Kapitel 5

Assemblies und Deployment

5.1	**Assemblies in .NET**	**670**
5.2	**Konfiguration von .NET Applikationen**	**698**
5.3	**Deployment – was ist das?**	**712**

Nachdem die Grundlagen der komponentenorientierten Entwicklung in .NET erläutert worden sind, wird in diesem Kapitel ein genauerer Blick auf die interne Struktur der erzeugten Programme und Bibliotheken geworfen. Dabei wird ebenfalls das *Deployment* behandelt, d.h. wie die geschriebenen .NET-Anwendungen und Komponenten verteilt und installiert werden können.

Es wurde bereits erwähnt, dass .NET auch Lösungen für die Versionierungs- und DLL-Konflikte bereithält, die Programmierer, Administrator und Anwender zu gleichen Teilen betreffen. In vorhergehenden Kapiteln wurden die Probleme bereits diskutiert (siehe Kapitel 1 und 3). Hier folgt nun eine detaillierte Erläuterung dessen, was Assemblies sind, wie sie aufgebaut sind und warum sie gerade so aussehen wie sie aussehen, und wozu sie verwendet werden.

5.1 Assemblies in .NET

Egal, welche Projekt-Vorlage benutzt wird, ob nun z.B. ein benutzerdefiniertes Steuerelement, eine Web-Anwendung oder ein Windows-Dienst, beim Kompilieren erzeugt ein von der Entwicklungsumgebung aufgerufener und der verwendeten Sprache entsprechender Compiler ein *Assembly*. Prinzipiell besitzen alle Assemblies den gleichen Aufbau, sie unterscheiden sich nur (von der Implementierung einmal abgesehen) in der Dateiendung und dem Vorhandensein und Nicht-Vorhandensein eines Einsprungpunkts.

Assemblies machen das oft zitierte *XCopy-Deployment* möglich. Bei dieser Art der Verteilung von Applikationen reduziert sich die Installation eines .NET-Programms auf das einfache Kopieren der kompletten Anwendung inklusive aller Verzeichnisse, Unterverzeichnisse, referenzierten Assemblies und externen Ressourcen-Dateien. Zur Deinstallation müssen die entsprechenden Dateien und Verzeichnisse lediglich wieder gelöscht werden.

Dieses Vorgehen trifft auf lokal auf einem Computer installierte Anwendungen und deren Komponenten zu. Werden beispielsweise eine Web-Anwendung genutzt oder Assemblies über das Internet hinweg referenziert, werden die darin enthaltenen Komponenten nicht lokal installiert, wodurch auch nicht die Notwendigkeit entstehen kann, diese wieder zu deinstallieren.

5.1.1 Prinzipieller Aufbau eines Assemblies

Ein Assembly ist die kleinste verteilbare Einheit in .NET und enthält wesentlich mehr als nur den Code. Zusätzlich sind umfangreiche Informationen über das Assembly selbst und den darin enthaltenen Code vorhanden. Alle Informationen und der Code selbst sind Teil der so genannten *Metadaten*. Diese Metadaten sind der Grund, warum Assemblies als *selbstbeschreibende* Anwendungen bzw. Bibliotheken bezeichnet werden. Die zusätzlichen Informationen über den Code werden als *Manifest* bezeichnet.

Der Code liegt in den Assemblies in *IL – Intermediate Language* – vor. Dies ist das geforderte Format, das die Runtime benötigt, um ein .NET-Programm bzw. -Bib-

liothek laden und ausführen zu können. Zusammen mit allen enthaltenen Informationen dient ein Assembly als fundamentale Einheit zur Versions-Kontrolle, als Baustein zur Wiederverwendung, als Einheit zur Abgrenzung der Gültigkeit von Typen (*Scoping*) und als leicht zu handhabendes Objekt (hier nicht im Sinne von Objektorientierung) im Sicherheitsmechanismus von .NET.

Außer der Unterscheidung in ausführbare Assemblies und Bibliotheken kann auf anderer Ebene unterschieden werden zwischen statischen und dynamischen Assemblies. Auf anderer Ebene bedeutet dies in diesem Zusammenhang, dass ein Assembly unabhängig vom Typ (ausführbares Programm oder Bibliothek) zum Kompilierungszeitpunkt oder zur Laufzeit erzeugt werden kann.

Die verschiedenen Arten von Assemblies

Prinzipiell lassen sich Assemblies zunächst einmal in ausführbare Programme und Bibliotheken (Komponenten) unterscheiden. Darüber hinaus gibt es jedoch noch weitere Unterscheidungsmerkmale:

Statische Assemblies

Ein *statisches Assembly* ist die Art von Assembly, mit der ein Programmierer oder Anwender in aller Regel arbeitet bzw. in Kontakt kommt. Nachdem ein .NET-Programm oder eine Bibliothek implementiert und kompiliert worden ist, liegt auf der Platte physikalisch eine Datei vor. Dies ist unabhängig von der Art, wie kompiliert worden ist, ob mit dem jeweiligen Compiler über die Eingabeaufforderung oder mit Visual Studio .NET. Diese Datei ist ein statisches Assembly und besteht aus einem oder mehreren Code-Modulen und eventuell vorhandenen Ressourcen-Dateien. Zur Laufzeit werden das Assembly selbst und alle referenzierten Assemblies und Ressourcen zum entsprechenden Zeitpunkt geladen.

Dynamische Assemblies

Im Gegensatz zu den statischen Assemblies gibt es die *dynamischen Assemblies*. Sie werden erst zur Laufzeit einer anderen .NET-Anwendung erzeugt. Dies kann via Skript-Code auf einer Web-Seite oder durch Verwendung von *Reflection.Emit* geschehen. Dynamische Assemblies sind *transient*, das bedeutet, sie werden normalerweise nicht persistent abgelegt, also nicht auf der Festplatte gespeichert.

Dennoch besteht die Möglichkeit dies zu tun, was beispielsweise von der ASP .NET Laufzeitumgebung so gehandhabt wird. Dort wird jeweils ein Assembly für jede angeforderte Web-Seite dynamisch auf einem Web Server generiert und in einem Unterverzeichnis des Framework-Verzeichnisses (*%Windows%\Microsoft.NET\Framework\<Version>\Temporary ASP.NET Files*) temporär abgespeichert (siehe Abschnitt über ASP .NET in *Kapitel 4*).

Mittels *Reflection* in .NET ist es auch möglich, ein Programm sprachenunabhängig im Code zu *beschreiben* und zur späteren Verwendung als reinen Quell-Code im Textformat oder in Form eines Assemblies (dann in IL) zu speichern. Dabei wird das so genannte *Code Dokumenten Objektmodell* (*Code Document Object Model – CodeDOM*) verwendet (auch ASP .NET nutzt dieses Modell). Der wichtigste

Anwendungsfall für die dynamische Code-Erzeugung ist eine Template-basierte Codegenerierung, wie sie außer bei ASP .NET auch in der Entwicklungsumgebung Visual Studio .NET und dessen Assistenten (*Wizards*) Verwendung findet.

Private und öffentliche Assemblies

Völlig unabhängig von der Implementierung der Assemblies gibt es noch eine Unterscheidung in *private* und *öffentliche* (*shared*) Assemblies. Private Assemblies befinden sich physikalisch immer im Verzeichnis der referenzierenden .NET-Anwendung oder einem lokalen Unterverzeichnis der Applikation. Öffentliche Assemblies sind etwas komplizierter zu behandeln und werden in den *Global Assembly Cache* (*GAC*) installiert.

Der GAC ist ein globales Verzeichnis, in das Assemblies hinein kopiert (installiert) werden können. Dort können sie ähnlich wie klassische DLLs von mehreren Anwendungen gleichzeitig verwendet werden. Im Unterschied zu den altbekannten DLLs benötigen sie jedoch keinerlei Einträge in der System-Registrierung (*Registry*).

Metadaten und das Manifest

Ein Assembly kann ein sehr komplexes und kompliziertes Gebilde sein. Deshalb beinhaltet es neben dem Code noch umfangreiche Informationen, die als *Manifest* bezeichnet werden. Dieses Manifest wird von der Runtime benötigt, um den Code laden, kompilieren (von IL in den ausführbaren Code) und ausführen zu können.

Die Informationen werden von der CLR benötigt, um

- weiteren Code finden zu können, auf den im Assembly verwiesen wird. Dies betrifft z.B. Aufrufe von Methoden, die von anderen Assemblies zur Verfügung gestellt werden. Deshalb ist eine Liste mit allen Namen aller Dateien des Assemblies enthalten.
- die Verweise auflösen zu können. Dazu sind alle Namen und Metadaten aller Assemblies und Dateien, von denen das aktuelle Assembly abhängig ist, Teil des Manifests.
- eine Versions-Kontrolle zu ermöglichen. Dazu beinhaltet das Assembly Informationen über sich selbst, aber auch über die Assemblies, auf die verwiesen wird.
- Typen für andere Assemblies zur Verfügung stellen zu können. Die Informationen über Klassen, Interfaces, Strukturen, etc. sind von außen von anderen Assemblies abrufbar.

