1 Das .NET Framework und Visual Studio

Eine erste Definition von .NET

Einen Computer mit einem aufwändigen Betriebssystem wie Windows auszustatten, ist eine tolle, aber im Grunde genommen eine sinnlose Sache. Erst die konkreten Anwendungen (die benutzerorientierten Programme) machen ein Computersystem zu mehr als einem Haufen mehr oder weniger intelligenter Elektronik. Das Betriebssystem Windows selber besteht aus vielen Anwendungen, die wir tagtäglich benutzen. Man kann sogar sagen, dass alles, was wir vom Betriebssystem sehen, Anwendungen sind. Das beginnt beim Windows-Explorer, mit dem wir die Daten auf dem Rechner suchen und öffnen können, geht über den Internet Explorer, mit dem wir im Internet einen Begriff nachschlagen, bis hin zum branchenspezifischen Programm, das wir im Rahmen unserer Arbeit ausführen.

Alle diese Anwendungen wurden von Softwareentwicklern zur Erleichterung der täglichen Arbeit eines Benutzers oder einer Benutzerin entwickelt. Natürlich gibt es auch Ausnahmen! Ich kämpfe auch immer wieder mit Software, die einem das Leben schwer macht. Besonders frustrierend ist es, wenn diese Software von mir selbst geschrieben ist! Nun, der Softwareentwickler spezialisiert sich darauf, ein Programm für einen bestimmten Zweck zu erstellen. Er analysiert dazu die Problemstellung, entwirft eine Lösung und setzt diese Lösung letztendlich in Form von Code in einer Programmiersprache um.

Die Softwareentwickler nutzen dabei nach Möglichkeit bereits vorhandene Funktionen, die sie in einer eigenen Logik zusammensetzen. Man will damit erreichen, dass die Codierung möglichst effizient umgesetzt werden kann. Das ist vergleichbar mit dem Herstellen eines Essens. Dort nehme ich nach Bedarf oder Möglichkeit fertige und rohe Zutaten. Ich werde dabei beeinflusst von meinen persönlichen Philosophien, dem Zeitfaktor und anderen Elementen. Genauso verhalte ich mich beim Erstellen von Software: Mal schreibe ich aus einem bestimmten Grund eine Funktion selber, mal benutze ich die fertige Funktion.

Woher stammen die gebrauchsfertigen Funktionen? Sie sind im Betriebssystem und weiteren eingekauften und installierten Komponenten auf den Computersystemen vorhanden. Das Betriebssystem stellt uns eine reichhaltige Fülle von solchen Funktionen für die verschiedenen Anwendungsfälle zur Verfügung. Die Ebene der bei der Softwareentwicklung benutzbaren Funktionen wird allgemein API genannt (englisch: Application Programming Interface, deutsch: Anwendungsprogrammierschnittstelle). Versuchen wir die Verhältnisse zwischen Hardware und Anwendung grafisch darzustellen, ergibt sich ein Schichtenmodell wie in **Abbildung 1.1**.

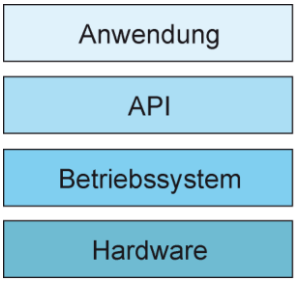

Abbildung 1.1: *Schichtenmodell einer herkömmlichen Anwendungsarchitektur*

Auch Windows hat seit jeher eine solche Schnittstelle. Die API hat einen speziellen Namen, der ungefähr zur Zeit der Entwicklung von Windows NT 3.1 geprägt wurde: WIN16-API (16 Bit Schnittstelle) oder WIN32-API (32-Bit Schnittstelle).

Geschichtlich bedingt basieren die Schnittstellen der großen Betriebssysteme wie Windows oder Unix auf den Programmiersprachen C oder C++. Beide Sprachen gehören im Betrachtungshorizont der Informatik nicht gerade zu den jüngsten Erkenntnissen und wurden in den letzten Jahren nicht der erhofften Weiterentwicklung unterworfen.

Ebenso wie die beiden genannten Sprachen sind auch meistens die Schnittstellen der Betriebssysteme ein bisschen in die Jahre gekommen. Das macht sich durchaus nicht in Sachen Geschwindigkeit oder mangelnder Funktionalität bemerkbar. Vielmehr ist es die nicht konsequent gleichartige Nutzung der verschiedenen Teile der API. Dies wiederum basiert auf dem Umstand der Jahrzehnte langen Entwicklung und der dahinter wechselnden Gesichter mit unterschiedlichen Philosophien.

Kurzum: Wenn ich moderne Software entwickeln will, muss ich mich mit zum Teil mehrere Jahrzenten alten Techniken und Praktiken auseinandersetzen und gelange so immer wieder an die Grenzen der Möglichkeiten oder muss diese mit viel Aufwand umschiffen.

Hier kommt nun das .NET Framework zum Zug. Das Framework definiert folgende Zielsetzungen:

- Abstraktion des Programmiermodells auf Stufe des Betriebssystems
- Unabhängigkeit gegenüber dem darunterliegenden Betriebssystem erreichen, indem eine neuartige Laufzeitumgebung geschaffen wird
- Ermöglichen, dass neuere Sprachen als C oder C++ verwendet werden können
- Vereinheitlichung des Programmiermodells für verschiedenste Teilgebiete der Informationsverarbeitung

.NET selber braucht also ein Betriebssystem, auf dem es installiert werden und ablaufen kann. Die Anwendung selbst wird aber für .NET entwickelt und braucht dann für ein reibungsloses Funktionieren .NET.

Hinweis Beginn

Selbstverständlich ist es für eine Anwendung auf .NET technisch auch weiterhin möglich, durch .NET hindurch direkt auf das Betriebssystem zuzugreifen. Man muss dabei allerdings beachten, dass man Gefahr läuft, den gewonnenen Vorzug einer gewisse Unabhängigkeit vom Betriebssystem zu verlieren.

Hinweis Ende

Vereinfacht dargestellt ergibt sich aus dem Gesagten die **Abbildung 1.2**:

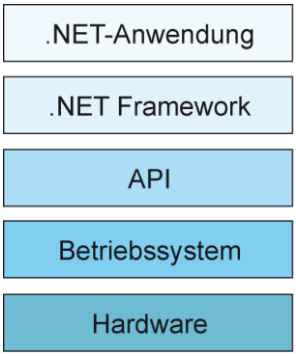

Abbildung 1.2: *Schichtenmodell einer .NET-Anwendungsarchitektur*

Der Rest des Buches dient nun also dazu, den Umgang mit dem Inhalt des Rechtecks ».NET Framework« gemäß **Abbildung 1.2** zu erklären! Dabei gibt es durchaus verschiedene Betrachtungen, die man zu diesem Rechteck vornehmen kann. Zum einen ist da die Untersuchung in Bezug auf denBetrieb von Rechnersystemen. Hierbei steht im Vordergrund, zu verstehen, wie die Schnittstelle zum Betriebssystem und die Parametrisierung einer Anwendung für .NET geschieht oder unterhalten wird. Zum andern ist es der Aufbau einer .NET-Anwendung selbst, der uns interessiert.

Als Softi (Kurzform für Softwareentwickler oder Softwareentwicklerin) werden Sie primär am zweiten Thema interessiert sein. Das ist selbstverständlich richtig, und der größte Teil meines Handbuchs der .NET-Programmierung adressiert auch dieses Thema. Allerdings ist es kaum möglich, eine Anwendung korrekt in ein System zu integrieren, wenn das .NET Framework nicht korrekt installiert und seine Funktionsweise nicht korrekt konfiguriert ist. Entsprechend werden im ersten Kapitel vor allem die betrieblichen Aspekte einer Anwendung respektive deren Zusammenspiel mit dem .NET Framework untersucht.

Seit Windows XP wird .NET mit dem Betriebssystem ausgeliefert und standardmäßig installiert. Bereits hier beginnt aber die Sache etwas komplizierter zu werden, denn die Entwicklung und die Freigabe von .NET-Versionen ist nicht zwingend an Freigabe von Windows-Versionen gekoppelt. Das heißt, bei unterschiedlichen Versionen von Windows sind unterschiedliche Versionen des .NET Framework vorhanden. Dabei kommen in der Praxis alle Mischformen der Versionen vor, die auch technisch denkbar sind, respektive von Microsoft vorgesehen sind. Um Ihnen eine Übersicht der verschiedenen Versionen zu zeigen habe ich in der **Abbildung 1.3** eine Übersicht erarbeitet.

Der zweite sehr wichtige Punkt in Bezug auf die verschiedenen .NET-Versionen ist die Funktionalität der Versionen. Wenn Sie eine Anwendung entwickeln, werden Sie in der Regel davon ausgehen, dass Sie die aktuelle Version von .NET verwenden können. Aber Vorsicht! Sprechen Sie sich unbedingt mit Ihrem Kunden ab, welche Version von Windows und dem .NET Framework auf den Rechnern des Kunden installiert ist. Unter Umständen hat der Kunde eine etwas ältere Version des Systems im Einsatz und kann nicht rasch eine Migration durchführen. Gerade in großen Betrieben ist heutzutage ein Wechsel des Betriebssystems oder das Upgrade einer .NET-Version nicht einfach so möglich und braucht eine lange Planung und viele Tests, ob alle benutzten Programme immer noch lauffähig sind. Zwar ist bei .NET die Versionsempfindlichkeit nicht so groß wie bei herkömmlichen Anwendungen, aber es gibt auch hier genügend Fallen, in die man hineintappen kann, um eine Anwendung zum Nicht-Funktionieren zu bringen.

Hinweis Beginn

Im Abschnitt »Visual Studio 2010«, werde ich Ihnen zeigen, wie Sie Ihre Projekte so konfigurieren können, dass Sie mit der .NET-Version der Kunden garantiert kompatibel ist.

Hinweis Ende

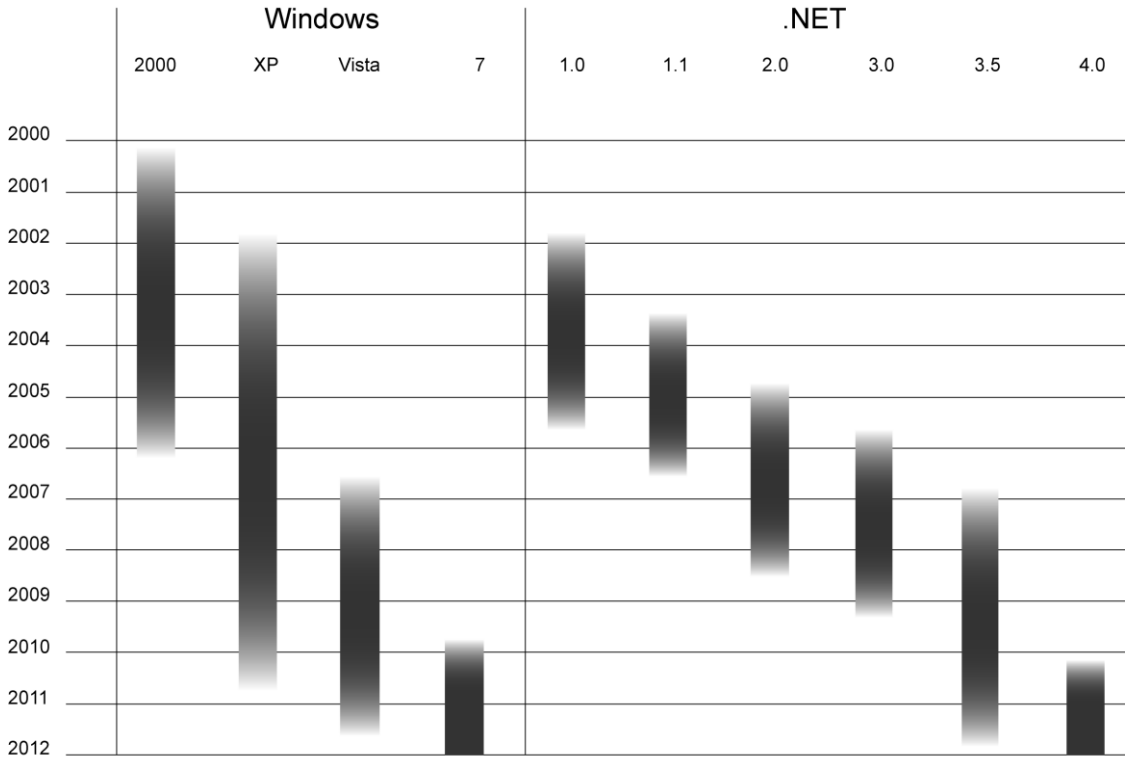

Abbildung 1.3: *Versionen des Betriebssystems Windows und des .NET Frameworks im Laufe der Zeit*

Wie bereits erklärt, wird das Betriebssystem bei der Installation bereits mit .NET ergänzt. Die dabei installierte Version ist jeweils abhängig von der Version des installierten Betriebssystems. Windows 7 zum Beispiel wird mit .NET 3.5 und integriertem SP1 ausgeliefert (SP1: Service Pack 1).

Für unsere Untersuchungen installieren wir also zunächst .NET 4.0 auf unserem Rechner. Als Softi werden Sie das in der Regel zusammen mit *Visual Studio* vornehmen. Wenn dem nicht so ist, werden Sie die sogenannte .NET Runtime direkt installieren.

Hinweis Beginn

Die .NET Runtime (allgemein auch als .NET Redestributable bekannt) existiert in verschiedenen Versionen und in verschiedenen Typen. Der Name des funktionell vollständig ausgestatteten Frameworks heißt schlicht *.NET Framework* und ist in folgenden Versionen bekannt:

- Version 1.0: Die Basisversion ist im Februar 2002 auf den Markt gekommen
- Version 1.1: Das Stabilitäts- und Verbesserungspaket für 1.0 ist im April 2003 auf den Markt gekommen
- Version 2.0: Im November 2005 werden generische Typen eingeführt und die Funktionalität der Klassenbibliothek wird massiv erweitert. Diese Version war Grundlage der 1. Auflage des Handbuch der .NET-Programmierung.
- Version 3.0: Die Version erweitert .NET im November 2006 mit Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF) und Workflow Foundation (WF). Der Support der Erweiterungen durch Visual Studio ist nicht optimal, da keine neue Version von Visual Studio auf den Markt kommt.
- Version 3.5: Die Version wird mit Visual Studio 2008 im November 2007 verfügbar. .NET ist nun

erwachsen und stellt viele schlagkräftige Neuerungen wie LINQ (Language Integrated Query) zur Verfügung.

- Version 3.5 SP1: Normalerweise werden Service Packs für Korrekturen und kleinste Erweiterungen benutzt. Nicht so bei .NET 3.5 SP1 im August 2008. Dieses Paket führt neben Korrekturen das Entity Framework ein.
- Version 4.0: Das aktuell letzte Kapitel der .NET-Geschichte bricht im 2.Quartal 2010 alle Rekorde der Funktionalität. Neue Möglichkeiten der Sprache C# werden von einer Vielzahl von Erweiterungen bestehender Klassen und der Einführung neuer Klassen begleitet. Das große Highlight ist aber die systematische Unterstützung der Verarbeitung paralleler Aufgaben.

Neben dem Typ Vollversion des .NET Frameworks existieren zusätzliche Typen des Frameworks, die allesamt verschiedene Untermengen des Gesamtsystems zur Verfügung stellen:

- .NET Framework Client Profile: Es existiert seit .NET 3.5 und stellt nur die Teile des .NET Frameworks zur Verfügung, die typischerweise auf einem Client verwendet werden (Es fehlen die Serverbestandteile von .NET). Damit wird dieses Framework deutlich schlanker und braucht weniger Platz im System.
- .NET Compact Framework:.NET für Windows CE basierte Systeme
- .NET Micro Framework: Es ist eine extrem verkleinerte Version von .NET, das für kleinste Systeme ausgelegt ist. Es ist gemäß Microsoft sogar in der Lage, ohne darunter liegendes Betriebssystem auszukommen.
- .NET XNA Framework: Es ist eine spezielle Version des .NET Compact Frameworks für die Entwicklung von Anwendungen (vornehmlich Spielen) für die Xbox 360

Hinweis Ende

Welche Alternativen zu .NET existieren?

Da Windows eine enorme Verbreitung auf dem Weltmarkt aufweist (> 90% der Rechner), ist es nicht verwunderlich, dass heutzutage sehr viele Entwicklungen auf der Basis von .NET geschehen. Neben .NET sind jedoch klar zwei weitere Gruppen von Entwicklungen zu nennen, die zum Teil mehr als nur eine Daseinsberechtigung aufweisen.

Entwickeln mit Java

Die Architektur von .NET gleicht derjenigen von Java sehr. Wenn man in Betracht zieht, dass Java vor .NET auf dem Markt etabliert war, könnte man in Versuchung kommen zu glauben, dass die Entwicklung von .NET von Java beeinflusst war.

Beide Systeme bedienen sich einer virtuellen Maschine und bieten grob gesehen sowohl technologisch als auch funktionell die gleichen Möglichkeiten an. Die Hauptunterschiede der beiden Systeme sind wie folgt:

- Java ist auf allen gängigen Plattformen verfügbar. .NET installiert sich im Wesentlichen auf der Windows-Plattform. Mit dem Projekt Mono existiert für .NET eine allgemeine Anwendung auf der Basis von Unix. Diese hat allerdings den Nachteil, dass Sie noch wenig verbreitet ist und vor allem deren Codebasis eine Mischung aus verschiedenen Versionen des effektiven Frameworks darstellt, ohne die neuste Version vollumfänglich zu unterstützen.
- Bei Java besteht alles aus Java. Ich meine damit, dass das System den Namen Java trägt und die Programmierungen mit der Sprache Java durchgeführt werden. Das fördert die Einheitlichkeit des Systems, hat aber den Nachteil, dass die Wahl der Programmiersprache nicht frei ist. Bei .NET ist

das nicht so. Hier heißt das System .NET und als Programmiersprachen kommen so ziemlich alle Sprachen zur Anwendung. Die hauptsächlichen von Microsoft unterstützten Sprachen sind C# (sprich ßie Sharp), Visual Basic und C++.

Native Entwicklungen

Neben der Softwareentwicklung für .NET und Java ist der Bereich der nativen Entwicklung in Hochsprachen immer noch ein wichtiger Markt. Dabei kommen immer noch Sprachen wie C, C++ aber auch COBOL und andere Sprachen in großen Mengen vor. Es wäre meiner Meinung nach falsch und für viele unserer Berufskollegen und Kolleginnen schlicht eine Unverschämtheit zu sagen, dass nur die .NET-Entwicklung wahre Softwareentwicklung sei. Die Vielfalt der nativen Systeme ist dabei so groß, dass ich an dieser Stelle einfach nicht darauf eingehen kann.

Wie funktioniert .NET

Nach dieser einleitenden Übersicht, wollen wir uns nun an die Sache machen, die .NET-Plattform in ihrer Funktionsweise kennen zu lernen. Wir bedienen uns dabei der Verfeinerungstechnik und teilen das bereits bekannte Rechteck zuerst einmal grob in seine Blöcke auf.

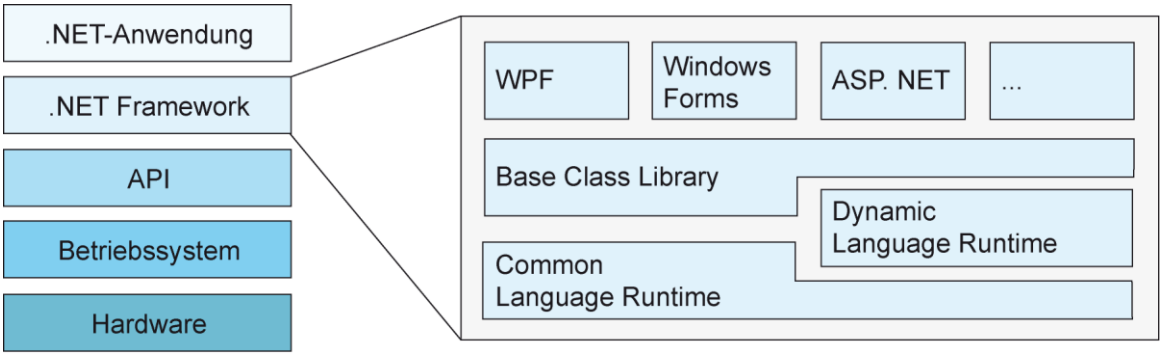

Abbildung 1.4: *Blockdiagramm des .NET Frameworks*

In der **Abbildung 1.4** erkennen wir drei Hauptbestandteile des .NET Frameworks:

- Die Common Language Runtime (CLR) bildet die Laufzeitumgebung, in der die Programme kontrolliert ablaufen
- Die Dynamic Language Runtime (DLR) fügt der Laufzeitumgebung Elemente hinzu, die der speziellen Unterstützung dynamischer Sprachen dienen
- Die Base Class Library (BCL) definiert eine allgemein verwendbare Basisfunktionalität. Das Wort allgemein bezieht sich hier auf Anwendungsgebiete. Die BCL stellt also Funktionalitäten zur Verfügung, die sowohl in clientbasierten als auch serverbasierten Anwendungen oder in verschiedenen Techniken für die Herstellung von Benutzerschnittstellen angewendet werden können.
- Die Rechtecke der oberen Reihe in **Abbildung 1.4**, stehen stellvertretend für verschiedene vorhandene Techniken der Herstellung von Anwendungen. Die Verwendung einer bestimmten Technik ist getrieben durch die Architektur der Lösung und das wiederum hat in der Regel etwas mit Benutzeranforderungen zu tun.

In der Folge betrachten wir in diesem Kapitel die wissenswerten Elemente der CLR. Der Rest des ersten Bands dieses Buches setzt sich mit der Programmiersprache C# und der BCL auseinander. Die beiden

Folgebände 2 und 3 sind der obersten Ebene der verschiedenen Techniken gewidmet.

Die Installation der Common Language Runtime

Nach der Installation der Common Language Runtime interessiert zunächst, wo sich .NET auf dem Rechner befindet und welches für Benutzer und Systemspezialisten die Möglichkeiten sind, um auf .NET einzuwirken.

Hinweis Beginn

Die **Tabelle 1.1** zeigt die Verzeichnisse des Betriebssystems, die im Zusammenhang mit .NET von Bedeutung sind. In der Erklärung wird der Begriff Assembly benutzt. Mit diesem Begriff bezeichnet die .NET-Welt eine Programmdatei (*exe* oder *dll*).

Beachten Sie ferner, dass zwischen Versionen der CLR und Versionen der BCL unterschieden wird. Währen die CLR und die BCL in den Versionen 1.0, 1.1, 2.0 und neu 4.0 veröffentlicht wurden, hat Microsoft folgende zusätzliche Versionen der BCL herausgegeben: 3.0, 3.5 und 3.5 SP1. Im Klartext heißt das, dass die Version 2.0 wohl vorerst auf jedem Windows-Rechner installiert ist und dazu verschiedene Versionen der Bibliothek verfügbar sind.

Hinweis Ende

Das .NET-System installiert sich im Wesentlichen in folgende Pfade des Betriebssystems:

Pfad	Inhalt
%windir%\Assembly	Dieses Verzeichnis enthält den globalen Assemblycache (kurz: GAC). Mehr dazu finden Sie im Abschnitt »Der globale Assemblycache«.
%windir%\Microsoft.NET	Auf einem Rechner können mehrere Versionen des .NET Frameworks gleichzeitig installiert sein. Der Code aller installierten Versionen wird in diesem Pfad untergebracht.
%windir%\Microsoft.NET\Framework\v…	Die Verzeichnisse beginnend mit dem Buchstaben *v* definieren die installierten Versionen des .NET Framework. Beachten Sie, dass die größte Nummer die effektiv installierte Version darstellt. Dabei kann es durchaus sein, dass mehrere Versionen vorhanden sind. Insbesondere .NET 3.5 und .NET 4.0 werden vorerst parallel auf den Rechnern installiert sein.
%windir%\System32	Dieses Verzeichnis ist kein typisches .NET-Verzeichnis. Der größte Teil des Inhalts dieses Systems stammt vom Betriebssystem selber. .NET hat hier einige wenige DLLs, die aber umso wichtiger sind.

Tabelle 1.1: *Pfade die von Microsoft .NET im Betriebssystem Windows verwendet werden*

Neben dem Dateisystem befinden sich auch einige Einträge des .NET Frameworks in der Registrierung. Allerdings handelt es sich dabei um vergleichsweise wenige Daten, da die Hauptsache der Konfiguration von .NET in speziellen Konfigurationsdateien im Dateisystem untergebracht sind. Ich habe zumindest im Laufe meiner .NET-Karriere in diesen Einträgen keine Änderungen vorgenommen. Für alle Interessierten hier die wichtigsten Einträge:

Pfad	Inhalt
HKLM\SOFTWARE\Microsoft\.NETFramework	Einträge für die Laufzeitumgebung von .NET
HKLM\SOFTWARE\Wow6432Node\Microsoft\.NETFramework	Diesen Schlüssel sehen Sie nur auf einem 64-Bit-System. Er dient dazu, die Einträge der 32-Bit-Software dem 64-Bit-System zugänglich zu machen.

HKLM\SOFTWARE\Microsoft\ASP.NET	Einträge für die Laufzeitumgebung von ASP.NET
HKLM\SYSTEM\CurrentControlSet\Services\.NET%	Einträge für die Laufzeitumgebung von .NET

Tabelle 1.2: *Pfade der .NET-Konfiguration in der Registrierung von Windows*

Das .NET Framework stellt ansonsten keine interaktiven Einstellungen zur Verfügung. Einzige Ausnahme ist ein Plug-In in der Microsoft Management Console (MMC) im Fall der Installation des .NET Framework SDK (Software Development Kit). Mit diesem Plug-In können Sie Einstellungen in der CLR und der installierten Konfigurationen interaktiv vornehmen. Den Startpunkt für dieses Plug-In finden Sie bei installiertem SDK in der Systemverwaltung.

Die Common Language Runtime

Die Common Language Runtime bildet die virtuelle Maschine von .NET. Mit Hilfe der CLR werden .NET-Programme übersetzt und ausgeführt. Anders als bei der Herstellung herkömmlicher Anwendungen, bei denen während der Entwicklung direkt der Code erzeugt wird, der auf einem Prozessor ausgeführt werden kann, wird bei .NET ein Zweischrittverfahren angewendet. Der erste Schritt ist die Entwicklung und Herstellung des Programms. Dieser Schritt unterscheidet sich von der Herstellung herkömmlicher Programme im Resultat. Im Gegensatz zum direkt ausführbaren Code erstellen wir mit der Entwicklungsumgebung *Visual Studio* Code, der vom Prozessor unabhängig ist. Diesen unabhängigen Code nennen wir IL-Code (Intermediate Language Code, auch MSIL-Code für Microsoft Intermediate Language Code). Der IL-Code wird wie bei herkömmlichen Programmen in *exe*- oder *dll*-Dateien geformt und nach der Erstellung auf der Maschine des Benutzers installiert.

Der zweite Schritt der Übersetzung des Codes geschieht erst unmittelbar vor der eigentlichen Ausführung einer Methode durch die CLR selber. Diese späte Übersetzung erlaubt es, den hardwareunabhängigen IL-Code erst dann in hardwareabhängigen Maschinencode zu übersetzen, wenn die Hardware, auf der der Code ausgeführt wird, auch wirklich bekannt ist. Diese Teilung ermöglicht es, eine Optimierungen im Code vorzunehmen, die ansonsten nur durch mehrfache Erstellung der Programme möglich sind (Beispiel 32/64-Bit-Problematik).

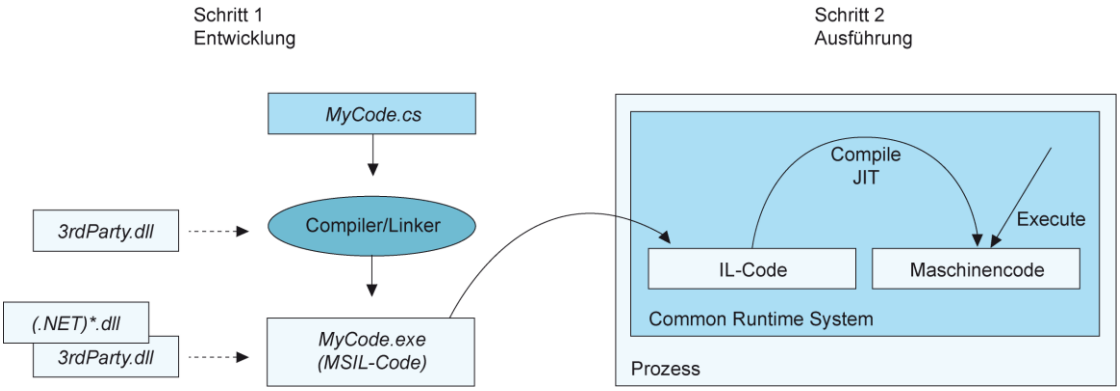

Abbildung 1.5: *Prinzip der Erstellung und Ausführung eines .NET-Programms*

Hinweis Beginn

Die Grundidee einer virtuellen Maschine ist die zur Verfügungsstellung einer immer gleichen Umgebung für den Programmablauf. Dies wird in .NET mit der CLR erreicht. Damit die CLR auf verschiedenen Betriebssystemen (Windows, Mobile Windows, Linux, Solaris …) auf unterschiedlicher Hardware (32 Bit, 64 Bit) gleich funktioniert, muss die CLR die Common Language Infrastructure (CLI) implementieren. Die CLI ist in der Norm ECMA-335 definiert. Diese Norm umfasst weite Teile

von .NET und ist somit für alle Interessierten frei zugänglich.

Dies ist eine sehr interessante Tatsache, hat hier doch der führende Hersteller im Markt der Betriebssysteme ein Kernelement des eigenen Systems in die Hand fremder Verwaltung gegeben. Der Zugang zu den Normen ist übrigens frei. Somit können auch Sie sich in der über 550 Seiten umfassenden Beschreibung direkt informieren.

Die CLI definiert folgende Grundregeln für die Umsetzung eines .NET-Systems auf einer beliebigen Hardware (4. Ausgabe Juni 2006[1]):

- Teil I: Konzepte und Architekturen – Beschreibung der Gesamtarchitektur der CLI mit Beschreibung des allgemeinen Typsystems (CTS), des virtuellen Ausführungssystem (VES) und der neutralen Sprache (CLS)
- Teil II: Metadatendefinition und Semantik – Beschreibung der notwendigen Metadaten des Systems mit seinem Layout (Beispiel Dateiformat) und den entsprechenden logischen Komponenten und deren Semantik aus Sicht eines hypothetischen Assemblers
- Teil III: CIL-Befehlssatz – Beschreibung der Common Intermediate Language (CIL)
- Teil IV: Profile und Bibliotheken – Beschreibung der gesamten CLI-Bibliothek inklusive deren Klassen und Schnittstellen
- Teil V: Debug Interchange Format
- Teil VI: Anhang mit IL-Beispielen, zusätzlichen Informationen zum Assembler und Angaben zum Aufbau von Bibliotheken.

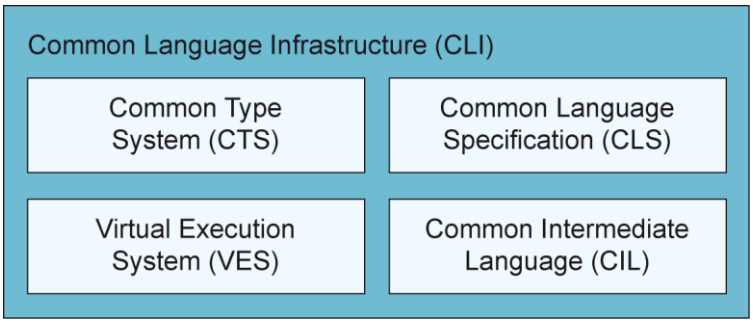

Abbildung 1.6: *Umfang der Norm ECMA-335*

Hinweis Ende

Wie wir bereits gesehen haben, ist die CLR zuständig, Programme zu übersetzen und auszuführen. Wir können auch sagen, dass die CLR ein Programm und seine Ausführung verwaltet. Das Wort verwalten (engl. to manage) hat Pate gestanden, als es darum ging, die Ausführung von .NET-Programmen gegenüber der herkömmlichen Programmausführung begrifflich abzuheben. Wir sprechen bei der kontrollierten Ausführung von Programmen in .NET von so genanntem »verwaltetem Code« (engl. managed code). Demgegenüber wird herkömmlicher Code, der nicht unter der Kontrolle der CLR abläuft, »nicht-verwalteter« Code (engl. unmanaged code) genannt. Der Begriff »nicht-verwalteter Code« wurde in den letzten Monaten zunehmend wieder durch den Begriff »nativer Code« ersetzt, und von mir in diesem Buch auch so verwendet.

