL)
System.Timers		Zeitgesteuerte Ereignisse
System.Transactions	2.0	Transaktionssteuerung mit und ohne MSDTC
System.Web		Kommunikation zwischen Browser und Webserver (Objekte für ASP.NET)
System.Web.ApplicationServices	3.5	Zugriff auf ASP.NET-Anwendungsdienste von Desktop-Anwendungen über WCF
System.Web.Caching		Zwischenspeicher (Cache)
System.Web.Configuration		Konfiguration von ASP.NET
System.Web.Hosting		Unterstützung von ASP.NET außerhalb des IIS
System.Web.Management	2.0	Laufzeitüberwachungssystem in ASP.NET
System.Web.Profile		Profildatensystem in ASP.NET
System.Web.Security		Sicherheit für ASP.NET-Anwendungen
System.Web.Services		Unterstützung für XML-Webservices
System.Web.Services.Configuration		Konfiguration von XML-Webservices
System.Web.Services.Description		WSDL-Beschreibungen von XML-Webservices
System.Web.Services.Protocols		Definition von Protokollen zum Datenaustausch zwischen XML-Webservices
System.Web.SessionState		Verwalten des Sitzungszustands
System.Web.UI		Basisklassen und Hilfsklassen für serverseitige Steuerelemente
System.Web.UI.DataVisualization.Charting	4.0	WS-Discovery (Auffinden von Diensten) in WCF
System.Web.UI.DataVisualization.Charting	4.0	Chart-Steuerelement für ASP.NET
System.Web.UI.Design		Unterstützung von Steuerelementen zur Entwurfszeit
System.Web.UI.HtmlControls		Implementierung der HTML-Serversteuerelemente
System.Web.UI.MobileControls		Steuerelemente für mobile ASP.NET-Webanwendungen
System.Web.UI.WebControls		Implementierung der Web-Serversteuerelemente
System.Web.UI.WebControls.WebParts		ASP.NET-Webparts zum Aufbau von Portalen
System.Windows.* (zahlreiche Unternamensräume, außer "Forms")	3.0	Steuerelemente und andere visuelle Elemente für die Gestaltung von Windows-Desktop-Anwendungen mit WPF
System.Windows.Forms		Steuerelemente für die Gestaltung von Windows-Desktop-Anwendungen mit Windows Forms
System.Workflow	3.0	Computergestützte Arbeitsabläufe ►

Namensraum	Eingeführt in .NET... (Keine Angabe = .NET 1.0)	Beschreibung
System.Xaml	4.0	Allgemeine Unterstützungsklassen für die Arbeit mit XAML (ein XML-Dialekt, der u.a. in WPF/Silverlight und Workflow verwendet wird)
System.Xml		Zugriff auf das Document Object Model (DOM) der Extensible Markup Language
System.Xml.Linq	3.5	LINQ to XML (Zugriff auf XML-Dokumente)
System.Xml.Schema		Unterstützung von XML-Schemata (XSD)
System.Xml.Serialization		Serialisierung von Objekten in XML-Daten
System.Xml.XPath		Einsatz der XPath-Sprache
System.Xml.Xsl		Ausführung von Extensible Stylesheet Language-Transformationen (XSLT)

Tabelle 4.5 Die wichtigsten Namensräume der FCL

FCL versus BCL

Als *.NET Base Class Library (BCL)* werden die fundamentalen Teile der FCL bezeichnet. Folgende Namensräume gehören zur BCL:

- System
- System.CodeDom
- System.Collections
- System.Diagnostics
- System.Globalization
- System.IO
- System.Resources
- System.Text
- System.Text.RegularExpressions

Softwarekomponentenkonzept

Das .NET Framework ist nicht nur objekt-, sondern auch komponentenorientiert.

Der Softwarekomponentenbegriff in .NET Framework

Im Component Object Model (COM) ging es schon drunter und drüber hinsichtlich der Differenzierung von Klassen und Komponenten. Leider herrscht auch im .NET Framework immer noch Verwirrung, denn es gibt zwei unterschiedliche Definitionen von Softwarekomponenten:

- In der .NET Framework-Dokumentation findet man folgende Definition: »While the term component has many meanings, in the .NET Framework a component is a class that implements the System.ComponentModel.IComponent interface or one that derives directly or indirectly from a class that implements this interface.« [MSDN01]. Nimmt man diese Aussage wörtlich, dann sind Komponenten keine ganzen Assemblys, sondern nur einzelne Klassen, die die Schnittstelle IComponent besitzen.
- Diese Definition steht jedoch im Widerspruch zu der Gleichsetzung von kompilierten Dateien (DLL, EXE) und Softwarekomponenten, die sich in vielen anderen MSDN-Dokumenten (z. B. [MSDN02] und [MSDN03]) findet. DLLs und EXEs heißen im .NET Framework Assemblys. Eine EXE-Assembly (alias Managed EXE) ist eine startbare Anwendung. Eine DLL-Assembly (alias Managed DLL) kann nicht einzeln gestartet werden. Ihr Zweck ist die Verwendung im Rahmen einer anderen Anwendung. Sowohl EXE- als auch DLL-Assemblys sind wieder verwendbare Softwarekomponenten. Eine echte Unterscheidung in Anwendungen und Komponenten gibt es nicht mehr.

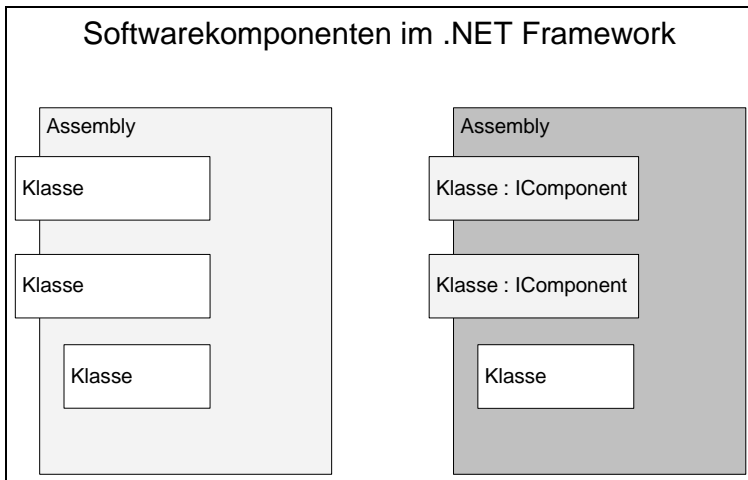


Abbildung 4.12 Ambivalenter Softwarekomponentenbegriff im .NET Framework

Dieser Widerspruch ist nicht schön; in der Praxis muss man aber damit leben, dass in Redmond keine Einigkeit über den Komponentenbegriff existiert.

Die Definition, dass eine Komponente auch eine einzelne Klasse sein kann, ist dabei durchaus sinnvoll. Es hat sich eingebürgert, von einem Steuerelement als »Komponente« zu sprechen. Mehrere Steuerelemente sind aber üblicherweise zu einer Assembly zusammengefasst, wobei die Assembly auch andere Klassen bereitstellen kann, die keine eigenständigen Komponenten sind. Hingegen ist es auch sinnvoll, ganze Assemblys als Komponenten zu verstehen, weil diese die Einheiten für Wiederverwendung, Verteilung und Versionierung sind.

Unter dem Strich bleibt also die unbefriedigende Quintessenz, dass im .NET Framework Komponenten sowohl eine ganze Assembly als auch einzelne Klassen einer Assembly sein können, sofern diese Teile IComponent implementieren. Eine grundsätzliche Beschränkung des Komponentenbegriffs auf IComponent-implementierende Klassen wie in [MSDN01] ist aber nicht nachvollziehbar.

Aufbau von Assemblys

Eine Assembly ist ein Verbund aus einer oder mehreren MSIL-Dateien (genannt *Managed Module*), wobei mindestens eine der Dateien eine DLL oder EXE ist. Optional können auch Nicht-MSIL-Dateien, so genannte Ressourcendateien (z.B. Datenbank-, Grafik- oder Sound-Dateien), Teil der Assembly sein. Welche Dateien zum Verbund gehören, wird durch ein Manifest bestimmt. Vor der Ausführung von Code aus einer Assembly wird durch die CLR geprüft, ob alle benötigten Dateien in der gewünschten Version vorhanden sind. Das Manifest ist Teil der DLL oder EXE.

Meistens bestehen Assemblys nur aus einem Managed Module und heißen daher *Ein-Datei-Assembly* (*Single-File-Assembly*, *SFA*). Möglich ist aber auch eine *Mehr-Dateien-Assembly* (*Multi-File-Assembly*, *MFA*), bei der die Assembly aus mehreren Managed Modules besteht. In diesem Fall liegt das Manifest nur in einem der Module. Visual Studio unterstützt auch in der aktuellen Version nur die Erstellung von Ein-Datei-Assemblys; Mehr-Dateien-Assemblys können nur mit den Kommandozeilen-Compilern und mit dem *Assemblylinker* (*al.exe*) aus dem .NET Framework SDK erstellt werden.

Es ist auch möglich, eine Assembly ohne Code und nur mit Ressourcen zu generieren. Solche zu Lokalisierungszwecken eingesetzten Assemblys heißen *Satelliten-Assemblys*.

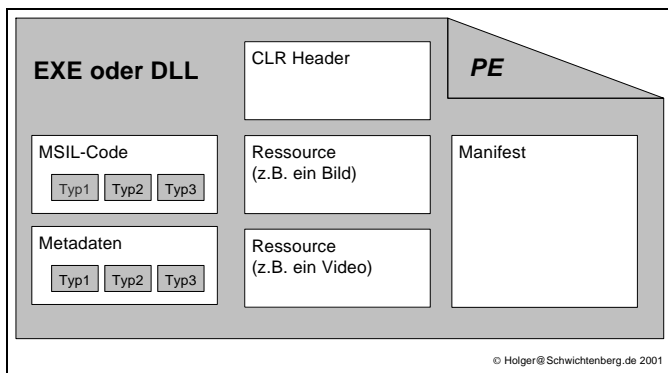


Abbildung 4.13 Aufbau einer Ein-Datei-Assembly

Die folgende Grafik zeigt den Aufbau einer Assembly mit mehreren Dateien. Die MSIL-Dateien verwenden als Dateiformat das *Portable Executable-Format* (*PE*), das auch von anderen Betriebssystemen gelesen werden kann. Der CLR-Header enthält etwas Unmanaged Code, um den Windows Loader dazu zu bringen, den entsprechenden Runtime Host zu starten. Hierin liegt ein Problem bei der Portierung von .NET-Assemblys: Andere Betriebssysteme kennen von Hause aus weder den Windows Loader noch CIL-Code. In Mono müssen daher alle .NET-Anwendungen mithilfe des Kommandos *mono* und der Angabe des zu startenden Programms zum Leben erweckt werden.

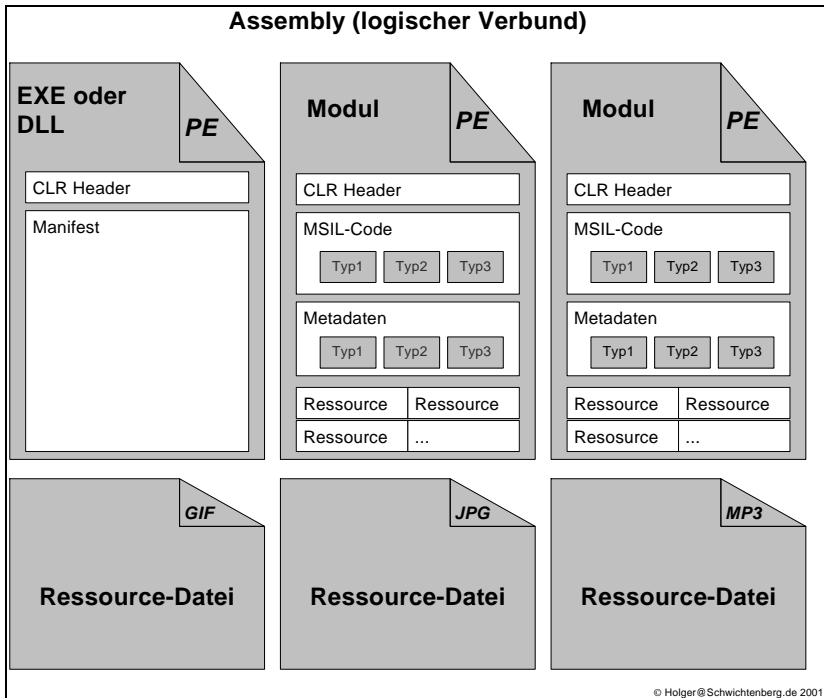


Abbildung 4.14 Aufbau einer Mehr-Dateien-Assembly

Der Name einer Assembly ist der Name der Hauptdatei ohne die Dateierweiterung. Das Manifest speichert folgende Daten:

- den Namen der Assembly
- die Version der Assembly
- ein Länderkürzel (optional), beispielsweise *de-de* oder *en-us*
- eine Liste der zur Assembly gehörenden .NET-Module
- eine Liste der zur Assembly gehörenden Ressourcendateien
- die Abhängigkeiten von anderen Assemblys (Referenzen)
- die Rechte, die zur Ausführung der Assembly notwendig sind
- den öffentlichen Schlüssel des Herstellers (optional)

Signierte Assemblys

Eine große Herausforderung im Bereich komponentenbasierter Softwareentwicklung sind Sicherheitsfragen. Sowohl der Entwickler einer Komponente als auch der spätere Endnutzer einer komponentenbasierten Anwendung will sicher sein, wer der Hersteller einer Komponente ist (Authentizitätsanforderung) und dass die Komponente auf dem Weg vom Hersteller nicht verändert wurde (Integritätsanforderung). Ein Angreifer könnte sonst eine DLL auch einfach durch eine gleichnamige DLL mit gleicher Versionsnummer und mit den gleichen implementierten Typen austauschen.

Das .NET Framework bietet gegen diesen Missbrauch digitale Signaturen, die einer jeden Assembly hinzugefügt werden können. In der .NET-Fachsprache erhält eine Assembly dadurch einen *starken Namen* (*Strong Name*).

Bei der digitalen Signierung einer Assembly wird zunächst ein Hash-Wert (ca. 100-200 Bytes) über die komplette PE-Datei gebildet und dieser Hash-Wert wird mit dem privaten Schlüssel des Herstellers versehen. Die daraus entstehende RSA-Signatur wird in dem CLR-Header abgelegt. Ebenfalls nachträglich in die Assembly abgelegt wird der öffentliche Schlüssel des Herstellers.