Erst das Manifest ermöglicht es, dass ein Assembly aus mehreren Dateien bestehen kann. Außerdem wird dank der darin enthaltenen Informationen keine Registrierung in der Windows-Registrierungsdatenbank benötigt. Einen ersten Einblick in ein Assembly und sein Manifest ermöglicht die Datei *AssemblyInfo.cs* beziehungsweise *.vb*, die per Doppelklick im Projektmappen-Explorer eines bestehenden Projekts geöffnet werden kann. Das folgende Listing zeigt ein Beispiel einer solchen Datei, die aus einer Reihe von Attributen besteht. Auf einige der darin ent-

haltenen Attribute wird später noch genauer eingegangen werden, so z.B. auf die Versionsinformationen in Abschnitt Versionsmechanismus von Assemblies.

Listing 5.1: Beispiel einer Datei AssemblyInfo.vb, die Informationen über das Assembly enthält wie z.B. den Firmennamen des Herstellers (Zeile 7).

```
 1:  Imports System.Reflection
 2:  Imports System.Runtime.InteropServices
 3:
 4:  [...]
 5:  <Assembly: AssemblyTitle("Konsolenanwendung 1")>
 6:  <Assembly: AssemblyDescription("")>
 7:  <Assembly: AssemblyCompany("newtelligence AG")>
 8:  <Assembly: AssemblyProduct("")>
 9:  <Assembly: AssemblyCopyright("jmf")>
10: <Assembly: AssemblyTrademark("")>
11: <Assembly: CLSCompliant(True)>
12:
13: [...]
14: <Assembly: Guid("ACEC2C7A-6C7C-40DC-A97B-6F1E408FDDC8")>
15:
16: [...]
17: <Assembly: AssemblyVersion("1.0.0.42")>
```

Die hierin enthaltenen Attribute können auch über verschiedene Dialoge und Eigenschaften in der IDE eingestellt werden, lassen sich jedoch natürlich auch hier direkt im Code manipulieren. Alle gemachten Einträge werden beim Kompilieren in ein Assembly übernommen. Diese Informationen können unter Verwendung des Disassemblers *ildasm.exe* eingesehen werden.

ASSEMBLER UND DISASSEMBLER

Das .NET Framework bietet zwei zusätzliche Werkzeuge für die Kommandozeile, die dazu verwendet werden können, Assemblies zu inspizieren und selbst geschriebenen IL-Code in ein Assembly zu verwandeln.

DER DISASSEMBLER ILDASM.EXE

Der Disassembler ildasm.exe ist ein grafisches Tool, das über die Kommandozeile gestartet werden kann. Es ist zu finden im Verzeichnis *%Microsoft Visual Studio .NET%\FrameworkSDK\Bin*.

Das Tool bietet die Möglichkeit, .NET-Assemblies (Dateien mit den Endungen *.exe, .dll, .mod, .mdl*) einer sehr genauen Inspektion zu unterziehen und liefert u.a. tiefe Einblicke in die Eigenschaften eines Assemblies, die darin enthaltenen Verweise auf externe Assemblies, den Quellcode im IL-Format und vieles mehr.

Unter dem Menüpunkt FILE ist der Eintrag OPEN zu finden. Bei Auswahl öffnet sich ein Dialog, der einen Benutzer bei der Suche nach der zu öffnenden .NET Komponente unterstützt.

Das Tool ist eine unverzichtbare Hilfe, wenn es darum geht, die Interna eines Assemblies zu erforschen. Es bietet die Möglichkeit, ein Manifest zu studieren, einschließlich des IL-Codes eines Assemblies. Dabei gibt es auch eine Einstellung im Tool, wodurch der Disassembler den IL-Code direkt als Quell-Code-Zeilen anzeigen kann. Dabei wird der im Assembly enthaltene IL-Code in den Quellcode der jeweiligen Sprache zurück transformiert. Die angezeigten Informationen (inkl. IL- und Quellcode) können in einer Textdatei abgespeichert werden (Hauptmenü FILE\DUMP).

Damit lässt sich also fremder Code inspizieren. Das mag für einen Programmierer eine unangenehme Erfahrung darstellen, ist jedoch nichts Neues. Denn bereits für die bisherigen Programme in z.B. C/C++ gibt es entsprechende Werkzeuge. Diese Tatsache sollte jedoch von der richtigen Seite aus betrachtet werden: Nun ist es möglich, auf recht einfache und zuverlässige (!) Art und Weise festzustellen, ob der eigene Code unerwünscht an anderer Stelle wieder verwendet worden ist.

DER ASSEMBLER ILASM.EXE
Das Kommandozeilentool *ilasm.exe* nimmt eine Textdatei oder einen Stream, bestehend aus IL-Code, als Eingabe und generiert daraus ein Assembly inklusive der notwendigen Metadaten.

Eine einfachere Möglichkeit, Informationen wie z.B. die Produkt-Version oder den Hersteller über ein Assembly zu erhalten, bietet jedoch der Windows-Explorer, indem dort per Rechtsklick das Eigenschaften-Fenster eines Assemblies geöffnet wird:

Abbildung 5.1: Der Dialog Eigenschaften im Windows-Explorer bietet Einsicht in Informationen über das Assembly.

Als Teil des Manifests stellen sich dieselben Informationen in etwa so dar:

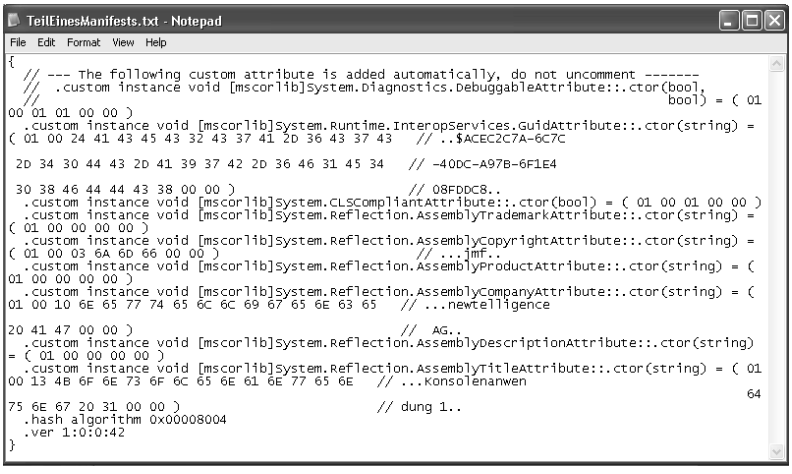

Abbildung 5.2: Ein Ausschnitt aus den in einem Manifest enthaltenen Informationen

Die Informationen, die in einem Manifest enthalten sind, beziehen sich auf alle Bestandteile eines Assemblies und umfassen:

- **Assembly-Name**: Ein einfacher Text (String), der den Namen des Assemblies repräsentiert.
- **Versionsinformationen**: Die Versionsnummer, bestehend aus vier Teilen (Build-, Revisions-, Major- und Minor-Nummer), wird von der Runtime zur Durchsetzung des Versionsmechanismus verwendet (siehe *Abschnitt 5.1.3*).
- **strong name**: Dieser besondere Name eines Assemblies besteht aus einem öffentlichen Schlüssel des Herstellers, einem mit der Signatur des Herstellers signiertem Hash-Wert der Datei, die das Manifest enthält.
- **Sprachenunterstützung bzgl. Internationalisierung**: Assemblies können mit Informationen bestückt werden, um feststellen zu können, welche Sprachen unterstützt werden. Damit kann beispielsweise für eine Bibliothek spezifiziert werden, ob sie auf einem englischen Betriebssystem mit englischen Sprach-Einstellungen genutzt werden kann. Dies ist beispielsweise beim Einlesen und Verarbeiten von Geldbeträgen wichtig. Auf einem Computer mit deutschen Spracheinstellungen (Die *CultureInfo* hat dort den Wert *de-DE*) werden die Trennungszeichen für Dezimalstellen anders interpretiert als bei US-amerikanischen Einstellungen (*en-US*). Im ersten Fall wird ein Wert von *51.299_* als *51299_* (Einundfünfzigtausendzweihundertneunundneunzig Euro) interpretiert, während sie im zweiten Fall – bei US-amerikanischen Einstellungen – eine *51.299_* (Einundfünfzig Euro und Neunundzwanzig Komma Neun Cents) repräsentiert. Das liegt in diesem Beispiel daran, dass im deutschen Zahlensys-

tem der Punkt dazu dient, die Tausender-Stellen zu markieren, im amerikanischen Zahlensystem dies jedoch das Dezimaltrennzeichen ist.

- **Prozessor und Betriebssystem**: Die Informationen über den Prozessor und das Betriebssystem sind in dieser Version des .NET Framework von geringerem Nutzen. Sie sind für diejenigen Versionen des Framework von Interesse, die auf anderen Betriebssystemen als Windows und auf den entsprechenden Prozessoren ausgeführt werden können.
- **Liste aller Dateien des Assemblies**: Diese Liste besteht aus den Hash-Werten jeder Datei, die Teil des Assemblies ist, und einer Pfad-Angabe je Datei relativ zur Manifest-Datei.
- **Verweise zu Typen und Ressourcen**: Diese Informationen werden von der CLR benutzt, um eine Implementierung bezüglich eines Verweises zu einem Typen finden zu können.
- **Verweise zu anderen Assemblies**: Eine Liste aller Assemblies, die statisch referenziert werden. Jeder Verweis enthält den Namen des entsprechenden Assemblies, Metadaten (Version, Internationalisierung, Betriebssystem, etc.) und evtl. einen öffentlichen Schlüssel.