Wenn wir die konkreten Aufgaben der CLR anschauen, stellen wir fest, dass darunter viele Aufgaben fallen, die eigentlich Kernfunktionen eines Betriebssystems sind. Daher spreche ich bei meinen Ausbildungen für .NET auch gelegentlich von einem neuen Betriebssystem. Um das zu verdeutlichen,

[1] Siehe auch *http://www.ecma-international.org/publications/standards/Ecma-335.htm*

schauen wir uns die Liste der Aufgaben der CLR genauer an:

- Programme in den Speicher laden (Class Loader)
- Code von der Intermediate Language (IL) in Maschinencode übersetzen (JIT)
- Kontrollieren von Typenkonvertierungen zur Laufzeit (Type Checker)
- Verwalten des Speichers (Garbage Collector)
- Den Code kontrolliert ausführen (Code Manager)
- Verwalten von Ausnahmesituationen (Exception Manager)
- Durchsetzen der Sicherheit (Security Engine)
- Unterstützen des Entwicklers für das Debugging (Debug Engine)
- Ermöglichen von Multitasking in Anwendungen (Thread Support)
- Sicherstellen der Interoperabilität zwischen verwalteten und unverwalteten Anwendungen (COM Marshaler)

Einige dieser Aufgaben interessieren im Alltag nicht. Die CLR nimmt sie wahr, das ist gut so, die Funktionsweise ist aber für die Entwicklungspraxis kaum relevant. Hingegen ist das Verständnis für die Funktionsweise anderer Mechanismen von Bedeutung, weil das eine Grundvoraussetzung ist, um eine Anwendung korrekt zu erstellen.

Der Just-In-Time-Compiler

Der Just-In-Time-Compiler (JIT-Compiler) übersetzt den geladenen Code vom IL-Format in Maschinencode. Das Vorgehen in der Zusammenarbeit zwischen der CLR und dem JIT-Compiler ist methodenbasiert. Das heißt, dass der IL-Code einer Methode in Maschinencode übersetzt wird, sobald diese Methode zum ersten Mal aufgerufen wird. Dieser Maschinencode wird zur Laufzeit des Programms im Speicher gehalten, so dass die Übersetzung kein weiteres Mal notwendig wird, wenn die Methode erneut aufgerufen werden sollte.

Aufgrund der Leistungsfähigkeit des JIT-Compilers (>10.000 Zeilen pro Sekunde) und des methodenbasierten Ansatzes ist das Just-In-Time-Verfahren in der Praxis in Bezug auf die Geschwindigkeit überhaupt kein Problem. Bei großen Anwendungen kann es gelegentlich vorkommen, dass man eine leichte Startverzögerung der Anwendung spürt. Die Verzögerung ist meist nur spürbar, wenn ein direkter Vergleich mit einer nativen Anwendung möglich ist.

Die Ausführungsgeschwindigkeit von Code unter .NET ist in etwa gleich groß wie diejenige eines konventionell erstellten C- oder C++-Programms. Da der sprachspezifische Code der .NET-Sprachen über die Zwischenstation IL in Maschinencode übersetzt wird und somit immer der gleiche Compiler zur Anwendung kommt, ist die Ausführungsgeschwindigkeit der Programme verschiedener Sprachen in etwa gleich. Unterschiede können noch entstehen durch verschiedenartige Übersetzungen von Sprachcode in IL-Code.

Verwaltung der Typen und Typkonvertierung zur Laufzeit

Die objektorientierte Welt definiert als eines ihrer Grundkonzepte die Typenprüfung. Es geht dabei darum, dass die Sprache und das System eine nicht-natürliche Überführung von Informationen vom einen Datentyp in einen anderen verhindern soll. Als natürliche Überführung definieren wir dabei Vererbungsbeziehungen von den spezialisierten Klassen in Richtung allgemeinerer Klassen und einfache Konvertierung von grundlegenden Datentypen, bei denen kleine Werte in große Werte ohne Datenverlust überführt werden können.

Hinweis Beginn

Die Details für den Umgang mit Typen im Rahmen der Programmierung können Sie in Kapitel 2 den

Abschnitten »Vordefinierte Typen« und »Klassen organisieren« entnehmen. Sollten Sie sich unter einem Typ noch keine wirklich konkrete Vorstellung machen können, interpretieren Sie im Rahmen dieses Abschnitts den Begriff Typ als Information eines bestimmten Inhalts wie zum Beispiel eine Ganzzahl oder eine gebrochene Zahl. In diesem Sinne darf es nicht einfach möglich sein, eine gebrochene Zahl in eine Ganzzahl zu überführen, weil mit einer solchen Überführung der Bruchteil der Zahl verloren gehen würde.

Hinweis Ende

In herkömmlichen Umgebungen von Microsoft war die Typenprüfung eine Funktion innerhalb der Übersetzung des Programms. Das heißt, dass der Compiler entsprechende Prüfungen vornahm und zur Laufzeit keine Prüfungen mehr durchgeführt wurden. Mit ein bisschen kreativer Programmierung war es jedoch früher möglich, eine Information mit falschem Typ anzuschauen und so undefinierte Programmzustände zu erreichen, was sich oft in Form eines Programmabsturzes manifestieren konnte.

Unter .NET sorgt sich der Compiler um die Typenprüfung. Zusätzlich sorgt zur Laufzeit das Typensystem dafür, dass Daten nicht einfach durch eine falsche Brille betrachtet werden können. Diese strengere Umsetzung eines Grundkonzepts der objektorientierten Technik hat natürlich den Vorteil, dass unsere Software zuverlässiger wird.

Trotz aller Anstrengungen ist es Microsoft noch nicht gelungen, ein 100% wasserdichtes Typensystem zu erstellen. Jedoch ist die aktuelle Umsetzung bereits so gut, dass die meisten Fehler vom System korrekt erkannt werden und zu adäquaten Fehlersituationen führen. So lassen sich in Kombination mit dem Fehlermanagement nicht nur Programmabstürze vermeiden, sondern es ist auch möglich, eine Fehlereingrenzung vorzunehmen. Durch die korrekte Nutzung von Klassen und dem geschickten Aufbau eigener Klassen ist es zunehmend möglich, auch Code zu schreiben, der ohne explizit programmierte Typumsetzungen auskommt.

Hinweis Beginn

Die CLR verwaltet von jedem Typen, der geladen wurde, Metainformationen im Speicher. Diese Metainformation beschreiben den Typen mit Namen und Inhalt. Unter Inhalt wird an dieser Stelle die Menge der Methoden, Eigenschaften und weiteren Elementen eines Typs (siehe Kapitel 2) verstanden. Mit Reflektion ermöglicht es uns .NET auch zur Laufzeit an diese Typinformationen heranzukommen und uns diese zu Nutzen zu machen. Dies ist unabhängig davon möglich, ob als Ausgangslage nur der Typ oder eine konkrete Information eines bestimmten Typs vorliegt.

Hinweis Ende

Das Typensystem

Das Typensystem der CLR definiert die grundlegenden Datentypen, mit denen ein jedes Programm arbeitet. Mit Hilfe der Datentypen werden Informationen im Speicher gehalten. Jeder Datentyp dient dabei zum Speichern einer anderen Informationsart. Hier finden wir einen entscheidenden Unterschied zur nativen Programmierung, weil die grundlegenden Datentypen im Basissystem und nicht in der Programmiersprache definiert sind.

Die Compiler haben zwar die Möglichkeit, die grundlegenden Datentypen mit andern Namen zu definieren, so dass ein Programmierer den Typ unter anderem Namen anspricht. Allerdings muss die Sprache die Typen implementieren, ansonsten ist die Kompatibilität nicht gewährleistet.

Grundlegende Typendefinition

Das Common Type System (CTS) definiert die Datentypen, die vom Virtual Execution System (VES) direkt unterstützt werden:

Name in der .NET-Bibliothek	Name im CIL-Assembler	CLS-konform	Beschreibung

System.Boolean	bool	Ja	Wahr/Falsch-Wert
System.Char	char	Ja	Unicode-16-Bit-Zeichen
System.SByte	int8	Nein	Vorzeichenbehaftete 8-Bit-Ganzzahl
System.Int16	int16	Ja	Vorzeichenbehaftete 16-Bit-Ganzzahl
System.Int32	int32	Ja	Vorzeichenbehaftete 32-Bit-Ganzzahl
System.Int64	int64	Ja	Vorzeichenbehaftete 64-Bit-Ganzzahl
System.Byte	unsigned int8	Ja	Vorzeichenlose 8-Bit-Ganzzahl
System.UInt16	unsigned int16	Nein	Vorzeichenlose 16-Bit-Ganzzahl
System.UInt32	unsigned int32	Nein	Vorzeichenlose 32-Bit-Ganzzahl
System.UInt64	unsigned int64	Nein	Vorzeichenlose 64-Bit-Ganzzahl
System.Single	float32	Ja	IEC 60559:1989 32-Bit-Gleitkommazahl
System.Double	float64	Ja	IEC 60559:1989 64-Bit-Gleitkommazahl
System.IntPtr	native int	Ja	Vorzeichenbehaftete Ganzzahl in der Hardwarewortbreite
System.UIntPtr	native unsigned int	Nein	Vorzeichenlose Ganzzahl in der Hardwarewortbreite
System.TypedReference	typeref	Nein	Zeiger mit exaktem Typ
System.String	string	Ja	Unicodezeichenkette
System.Object	object	Ja	Objekt oder geboxter Wert

Tabelle 1.3: *Definierte Datentypen gemäß CTS nach ECMA-335*

Wenn Sie bereits Erfahrung mit .NET haben, sehen Sie auf den ersten Blick, dass in der obigen Tabelle der Typ `System.Decimal` fehlt. Das kommt daher, dass dieser Typ nicht im Basisprofil definiert ist, sondern als Bestandteil der so genannten erweiterten numerischen Bibliothek. Vergleichen Sie dazu auch in Kapitel 2 den Abschnitt »Vordefinierte Typen«.

Neben der Art der Daten, die ein Typ aufnehmen kann, definiert dieser auch den Speicherort (siehe auch nächsten Abschnitt »Speicherverwaltung (Garbage Collector)«). Das Verhalten von .NET ist abhängig von den Typenhauptgruppen *Werttyp* und *Referenztyp*.

Werttypen

Werttypen (engl. value type) sind Typen, die im Typsystem eine feste Größe in Bytes aufweisen. Der Speicherort von Werttypen ist automatisch der Stack.

Referenztypen

Referenztypen (engl. reference type, deutsch auch synonym als Verweistyp verwendet) sind Typen, die im Typsystem keine feste Größe aufweisen. Der Speicherort von Referenztypen ist automatisch der Heap. Für das Anlegen von Objekten muss speziell eine Speicheranforderung an die Heap-Verwaltung abgesetzt werden, das benötigt ein wenig mehr Zeit als die Speicherung von Daten auf dem Stack.

Werttypen umwandeln (Boxing und Unboxing)

Werttypen können in Referenztypen umgewandelt werden und so ebenfalls auf dem Heap gespeichert werden. Dieser Vorgang wird in .NET *Boxing* genannt. Der umgekehrte Weg, einen vorher zum Referenztyp umgewandelten Werttyp vom Heap wieder als echten Werttyp auf den Stack zu legen, heißt »Unboxing«.

Dieser Vorgang kann explizit programmiert werden, jedoch sollten Sie das nur dann tun, wenn es gar nicht anders geht. Da der Werttyp für die Speicherung auf dem Heap in ein Objekt eingelagert werden muss, und beim Umkehrprozess das Objekt auch wieder freigegeben werden muss, werden hier Ressourcen und Zeit verwendet.

Wozu dann das Ganze? Bei der Übergabe von Werttypen an Funktionen kann es in bestimmten Fällen notwendig sein, Werttypen als Objekte zu übergeben. In diesem Fall ist ein Boxing notwendig. Übrigens erledigt .NET Boxing und Unboxing oft intern, ohne dass Sie es programmieren. Seien Sie also nicht erstaunt, wenn Sie im Debugger die Werttypen Ihres Programms in Objekten eingepackt wiederfinden.

Speicherverwaltung (Garbage Collector)

Daten werden unter .NET wie in anderen Entwicklungssystemen im Speicher gehalten. Dabei kommen die beiden altbewährten Konzepte des Stack (auch als Stapel bezeichnet) und des Heap (Halde) zum Einsatz.

Der Stack wird durch die eigentliche Codeausführung unter dem so genannten Code Manager verwaltet. Auf dem Stack werden die Adressen der Funktionseintritte und alle Daten verwaltet, deren Typen eine feste Größe aufweisen (siehe auch den vorangehenden Abschnitt »Das Typensystem«).

Der Heap beinhaltet alle restlichen Daten, insbesondere die Daten, deren Typen keine feste Größe aufweisen. Dazu gehören Zeichenketten (Strings) und Objekte (Instanzen von Klassen). Daten auf dem Heap werden unter .NET nicht direkt vom Programm angelegt, sondern vom Verwalter des Heap. Eine Unterkomponente der Heap-Verwaltung ist der so genannte Garbage Collector (GC). Sein Name (deutsch: Müllsammler) deutet schon an, dass sich die Heap-Verwaltung in .NET nicht nur um die Datenaufbewahrung kümmert, sondern auch kontrolliert, ob die Daten noch verwendet werden. Sobald eine Information im Heap durch eine Anwendung nicht mehr referenziert wird, kann der Garbage Collector diese Daten zerstören und den belegten Speicher für die freie Verwendung an das Betriebssystem zurückgeben. Der GC ist dabei in der Lage, interne Referenzierungen innerhalb der verwalteten Daten von Referenzierungen zu unterscheiden, die auf dem Stack liegen, und so ganze Objektnetze oder -bäume freizugeben.

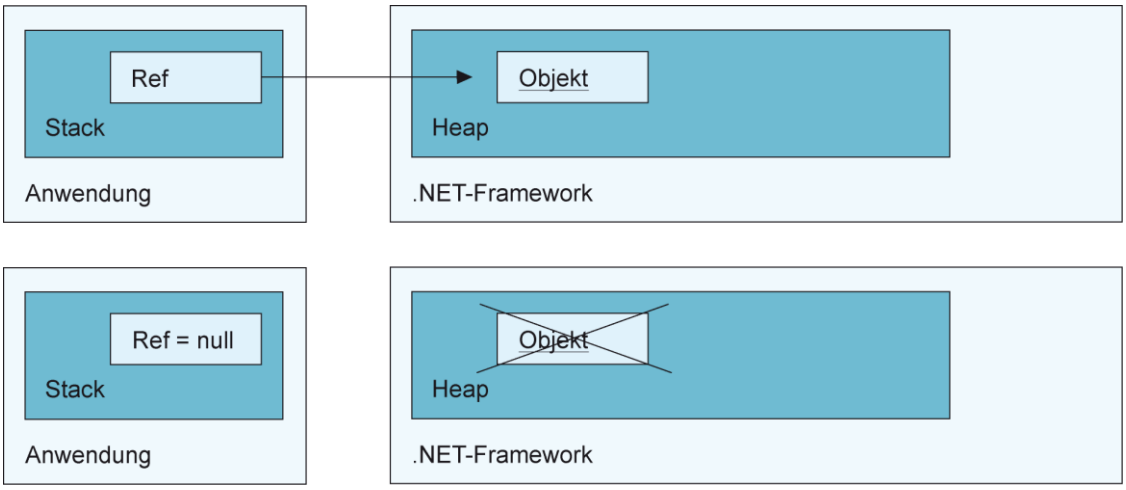

Abbildung 1.7: *Der GC kann ein Objekt automatisch zerstören sobald es nicht mehr referenziert wird*

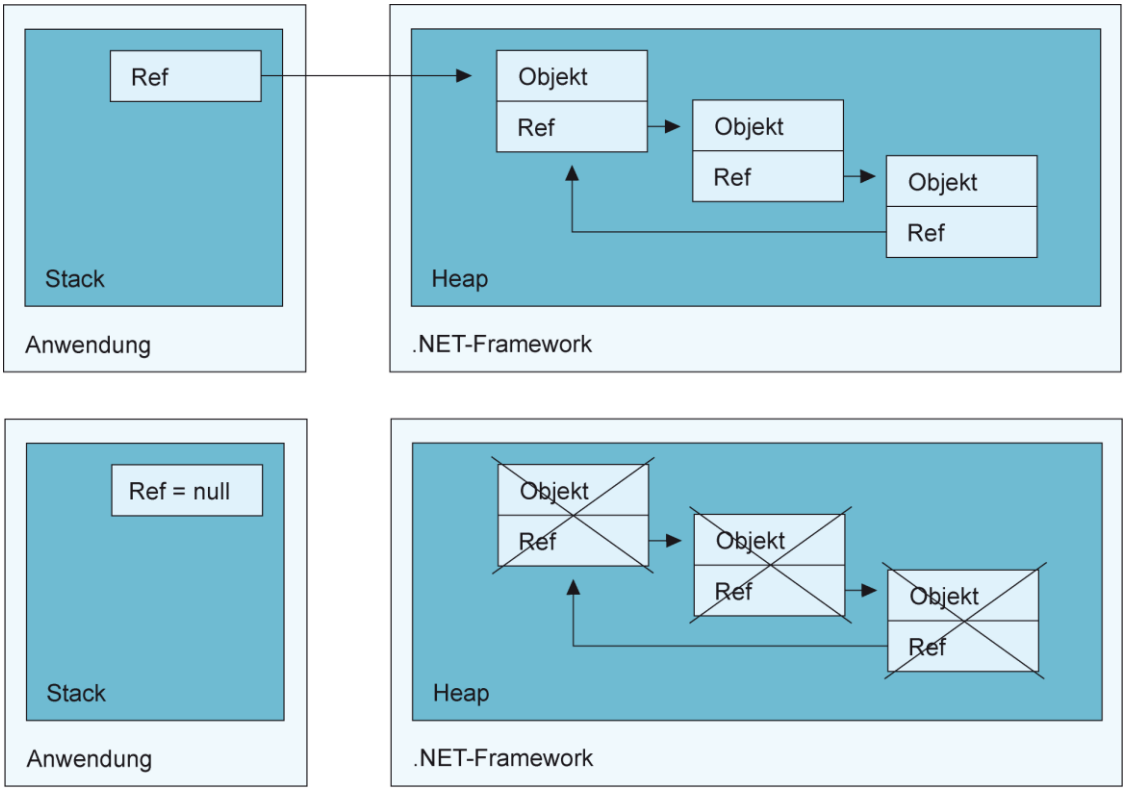

Abbildung 1.8: *Der GC erkennt interne Referenzierungen zuverlässig*

Der GC ist zuständig für die Erkennung der zerstörbaren Objekte. In diesem Zusammenhang sind für die Praxis folgende Informationen von entscheidender Bedeutung: Der GC kommt seiner Aufgabe automatisch nur dann nach, wenn das System keine oder fast keine Speicherressourcen mehr zur Verfügung stellen kann. Dieses Verhalten impliziert, dass unsere Objekte nicht deterministisch zerstört werden. Dies wiederum kann auf unsere Programmierung Einfluss haben. Der Zeitpunkt der Objektzerstörung kann programmatisch beeinflusst werden. Der GC gibt vor dem Freigeben des Speichers den betroffenen Objekten eine letzte Möglichkeit einer Aktivität. Wir nennen diese Aktivität Finalisierung eines Objekts. Auch der Aufruf der entsprechenden Methode kann programmatisch beeinflusst werden (siehe auch Kapitel 2 Abschnitt » Deterministische Kontrolle des Lebenszyklus« und Kapitel 4 Abschnitt »Der Garbage Collector«).

Durchsetzen der Sicherheit

Selbstverständlich unterstützt .NET die Sicherheitsmechanismen von Windows transparent. Dies impliziert, dass die Einstellungen für Dateizugriffe und andere mögliche Einstellungen Vorrang vor den .NET-Einstellungen haben. Zusätzlich definiert .NET Mechanismen, die in Windows so noch nicht eingeflossen sind.

Da ist zum Beispiel die so genannte Codezugriffssicherheit. Mit ihr schützen Sie ihren Code vor unberechtigten Aufrufen oder teilen dem System mit, ob Code auf einem System ausgeführt werden darf oder nicht. Die Auswirkung der Codezugriffssicherheit merken Sie schon bald nach dem Installieren von .NET. Ohne weitere Einstellung im neu installierten System ist es nicht möglich, eine .NET-Anwendung von einem Netzwerklaufwerk eines Servers zu starten. .NET unterscheidet Remotespeicher von lokalen Speichern und erlaubt das direkte Starten von .NET-Anwendungen ausschließlich von

lokalen Speichern. Mit den Codezugriffsberechtigungen können sehr feingliedrige Berechtigungsprofile definiert werden, indem .NET für die meisten Ressourcentypen eigene Codezugriffsberechtigungen zur Verfügung stellt.

Ein weiteres Element bildet die mögliche Kontrolle, ob Programme wirklich von dem Hersteller stammen, von dem sie zu sein behaupten. Dies wird erreicht, indem ein Programm vom Hersteller signiert werden kann. Ein Aufruf in eine signierte Datei wird so verknüpft, dass beim Laden der Programme geprüft wird, ob die Signaturen noch übereinstimmen. Wenn nicht, ergibt sich ein Bindungsfehler zur Laufzeit und .NET verweigert die Arbeit. Programme können mit .NET somit fälschungssicher gemacht werden.

Sicherstellen der Interoperabilität

Microsoft hat in der Vergangenheit viele Versionen von Betriebssystemen auf den Markt gebracht. Dabei musste einem Element immer wieder besondere Beachtung geschenkt werden: der Kompatibilität. Durch Wahrung der Kompatibilität von einem neuen zu den alten Systemen stellte der Hersteller sicher, dass weitgehender Investitionsschutz für Anwendungen der Kunden sichergestellt war.

Nun haben wir aber bereits gesehen, dass .NET hier offenbar eine Ausnahme darstellt, weil .NET-Code als IL-Code grundsätzlich nicht kompatibel ist zu nativem Code. Trotzdem sind die beiden Welten elegant miteinander verknüpfbar. .NET stellt nämlich eine weite Palette von Möglichkeiten zur Verfügung, die Interoperabilität zwischen der nativen und der verwalteten Welt zu gewährleisten. Unter dem Namen PInvoke (platform invoke) definiert die Interoperabilität zunächst Mechanismen zur reinen Nutzung aller APIs mit C-Bindung. Da sind natürlich die WIN32-Funktionen zu nennen, aber auch die vielen Bibliotheken von Firmen, die im Laufe der Jahre entwickelt wurden.

Als weiteren Mechanismus finden wir die Managed Extensions for C++, die es C++-Entwicklern ermöglichen, ihre objektorientierten Bibliotheken für die Nutzung unter .NET fit zu machen. Letztendlich ist dann noch die Interoperabilität mit COM zu erwähnen, die Microsoft so elegant in .NET integriert hat, dass die Nutzung, ja sogar die Erstellung neuer COM-Komponenten, noch nie so einfach war wie unter .NET.

Die Dynamic Language Runtime

Die DLR bildet eine in .NET 4.0 neu eingeführte Ergänzung der Laufzeitumgebung. Sie ist zuständig für die verbesserte Unterstützung von dynamischen Sprachen, wie sie namentlich im Bereich von Scripting vorkommt. Beispiele für solche Sprachen sind JavaScript, PHP, Ruby, Python, Groovy.

Die Funktionalität der DLR lässt sich in drei Hauptgebiete unterteilen:

- Ausdrucksbäume (Expression trees). Die DLR unterstützt Ausdrucksbäume damit die Semantik der nutzenden Sprache einfacher abgebildet werden kann. Ausdrucksbäume unterstützen die Flusskontrolle, Zuweisungen und andere sprachspezifische Knoten.
- Caching von Aufrufinformationen. Die DLR erstellt die notwendigen Typinformationen für die Behandlung von Objekten erst zur Laufzeit. Da diese Operationen relativ aufwändig sind und im Vergleich zur statischen Bindung eines streng typisierten Systems viel Laufzeit erfordern, stellt die DLR einen speziellen Zwischenspeicher für gewonnene Typinformationen zur Verfügung. Damit lassen sich wiederholte Aufrufe auf Objekte oder Methoden massiv beschleunigen.
- Ferner unterstützt die DLR die dynamische Objektinteraktion mit der zur Verfügungsstellung neuer Typen der CLR, die die Entwicklung von dynamischen Typen vereinfachen.

Visual Studio 2010

Visual Studio ist das Produkt von Microsoft für die Entwicklung von Software. Das Produkt wird durchschnittlich alle drei Jahre in einer neuen Version (aktuell 2010) aufgelegt, um die entsprechenden neuen Technologien von .NET, aber auch des darunter liegenden Betriebssystems zu unterstützen.

Das Produkt ist in verschiedenen Editionen verfügbar. Eine Edition beschreibt dabei die dem Entwickler zur Verfügung gestellte Funktionalität. Visual 2010 ist in folgenden Editionen verfügbar:

Edition	Inhalt
Visual Studio Professional	Die Professional Edition ist eine komfortable Entwicklungsumgebung mit allen nötigen Kernfunktionen für die professionelle Entwicklung von Anwendungen für Windows, das Web, Microsoft Office, die Cloud, SharePoint, Silverlight und Multi-Core-Szenarien
Visual Studio Premium	Die Premium Edition ist die Vollausstattung für Softwareentwickler und -tester, um hoch skalierende Anwendungen auf Enterprise-Niveau zu entwickeln. Sie enthält alle Funktionen der Professional-Variante sowie zusätzliche Funktionen, die komplexe Datenbankentwicklung und eine durchgängige Qualitätssicherung ermöglichen
Visual Studio Ultimate	Die Ultimate Edition ist die mittlerweile dritte Generation der Microsoft-Komplettlösung für effiziente Verwaltung des Lebenszyklus einer Anwendung für Teams jeder Größe. Sie enthält den kompletten Umfang der ehemaligen Team Suite Edition

Tabelle 1.4: *Produktportfolio von Visual Studio*

Für die Entwicklung im Team kann jede Edition mit dem *Team Foundation Server* (TFS) verbunden werden. Der TFS stellt eine Versionskontrolle zur Verfügung und integriert Informationen rund um das Management des Lebenszyklus. Detailliertere Informationen über die Unterschiede der verschiedenen Editionen können Sie der Produktseite von *Visual Studio 2010* auf der Webseite von Microsoft entnehmen.

Dieses Kapitel beschreibt die wichtigsten allgemeinen Funktionen von Visual Studio. Spezifische technologieabhängige Techniken wie Herstellen von Anwendungen für graphische Benutzeroberflächen oder Client-Server-Anwendungen werden in den einzelnen Kapiteln des Buches (u.U. in folgenden Bänden) beschrieben.

Die Installation von Visual Studio erfolgt als Anwendung im gewohnten Rahmen über den Installations-Assistenten von Microsoft. Es gibt nur wenige Einstellungen vorzunehmen, wobei zu sagen ist, dass es nur selten Gründe gibt, nicht eine Vollinstallation des Produkts vorzunehmen. Der notwendige Speicherplatz von mehreren GB (inklusive der Dokumentation) sollte auf einem modernen Entwicklungssystem kein Problem darstellen.

Hinweis Beginn

Unter Windows Vista und Windows 7 müssen Sie darauf achten, dass Visual Studio nach der Installation jeweils mit Administratorrechten gestartet wird. Nur so ist gewährleistet, dass die Funktionalität in allen Belangen korrekt erfüllt wird.

Beachten Sie ferner, dass nach der Installation nicht nur Visual Studio im Startmenü eingetragen ist, sondern ebenfalls einige Tools darin verknüpft sind. Neben diesen wenigen Tools, die direkt über das Startmenü erreichbar sind, stellt die Entwicklungsumgebung eine Fülle von Kommandozeilen-Werkzeugen zur Verfügung. Entnehmen Sie dem Anhang des Buches einer Liste der verfügbaren Werkzeuge.

Um die mitgelieferten Werkzeuge von Visual Studio und .NET in einem Kommandofenster benutzen zu

können, empfehle ich Ihnen das Kommandofenster mittels des ebenfalls im Startmenü unter *Visual Studio Tools* eingetragenen *Visual Studio Command Prompt* zu erstellen. Die Konfiguration dieses Kommandofensters ist mit den entsprechenden Pfaden konfiguriert, so dass die Tools im System automatisch lokalisiert und ausgeführt werden können.

Für eine manuelle Konfiguration der Umgebungsvariablen *PATH* benutzen Sie folgende Pfade ihres Systems (Kontrollieren Sie die Angaben auf Ihrem System vor der Benutzung):

Verzeichnis	Inhalt
%windir%\Microsoft.NET\Framework\v4.0.xxx	Dateien des .NET Frameworks. Darin sind auch einige Werkzeuge vorhanden (siehe auch Anhang).
%programfiles%\Microsoft SDKs\Windows\v7.0A\bin	Dateien der Entwicklungsumgebung. Darin enthalten sind sowohl Kommandozeilenwerkzeuge als auch Werkzeuge mit grafischer Benutzeroberfläche.

Tabelle 1.5: *Für die Softwareentwicklung wichtige Pfade auf dem System*

Hinweis Ende

Konfiguration und Einstellungen

Nach der Installation empfehle ich jeweils die Einstellungen von Visual Studio zu studieren, oder bereits bekannte Einstellung vorzunehmen. Es geht dabei nicht primär darum, möglichst alle Standardeinstellungen abzuändern, sondern vielmehr darum, dass Sie ahnen welche Einstellungen vorgenommen werden können. Es kommt immer mal wieder vor, dass man sich später bei der Arbeit an irgend einem Detail des Verhaltens von Visual Studio stört, und man gar nicht weiß, wo und ob das Verhalten für die eigenen Bedürfnisse angepasst werden kann oder nicht. Deshalb hier auch ein paar Tipps worauf Sie speziell achten sollten.

Die Einstellungen finden Sie im Menü *Tools/Options…*. Das Dialogfeld gemäß **Abbildung 1.9** zeigt auf der linken Seite die hierarchisch gegliederten Themen der Einstellungen und auf der rechten Seite das jeweils dazugehörige Dialogfeld mit den Details. Beachten Sie auch, dass einige Einstellungen hierarchisch wirken.