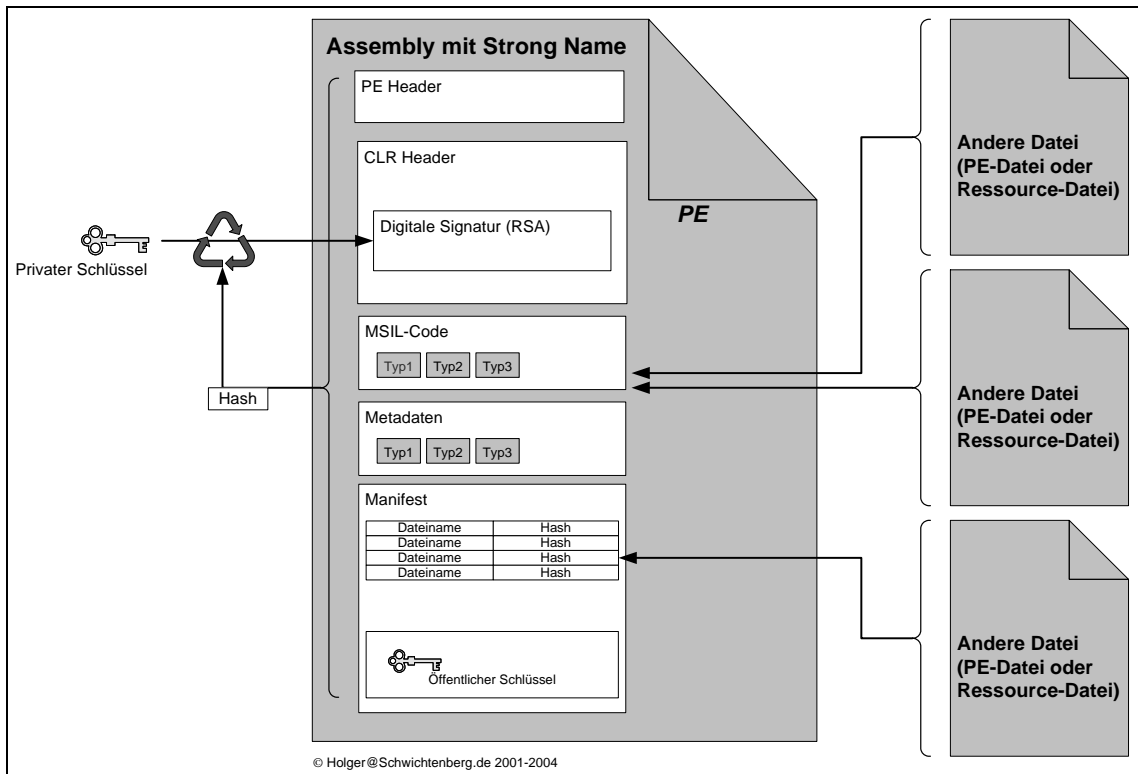


Abbildung 4.15 Assembly mit Strong Name

Der Empfänger der Assembly kann mithilfe des öffentlichen Schlüssels den Hash-Wert zurückrechnen und dann selbst den Hash-Wert über die PE-Datei bilden (wobei natürlich die Bereiche, in denen die Signatur und der öffentliche Schlüssel liegen, auszusparen sind). Nur wenn die beiden Hash-Werte übereinstimmen, ist die Integrität der Assembly gewahrt. Über ein Zertifikat des Herstellers kann der Empfänger zudem prüfen, ob der öffentliche Schlüssel tatsächlich dem Hersteller gehört. Der Softwarehersteller kann mithilfe des Werkzeugs *signcode.exe* ein Zertifikat (Software Publisher Certificate, SRC) in der Assembly ablegen.

Befreundete Assemblys (Friend Assemblies)

Neu ab .NET 2.0 sind so genannte *Friend Assemblies*. Beim Einsatz signierter Assemblys können auch die nicht-öffentlichen Typen einer Assembly für ausgewählte andere Assemblys sichtbar gemacht werden. Damit wird ermöglicht, eine Softwarekomponente bereitzustellen, die nur von bestimmten anderen Softwarekomponenten verwendet werden darf. Eine befreundete Assembly wird deklariert mit der Annotation [System.Runtime.CompilerServices.InternalsVisibleTo]. Friend Assemblys ließen sich nicht in Visual Basic 2005 nutzen. Seit Visual Basic 2008 hat die Sprache hier aber mit C# gleichgezogen.

Speicherorte für Assemblys

Assemblys können an jedem beliebigen Ort auf Speichermedien abgelegt werden. Die beiden üblichen Speicherorte sind jedoch das jeweilige Anwendungsverzeichnis und der so genannte Global Assembly Cache (GAC).

Der Standardspeicherort ist das Anwendungsverzeichnis, d. h., eine Assembly wird direkt zu der Anwendung abgelegt, die sie verwendet. Die DLL-Hölle (die gegenseitige Störung von Anwendungen durch die Verwendung einer DLL in unterschiedlichen, inkompatiblen Versionen) wird dadurch vermieden. Verloren geht etwas Speicherplatz, wenn mehrere Anwendungen die gleiche Assembly verwenden und diese dann mehrfach auf dem System vorhanden ist.

Nur für wenige Ausnahmen, in denen eine Mehrfachnutzung einer .NET-Softwarekomponente (Assembly in DLL-Form) sinnvoll ist (z. B. bei der .NET-Klassenbibliothek), gibt es weiterhin einen zentralen Speicherort, den GAC, der unter %Windows%/Assembly liegt. Eine Assembly im GAC wird *Shared Assembly* oder *Global Assembly* genannt. Shared Assemblys benötigen einen Strong Name.

Der GAC ist kein einfaches, flaches Verzeichnis, sondern eine komplexe Verzeichnishierarchie, die es ermöglicht, gleichnamige Assemblys in beliebig vielen verschiedenen Versionen zu speichern. Microsoft empfiehlt, keine Anwendungen zu entwickeln, die sich auf die interne Struktur des GAC verlassen, weil diese in zukünftigen Versionen geändert werden könnte.

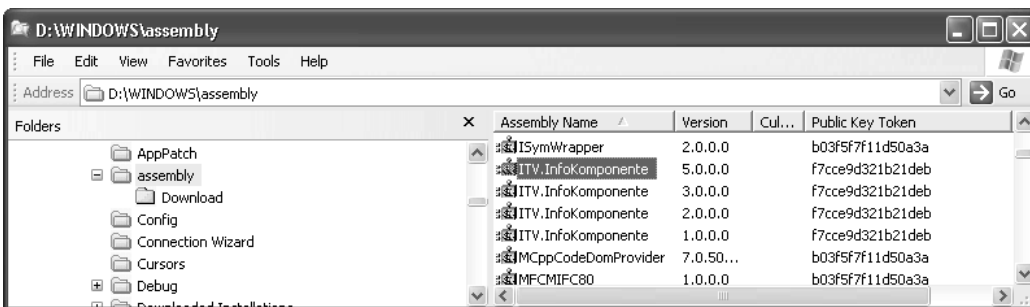


Abbildung 4.16 Ansicht des GAC mit der *ITV.InfoKomponente.dll* in vier verschiedenen Versionen

TIPP

Es ist empfehlenswert, auf die Ablage von .NET-Softwarekomponenten im GAC zu verzichten. Sinnvoll ist der GAC nur, wenn ein Baustein von sehr, sehr vielen Produkten auf einem System verwendet wird (z. B. eine Steuerelementbibliothek auf dem Webserver eines Webhosts).

Metadaten

Jede Assembly (folglich also jede .NET-Komponente) ist komplett selbstbeschreibend, d. h., es sind ausführliche Informationen über die in der Komponente enthaltenen Klassen und deren Mitglieder in der Assembly enthalten. Metadaten sind ein Pflichtbestandteil einer jeden Komponente. Dies ist ein großer Fortschritt gegenüber COM, wo die Selbstbeschreibung in Form von Typbibliotheken eine oft vernachlässigte Option war.

Reflection

Das Auslesen der Metadaten einer .NET-Komponente nennt man *Reflection*. Reflection ist ein integraler Bestandteil des .NET Framework, auf dem vier wichtige Mechanismen beruhen:

- Beim Bindungsmechanismus ermittelt die CLR mittels Reflection den aufzurufenden Programmcode.
- Die in .NET eingebauten Mechanismen zur Objektserialisierung benötigen die Metadaten, die sie via Reflection ermitteln. Objektserialisierung ist wiederum die Basis für das Remoting in .NET.
- Der Garbage Collector verwendet Reflection, um festzustellen, welche Objekte noch in Benutzung sind.
- Mittels des Reflection-Mechanismus kann man dynamisch Code zur Laufzeit erzeugen.

Annotations (.NET-Attribute)

Der Entwickler selbst kann Komponenten, Klassen und Klassenmitglieder mit zusätzlichen Informationen (Metadaten) versehen, die entweder während der Kompilierung oder zur Laufzeit der Anwendung ausgewertet werden können. Typische Beispiele für derartige Zusatzinformationen sind:

- Die Komponente hat die Version x (*AssemblyVersionAttribute*)
- Instanzen einer Klasse sind serialisierbar (*SerializableAttribute*)
- Instanzen der Klasse sollen Teil einer Transaktion sein (*TransactionAttribute*)
- Ein Mitglied einer Klasse ist aus Kompatibilitätsgründen zwar noch vorhanden, sollte aber nicht mehr verwendet werden, weil ein anderes, besseres Mitglied zur Verfügung steht (*ObsoleteAttribute*)

Leider verwendet Microsoft für diese Metadaten eine stark von der objektorientierten Lehre abweichende Begriffswelt: Die Firma nennt eine derartige Auszeichnung *Attribut* (engl. *Attribute*), was einen Namenskonflikt zu dem Begriff *Attribut*, also dem Datenmitglied einer Klasse darstellt (vgl. für den deutschen Sprachraum [OES97, S. 157] und [ScWe04, S. 277] und für den englischen Sprachraum [OXF97, S. 243]). Die Datenmitglieder einer Klasse heißen bei Microsoft *Felder* (engl. *Fields*) und *Eigenschaften* (engl. *Properties*). Dabei denkt man doch bei Feldern eher an Arrays. Ein klarer Fall von MINFU,² der sich in der deutschen Übersetzung besonders schlimm auswirkt.

HINWEIS In diesem Buch wird deshalb zur deutlicheren Trennung der Begriff *Annotations* (wie in Java ab Version 5.0) verwendet für die Metadaten einer Komponente, einer Klasse oder eines Klassenmitglieds.

Eine andere mögliche Nomenklatur wäre *Meta-Attribut* oder *Aspekt* (weil diese Metadaten eine aspektorientierte Programmierung ermöglichen).

² Auf Basis der Erkenntnis, dass Microsoft regelmäßig Probleme mit der Bezeichnung der eigenen Produkte und Konzepte hat, schuf der amerikanische Autor David S. Platt ein neues Wort: MINFU. Dies ist eine Abkürzung für Microsoft Nomenclature Foul-Up.

Annotationen werden in Form von Klassen implementiert, die von der Basisklasse `System.Attribute` abgeleitet sind. Sie haben Namen, die auf *Attribute* enden, wobei bei ihrer Verwendung das Wort *Attribute* weggelassen werden kann (z.B. `System.ObsoleteAttribute` → `[Obsolete]`). Jeder Entwickler kann eigene Annotationen definieren. Annotationen können ein Verhalten besitzen; sie werden aber erst verarbeitet, wenn ein Typ explizit von einem Host (z.B. einer Entwicklungsumgebung) oder einem anderen Typ via Reflection nach Annotationen gefragt wird.

Komponentenkonfiguration

Anders als in COM besteht in .NET keine zwingende Notwendigkeit mehr, externe Konfigurationsinformationen zu speichern, da eine Komponente bereits im Manifest umfangreiche Metadaten enthält. Wenn in die Assembly hineinkompilierte Informationen jedoch bei der Installation oder im laufenden Betrieb geändert werden sollen, ist dies durch externe Konfigurationsdateien möglich.

Das .NET Framework verwendet als Dateiformat für Konfigurationsdateien XML-Dateien mit dem Wurzelement `<configuration>`. Eine Konfigurationsdatei ist eine XML-Datei, die einer .NET-Anwendung zugeordnet werden kann. Name und Ablageort der Konfigurationsdatei sind vom Anwendungstypus abhängig:

- Für das .NET Framework insgesamt existiert eine zentrale Konfigurationsdatei im Unterverzeichnis */Config* im .NET Framework-Installationsverzeichnis. Die wichtigste Datei ist hier *machine.config*.
- Für Windows- und Konsolenanwendungen trägt diese Anwendungskonfigurationsdatei den Namen der *.exe*-Datei mit angehängtem *.config*. Für eine Anwendung *WindowsUI.exe* wäre der richtige Name also *WindowsUI.exe.config* (nicht: *WindowsUI.config*). Die Konfigurationsdatei muss im selben Verzeichnis wie die *.exe*-Datei liegen.
- Für ASP.NET-Anwendungen muss es im Wurzelverzeichnis der Anwendung eine Datei mit Namen *web.config* geben

ACHTUNG

Leider können Konfigurationsdateien nur für Anwendungen, nicht aber für einzelne Komponenten existieren.

Eine Konfigurationsdatei darf nur einen bestimmten Satz von Elementen enthalten. Die Mehrzahl der vordefinierten Elemente beeinflusst das Verhalten einzelner Anwendungen oder der .NET-Laufzeitumgebung insgesamt. Es existiert auch ein Element für die Ablage von anwendungsspezifischen Dateien. Außerdem lässt das .NET Framework die Definition eigener Konfigurationselemente zu.

Im Bereich der Konfigurationsdateien gibt es seit .NET 2.0 drei wesentliche Neuerungen:

- Viele Elemente in einer Konfigurationsdatei können durch XML Encryption [W3C01] verschlüsselt werden (Protected Configuration)
- Es existiert ein spezielles Konfigurationselement zur Ablage von Verbindungszeichenfolgen für ADO.NET
- Über die neue Klasse `ConfigurationManager` können die XML-Konfigurationsdateien nicht nur gelesen, sondern auch verändert werden



```
<?xml version="1.0" encoding="utf-8" ?>
- <configuration>
+ <configSections>
- <connectionStrings>
  <add name="de.ITVisions.ECM.GUI.Properties.Settings.eCliStuDatabase"
    connectionString="Jq1Un7JeIntwjskEY8DV1ddrJxpWBvfCNGcHTjPborX7Jx79lmc"
    providerName="System.Data.SqlClient" />
</connectionStrings>
- <userSettings>
- <applicationSettings>
- <de.ITVisions.ECM.GUI.Properties.Settings>
- <setting name="TempFolder" serializeAs="String">
  <value>c:\temp</value>
</setting>
- <setting name="Debug" serializeAs="String">
  <value>false</value>
</setting>
- <setting name="ErrorLog" serializeAs="String">
  <value>True</value>
</setting>
- <setting name="ErrorLogWebservice" serializeAs="String">
  <value>True</value>
</setting>
</de.ITVisions.ECM.GUI.Properties.Settings>
</applicationSettings>
- <runtime>
- <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
  <probing privatePath="BIN" />
</assemblyBinding>
</runtime>
</configuration>
```

Abbildung 4.17 Beispiel für eine Konfigurationsdatei für eine .NET-Anwendung mit verschlüsselter Sektion

Bearbeitung der Konfigurationsdateien

Anwendungskonfigurationsdateien können direkt durch jeden Texteditor oder einen XML-Editor (z.B. in Visual Studio) bearbeitet werden. Microsoft hat entsprechende XML-Schemata definiert unter dem fiktiven URL <http://schemas.microsoft.com/.NetConfiguration/v2.0>. Eine komfortablere Bearbeitung der Konfigurationsdateien ist durch mit .NET mitgelieferte Werkzeuge möglich.