Assemblies und Module

Eine Datei, die Code, also Typen wie z.B. Klassen oder Interfaces enthält, kann in .NET als ein *Modul* kompiliert werden. Ein solches Modul beinhaltet den Code in IL, die darin implementierten Typen und dazugehörenden Metadaten. Es kann jedoch nicht von der Runtime geladen und ausgeführt werden. Dazu werden zusätzliche Informationen benötigt, die dann hinzukommen, wenn aus dem Modul ein Assembly gemacht wird. Ein Assembly besteht aus einem oder mehreren Modulen, wobei die meisten Assemblies in der Regel lediglich ein (1) Modul enthalten.

Ein einfaches Kommandozeilen-Programm zur Ausgabe der Zeichenkette »Hallo Welt.« auf die Konsole sieht in Visual Basic .NET etwa so aus:

```
Module Module1

    Sub Main()
        System.Console.WriteLine("Hallo Welt.")
    End Sub

End Module
```

Visual Studio .NET bietet keine Möglichkeit, ein Modul zu erstellen. Deshalb erzeugt die Entwicklungsumgebung beim Kompilieren ein vollständiges Assembly, inklusive umfangreicher Metadaten. Der gezeigte Code kann jedoch über die Kommandozeile in ein Modul kompiliert werden. Der entsprechende Aufruf sieht für Visual Basic .NET in etwa so aus:

```
vbc /r:System.dll /t:module Module1.vb
```

Dies ist der einzige Weg, ein Modul zu erstellen. Dies ist auch der einzige Weg, der zu einem *Multifile*-Assembly führt; einem Assembly also, das aus mehreren Dateien besteht. Zum Zeitpunkt der Fertigstellung dieses Buches gab es keine Entwicklungsumgebung (auch nicht Visual Studio .NET), welche die Erstellung dieser Art von Assemblies unterstützt. Vielleicht gibt es in naher Zukunft jedoch IDEs von anderen Anbietern als Microsoft, die entsprechendes leisten können.

Für den Fall, dass dies notwendig sein sollte, muss auf die entsprechenden Kommandozeilen-Tools zurückgegriffen werden. So sind alle .NET-Kommandozeilen-Compiler in der Lage, Module zu generieren. Um mehrere Module und/oder Assemblies miteinander zu verlinken und ein einzelnes Multifile-Assembly zu erzeugen, wird das Tool *al.exe* – der *Assembly-Linker* – verwendet.[1] Dabei werden auch die noch fehlenden Informationen erzeugt, um das Manifest des Assemblies zu vervollständigen. Für das gezeigte Modul kann der entsprechende Befehl so aussehen:

```
al /t:exe Module1.netmodule /out:Module1.exe
   /main:Module1.Main
```

Das zuerst generierte Modul mit dem Namen *Module1.netmodule* stellt sich im Disassembler so dar:

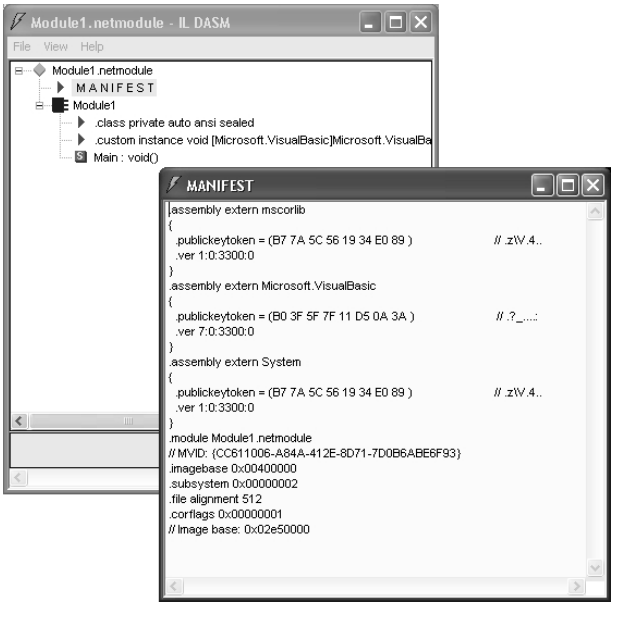

Abbildung 5.3: Auch ein Modul enthält ein Manifest – jedoch mit wesentlich weniger Informationen, als ein vollständiges Assembly.

1 Das Tool ist im Verzeichnis *%Framework%\bin* zu finden. Das Standardverzeichnis für *%Framework%* ist z.B. unter WindowsXP *C:\WINDOWS\Microsoft.NET\Framework\<Version>*.

Der Assembly-Linker erzeugt ein Assembly, das einen Verweis auf das Modul *Module1.netmodule* enthält. Nun muss zur Ausführungszeit dieses Modul aber für die ausführbare Datei (*.exe*) erreichbar sein. Die exe-Datei muss zusammen mit dem Modul verteilt werden!

Eine weitere wichtige Erweiterung, die der Assembly-Linker erzeugt, ist der Verweis zum Eintrittspunkt für das Programm. Dieser Eintrittspunkt ist eine Methode, die beim Starten der Anwendung ausgeführt werden soll. Sie wird dem Tool al.exe als Parameter */main:* übergeben und spezifiziert eine Methode in einem der zu verbindenden Module oder Assemblies.

5.1.2 Abhängigkeiten, Verweise und Namespaces

Von außen betrachtet ist ein Assembly eine Ansammlung von Ressourcen, die zum Zweck der Wiederverwendung exponiert werden. Die Ressourcen eines Assemblies beinhalten auch Typen und werden durch einen Namen nach außen getragen.

Von der anderen Seite – also aus der Sicht des Assemblies – stellt sich ein Assembly als eine Ansammlung von öffentlichen (*exponierten*) und privaten (*internen*) Typen dar. Das Assembly selbst entscheidet darüber, welche Typen und Ressourcen von außen sichtbar sind und welche nur innerhalb genutzt werden können. Auch die Kontrolle darüber, wie auf die eigentliche Implementierung referenziert wird, obliegt dem aktuellen Assembly.

Abhängigkeiten

Von Abhängigkeiten ist im Zusammenhang mit Assemblies dann die Rede, wenn ein Verweis auf Ressourcen oder Typen im Scope eines anderen Assemblies vorhanden ist. Damit ist ein Assembly von einem anderen Assembly abhängig, wenn z.B. eine Methode aufgerufen oder eine Bild-Datei (bmp, gif, etc.) referenziert wird. Alle Verweise werden unter Kontrolle des aktuellen Assemblies aufgelöst. Damit bietet sich die Möglichkeit, einen Verweis unter Einbeziehung von Versions-Informationen oder anderen Charakteristika zu bilden. Das referenzierte Assembly kann jedoch bestimmen, wie dieser Verweis auf die eigentliche Implementierung geleitet (*gemappt*) wird.

Ein Assembly kann allein stehend verteilt werden, wobei davon ausgegangen wird, dass alle weiteren benötigten Assemblies und Ressourcen auf dem Ziel-Computer bereits vorhanden sind. Die andere Lösung ist, alle notwendigen Dateien zusammen mit der Anwendung mitzuliefern.

Abhängigkeiten werden immer zur Laufzeit aufgelöst; es gibt kein statisches Linken. Zu beachten ist der Unterschied zwischen statischen und dynamischen Verweisen gegenüber statischem und dynamischem Linken. Direkte Verweise zu einem Typ in einem Stück Code sind statisch. Es ist jedoch auch möglich, dynamische Verweise einzubringen, was über den Reflection-Mechanismus geschieht. *Linken* ist ein Begriff, der die tatsächliche, physikalische Bindung zwischen Ressourcen betrifft.

Die folgenden Zeilen Code laden ein Assembly dynamisch – zur Laufzeit:

```
System.Reflection.Assembly BeispielAssembly;
BeispielAssembly =
      System.Reflection.Assembly.Load("System.Drawing");
```

Typen-Verweise

Verweise zu Typen wie Klassen oder Interfaces können unterschiedliche Gültigkeitsbereiche überschreiten. Ein Assembly trägt die Informationen über die eigenen Typen mit sich und kann so einem Benutzer die Arbeit wesentlich erleichtern.

Module haben eine Liste aller eigenen Typen; und zwar der *internen* bzw. *privaten* Typen, die nur innerhalb des Moduls verwendet werden sollen und können. Außerdem enthält diese Liste auch Einträge der Typen, die von außen sicht- und nutzbar sein sollen. Dies sind die *öffentlichen* (*shared*) Typen. Das Assembly selbst hat ebenfalls eine solche Liste mit den privaten und den öffentlichen Typen. Öffentliche Typen eines Moduls haben einen Eintrag in der Liste des entsprechenden Moduls, aber auch in der Typen-Liste des Assemblies, in dem es enthalten ist. Dort ist dann zusätzlich festgehalten, ob es ein interner oder öffentlicher Typ bezüglich des Assemblies ist. Somit ist eindeutig festgelegt, welche Klassen, Interfaces, etc. nur in einem Modul, welche zwischen Modulen –, aber nur innerhalb des Assemblies – und welche Typen außerhalb eines Assemblies bekannt sein sollen.