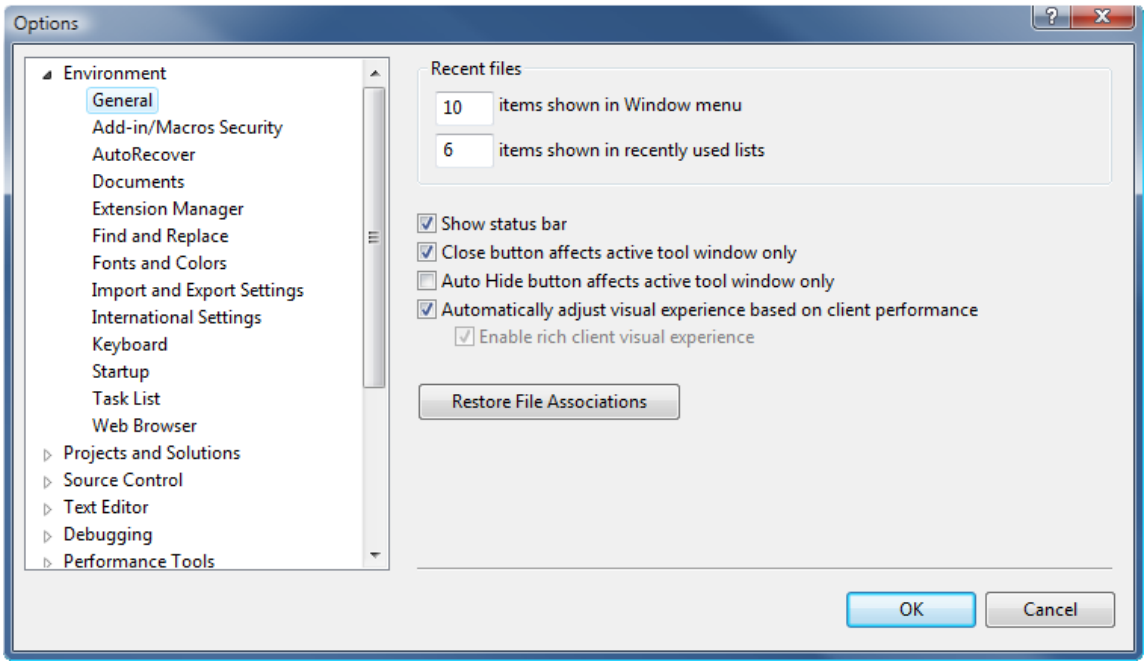

Abbildung 1.9: *Dialogfeld für Einstellungen in Visual Studio*

Einstellung	Erklärung
Environment/Recent files	Gleich auf der ersten Seite kann die Größe der Liste zuletzt gebrauchter Dateien konfiguriert werden. Eine große Liste erleichtert das Auffinden von Dateien. Ich habe mit dem Wert 25 bessere Erfahrungen gemacht als mit dem Standardwert.
Environment/Font and Colors	In diesem Teil des Dialogfeldes können Sie die Farbgebung von Elementen in allen erdenklichen Editoren steuern. Die Standardeinstellungen sind meistens OK, allerdings empfehle ich die drei Elemente von *XML Doc ...* mit einer hellen von weiß unterschiedlichen Hintergrundfarbe zu versehen. Damit erreichen Sie ein optische bessere Trennung von Codeteilen bei der Verwendung von XML-Kommentaren (siehe auch Kapitel 2 und dort den Abschnitt, »Dokumentationskommentare«).
Environment/Import and Export	Mit Hilfe dieses Dialogfelds können Sie Ihre Einstellungen in eine Datei exportieren, (sichern) um sie auf einer andern Maschine erneut einzulesen, so dass die Einstellungen bei mehreren Installationen nur einmal vorgenommen werden müssen.
Environment/Keyboard	Visual Studio definiert ein Benutzerprofil. Wählen Sie in diesem Dialogfeld im obersten Element das Ihnen am besten entsprechende Profil aus. Ferner erlaubt dieses Dialogfeld die Änderung einzelner im Profil definierter Tastenkürzel oder die Definition von neuen Tastenkürzel.
Source Control	In diesem Bereich konfigurieren Sie das Verhalten von Visual Studio im Zusammenhang mit der Verwaltung von Quellcode.
Text Editor	Visual Studio stellt den Texteditor für das Editieren von Inhalten in verschiedensten Varianten zur Verfügung. Dabei erlaubt die individuelle automatische Formatierung von verschiedenen Inhalten (C# ist anders als HTML) unterschiedliche Definitionen. In diesem Bereich definieren Sie nicht nur, welcher Editor eine Nummerierung der Zeilen zeigt und welcher nicht, sondern

	Sie definieren auch die Formatierung der Inhalte nach Ihrem Wohlgefallen.
Debugging	Auch das Verhalten des Debuggers ist nicht in jedem Fall optimal. Studieren Sie die Hauptseite dieser Einstellungen, um das Verhalten Ihren aktuellen Bedürfnissen anzupassen.

Tabelle 1.6: *Empfohlene Einstellungen in Visual Studio*

Mit Visual Studio eine Lösung aufbauen

Visual Studio 2010 organisiert die Projekte wie alle seine Vorgänger mittels eines zweistufigen Konzepts. Die obere Stufe wird *Solution* genannt. Sie ist Container für ein oder mehrere Projekte. Beide Stufen können mit Verzeichnissen beliebig weiter strukturiert werden.

Eine neue *Solution* wird am einfachsten über den Menübefehl *New/Project* angelegt. Nach dieser Auswahl startet *Visual Studio* den Assistenten für die Erstellung eines neuen Projekts (siehe **Abbildung 1.10**). Visual Studio merkt automatisch, dass noch keine Lösung vorhanden ist, und bietet integriert im Assistenten für das Projekt an, den Namen der Lösung mit zu definieren.

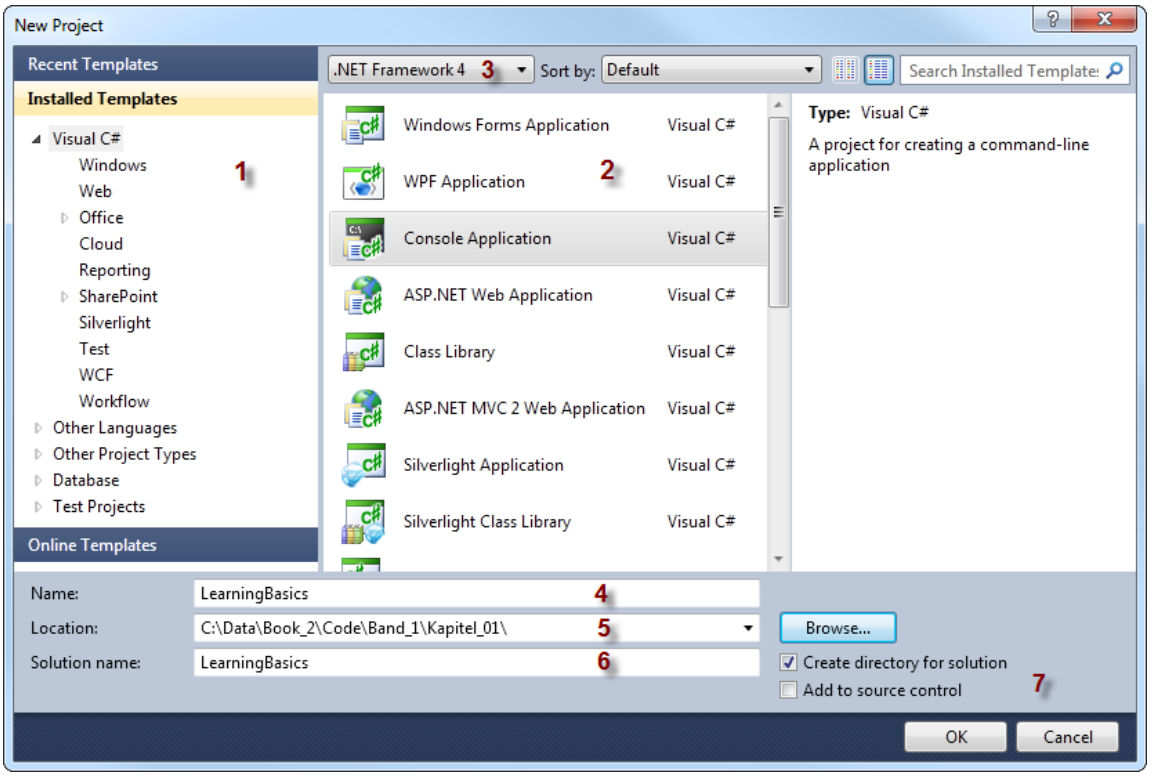

Abbildung 1.10: *Assistent für die Erstellung einer neuen Solution in Visual Studio 2010*

Der Assistent bedarf folgender Bedienung:

#	Benennung	Inhalt
1	Struktur der Vorlagen	Zeigt die Struktur, in die die verschiedenen, verfügbaren Vorlagen für die Erstellung unterschiedlicher Projekte eingeteilt sind.
2	Liste der Vorlagen	Diese Liste zeigt die Vorlagen, die im selektieren Strukturelement 1 verfügbar sind.
3	Zielframework	Erlaubt die Auswahl des .NET Frameworks, auf dem das zu erstellende Projekt

		lauffähig sein soll.
4	Projektname	Definieren Sie in diesem Textfeld den Namen des zu erzeugenden Projekts
5	Speicherort	Definieren Sie in diesem Textfeld, wo die Lösung und das darin enthaltene Projekt erzeugt werden soll.
6	Name der Solution	Definieren Sie in diesem Textfeld den Namen der Solution. Standardmäßig schlägt der Assistent vor, den Projektnamen auch als Namen der Solution zu verwenden.
7	Zusatzeinstellungen	Wählen Sie hier die Optionen für die Erstellung. Ich empfehle unbedingt das erste Kontrollkästchen *Create Directory for solution* eingeschaltet zu lassen. Durch diese Auswahl bewahren Sie sich die Freiheit später ohne Probleme weitere Projekte in die gleiche Lösung zu ergänzen.

Tabelle 1.7: *Erklärungen zu den Eingaben des Assistenten gemäß* ***Abbildung 1.10***

Hinweis Beginn

In diesem Band von »Handbuch der .NET 4.0 Programmierung« verwende ich fast ausschließlich die Projekttypen *Console Application* und *Class Library*. Der Typ *Console Application* erstellt ein ausführbares Programm (*.exe*), das mit einer zeichenorientierten Benutzerschnittstelle ausgestattet ist. Der Typ *Class Library* erstellt eine zusätzliche Bibliotheksdatei, die von einer andern Bibliotheksdatei oder einem ausführbaren Programm mitverwendet werden kann (*.dll*).

Über die Vorlagenstruktur *Other Project Types/Visual Studio Solutions* erreichen Sie die Möglichkeit, eine leere Solution anzulegen. In diese hinein kann anschließend ein Projekt beliebigen Typs ergänzt werden. Zum Ergänzen einer Solution mit einem weiteren Projekt lesen Sie später in diesem Kapitel den Abschnitt »Die Lösung mit einer eigenen Bibliothek ergänzen«.

Die restlichen zur Verfügung stehenden Projekttypen werden im Rahmen der andern Bände dieses Buches erklärt.

Hinweis Ende

Das Resultat einer durch Visual Studio generierten *Solution* wird nach einem kurzen Moment im Fenster *Solution-Explorer* angezeigt (siehe **Abbildung 1.11:**). Gleichzeitig öffnet *Visual Studio* auch den Editor für das Bearbeiten der erzeugten Daten. Beachten Sie, dass der Typ des gestarteten Editors abhängig vom Projekttyp ist.

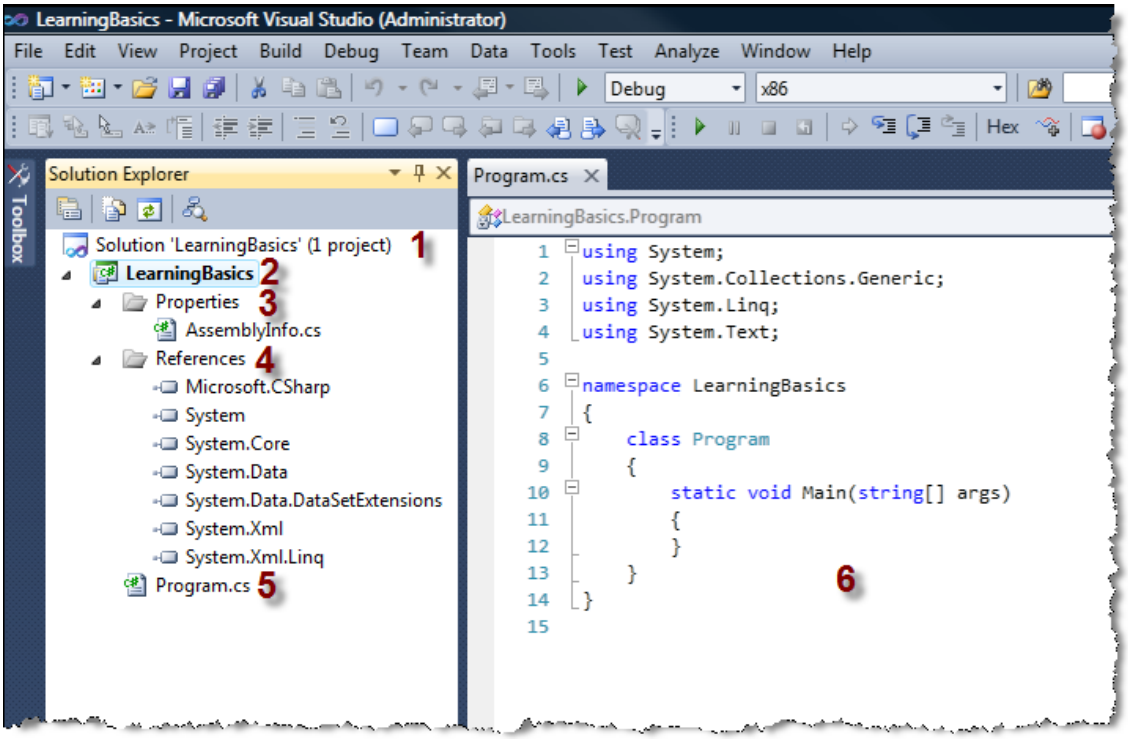

Abbildung 1.11: *Visual Studio nach dem Erstellen eines Konsolenprojekts*

#	Benennung	Inhalt
1	Name der Solution	Die Solution wird in Form einer hierarchischen Struktur dargestellt. Der oberste Knoten dieser Struktur bildet die Solution.
2	Projektname	Das erstellt Projekt ist der Solution untergeordnet und wird mit seinem Namen entsprechend eingerückt dargestellt. Eine Solution kann beliebig viele Projekte beinhalten. In diesem Fall sind die Projekte innerhalb der Solution alle auf der gleichen Stufe angeordnet. Das Resultat eines Projekts ist eine Assembly.
3	Projekteigenschaften	Das Verzeichnis *Properties* enthält Versionsinformationen zur erzeugten Assembly dieses Projekts (siehe auch in diesem Kapitel den Abschnitt »Assembly-Informationen«). Je nach Projekttyp werden in diesem Verzeichnis weitere Einträge für Einstellungen der Anwendung und Ressourcen generiert.
4	Referenzen	Das Verzeichnis *References* enthält die von diesem Projekt referenzierten Assemblys. Die Typen der referenzierte Assemblys können im entsprechenden Projekt verwendet werden (siehe auch in diesem Kapitel den Abschnitt »Die Klassenbibliothek benutzen«).
5	Codedateien	Das erzeugte Projekt enthält in der Regel Code. Die Anzahl der erzeugten Codedateien und deren Inhalte ist dabei ebenfalls vom gewählten Projekttyp abhängig. Es können C#-Dateien aber auch XML-Dateien und andere Typen vorkommen.
6	Editor	Mit dem Erzeugen des Projekts wird in der Regel auch der Standardeditor für das Bearbeiten der generierten Daten angezeigt. Auch hier ist der effektiv angezeigte Editor abhängig vom Typ des erzeugten Projekts.

Tabelle 1.8: *Erklärungen zum generierten Projekt gemäß* ***Abbildung 1.11***

Hinweis Beginn

Die erzeugte Struktur in Visual Studio wird fast identisch auf dem Dateisystem abgebildet. Sie können das im *Windows-Explorer* nachvollziehen, aber auch im *Solution-Explorer* die Schaltfläche *Show all files* betätigen (Nummer 1 in **Abbildung 1.12**). Nun werden alle Verzeichnisse und Dateien in der Struktur unterhalb des Verzeichnisses der Lösung angezeigt.

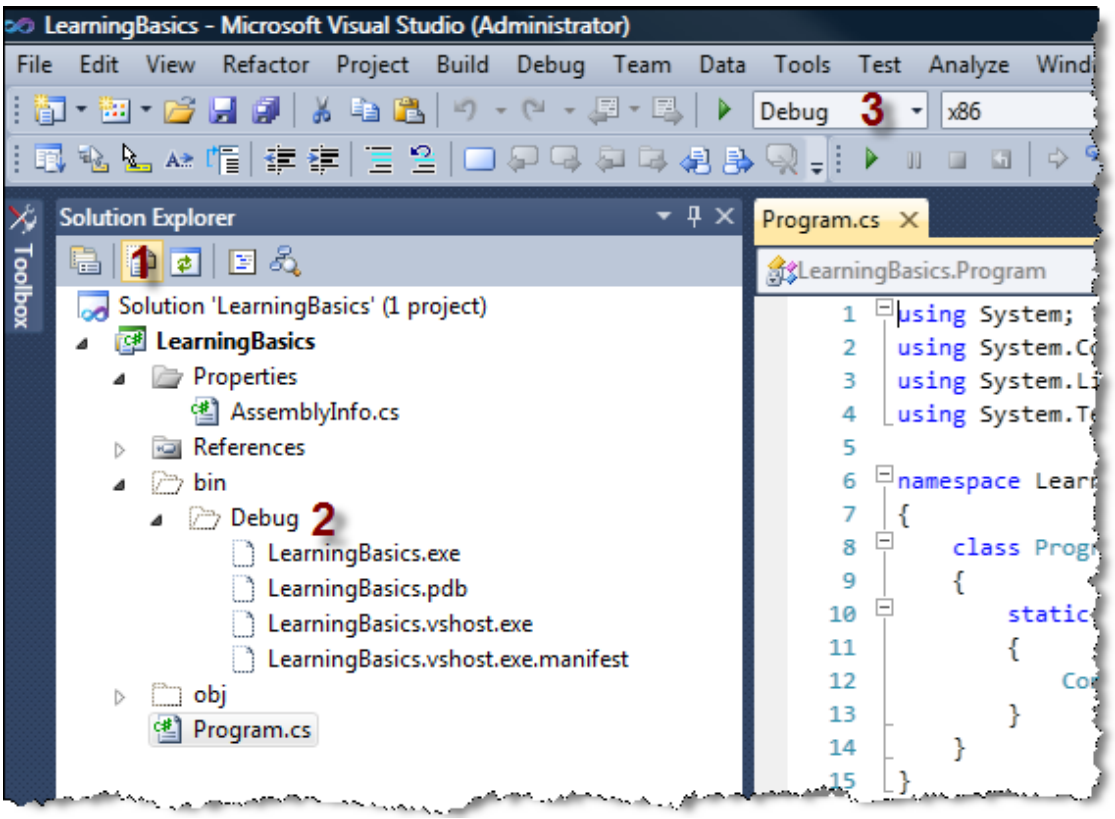

Abbildung 1.12: *Struktur im Solution-Explorer nach Betätigen der Schaltfläche* Show all files

Hinweis Ende

Von besonderem Interesse ist dabei das Verzeichnis *bin*. Dieses enthält je nach Projektfortschritt das Verzeichnis *Debug* und das Verzeichnis *Release*. In diese Verzeichnisse überträgt *Visual Studio* beim Kompilieren des Projekts die Dateien, die für die Ausführung notwendig sind. Das Verzeichnis *Debug* wird dabei jeweils zum Zielverzeichnis, wenn Sie die Konfiguration der Lösung auf *Debug* einstellen. Die Verwendung des Verzeichnisses *Release* ist sinngemäß.

Das Verzeichnis *obj* wird von Visual Studio selbstständig während der Erstellung von Anwendungen verwendet. Die beiden Verzeichnisse *bin* und *obj* können Sie zur Übertragung von Lösungen auf andere PCs einfach löschen. Die Inhalte beider Verzeichnisse sind mit einfachem Erstellen der Lösung wiederherstellbar.

In dem gezeigten einfachen Beispiel ist die Datei *LearningBasics.exe* im Verzeichnis *bin\Debug* des Projekts das eigentliche Resultat des Projekts. Alle andern Dateien in dem Verzeichnis sind zusätzliche, von Visual Studio verwendete Dateien zum Debuggen der Anwendung. Diese Dateien sind im Normalfall nicht auf das System des Endbenutzers zu übertragen.

Die Konfiguration kontrollieren und korrigieren

Zu jedem Projekt gehören detaillierte Projekteinstellungen, die von Visual Studio mit dem Projektassistenten angelegt werden. Nach der Erstellung eines Projekts in Visual Studio liegt der nächste Schritt in der Kontrolle und allfälligen Korrektur dieser projektspezifischen Konfiguration. Die Konfiguration ist direkt über die Eigenschaften des Projekts einsehbar und kann entweder über das Kontextmenü auf dem Projektknoten der Hierarchie im *Solution-Explorer* oder über einen Doppelklick auf dem Knoten *Properties* des Projekts erreicht werden.

Hinweis Beginn

Die Visual Studio-Solution selbst verfügt ebenfalls über Einstellungen, die über das Kontextmenü des entsprechenden Knotens erreicht werden. Auf die Erklärung dieser Einstellungen verzichte ich an dieser Stelle, da diese vergleichsweise einfach sind und ich in meinem »Handbuch der .NET 4.0-Programmierung« zu gegebenen Zeitpunkt darauf zurückkomme.

Hinweis Ende

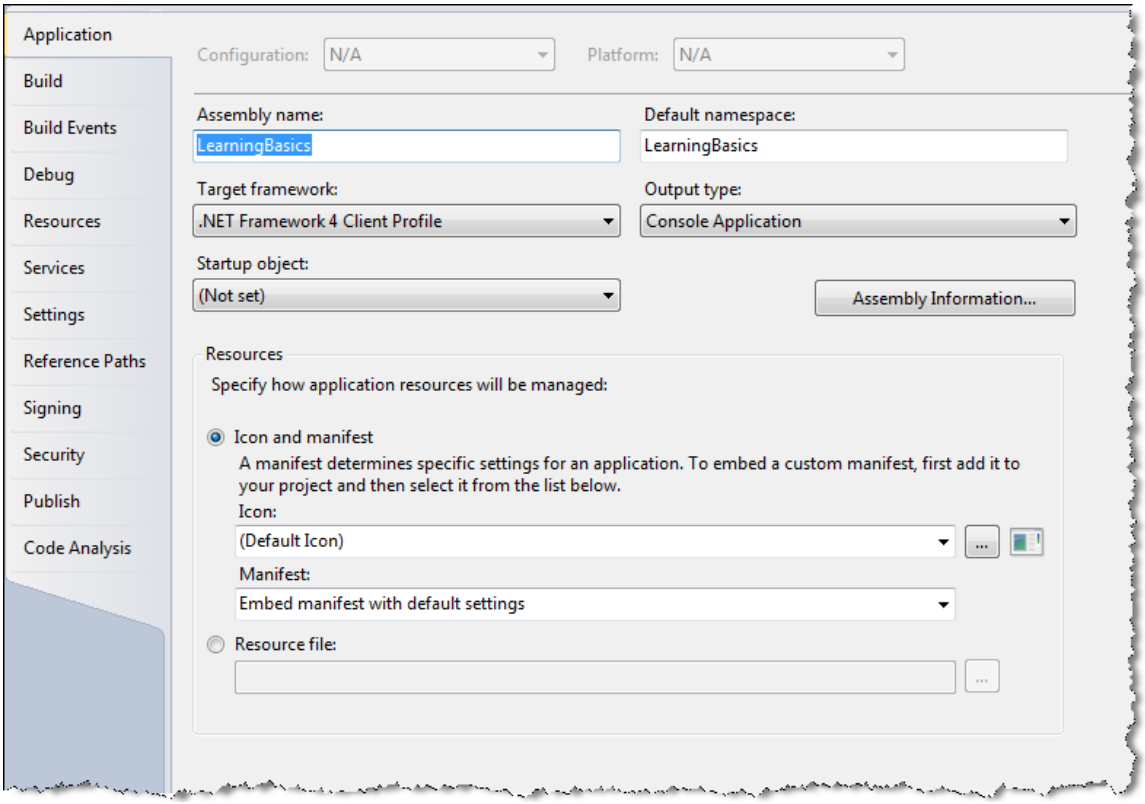

Abbildung 1.13: *Projekteinstellungen auf Anwendungsstufe*

Die Projekteinstellungen sind umfangreich und aufgrund dessen in mehrere Dialogfelder aufgeteilt. Die verschiedenen Dialogfelder sind über Tabs auf der linken Seite des Konfigurationsdialogfeldes erreichbar.

Grundeinstellungen der Anwendung

Im Dialogfeld *Application* nehmen Sie Grundeinstellungen für das gesamte Projekt vor (siehe **Abbildung 1.13**).

Feld	Beschreibung
Assembly name	Dateiname der Assembly, die mit diesem Projekt hergestellt wird. Die Namenserweiterung (*.exe* oder *.dll*) ist abhängig von der Einstellung im Feld *Output type* und wird nicht beim Namen definiert.
Default namespace	Namensraum, der Verwendung findet, wenn Sie ein neues Codeelement in Ihr Projekt einfügen (zum Beispiel eine neue Klasse ergänzen). Mehr Informationen zum Namensräumen finden Sie in Kapitel 2, »Organisation, Schreibweise und Verwendung von Namensräumen«.
Target framework	Auswahl des minimalen .NET Frameworks, auf dem Ihre Anwendung funktionieren soll. Bei Einschränkung auf eine ältere Framework-Version als .NET 4.0 wird auch während der Codierung jeweils sichergestellt, dass nur Assemblys, Typen und Methoden von .NET verwendet werden, die auch tatsächlich in der eingestellten Version verfügbar sind.
Output type	Definiert den erstellten Typ der Assembly. Diese Einstellung bitte mit Vorsicht behandeln. Es ist zwar möglich, dass Sie hier aus einer Konsolenanwendung eine Windows-Anwendung herstellen, aber die alleinige Veränderung dieser Einstellung reicht nicht aus, dass ein vollgraphisches Fenster für Ihre Anwendung entsteht. Da müsste auch codierungstechnisch noch einiges eingebaut werden.
Startup object	Bei einigen Projekttypen ist es nicht ausreichend, das Projekt als Startobjekt zu erklären, sondern es muss ein einzelnes Objekt innerhalb des Projekts als Startobjekt deklariert werden. Die Definition kann hier oder über das Kontextmenü direkt auf dem betreffenden Objekt vorgenommen werden (Beispiel ASP.NET-Webseite in einem Webprojekt).
Assembly Information ...	Ausführliche Informationen hierzu finden Sie im folgenden Abschnitt.

Tabelle 1.9: *Wichtige Einstellungen des Dialogfelds* Application

Assembly-Informationen

Eine Assembly verfügt intern über Informationen, anhand deren später in einer realen Installation erkannt werden kann, von wem diese Assembly stammt und in welcher Version sie vorliegt.

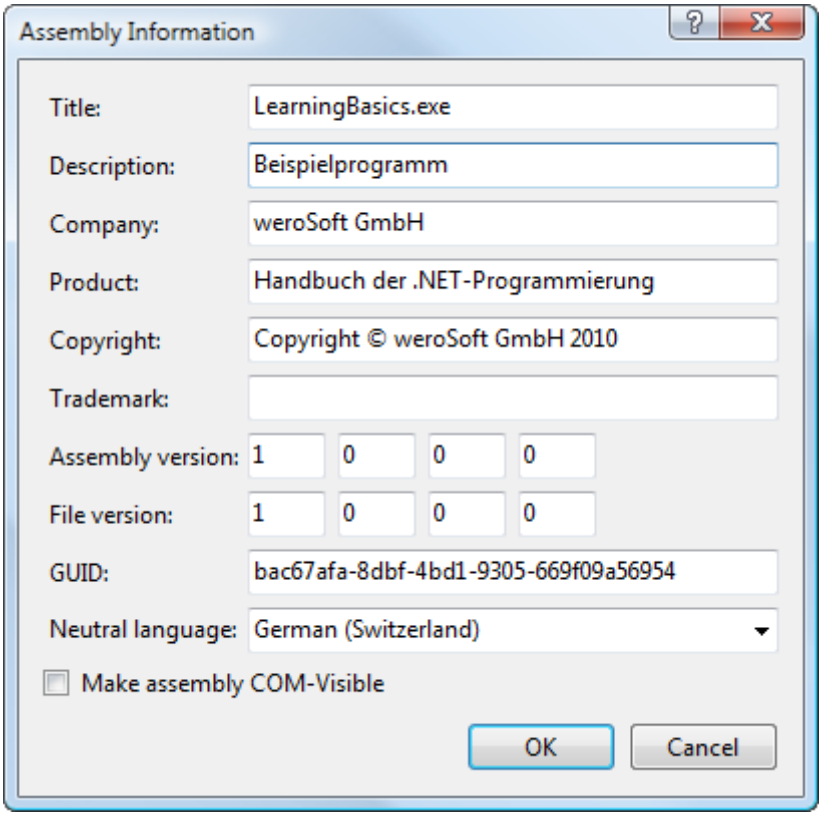

Abbildung 1.14: *Das Dialogfeld* Assembly Information

Feld	Beschreibung
Title	Grundsätzlich definieren Sie in diesem Feld einen beliebigen Titel für Ihre Assembly. Ich empfehle Ihnen allerdings den exakten Dateinamen inklusive der Namenserweiterung anzugeben. Das hat insbesondere bei Bibliotheksdateien den Vorteil, dass bei der Referenzierung die Datei korrekt erkannt wird
Description	Hier definieren Sie eine beliebige Kurzbeschreibung der Datei. Ist eine Kurzbeschreibung kaum oder nur schwer möglich, empfehle ich erneut, den Namen der Assembly einzusetzen.
Company	Hier definieren Sie den Namen der Herstellerfirma, die für diese Assembly verantwortlich ist. Beachten Sie, dass bei Auftragsarbeiten der Name des Besitzers und nicht des Herstellers eingetragen wird.
Product	Meistens bilden mehrere Assemblys zusammen ein Produkt. Definieren Sie in diesem Feld den Namen des Produkts. So kann die Assembly später besser zugeordnet werden.
Copyright	Definieren Sie hier Ihre Copyright-Angabe, wie zum Beispiel: • Copyright ©, Firma Jahr • © Firma, alle Rechte vorbehalten
Trademark	Definieren Sie eine allfällig vorhandene, geschützte Handelsbezeichnung, unter der die Assembly einzuordnen ist. Haben Sie keine solche Handelsbezeichnung, lassen Sie das Feld leer.

Assembly version	Dieses Feld definiert die Version der Assembly. Die Versionsbezeichnung besteht aus den vier Teilen <Hauptversion>.<Nebenversion>.<Buildnummer>.<Revision>. Die Version akzeptiert folgende Muster: • Explizite Definition einer Nummer pro Angabe (Beispiel 2.4.53.24567) • Automatische Erstellung der Revision (Beispiel: 2.4.53.*) • Automatische Erstellung der Buildnummer und der Revision (Beispiel: 2.4.*) Sie sollten darauf achten, dass Sie keine Version 0.0.0.0 definieren. Das kann in wenigen speziellen Konstellationen zu Schwierigkeiten führen. Beginnen Sie einfach bei 1.0.0.0. Die Assembly-Versionsangabe ist im Zusammenhang mit dem Laden von Assemblys in einem häufig benutzten Fall von entscheidender Wichtigkeit. Mehr dazu lesen Sie später in diesem Kapitel im Abschnitt »Suchen und Finden von Programmdateien und Assemblys.
File version	Mit dieser Angabe ist es möglich eine von der Assembly-Version abweichende Dateiversion zu erstellen. Diese Angabe wird benutzt, um es Programmen, die nicht auf der Basis von .NET arbeiten, zu ermöglichen eine Dateiversion zu erkennen.
GUID	Jede Assembly wird mit einer eindeutigen Nummer versehen. Der Inhalt dieses Felds wird von Visual Studio bereits generiert und bedarf keiner Änderung.
Neutral language	Definiert die Sprache, in der Sie Daten in der Assembly einlagern. Die korrekte Definition dieses Feldes ist bei DLLs besonders wichtig. Die Angabe beschleunigt unter Umständen das Auffinden von Daten während der Benutzung des Programms und kann auch helfen, den Speicherbedarf zu reduzieren.
Make assembly COM-Visible	Dieses Kontrollkästchen hat den Charakter eines Hauptschalters. Wenn es ausgeschaltet ist, können keine Typen aus der Assembly für die COM-Technik zugänglich gemacht werden, selbst wenn diese korrekt definiert sind. Mehr dazu entnehmen Sie Kapitel 10.

Tabelle 1.10: *Wichtige Einstellungen des Dialogfelds* Assembly-Information

Einstellungen zur Kompilation der Anwendung

Die Dialogfeldseite *Build* nimmt erweiterte Einstellungen für den effektiven Herstellungsvorgang (Kompilation) vor.

Wichtig Beginn

Diese Dialogfeldseite definiert oben in den beiden Dropdown-Listenfeldern *Configuration* und *Platform* die Konfiguration, für die diese Einstellungen vorgenommen werden. Standardmäßig arbeiten Sie mit einer Plattform und zwei Konfigurationen. Die Konfiguration unterscheidet zwischen *Debug* (Entwicklungsversion mit Debug-Code) und *Release* (Kundenversion ohne Debug-Code). Durch Wechseln der aktuellen Einstellung werden im Dialogfeld selbst die Daten entsprechend der Auswahl ausgetauscht.