- Konfigurationsdateien für Konsolen- und Windows-Anwendungen können mit dem MMC-Snap-in *.NET Framework-Konfiguration* bearbeitet werden. In der MMC muss eine zu konfigurierende Anwendung zunächst zum Ast *Anwendungen* hinzugefügt werden (*Hinzufügen* im Kontextmenü). In den Eigenschaften der Anwendung können dann Suchpfade für die verwendeten Komponenten angegeben werden. Im Ast *Konfigurierte Assemblies* können Versionsumlenkungen oder dedizierte Standorte für einzelne Assemblys festgelegt werden.
- Konfigurationsdateien für Webanwendungen können mit einer webbasierten Oberfläche oder über das MMC-Snap-in *Internet-Informationdienste* verändert werden. Beides wird im Kapitel »ASP.NET Webforms« näher beschrieben. Dieses Zusatzkapitel können Sie als PDF auf dem Leser-Portal herunterladen.

Assembly-Referenzen

Eine Assembly-Referenz stellt eine Nutzungsbeziehung zwischen zwei .NET-Assemblys dar. Eine Assembly, die eine andere Assembly referenziert, kann alle Typen nutzen, die die referenzierte Assembly exportiert. Exportiert werden alle Typen, die als öffentlich (*public*) deklariert sind. Typen können auch versteckt (Sichtbarkeitstyp *Assembly*) deklariert werden.

Eine Referenz von A1 zu A2 ist notwendig, wenn ein Typ x aus Assembly A1 einen Typ y aus A2 nutzen will. Eine Referenz von A1 zu A2 und A3 ist notwendig, wenn ein Typ x aus Assembly A1 einen Typ y aus A2 nutzen will und Typ y von einem Typ z aus A3 erbt.

Zwischen Managed DLLs und Managed EXEs können beliebige Referenzen erzeugt werden, d. h., neben der selbstverständlichen Referenzierung einer DLL durch eine EXE kann eine EXE auch andere EXEs referenzieren oder eine DLL eine EXE.

Visual Studio .NET 2002 und 2003 haben die Erstellung einer Referenz auf eine .exe-Datei verweigert. Die Erstellung derartiger Referenzen war nur über die Kommandozeilen-Compiler möglich. Diese Restriktion ist ab Visual Studio 2005 aufgehoben worden (sie gilt jetzt nur noch für Webanwendungen).

Assembly Resolver

Im Rahmen einer *Assembly-Referenz* wird in der referenzierenden Assembly lediglich der Name der referenzierten Assembly (ohne die Dateierweiterung) abgelegt. Der Pfad zu der referenzierten Assembly wird weder relativ noch absolut gespeichert. Zur Laufzeit einer Komponente ist es die Aufgabe des *Assembly Resolver*, eine referenzierte Komponente zu finden. Der Assembly Resolver verwendet dabei folgende Standardsuchstrategie:

- *Global Assembly Cache* (nur wenn die Assembly einen *Strong Name* besitzt!)
- *Anwendungsverzeichnis/Assemblyname.dll*
- *Anwendungsverzeichnis/Assemblyname/Assemblyname.dll*
- *Anwendungsverzeichnis/Assemblyname.exe*
- *Anwendungsverzeichnis/Assemblyname/Assemblyname.exe*

Im Normalfall wird eine Assembly nur anhand ihres Namens identifiziert. Bei der Referenzierung einer Assembly mit *Strong Name* werden neben dem Namen auch die Versionsnummer, die Kulturinformation und das *Public Key Token* (eine Kurzform des öffentlichen Schlüssels) berücksichtigt. Nur wenn alle vier Informationen zutreffend sind, akzeptiert der Assembly Resolver eine Assembly.

Satelliten-Assemblies werden in Unterverzeichnissen gesucht, die dem Sprachkürzel entsprechen:

- *Anwendungsverzeichnis/Sprachkürzel/Assemblyname.dll*
- *Anwendungsverzeichnis/Sprachkürzel/Assemblyname/Assemblyname.dll*
- *Anwendungsverzeichnis/Sprachkürzel/Assemblyname.exe*
- *Anwendungsverzeichnis/Sprachkürzel/Assemblyname/Assemblyname.exe*

Das Verhalten des Assembly Resolvers kann durch eine Anwendungskonfigurationsdatei beeinflusst werden. Im Rahmen einer Anwendungskonfigurationsdatei sind folgende Anweisungen möglich:

- Hinzufügen von Suchpfaden zur Suchstrategie des Assembly Resolvers (z. B. ein Unterverzeichnis */bin*)
- Angabe eines expliziten Standorts für einzelne Assemblies
- Anweisung, dass eine andere Versionsnummer einer Assembly verwendet werden soll

Eine *Publisher Policy* ist eine als Assembly kompilierte XML-Konfigurationsdatei, die eine Versionsnummernumleitung für eine Shared Assembly ausführt. Die Publisher-Policy-DLL wird selbst als Shared Assembly im Global Assembly Cache installiert.

Side-by-Side Execution

Als *Side-by-Side Execution* wird die Funktionalität des .NET Framework bezeichnet, dass verschiedene Anwendungen die gleiche Komponente in verschiedenen Versionen auf einem einzigen System nutzen können, ohne dass es zu einer »DLL-Hölle« kommt. Side-by-Side Execution bezieht sich auch darauf, dass unterschiedliche Versionen einer Assembly verschiedene auf einem System vorhandene Versionen der .NET-Laufzeitumgebung nutzen können.

Verbreitung und Installation von .NET-Anwendungen

Zur Installation von .NET-Anwendungen und .NET-Softwarekomponenten bietet das .NET Framework seit .NET 2.0 verschiedene Optionen. Gegenüber dem Framework 1.x ist das Click-Once-Deployment hinzugekommen. Die nachfolgende Tabelle gibt einen Überblick über die Anwendbarkeit der verschiedenen Verteilungsverfahren auf die unterschiedlichen Anwendungstypen.

Installationsverfahren Anwendungstyp	Reines XCopy	XCOPY + CLI	XCOPY + GUI	MSI	Not-Touch- Deployment	Click-Once Deployment
Konsolenanwendung	X			X	X	X
Windows-Anwendung	X			X	X	X
Windows-Dienst	(X)	X		X		
Web-UI	(X)	X	X	X		
XML-Webservice		X	X	X		
Komponente (privat)	X			X		
Komponente (global)		X		X		
Komponente (Serviced Component)		X	X	X		
WMI-Provider		X		X		
SQL Server-Erweiterung		X	X	X		

Tabelle 4.6 Überblick über die möglichen Deployment-Typen in Bezug auf die verschiedenen Anwendungstypen

XCOPY-Installation

Die meisten .NET-Anwendungen müssen nicht mehr installiert werden, sondern können einfach an einen beliebigen Ort kopiert und von dort gestartet werden. Der Begriff XCopy-Deployment nimmt Bezug darauf, dass zum Installieren einer .NET-Anwendung der DOS-Befehl XCopy ausreicht.

Diese Rückbesinnung auf die Wurzeln von Windows ist möglich durch den Verzicht auf die Registrierungsdatenbank als Konfigurationsspeicher und durch die generelle Speicherung von DLLs in einem zentralen Verzeichnis (/System32). Im .NET Framework werden Komponenten üblicherweise im Anwendungsver-

zeichnis abgelegt. Anwendungsspezifische Konfigurationsinformationen werden in Form von XML-Dateien ebenfalls im Anwendungsverzeichnis abgelegt.

Das XCopy-Deployment hat seine Grenzen, wenn

- gewünscht ist, dass mehrere Anwendungen sich Komponenten teilen (d.h. eine Installation der Komponente in den Global Assembly Cache – GAC – gewünscht ist)
- die Anwendungsart eine spezielle Verzahnung mit dem Betriebssystem (z. B. WMI-Provider, Windows-Dienste) oder einer Anwendung (Web- und SQL Server) erfordert

Microsoft Windows Installer (MSI)

Mithilfe des Microsoft Windows Installer (MSI) kann man jegliche Form von .NET-Anwendungen (ebenso wie Nicht-.NET-Anwendungen) installieren. Visual Studio oder Produkte von Drittanbietern (z.B. *Wise Installer* und *InstallShield*) helfen bei der Erstellung von MSI-Installationspaketen. MSI-Pakete können sowohl das Kopieren der Dateien als auch sämtliche Konfigurationseinstellungen vornehmen.

Kommandozeilenwerkzeuge

Für einige Anwendungsarten, die nicht mit XCopy-Deployment alleine auskommen, ist die Verwendung spezifischer Kommandozeilenwerkzeuge aus dem .NET Framework SDK leichtgewichtiger und schneller im Vergleich zur Verwendung von MSI-Paketen:

- Zur Aufnahme einer Assembly in den GAC kann das Werkzeug *gacutil.exe* aus dem .NET Framework SDK verwendet werden. Auf keinen Fall sollte man eine Assembly durch einfache Kopierbefehle in die Verzeichnisstruktur des GAC packen, weil Microsoft sich vorbehält, die interne Struktur des GAC jederzeit zu ändern, und weil dieses Verfahren auch unnötig kompliziert wäre.
- *InstallUtil.exe* ist ein Werkzeug aus dem .NET SDK, das der Installation von eng mit dem Betriebssystem verzahnten Anwendungen dient. Diese Installationsmethode wird insbesondere benötigt für Windows-Systemdienste, PowerShell-Erweiterungen und WMI-Provider.
- *Regsvcs.exe* dient der Installation von Serviced Components im COM+-Anwendungsserver
- Eine Webanwendung unterstützt reines XCopy-Deployment nur, wenn sie bereits existiert. Sofern sie noch anzulegen ist, müssen die mit dem Internet Information Server mitgelieferten Scripts *iisweb.vbs* und *iisvdir.vbs* zusätzlich verwendet werden, damit die entsprechenden Eintragungen in der IIS-Metabase erfolgen. Erst ab Internet Information Server 7.0 ist ein komplettes XCopy-Deployment möglich.

ACHTUNG Zu beachten ist, dass die Installation in das Global Assembly Cache, das Windows-Dienste-Verzeichnis in der Registrierungsdatenbank, die WMI-Repository und die Metabase des IIS-Webserver in der Regel Administratorrechte erfordert.

Auf 64-Bit-Systemen gibt es eine 32-Bit- und eine 64-Bit-Version von *InstallUtil.exe*. Wenn Sie z. B. eine PowerShell-Erweiterung installieren wollen, die auf Access-Datenbanken zugreift, geht das im 64-Bit-Modus nicht mangels Treiber für Microsoft Access. In diesem Fall müssen Sie mit der 32-Bit-Version von *InstallUtil.exe* arbeiten und auch anschließend die 32-Bit-PowerShell starten.

GUI-Werkzeuge

Für einige .NET-Anwendungstypen existieren GUI-Werkzeuge zur Installation:

- das MMC-Snap-in *Internetinformationsdienste-Manager für Webanwendungen*,
- das MMC-Snap-in *Komponentendienste für Serviced Components* sowie
- das SQL Server 2005/2008 Management Studio für SQL Server-Erweiterungen

No-Touch-Deployment (NTD)

Als No-Touch-Deployment bezeichnet Microsoft die Installation von Anwendungen von einem Webserver. Dabei erhält der Benutzer einen URL zu einer .NET-Anwendung (.exe). Das .NET Framework lädt die Anwendung und alle referenzierten Assemblys in ein spezielles Unterverzeichnis des GAC, den *Assembly Download Cache (ADC)* (*c:/windows/assembly/download*), und führt die Anwendung von dort aus. Wenn auf dem Webserver eine neue Version einer der beteiligten Assemblys bereitgestellt wird, findet eine automatische Aktualisierung statt.

NTD funktioniert allerdings nur im Online-Betrieb (der Webserver muss immer verfügbar sein) und erlaubt keine Einträge der Anwendung in das Startmenü, in die Softwareliste in der Systemsteuerung oder in die Registrierungsdatenbank. Der ADC kann verwaltet werden über das SDK-Werkzeug *gacutil.exe*.

Click-Once-Deployment (COD)

Click-Once-Deployment ist eine mit .NET 2.0 eingeführte Weiterentwicklung des No-Touch-Deployment (NTD). Click-Once erlaubt die Verteilung von Windows-Anwendungen über Webserver und Netzwerklaufwerke. Auch eine automatische Aktualisierungsfunktion ist enthalten. Ein vergleichbares Konzept in der Java-Welt ist Java Web Start.

Eine Click-Once-Anwendung wird beim ersten Aufruf für den aktuellen Benutzer installiert und bei Bedarf automatisch aktualisiert. Der Endbenutzer startet die Anwendung durch einen Mausklick auf einen Link in Windows oder auf einer Webseite. Sofern die Anwendung in seinem lokalen Application Cache (*/Dokumente und Einstellungen/(User)/Lokale Einstellungen/My Applications*) noch nicht in der aktuellsten Fassung existiert, lädt das .NET Framework die Anwendung herunter und installiert sie. Ein Vorteil gegenüber NTD ist die Möglichkeit, Einträge im Startmenü und in der Softwareliste in der Systemsteuerung vorzunehmen. Click-Once-Anwendungen funktionieren auch, wenn die Quelle nach dem ersten Herunterladen nicht mehr verfügbar ist. Das .NET Framework und andere Voraussetzungen (MDAC 9, MSDE, J# etc.) werden bei Bedarf mitinstalliert. Visual Studio (seit Version 2005) bietet eine Unterstützung für das Publizieren von Click-Once-Anwendungen direkt aus der Entwicklungsumgebung heraus und erstellt automatisch eine HTML-Seite für den Download. Gegenüber einer Installation mit Microsoft Installer gibt es aber weiterhin einige Einschränkungen (beispielsweise nur beschränkter Zugriff auf die Registrierungsdatenbank, Installation gilt immer nur für angemeldeten Nutzer). Nicht möglich ist daher die Registrierung von Dateierweiterungen, mit denen die Anwendung verbunden werden soll. Erst seit Windows Vista hat Click-Once diese Funktion angeboten.