Der einfachste Verweis ist der auf einen Typen innerhalb desselben Moduls. Die Auflösung des Verweises beschränkt sich auf ein Überprüfen, ob der Typ als interner Typ im selben Modul vorhanden ist. Die Abbildung 5.4 zeigt einen solchen Verweis in Modul A (Assembly 1, Referenz auf Typ A) als erste Zeile in der »Referenzliste«.

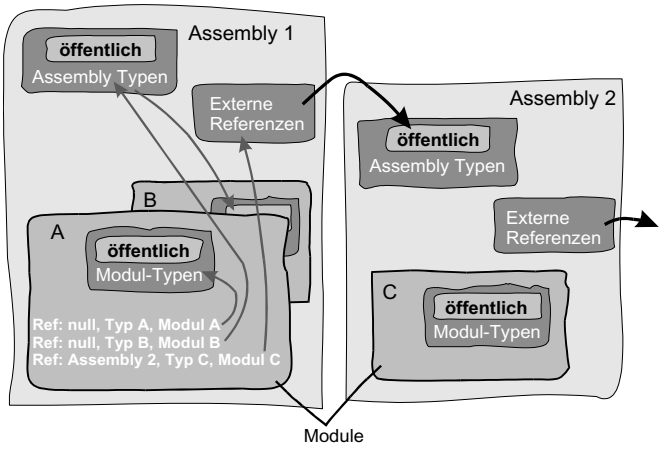

Abbildung 5.4: Es gibt verschiedene Arten von Typ-Referenzen in .NET. Dazu gehören Referenzen innerhalb von Modulen, innerhalb von Assemblies und zwischen Assemblies.

Die zweite Art von Verweisen ist die auf einen Typen in einem anderen Modul, aber innerhalb desselben Assemblies. In der Abbildung dargestellt als Zeile 2 in der »Referenzliste« (Referenz auf Typ B in Modul B). Dieser Verweis wird aufgelöst, indem die Liste der Assembly-Typen entsprechend durchforstet wird.

Die dritte Art von Verweisen ist eine Referenz auf einen Typen in einem anderen Modul, das Bestandteil eines anderen Assemblies ist. In der Abbildung ist dies als Referenz auf Typ C in Modul C in Assembly 2 dargestellt.

Zusammenhang zwischen Assemblies und Namespaces

Das Prinzip der Namespaces wurde bereits eingeführt und erläutert. Namespaces dienen zur Gruppierung von zusammenhängenden Typen. Dies ist unabhängig vom Ort, an dem diese Typen (physikalisch) implementiert sind; Namespaces dienen rein der logischen Gruppierung.

Damit sind im .NET Framework Typen, die jeweils thematisch bzw. funktional zusammengehören, durch ein hierarchisch strukturiertes Namenssystem in Kategorien geordnet. Entwicklungstools wie Visual Studio .NET können von den Namespaces Gebrauch machen, um es einem Programmierer einfacher zu gestalten, die große Menge an Typen zu benutzen und nach den gesuchten Funktionalitäten zu forschen.

Auch die selbst implementierten Typen wie z. B. eigene Klassen liegen immer (!) in einem Namespace. Auch wenn in Visual Basic .NET nicht immer ein Namespace im Code angegeben wird, so liegen die dort implementierten Klassen in einem vorgegebenen *Default*-Namespace, der über den Dialog EIGENSCHAFTEN des aktuellen Projekts eingestellt werden kann. Der dort spezifizierte Namespace findet sich dann auch im Manifest des Assemblies wieder. Kurz gesagt: Alle Typen in .NET liegen in einem Namespace!

Wo befindet sich nun jedoch die eigentliche Implementierung der Typen eines Namespace? Da das Konzept *Namespace* orthogonal zu dem der Assemblies liegt, kann ein einzelnes Assembly mehrere Namespaces implementieren, ein Namespace kann sich jedoch gleichzeitig über mehrere Assemblies erstrecken.

Listing 5.2: Implementierung zweier Namespaces innerhalb ein und derselben Datei

```
 1:  using System;
 2:
 3:  namespace myNamespace
 4:  {
 5:     class myClass
 6:     {
 7:        [STAThread]
 8:        static void Main(string[] args)
 9:        {
10:          Console.WriteLine("Hallo Welt!");
```

```
11:           otherNamespace.otherClass.CallMe();
12:           Console.ReadLine();
13:        }
14:     }
15: }
16:
17: namespace otherNamespace
18: {
19:     class otherClass
20:     {
21:        public static void CallMe()
22:        {
23:           Console.WriteLine("Methode CallMe().");
24:        }
25:     }
26: }
```

Das Listing 5.2 zeigt einen C#-Quellcode. Diese 26 Zeilen stehen in ein und derselben Datei und werden bei Kompilierung auch zu einem einzigen Assembly führen, welches dann beide Namespaces enthält:

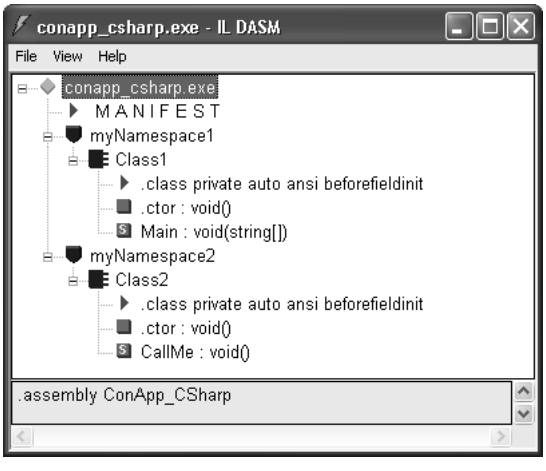

Abbildung 5.5: Ein Assembly kann mehrere Namespaces implementieren.

Wie gesagt kann ein Namespace auch auf mehrere Assemblies verteilt werden. Jedes der beteiligten Assemblies kann einen Teil des entsprechenden Namespaces implementieren. Ein einzelner Typ kann dabei jedoch nicht aufgeteilt und beispielsweise in zwei verschiedenen Assemblies implementiert werden. Auch im .NET Framework gibt es mehrere solcher Namespaces. So erstreckt sich der Namespace *System.Windows.Forms* im Wesentlichen über die beiden Assemblies *System.Windows.Forms.dll* und *System.Design.dll*.

Zur Unterscheidung von Typen wird jedoch nicht nur der Namespace (und der Name) herangezogen; auch das Assembly spielt dabei eine wichtige Rolle. Ein Typ, der zweimal innerhalb desselben Namespace, jedoch in verschiedenen Assemblies implementiert ist, wird als zwei unterschiedliche Typen betrachtet und entsprechend von der CLR behandelt. Das implementierende Assembly ist Bestandteil der Identität eines Typs.

Die Trennung der Konzepte von Namespace und Assembly hat zur Folge, dass beides in einem Projekt referenziert werden muss, wenn ein bestimmter Typ verwendet werden soll. Der Namespace ist eine (logische) Namenskonvention, die zum Entwicklungszeitpunkt (design-time) eingehalten werden muss. Das Assembly, das den benutzten Typen implementiert, muss dann zur Laufzeit vorhanden sein und bildet den Gültigkeitsbereich des Typs.

5.1.3 Konzepte und Lösungen

Hier werden nun Details der Konzepte des Manifests eines Assemblies erläutert, wie z. B. der Versionsmechanismus oder welche Rolle die Informationen über ein Assembly im Sicherheitssystem spielen.

Versionsmechanismus von Assemblies

Jedes Assembly trägt eine spezielle Versionsnummer, die zur Bestimmung der Kompatibilität bezüglich früherer Versionen dient. Diese Versionsnummer ist so wichtig, dass zwei Assemblies, die sich lediglich in diesem Punkt unterscheiden als zwei völlig unterschiedliche Assemblies von der CLR behandelt werden.

Die Versionsnummer besteht physikalisch aus vier Nummern, die durch Punkte getrennt dargestellt werden:

```
Major.Minor.Build.Revisions
```

Eine Nummer wie beispielsweise *2.7.42.0* besagt, dass dieses Assembly der *42*te Build der Version mit der Major-Version *2* und der Minor-Version *7* – Revision 0 – ist. Jede einzelne dieser Nummern hat eine spezielle Bedeutung, die dann eine Rolle spielt, wenn die CLR ein bestimmtes Assembly laden soll.

Die ersten beiden Nummern (Major und Minor) bilden den Teil der Versionsnummer, der eine Inkompatibilität mit anderen Versionen kennzeichnet. Jede Änderung einer dieser beiden Nummern signalisiert der Runtime, dass diese Version inkompatibel ist zu Assemblies mit anderen Major- und Minor-Nummern. Änderungen dieser Nummern machen deutlich, dass es sich bei dem vorliegenden Assembly um eine neue Release-Version handelt.

Die Build-Nummer kennzeichnet ein Assembly als *vielleicht kompatibel* zu anderen Versionen. Eine Änderung dieser Nummer signalisiert der Runtime, dass zwar Änderungen an dem Assembly vorgenommen worden sind, diese jedoch ein geringes Risiko bezüglich der Kompatibilität enthalten.

Werden lediglich sehr kleine Änderungen bzw. Korrekturen am Code vorgenommen, die einen *Quick-Fix* (schnelle Fehlerbehebung) darstellen, wird die letzte Nummer – die Revisions-Nummer – geändert. Eine neue Revisions-Nummer signalisiert der Runtime, dass diese Version des Assemblies auf jeden Fall als voll kompatibel zu anderen Versionen betrachtet und behandelt werden kann.