Wichtig Ende

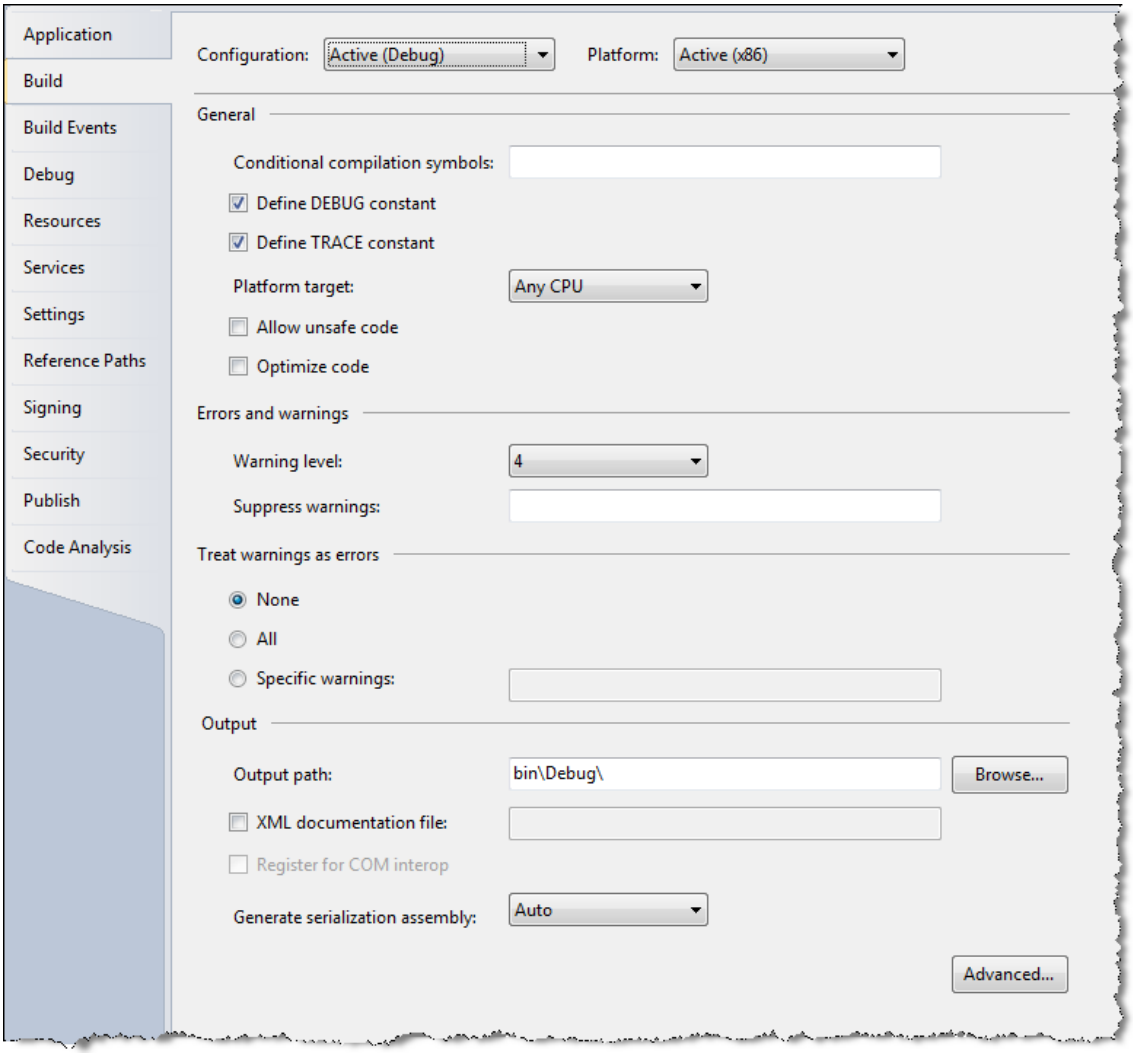

Abbildung 1.15: *Projekteinstellungen für den Build*

Feld	Beschreibung
Define DEBUG constant	Einstellung für die Zulassung der `DEBUG`-Sektionen im Code. • Einstellung in Debugkonfiguration: Ja • Einstellung in Releasekonfiguration: Nein Mehr dazu lesen Sie in Kapitel 2 und dort im Abschnitt »Präprozessoranweisungen«.
Define TRACE constant	Einstellung für die Kontrolle der .NET-Tracefunktionalität. Diese Einstellung kann sowohl in der Debug- als auch in der Releasekonfiguration aktiviert werden. Mehr dazu lesen Sie in Kapitel 4 und dort im Abschnitt »Den Ablauf einer Anwendung verfolgen«.
Platform target	Angabe, auf welcher Plattform der Code ablauffähig sein soll. Wählen Sie hier die Einstellung *Any CPU*, wenn .NET abhängig von der jeweiligen Maschine des Kunden automatisch entscheiden soll, ob Code als 32-Bit- oder als 64-Bit-Code ausgeführt

	werden soll.
Allow unsafe code	Angabe ob eingebettete Zugriffe auf nativen Code erlaubt sind oder nicht. Diese Einstellung ist nur in speziellen Fällen zu aktivieren.
Optimize code	Einstellung, die festlegt, ob der Compiler eine Codeoptimierungen vornehmen darf oder nicht. • Einstellung in Debugkonfiguration: Nein • Einstellung in Releasekonfiguration: Ja
Errors and warnings	Ich empfehle hier, die höchste Stufe zu verwenden und keine Warnungen zu unterdrücken. Es ist generell besser, Warnungen im Code direkt und nur punktuell zu unterdrücken (wenn überhaupt). Mehr dazu lesen Sie in Kapitel 2 und dort im Abschnitt »Präprozessoranweisungen«.
Treat warnings as errors	Die Standardeinstellung, keine Warnungen als Fehler zu behandeln, ist in den allermeisten Fällen zweckmässig.
Output path	Definieren Sie hier den Pfad, wo die erzeugten Resultate des Projekts abgelegt werden. Der angegeben Standardpfad *bin\Debug* respektive *bin\Release* ist relativ zum Projektverzeichnis zu verstehen und normalerweise so zu belassen.
XML documentation file	Mit dieser Angabe schalten Sie die Generierung einer Codedokumentation ein. Das Einschalten diese Angabe ist insbesondere bei Bibliotheksassemblys zu empfehlen. Mehr dazu entnehmen Sie Kapitel 2 und dort dem Abschnitt »Kommentierungen«.
Advanced…	Mit dem Dialogfeld *Advanced Build Settings* lassen sich zusätzlich Einstellungen des Build-Vorgangs verändern. Da diese Einstellungen extrem selten genutzt werden, verzichte ich an dieser Stelle auf eine weitere Erklärung.

Tabelle 1.11: *Wichtige Einstellungen des Dialogfelds* Build

Zusätzliche Schritte automatisieren

Mit Hilfe des Dialogfelds *Build Events* kann der Ablauf der Herstellung der Assembly (Buildvorgang) mit zusätzlichen Schritten ergänzt werden. Betrachten Sie dabei die beiden Textfelder *Pre-build event command line* und *Post-build event command line* als eigentliche Eingabezeilen eines Kommandofensters. Entsprechend können dort ein oder mehrere Zeilen von auszuführenden Befehlen definiert werden (jede Zeile ist ein abgeschlossener Befehl).

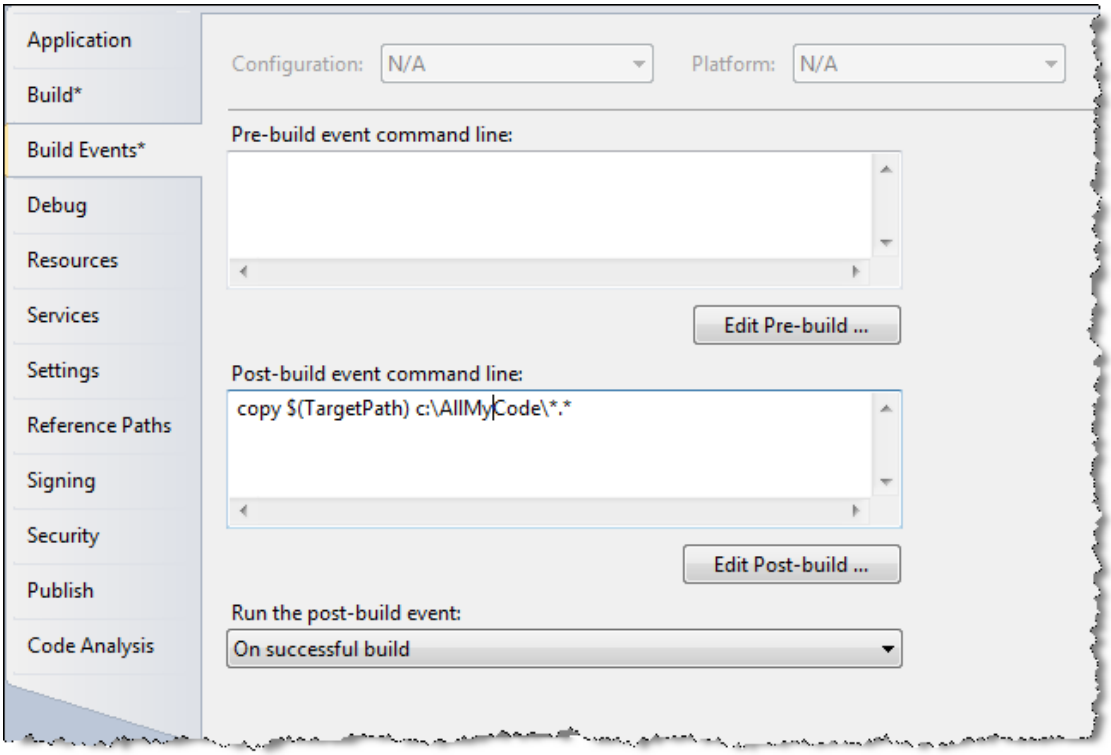

Abbildung 1.16: *Projekteinstellungen für die Automation*

Achten Sie darauf, dass die verwendeten Befehle korrekt geschrieben sind und im Rahmen des Prozesses, in dem Visual Studio selbst läuft, auch erreichbar sind. Maßgebend für das Auffinden eines Befehls ist die Variable *PATH* der Betriebssystemumgebung.

Über die Schaltflächen *Edit Pre-build* und *Edit Post-build* ist jeweils ein spezieller Editor erreichbar, der zusätzlich die Möglichkeit bietet, Makros in Bezug auf das Projekt zu verwenden. Das Beispiel in der **Abbildung 1.16** zeigt einen Kopierbefehl, der nach Beendigung des Builds durch Visual Studio die effektiv erzeugte Datei (*Makro $(TargetPath)*) in den Ordner *C:\AllMyCode* umkopiert.

Wichtig Beginn

Der aktuelle Pfad, der für den Ablauf der Befehlszeilen verwendet wird, ist der konfigurierte Pfad für die Ausgabe (siehe *Output path* **Abbildung 1.15**). Bei Verwendung der Debugkonfiguration lautet der Teilpfad relativ zum Projektverzeichnis *bin\Debug*.

Wichtig Ende

Einstellungen für das Ausführen des entwickelten Programms

Das Dialogfeld *Debug* stellt Einstellungen zur Verfügung, die beim Starten der Anwendung verwendet werden. Der Name *Debug* dieses Dialogfelds ist verwirrend, weil die vorgenommenen Einstellungen nicht abhängig sind vom Debugging, sondern von der gewählten Konfiguration. Besser wäre es somit, das Dialogfeld *Run* oder *Start Application* zu nennen.

Wie auch immer, die hier gemachten Angaben dienen dazu, den Start der Anwendung aus Visual Studio heraus zu beeinflussen.

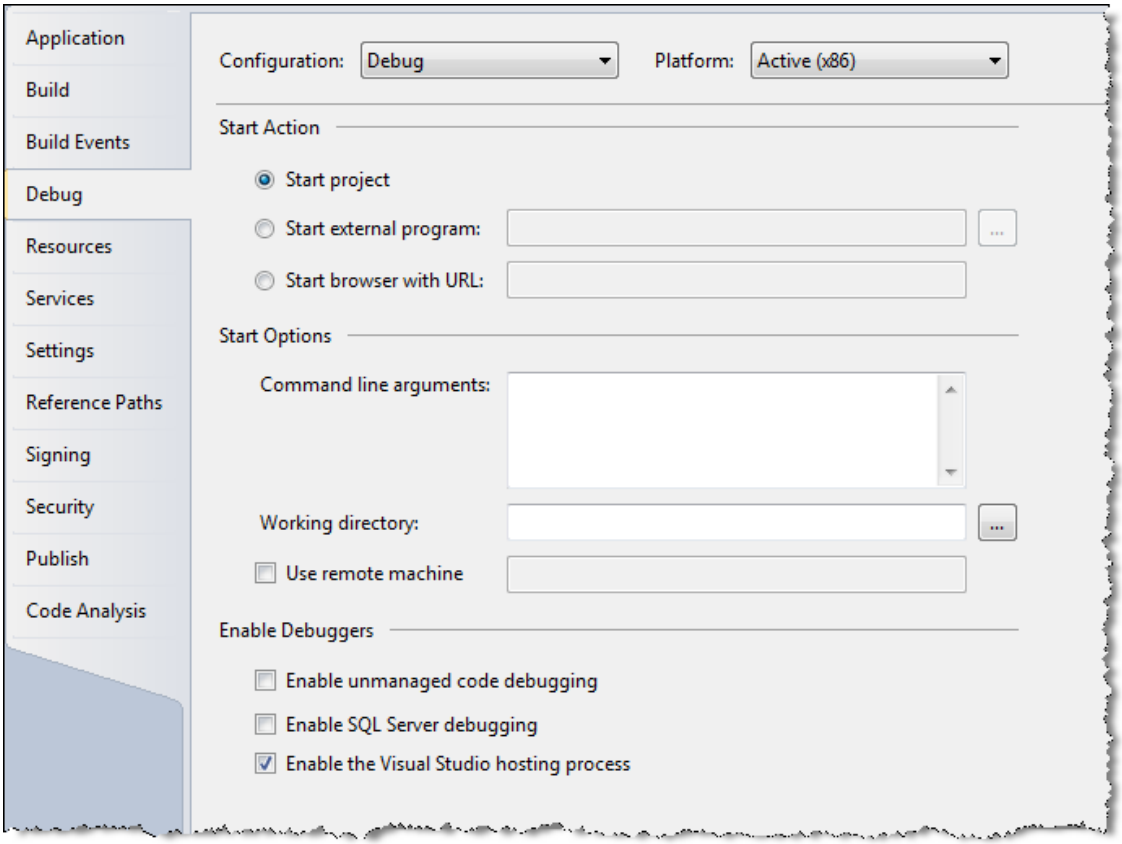

Abbildung 1.17: *Projekteinstellungen für das Debugging*

Feld	Beschreibung
Start project	Diese Einstellung definiert, dass beim Starten des Debuggingprozesses das aktuelle Projekt verwendet wird.
Start external program	In diesem Textfeld können Sie ein externes Programm angeben, das anstelle der mit dem Projekt erzeugten Assembly gestartet werden soll. Diese Angabe ist zum Beispiel praktisch, wenn Sie eine Bibliothek erstellen, diese aber mit einem andern Programm testen wollen. Bedenken Sie, dass beim Starten des Prozesses Visual Studio sich automatisch mit dem Debugger verbindet.
Start browser with URL	Analoge Angabe wie oben, allerdings zum Erreichen eines URLs. Diese Einstellung verwenden Sie insbesondere beim Entwickeln von Webanwendungen.
Command line arguments	In dieses Textfeld können Sie Befehlsargumente eingeben, die von Visual Studio an den zu startenden Prozess übergeben werden.
Working directory	In diesem Textfeld kann der Pfad definiert werden, der als Arbeitspfad an den Prozess übergeben wird. Wenn diese Angabe fehlt, wird der Pfad übernommen, an dessen Stelle die gestartete Anwendung gespeichert ist.
Use remote machine	Visual Studio unterstützt auch das Debugging von Prozessen auf anderen Maschinen. In diesem Fall muss dieses Kontrollkästchen eingeschaltet werden, und anschließend im frei geschalteten Textfeld der Name der Maschine definiert werden, auf der das Debugging durchgeführt werden soll.

Enable unmanaged code debugging	Schalten Sie dieses Kontrollkästchen ein, wenn Sie nativen Code auch debuggen wollen
Enable SQL Server debugging	Schalten Sie dieses Kontrollkästchen ein, wenn Sie SQL-Prozeduren debuggen wollen. Achtung: Diese Option wird nicht in allen Sprachen und Umgebungen unterstützt.
Enable the Visual Studio hosting process	Schalten Sie dieses Kontrollkästchen ein, wenn Sie in Visual Studio Gebrauch von der Optimierung für das Debugging machen wollen (Verwendung der Datei *<Project>.vshost.exe*).

Tabelle 1.12: *Wichtige Einstellungen des Dialogfelds* Debug

Ressourcen definieren

.NET unterstützt die zentralisierte Nutzung von Ressourcen einer Anwendung. Ressourcen sind Datenelemente wie Texte, Bilder, Sounds oder andere beliebige Daten. Über das Dialogfeld *Resources* in den Projekteinstellungen kann im Projekt ein Ressourcencontainer generiert und anschließend verwaltet werden.

Der Umgang mit Ressourcen in einem Projekt ist in Kapitel 9, »Umgang mit Ressourcen«, beschrieben.

Client Application Services

Die Client Application Services stellen spezielle Funktionalitäten für Authentifizierung und Verwendung von benutzerspezifischen Profilen zur Verfügung. Die Nutzung dieser Funktionalitäten ist ursprünglich für die Verwendung von Webanwendungen mit ASP.NET gedacht gewesen. Seit einigen .NET-Generationen sind nun diese Funktionalitäten auch für andere Anwendungstypen zugänglich und können dabei in dem entsprechenden Dialogfeld konfiguriert werden.

Die Verwendung der Client Application Services ist in Band 2 dieses Buches erklärt.

Einstellungsdatei der Anwendung nutzen

.NET unterstützt die zentralisierte Nutzung von Einstellungen einer Anwendung. Als Einstellungen sind beliebige Datenelemente irgendwelcher verwendeter Klassen nutzbar (zum Beispiel die Koordinaten der letzten Position des Anwendungsfensters auf dem Desktop). Über das Dialogfeld *Settings* in den Projekteinstellungen kann im Projekt ein Einstellungscontainer generiert und anschließend verwaltet werden.

Der Umgang mit Einstellungen in einem Projekt ist in Kapitel 4 und dort im Abschnitt »Anwendungen konfigurierbar gestalten« erklärt.

Zusätzliche Pfade für benutzte Assemblys einstellen

Visual Studio stellt in einem Projekt mit dem Ordner *References* die Möglichkeit zur Verfügung, mitbenutzte Assemblys in einer Anwendung zu definieren (siehe auch **Abbildung 1.11**). Zusätzlich können im Dialogfeld *Reference Paths* Pfade definiert werden, die von Visual Studio bei der Herstellung der Assembly benutzt werden, um die referenzierten Assemblys aufzufinden.

Ich empfehle von dieser Möglichkeit keinen Gebrauch zu machen, da diese Pfade hinsichtlich der Frage, welche Assembly nun tatsächlich benutzt wird, durchaus für Verwirrung sorgen können, da eine Assembly mehrfach auf einem System gespeichert ist.

Signierung benutzen

Die Seite *Signing* dient zur Definition der in diesem Projekt zu verwendenden Signatur. Signaturen diesen zur Erhöhung der Sicherheit bei der Benutzung der erstellten Assembly. Die Signierung von Assemblys und die daraus entstehenden Konsequenzen sind so wichtig, dass ich in diesem Kapitel extra

einen entsprechenden Abschnitt erstellt habe (siehe »Eine Assembly mit einem starken Namen versehen«).

Sicherheit und Verteilung konfigurieren

Sowohl das Dialogfeld *Security* als auch das Dialogfeld *Publish* erlaubt Definitionen im Zusammenhang mit dem Verteilmechanismus von Anwendungen namens ClickOnce. ClickOnce ist im dritten Band dieses Buches beschrieben.

Code-Analyse nutzen

Das Dialogfeld *Code Analysis* stellt Ihnen eine Möglichkeit zur Verfügung, Ihren Code nicht nur durch die Syntaxprüfung des C#-Compilers prüfen zu lassen. Sie können ihn auch weiteren, von Microsoft festgelegten Prüfungen unterziehen. Dabei ist es möglich, für die zusätzlichen Prüfungen verschieden streng agierende Regelwerke zu referenzieren.

Profitipp Beginn

Ein Entwicklungsteam sollte sich am Anfang eines Projekts zu allen möglichen Einstellungen und zur Organisation der Visual Studio-Solutions und -Projekte Gedanken machen, und sich auf die gemeinsame Verwendung von Einstellungen einigen. Damit soll erreicht werden, dass alle Entwickler und Entwicklerinnen mit der gleichen Philosophie und den gleichen Einstellungen arbeiten. Nur so ist es möglich, dass Sie eine gemeinsame Qualitätsstufe erreichen, und das Projekt selbst dann erfolgreich weitergeht, wenn Personalfluktuation auftritt.

Ich empfehle zur gemeinsamen Regelung:

- Festlegen der Inhalte der Registerkarten *Application*, *Build*, *Build Events*, *Resources*, *Settings*, *Reference Paths*, *Signing* und *Code Analysis*
- Festlegen der Architektur. Dies hat Einfluss auf die Einstellungen von *Services*, *Security* und *Publish*

Profitipp Ende

Code verändern

Die Erstellung und Veränderung von Code ist im Allgemeinen die Tätigkeit, mit der wir Entwickler die meiste Zeit verbringen. Ich empfehle Ihnen deshalb, sich mit den Eigenheiten des Editierens von Code besonders vertraut zu machen. Hier erleichtern Sie sich nicht nur Ihre tägliche Arbeit, sondern beeinflussen auch die Wirtschaftlichkeit Ihres Projekts positiv.

Hinweis Beginn

Stellen Sie sicher, dass die automatische Formatierung der Codeeditoren Ihren Vorstellungen entspricht. Sie finden diese Einstellungen im Menü unter *Tools/Options*. Im anschließend erscheinenden Dialogfeld *Options* navigieren Sie in der Liste links zum Eintrag *Text Editor/C#/Formatting*. Erweitern Sie den gefundenen Eintrag und nehmen Sie im Dialogfeld rechts die gewünschten Einstellungen vor.

Hinweis Ende

Wir haben bereits ein Projekt erstellt (siehe **Abbildung 1.10**). Schließen Sie das etwaig offene Dialogfeld für die Projekteinstellungen. Sollten Sie nun den Code nicht mehr im Editor sehen, öffnen Sie die Datei *Program.cs* durch einen Doppelklick im *Solution-Explorer* um den Code gemäß **Listing 1.1** zu editieren. Anschließend können Sie das Programm mit der Tastenkombination (Strg)+(F5) starten.

```
namespace LearningBasics {
    class Program {
        static void Main(string[] args) {
            Console.WriteLine("Handbuch der .NET 4.0 Programmierung.");
```

```
            WriteExecutionTime();
        }

        static string WriteExecutionTime(){
            string strResult = string.Format("Das Programm wurde am {0:d} um {0:T} ausgeführt", DateTime.Now);
            Console.WriteLine(strResult);
            return strResult;
        }
    }
}
```

Listing 1.1: *Unser erstes .NET-Programm zum Kennenlernen von Visual Studio*

Hinweis Beginn

Der Code beinhaltet zunächst eine Klasse mit Namen `Program` innerhalb eines Namensraums `LearningBasics`. Der Namensraum wurde vom Assistenten aus dem Projektnamen abgeleitet. Des Weiteren ist in der Klasse `Program` eine statische Methode `Main()` untergebracht. Diese Methode bildet die Startmethode zur Ausführung des Programms. Die Methode `Main()` wird vom Loader der CLR im Programm lokalisiert und als Startmethode angesprochen. Entsprechend soll ein Programm die Methode `Main()` genau einmal enthalten.

Für weitergehende Erklärungen des C#-Codes verweise ich auf das Kapitel 2.

Hinweis Ende

Die wichtigste Unterstützung von Visual Studio für die Codierung ist in Form von IntelliSense in verschiedene Editoren eingebaut. IntelliSense hat die Aufgabe, das bereits Geschriebene zu analysieren und sinnvoll zu einem Ganzen zu ergänzen, respektive eine Auswahl anzuzeigen, die den bereits geschriebenen Text sinnvoll vervollständigt.

Die **Abbildung 1.18** zeigt IntelliSense im Einsatz. Dabei wird die Klasse `Console` mit einer Methode, deren Namen mit den Zeichen `Wri` beginnt, benutzt. IntelliSense hat in der laufenden Analyse aus allen zur Verfügung stehenden Möglichkeiten die zwei Möglichkeiten `Write()` und `WriteLine()` herausgefiltert und schlägt diese in Form einer eingeblendeten Liste unterhalb des zu editierenden Texts vor.

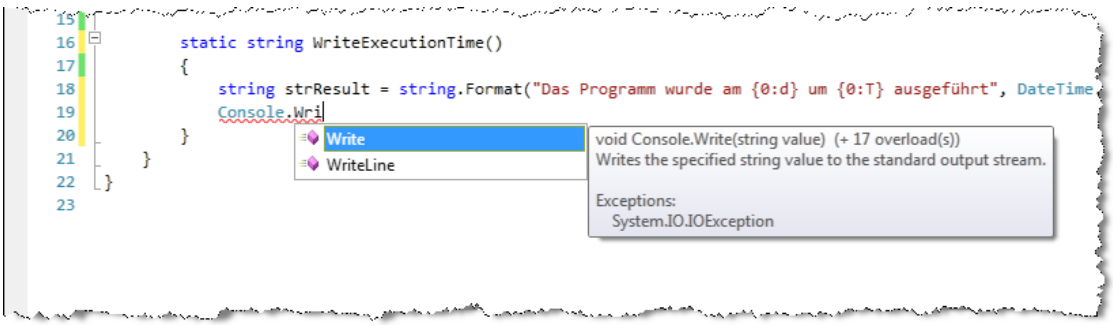

Abbildung 1.18: *IntelliSense von Visual Studio im Einsatz*

Die Auswahl wird nach der Selektion in der Liste durch das logisch nächste Zeichen vervollständigt. Um die Auswahl in der IntelliSense-Liste zu verändern, benutzen Sie die Tasten (Pfeil oben) oder (Pfeil unten). In unserem Fall handelt es sich in beiden Fällen um einen Methodenaufruf, weshalb die öffnende Klammer den Text vervollständigt, und IntelliSense in die nächste Eingabephase überführt wird.

Die zweite Eingabephase von IntelliSense besteht darin, dass die zur Verfügung stehenden Varianten des Methodenaufrufs angezeigt werden. Sofern von der ausgewählten Methode mehrere Varianten existieren, können Sie wiederum mit den Tasten (Pfeil oben) oder (Pfeil unten) Ihre Wahl treffen und den Inhalt vervollständigen.

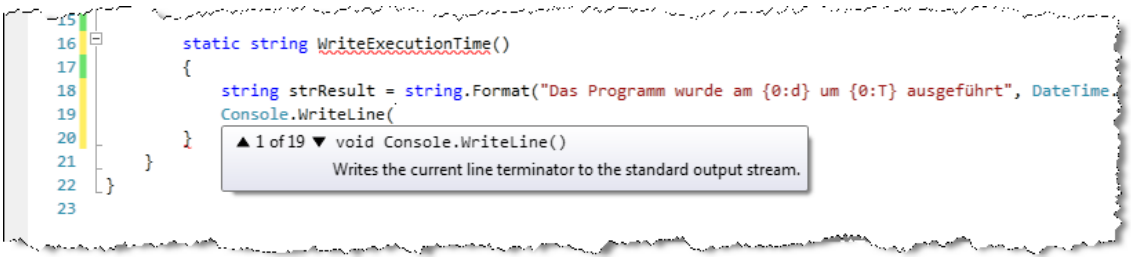

Abbildung 1.19: *IntelliSense zeigt die Methodenüberladung*

Profitipp Beginn

IntelliSense in Visual Studio 2010 wurde dahingehend optimiert, dass bei der Filterung der gezeigten Auswahl die Eingabe nach dem Prinzip der Volltextsuche verwendet wird. **Abbildung 1.20** zeigt ein entsprechendes Beispiel. Die Klasse `Console` soll mit einer Methode benutzt werden, in deren Namen die Buchstabenfolge `line` vorkommen soll.

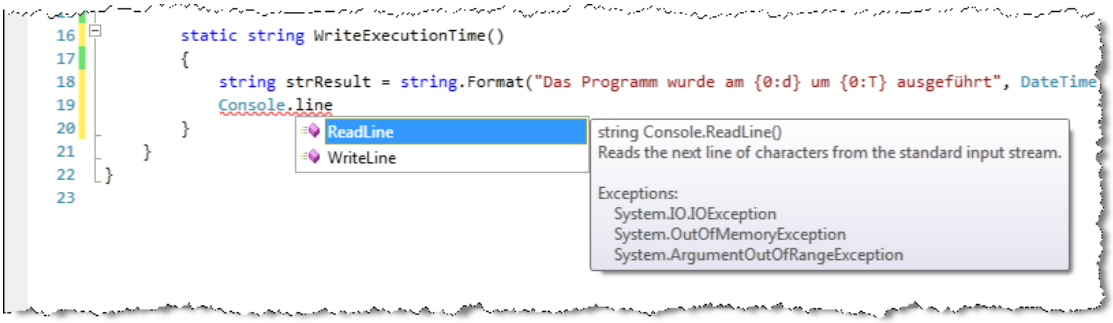

Abbildung 1.20: *IntelliSense arbeitet neu mit Volltextsuche*

Es kommt immer mal wieder vor, dass IntelliSense unabsichtlich zugeklappt wird. Merken Sie sich für diesen Fall die Tastenkombination (Strg)+(j). Mit dieser Tastenkombination können Sie IntelliSense an der aktuellen Cursorposition wieder aktivieren.

Profitipp Ende

Code mit Hilfe von Codierungsunterstützung erstellen

Eine weitere Hilfestellung bietet der Code-Editor durch die aktive Bereitstellung von Codierungsunterstützungen (engl. Code Snippets). Codierungsunterstützungen sind kleinste Codegeneratoren, die während des Editierens mit Befehlen angestoßen werden können.

Ausgangslage der Nutzung dieser Funktionalität ist wiederum IntelliSense. Beginnen Sie mit der Eingabe des Namens der Codierungsunterstützung und aktivieren Sie die gefundene Unterstützung durch zweimaliges betätigen der Taste (Tab). Je nach Wahl der Unterstützung werden eine oder mehrere Zeilen Code durch die Unterstützung hergestellt.

In der Auswahlliste von IntelliSense erkennen Sie die Codierungsunterstützung am entsprechenden Symbol (siehe **Abbildung 1.21**). Nehmen Sie sich die Zeit und studieren Sie die Möglichkeiten von IntelliSense. Das zahlt sich aus!

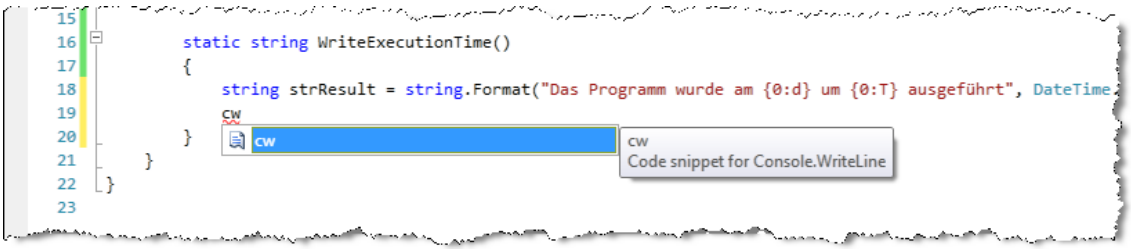

Abbildung 1.21: *Auswahl der Codierungsunterstützung* cw *(*Console.WriteLine()*)*

Code ausführen

Wie für viele andere Funktionen in Visual Studio, ist das Ausführen von Code auf unterschiedliche Weise möglich:

- durch Benutzung des Menüs
- durch Benutzung der Symbolleiste
- durch Benutzen von Tastenkürzeln

Abbildung 1.22 illustriert die verschiedenen Möglichkeiten: Starten der Anwendung über die Symbolleiste (Nummer 1) und Starten über das Menü *Debug* (links von Nummer 2), sowie dem zugewiesenen Tastenkürzel (rechts von der Nummer 2).