Möglich wird die automatische Aktualisierung durch zwei Zwischenschritte (*Manifest-based Activation*): Der Link zeigt nicht auf die EXE, sondern auf ein Deployment-Manifest (XML-Datei), das wiederum auf das Application-Manifest der aktuellsten Version zeigt. Das Application-Manifest enthält schließlich die Informa-

tionen über den Start einer Anwendung und die Abhängigkeit von anderen Anwendungen. Wichtig für die automatische Aktualisierung ist, dass die Anwendung nicht nur erneut publiziert wird, sondern auch tatsächlich eine neue Versionsnummer erhält. Signifikant sind hierbei die ersten drei Sektionen der Versionsnummer.

Weitere Fähigkeiten der Laufzeitumgebung

Dieser Abschnitt stellt kurz drei wichtige Dienste der Common Language Runtime (CLR) vor:

- Automatische Speicherbereinigung durch den Garbage Collector
- Prozessabgrenzung durch Application Domains
- Absicherung durch die Code Access Security (CAS)

Speicherbereinigung (Garbage Collector)

Im Gegensatz zu COM verfügt das .NET Framework über eine automatische Speicherverwaltung, die in der Common Language Runtime (CLR) implementiert ist. Die CLR enthält einen *Garbage Collector (GC)*, der im Hintergrund (in einem System-Thread) arbeitet und den Speicher aufräumt. Der Speicher wird allerdings nicht sofort nach dem Ende der Verwendung eines Objekts freigegeben, sondern zu einem nicht festgelegten Zeitpunkt bei Bedarf (*Lazy Resource Recovery*). Beim Aufräumen des Speichers erzeugt der Garbage Collector einen Baum aller Objekte, auf die es aktuell einen Objektverweis gibt. Der Speicher aller nicht mehr erreichbaren Objekte wird freigegeben.

Der Garbage Collector kann von einer Anwendung nur bedingt beeinflusst werden. Die Anwendung kann mit dem Befehl `System.GC.Collect()` dem Garbage Collector den Auftrag geben, tätig zu werden. Eine Anwendung kann jedoch eine Speicherbereinigung nicht verhindern.

Der Garbage Collector ruft die *Destruktoren (alias Finalizer)* der .NET-Objekte auf. Die Reihenfolge des Aufrufs und ob der Destruktor überhaupt aufgerufen wird, ist jedoch nicht deterministisch, d. h., es kann sein, dass ein Destruktor nicht aufgerufen wird. Beim Schließen einer .NET-Anwendung werden die Destruktoren der verbliebenen Objekte *nicht* aufgerufen.

HINWEIS

Um sich von den deterministischen Destruktoren der Sprache C++ abzuheben, spricht man in .NET von *Finalisierung* statt von *Destruktion*.

Klassen, bei denen der Aufruf des Destruktors wichtig ist, weil dabei Ressourcen freigegeben werden, müssen dem Disposable-Muster folgen und die Schnittstelle `System.IDisposable` mit der Methode `Dispose()` implementieren. Die Anwendung muss `Dispose()` manuell aufrufen. Die Programmiersprachen C# und Visual Basic unterstützen ein Programmblockkonstrukt mit Namen `using`. Am Ende eines `using`-Blocks wird für die im Kopf des Blocks angegebenen Variablen automatisch die `Dispose()`-Methode aufgerufen.

Prozessabgrenzung durch Application Domains

Eine Application Domain ist ein Konzept zur Abgrenzung von Anwendungen innerhalb eines Betriebssystemprozesses. Eine solche Domain teilt einen Windows-Prozess in mehrere Bereiche. Anwendungen, die in verschiedenen Application Domains laufen, sind so voneinander isoliert, als liefen sie in verschiedenen Prozessen. Eine Application Domain nennt man auch einen *Pseudo-Prozess* oder einen *logischen Unterprozess*.

Der Vorteil einer Application Domain gegenüber der Erzeugung verschiedener Prozesse ist, dass der Aufwand zur Erzeugung einer Application Domain und für einen Programmcode-Aufruf zwischen zwei Application Domains geringer ist als der für die Verwendung von Prozessen. Mehrere Assemblys können sich eine Application Domain teilen.

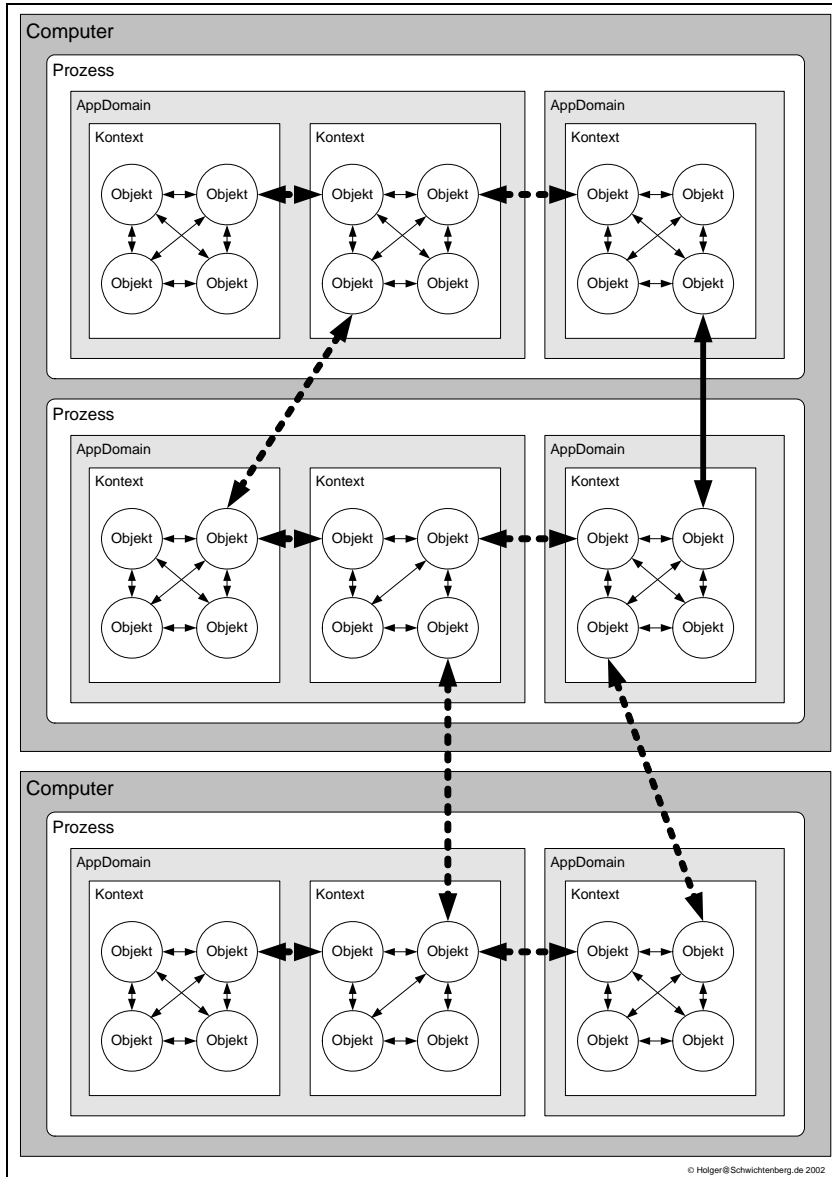


Abbildung 4.18 Application Domains

ACHTUNG Code in verschiedenen Application Domains muss ebenso mit Fernaufrufen kommunizieren wie Code in verschiedenen Prozessen.

Sicherheit (Code Access Security)

Der Schutz vor schädlichen Codes wird ein immer wichtigeres Thema. Die CLR bietet ein neues Sicherheitssystem, das nicht mehr nur die Rechte des Benutzerkontos berücksichtigt, unter dem der Code ausgeführt wird, sondern auch die Herkunft des Programmcodes. Das Sicherheitskonzept heißt Code Access Security (CAS) und ist die Weiterentwicklung des Zonenkonzepts des Internet Explorers (*My Computer, Local Intranet, Trusted Sites, Internet* etc.), des Microsoft Authenticode-Verfahrens zur digitalen Signierung von Programmcode und der Software Restriction Policy (SRP) in Windows (ab XP).

Die CLR ermittelt zur Bestimmung der Ausführungsrechte von Managed Code zunächst die Beweislage (engl. *Evidence*). Zur Beweislage gehören insbesondere der Autor des Codes (hinterlegt durch das Authenticode-Verfahren) und der Speicherort des Codes (Zonenkonzept). Auf dieser Basis werden die Rechte des Codes ggf. eingeschränkt. Selbstverständlich erhält der Code niemals mehr Rechte als der Benutzer, unter dem der Code läuft, denn das Windows-Sicherheitssystem wirkt nach wie vor zusätzlich.

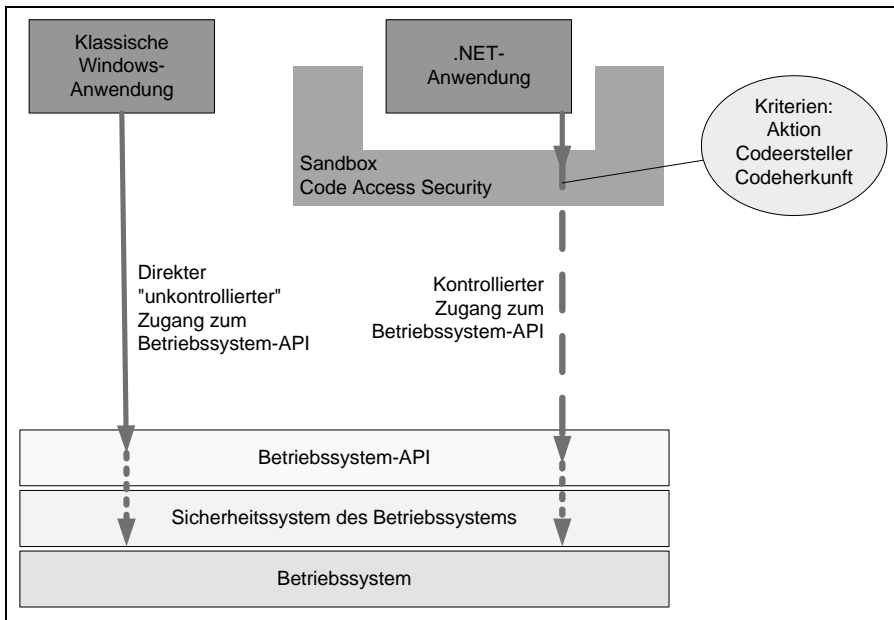


Abbildung 4.19 Sandbox-Konzept in der CAS

Codegruppen und Berechtigungssätze

Die CAS kann durch die Definition von Codegruppen und Berechtigungssätzen sehr fein konfiguriert werden. Codegruppen definieren Herkunftsbedingungen und Berechtigungssätze definieren erlaubte Aktionen. Jeder Codegruppe kann genau ein Berechtigungssatz zugeordnet werden.

Bei der Ausführung von Programmcode ordnet die CLR den Code in alle zutreffenden Codegruppen ein. Anschließend bildet sie die Vereinigungsmenge der Berechtigungssätze der zugeordneten Codegruppen. Bei referenzierten Assemblys werden auch die Rechte des Aufrufers berücksichtigt: Nur wenn die aufrufende Assembly die notwendigen Rechte besitzt, wird eine Aktion ausgeführt, selbst wenn die aufgerufene Assembly die Rechte besitzt.

Ein Berechtigungssatz ist eine Menge von Zugriffsrechten auf Ressourcen. Es gibt verschiedene Ressourcen (z.B. Dateisystem, Verzeichnisdienst, Benutzerschnittstelle etc.) und zu jeder Ressource spezifische Einstellungen.

TIPP

Wenn Sie die Möglichkeit suchen, Rechte auf eine Ressource zu verweigern, werden Sie im .NET Framework Version 2.0/3.0/3.5 nicht fündig. Es gibt nur die Möglichkeit, Rechte zu erteilen (Allow), nicht aber die Möglichkeit, Rechte zu verweigern (Deny). Wollen Sie eine Einschränkung definieren, müssen Sie zunächst einen geringeren Berechtigungssatz für diese Codegruppe wählen. Dieser Berechtigungssatz muss minimal sein, also nur das enthalten, was der am wenigsten berechtigte Code dürfen soll. Danach müssen Sie eine neue Codegruppe (und ggf. einen neuen Berechtigungssatz) für den Code definieren, der mehr darf.

Ebenen

Weiterhin ist zu beachten, dass Codegruppen und Berechtigungssätze auf vier Ebenen festgelegt werden können: Benutzer, Computer, Organisation und Application Domain. Hierbei wird die Schnittmenge gebildet, ein Recht wird also nur dann zugeordnet, wenn es auf allen vier Ebenen gewährt wird. Die Standardeinstellung auf Organisations-, Benutzer- und Application Domain-Ebene ist jedoch, dass die Codegruppe *All Code* den Berechtigungssatz *Full Trust* besitzt. In der Standardkonfiguration ist also allein die Computer-Ebene entscheidend.