Die Versionsnummer ist Teil des Manifests:

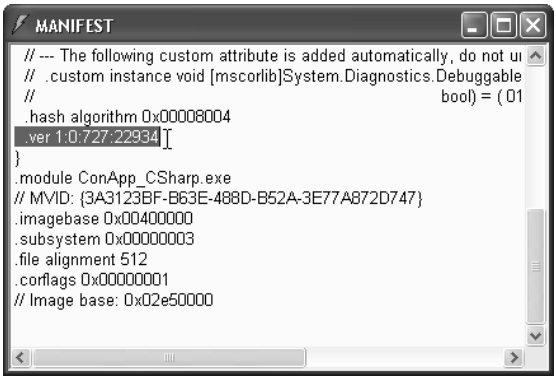

Abbildung 5.6: Die Versionsnummer eines Assemblies ist Bestandteil des Manifests.

Sie kann im Code in der Datei *AssemblyInfo.cs* bzw. *.vb* eines Projekts fest vorgegeben werden. Es ist dabei auch möglich, Wildcards zu verwenden, wie das folgende Beispiel als Änderung zum Listing 5.1. zeigt:

```
17: <Assembly: AssemblyVersion("2.7.*")>
```

Wenn ein Assembly kompiliert wird, legt der Compiler die Versionsnummern aller externen Assemblies im aktuellen Assembly ab. Diese Informationen werden von der CLR später benutzt, um die passende Version eines referenzierten Assemblies zu laden. Dabei benötigt die Runtime eventuell ein wenig Unterstützung durch Konfigurationseinstellungen eines Administrators, die entweder auf Maschinen- oder auf Applikationsebene gemacht werden können.

Zu beachten ist, dass dieser Versionsmechanismus jedoch nur mit Assemblies funktioniert, die einem so genannten *strong name* haben (siehe *Abschnitt Ein starker Name*).

Gleichzeitige Ausführung

Bei der *gleichzeitigen Ausführung* (*side-by-side execution*) handelt es sich darum, dass Assemblies gleichen Namens, aber unterschiedlicher Versionsnummer gleichzeitig genutzt werden können. Die CLR bietet die entsprechenden Leistungsmerkmale, um verschiedene Versionen eines Assemblies auf einer Maschine und sogar innerhalb desselben Prozesses auszuführen. Komponenten, die diese

side-by-side Ausführung zulassen, sind flexibler, was ihre Rückwärtskompatibilität betrifft.

Ein Assembly, welches z.B. zusammen mit einer bestimmten Version einer .NET-Klassenbibliothek kompiliert worden ist, wird immer versuchen, diese spezifische Version der Bibliothek zu referenzieren, mit der kompiliert worden ist. Wie viele neuere Versionen der Bibliothek auch immer auf demselben Computer installiert werden, ohne spezielle Konfiguration oder eine Neukompilierung des Assemblies wird immer versucht, die alte Version zu benutzen.

Die Unterstützung für die side-by-side Ausführung durch die CLR ist ein essentieller Bestandteil des Versionsmechanismus. Der Schlüssel zu dieser Technologie ist, dass die Informationen als Teil des Assemblies selbst abgelegt werden können.

Es wurde bereits erwähnt, dass unterschiedliche Versionen nicht nur auf einem Computer, sondern sogar innerhalb eines Prozesses verwendet werden können: Beides erfordert jedoch besondere Beachtung durch den Programmierer, der das Assembly entwickelt (siehe *Abschnitt 5.2.2*).

Wenn z.B. eine Datei von einer Bibliothek dazu verwendet wird, temporäre Daten abzulegen, müssen mehrere Versionen des Assemblies in besonderer Weise implementiert werden, um eine Kollision des konkurrierenden Zugriffs der unterschiedlichen Bibliotheken zu vermeiden. Entweder muss die Datei beim Zugriff durch eine Bibliothek entsprechend gesperrt werden oder es müssen mehrere Dateien – je eine pro vorhandene Version – angelegt werden.

Bei einer parallelen Ausführung innerhalb desselben Prozesses entstehen höhere Anforderungen an die Programmierung. Für eine erfolgreiche side-by-side Ausführung dürfen keine Abhängigkeiten von prozessweit verwendeten Ressourcen bestehen. Der Vorteil einer solchen Verwendung unterschiedlicher Versionen eines Assemblies liegt darin, dass die Funktionalität neuerer Versionen genutzt werden kann, der alte Code jedoch nicht neu geschrieben werden muss.

Ein großer Nachteil, den dieser Mechanismus der Versionsverwaltung von Komponenten mit sich bringt, ist der erhebliche Speicherplatzbedarf. Die verschiedenen Versionen der Komponenten liegen immerhin physikalisch als Dateien auf der Festplatte. Wenn diese jeweils als lokale Datei innerhalb des Verzeichnisses einer Anwendung liegen, gibt es auf einem Computer unter Umständen mehrere Kopien ein und derselben Komponente. Wird die Komponente jedoch in den GAC eingespielt, müssen entsprechende Konfigurationsdateien für die verschiedenen Anwendungen mitgeliefert werden, damit auch auf die kompatiblen Versionen der referenzierten Komponenten verwiesen werden kann. Damit verlagert sich das Problem der Registrierungseinträge auf eine Administration der mehr oder weniger zahlreichen Konfigurationsdateien.

Sicherheitsbetrachtungen

Die Informationen, die im Manifest eines Assemblies abgelegt sind, können auch im Zusammenhang mit der Sicherheit genutzt werden.

Die CLR muss sicherstellen können, dass das Assembly und alle enthaltenen Dateien nicht manipuliert worden sind. Zwischen dem Kompilieren des Assemblies und der Verwendung (zur Laufzeit) darf niemand den Code geändert haben. Die Integrität aller zu ladenden und auszuführenden Assemblies muss garantiert werden. Dies wird durch die Generierung von Hash-Werten gewährleistet, die eine Quersumme über alle Bytes einer jeweiligen Datei darstellen.

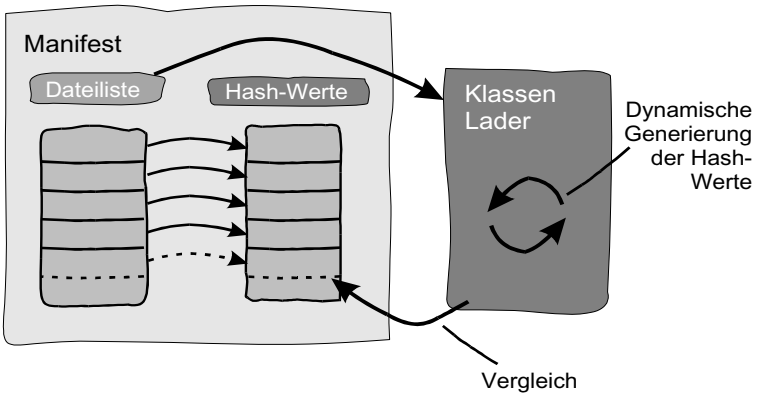

Abbildung 5.7: Die Liste aller Dateien eines Assemblies und deren Hash-Werte werden von der CLR verwendet, um zur Laufzeit die Integrität zu überprüfen.

Neben einer Liste aller Dateien des Assemblies ist auch eine Liste mit Hash-Werten enthalten. Zur Laufzeit, wenn der Klassen-Lader ein Assembly lädt, werden erneut Hash-Werte für alle Dateien des Assemblies generiert und mit den Hash-Werten im Manifest verglichen. Wenn Abweichungen festgestellt werden, muss davon ausgegangen werden, dass Manipulationen an Dateien durchgeführt worden sind.

Die CLR bestimmt den Satz an Permissions, der einem Assembly angeheftet werden kann und wodurch festgelegt ist, was einem Assembly erlaubt wird zu tun. Zum Zeitpunkt des Implementierens kann ein Programmierer einen minimalen Satz an Permissions für ein Assembly spezifizieren, der unbedingt notwendig ist, damit das Assembly geladen und ausgeführt werden kann, und überhaupt etwas Sinnvolles machen kann. Zusätzliche Zugriffsrechte können auf dem entsprechenden Computer pro Assembly konfiguriert werden. Der Satz an Permissions, den ein Assembly zur Ausführung benötigt und der dementsprechend von diesem angefordert werden muss (*permission request*), ist ebenfalls Teil des Manifests.

Zum Ladezeitpunkt werden die Anforderung und der Umfang der angeforderten Permissions als Eingabe für das Sicherheitsregelwerk (*policy*) der CLR benutzt. Im Zusammenhang mit weiteren sicherheitsrelevanten Informationen (Signatur, Herkunft, etc.) kann die Runtime entscheiden, was einem Assembly in dem aktu-

ellen Ausführungskontext erlaubt werden kann und was nicht. Zu den sicherheitsrelevanten Informationen gehört auch der *strong name* eines Assemblies:

Ein starker Name

Der Name eines Assemblies hat einen wesentlichen Einfluss auf seinen Gültigkeitsbereich (*scope*) und seine Verwendung durch andere Applikationen.