Hinweis Beginn

Um **Abbildung 1.22** herstellen zu können habe ich an der Standardinstallation folgende zwei Veränderungen vorgenommen:

- Einschalten des Symbolleiste *Debug*
- Ergänzen des Symbolleiste *Debug* mit der Schaltfläche *Start without Debugging*

Wir lernen daraus, dass wir das Menüsystem und die Symbolleisten gemäß unseren Bedürfnissen frei anpassen können. Die notwendige Unterstützung dazu erreichen Sie über das Kontextmenü im Bereich der Symbolleisten und des Menüs (Klicken Sie dazu rechts im Fenster auf einer freien Position).

Jedes Mal wenn Sie Code ausführen, kompiliert Visual Studio automatisch die veränderten Teile Ihrer Solution. Eine explizite Herstellung kann über das Menü *Build/Build Solution* oder *Build/Rebuild Solution* jederzeit erreicht werden. Wird der Befehl *Build/Rebuild Solution* angewählt, wird ungeachtet der vorhandenen Veränderungen der gesamte Inhalt der Solution kompiliert.

Hinweis Ende

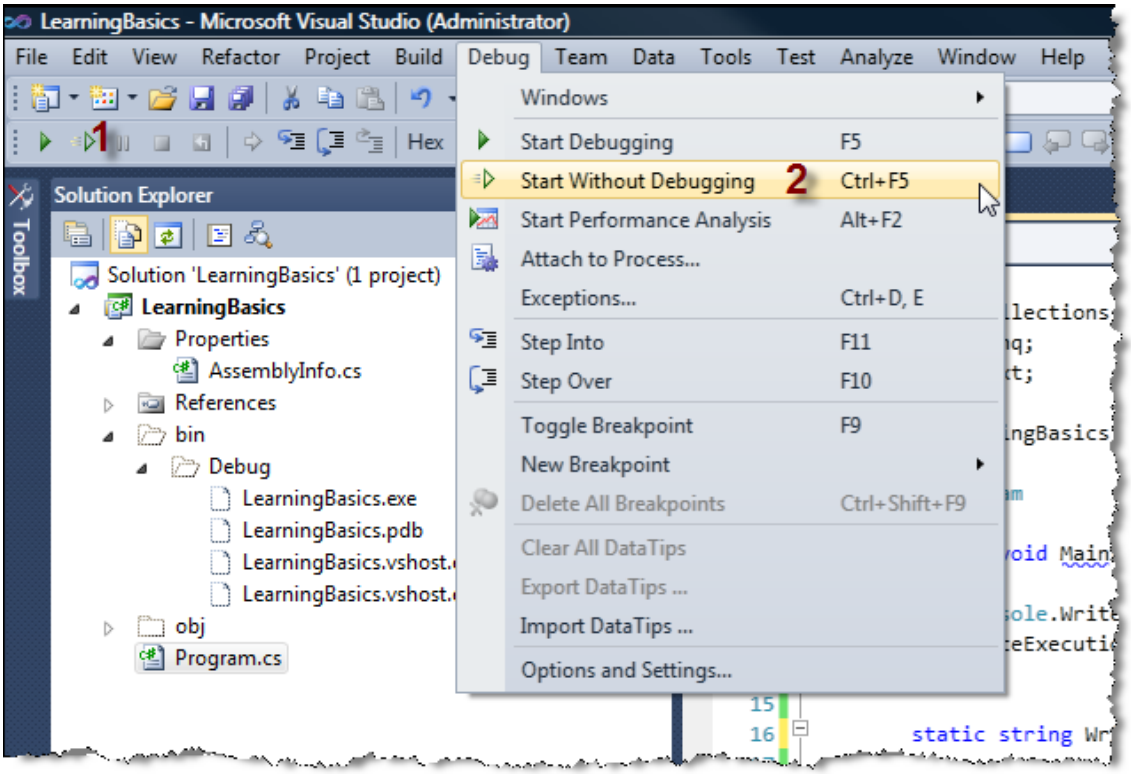

Abbildung 1.22: *Möglichkeiten des Starts der Ausführung eines Programms*

Die Codeausführung bezieht sich immer auf das zum Starten gekennzeichnete Projekt in der Visual Studio-Lösung. Zur besseren Erkennung ist das entsprechende Projekt in Fettschrift geschrieben. Bei mehreren Projekten können Sie die Kennzeichnung über das Kontextmenü des Projekts erreichen. Beachten Sie, dass zwar alle Projekttypen gekennzeichnet werden können, aber nur Projekte, die eine *.exe*-Datei erzeugen, auch tatsächlich gestartet werden können.

Das Starten der Anwendung ist ohne und mit Debugging möglich. In der Version ohne Debugging erfolgt der Start der Anwendung in einem von Visual Studio unabhängigen Prozessraum. Die Anwendung läuft so wie sie später beim Benutzer auch ablaufen wird. Beim Starten mit Debugging wird der Prozess mit dem Namen *<Projektname>.vshost.exe* gestartet. Hierbei handelt es sich um eine spezielle ausführbare Datei, die von *Visual Studio* automatisch im Projekt erzeugt wird. Mit Hilfe dieser Datei wird der Startvorgang für das Debugging beschleunigt, indem die Verbindung zwischen dem Debugger von Visual Studio und der Anwendung im Hintergrund erfolgt. Die Verwendung der speziellen *vshost.exe*-Datei kann in den Projekteinstellungen ausgeschaltet werden (siehe **Tabelle 1.12**).

Eine Anwendung entwanzen (debuggen)

Visual Studio stellt für das Debugging zwei grundsätzliche Verfahren zur Verfügung. Die Auswahl des geeigneten Verfahrens hängt von der jeweiligen Situation der Entwicklung ab.

Debugging ohne Haltepunkt

Bei Anwendung des ersten Verfahrens starten wir das Programm und arbeiten damit bis ein Fehler auftritt. Mit den Standardeinstellungen hält Visual Studio die Codeausführung automatisch an, sobald ein Fehler auftritt. Dieser wird in einem speziellen Ausnahmefenster visualisiert (siehe **Abbildung**

1.23). Der Nachteil dieser Methode ist, dass die Entstehung des Fehlers nicht verfolgt werden kann, da das System erst anhält, wenn der Fehler aufgetreten ist. Die Schwierigkeit dabei ist manchmal, dass der Fehler nur eine Folge einer falschen Entscheidung ist, die dutzende Zeilen zuvor gefällt wurde.

Visual Studio zeigt mit dem Pfeil (Nummer 1 in **Abbildung 1.23**) die aktuelle Zeile, in der der Fehler festgestellt wurde. Das Ausnahmefenster wird mit dem Typ der Ausnahme und einem Zusatztext übertitelt (Nummer 2 in **Abbildung 1.23**) und ein allgemeiner Tipp für die Fehlerbehebung wird ausgegeben (Nummer 3 in **Abbildung 1.23**). Am unteren Rand des Dialogfelds werden zudem Aktionen angeboten (Nummer 4 in **Abbildung 1.23**). Mit der Aktion *View Details* wird das interaktive Dialogfeld für die Analyse einer Ausnahme angezeigt (siehe **Abbildung 1.24**).

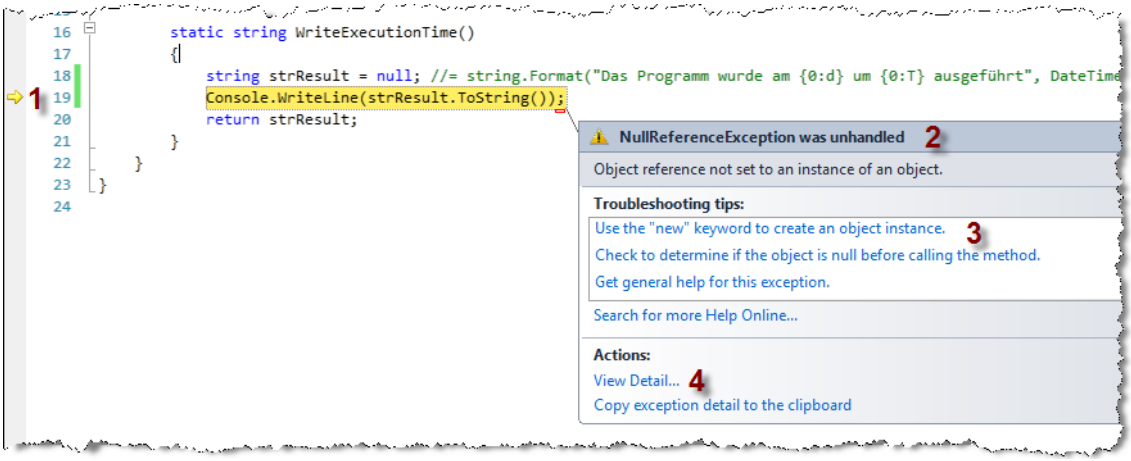

Abbildung 1.23: *Ausnahme-Fenster in Visual Studio während des Debuggens*

Mit Hilfe des interaktiven Ausnahmedialogfelds können Sie die Inhalte der Ausnahme im Detail studieren und dabei auch die aktuelle Aufrufliste (engl. call stack) betrachten. In der **Abbildung 1.24** zeige ich die Aufrufliste. Die Quickinfo wurde dazu mit der Maus aktiviert.

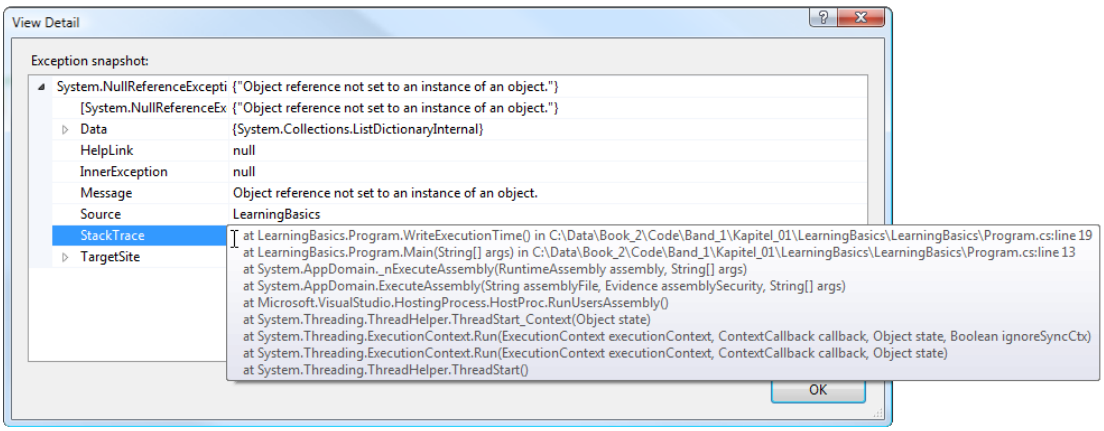

Abbildung 1.24: *Detailfenster einer Ausnahme*

Profitipp Beginn

Vergleichen Sie die Aufrufliste in der **Abbildung 1.24** mit der nachstehenden Liste, die durch die Ausführung ohne Debugging entstanden ist. Der Einfluss des Starts über Visual Studio ist daran erkennbar, dass die Methode `Main()` des Programms an achter Stelle steht, wenn das Debugging startet, und an erster Stelle, wenn ohne Debugging gestartet wird.

```
Handbuch der .NET-Programmierung.
Unhandled Exception: System.NullReferenceException: Object reference not set to an instance of an object.
   at LearningBasics.Program.WriteExecutionTime() in
      C:\Data\Book_2\Code\Band_1\Kapitel_01\LearningBasics\LearningBasics\Program.cs:line 19
   at LearningBasics.Program.Main(String[] args) in
      C:\Data\Book_2\Code\Band_1\Kapitel_01\LearningBasics\LearningBasics\Program.cs:line 13
```

Listing 1.2: *Aufrufliste des gleichen Fehlers bei der Ausführung ohne Debugging*

Profitipp Ende

Debugging mit Haltepunkt

Das zweite Verfahren arbeitet mit Haltepunkten. Bei diesem Verfahren gehen wir davon aus, dass wir ahnen, wo sich der Fehler befindet. Wir sind deshalb in der Lage, an geeigneter Position vor dem Fehler einen Haltepunkt zu setzen. Einen Haltpunkt können Sie in der gewünschten Zeile mittels Tastenkombination (F9) oder durch einen Klick im grauen Band ganz links im Editor ein-, respektive durch wiederholtes Betätigen ausschalten. Anschließend starten wir die Anwendung wiederum als Debuggingsitzung und bedienen die Anwendung, bis der Haltepunkt erreicht wird.

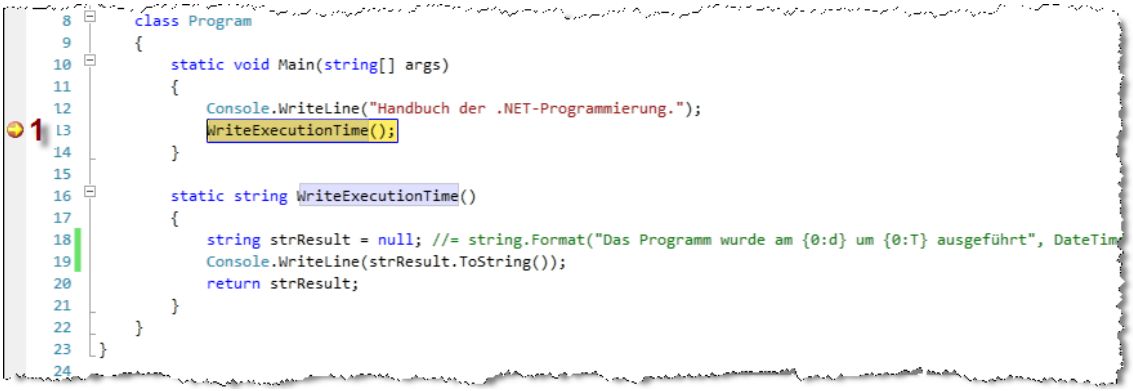

Abbildung 1.25: *Der Haltepunkt in Zeile 13 wurde erreicht*

Ist ein Haltepunkt erreicht, können verschiedene Aktionen vorgenommen werden. Zum einen kann der Code vor dem Haltepunkt noch einmal ausgeführt werden, indem Sie den gelben Pfeil mit der Maus auf eine bereits ausgeführte Stelle oberhalb des Haltepunktes verschieben. Umgekehrt kann die Zeile und weitere Zeilen darunter in der Ausführung übersprungen werden, indem Sie den gelben Pfeil mit der Maus auf eine Stelle unterhalb des Haltepunktes verschieben.

Wichtig Beginn

Beim erneuten Ausführen von Code oder beim Überspringen von Code mit dem Debugger muss beachtet werden, dass der Code nicht mehr exakt die gedachte Funktionalität erfüllt. Wiederholung heißt ein zweites Mal ausführen. Das kann Einfluss auf Programmvariablen haben, deren Werte wiederholt verändert werden. Umgekehrt kann das Überspringen von Code Variablen nicht oder falsch zuweisen, was wiederum ein Problem darstellen kann.

Wichtig Ende

Die häufigste Aktion, die Sie an dieser Stelle jedoch benutzen werden, ist die normale schrittweise Ausführung des Codes mit Hilfe der Debugging-Navigation:

- Mit der Taste (F10) einen Schritt ausführen und dabei nicht in die Methode springen
- Mit der Taste (F11) einen Schritt ausführen und dabei in die Methode springen
- Mit der Tastenkombination (Umschalt)+(F11) die aktuelle Methode verlassen und nach Rückkehr in

der aufrufenden Methode die Ausführung wieder anhalten

- Den Cursor an eine Stelle weiter unten im Code setzen und dann den Code bis zu dieser Stelle mit der Tastenkombination (Strg)+(F10) ausführen

Inhalte eigener Daten beim Debuggen ansehen und verändern

Eine zweifelsfrei praktische und genauso wichtige Funktionalität bietet der Debugger mit diversen unterstützenden Fenstern für die Kontrolle und Veränderung von Dateninhalten zur Laufzeit. Im Detail sind die Möglichkeiten gemäß **Abbildung 1.26** und dazugehörender Legende in **Tabelle 1.14** vorhanden.

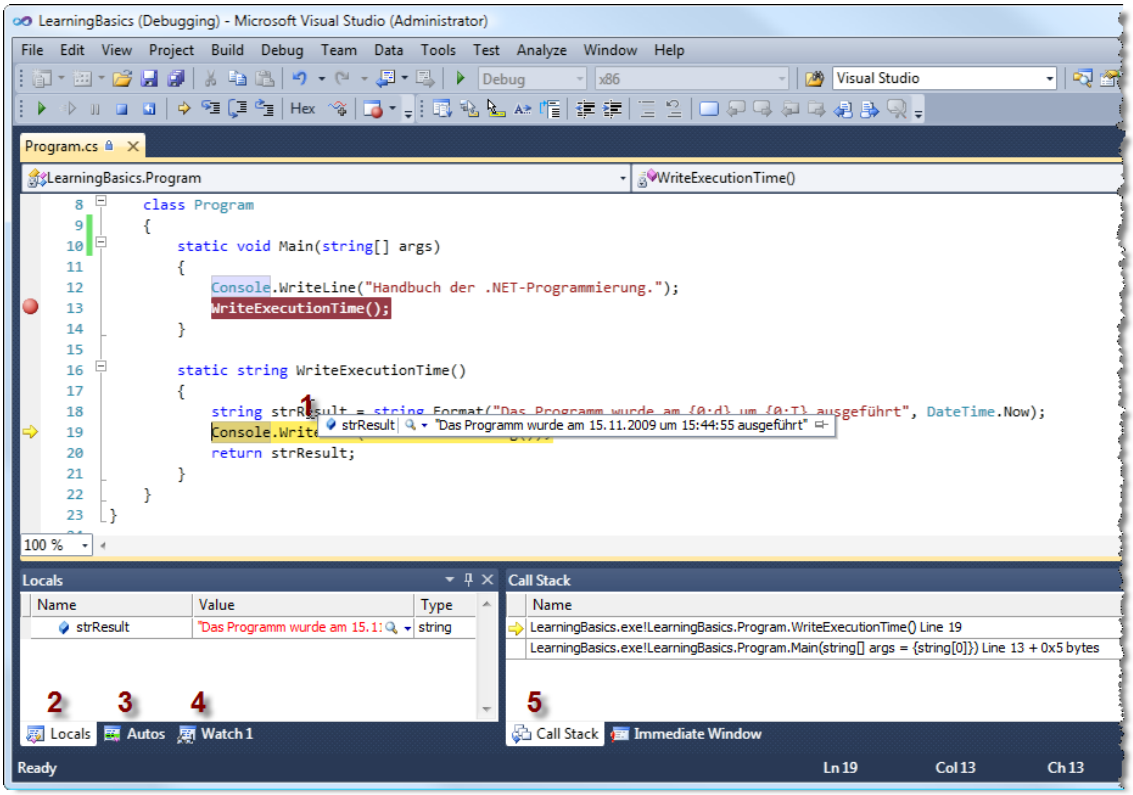

Abbildung 1.26: *Wichtige unterstützende Elemente für das Debugging*

#	Benennung	Inhalt
1	Quickinfo	Durch das Bewegen auf ein Datenelement (hier die Stringvariable), wird der Inhalt des Elements in einer Quickinfo angezeigt. Diese kann mit dem Pinnwandknopf im Debuggingfenster rechts vom Ausdruck fixiert werden, so dass die Info in der nächsten Debuggingsitzung wieder sichtbar ist.
2	*Locals*	Die Registerkarte *Locals* zeigt die lokalen Datenelemente der Methode, in der sich die Codeausführung gerade befindet. In diesem Fenster können Sie die Daten eines Elementes zur Laufzeit anpassen.
3	*Autos*	Die Registerkarte *Autos* zeigt alle Daten der aktuellen Zeile und der Zeile davor. In diesem Fenster können Sie die Daten eines Elementes zur Laufzeit anpassen.
4	*Watch 1*	Die Registerkarte *Watch 1* zeigt die Daten, die Sie manuell in dieses Fenster konfiguriert

		haben. Diese Konfiguration wird von einer Debuggingsitzung zur nächsten beibehalten. Die Inhalte können entweder direkt aus dem Coder heraus mit Drag & Drop, direkt im Code über das Kontextmenü des Elements, oder im Fenster selbst durch manuelles Schreiben eingebracht werden. In diesem Fenster können Sie die Daten eines Elementes zur Laufzeit anpassen.
5	*Call Stack*	Dieses Fenster zeigt die jeweilige Aufrufliste während des Debuggings.

Tabelle 1.13: *Erklärungen zu unterstützenden Elementen für das Debugging gemäß* ***Abbildung 1.26***

Dynamische Evaluation von Ausdrücken

Während der Debugging-Sitzung kann es wichtig sein, einen Ausdruck eines Elements zu prüfen, ohne das Programm anzuhalten. Zu diesem Zweck kann das Dialogfeld *QuickWatch* mit der Tastenkombination (Umschalt)+(F9) angefordert werden. Ist dabei ein Datenelement markiert, wird dieses automatisch in das Textfeld *Expression* übernommen, dessen Inhalt evaluiert und das Resultat in der Liste des Dialogfelds eingetragen. Die **Abbildung 1.27** zeigt dieses Vorgehen für unsere lokale Stringvariable aus der Methode `WriteExecutionTime()`. Sie können die Inhalte übrigens auch in diesem Fenster verändern.

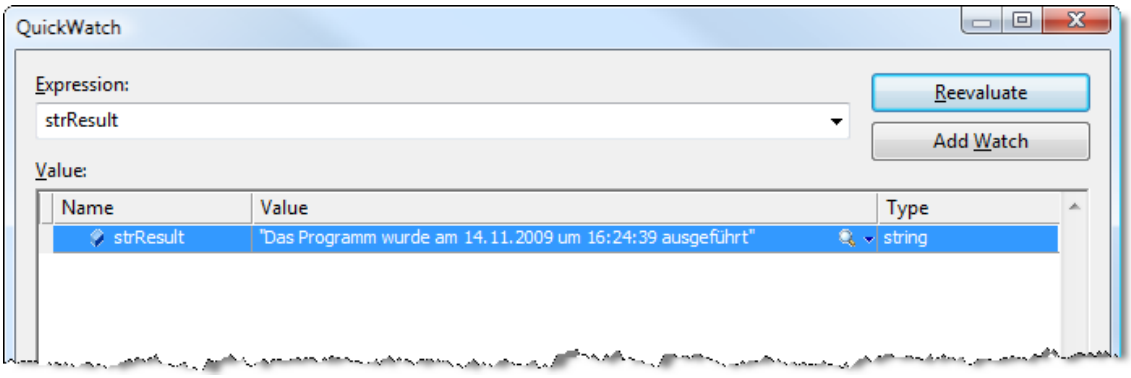

Abbildung 1.27: *Mit Hilfe der Schnellansicht kann ein Objekt des Programms analysiert werden*

Neben der x-ten Möglichkeit eigene Objekte zu betrachten und deren Inhalt während der Laufzeit zu verändern, können Sie im Textfeld des Dialogfelds *QuickWatch* beliebige Ausdrücke eingeben. Dabei unterstützt Sie IntelliSense im gewohnten Rahmen. Nach der Eingabe betätigen Sie die Schaltfläche *Reevaluate*, um das Resultat des Ausdrucks zu evaluieren und in der Liste anzuzeigen.

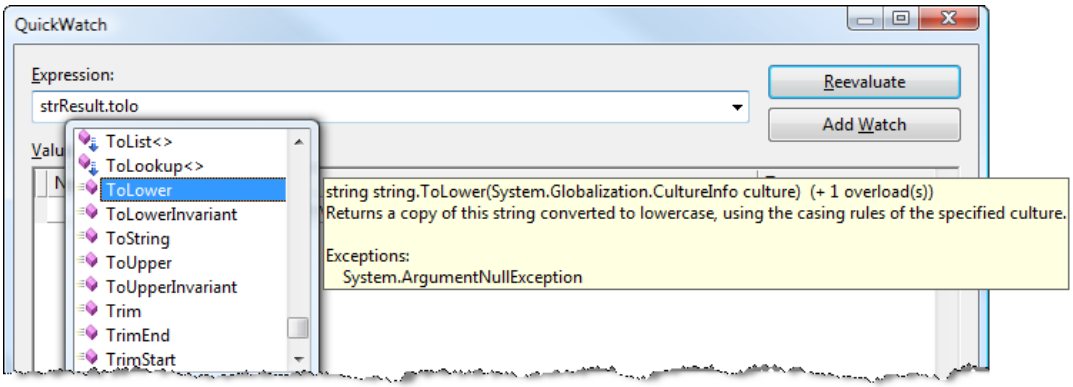

Abbildung 1.28: *Mit Hilfe der Schnellansicht können dynamisch Ausdrücke evaluiert werden*

Tipp Beginn

Ausdrücke können auch im Fenster *Watch 1* gemäß **Abbildung 1.26** eingegeben und evaluiert werden. Ich bin allerdings der Meinung, dass dort die Eingaben eine Hakelei sind, weil die Evaluation des Ausdrucks automatisch bei einer unfertigen Eingabe erfolgen kann und dementsprechend Fehler auftreten, respektive die Eingabe erneut editiert werden muss.

Tipp Ende

Abbrechen einer Debugging-Sitzung

Ist die Fehleranalyse soweit fortgeschritten, dass der Debugger verlassen werden kann, wird die Ausführung des Programms oft abgebrochen. Dazu stellt *Visual Studio* in der Symbolleiste die Schaltfläche *Stop Debugging*, respektive die Tastenkombination (Umschalt)+(F5) zur Verfügung.

Tastenkürzel

Durch die Nutzung von Tastenkürzeln können Sie die Effizienz in der Bedienung von *Visual Studio* entscheidend steigern. Viele Tastenkürzel sind vom Fokus innerhalb der Anwendung abhängig, da Sie aber die meiste Zeit im Code-Editor verbringen, habe ich mich beim Zusammenstellen der **Tabelle 1.14** auf die Kürzel dafür konzentriert. Beachten Sie, dass diese Angaben für das *Profil Visual C# 2005* gilt. Zudem habe ich die Trivialnavigation mit den Pfeiltasten, Tasten für seitenweise Navigation und weitere Tastenkombinationen wie (Strg)+(Ende) als allgemein bekannt vorausgesetzt und demzufolge nicht aufgeführt.

Bereich	Bedeutung	Tastenkombination
Code herstellen/ausführen	Erstellen der gesamten Lösung	(Strg)+(Umschalt)+(B)
	Code kompilieren	(F6)
	Code ohne Debugging ausführen	(Umschalt)+(F5)
	Code mit Debugging ausführen	(F5)
	Ausführung im Debugger stoppen	(Umschalt)+(F5)
	Haltepunkt setzen/entfernen	(F9)
	Debugging: Schritt über eine Methode ausführen	(F10)
	Debugging: Schritt bis zur Mausposition ausführen	(Strg)+(F10)
	Debugging: Schritt in eine Methode hinein	(F11)
	Debugging: Schritt aus einer Methode heraus	(Umschalt)+(F11)
	Debugging: Schnellüberwachung aktivieren	(Umschalt)+(F9)
Editor	Text im Fenster auf oder abwärts schieben. Der Cursor bleibt dabei in der aktuellen Zeile.	(Strg)+(Pfeil oben) (Strg)+(Pfeil unten)
	Suchen in der Lösung	(Strg)+(f)
	Suchen in Dateien	(Strg)+ (Umschalt)+(f)
	Wiederholung der letzten Suche nach unten	(F3)
	Wiederholung der letzten Suche nach oben	(Umschalt)+(F3)
	Suchen/Ersetzen in der Lösung	(Strg)+(h)
	Suchen/Ersetzen in Dateien	(Strg)+(Umschalt)+(H)

	Gesamten Code markieren	(Strg)+(a)
	Code zwischen Klammern auswählen	(Strg)+(Umschalt)+(^)
	Markierung nach Kleinschrift umwandeln	(Strg)+(u)
	Markierung nach Großschrift umwandeln	(Strg)+(Umschalt)+(U)
	Markierung formatieren	(Strg)+(k), (Strg)+(f)
	Lesezeichen setzen/löschen	(Strg)+(b), (Strg)+(t)
	Zum nächsten Lesezeichen springen	(Strg)+(b), (Strg)+(n)
	Zum vorangehenden Lesezeichen springen	(Strg)+(b), (Strg)+(p)
	Alle Lesezeichen löschen	(Strg)+(b), (Strg)+(c)
	Coderegionen erweitern/reduzieren	(Strg)+(m), (Strg)+(l)
	Springe zur Definition	(F12)
	Umbenennen	(F2)
	Elemente einer Klasse anzeigen	(Strg)+(j)
	Markierung als Kommentar definieren	(Strg)+(k), (Strg)+(c)
	Kommentar bei Markierung entfernen	(Strg)+(k), (Strg)+(u)
	Smarttag aktivieren	(Umschalt)+(Alt)+(F 10) (Strg)+(.)

Tabelle 1.14: *Wichtige Tastenkürzel in Visual Studio für das Profil* Visual C# 2005

Die Lösung mit einer eigenen Bibliothek ergänzen

Nachdem wir uns den grundlegenden Umgang mit Visual Studio für die Erstellung einer Anwendung angeeignet haben, können wir diese mit einem zweiten Projekt ergänzen. Setzen Sie zu diesem Zweck den Fokus auf den Knoten der *Solution* und wählen Sie den Menübefehl des Kontextmenüs *Add/New Project*.

Im nun angezeigten Dialogfeld (siehe **Abbildung 1.29**) zur Auswahl des Assistenten wählen wir eine *Class Library* (deutsch Klassenbibliothek oder auch Bibliotheksprojekt). Ein Bibliotheksprojekt erstellt eine Assembly mit Namenserweiterung *.dll*. Solche Assemblys können für die Begründung eines Prozessraums nicht verwendet werden, wohl aber kann ihre Funktionalität durch beliebige andere Assemblys verwendet werden.

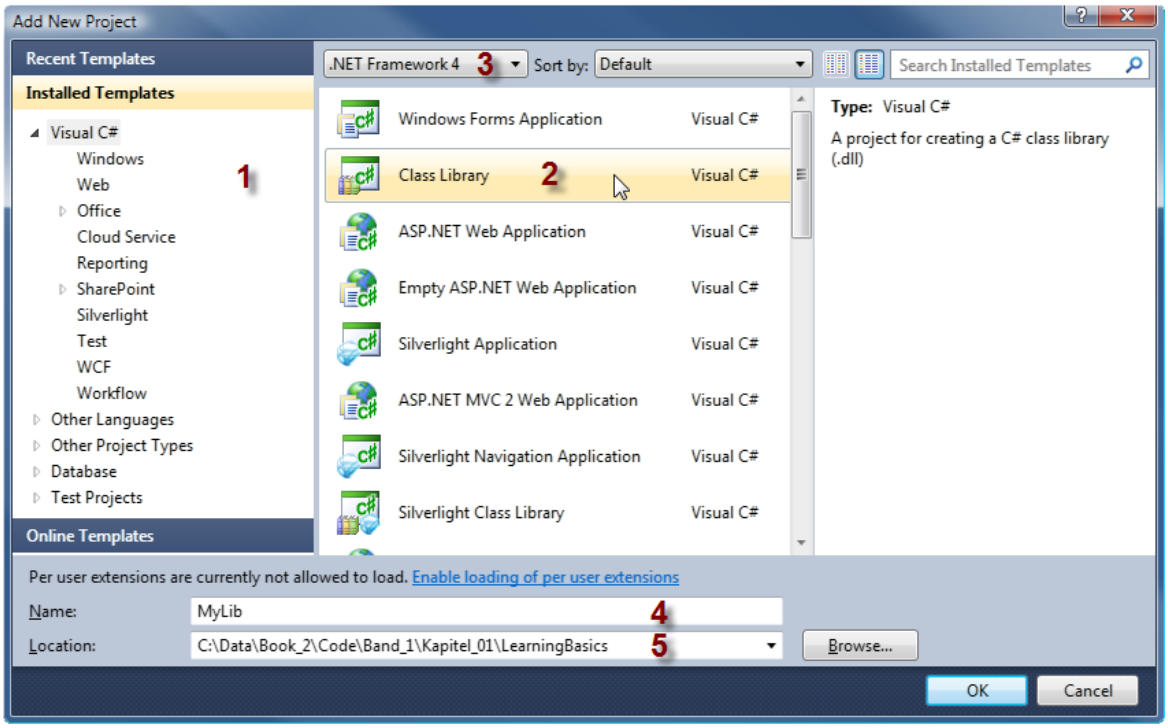

Abbildung 1.29: *Eine Solution mit einem Projekt ergänzen*

Der Assistent bedarf folgender Bedienung:

#	Benennung	Inhalt
1	Struktur der Vorlagen	Zeigt die Struktur, in die die verschiedenen verfügbaren Vorlagen für die Erstellung unterschiedlicher Projekte eingeteilt sind.
2	Liste der Vorlagen	Diese Liste zeigt die Vorlagen, die im selektieren Strukturelement 1 verfügbar sind.
3	Zielframework	Erlaubt die Auswahl des .NET Frameworks, auf dem das zu erstellende Projekt lauffähig sein soll. Achten Sie beim Ergänzen einer Solution mit weiteren Projekten darauf, dass alle Projekte gleich konfiguriert sind.
4	Projektname	Definieren Sie in diesem Textfeld den Namen des zu erzeugenden Projekts.
5	Speicherort	Definieren Sie in diesem Textfeld, wo das neue Projekt erzeugt werden soll. Standardmäßig schlägt Visual Studio hier den Namen der Solution als Verzeichnis vor. Da wir beim Erstellen des ersten Projekts bereits eine weitere Verzeichnisebene vorgesehen haben, kommt das neue Projekt neben dem bereits existierenden Projekt zu liegen.