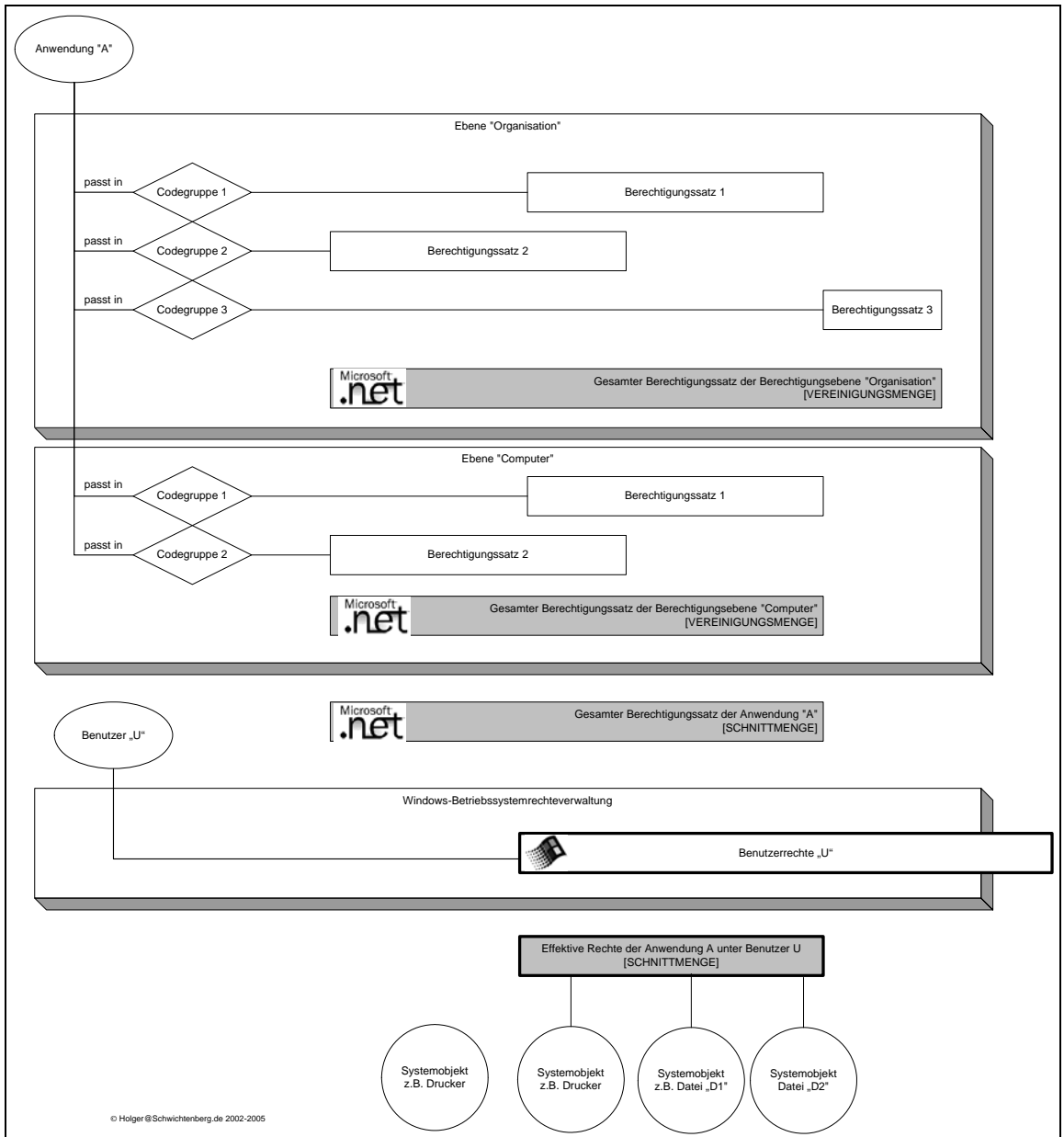


Abbildung 4.20 Mischung aus Codeberechtigungen und Benutzerberechtigungen in der .NET Framework Code Access Security

Standardrechte

Abbildung 4.21 zeigt die Standardzuordnung von Codegruppen und Berechtigungssätzen.

TIPP Für .NET-Softwareentwickler ist die Standardkonfiguration oft unzureichend, wenn Quellcodedateien auf einem Netzlaufwerk gespeichert und von dort in Visual Studio geöffnet werden. Programmcode auf Netzlaufwerken hat in der Standardkonfiguration nur eingeschränkte Rechte. Hier sollte man die Sicherheit für die Entwickler auf *Full Trust* erhöhen.

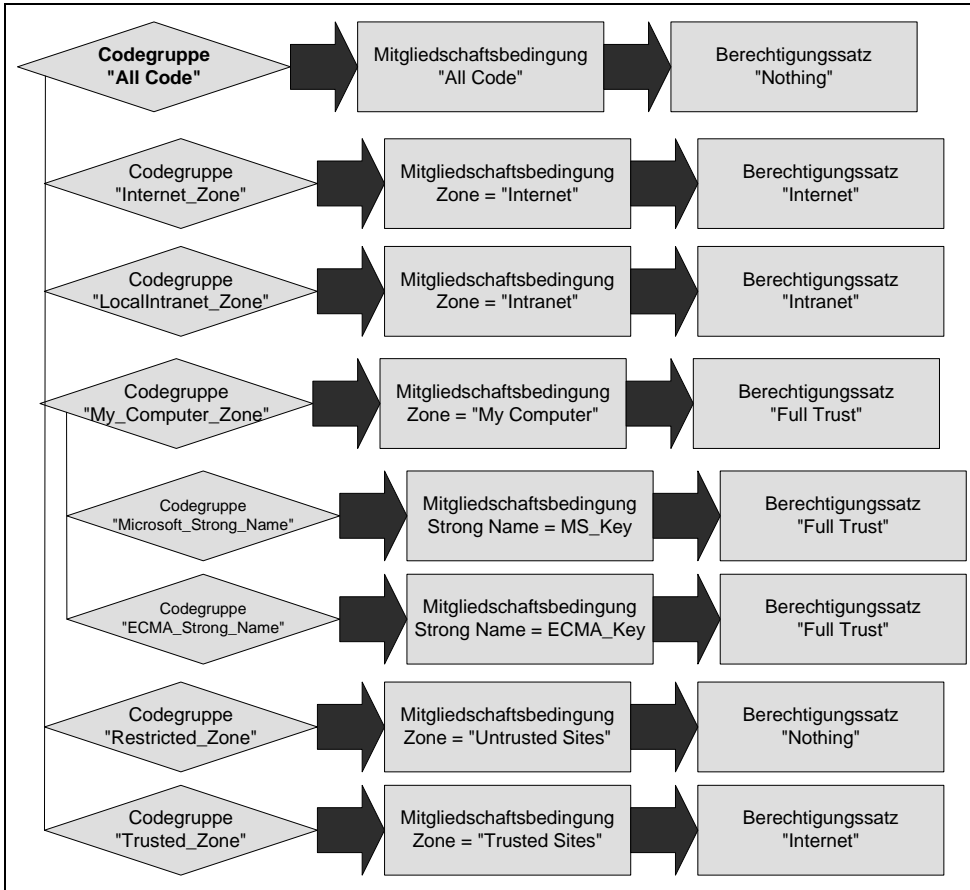


Abbildung 4.21 Standardzuordnung von Codegruppen und Berechtigungsätzen

Deklarative Sicherheitsanforderungen

Eine .NET-Anwendung kann und sollte durch Metadaten deklarieren, welche Rechte sie zur Ausführung benötigt. Die CAS verhindert, dass eine .NET-Anwendung startet, die nicht alle deklarierten Rechte besitzt. Ohne eine Rechtedeklaration könnten fehlende Rechte auf für den Benutzer unangenehme Weise erst später auffallen, z.B. wenn der Benutzer versucht, ein erstelltes Dokument zu speichern, aber die Anwendung keine Zugriffsrechte auf das Dateisystem besitzt.

Beispiel

Die folgende Anwendung fordert volle Zugriffsrechte auf das Dateisystem.

```
[assembly: System.Security.Permissions.FileIOPermission
(System.Security.Permissions.SecurityAction.RequestMinimum)]
```

Werkzeuge

Die gesamten CAS-Sicherheitseinstellungen werden in XML-Konfigurationsdateien im `/Config`-Verzeichnis der .NET Framework-Installation abgelegt. Die Konfiguration ist möglich über die direkte Manipulation der XML-Dateien (nicht empfehlenswert), über das Kommandozeilenwerkzeug *caspol.exe* oder über ein mit dem .NET Framework SDK bzw. Windows SDK mitgeliefertes MMC-Snap-in mit Namen *.NET Configuration (mscorlib.msc)*.

HINWEIS

Es gibt in .NET 3.0/3.5/4.0 keine neuere Version des MMC-Snap-Ins.

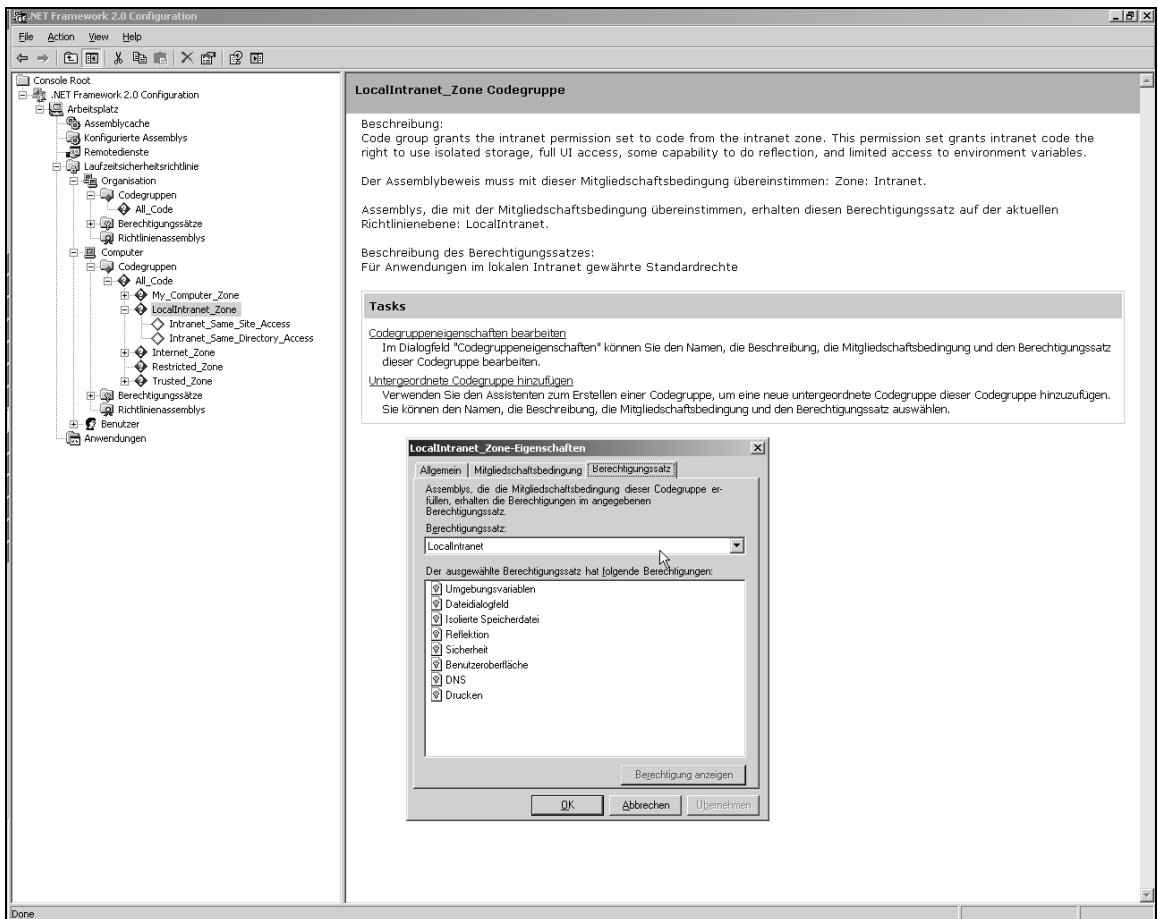


Abbildung 4.22 Anzeige des eingestellten Berechtigungssatzes für die Codegruppe *Local_Internet_Zone*

Security Transparency in .NET 4.0

Weil die CAS als zu komplex und dennoch wirkungslos galt (viele schalteten sie wegen der Komplexität ab!), verabschiedet sich Microsoft in .NET 4.0 von dem Richtlinienkonzept der CAS. Anwendungen, die vom Windows Explorer oder der Kommandozeile gestartet werden (.NET Shell Runtime Host) laufen unter vollen Rechten. Lediglich noch *gehostete .NET-Anwendungen*, also solche, die in einer anderen Anwendung laufen (z.B. Internet Explorer oder Microsoft Office sowie dem Internet Information Server – IIS) werden eingeschränkt.

In .NET 4.0 baut Microsoft einen anderen Mechanismus aus, den es seit .NET 2.0 gibt: Security Transparency. Security Transparency kennt drei Sicherheitsstufen:

- **Transparent Code:** Die Rechte sind eingeschränkt durch den Host. Dieser Code kann nur Transparent Code und Safe-Critical Code aufrufen. Auch Native Code kann nicht aufgerufen werden.
- **Safe-Critical Code:** Dieser Code hat volle Rechte. Transparent Code kann diesen Code aufrufen.
- **Security-Critical Code:** Dieser Code hat volle Rechte. Transparent Code kann diesen Code nicht aufrufen.

HINWEIS Das Sicherheitssystem (weder CAS noch Security Transparency) wird in diesem Buch nicht näher behandelt. An dieser Stelle soll es bei dem Hinweis auf die Grundgedanken bleiben.

Interoperabilität

Um die Akzeptanz des .NET Framework zu fördern, hat Microsoft sinnvollerweise eine Interoperabilität mit klassischen Windows-Funktionsbibliotheken und COM-Komponenten sichergestellt.

Interoperabilität zu klassischen C-Bibliotheken

Die CLR ermöglicht den Aufruf von klassischen C-Bibliotheken (beispielsweise der WIN32-API) durch einen Mechanismus mit Namen *Platform Invoke (P/Invoke)*. Aus der anderen Richtung ist eine Steuerung der CLR und in ihr geladener Komponenten möglich durch die COM-basierten Programmierschnittstellen der *mscorlib.dll*.

Interoperabilität zu COM

.NET-Komponenten können COM-Komponenten nutzen. Andersherum kann eine .NET-Anwendung auch als COM-Komponente aufgerufen werden. Die Vermittlung zwischen COM und .NET ist gelöst durch Proxy-Objekte. COM-Komponenten werden von .NET aus über einen so genannten *Runtime Callable Wrapper (RCW)* angesprochen. Der RCW setzt die COM-Schnittstellen in .NET-kompatible Schnittstellen um. Wenn ein Hersteller für eine COM-Komponente einen RCW mitliefert, dann spricht man von einer *Primary Interop Assembly (PIA)*.

Ein RCW kann erstellt werden mit dem Kommandozeilenwerkzeug *tlbimp.exe* aus dem .NET Framework SDK oder durch Generierung einer Referenz auf eine COM-Komponente in einem Visual Studio-Projekt. Eingabe für *tlbimp.exe* kann eine COM-DLL/-EXE mit integrierter Typbibliothek oder eine eigenständige COM-Typbibliothek (*.tlb*-Datei) sein.

.NET-Komponenten werden von COM-Clients über einen *COM Callable Wrapper (CCW)* angesprochen. Normalerweise stellen die .NET-Entwicklungswerkzeuge einen CCW automatisch zur Verfügung. Ein CCW kann aber auch selbst entwickelt werden. Ein CCW kann mit *tlbexp.exe* erstellt werden.

Die Interoperabilität zu COM wurde mit .NET 2.0 verbessert durch den vereinfachten Umfang mit Funktionszeigern und Arrays.

HINWEIS Neu in .NET 4.0 ist, dass man den RCW-Code direkt in die aufrufende Assembly integrieren kann. Wrapper-Assemblys bzw. PIAs entfallen dann. Visual Studio 2010 leitet standardmäßig die benötigten Informationen über die referenzierte COM-Bibliothek direkt in die aktuelle Assembly, wobei dabei eine Beschränkung auf jene Aspekte, welche von dieser auch wirklich verwendet werden, stattfindet. Um dies zu verhindern und somit die Erstellung von PIAs zu veranlassen, ist die Eigenschaft *Embed Interop Types* bei den hinzugefügten COM-Bibliotheken auf *false* zu setzen. Microsoft nennt diese neue Funktion *Embedded Types* oder *NoPIA*.