Ein Assembly, das für die Benutzung durch lediglich einen einzigen Anwender entwickelt worden ist und im Applikations-Verzeichnis mit ausgeliefert wird, benötigt in aller Regel keine weitere besondere Behandlung. Zu beachten ist hierbei jedoch, dass es im .NET Framework keinen Mechanismus gibt, der vor Namenskollidierungen schützt.

Code jedoch, der dazu bestimmt ist, von mehreren Anwendungen gleichzeitig benutzt zu werden, braucht strengere Bestimmungen bezüglich der Namensvergabe für ein Assembly. Die Lösung ist ein starker Name für ein Assembly: der so genannte *strong name*.

Dieser strong name ist jedoch auch nichts anderes als ein einfacher Text (String), der sich aus dem Namen des Assemblies, einem öffentlichem Schlüssel und einer digitalen Signatur zusammensetzt.

Er wird über die Datei des Assemblies selbst generiert, und zwar die Datei, die das Manifest des Assemblies enthält. (Diese Datei enthält auch die Liste aller Dateien des Assemblies und die Hash-Werte.) Bei der Generierung wird außerdem noch ein dazugehöriger öffentlicher Schlüssel benötigt. Auch Visual Studio .NET kann einen *strong name* für ein Assembly erzeugen.

Der *strong name* schützt ein Assembly davor, dass der implementierte Namespace von anderen erweitert oder übernommen werden kann. Denn nur derjenige, der einen privaten Schlüssel passend zum öffentlichen Schlüssel im *strong name* besitzt, kann den entsprechenden Namen generieren. Ein einzelnes Assembly, das mit unterschiedlichen Schlüsseln kompiliert wird, erhält jeweils einen anderen *strong name*.

Ein *strong name* schützt ebenfalls die Versions-Reihe eines Assemblies. Das bedeutet, ein *strong name* kann sicherstellen, dass kein anderer Programmierer außer dem Entwickler des ursprünglichen Codes eine Folge-Version eines Assemblies erstellen kann. Ein späterer Benutzer kann dann sicher sein, dass er auf jeden Fall ein Assembly in einer neueren Version bekommen hat, dass von demselben Hersteller stammt.

Assemblies mit unterschiedlichen *strong names* werden von der CLR als völlig unterschiedliche Assemblies betrachtet und dementsprechend behandelt. Angenommen, eine .NET-Applikation ist zusammen mit einer bestimmten Version eines Assemblies, das einen *strong name* besitzt, kompiliert worden. Liefert der Hersteller zu einem späteren Zeitpunkt eine neue kompatible Version des Assemblies (die sich lediglich in der Revisions-Nummer und ein paar kleinen Fehlerkor-

rekturen unterscheidet) mit einem anderen *strong name*, kann die Applikation jedoch nur unter Verwendung des alten Assemblies benutzt werden.

Zwei Assemblies, die den gleichen *strong name* haben, werden als identisch betrachtet. Es ist sehr (sehr!) unwahrscheinlich, dass für zwei unterschiedliche Assemblies mit verschiedenen Implementierungen derselbe *strong name* generiert werden könnte.

Es wurde bereits erwähnt, dass der Versionsmechanismus nur mit Assemblies funktioniert, die einen *strong name* haben. Der Grund dafür ist, dass der Versionsmechanismus eigentlich nur mit Assemblies funktioniert, die im *Global Assembly Cache* (*GAC*) installiert sind. Ein Assembly kann jedoch nur dann im GAC installiert werden, wenn es einen *strong name* hat.

Ein starkes Tool für einen starken Namen

Die Entwicklungsumgebung Visual Studio .NET generiert einen *strong name* beim Kompilieren eines Assemblies. Doch zuvor muss ein Schlüsselpaar (privat/öffentlich) vorhanden sein. Zur Generierung dieser Schlüssel gibt es ein Kommandozeilen-Tool *sn.exe*, das im Verzeichnis *%FrameworkSDK%\bin*[2] zu finden ist. *sn* steht für *strong name*.[3]

Das Tool bietet aber auch weitere Möglichkeiten, um beispielsweise die Schlüssel zu verwalten, Signaturen zu generieren und zu verifizieren. Um ein Schlüsselpaar zu erzeugen, kann auf der Konsole der folgende Befehl eingegeben werden:

```
sn -k meinKey.snk
```

Es wird dabei ein Schlüsselpaar in der Datei *meinKey.snk* angelegt.

Weitere wichtige Kommandozeilen-Parameter für sn.exe sind in der nachstehenden Tabelle wiedergegeben.

Kommandozeilen-Parameter	Beschreibung
-k	Generieren eines neuen Schlüsselpaares und speichern in einer Schlüssel-Datei.
-e	Extrahieren des öffentlichen Schlüssels aus einer Schlüssel-Datei in eine zu spezifizierende Datei.

Tabelle 5.1: Auswahl an Kommandozeilen-Parameter des strong name-Tools sn.exe

2 *%FrameworkSDK%* entspricht standardmäßig z.B. *C:\Programme\Microsoft Visual Studio .NET\FrameworkSDK*.

3 Der frühere Name war *Shared*-Name, wenn also irgendwo (z.B. in der Dokumentation) dieser Begriff auftaucht, kann davon ausgegangen werden, dass ein *Strong*-Name gemeint ist.

Kommandozeilen-Parameter	Beschreibung
-R	Erneute Zuweisung eines *strong name* zu einem Assembly, das bereits einen solchen Namen (vollständig oder teilweise) hatte.
-t/-T	Anzeige eines Ausschnitts (*Token*) aus dem öffentlichen Schlüssel einer Schlüssel-Datei (-t) oder eines Assemblies (-T).
-v	Verifizierung eines *strong name*

Tabelle 5.1: Auswahl an Kommandozeilen-Parameter des strong name-Tools sn.exe (Forts.)

Liegt das Schlüsselpaar erst einmal vor, ist es recht einfach, einem Assembly einen *strong name* zuzuweisen. Dies kann zum einen über Attribute geschehen, wie es unter Verwendung der Datei *AssemblyInfo.cs* bzw. *.vb* gemacht werden kann. Als Beispiel dient erneut eine Erweiterung zum Listing 5.1:

```
18: <Assembly: AssemblyKeyFile("../../meinKey.snk")>
```

Eine zweite Möglichkeit ist die Nutzung eines weiteren Kommandozeilen-Tools: dem Assembly-Linker *al.exe*. Der Befehl

```
al ConApp_CSharp.netmodule /keyf:meinKey.snk
```

generiert ein *strong name* Assembly für das Modul *ConApp_CSharp.netmodule* unter Verwendung der Schlüssel-Datei *meinKey.snk*. Das Tool kann nicht auf Assemblies angewandt werden!

Vorschläge zur Handhabung des Schlüsselpaares

Der öffentliche Schlüssel wird im Manifest des Assemblies abgelegt. Der private Schlüssel ist nicht im Manifest zu sehen, das jedoch mit dem vollständigen öffentlichen Schlüssel signiert wird. Diese Signatur wiederum ist Teil des Manifests.

Wenn nun eine Client-Anwendung ein Assembly mit einem *strong name* referenziert, gibt es dabei zunächst keinen Unterschied zu einem Verweis auf ein privates Assembly. Wenn jedoch ein Compiler den IL-Code des Clients erzeugt, wird ein *Token* des öffentlichen Schlüssels des referenzierten Assemblies ins Manifest des Clients eingetragen. Dieses Token ist ein Hash-Wert des vollständigen privaten Schlüssels.

Die CLR benutzt entsprechende kryptografische Verfahren, um sicherzustellen, dass das korrekte Assembly geladen worden ist. Wenn ein Verweis auf ein Assembly mit einem *strong name* zur Laufzeit aufgelöst werden soll, vergleicht die CLR das Token im Manifest des Konsumenten (Client) mit dem öffentlichen Schlüssel des referenzierten Assemblies. Soll ein Assembly mittels Reflection (*Assembly.Load*) geladen werden, muss das Token bei dem Verweis im Code mit angegeben werden.

Auf der Seite eines Clients, z.B. einem Benutzer einer .NET-Bibliothek, kann auf diese Weise kontrolliert werden, ob ein Assembly von einem bestimmten Hersteller stammt und es dementsprechend vertrauenswürdig ist. Es ist dann nicht mehr möglich, dass sich ein Dritter in die Auslieferungskette vom Bibliotheks-Hersteller zum Client einklinkt und letzterem schlechten, böswilligen Code unterjubelt. Er muss nämlich dann im Besitz des Schlüsselpaars des Bibliotheks-Herstellers sein, um denselben *strong name* erzeugen zu können.