Tabelle 1.15: *Erklärungen zu den Eingaben des Assistenten gemäß* ***Abbildung 1.29***

Hinweis Beginn

Obwohl nicht alle Projekttypen in ein- und derselben Solution sinnvoll kombiniert werden können, kann eine Solution rein technisch betrachtet durch Projekte beliebiger Typen ergänzt werden. Es sind auch mehrere Projekte möglich, die eine ausführbare Datei (*.exe*-Datei) erzeugen. Es entscheidet lediglich die Logik Ihres Projekts, welche Typen Sie verwenden.

Die typische Kombination von Projekttypen in einer Solution ist ein Projekt, das eine ausführbare Datei erzeugt, und ein oder mehrere Projekte, die Bibliotheken beinhalten. Achten Sie darauf, dass große

Projekte nicht nur aus einer Solution mit sehr vielen Projekten entstehen, sondern teilen Sie die Projekte in mehrere Solutions auf. Der daraus resultierende Vorteil ist die bessere Möglichkeit der Zuweisung von Verantwortlichkeiten und das einfachere Verwalten von Zugriffsrechten in einem System für Quellcodeverwaltung, wie es in großen Projekten verwendet wird. Der entstehende Nachteil ist im Wesentlichen die zu beherrschende Abhängigkeit zwischen den einzelnen Assemblys.

Hinweis Ende

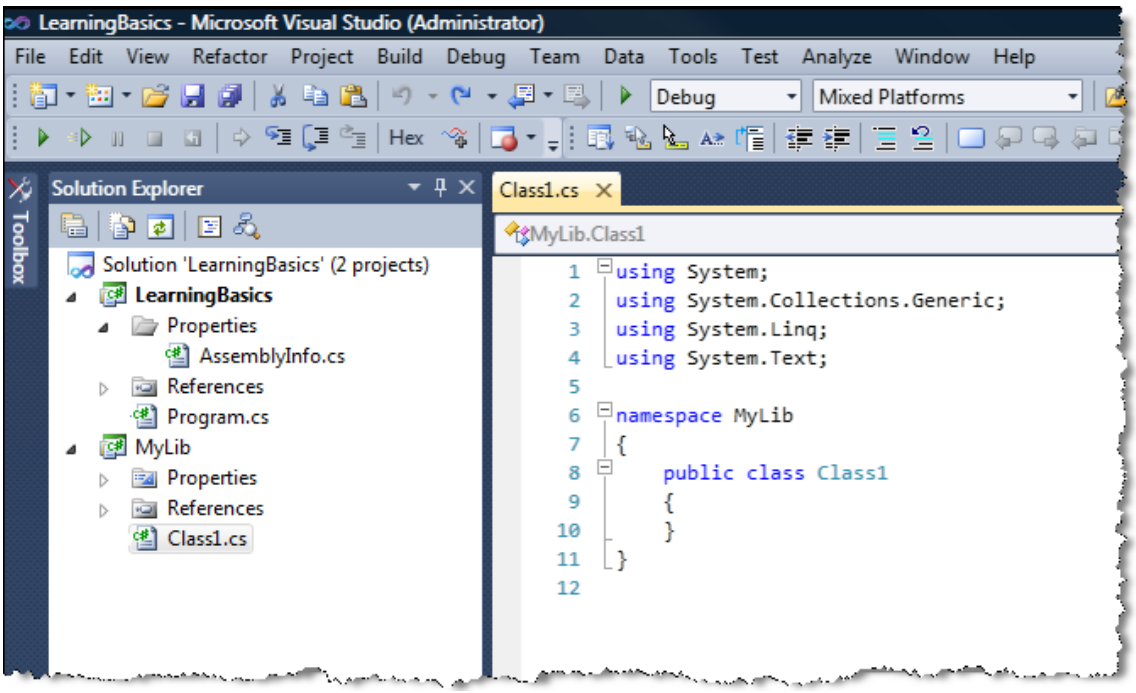

Abbildung 1.30: *Inhalt der Solution nach dem Ergänzen mit einem Bibliotheksprojekt*

Nach der Erstellung des Bibliotheksprojekts erinnern Sie sich daran, dass jedes Projekt seine eigenen Einstellungen hat. Sie können die Einstellungen für das soeben erstellte Projekt in der gleichen Art und Weise anpassen, wie Sie es für das erste Projekt im Abschnitt »Die Konfiguration kontrollieren und korrigieren« bereits gelernt haben.

Codieren der Klassenbibliothek

Bei der Erstellung eines Bibliotheksprojekts erstellt Visual Studio direkt das Skelett einer Klasse in der Datei *Class1.cs*. Diesen Typ können Sie selbstverständlich für Ihre eigenen Zwecke verwenden. Am besten passen Sie gleich den Dateinamen für Ihre eigenen Bedürfnisse an. Visual Studio bemerkt die Umbenennung und wird Sie fragen, ob der Name der Klasse sinngemäß auch angepasst werden soll. Die Gleichbenennung von Datei und Inhalt ist zwar technisch nicht Pflicht, aber trotzdem eine gute Sache, weil es so einfacher ist, sich zurecht zu finden.

Tipp Beginn

Selbstverständlich können Sie die generierte Datei *Class1.cs* durch das Betätigen der Taste (Entf) auch einfach löschen. Beachten Sie, dass dabei die Datei im Dateisystem auch tatsächlich gelöscht wird. Anschließend können Sie über den Kontextmenübefehl *Add/New Item* des Projekts den Assistenten für die Erstellung einzelner Projektelemente starten.

In diesem Dialogfeld wird ebenfalls eine Hierarchie zur Ordnung (Nummer 1 in **Abbildung 1.31**) und eine Liste der zur Verfügung stehenden Elementtypen (Nummer 2 in **Abbildung 1.31**) angeboten. Nach

der Auswahl des Typs ändern Sie den angebotenen Standardnamen des neuen Elements im Textfeld *Name* (Nummer 3 in **Abbildung 1.31**) Ihren Bedürfnissen entsprechend, und erstellen dann das Element durch Klicken auf die Schaltfläche *Add*.

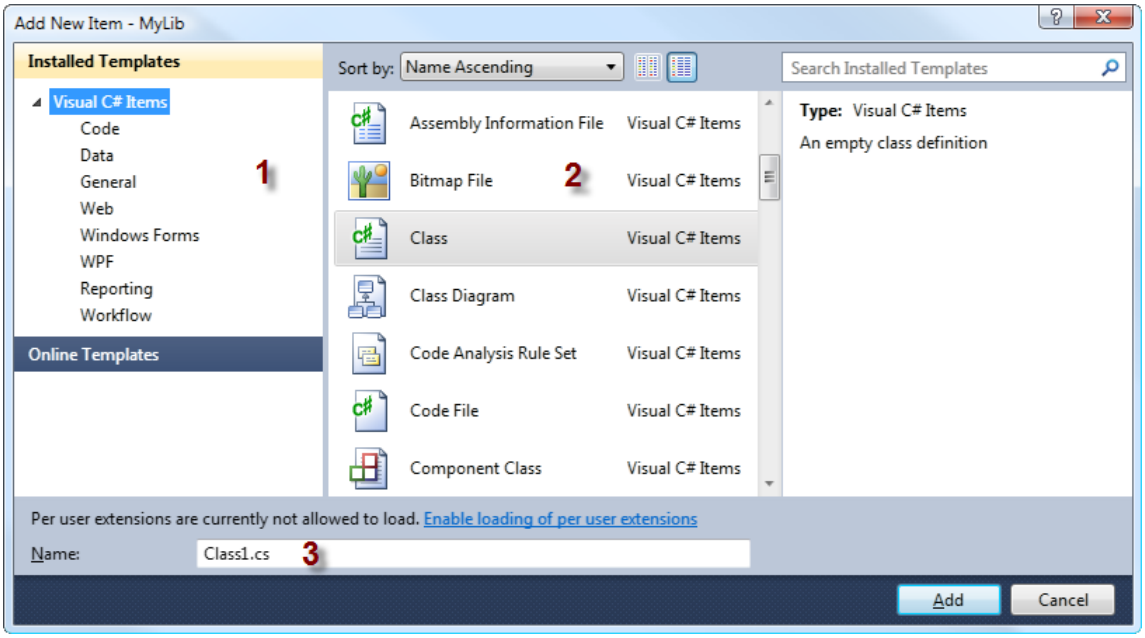

Abbildung 1.31: *Assistent für die Erstellung einzelner Elemente in einem Projekt*

Tipp Ende

Wenn die Äußerlichkeiten der Datei richtig gestellt sind, codieren Sie den Inhalt der Klassenbibliothek. Das Codieren selbst unterscheidet sich dabei nicht vom Codieren des ausführbaren Programms wie wir es in den vorangehenden Abschnitt »Code verändern« kennen gelernt haben.

Um aber wieder einen neuen Schritt vollziehen zu können, wollen wir nun einen Typ in unserer Klasse benutzen, der als solcher zwar existiert, der aber für IntelliSense noch nicht sichtbar ist. Dazu programmieren wir den Code gemäß **Abbildung 1.32**. Anhand der beiden Elemente »Wellenlinie« und »Smarttag« (Rechteck unterhalb des ersten Buchstaben) erkennen wir, dass die IntelliSense den verwendeten Typ `Assembly` nicht erkennt.

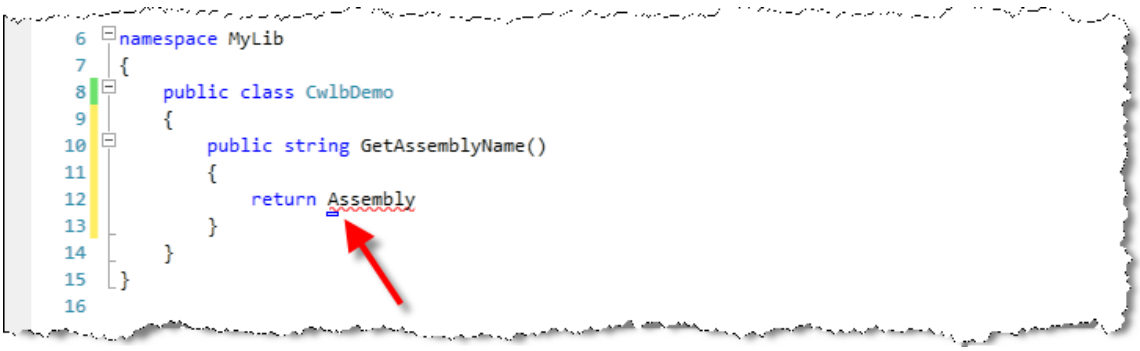

Abbildung 1.32: *Unfertiges Codefragment mit einem scheinbar unbekannten Typ*

Die Existenz des Smarttags verrät uns hier, dass IntelliSense Vorschläge für das weitere Vorgehen in der Hinterhand hält. Um diese Vorschläge einsehen zu können, aktivieren Sie das Smarttag wahlweise mit

der Maus oder mit der Tastenkombination (Umschalt)+(Alt)+(F10). Das Resultat (**Abbildung 1.33**) zeigt das Vorhandensein des Typs `Assembly` in einem noch nicht bekannt gemachten Namensraum. Die fehlende Definition `using System.Reflection` kann nun durch Auswahl der obersten Zeile einfach ergänzt werden, und anschließend die Zeile gemäß **Abbildung 1.34** fertiggestellt werden.

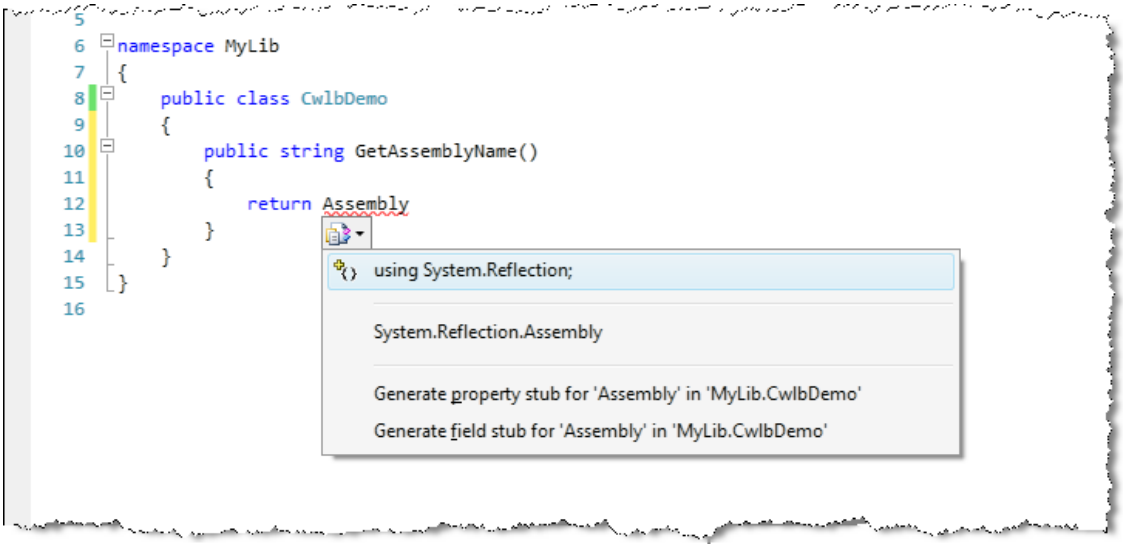

Abbildung 1.33: *Aktiviertes Smarttag für den Typ* `Assembly`

Abbildung 1.34: *Fertiger Code der Klassenbibliothek*

Tipp Beginn

Achten Sie darauf, dass der Code fehlerfrei kompiliert bevor Sie fortfahren.

Tipp Ende

Die Klassenbibliothek benutzen

Die zugegebenermaßen äußerst einfache Klassenbibliothek wollen wir nun in unserem ersten erstellten Projekt mitverwenden. Das erreichen wir, indem wir die beiden Projekte mit einer Referenz verbinden und anschließend den zu verwendenden Namensraum bekannt machen.

Um eine Referenz von einer Assembly auf eine andere festzulegen, verwenden Sie den Kontextmenübefehl *Add Reference…* auf dem Knoten *References* des Projekts, das das andere Projekt

mitverwendet. In unserem Fall definieren wir also eine Referenz vom ersten Projekt mit Namen *LearningBasics* (Nummer 1 in **Abbildung 1.35**) auf das Bibliotheksprojekt mit Namen *MyLib* (Nummer 2 in **Abbildung 1.35**).

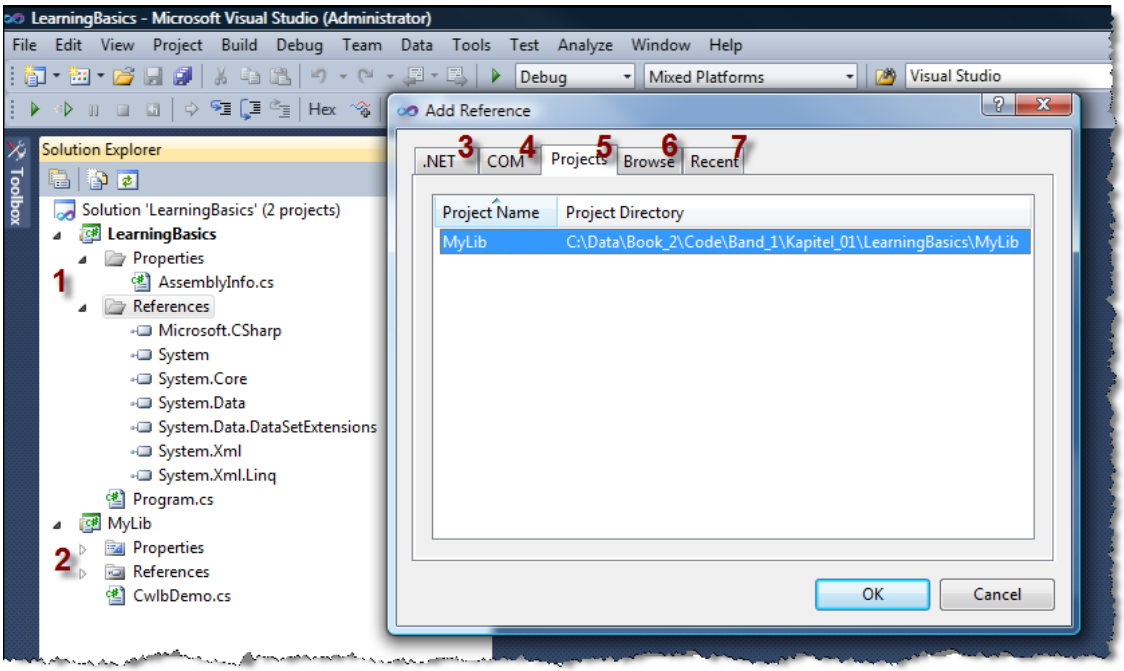

Abbildung 1.35: *Erstellen einer Referenz auf eine andere Assembly*

Als Antwort auf den Menübefehl *Add Reference...* liefert *Visual Studio* das gleichnamige Dialogfeld. Dieses Dialogfeld verfügt über mehrere Registerkarten, mit Hilfe derer verschiedene Quellen von Referenzen unterschieden werden:

#	Benennung	Inhalt
3	*.NET*	Zeigt Assemblys aus verschiedenen konfigurierten Ordnern des Systems. Achtung: Diese Registerkarte zeigt nicht den Inhalt des globalen Assemblycaches (siehe Abschnitt »Der globale Assemblycache«). Um eigene Assemblys in dieser Registerkarte anzeigen zu können, erstellen Sie einen eigenen Registrierungsschlüssel mit dem entsprechenden Pfad im Standardwert in: *HKLM\SOFTWARE\Microsoft\.NETFramework\<Version>\AssemblyFoldersEx*
4	*COM*	Zeigt die installierten COM-Komponenten auf dem System an.
5	*Projects*	Zeigt die jeweils anderen Assemblys der eigenen Solution an.
6	*Browse*	Erlaubt das freie Suchen einer Assembly im gesamten System.
7	*Recent*	Zeigt die zuletzt verwendeten Assemblys.

Tabelle 1.16: *Erklärungen zu den Eingaben des Assistenten gemäß* ***Abbildung 1.29***

Wie oben beschrieben und in der **Abbildung 1.35** visualisiert, verwenden wir die eigene Bibliotheksdatei *MyLib* um eine Referenz zu erstellen.

Wichtig Beginn

Beim Herstellen von Referenzen müssen Sie darauf achten, dass die Referenzen stets unär sind. Das heißt, dass eine Assembly A eine Assembly B referenzieren kann, aber nicht gleichzeitig die Assembly

B auch noch die Assembly A referenzieren darf.

Des Weiteren sind auch zirkuläre Referenzen verboten, die über mehr als zwei Assemblys gehen. Wenn Sie die Abhängigkeiten von Assemblys aufzeichnen, muss stets ein azyklischer, gerichteter Graph entstehen.

Wichtig Ende

Nach der Herstellung der Referenz kann in unserem Hauptprogramm der Typ *CwlbDemo* in der Klassenbibliothek *MyLib* genutzt werden. Ergänzen Sie in *Program.cs* in der Methode *Main()* die Zeile 15 gemäß **Abbildung 1.36**.

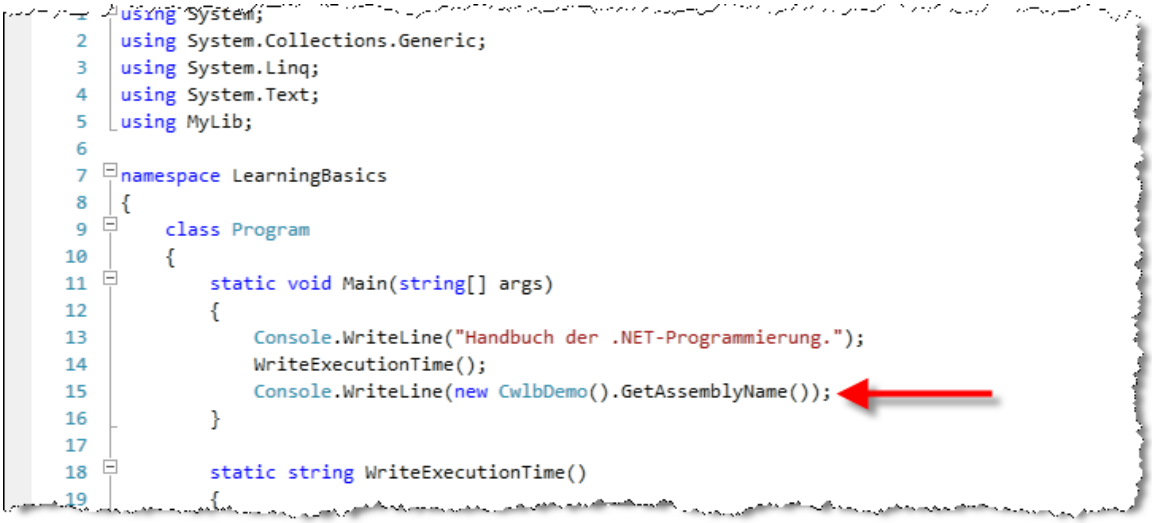

Abbildung 1.36: *Nutzung der Klassenbibliothek am Beispiel*

Die Ausführung des Codes zeigt in etwa den Inhalt gemäß **Listing 1.3**:

```
Handbuch der .NET-Programmierung.
Das Programm wurde am 16.03.2010 um 20:41:51 ausgeführt
MyLib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null
```

Listing 1.3: *Ausgabe des Demoprogramms* LearningBasics *nach Verbindung*

Von Assemblys und Prozessen

Im vorangehenden Abschnitt über den grundlegenden Umgang mit Visual Studio haben wir in zwei Projekten zwei verschiedene Assemblys hergestellt. Diese Assemblys können ohne weiteres zusammen auf einer Maschine mit .NET Framework installiert und gestartet werden. Nun wollen wir uns mit der Frage auseinandersetzen, wie die Anwendung beim Starten geladen wird, welche ungefähren Schritte dabei ablaufen, und wie wir den Startvorgang respektive unsere Anwendung mittels Konfiguration beeinflussen können.

Einen Prozessraum aufbauen

Eine .NET-Anwendung starten wir, indem wir die ausführbare Datei (*.exe*-Datei) starten. Das Betriebssystem kann dabei nicht unterscheiden, ob es sich um eine .NET- oder eine native Anwendung handelt. Somit müssen aus Sicht des Betriebssystems auch keine speziellen Konfigurationen vorgenommen werden. Die notwendigen Angaben für den Start der Anwendung können kompatibel zu

allen nicht .NET-Anwendungen mit den vom Betriebssystem vorgesehenen Möglichkeiten hinterlegt werden. Dazu gehören:

- Batchdateien
- Verknüpfungen im Dateisystem
- Verknüpfungen im Menüsystem

Die .NET-Anwendung verfügt über einen PE-Header (Portable Executable), an dem das Betriebssystem zunächst erkennt, ob die Datei für das Betriebssystem ausführbar ist oder nicht. Als zweite wichtige Information kann daraus der Typ der zu verwendenden Benutzeroberfläche gelesen werden. Windows unterscheidet zwischen grafischer oder zeichenorientierter Benutzeroberfläche.

Nach dem Start und der Konfiguration des Prozessraums führt das Betriebssystem den Code aus, der im PE-Header der Datei als Einstiegspunkt verzeichnet ist. Das ist zunächst nativer Code, denn das Betriebssystem selbst kennt zu diesem Zeitpunkt nichts anderes. Dieser Code führt zur Datei *mscoree.dll* mit der Methode `_CorExeMain()`. Diese Methode ist verantwortlich für die Etablierung der CLR im Prozessraum, zeichnet sich verantwortlich für das Laden des effektiven IL-Codes der ausführbaren Datei und startet die eigentliche Ausführung des Programms.

Im Prozessraum sind nach dem Start sowohl das native System als auch das .NET-System etabliert. Das native System sieht im Prozessraum das .NET-System nur als Daten. Das .NET-System hingegen sieht sich als Daten und Programm und kann dazu auch noch auf die Daten und Programme des nativen Systems zugreifen.

Hinweis Beginn

Der PE-Header einer Datei definiert systematisch den Inhalt der Datei. Er bildet die Schnittstellenbeschreibung zwischen dem Betriebssystem und der Programmdatei. Das Betriebssystem erkennt, ob die Datei für das System kompatibel ist oder nicht. Interessierte können die Spezifikation des Formats unter *http://www.microsoft.com/whdc/system/platform/firmware/PECOFFdwn.mspx* kostenlos beziehen.

Weitere Erklärungen zum Aufbau des Prozessraums entnehmen Sie Kapitel 4.

Hinweis Ende

Konfiguration von .NET-Anwendungen

Zusätzlich zu der Bereitstellung der Anwendung ist es dem Entwickler oder dem Systemtechniker möglich, die CLR mittels einer separaten Konfigurationsdatei auch nach der Herstellung, also sozusagen kundenseitig, zu beeinflussen. Die Konfigurationsdatei ist eine XML-basierte Datei, die sich in das Gefüge der .NET-Konfigurationsdateien einfügt.

Die Struktur und in weiten Teilen auch die effektiven Inhalte sind von der CLR und der .NET-Bibliothek vorgeschrieben. Gleichzeitig hat Microsoft auch Möglichkeiten vorgesehen, dass wir eigene anwendungsspezifische Werte in die Konfigurationsdatei einbringen können (siehe Kapitel 4).

Die Anwendung einer Konfigurationsdatei ist nicht zwingend vorgeschrieben. Ihre selbst hergestellten Anwendungen sind somit in der Regel lauffähig, sobald Sie eine startbare Datei erzeugt haben. Bei der Verwendung bestimmten Techniken wird allerdings Visual Studio automatisch eine Konfigurationsdatei erzeugen und gewisse Werte darin einlagern.

Hinweis Beginn

Grundsätzlich können alle Techniken von .NET mittels imperativer Programmierung[2] auch ohne

[2] Die Programmierung mittels Codeanweisungen in einer Programmiersprache wie C# wird auch als imperative Programmierung bezeichnet

Konfigurationsdateien verwendet werden. In gewissen Situationen macht es aber durchaus Sinn, den von Visual Studio vorgeschlagenen Weg nicht nur zu benutzen, sondern eventuell sogar zu erweitern. Dies trifft dann zu, wenn in den Programmen Werte benutzt werden, die installationsspezifisch oder gar benutzerspezifisch sind. Solche Werte sollten nie in Form von konstanten Werten im Code eingebracht werden, weil deren Anpassung nicht flexibel genug geschehen kann.

Wird ein änderbarer Wert außerhalb des Programms gehalten, von diesem beim Starten eingelesen und das Programm entsprechend dynamisch konfiguriert, erhalten Sie als Gewinn die entsprechende Flexibilität.

In der Zeit vor .NET wurden entsprechende Werte in der Windows-Registrierung abgelegt. Sie sollten mit .NET für die Anwendungskonfiguration Abstand von der Registrierung nehmen und diese Werte in den in diesem Kapitel vorgestellten Konfigurationsdateien einbringen.

Hinweis Ende

Beim Laden der Anwendung wird von der CLR die Konfigurationsdatei automatisch berücksichtigt, sofern diese vorhanden ist. Die spezifischen CLR-Werte und die Werte, die zu einer Konfiguration der .NET-Klassen führen werden von den betroffenen Teilen automatisch adaptiert. Die eigenen Teile müssen programmatisch eingelesen und so zur Wirkung gebracht werden (siehe Kapitel 4 und dort den Abschnitt »Eigene Konfigurationswerte erstellen«).

Damit der Mechanismus von der CLR korrekt initialisiert wird, müssen folgende Voraussetzungen erfüllt werden:

- Die Konfigurationsdatei muss der Technik entsprechend benannt und am richtigen Ort im Dateisystem eingebracht worden sein.
- Der Standardname der Datei ist gleich dem Namen der zu startenden Datei der Anwendung mit dem zusätzlichen Suffix *.config* (Beispiel: *MyApplication.exe.config*).
- Der Speicherort entspricht dem der ausführbaren Datei.
- Bei Webanwendungen ist der Name immer *web.config*.
- Der Speicherort bei Webanwendungen kann in jedem Verzeichnis der Webanwendung sein. Die Konfigurationsdateien von Webanwendungen wirken hierarchisch additiv.
- Der Inhalt der Datei ist eine nach .NET-Vorschriften gültige XML-Datei.

Hinweis Beginn

Die detaillierte Konfiguration von Webanwendungen wird im zweiten Band dieses Buches behandelt.

Hinweis Ende

Für das Erstellen einer Anwendungskonfigurationsdatei verwenden wir einmal mehr den Assistenten von Visual Studio. Darin wählen wir die Vorlage *Application Configuration File* und belassen den Namen der Datei bei *App.config*. Visual Studio erzeugt die Datei im Stammverzeichnis der Anwendung. Auch diese Einstellung sollten Sie nicht ändern, denn Visual Studio erzeugt jedes Mal aus dieser Datei eine Kopie, die in das jeweilige Ausgabeverzeichnis der Anwendung gelegt wird (*bin/Debug* oder *bin/Release*). Der Zielort des Kopiervorgangs ist von den aktuellen Einstellungen für das Kompilieren der Anwendung abhängig. Somit können Sie die Änderungen jederzeit in der Datei *App.config* mit dem XML-Editor vornehmen und haben die Gewähr, dass beim Testen der Anwendung die richtige Konfiguration verwendet wird.

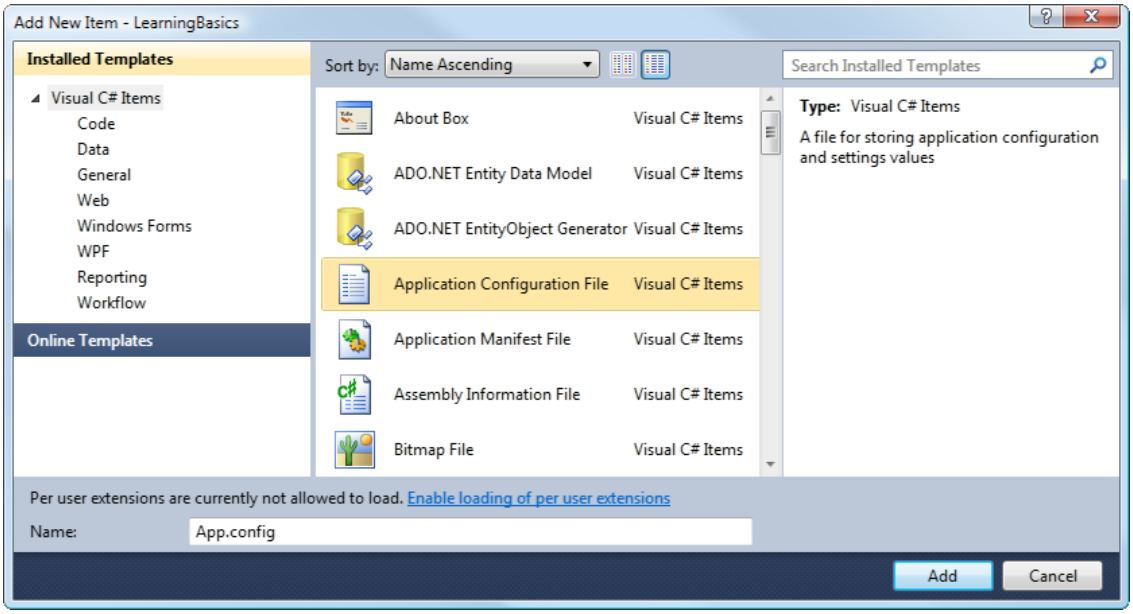

Abbildung 1.37: *Assistent in Visual Studio für das Erstellen der Anwendungskonfigurationsdatei*

Abbildung 1.38 visualisiert den Vorgang der Nutzung der Konfigurationsdatei Ihrer Anwendung durch Sie und Visual Studio.