Interoperabilität zu anderen Komponentenplattformen

Microsoft stellt als Interoperabilität zu anderen Komponentenplattformen nur zwei Möglichkeiten bereit:

- Migration von Java-Quellcode in C#-Quellcode (Java Language Conversion Assistant)
- Interoperabilität mit XML-Webservices (mit ASMX oder WCF)

Eine direkte Interoperabilität über Komponentenbrücken (*Component Bridges*) ist von Microsoft nicht vorgesehen. Komponentenbrücken können eine sehr viel engere Kopplung zwischen zwei Komponentenplattformen realisieren (bis hin zur In-Prozess-Kopplung). Component Bridges werden derzeit lediglich von Drittanbietern zur Verfügung gestellt.

Kopplung	Verfahren	Produkte
.NET → Java	Neukompilierung von Java-Code für .NET	Programmiersprache J# der Firma Microsoft
.NET → Java	Transfer von Java-Quellcode nach C#	Java Language Conversion Assistant (JLCA) der Firma Microsoft
.NET → Java	Transformation von Java-Bytecode nach CIL	IKVM.NET (Open Source)
.NET → Java	In-Prozess-Kopplung (Laden der JVM in den CLR-Prozess)	JuggerNET der Firma Codemesh
.NET ↔ Java	.NET Remoting-Implementierung für Java	J-Integra.NET der Firma Intrinsic JNBridgePro der Firma JNBridge
.NET ↔ Java/CORBA	IIOP-Channel für .NET Remoting	Janeva der Firma Borland IIOP.NET (Open Source) Remoting.Corba (Open Source)
.NET ↔ CORBA	CORBA-ORB in .NET geschrieben	MiddCor der Firma MiddTec

Tabelle 4.7 Interoperabilitätsprodukte

Auch die Implementierung einer individuellen In-Process-Bridging-Lösung zwischen Java und .NET ist unter Nutzung des .NET-P/Invoke-Mechanismus und des Java Native Interface (JNI) mit vertretbarem Aufwand realisierbar.

.NET auf 64-Bit-Systemen

Seit der Version 2.0 bietet Microsoft auch eine 64-Bit-Version des .NET Framework an. Durch das Zwischensprachkonzept des .NET Framework ist die Portierung einer .NET-Anwendung auf eine andere Plattform grundsätzlich möglich. Dies gilt auch für die neuen 64-Bit-Windows-Betriebssysteme. Egal ob der Entwickler eine 32- oder 64-Bit-Plattform verwendet hat, die Anwendung wird beim Kunden sowohl auf 32- als auch auf 64-Bit-Systemen laufen. Im Detail lohnt jedoch eine nähere Betrachtung.

Die 64-Bit-Welt ist keine einfache, da es nicht eine, sondern zwei verschiedene 64-Bit-Windows-Betriebssysteme gibt. Zum einen existiert ein 64-Bit-Windows für die Intel Itanium-Architektur (kurz: IA64), zum anderen eines für die 64-Bit-Prozessortypen von AMD (AMD 64), die es bei Intel auch gibt unter den Namen Intel 64 bzw. "Extended Memory 64 Technology" (EM64T). 32-Bit-Anwendungen können auf 64-Bit-Systemen mithilfe eines Emulators laufen, den Microsoft WOW64 (Abkürzung für Windows-On-Windows64 oder Windows32-On-Windows64) nennt.

Der vor .NET 2.0 gebräuchliche Just-in-Time-Compiler des .NET Framework erzeugte immer 32-Bit-Code. Grundsätzlich ist dieser mithilfe des WOW64 unter 64-Bit-Systemen lauffähig und wird auch dort automatisch installiert im Verzeichnis »Microsoft.NET/Framework«. Um die Leistung eines 64-Bit-Systems voll zu nutzen, wäre aber eine Umwandlung des Zwischencodes (Common Intermediate Language, CIL) in 64-Bit-Maschinencode wünschenswert. Microsoft hat daher für 64-Bit-Systeme einen neuen Just-in-Time-Compiler geschrieben, der sich installiert im Verzeichnis »Microsoft.NET/Framework64«. Außerdem war es notwendig bzw. sinnvoll, auch den Garbage Collector, die Ausnahmebehandlung und die Codegenerierung anzupassen. Der Großteil der Klassen der .NET-Klassenbibliothek, die ja selbst als CIL-Code vorliegt, benötigt hingegen keine Sonderanpassung an die 64-Bit-Welt.

An diesem Punkt stellt sich die Frage, ob alle .NET-Anwendungen beim Start auf 64-Bit-Systemen in 64-Bit-Maschinencode umgewandelt werden. Microsoft hat sich dagegen entschieden, um Kompatibilitätsprobleme zu vermeiden.

Folgende .NET-Anwendungen werden an den 32-Bit-Just-in-Time-Compiler übergeben:

- alle .NET-Assemblies, die mit .NET-1.0- und -1.1-Compilern erzeugt wurden
- alle .NET-Assemblies, die nicht ausschließlich aus CIL-Code bestehen, sondern auch Native Code enthalten (gemischte Assemblies, die mit Managed C++ erzeugt werden können)
- alle .NET-Assemblies, die explizit die Umwandlung in 32-Bit-Code mit dem Flag *32-Bit-Required* anfordern

WICHTIG Besondere Aufmerksamkeit sollten Sie bei der Verwendung von Reflection-Methoden wie *GetCallingAssembly* und *GetExecutingAssembly* widmen. Durch das aggressivere Inlining-Verhalten des 64-Bit-Just-In-Time-Compilers kann es passieren, dass Code einer Assembly gar nicht ausgeführt wird, in dem die Assembly geladen und aufgerufen, sondern der Code einfach von der zu ladenden Assembly in die ausgeführte übertragen wird. Dadurch verhalten sich die Methoden *GetCallingAssembly* oder *GetExecutingAssembly* unter Umständen nicht mehr wie erwartet.³

³ Siehe auch: <http://www.hanselman.com/blog/ReleaseISNOTDebug64bitOptimizationsAndCMethodInliningInReleaseBuildCallStacks.aspx>.

.NET auf 64-Bit-Systemen: Entscheidungsdiagramm

Die nachstehende Abbildung zeigt ein Entscheidungsdiagramm für .NET-Code auf 64-Bit-Systemen. Die erste Entscheidung betrifft dabei nicht die .NET Framework-Version, sondern das Dateiformat. .NET 1.x verwendet immer das PE32-Dateiformat.

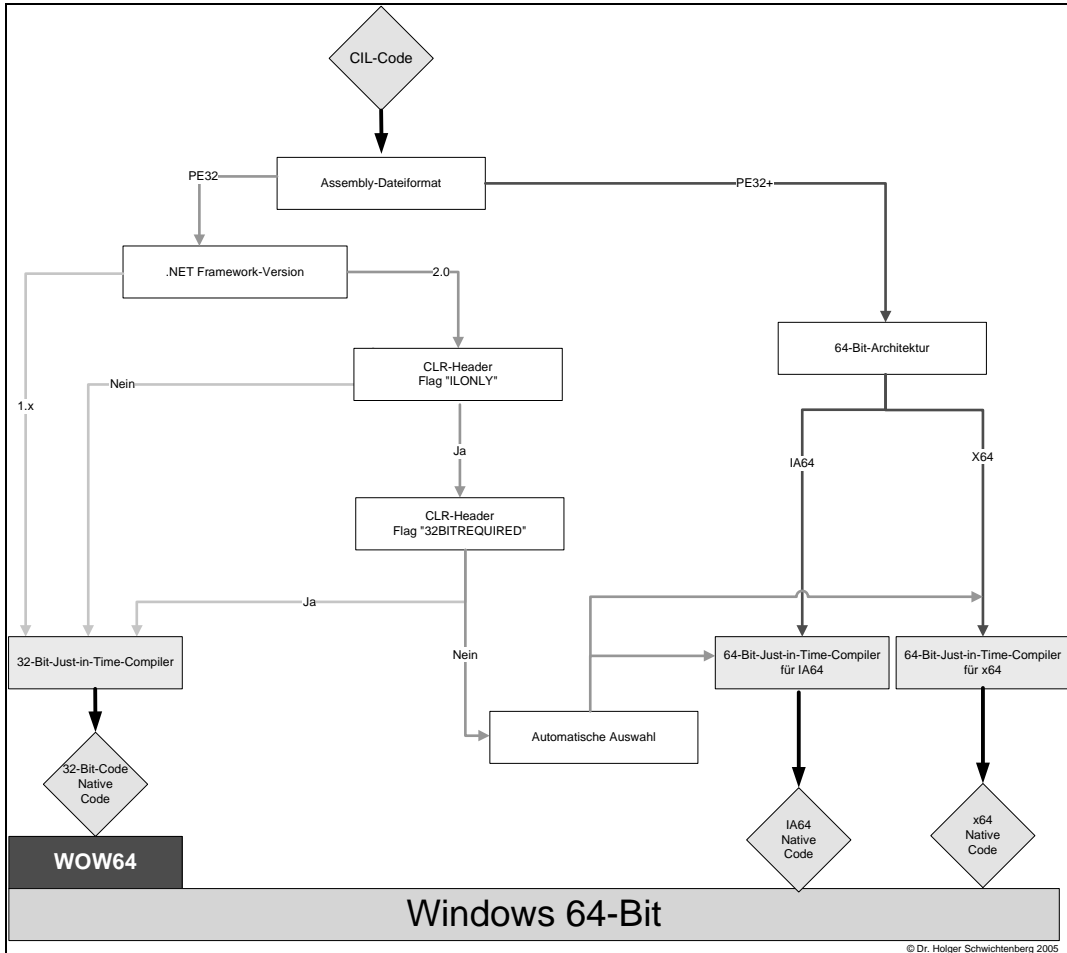


Abbildung 4.23 Entscheidungsbaum für .NET-Anwendungen auf 64-Bit-Systemen

Das neuere PE32+ mit einem größeren Adressraum verstehen nur 64-Bit-Systeme. Visual Studio (ab Version 2005) bietet die Option, PE32+ anstelle von PE32 zu erzeugen. Dem Vorteil des größeren Adressraums stehen aber zwei gravierende Nachteile gegenüber: Erstens können PE32+-Assemblies nicht auf 32-Bit-Systemen ausgeführt werden. Und zweitens sind die erzeugten PE32+-Assemblies auf einen bestimmten Prozesstyp beschränkt, d. h. entweder PE32+ für IA64 oder PE32+ für x64. Man benötigt also immer zwei Kompilate, wenn man beide 64-Bit-Prozesstypen abdecken will. Der Versuch, eine PE32+ für IA64-Assembly auf einem x64-System auszuführen, resultiert in einem Fehler. In der Praxis schwindet aber die Bedeutung der IA64-Architektur.

Wer plattformunabhängig sein will, muss also PE32 verwenden. Wenn ein 64-Bit-Windows eine PE32-Assembly starten soll, wird zunächst geprüft, ob es aus oben genannten Gründen gilt, die Assembly an den 32-Bit-Compiler zu übergeben. Wenn es keinen Grund für 32-Bit gibt, erhält der vorhandene 64-Bit-Just-in-Time-Compiler die Assembly. Dabei ist es dann egal, welcher Prozessortyp vorliegt.

Das Flag *32-Bit-Required* muss ein Entwickler setzen, wenn die .NET-Assembly externen 32-Bit-Native-Code in Form von C-DLLs oder COM-Komponenten verwendet, die es nicht in einer 64-Bit-Variante gibt. Würde eine solche .NET-Assembly in 64-Bit-Maschinencode umgewandelt, käme es zu einem Laufzeitfehler im Moment des Aufrufs des 32-Bit-Codes aus der Assembly heraus. Der Fehler kann beim Start der Anwendung nicht bemerkt werden.

Genauso wie eine in 64-Bit-Code übersetzte Assembly keinen Native Code in 32-Bit aufrufen kann, kann auch aus einem 32-Bit-Kompilat heraus kein 64-Bit-Native-Code aufgerufen werden. Wichtig ist also, dass auf der Maschinencode-Ebene die eigene Welt nicht verlassen wird. Die Flags werden abgelegt im CLR-Header der Assembly. Mithilfe von Klassen im Namensraum `System.Reflection` kann man die Konfiguration auslesen.

.NET auf 64-Bit-Systemen: Datentypen

Das Datentypproblem zwischen verschiedenen Sprachen hat .NET gelöst durch das Common Type System (CTS) und die Common Language Specification (CLS). Gleichzeitig gelöst ist damit auch das Datentypproblem zwischen verschiedenen Prozessoren. Der Typ `System.Int64` umfasst 64 Bit in 8 Bytes. Ob diese 64 Bit physikalisch in einer oder zwei Speicheradressen liegen, kann einem .NET-Programm völlig egal sein. Die Größe der .NET-Datentypen ist unabhängig von der Plattform.

Aber keine Regel ohne Ausnahme. Ein einziger Typ im .NET Framework kümmert sich sehr wohl um die Speicherzellenbreite: `System.IntPtr`. Dieser Typ zur Repräsentation von Zeigern umfasst prozessorabhängig entweder 32 oder 64 Bit. Diese Eigenschaft kann man sich zu Nutze machen und diese Eigenschaft muss man fürchten in Hinblick auf die Interoperabilität mit Unmanaged Code. Hier helfen aber die Klassen in `System.Runtime.InteropServices` durch die Vorgabe eines Speicherlayouts, den Auftritt von Doktor Watson zu verhindern. Keineswegs sollten Entwickler `System.Int32` und `System.IntPtr` synonym verwenden.

Eine weitere (kleine) Herausforderung stellen Fließkommazahlen dar. Gemäß IEEE-Standard 754 können Fließkommaoperationen abweichen in Abhängigkeit von der Prozessorbreite. Dieses Problem lässt sich umgehen, indem man exakte Vergleiche zwischen Fließkommazahlen vermeidet.

Keine Probleme gibt es bei der Speichergröße. Assemblys, die mit einem 64-Bit-Just-in-Time-Compiler übersetzt werden, können den größeren Hauptspeicher (maximal 16 Terabyte statt 4 Gigabyte) nutzen. Die Größenrestriktion für einzelne .NET-Objekte (2 Gigabyte) ist jedoch gleich geblieben.

.NET auf 64-Bit-Systemen: Leistung

Neben der Kompatibilität stellt sich auch die Frage, ob .NET-Anwendungen auf 64-Bit-Systemen eine größere Performanz erreichen. Tatsächlich vergleichen kann man dabei aber nur das Leistungsverhalten innerhalb eines Systems, also wenn man ein Programm auf einem 64-Bit-System einerseits mit dem 64-Bit-Just-in-Time-Compiler und andererseits mit dem 32-Bit-Kollegen übersetzen lässt.