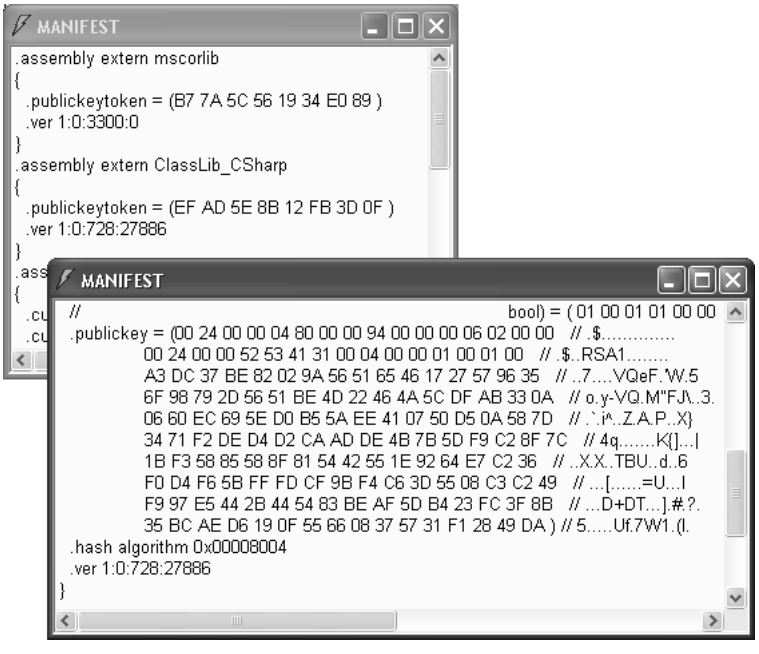

Abbildung 5.8: Ein Client-Assembly referenziert ein anderes strong name-Assembly unter Verwendung eines Tokens des öffentlichen Schlüssels des strong name-Assemblies.

Deshalb ist zu empfehlen, ein Schlüsselpaar nur einmal für eine Firma zu generieren (evtl. pro Projekt) und diese Datei – vor allem den *privaten* Schlüssel – gut unter Verschluss zu halten!

Wie aber sollen dann die Programmierer in der Entwicklungsabteilung mit ihrem Code weiterarbeiten, wenn das Schlüsselpaar für sie nicht zugreifbar ist? Wenn dort mit den *strong name*-Assemblies gearbeitet werden soll, werden öffentlicher und privater Schlüssel benötigt. Dazu gibt es ebenfalls eine entsprechende Lösung: das *verzögerte Signieren* (*delayed signing*).

Zunächst einmal werden die betreffenden Assemblies nicht vollständig, sondern nur teilweise signiert. Dabei wird nur der öffentliche Schlüssel verwendet. Für den

privaten Schlüssel und die vollständige Signierung wird entsprechend Platz reserviert. Der öffentliche Schlüssel kann mit dem sn.exe-Tool aus der Schlüssel-Datei herausgefiltert werden. Der Befehl

```
sn -p meinKey.snk meinPublicKey.snk
```

generiert eine neue Datei *meinPublicKey.snk*, die dann nur den öffentlichen Schlüssel der Datei *meinKey.snk* enthält.

Diese neue Datei wird genauso verwendet wie die ursprüngliche Datei mit beiden Schlüsseln. In der Datei *AssemblyInfo.cs* bzw. *.vb* wird der neue Name der Datei in dem entsprechenden Attribut *[assembly:AssemblyKeyFile]* eingetragen. Dort ist auch ein weiteres Attribut zu finden, mit dem das verzögerte Signieren zu aktivieren ist. Dazu muss der dort spezifizierte Wert nach *true* geändert werden: *[assembly: AssemblyDelayedSign(true)]*.

Wird ein Assembly dann kompiliert, ist es nur teilweise signiert, besitzt jedoch einen *strong name* und kann in den *GAC* (*Global Assembly Cache*) installiert und dort verwendet werden. Zuvor muss aber noch die Verifikation des Schlüssels ausgeschaltet werden. Dies kann explizit für ein einzelnes Assembly geschehen. Das Tool *sn.exe* bietet auch hier wieder Unterstützung unter Verwendung der Option `-Vr`:

```
sn -Vr meinAssembly.dll.
```

Hat die Entwicklungsabteilung den ersten Teil ihrer Arbeit beendet und den Code fertig gestellt, kann ein Assembly mit dem kompletten Schlüsselpaar vollständig signiert werden:

```
sn -R meinAssembly.dll meinKey.snk
```

Nun muss nur noch die Verifikation für das Assembly wieder eingeschaltet werden (*sn.exe* mit Option `-Vu`):

```
sn -Vu meinAssembly.dll
```

Anmerkungen zum Global Assembly Cache

Natürlich muss ein lokal auf einem Computer installiertes Assembly einer .NET-Applikation nicht zwangsweise als *Shared-Assembly* in den Global Assembly Cache installiert werden. Es gibt aber verschiedene Gründe, die für eine Verwendung des GAC sprechen:

- **Versionierung**: Mehrere Versionen eines Assemblies können in den GAC parallel eingestellt werden.
- **Performance-Vorteile**: Die Performance einer Anwendung wird bei Verwendung des GAC auf verschiedene Arten verbessert. Zunächst einmal findet für Assemblies, die im GAC installiert sind, keine Verifikation statt, wodurch

sich bei häufiger Nutzung des Assemblies ein Zeitvorteil ergibt. Ein zweiter Vorteil ist, dass von einem Assembly im GAC nur eine einzige Instanz gebildet wird. Dadurch wird bei mehrfacher Benutzung durch mehrere Clients der Ladevorgang beschleunigt. Ein dritter Grund ist, dass ein Assembly von der Runtime schneller gefunden wird.

- **Überprüfung der Integrität**: Beim Installieren eines Assemblies in den GAC wird die Integrität jeder Datei des Assemblies überprüft.
- **Datei-Sicherheit**: Nur ein Administrator mit entsprechenden Rechten auf einer Maschine kann Assemblies in den GAC einstellen. Auch alle anderen Verwaltungsaufgaben bezüglich des GAC können nur von einem System-Account aus vorgenommen werden, der mit Administrator-Rechten versehen ist. Dazu ist jedoch mindestens ein Betriebssystem wie WindowsNT, Windows 2000, WindowsXP oder höher erforderlich, denn andere Versionen bieten keine Dateisicherheit.

5.1.4 Assemblies zur Laufzeit

Dieser Abschnitt betrachtet die Behandlung von Assemblies durch die Runtime zur Laufzeit. Es wird erläutert, wie Assemblies geladen werden, was eine Applikations-Domäne ist und wozu sie gut ist, wie Assemblies zur Laufzeit von IL in den ausführbaren Code transformiert werden und wie dies auch zum Installationszeitpunkt ereicht werden kann.

Eine Domäne – eine Domäne für ein Assembly

Die CLR stellt für eine .NET-Anwendung einen geschützten, isolierten und verwalteten Ausführungsraum bereit. Dieser Bereich wird als *Applikations-Domäne* bezeichnet. Sie profitieren vom Sicherheitssystem des Framework und sind die Entität, auf die jeweils ein bestimmter Satz an Sicherheits-Regeln (*policies*) angewandt wird.

Betriebssysteme und Laufzeitumgebungen stellen für ein Programm immer einen (mehr oder weniger) isolierten Ausführungs-Bereich zur Verfügung. Diese Isolierung, die eine Anwendung auf einem bestimmten System erfährt, ist notwendig, um sicherstellen zu können, dass der Code, der in einem Bereich ausgeführt wird, nicht von Code in einem anderen Bereich beeinflusst oder gestört wird.

Typischerweise bedeutet die Isolierung von Programmen, dass:

- Fehler, die eine Anwendung schlimmstenfalls zum Absturz bringen, eine andere Anwendung mitreißen.
- Anwendungen unabhängig voneinander gestartet, gestoppt oder debugged werden können.
- Code der einen Anwendung unter keinen Umständen Code oder Ressourcen einer anderen Anwendung antasten kann.

- das Verhalten von ausgeführtem Code beschränkt ist auf die eine Anwendung, in der er gerade ausgeführt wird.

Moderne Betriebssysteme schaffen eine solche Isolation, indem Prozessgrenzen festgelegt werden. In einem Prozess ist genau eine Anwendung eingebettet, und er bestimmt auch über die Ressourcen, die für die Anwendung erreichbar sind. Für ein Win32-Programm bedeutet das z.B., dass Speicheradressen immer nur auf einen Prozess bezogen sind. Ein Zeiger im Code in einem Prozess ist nutzlos im Code eines anderen Prozesses.

Die Runtime in .NET verlässt sich darauf, dass .NET-Code typensicher ist. In .NET *spielt niemand auf der Wiese des Nachbarn*. Dadurch kann eine Isolierung mit weniger Kosten bewerkstelligt werden, als beispielsweise die Prozess-Isolation in Win32.

.NET-Anwendungen werden erzeugt, indem zusammengehörende und verbundene Assemblies zusammen geladen und ausgeführt werden. Anders als bei Win32-Programmen wird eine Anwendung in .NET von einer so genannten *Applikations-Domäne* (*application domain*) gehostet. Es ist wichtig im Hinterkopf zu behalten, dass eine Applikations-Domäne nicht (!) das .NET-Gegenstück zu einem Win32-Prozess ist.