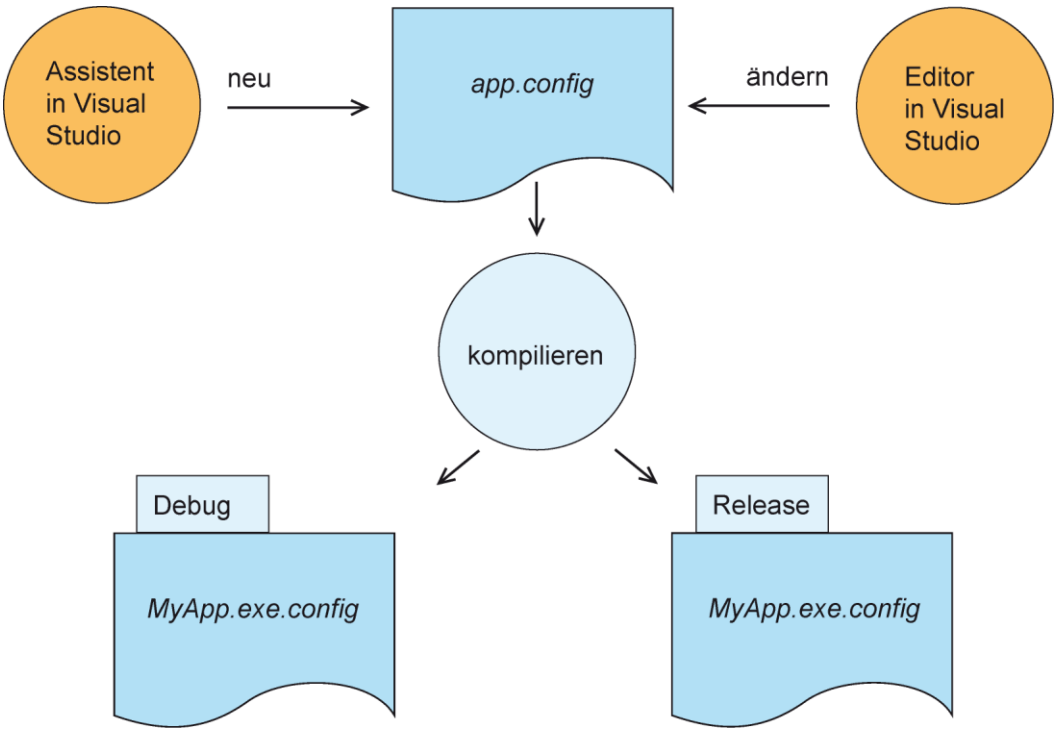

Abbildung 1.38: *Schematische Darstellung der Vorbereitung der Anwendungskonfigurationsdatei*

Wichtig Beginn

.NET liest automatisch nur die Anwendungskonfiguration, deren Dateiname aus dem Namen der ausführbaren Datei abgeleitet wird (Beispiel: *MyApplication.exe.config*). Anders herum definiert heißt das, dass selbst dann, wenn eine Konfigurationsdatei einer DLL mit einem korrekt lautenden Namen vorhanden ist (*MyLibrary.dll.config*), diese Datei nicht eingelesen wird, da eben nur diejenige Datei der eigentlichen startbaren Datei geladen wird.

Einträge von andern Konfigurationsdateien müssen mit Hilfe eines selbst programmierten Codes gelesen werden. Alternativ können Sie die Inhalte aus den andern Konfigurationsdateien manuell in die Hauptdatei umkopieren.

Die Ausnahme der obigen Regel bilden die Systemkonfigurationen von .NET selbst. Diese werden hierarchisch vor der eigentlichen Anwendungskonfiguration mitberücksichtigt und für ihre Anwendung jeweils interpretiert. Dazu gehören auch Standardwerte für die verschiedenen Teilgebiete.

Beachten Sie ferner, dass eine alleinige Änderung des Inhalts der Datei *app.config* ohne Kompilation keine Änderung der effektiv verwendeten Datei *<Projekt>.exe.config* in den Verzeichnissen *bin\Debug* respektive *bin\Release* zur Folge hat. Das ist somit eine Situation, in der Sie bewusst einen Buildvorgang auslösen müssen.

Wichtig Ende

Die .NET-Bibliothek und die CLR weisen viele vordefinierte Konfigurationseinträge für Ihre Anwendungen auf. Das Gute an diesen Definitionen ist, dass sie standardmäßig nicht nötig sind. Die eher schwierige Seite daran ist, dass sie sehr vielfältig sind und dadurch beim Gebrauch manchmal zu Komplikationen führen.

Da die Standardeinträge der CLR und der .NET-Bibliothek zweckgebunden sind, macht es aus meiner Sicht keinen Sinn, hier eine detaillierte Übersicht aller Einträge wiederzugeben. Vielmehr zeigt dieses Kapitel das Prinzip und die Gliederung der Einträge einer Konfigurationsdatei. Ferner gebe ich eine Übersicht der Gebiete, mit denen Sie Einfluss auf die CLR und die .NET-Bibliothek nehmen können.

Die generierte Konfigurationsdatei sieht sehr bescheiden aus (**Listing 1.4**). Diese Datei können Sie jederzeit mit dem XML-Editor erweitern. Das Grundprinzip dabei ist, dass die nächste Stufe im Stammeintrag `<configuration>` das Teilgebiet der Konfiguration widerspiegelt. Die darauf folgenden Stufen sind spezialisiert nach Teilgebiet. Diese Teilgebiete werden in **Tabelle 1.17** zusammengefasst. Die spezifischen Erklärungen dazu werden in den Kapiteln der verschiedenen Bände zu den Teilgebieten zweckgebunden erklärt.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
</configuration>
```

Listing 1.4: *Eine generierte, leere Anwendungskonfigurationsdatei*

Eintrag im Stammverzeichnis	Verwendung
`<assemblyBinding/>`	Einstellungen für die Konfiguration von zu verwendenden Assemblys
`<startup/>`	Einstellungen für die Konfiguration der Anwendung in Bezug auf Unterstützung der CLR
`<runtime/>`	Einstellungen für die Konfiguration der Runtimeumgebung selbst

`<mscorlib>`
`  <cryptographySettings/>`
`</mscorlib>`
	Einstellungen für die Definition von Klassen und Namen der Kryptographie

`<configSection/>`	Definiert die Einstiegspunkte der benutzerdefinierten Settings, die in der Sektion `<applicationSettings/>` untergebracht sind

<applicationSettings/>	Benutzerspezifische Einstellungen
<connectionStrings/>	Verbindungszeichenfolgen für die Konfiguration von Datenzugriffen wie Datenbanken
<system.data/>	Konfiguration für die Konfiguration von verwendbaren Techniken für Datenbankzugriffe
<system.diagnostics/>	Definiert das Verhalten von .NET in Bezug auf Diagnose und damit verbunden das Verhalten von Trace
<system.codedom/>	Einstellungen für die Konfiguration der Compilerumgebung
<system.net/>	Einstellungen für die Konfiguration von Netzwerkfunktionen
<system.runtime.remoting/>	Einstellungen für die Konfiguration von Remoting
<system.web/>	Einstellungen für die Konfiguration von Webanwendungen
<system.serviceModel/>	Einstellungen für die Konfiguration der Windows Communication Foundation (WCF)

Tabelle 1.17: *Tabelle der wichtigsten Konfigurationseinträge im Stammverzeichnis*

Listing 1.5 zeigt eine Konfigurationsdatei mit ihren Inhalten. Achtung: Diese Datei zeigt lediglich ein paar ausgewählte Beispiele von Einträgen in einer Anwendungskonfigurationsdatei. Sie dient lediglich dazu, Ihnen an dieser Stelle eine konkrete Vorstellung einer Anwendungskonfigurationsdatei zu geben. Die Datei ist in dieser Form nicht nutzbar.

```
<?xml version="1.0"?>
<configuration>
  <appSettings/>
  <connectionStrings/>
  <system.web>
    <httpRuntime maxRequestLength="10000"/>
    <siteMap defaultProvider="SiteMapProvider">
      <providers>
        <clear/>
        <add
          name="SiteMapProvider" type="System.Web.XmlSiteMapProvider" siteMapFile = "~/App_Data/UI.sitemap"/>
      </providers>
    </siteMap>
    <pages theme="Demo"/>
    <compilation debug="true"/>
    <authentication mode="Forms">
      <forms loginUrl="LogonPage.aspx">
        <credentials passwordFormat="MD5">
          <user name="WebUser" password="3E45AF4CA27EA2B03FC6183AF40EA112"/>
        </credentials>
      </forms>
    </authentication>
    <customErrors mode="RemoteOnly" defaultRedirect="GenericErrorPage.htm">
        <error statusCode="403" redirect="NoAccess.htm" />
        <error statusCode="404" redirect="FileNotFound.htm" />
    </customErrors>
    <httpHandlers>
      <add path="*.myExt" verb="*" type="System.Web.HttpForbiddenHandler" validate="true" />
    </httpHandlers>
  </system.web>
</configuration>
```

Listing 1.5: *Beispiel einer Anwendungskonfigurationsdatei*

Suchen und Finden von Programmdateien und Assemblys

Wir sind von den allermeisten Anwendungen gewohnt, dass diese mindestens fehlerfrei starten. Doch was tun wir, wenn wir beim Starten einer Anwendung vom System ein Feedback bekommen, das etwa dem von **Listing 1.6** gleicht?

```
Unhandled Exception: System.IO.FileNotFoundException:
   Could not load file or assembly 'MyLib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null'
   or one of its dependencies. Das System kann die angegebene Datei nicht finden.
```

Listing 1.6: *Fehlermeldung beim Laden eines .NET-Programms*

Die Fehlermeldung in **Listing 1.6** besagt, dass die Assembly *MyLib.dll* oder eine Assembly, die von *MyLib.dll* mitverwendet wird, nicht gefunden wird, und aufgrund dessen die Assembly *MyLib.dll* nicht geladen wurde. Um diese relativ einfachen Fehlersituation zu provozieren habe ich im Verzeichnis einfach die Datei *MyLib.dll* gelöscht und anschließend das Programm *LearningBasics.exe* gestartet.

Eine solche Situation kann Ausgangspunkt für eine langwierige Fehlersuche sein. Um sich dabei nicht zu verlieren, sondern möglichst gezielt vorgehen zu können, müssen folgende zwei Dinge bekannt sein:

- Welche Assemblys benutzt meine Anwendung tatsächlich?
- Findet meine Anwendung alle benötigten Assemblys in der richtigen Version auf dem System?

In den nun folgenden Abschnitten wollen wir diese beiden Fragen beleuchten und die nicht immer kurzen Antworten dazu studieren.

In eine Assembly hineinschauen

Als Erstes wollen wir der Frage nachgehen, welche Assemblys meine Anwendung benutzt, und zusätzlich wollen wir wissen, wie denn .NET erkennt, dass diese Assemblys benutzt werden. Die Antwort dazu ist zunächst einmal einfach: Bei der Erstellung der Datei mit dem Compile/Link-Vorgang werden die dazu notwendigen Daten erhoben und in der erstellten Assembly gespeichert.

Doch beginnen wir ganz vorn: Eine Assembly wird durch den Compiler *csc.exe* erstellt. Visual Studio bedient sich im Hintergrund des Compilers und versteckt dabei die gesamte Parametrisierung, so dass wir Entwickler uns nicht um diese Dinge zu kümmern brauchen. Die Assembly, die als Resultat dieses Vorgangs geboren wird, entspricht in ihrem Inhalt zum einen den Normen des Betriebssystems Windows und zum andern den Normen von .NET. Beide Systeme müssen die Inhalte richtig interpretieren können, um die Arbeit wie erwartet zu erledigen. Die dazu notwendige Struktur der Assembly umfasst:

- Einen PE-Header nach Windows-Norm für den direkten Start aus dem Betriebssystem
- Einen .NET-Header für den Start des Programms durch die CLR
- Ein Manifest als Information über das Umfeld und die Versionen
- Den IL-Code als eigentlicher .NET-Programmcode
- Weitere Daten, die das Programm nutzen kann

Die für uns wichtigen Dinge der oben aufgezählten Liste sind die .NET-spezifischen Daten des Manifests der Datei und allenfalls der IL-Code. Für die Visualisierung dieser Daten finden wir in den Zusatzwerkzeugen von Visual Studio das Programm *ildasm.exe* (IL-Code Disassembler). Starten Sie das Programm *ildasm.exe* unter Angabe unseres Programms *LearningBasics.exe*, so präsentiert sich ein Fenster gemäß **Abbildung 1.39**.

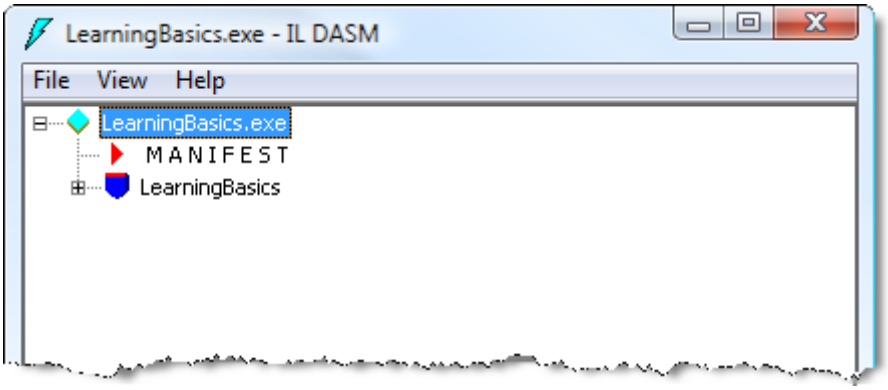

Abbildung 1.39: *Das Werkzeug* ildasm.exe *für die Datei* LearningBasics.exe *nach dem Start*

Der IL-Code Disassembler zeigt uns zwei Einträge innerhalb des Knotens *LearningBasics.exe*. Der erste Knoten mit der Bezeichnung *MANIFEST* enthält für uns wichtige Metainformationen über die Assembly selbst. Mit einem Doppelklick auf das Manifest können Sie diese Information in einem eigenen Dialogfeld betrachten (**Abbildung 1.40**).

Das Manifest zeigt uns folgende Inhalte:

#	Benennung	Inhalt
1	Externe Assembly	Dieses Programm verwendet die externe Assembly *mscorlib.dll* in der Version 4.0.0.0
2	Externe Assembly	Dieses Programm verwendet die externe Assembly *MyLib.dll* in der Version 1.0.0.0
3	Assembly	Die Assembly dieser Datei heißt *LearningBasics*
4	Zusatzinformationen	Die Assembly besitzt diverse Zusatzinformationen (gekürzt wiedergegeben)
5	Weitere CLR Infos	Dieser Bereich zeigt weitergehende Informationen für die CLR

Tabelle 1.18: *Erklärungen zu bezeichneten Elementen in der* ***Abbildung 1.40***

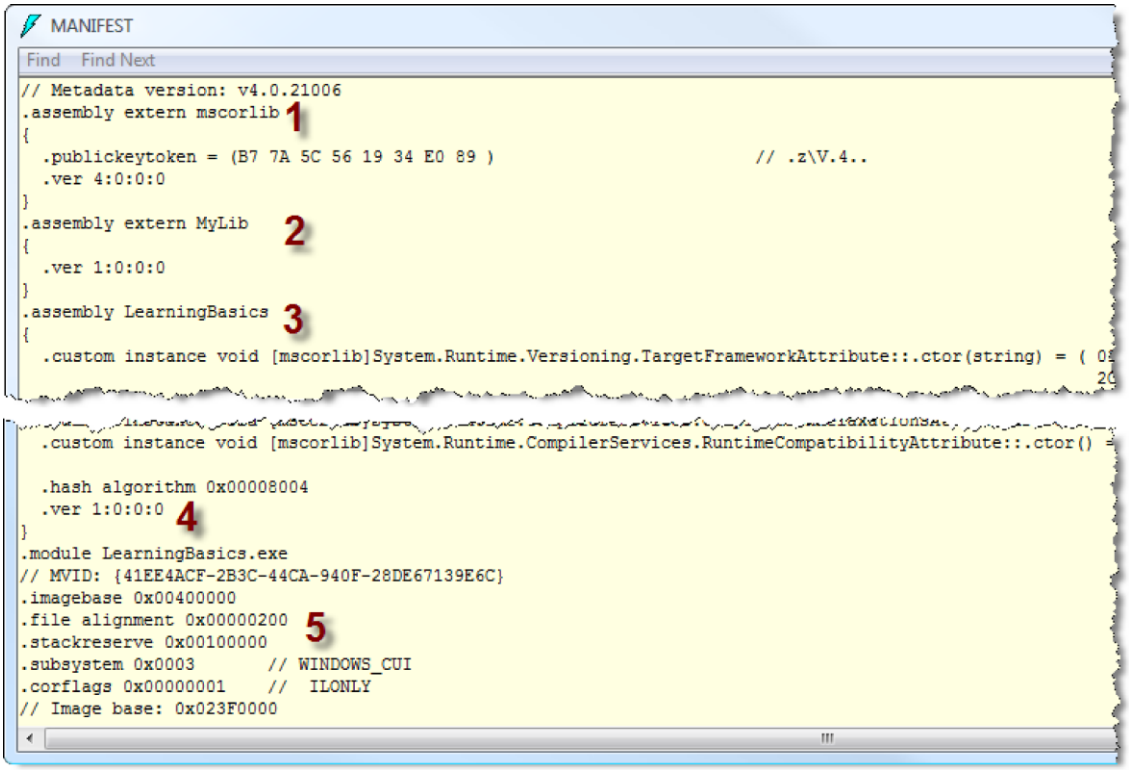

Abbildung 1.40: *Das Manifest unseres ersten .NET-Programms*

Somit haben wir die vollständige Lösung des ersten Rätsels gefunden: Im Manifest speichert der C#-Compiler die Namen und Versionsinformationen der mitverwendeten Assemblys (Nummern 1 und 2 in der **Abbildung 1.40**) und die Versionsinformation der eigenen Assembly (Nummer 4 in der **Abbildung 1.40**). Wenn Sie sich in Ihrem System das Manifest noch genauer anschauen, werden Sie erkennen, dass die Daten von *AssemblyInfo.cs* (Nummer 3 in der **Abbildung 1.40**, siehe auch Abschnitt »Assembly-Informationen«) ebenfalls im Manifest abgelegt sind. Die Form der gespeicherten Informationen sind Attributeobjekte. Mehr dazu lesen Sie in Kapitel 4 »Der Umgang mit Attributen«.

Profitipp Beginn

Der C#-Compiler wird mit dem .NET Framework installiert und ist somit auch ohne Visual Studio verfügbar. Neben dem C#-Compiler können Assemblys auch mit dem Assembly-Linker *al.exe* erzeugt werden. Diese Variante wird vor allem beim Herstellen von Satelliten-Assemblys für Sprachübersetzungen oder andern speziellen Assemblys benutzt.

Eine tiefergehende Erklärung für Satelliten-Assemblys entnehmen Sie Kapitel 9.

Profitipp Ende

Eine Assembly mit einem starken Namen versehen

In der einfachen Assembly, die wir eben erstellt haben, ist keine Herstellersignierung vorhanden. Das heißt, dass die Assembly in dieser Art auf einfachste Weise nachgebaut werden kann. Dieser Umstand könnte von böswilligen Individuen ausgenutzt werden. Jemand könnte die Methoden mit anderem Inhalt wieder zur Verfügung stellen. Ein einfacher Austausch der Assembly würde genügen und die Täuschung wäre perfekt.

Um dem entgegen zu wirken, kann eine Assembly mit einer nicht austauschbaren Signatur versehen

werden. Diese Signatur nennen wir in der .NET-Fachsprache einen »starken Namen« (engl.: strong name). Die Erzeugung von starken Namen basiert auf einem System, das mit öffentlichen und privaten Schlüsseln arbeitet. Der Hersteller einer Assembly erzeugt mittels eines Werkzeugs einen einmaligen privaten Schlüssel, mit dem er sein Werk signiert. Ist eine Assembly erst einmal signiert, wird bei jeder Nutzung der Assembly durch eine andere Assembly diese Signatur vermerkt, so dass ein nachträglicher Austausch der benutzten Assembly bemerkt werden kann. Dazu lagert die benutzende Assembly den öffentlichen Schlüssel der eingebundenen Assembly in Form eines Hashcodes bei sich ein. Der Hashcode wird mit dem Namen *.publickeytoken* im Manifest gekennzeichnet. Die Assembly *mscorlib.dll* wird in unserem Beispiel entsprechend referenziert (siehe **Abbildung 1.40**).

Sobald die signierte Assembly geladen wird, prüft die CLR die Signatur der geladenen Assembly. Passen beide zusammen, wird die Assembly in den Speicher geladen, andernfalls verweigert die CLR das Laden und löst eine Ausnahme aus.

Für die Erstellung von signierten Programmen benutzen Sie einen selbst erstellten Schlüssel. Diesen Schlüssel können Sie mit dem Werkzeug *sn.exe* generieren. Aber Vorsicht! Die so erstellte Datei ist wie ein Schlüssel zu Ihrem Safe. Sie sollten dafür sorgen, dass nur Sie selbst und Ihre Vertrauenspersonen an diesen Schlüssel gelangen. Fällt der Schlüssel in falsche Hände, könnte Software von jemand anderem unter Ihrem Namen erstellt und veröffentlicht werden.

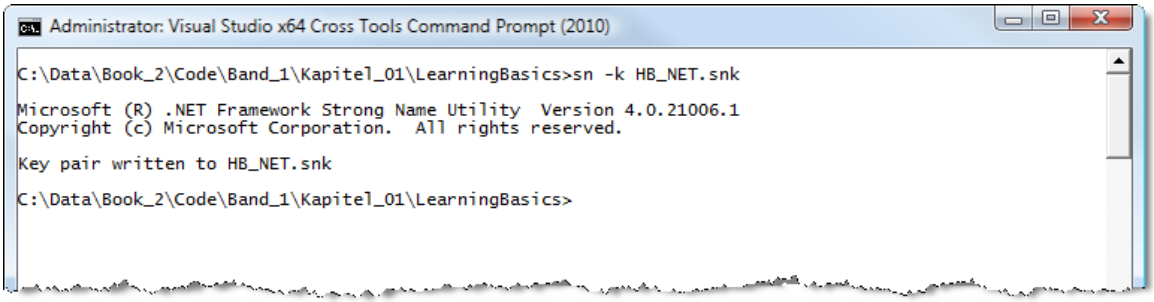

Abbildung 1.41: *Manuelle Erstellung einer Schlüsseldatei für die Signierung von Assemblys*

In der Regel werden Sie für alle Programme ein und dieselbe Schlüsseldatei verwenden. In größeren Firmen kann es allerdings durchaus sein, dass Entwicklungsteams oder größere Organisationseinheiten je einen eigenen Schlüssel verwenden. Der generierte Schlüssel kann in einem Projekt über die Projekteinstellungen mitbenutzt werden, indem Sie auf der entsprechenden Seite das Kontrollkästchen *Sign the assembly* einschalten und anschließend die generierte Datei auswählen.

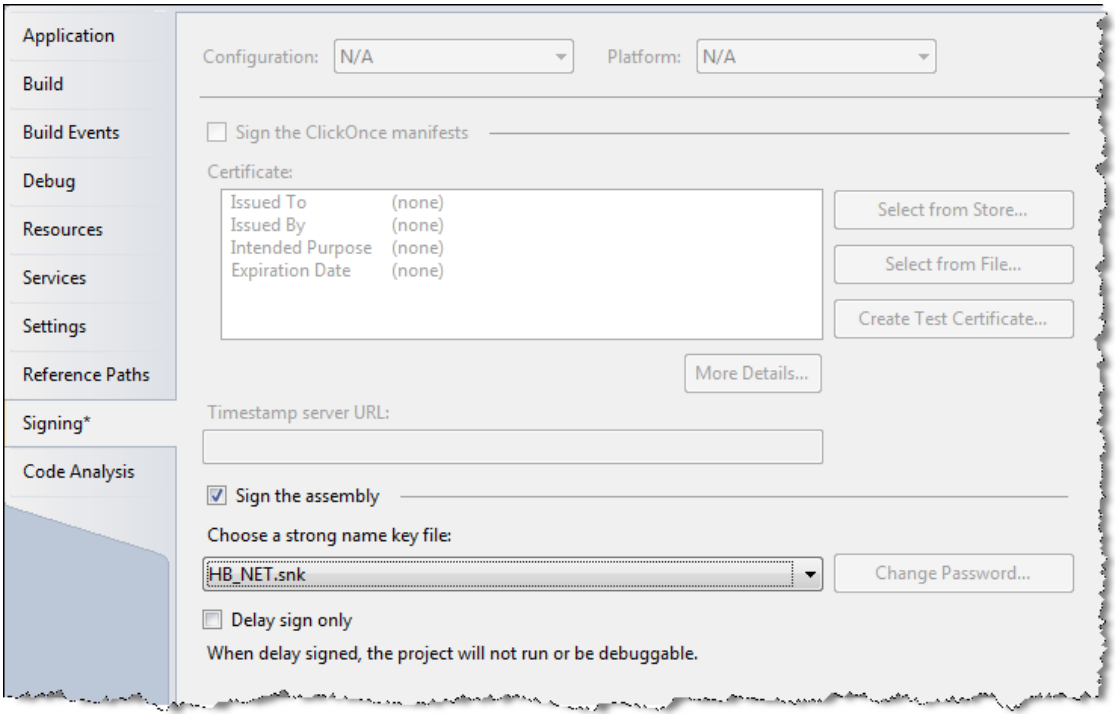

Abbildung 1.42: *Nutzung des generierten Schlüssels im Projekt*

Das Resultat der Bezeichnung der Assembly mit einem starken Namen können Sie wiederum dem Manifest entnehmen. Dort ist nun der Eintrag der eigenen Assembly mit einem öffentlichen Schlüssel ergänzt worden.

Wie bereits erwähnt, stellt die firmenweite Verwendung des privaten Schlüssels ein gewisses Risiko dar. Besonders in großen Firmen mit vielen Entwicklern sollte dem Umstand Rechnung getragen werden, dass die private Schlüsseldatei nur allzu leicht entwendet werden kann. Der Hersteller von .NET hat auch hier eine Lösung parat: verzögertes Signieren (engl.: delayed signing). Bei diesem Verfahren wird zunächst aus dem vollwertigen Schlüssel der öffentliche Teil in eine zweite Schlüsseldatei ausgelagert (**Abbildung 1.43**).

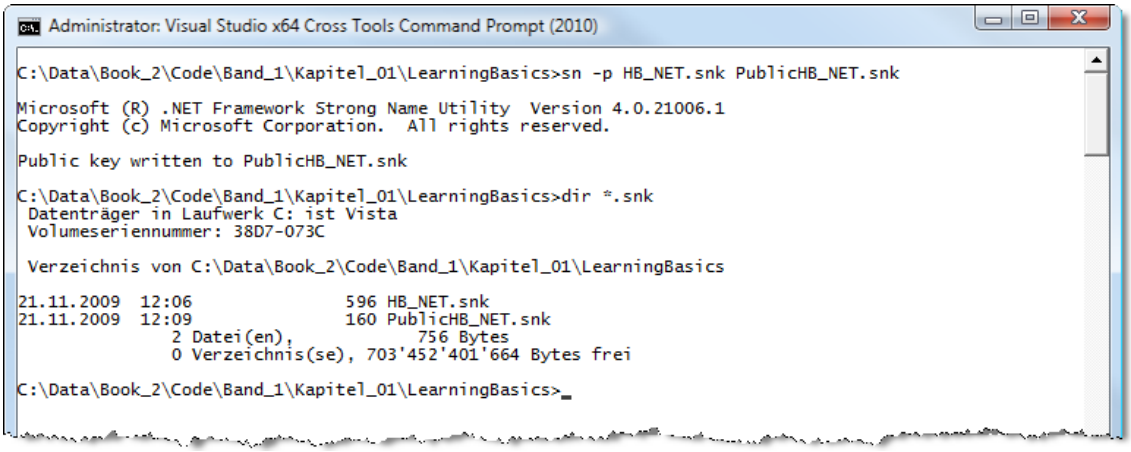

Abbildung 1.43: *Aufteilen der Schlüsseldatei in öffentlichen Teil und vollwertigen Teil*

Anschließend wird die Projektkonfiguration dahingehend verändert, dass sie nur noch den öffentlichen Teil verwendet. Wir nehmen die dazu notwendige Einstellung wiederum im Konfigurationsdialogfeld des Projekts und dort auf der Registerkarte für die Signierung vor.

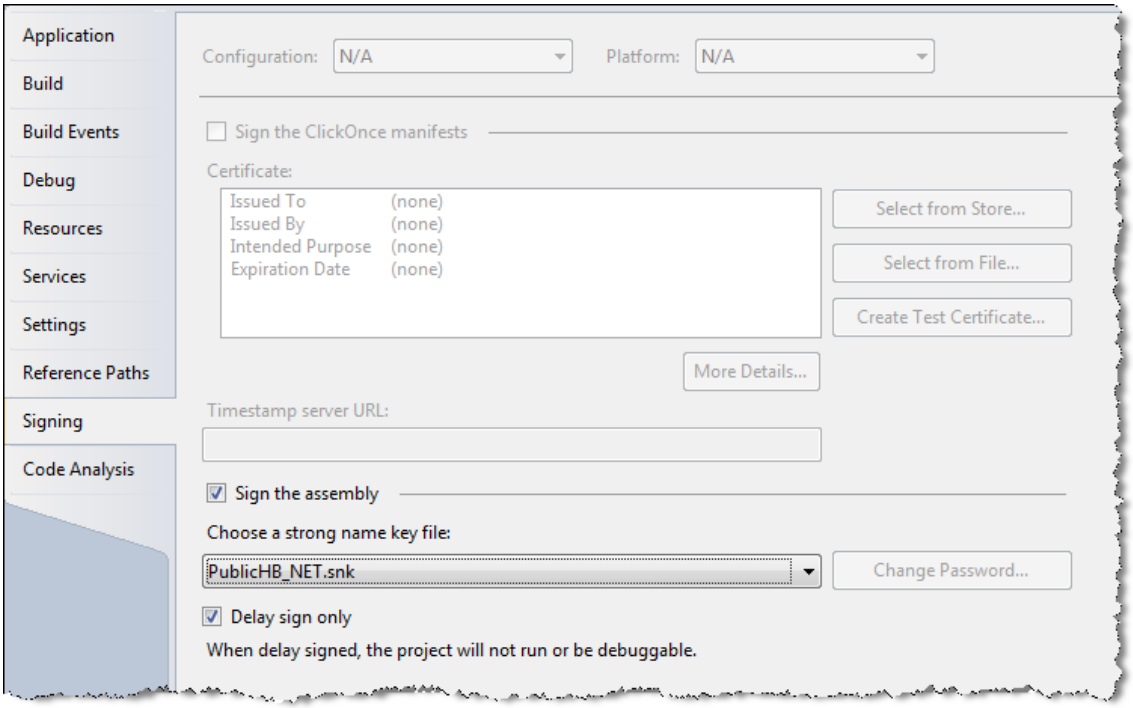

Abbildung 1.44: *Veränderung der Einstellung für die Verwendung der verzögerten Signatur*

Durch die Einstellung für die verzögerte Signatur wird in der zu erstellenden Datei zwar der Platz für die Signatur ausgespart, diese allerdings nicht übertragen. Das Resultat davon ist, dass die Verbindungen zwischen den Assemblys korrekt erstellt werden, aber die Assembly, die verzögert signiert wurde von der CLR nicht akzeptiert wird. Der Grund dafür liegt in der fehlenden Signatur, die verhindert, dass die CLR die Verifikation der Assembly nicht vornehmen kann. Es tritt eine Ausnahme des Typs `FileLoadException` auf.

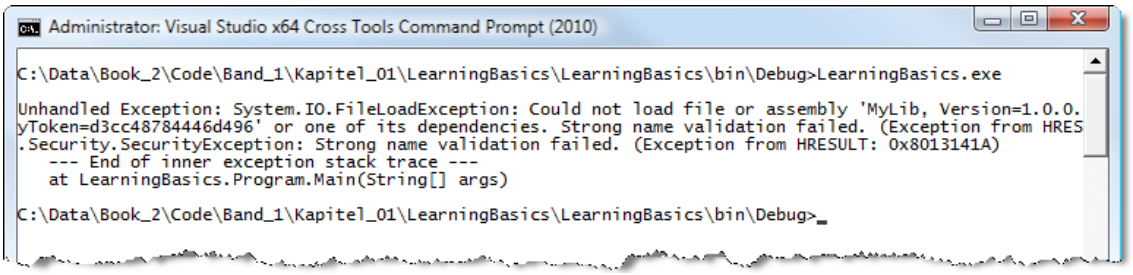

Abbildung 1.45: *Beim Starten einer Anwendung mit verzögerter Signatur tritt eine Ausnahme auf*

Damit die Entwickler ihre selbst erstellte Software testen können, muss auf dem Entwicklungsrechner das Überprüfen der Herstellersignatur für diese Assembly ausgeschaltet werden. Dies erreichen wir erneut mit dem Werkzeug *sn.exe* und der Option *–Vr*.

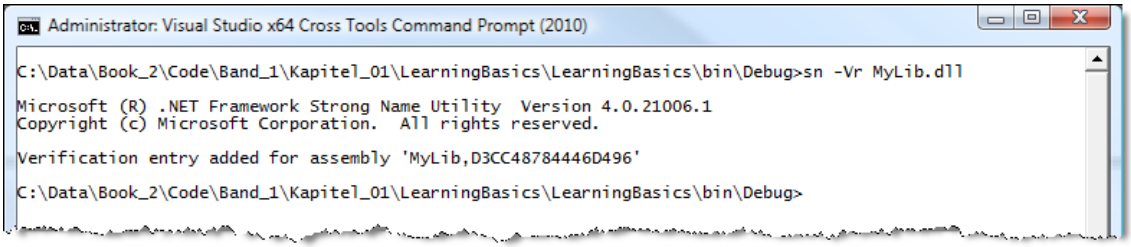

Abbildung 1.46: *Ausschalten der Überprüfung des starken Namens durch die CLR*

Nach Fertigstellung des Projekts oder am Ende eines automatischem Buildvorganges müssen die erzeugten Assemblys noch mit der effektiven Signatur versehen werden. Da der entsprechende Speicherplatz in der Assembly bereits vorhanden ist, muss dazu der Code nicht noch einmal kompiliert, sondern nur noch signiert werden. Dazu kann erneut das Werkzeug *sn.exe* benutzt werden.

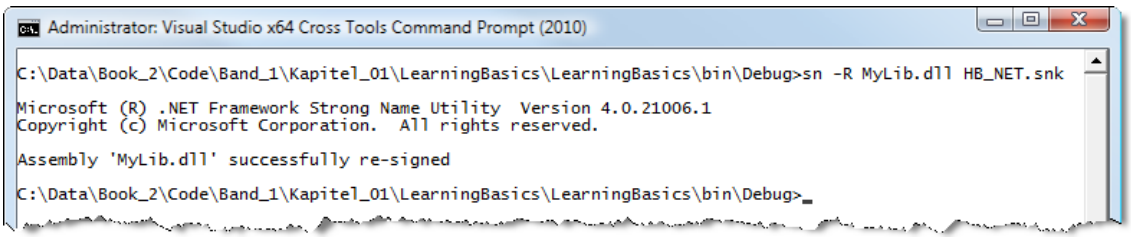

Abbildung 1.47: *Abschließende Signierung der fertigen Assembly*

Wichtig Beginn

Das Ausschalten der Überprüfung von Signaturen durch die CLR sollte nur auf Entwicklermaschinen vorgenommen werden, und nie auf einer produktiven Maschine, egal ob es sich um Server oder Arbeitsplatz handelt. Die Konsequenzen der Ausschaltung können eine gefährliche Sicherheitslücke öffnen.

Da die notwendigen Definitionen für die CLR in der Registrierung innerhalb des Schlüssels `HKEY_LOCAL_MACHINE` vorgenommen werden, hat ein normaler Benutzer keine Möglichkeit, solche Einträge vorzunehmen. Das Werkzeug *sn.exe* muss also für die Erstellung entsprechender Einträge mit Administratorrechten ausgeführt werden.

Die Installation von signierten Assemblys zieht nicht zwingend eine andere Behandlung gegenüber nicht signierten Assemblys nach sich. Trotzdem eröffnen sich mit der Signatur einer Assembly neue Nutzungsmöglichkeiten:

- Die CLR benutzt bei Referenzierung signierter Assemblys eine strenge Versionskontrolle beim Laden der betreffenden Assembly
- Signierte Assemblys können im Global Assembly Cache (GAC) installiert werden
- Die Signatur erfüllt den Zweck der Fälschungssicherheit
- Die Signatur erlaubt eine zuverlässige Erkennung des Herstellers

Wichtig Ende

Der globale Assemblycache

Die Bibliotheken von .NET und Ihre eigenen, für .NET entwickelten Bibliotheken, werden in Form von normalen Assemblys erstellt. Um diese Bibliotheken zu benutzen, stellt die CLR zwei Lösungsansätze bereit:

- Alle Bibliotheksassemblys, die zu einer Anwendung gehören, werden direkt im entsprechenden

Anwendungsverzeichnis gespeichert

- Die Bibliotheksassemblys werden zentral im globalen Assemblycache gespeichert

Ablage der Bibliothek bei der Anwendung

Durch die Ablage der Assemblys pro Anwendung können wir eine äußerst einfache Installation der Anwendung erreichen. Das reine Kopieren der Anwendung reicht aus, um eine lauffähige Version des Programms auf einer Maschine bereitzustellen. Zudem wird die Installation einer Anwendung den Betrieb einer bereits installierten Anwendung kaum stören. Allerdings kann das bei vielen Anwendungen irgendwann in einer Invasion von gleichen Bibliotheken enden, und dies wiederum hat den Nachteil, dass wir kaum mehr wissen, welche DLL in welcher Version auf der Maschine wo installiert ist. Eine generelle Änderung der Version einer bestimmten Assembly wird kaum mehr möglich sein.

Der Vorteil dieser Methode liegt darin, dass jede Anwendung genau die richtige Version der notwendigen Assemblys bei sich hat. Dies erlaubt wiederum ohne Probleme mehrere Versionen von ein und derselben Assembly auf einem System zu benutzen.

Gemeinsame Nutzung von Assemblys im globalen Assemblycache

Da vor allem die System-DLLs von praktisch jeder .NET-Assembly genutzt werden, ist es sinnvoll, diese Systembibliotheken zentral zu speichern. Dazu stellt .NET den GAC (Globaler Assemblycache) zur Verfügung. Der globale Assemblycache ist in der Lage, von einer bestimmten Assembly mehrere Versionen zu unterscheiden. Die Merkmale für die Unterscheidung liegen in folgenden vier Elementen:

- Name
- Version
- Sprache
- Öffentliches Schlüsseltoken

Die Abweichung nur eines dieser vier Merkmale reicht aus, um eine Assembly als different zu einer anderen zu erkennen und entsprechend ist es erlaubt, diese im globalen Assemblycache zu installieren. Da das öffentliche Schlüsseltoken in der Liste aufgeführt ist, und wir das Token nur bei der Signierung der Assembly mit einem starken Namen erhalten, können wir daraus schließen, dass nur Assemblys mit einem starken Namen in den globalen Assemblycache installiert werden können.

Der Vorteil der Installation einer Assembly im globalen Assemblycache liegt darin, dass die Assembly zentral gehalten werden und so auch vermeintlich besser gesteuert werden kann, welche Version benutzt wird. Hat Sie das Wort vermeintlich im vorangehenden Satz gestört? Wenn ja, dann studieren Sie unbedingt den Abschnitt »Laden von Assemblys«, gleich im Anschluss an diesen Abschnitt!

Organisation des globalen Assemblycaches

Der Assemblycache ist ein Bestandteil der .NET-Installation. Die Installation ist auch zuständig für die Übertragung der Assemblys von .NET in den globalen Assemblycache. Genau genommen existieren seit der Einführung von .NET 4.0 zwei globale Assemblycaches:

- Ein Cache für die CLR 2.0 und den darauf basierenden .NET-Versionen 2.0, 3.0 und 3.5 sowie sämtlichen Assemblys von Drittherstellern unabhängig davon, ob diese für die CLR 2.0 oder 4.0 vorgesehen sind.
- Ein Cache für die CLR 4.0 und die Systembibliotheken von .NET 4.0.

Um den Inhalt des globalen Assemblycaches zu prüfen, respektive eigene Assemblys in den Assemblycache zu installieren oder von diesem zu entfernen, stellt .NET zwei Möglichkeiten zur Verfügung:

Interaktiv kann integriert im Windows-Explorer das Verzeichnis *%windir%\Assembly* betrachtet werden. Dieser Blick zeigt den Inhalt des globalen Assemblycaches in der Version 2.0. Sie sehen also darin die Dateien von .NET 2.0 bis 3.5 und Ihre eigenen Assemblys, respektive diejenigen anderer auf dem System installierter Produkte. Diese Ansicht zeigt nicht den Assemblycache 4.0.

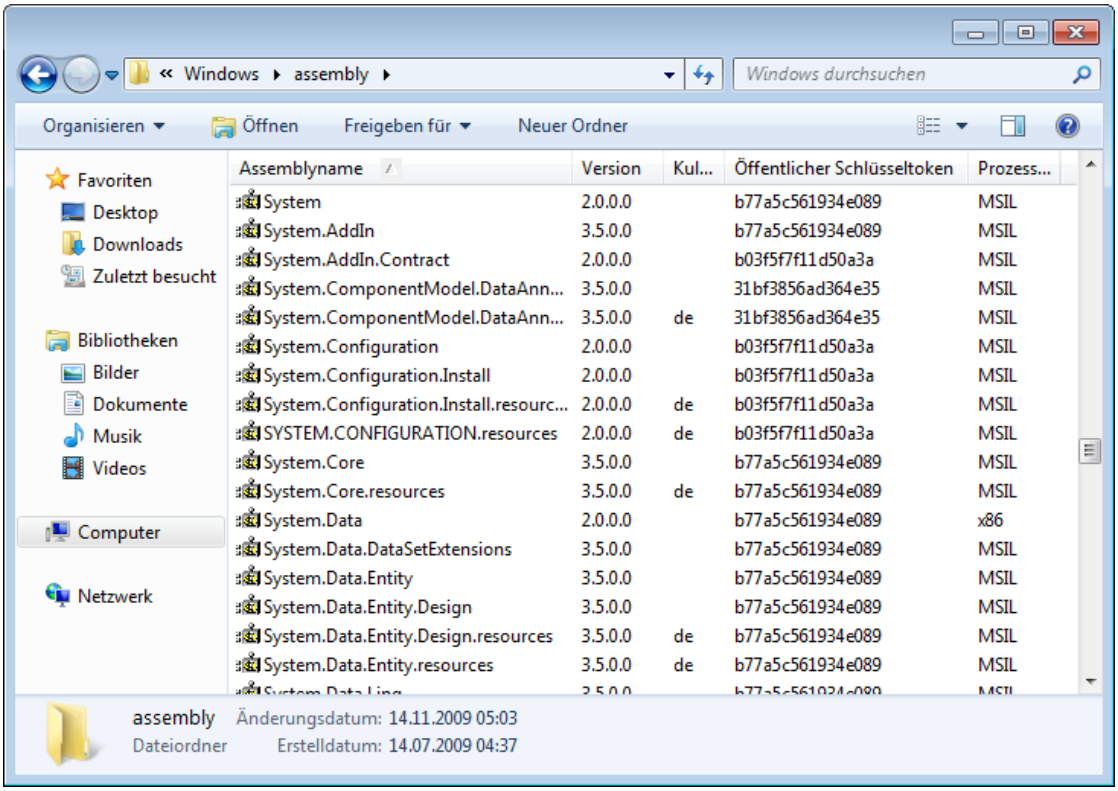

Abbildung 1.48: *Ansicht des globalen Assemblycaches im Windows-Explorer*

Die zweite Variante besteht aus dem Gebrauch des Kommandozeilenwerkzeugs *gacutil.exe*. Dieses Werkzeug ist Bestandteil des Lieferumfanges von Visual Studio. Mit *gacutil.exe* können Sie den Inhalt beider Versionen des Assemblycaches auslisten, aber auch eigene Assemblys installieren oder deinstallieren (in den GAC 2.0).

Abbildung 1.49 zeigt die Verwendung von *gacutil.exe* für das gezielte Auflisten der enthaltenen .NET-Bibliothek *system.dll*. Als Resultat werden hier beide Versionen der Assembly aufgeführt. Leider ist in der Auflistung kein Hinweis vorhanden, aus welchem Cache diese Information stammt.

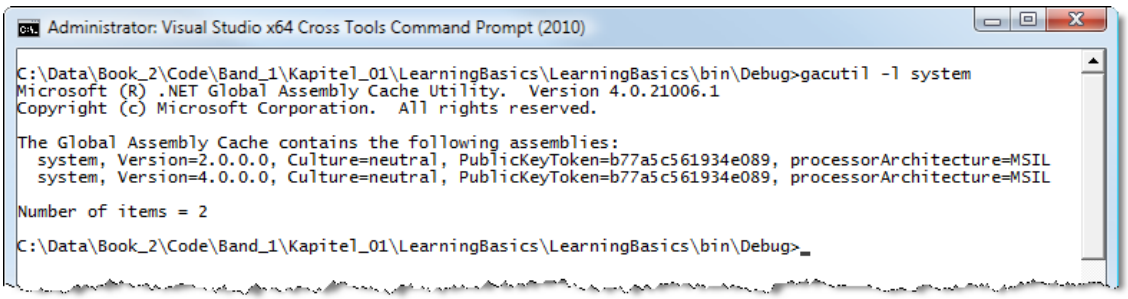

Abbildung 1.49: *Eine Verwendung von* gacutil.exe

Profitipp Beginn

Studieren Sie die möglichen Parameter von *gacutil.exe*. Da es sich um ein Werkzeug für die Nutzung in einer Kommandozeile handelt, ist das Werkzeug auch geeignet für die automatisierte Installation von eigenen Assemblys im globalen Assemblycache.

Profitipp Ende

Laden von Assemblys

Nachdem wir wissen, dass die CLR die Namen der zu ladenden Assemblys im Manifest der aufrufenden Datei findet, wollen wir der Frage nachgehen, wie und wo die CLR die mitverwendeten Assemblys findet. Auch hier ist die Antwort eigentlich einfach, aber nicht kurz.

Für das Laden von Assemblys benutzt die CLR einen Mechanismus, der in einer fixen Reihenfolge bestimmte Regeln anwendet und definierte Stellen des Systems nach der gefragten Assembly absucht. Wird während der Abarbeitung des Vorgangs die gesuchte Assembly gefunden, bricht die CLR den Suchvorgang ab und verwendet die gefundene Assembly.

Die Reihenfolge der Tätigkeiten definiert sich wie folgt:

1. Versionsrichtlinien prüfen und anwenden
2. Prüfen, ob die gesuchte Assembly bereits geladen ist und wenn ja, diese verwenden
3. Prüfen, ob die gesuchte Assembly einen starken Namen hat, und wenn ja, im globalen Assemblycache prüfen, ob die Assembly von dort geladen werden kann
4. Prüfen, ob ein Hinweis für die Codebasis definiert ist und wenn ja, prüfen, ob die Assembly von dort geladen werden kann
5. Prüfen, ob die gesuchte Assembly im oder unterhalb des Verzeichnisses liegt, von dem aus die Assembly gestartet wurde, die den Prozessraum begründete und wenn ja, von dort laden
6. Auslösen eines Ereignisses in die Anwendung und Verwenden der allfällig zurückgelieferten, manuell geladenen Assembly
7. Auslösen der Ausnahme: `FileNotFoundException`

In den folgenden Abschnitten sind einige der obigen Schritte detailliert erklärt, oder Sie finden dort Verweise auf die Stellen im Handbuch, die auf das jeweilige Know-how eingehen.

Wichtig Beginn

Eine Assembly hat immer eine Versionsinformation. Wenn Ihr Projekt keine Versionsinformation bereitstellt, erstellt der C#-Compiler die Version 0.0.0.0. Die Versionsinformation wird beim Laden einer Assembly nicht immer berücksichtigt. Wir unterscheiden zwischen Assemblys ohne und Assemblys mit starkem Namen.

Assemblys ohne starken Namen entstehen in den Schritten, wie wir sie auf den vorangehenden Seiten aufgebaut haben. Bei solchen Assemblys berücksichtigt die CLR bei der Suche nur den Dateinamen. Die Version wird nicht berücksichtigt. Daraus folgt, dass Assemblys ohne starken Namen beliebig versioniert und ausgetauscht werden können. Aber Achtung: Das hat auch zur Folge, dass Sie nicht mit Bestimmtheit wissen, welche Assembly gerade benutzt wird. Alleine die Tatsache, dass Ihr Installationsprogramm eine bestimmte Version einer Assembly installiert hat, gibt noch keine Garantie, dass nach ein paar Wochen des Betriebs der Anwendung immer noch die gleiche Version auf dem Rechner ist.

Sobald eine Assembly mit einem starken Namen versehen ist, wird beim Ladevorgang die Version geprüft. Stimmt dabei die Version nicht exakt mit der gesuchten Version überein, wird gemäß obiger Vorgangsliste solange gesucht, bis die richtige Assembly gefunden wurde, oder kein Resultat zu einer Ausnahme führt.

Angaben für die Herstellung einer Assembly mit einem starken Namen entnehmen Sie dem Abschnitt »Eine Assembly mit einem starken Namen versehen«.

Wichtig Ende

Versionsrichtlinien anwenden

Wir haben soeben gesehen, dass der Name einer mitverwendeten Assembly mitsamt ihrer Versionsinformation im Manifest derjenigen Datei gespeichert ist, die eine Assembly mitverwenden will. Als grundlegende Information ist das gut so. In der dynamischen Welt, in der wir leben, kommt es immer wieder vor, dass sich einige Programmteile schneller entwickeln als andere. Konkret muss die CLR auch damit umgehen können, dass von einer bestimmten Assembly eine neuere Version verwendet wird, auch wenn im Manifest etwas anderes steht.

Genau zu diesem Zweck steht an erster Stelle des Vorgehens beim Laden von Assemblys die Anwendung der Versionsrichtlinien. Als Ausgangslage ist die Information aus dem Manifest vorhanden. Aufgrund dieser Information wird in der Konfiguration der Anwendung geprüft, ob ein Upgrade vorhanden ist und wenn ja, welche Version anstelle der gesuchten geladen werden soll.

Die Verwendung von anderen Versionen einer Assembly als derjenigen, die im Manifest einer aufrufenden Assembly definiert ist, kann mittels zweier verschiedener Vorgehen beeinflusst werden:

- Versionsangabe in der Konfigurationsdatei der Anwendung
- Herstellen einer speziellen Assembly für den Fall der Verwendung des globalen Assemblycache

Die Herstellung einer Versionsangabe in der Anwendungskonfigurationsdatei geschieht nach dem Muster wie es in **Listing 1.7** gezeigt ist. Die Konfiguration definiert mit der Zeile `<assemblyIdentity .../>` den Dateinamen und das öffentliche Schlüsseltoken der Assembly, für die die Versionsumsetzung gilt. Mit der Zeile `<bindingRedirect .../>` wird angegeben welche Version durch welche neuere Version ersetzt werden soll.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
        <assemblyIdentity name="MyLib" publicKeyToken="485b04124812f10b" />
        <bindingRedirect oldVersion="1.0.0.0-1.9.9.9" newVersion="2.0.0.0" />
      </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Listing 1.7: *Beispiel einer Versionsangabe in einer Anwendungskonfigurationsdatei*

Wichtig Beginn

Beachten Sie, dass es im gezeigten Fall keine Rolle spielt, wo die gesuchte Assembly installiert ist. Sie kann sich sowohl lokal neben der ausführbaren Datei, als auch im globalen Assemblycache befinden. Ist die Bibliothek im selben Verzeichnis wie die ausführbare Datei gespeichert, bedeutet dies lediglich, dass nur die neuere Version vorhanden sein kann, weil das Dateisystem im Gegensatz zum globalen Assemblycache nur eine Datei mit einem bestimmten Namen in einem Verzeichnis halten kann.

Wichtig Ende

Die Variante der Definition einer sogenannten Versionsrichtlinie in der Anwendungskonfigurationsdatei bedingt, dass Sie bei der Installation einer neueren Version einer Bibliothek alle Anwendungen kennen müssen, die diese Bibliothek verwendet, damit Sie wissen, welche Anwendungskonfigurationsdatei(en) Sie anpassen müssen. Bei der Herstellung allgemeiner Bibliotheksdateien im Stil von Dateien in der .NET-Bibliothek, ist dieser Weg ziemlich unbefriedigend. Vermutlich weiß niemand genau, welche

Anwendung welche Bibliothek verwendet.

Für den Fall, dass die gesuchte Bibliothek im globalen Assemblycache installiert ist, gibt es die Variante der Anwendung einer sogenannten Herausgeberrichtlinie (engl. Publisher-Policy). Eine Herausgeberrichtlinie ist eine Assembly, die ausschließlich eine Versionsrichtlinie enthält und einem bestimmten Namensmuster folgt.

Die Herstellung und Installation einer Herausgeberrichtlinie erfolgt in drei Schritten:

1. Definieren der Versionsrichtline im gleichen Stil wie dies für eine Anwendungskonfiguration geschieht
2. Erstellen der Assembly mit dem Assemblylinker *al.exe* unter Angabe der entsprechenden Parameter (siehe **Listing 1.8**)
3. Installieren der erzeugten Assembly im globalen Assemblycache