Zum Testen der Leistung kam eine in C# geschriebene Konsolenanwendung zum Einsatz, die 100 Millionen Zufallszahlen in einem Array erzeugt und dann in einer Schleife, die 100 Mal durchlaufen wird, die Zufallszahlen aufaddiert. Der Speicherplatzbedarf im Hauptspeicher liegt bei rund 790 MB.

Das Durchlaufen des Arrays findet in drei Varianten statt:

- mit einer foreach-Schleife
- mit einer for-Schleife
- mit einer while-Schleife unter Einsatz von Zeigerarithmetik (Unsafe Code)

Auf jegliche Form von Zugriff auf Teile des Betriebssystems wurde bewusst verzichtet.

Bei den Testergebnissen (siehe Tabelle) überrascht nicht, dass durch nativen 64-Bit-Code die Leistung wesentlich besser ist als beim Einsatz des WOW64. Ebenfalls erwartungsgemäß schlägt es sich nicht in der Leistung nieder, ob man PE32 oder PE32+ als Dateiformat verwendet. Interessant ist jedoch, dass die 32-Bit-Systeme bei der Leistung in einigen Fällen nicht hinterherhinken, obwohl die Prozessoren älter sind. Dies deckt sich mit der Aussage von Microsoft in [MSDN14], dass die Geschwindigkeit von Anwendungen im WOW64 ähnlich ist wie die von Anwendungen auf 32-Bit-Systemen. Deutlich schlechter wären die Ergebnisse übrigens für 32-Bit-Kompile auf Itanium-Systemen, weil die IA64-Architektur keine Hardware-Unterstützung für die Emulation bietet.

Hardware	Betriebssystem	Visual Studio Konfiguration	Datei-format	Gesetzte CLR Header-Folge	Just-in-Time-Compiler	Emulator	For-each-Schleife	For-Schleife	Unsafe
32-Bit-System (Intel Pentium 4, 3,0 GHz)	Windows Server 2003 Service Pack 1	AnyCPU	PE32	ILONLY	32-Bit-Just-in-Time-Compiler		ca. 47 Sekunden	ca. 23 Sekunden	ca. 21 Sekunden
32-Bit-System (Intel Pentium M, 1,6 GHz)	Windows Server XP Professional Service Pack 2	AnyCPU	PE32	ILONLY	32-Bit-Just-in-Time-Compiler		ca. 52 Sekunden	ca. 41 Sekunden	ca. 38 Sekunden
64-Bit-System (Intel Pentium E, 3,2 GHz, Dual-Core)	Windows Server XP Professional x64 Service Pack 1	x86	PE32	ILONLY, 32-BIT-REQUIRED	32-Bit-Just-in-Time-Compiler	WOW 64	ca. 32 Sekunden	ca. 29 Sekunden	ca. 22 Sekunden
64-Bit-System (Intel Pentium E, 3,2 GHz, Dual-Core)	Windows Server XP Professional x64 Service Pack 1	AnyCPU	PE32	ILONLY	64-Bit-Just-in-Time-Compiler		ca. 11 Sekunden	ca. 11 Sekunden	ca. 11 Sekunden
64-Bit-System (Intel Pentium E, 3,2 GHz, Dual-Core)	Windows Server XP Professional x64 Service Pack 1	x64	PE32+	ILONLY	64-Bit-Just-in-Time-Compiler		ca. 11 Sekunden	ca. 11 Sekunden	ca. 11 Sekunden

Tabelle 4.8 Ergebnisse des Testprogramms, Array als int[] deklariert

Auf exakte Zahlen wurde verzichtet, weil es sehr viele Einflussfaktoren auf die Leistung gibt. Die Tabelle soll nur die viel zitierten *Hausnummern* veranschaulichen.

Aufgrund der Verschiedenartigkeit der Prozessoren konnte der durchgeführte Geschwindigkeitstest keineswegs beweisen, dass für .NET-Anwendungen 64-Bit-Systeme schneller sind als 32-Bit-Systeme. Deutlich wurde aber, dass .NET-Anwendungen, die auf 64-Bit-Systemen in 32-Bit-Maschinencode übersetzt werden und im WOW64-Emulator laufen, nicht schneller sind als auf 32-Bit-Systemen. Daraus folgt, dass man die Kraft der 64 Bit nur nutzen kann, wenn der Entwickler seine Anwendung so schreibt, dass sie auf 64-Bit-Systemen auch im 64-Bit-Just-in-Time-Compiler landet. *32-Bit-REQUIRED* ist also zu vermeiden.

Die Wahl zwischen PE32 und PE32+ spielt für die Leistung keine Rolle, wohl aber für die Kompatibilität. Assemblys im PE32+-Format laufen jeweils nur auf einem speziellen Prozessortyp.

Versionskompatibilität

Im Zeitalter vor .NET litten Entwickler unter Versionsproblemen: Durch Installation einer neuen Version einer DLL konnten bestehende Anwendungen gestört werden. Im Entwicklerjargon sprach man von der *DLL-Hölle*. Die neue Hölle, die Microsoft Schritt für Schritt in .NET einführt, ist die *Framework-Versionshölle*.

Hier gibt es zwei Problembereiche: Einerseits die *Entkopplung* der Versionsnummern von .NET-Bausteinen und andererseits die Inkompatibilitäten zwischen den Versionen.

Entkopplung der Versionsnummern

In .NET 1.0, 1.1 und 2.0 war die Versionszählung noch halbwegs einfach: Version des .NET Framework, der Laufzeitumgebung Common Language Runtime (CLR), der Kernbibliotheken (z.B. ADO.NET, ASP.NET, Windows Forms) sowie alle zugehörigen Softwarekomponentendateien (alias Assemblys) trugen die gleiche Versionsnummer, also z.B. .NET 2.0 enthielt CLR 2.0, ASP.NET 2.0, ADO.NET 2.0 und Windows Forms 2.0. Einzige Ausnahme waren die Sprachcompiler: Visual Basic zählt bei der letzten COM-basierten Version weiter (7.0, 7.1, 8.0) und der C#-Compiler hatte die gleiche Versionsnummer. C# 1.0 war also eigentlich C# 7.0.

Ab .NET 3.0 wurde dann aber alles anders, denn .NET Framework 3.0 ist eine echte Obermenge über .NET 2.0. Die bestehenden Bausteine blieben unangetastet, sodass bei der Installation von .NET 3.0 nun die »alten« Bausteine den Stand 2.0 haben (z.B. ADO.NET und ASP.NET) und die neuen (WPF, WCF, WF) die Version 3.0.

Noch etwas anders war es mit Version 3.5: Auch .NET 3.5 versteht sich als eine Obermenge über .NET 2.0 und .NET 3.0. Hier aber wurde ASP.NET von 2.0 auf 3.5 hochgezählt. Die zugehörigen Assemblys tragen aber zum Teil die Version 2.0 (weil es diese schon vorher gab) und zum Teil 3.5 (weil diese hinzugekommen sind).

Mit .NET 4.0 installiert sich dann wieder ein »komplettes« .NET Framework.

HINWEIS

Auf der Ebene der Assemblys spricht man von *Red Bits* und *Green Bits*. Red Bits sind aus einer Vorgängerversion übernommene Dateien, die aber (leicht) verändert werden. Es wurden Fehler behoben, aber auch neue Funktionen (Klassen und Klassenmitglieder) ergänzt. Green Bits sind neu hinzu gekommene Assemblys, die es vorher nicht gab.

ASP.NET ist auch ein gutes Beispiel dafür, dass die Versionsnummer nun nichts mehr über die Funktion aussagt. ASP.NET 2.0 hatte gegenüber ASP.NET 1.1 viel, viel mehr Änderungen als ASP.NET 3.5 gegenüber ASP.NET 2.0, obwohl ja die Versionsnummerndifferenz etwas anderes vermuten ließe.

Inkompatibilitäten zwischen den Versionen

Die oberen bereits benannten Red Bits können in zwei Richtungen zu Problemen führen. Durch das Verändern von Funktionen kann es dazu kommen, dass Anwendungen nicht mehr (einwandfrei) laufen. Hier spricht man von *Breaking Changes*. Microsoft betont zwar immer wieder, Funktionen nur zu verändern, wenn diese fehlerhaft waren. Aber manchmal nutzen Entwickler (unbewusst) solche Funktionen aus.

Hier seien nur vier Beispiele für Breaking Changes zwischen .NET 1.1 und .NET 2.0 exemplarisch genannt:

- In einer abgeleiteten C#-Klasse kann man einen Konstruktor der Oberklasse, der als *protected* deklariert ist, nur noch in einem Konstruktor aufrufen. Es ist nicht mehr möglich, eine Instanz der Oberklasse damit zu erzeugen.
- Die Sprachcompiler erzeugen eine Fehlermeldung, wenn mehrere Assemblys mit dem gleichen Typnamen (inkl. Namensraum) referenziert werden. Bisher wurde zufällig einer der Typen ausgewählt
- Die Methode `DateTime.Parse()` ist strenger und reagiert auf fehlerhafte Datumsangaben eher mit einem Laufzeitfehler
- Anwendungen stürzen nun ab, wenn in einem anderen Thread als dem Hauptthread ein unbehandelter Fehler auftritt (in der CLR 1.1 konnte die Anwendung weiterlaufen)

Ebenfalls Inkompatibilitäten gibt es natürlich, wenn in einer höheren .NET-Versionsnummer Funktionen ergänzt werden. Wenn ein Programm diese Funktionen aufruft, dann ist es nicht mehr möglich, das Programm mit einer niedrigeren .NET-Versionsnummer zu betreiben. Das ist soweit einleuchtend.

ACHTUNG Mit dem Service Pack 1 für .NET 2.0 und .NET 3.0 (die parallel zu .NET 3.5 erschienen sind und in .NET 3.5 auch vorausgesetzt werden), hat Microsoft nun auch bei einem Service Pack Funktionen ergänzt. Ein Programm, das mit .NET 2.0 Service Pack 1 kompiliert wurde, läuft also nicht unbedingt auf einem System mit .NET 2.0 ohne Service Pack 1. Das Gemeine in der konkreten Situation ist, dass Microsoft dies nicht klar kommuniziert. Beispiel: Die Einstellung *.NET Framework 2.0* in Visual Studio 2010 ist in Wirklichkeit *.NET Framework 2.0 Service Pack 1*. Das bedeutet: Eine damit kompilierte Anwendung läuft vielleicht gar nicht auf einem System ohne das Service Pack. Leider warnt eine .NET-Anwendung dann beim Start nicht, dass ihr das Service Pack fehlt. Vielmehr stürzt die Anwendung irgendwann an dem Punkt ab, wo das erste Mal eine Methode zum Just-in-Time-Compiler gelangt, die eine Klasse oder Methode verwendet, die in dem Service Pack ergänzt wurde. Das kann überall während des Programmablaufs sein. Sehr tückisch!

```

C:\WINDOWS\System32\cmd.exe
C:\Documents and Settings\Administrator>\\192.168.1.100\h$\temp\Demo_Multitargeting\Demo_Multitargeting20d.exe
Anwendung startet...
CLR-Version: 2.0.50727.26
Test Datum/Zeit-Funktionen...
Unhandled Exception: System.TypeLoadException: Could not load type 'System.DateTimeOffset' from assembly 'mscorlib, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089'.
   at Demo_Multitargeting.Program.DatumZeit()
   at Demo_Multitargeting.Program.Main(String[] args)
C:\Documents and Settings\Administrator>
  
```

Abbildung 4.24 Absturz beim Einstieg in eine Methode, die die in Service Pack 1 hinzugefügte Klasse `DateTimeOffset` verwendet

Leider ist die Dokumentation dieser *Red Bits* in der Visual Studio-Hilfe nur bei den jeweiligen Klassen und Klassenmitgliedern in der Sektion *Version Information* zu finden, wenn dort nicht *.NET 2.0*, sondern nur *.NET 2.0 SP1* erscheint.

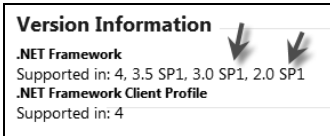


Abbildung 4.25 Beispiel für die Dokumentation einer Klasse, die es in .NET 2.0 und 3.0 nur mit Service Pack 1 gibt

Eine zusammenhängende Liste findet man nur in einem Blog: [HAN01]

Parallelbetrieb

Verschiedene .NET Framework-Versionen können auf einem System parallel betrieben werden. Es kann aber nicht gleichzeitig eine Version mit und ohne Service Pack parallel betrieben werden.

Welche .NET Framework-Versionen installiert sind, erkennt man im Verzeichnis */Windows/Microsoft/.Net Framework*. Dort gibt es ein Unterverzeichnis für jede Version. Man darf sich allerdings nicht täuschen lassen: Wenn eine ältere Version nur sehr wenige Dateien (insbesondere .config-Dateien) enthält, ist diese gar nicht installiert. Offenbar legt .NET 2.0 einige Dateien in den Verzeichnissen *v1.0* und *v1.1* ab. Tatsächlich installiert sind 1.0 und 1.1 nur, wenn die Verzeichnisse zahlreiche Dateien und Unterverzeichnisse besitzen.

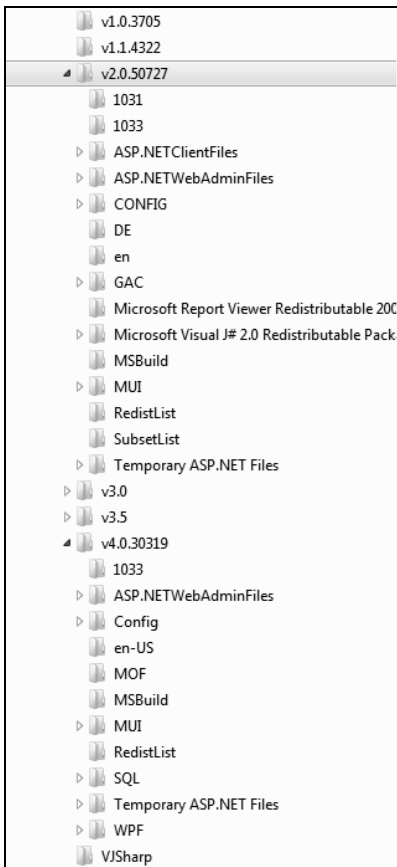


Abbildung 4.26 Auf diesem Windows Server 7 sind .NET 2.0 bis .NET 4.0 installiert

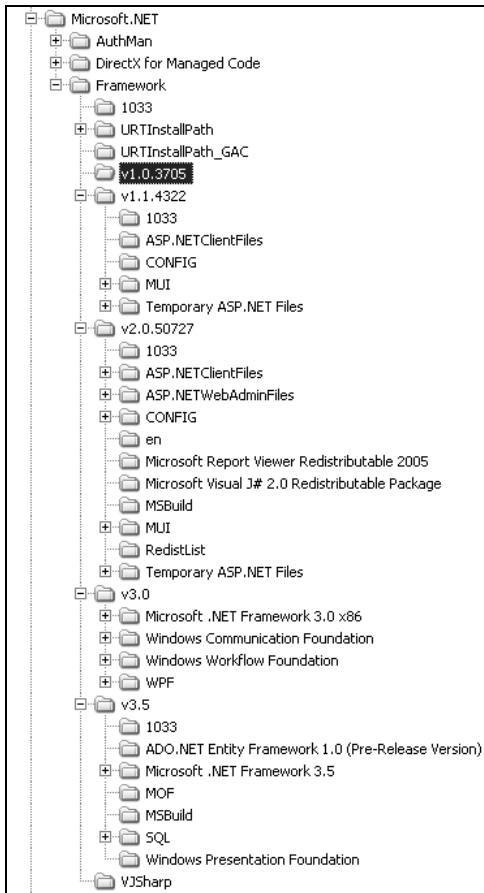


Abbildung 4.27 Auf diesem Windows Server 2008 sind .NET 1.1 bis 3.5 und diverse Zusatzkomponenten installiert

Versionsumstieg

Es gibt drei Möglichkeiten für eine .NET-Anwendung/-Komponente im Zeitalter einer neuen Version:

1. Keine Neukompilierung, Weiterbetreiben mit der alten CLR
2. Keine Neukompilierung, Betreiben mit der »neuen« CLR
3. Migrieren auf die neue CLR-Version durch Neukompilierung

TIPP

Es ist am sichersten, jede .NET-Anwendung unter der CLR-Version zu betreiben, mit der sie kompiliert wurde. Dies ist die automatische Voreinstellung in den meisten (aber nicht in allen) .NET Runtime Hosts.