```
al                                  // Assembly-Linker
/link:PublisherPolicy.config        // Input: Datei, die in Schritt 1 erstellt wurde
/out:policy.1.0.MyLib.dll           // Output: Name der Herausgeberrichtlinie
/keyfile:HB_NET.snk                 // Name der Datei mit der Signatur
/version:1.0.0.0                    // Version des neuen Assembly (Wenn nicht angegeben: 0.0.0.0)
```

Listing 1.8: *Kommandozeile für die Erstellung einer Herausgeberrichtlinie*

Wichtig Beginn

Beachten Sie, dass der Dateiname der Herausgeberrichtlinie zwingend folgender Definition folgen muss:

policy.<majorNumber>.<minorNumber>.AssemblyName**.dll**

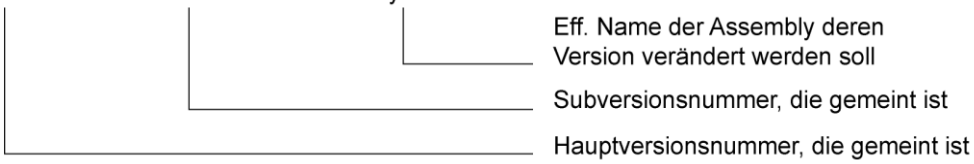

Abbildung 1.50: *Definition des Dateinamens einer Herausgeberrichtlinie*

Wichtig Ende

Im globalen Assemblycache suchen

Die Suche der Assembly im globalen Assemblycache erfolgt als erste Suche im System. Die vorangehenden Schritte dienen ausschließlich der Sicherstellung der Suche nach der richtigen Version und der Prüfung, ob die Assembly bereits geladen ist.

Hinweis auf die Codebasis

Kann die gesuchte Assembly nicht aus dem globalen Assemblycache geladen werden, prüft die CLR als Nächstes, ob in der Konfiguration der Anwendung allenfalls ein Hinweis auf die Codebasis vorhanden ist. Ein Hinweis auf die Codebasis besteht aus der Angabe des Speicherorts für die gesuchte Assembly in Form eines URLs.

Auch diese Angabe kann in Form eines Eintrages in die Anwendungskonfigurationsdatei eingebracht werden. **Listing 1.9** zeigt die Variante der Verwendung eines Webverweises und das **Listing 1.10** zeigt die Verwendung eines Verweises auf die lokale Maschine.

```
<?xml version="1.0"?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <dependentAssembly>
```

```
            <assemblyIdentity name="MyLib" publicKeyToken="4CDF7733C53947CF" culture="neutral" />
            <codeBase version="2.0.0.0" href="http://www.weroSoft.net/MyLib.dll"/>
        </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Listing 1.9: *Verweis für eine Codebasis auf eine webbasierte Lokation*

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <dependentAssembly>
          <assemblyIdentity name="MyLib" culture="neutral" publicKeyToken="4CDF7733C53947CF" />
          <codeBase version="1.0.0.0" href="file://c:\MyAssemblyPath\MyLib.dll"/>
       </dependentAssembly>
    </assemblyBinding>
  </runtime>
</configuration>
```

Listing 1.10: *Verweis für eine Codebasis auf eine absolute lokale Quelle*

Hierarchische Suche der Assembly

Wird eine Assembly auch nach Zuhilfenahme der Prüfung eines Verweises auf die Codebasis nicht gefunden, leitet die CLR eine systematische hierarchische Suche im Dateisystem ein. Dabei werden folgende Schritte vorgenommen (Reihenfolge ist relevant):

1. Prüfen des Verzeichnisses der Anwendungsbasis. Die Anwendungsbasis ist definiert als Verzeichnis von welchem das Programm gestartet wurde, das den Prozessraum begründet hat (ausführbare Datei).
2. Unterverzeichnisse der Anwendungsbasis mit dem Namen des Sprachattributes der Assembly prüfen (nähere Angaben zu Lokalisierung entnehmen Sie Kapitel 9)
3. Unterverzeichnis mit dem selben Namen wie die gesuchte Assembly im Verzeichnis der Anwendungsbasis prüfen.
4. Pfadangaben der Anwendungskonfigurationsdatei im Verzeichnis der Anwendungsbasis prüfen.

Der letzte Punkt der obenstehenden Aufzählung weist erneut auf einen möglichen Eintrag der Anwendungskonfigurationsdatei hin. Dieser gestaltet sich gemäß dem Beispiel in **Listing 1.11**. Das Beispiel konfiguriert die zwei zusätzlichen Pfade *\bin* und *\resource* innerhalb der Anwendungsbasis.

```
<?xml version="1.0"?>
<configuration>
   <runtime>
     <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
       <probing privatePath="bin;resource"/>
     </assemblyBinding>
   </runtime>
</configuration>
```

Listing 1.11: *Beeinflussung der Assemblysuche in der Anwendungskonfigurationsdatei*

Profitipp Beginn

Dieses Kapitel beschreibt eine Vielzahl von Möglichkeiten für die Einflussnahme des Suchvorgangs von Assemblys. Ich habe selbst schon die eine oder andere Möglichkeit genutzt und kann sagen, dass das auch wunderbar funktioniert. Trotzdem sollten Sie die Installationen der Anwendungen, die Sie herstellen, so einfach wie möglich halten, denn jede zusätzliche Konfiguration ist eine mögliche Fehlerquelle und kompliziert zudem die Fehlersuche.

Ich warne insbesondere vor der Anwendung von Kombinationen für das Versionsrouting. Da wird dann schell unklar, welche Version effektiv gesucht wird.

Bei Fehlern in komplizierteren Verhältnissen des Ladenvorgangs von Assemblys empfehle ich zudem das Werkzeug *Assembly Binding Log Viewer* für die Beobachtung des Ladeverhaltens der CLR zu verwenden. Starten Sie zu diesem Zweck das Programm *fuslogvw.exe*.

Konfigurieren Sie anschließend mittels der Schaltfläche *Settings…* die Einstellungen für das Aufzeichnen des Ladeverhaltens der CLR (Nummer 1 in **Abbildung 1.51**). Die Auswahl der vollständigen Aufzeichnung der Geschehnisse in einen definierten eigenen Pfad hat sich in der Praxis als taugliches Mittel erwiesen (Nummer 2 und 3 in **Abbildung 1.51**).

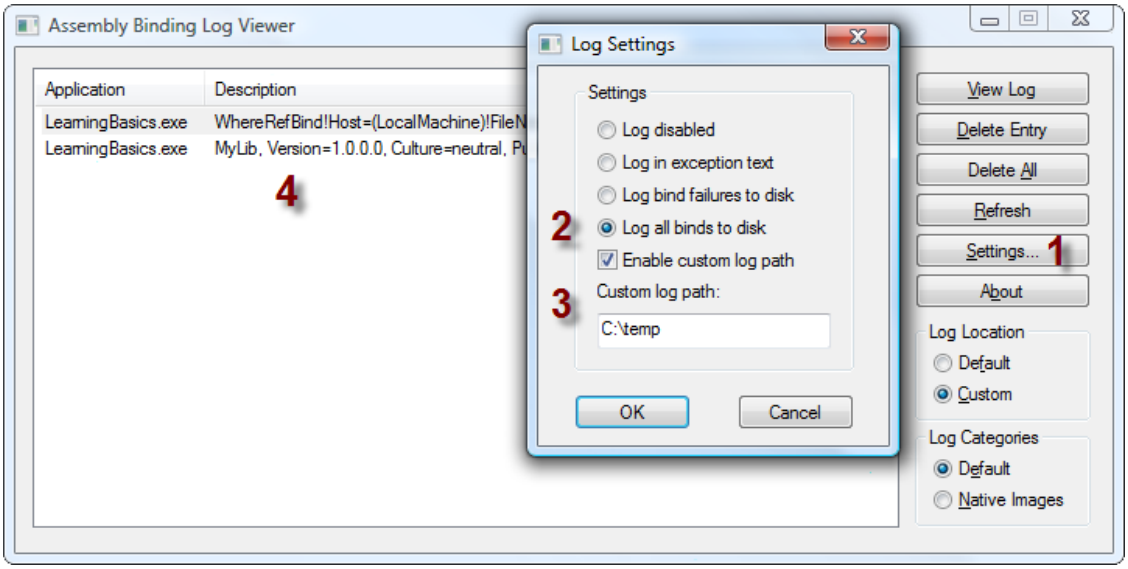

Abbildung 1.51: *Assembly Binding Log Viewer*

Nach der Konfiguration des *Assembly Binding Log Viewer* führen Sie Ihre Anwendung bis zum Auftreten des Ladefehlers aus. Die CLR hat bei der Ausführung das Laden von Assemblys im konfigurierten Pfad festgehalten. Nun können Sie im Werkzeug *Assembly Binding Log Viewer* nach dem Betätigen der Schaltfläche *Refresh* die entsprechenden Einträge finden, und mit einem Doppelklick auf die entsprechende Zeile die Details dazu lesen (Nummer 4 in **Abbildung 1.51**).

```
*** Assembly Binder Log Entry  (05.12.2009 @ 19:06:22) ***
The operation failed.
Bind result: hr = 0x80070002. Das System kann die angegebene Datei nicht finden.

Assembly manager loaded from:  C:\Windows\Microsoft.NET\Framework64\v4.0.21006\clr.dll
Running under executable
    C:\Data\Book_2\Code\Band_1\Kapitel_01\LearningBasics\LearningBasics\bin\Debug\LearningBasics.exe

=== Pre-bind state information ===
LOG: User = WEROSOFT\Rolf
LOG: DisplayName = MyLib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=d3cc48784446d496 (Fully-specified)
LOG: Appbase = file:///C:/Data/Book_2/Code/Band_1/Kapitel_01/LearningBasics/LearningBasics/bin/Debug/
LOG: Initial PrivatePath = NULL
LOG: Dynamic Base = NULL
LOG: Cache Base = NULL
LOG: AppName = LearningBasics.exe
Calling assembly : LearningBasics, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null.
===
```

```
LOG: This bind starts in default load context.
LOG: No application configuration file found.
LOG: Using host configuration file:
LOG: Using machine configuration file from
     C:\Windows\Microsoft.NET\Framework64\v4.0.21006\config\machine.config.
LOG: Post-policy reference: MyLib, Version=1.0.0.0, Culture=neutral, PublicKeyToken=d3cc48784446d496
LOG: The same bind was seen before, and was failed with hr = 0x80070002.
ERR: Unrecoverable error occurred during pre-download check (hr = 0x80070002).
```

Listing 1.12: *Die detaillierten Angaben des Logviewers helfen relativ rasch, das Problem zu lösen*

Vergessen Sie nicht, nach der Untersuchung den *Assembly Binding Log Viewer* wieder auszuschalten. Das kann ansonsten die Maschine mit der Zeit ganz schön lähmen!

Profitipp Ende