Weiterbetreiben mit der alten CLR

Das Grundmodell ist der Fall 1, da jede .NET-Anwendung mit der CLR-Version startet, mit der sie betrieben wurde, sofern die Version vorhanden ist. Dafür sorgen die Ablaufumgebungen (.NET Runtime Hosts). Diese Funktion nennt Microsoft *Side-by-Side Execution (SxS)*.

Von diesem Grundmodell gibt es aber vier Ausnahmen:

1. Eine .NET-Assembly wird mit einer neueren CLR-Version gestartet, wenn die ältere CLR nicht vorhanden ist.
2. Eine referenzierte .NET-Assembly wird mit der neueren CLR ausgeführt, wenn die aufrufende Assembly mit der neueren CLR läuft.
3. Einige .NET-Ablaufumgebungen (z.B. Internet Explorer, Office, SharePoint oder BizTalk) haben in der Vergangenheit immer die aktuellste vorhandene CLR-Version geladen. Dies konnte zu Problemen führen. Ab .NET 4.0 wird der Benutzer dann aufgefordert, die ältere CLR-Version zu installieren. Diese hostenden Anwendungen kann man nun auch dazu bringen, mehrere CLR-Versionen parallel in einem Prozess zu laden (*In-Process Side-by-Side* - *Inproc SxS*)
4. Der IIS lädt die CLR gemäß Konfiguration in der Metabase des Internet Information Servers.

WICHTIG

In dem obigen Text wurde bewusst von der CLR-Version und nicht von der .NET-Version gesprochen, denn die CLR-Version und die .NET-Version waren in .NET 3.0 und 3.5 entkoppelt. .NET 1.0 enthielt die CLR 1.0, .NET 1.1 die CLR 1.1. Die CLR 2.0 ist aber sowohl in .NET 2.0 als auch in .NET 3.0 und .NET 3.5 enthalten. .NET 4.0 enthält die CLR 4.0.

TIPP

Durch eine Einstellung in der Anwendungskonfigurationsdatei kann man das Laden der korrekten CLR-Version erzwingen:

```
<configuration>
  <startup>
    <supportedRuntime version="v1.1.4322" />
  </startup>
</configuration>
```

Betreiben mit einer neuen CLR

Ob eine .NET-Anwendung unter einer anderen .NET-Version betrieben werden kann, hängt von drei Faktoren ab:

- Kompatibilität der CLR
- Kompatibilität der Microsoft Intermediate Language (MSIL)
- Verfügbarkeit und Verhalten der Klassen

In allen drei Punkten gilt: Die Aufwärtskompatibilität ist (mit einigen Einschränkungen) gegeben, weil gegenüber den früheren Versionen nur ergänzt, aber nichts gestrichen wurde. Eine Abwärtskompatibilität gibt es hingegen nicht, d. h., eine Anwendung für die CLR 2.0 kann niemals unter der CLR 1.1 betrieben werden. Einzig möglich sein könnte der Betrieb einer .NET 3.x-Anwendung unter .NET 2.0, weil die CLR-Version gleich geblieben ist. Voraussetzung ist aber, dass keine der in .NET 3.0 / 3.5 neu eingeführten Klassen verwendet wurde. Eine .NET 4.0-Anwendung kann nicht in der CLR 1.x oder CLR 2.0 (also .NET 2.0 bis 3.5) laufen.

Die Einschränkungen bei der Aufwärtskompatibilität betreffen die .NET-Klassenbibliothek. Jede neue Version der Klassenbibliothek enthält gegenüber der vorhergehenden Version der Klassenbibliothek (umfangreiche) Erweiterungen. Diese betreffen sowohl Steuerelemente als auch nicht-visuelle .NET-Klassen.

Die Aufwärtskompatibilität ist gegeben, sofern keine Klassen entfernt wurden. In den bisher erschienenen .NET-Versionen sind keine Klassen entfernt, sondern nur Klassen als *Obsolete* (Annotation `System.Obsolete`) gekennzeichnet worden.

HINWEIS Wenn der Compiler auf eine Annotation `System.Obsolete` trifft, warnt er. Die Verwendung einer obsoleten Klasse bzw. eines obsoleten Klassenmitglieds ist aber kein Kompilierungsfehler (sofern man die Warnung nicht durch Compiler-Einstellungen zu einem Fehler erhebt). Obsolete Klassen sind in der Dokumentation besonders hervorgehoben.

Soweit die Theorie. Leider gibt es im Detail doch einige Änderungen zwischen den Versionen (siehe oben), die das korrekte Funktionieren einer alten Anwendung unter dem jeweils neuen .NET Framework verhindern könnten (*Breaking Changes*). Die Existenz von Breaking Changes bedeutet, dass eine Anwendung oder Komponente, die unter höheren .NET-Versionen betrieben werden soll, sehr umfassend getestet werden muss (siehe auch Tabelle).

Ursprung Ziel	.NET 1.0	.NET 1.1	.NET 2.0	.NET 3.0	.NET 3.5	.NET 4.0
.NET 1.0	Ja	wahrscheinlich	Nein	Nein	Nein	Nein
.NET 1.1	Ja	Ja	Nein	Nein	Nein	Nein
.NET 2.0	wahrscheinlich	wahrscheinlich	Ja	Ja, wenn keine in .NET 3.0 neu eingeführten Klassen verwen- det wurden	Ja, wenn keine in .NET 3.0 neu eingeführten Klassen verwen- det wurden	Nein
.NET 3.0	wahrscheinlich	wahrscheinlich	Ja	Ja	Ja, wenn keine in .NET 3.0 neu eingeführten Klassen verwen- det wurden	Nein
.NET 3.5	wahrscheinlich	wahrscheinlich	Ja	Ja	Ja	Nein
.NET 4.0	Nein	Nein	Nein	Nein	Nein	Ja

Tabelle 4.9 Kompatibilität der .NET-Versionen

TIPP Zu Testzwecken kann man das Standardverhalten der .NET Runtime Hosts deaktivieren und alle .NET-Assemblys dazu bringen, mit der aktuellsten vorhandenen CLR-Version zu starten. Dies erreicht man über eine Einstellung in der Registrierungsdatenbank:

```
[HKEY_LOCAL_COMPUTER\SOFTWARE\Microsoft\.NETFramework]
"OnlyUseLatestCLR"=dword:00000001
```

Berücksichtigen muss man, dass auch die älteren .NET-Compiler der .NET-Version in dieser Einstellung nicht mehr arbeiten können. Diese Einstellung kann man rückgängig machen durch:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework]
"OnlyUseLatestCLR"=dword:00000000
```

Neukompilierung

Die Neukompilierung einer .NET-Anwendung für eine neue .NET-Version ist grundsätzlich möglich. Fehler können allerdings beim Kompilieren bzw. Änderungen im Verhalten zur Laufzeit auftreten aufgrund der oben erwähnten *Breaking Changes*. Dies ist relevant für die Neukompilierung von CLR 1.x-Quellcode mit den Sprach-Compilern von .NET 2.0 oder 3.5, bzw. .NET 2.0/3.0 mit den Compilern aus .NET 3.5 sowie .NET 1.x/.NET 2.0/.NET 3.x mit den 4.0-Compilern.

Der Migrationsaufwand hält sich für Windows- und Konsolenanwendungen in Grenzen. Für Webanwendungen ist der Aufwand jedoch höher, denn hier gab es entscheidende Änderungen im Programmiermodell und zahlreiche Neuerungen in der Projektverwaltung innerhalb der Entwicklungsumgebung.

ACHTUNG

Nach der Neukompilierung sind auf jeden Fall sehr ausführliche Tests notwendig!

Ermitteln der CLR-Version

Sowohl im Test als auch im Betrieb kann es wichtig sein, die CLR-Version zu ermitteln. Hier muss man aber differenzieren zwischen der CLR-Version, für die eine Assembly kompiliert wurde, und der CLR-Version, mit der die Assembly tatsächlich ausgeführt wird. Die CLR-Version, für die kompiliert wurde, liefert die Klasse `Assembly` in der Eigenschaft `ImageRuntimeVersion` als eine Zeichenkette:

```
System.Reflection.Assembly.GetExecutingAssembly().ImageRuntimeVersion
```

Die tatsächlich verwendete CLR-Version, mit der die Assembly aktuell betrieben wird, findet man hingegen in `System.Environment`. Dort liefert `Version` eine Instanz der Klasse `System.Version`:

```
Environment.Version.ToString()
```

WICHTIG

Beide o.g. Befehle liefern unter .NET 3.0 und .NET 3.5 immer *.NET 2.0*, da .NET 3.0 und 3.5 nur Erweiterungen der Klassenbibliothek sind. Eine .NET 3.0- oder 3.5-Anwendung zeichnet sich nur dadurch aus, dass sie Bibliotheken verwendet, die es in .NET 2.0 noch nicht gab. Nirgendwo in dem Assembly-Manifest steht explizit die Versionsnummer des .NET Framework.

Mit einem einfachen Test kann man auch prüfen, welche CLR verwendet wird, wenn man die Verhaltensänderung bei einem der *Breaking Changes* ausnutzt. In dem nachfolgenden Listing wird eine Verhaltensänderung von `System.DateTime.Parse()` verwendet. `Parse()` ist in .NET 2.0 strenger geworden. `DateTime.Parse(@"01-01-10/10/2001")` liefert in .NET 1.1 die Ausgabe `1.10.2001 00:00:00`. In .NET 2.0 kommt es zu einem Laufzeitfehler (`System.FormatException`).

```
try
{ DateTime.Parse(@"01-01-10/10/2001");
ausgabe = ausgabe + ".NET 2.0-Test: fehlgeschlagen" + Umbruch;}
catch
{ ausgabe = ausgabe + ".NET 2.0-Test: OK" + Umbruch;}
```

Eine kleine Konsolenanwendung zum Testen von Migrationsszenarien finden Sie in den Downloads zu diesem Buch [*SonstigeBeispiele\NET_KomponentenDemos\08_Versionstestprogramm*]. Das Programm gibt aus, mit welcher CLR-Version es kompiliert wurde und unter welcher CLR-Version es läuft.

In der folgenden Bildschirmabbildung dargestellten Fall lädt eine .NET 4.0-Hauptanwendung eine .NET 2.0-Komponente. Beide laufen unter .NET 4.0.

```
Administrator: Visual Studio Command Prompt (2010)
m\4.0\Client.exe
.NET 1.1/2.0-Versionstest (C) Dr. Holger Schwichtenberg www.IT-Visions.de 2006

Information aus der referenzierten Komponente:
Hello World aus der .NET-Komponente!
Version der Komponente: 0.0.0.0
Kompilierte CLR-Version: v2.0.50727
Tatsächliche CLR-Version: 4.0.30319.1
Assembly Location: H:\TFS\Demo\NET_SonstigeBeispiele\NET_KomponentenDemos\08_Versionstestprogramm\4.0\ITU_InfoKomponente.dll
Aus GAC geladen? False
.NET 2.0-Test: OK

Information aus der Hauptkomponente:
Hello World aus der Hauptkomponente!
Version der Komponente: 4.0.0.0
Kompilierte CLR-Version: v4.0.30319
Tatsächliche CLR-Version: 4.0.30319.1
Assembly Location: H:\TFS\Demo\NET_SonstigeBeispiele\NET_KomponentenDemos\08_Versionstestprogramm\4.0\Client.exe
Aus GAC geladen? False
h:\TFS\Demo\NET_SonstigeBeispiele\NET_KomponentenDemos\08_Versionstestprogramm\4.0>
```

Abbildung 4.28 Testanwendung zur Ermittlung der .NET-Versionen

Visual Studio-Kompatibilität

Microsofts Entwicklungsumgebung Visual Studio war bisher versionsbezogen: Mit Visual Studio .NET 2003 kann man nur 1.1-Anwendungen erstellen; mit Visual Studio 2005 nur .NET 2.0- und .NET 3.0-Anwendungen. Mit Visual Studio 2008 kann man erstmals für mehrere Versionen entwickeln: .NET 2.0, .NET 3.0 und .NET 3.5. Mit Visual Studio 2010 kann man von .NET 2.0 bis .NET 4.0 entwickeln.

WICHTIG Exakt gesprochen (was Microsoft leider nicht tut): Mit Visual Studio 2008 und 2010 entwickelt man Anwendungen für .NET 3.5 und .NET 3.0 Service Pack 1 und .NET 2.0 Service Pack 1. Diese Service Pack-Problematik wurde weiter oben schon erläutert.

Ältere Visual Studio-Projekte müssen stets beim Öffnen in Visual Studio 2010 konvertiert werden, weil sich das Projektformat geändert hat. Projekte können danach nicht mehr in der alten Version geöffnet werden. Man sollte also unbedingt eine Sicherungskopie der alten Projektdaten anlegen. Visual Studio 2010 bietet zur Konvertierung der Projekte einen Konvertierungsassistenten (Visual Studio Conversion Wizard). Dieser ändert das Format der Projektdateien. Ob nur die Projektdatei geändert wird oder auch ein Eingriff in den Code erfolgt, hängt von der Anwendungsart und der früheren Version ab. In Visual Studio 2010 ändert der Migrationsassistent zum Beispiel bei Webprojekten die *web.config*-Dateien.