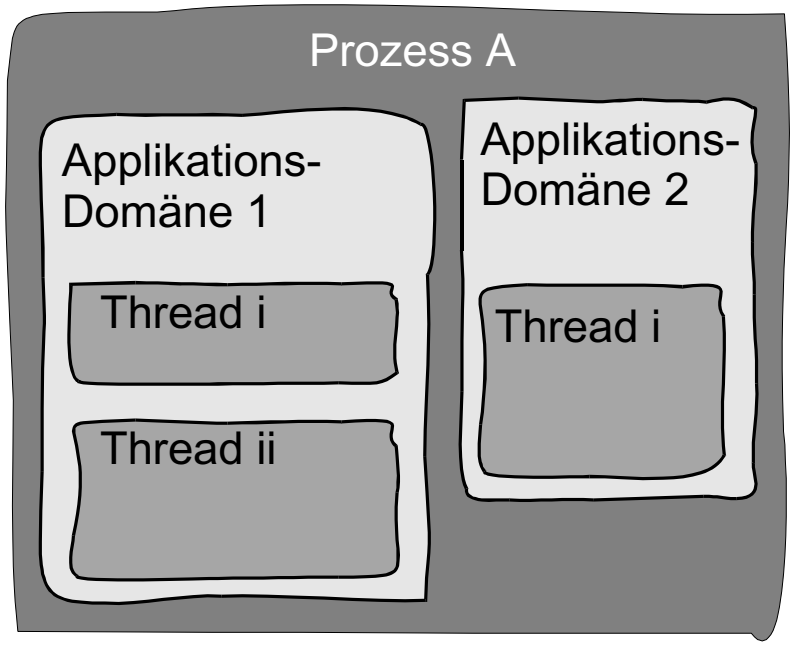

Abbildung 5.9: Ein Prozess kann eine oder mehrere Applikations-Domänen enthalten, von denen jede wiederum einen oder mehrere Threads beinhalten kann.

Jeder Prozess kann eine beliebige Anzahl von Applikations-Domänen hosten, wovon jede voll isoliert von anderen ist – auch innerhalb desselben Prozesses. Anwendungen, die in unterschiedlichen Applikations-Domänen ausgeführt werden, ist es unmöglich irgendwelche Arten von Daten oder Informationen auszutauschen, außer über .NET-Remoting.

In .NET kann ein einzelner Prozess mehrere Applikations-Domänen hosten. Und jede Applikations-Domäne kann einen oder mehrere Threads beherbergen. Applikations-Domänen werden programmatisch vom Typ *System.AppDomain* repräsentiert (zu Prozesse und Threads siehe *Kapitel 3*).

Das folgende kleine Programm listet alle in der aktuellen Applikations-Domäne geladenen Assemblies auf der Kommandozeile auf.

Listing 5.3: Mittels .NET-Reflection lassen sich Informationen über den Code selbst herausfinden; so auch eine Liste aller Assemblies, die in der aktuellen Applikations-Domäne geladen sind.

```
 1: using System;
 2: using System.Reflection;
 3:
 4: namespace appdomaene
 5: {
 6:    class AssemblyKlasse
 7:    {
 8:       [STAThread]
 9:       static void Main(string[] args)
10:      {
11:         AppDomain myDomain = AppDomain.CurrentDomain;
12:         Assembly[] geladeneAssemblies =
13:               myDomain.GetAssemblies();
14:         Console.WriteLine("Dies ist eine Liste "
15:               + "aller geladenen Assemblies in "
16:               + "dieser Domaene:");
17:         Console.WriteLine();
18:
19:         foreach(Assembly assembly
20:               in geladeneAssemblies)
21:         {
22:            Console.WriteLine(assembly.FullName);
23:         }
24:         Console.ReadLine();
25:      }
26:   }
27: }
```

Der Namespace *System.Reflection* wird benötigt, um auf die Klasse *AppDomain* zugreifen zu können. Diese Klasse hat eine Eigenschaft *CurrentDomain* (Zeile 11), die die aktuelle Applikations-Domäne für den laufenden Thread zurückgeben kann. Damit lässt sich die Methode *GetAssemblies()* aufrufen, die eine Liste aller Assemblies der Domäne liefert (Zeilen 12/13). In der *foreach*-Schleife in den Zeilen 19 bis 23 werden die Namen dieser Assemblies auf die Kommandozeile ausgegeben.

Abbildung 5.10: Ausgabe des Beispiel-Programms appdomaene.exe aus Listing 5..

Das Laden eines Assemblies

Alle ausführbaren .NET Assemblies (exe-Dateien) sind Standard-PE-Dateien; so genannte *portable executables*, die vom Betriebssystem ladbar sind. Traditionelle PE-Dateien haben einen *Header*. Traditionelle PE-Header beinhalten Informationen darüber, wo Code und Daten im Speicher lokalisiert werden können, für welches Betriebssystem die Anwendung bestimmt ist oder über die initiale Größe des *Stack* (Speicherbereichs).

Zusätzlich zu dem Standard-Header einer PE-Datei beinhaltet ein .NET-Assembly einen speziellen Header für die CLR. Dieser enthält spezielle Informationen wie die Metadaten-Tabelle, den IL-Code und weitere Daten.

Die CLR wird normalerweise von einem Host, wie z.B. ASP .NET, dem Internet Explorer oder der Windows Shell gestartet und gemanaged. Ein solcher Host bietet eine Umgebung, um .NET-Code zu laden und auszuführen.

Aufgrund der Rückwärtskompatibilität mit existierenden Windows-Plattformen werden Runtime-Images als x86-Images gekennzeichnet. Dadurch wird es möglich, dass auch ein Betriebssystem, das in das .NET noch nicht integriert ist, eine .NET-Anwendung starten kann. Der Einstiegspunkt der .NET-PE-Datei ist ein Stück Code (*stub*), der die CLR startet und den tatsächlichen Start-Punkt für die .NET-Anwendung übergibt. Dieser Stub wird vom Betriebssystem ausgeführt.

Die CLR lokalisiert anschließend die zusätzlichen Informationen im Header der PE-Datei, erzeugt eine Applikations-Domäne und lädt das Assembly.

Normalerweise wird ein Assembly pro Applikations-Domäne geladen. Das bedeutet jedoch, dass ein Assembly genauso oft geladen und kompiliert wird, wie es gestartet wird. Assemblies können aber auch entsprechend gekennzeichnet werden, so dass sie von mehreren Applikationen verwendet werden können. Ein Assembly kann in alle Domänen eines Prozesses gemappt werden. Der Code wird

von einem JIT-Compiler nur einmal kompiliert und steht dann allen Applikations-Domänen eines Prozesses zur Verfügung. Die einzelnen Domänen bekommen dann jeweils nur eine eigene Kopie von globalen statischen Daten.

Ein Assembly so zu markieren, dass eine Kopie von mehreren Applikationen benutzt werden kann hat einige Vorteile. Wird ein Typ eines solchen Assemblies zum ersten Mal geladen, wird er automatisch in alle Applikations-Domänen gemappt und nicht in jede einzelne Domäne geladen. Dadurch benötigt das Assembly weniger Ressourcen. Außerdem beschleunigt ein solches Assembly die Erzeugung einer Applikations-Domäne, da die CLR den Code nicht immer wieder kompilieren und mehrere Kopien der Datenstrukturen für die Typen erzeugen muss. Zu beachten ist, dass ein solches Assembly nicht entladen wird, bevor der Prozess endet.

Während die Verwendung eines solchen Assemblies die Speicherbenutzung reduziert, entsteht eine größere Menge an JIT-Code (Code, der zur Laufzeit aus dem IL-Code entsteht).

Um ein Assembly als *MultiDomain*-Assembly zu kennzeichnen, ist ein Attribut zu verwenden. Dieses *LoaderOptimization*-Attribut kann nur auf der *Main()*-Methode des auszuführenden Assemblies angewandt werden. Der Code aus Listing 5.3 könnte dann so aussehen:

Listing 5.4: Erweiterung des Listings 5.3 um das Attribut LoaderOptimization zur Kennzeichnung des Assemblies als MultiDomain-Assembly

```
6:        [LoaderOptimization(
7:              LoaderOptimization.MultiDomain)]
8:        [STAThread]
9:        static void Main(string[] args)
```

Jitten und Pre-Jitten

Bevor der IL-Code eines Assemblies ausgeführt werden kann, wird er zur Laufzeit in den eigentlichen ausführbaren Code transformiert. Diese Aufgabe wird von einem *Just-In-Time*-Compiler (*JIT*) übernommen, der CPU-spezifischen Code erzeugt, und zwar speziell auf den Prozessor abgestimmt, auf dem auch der JIT-Compiler zur Laufzeit das Assembly übersetzt. Da die Runtime einen JIT-Compiler für jede CPU-Architektur bereithält, auf dem die Runtime ausgeführt werden kann, muss sich ein Programmierer mit seinem .NET-Code nicht speziell auf eine Architektur festlegen. Wenn im Code jedoch plattform-spezifische Funktionen oder Komponenten (z.B. Win32-API-Systemaufrufe, COM- bzw. ActiveX-Komponenten) aufgerufen werden, ist er auf diese Plattform bzw. Betriebssystem reduziert.

Der Befehlssatz von IL wurde entworfen, um als Input für die entsprechenden Compiler dienen zu können. IL ist aber kein traditioneller Byte-Code. Die Transformation von IL in den nativen Code erfordert eine Analyse des Codes pro Methode, weshalb IL tatsächlich eine Zwischensprache (*intermediate language*) ist.

Die Idee der JIT-Kompilierung beachtet auch die Tatsache, dass mancher Code zur Laufzeit niemals aufgerufen wird. Deshalb wird nicht der komplette IL-Code transformiert. Dadurch wird natürlich eine Menge an Zeit und Ressourcen eingespart. IL-Code wird also bei Bedarf in nativen Code transformiert und in dieser Form zwischengespeichert, damit er für nachfolgende Aufrufe in dieser Form schneller ausgeführt werden kann.

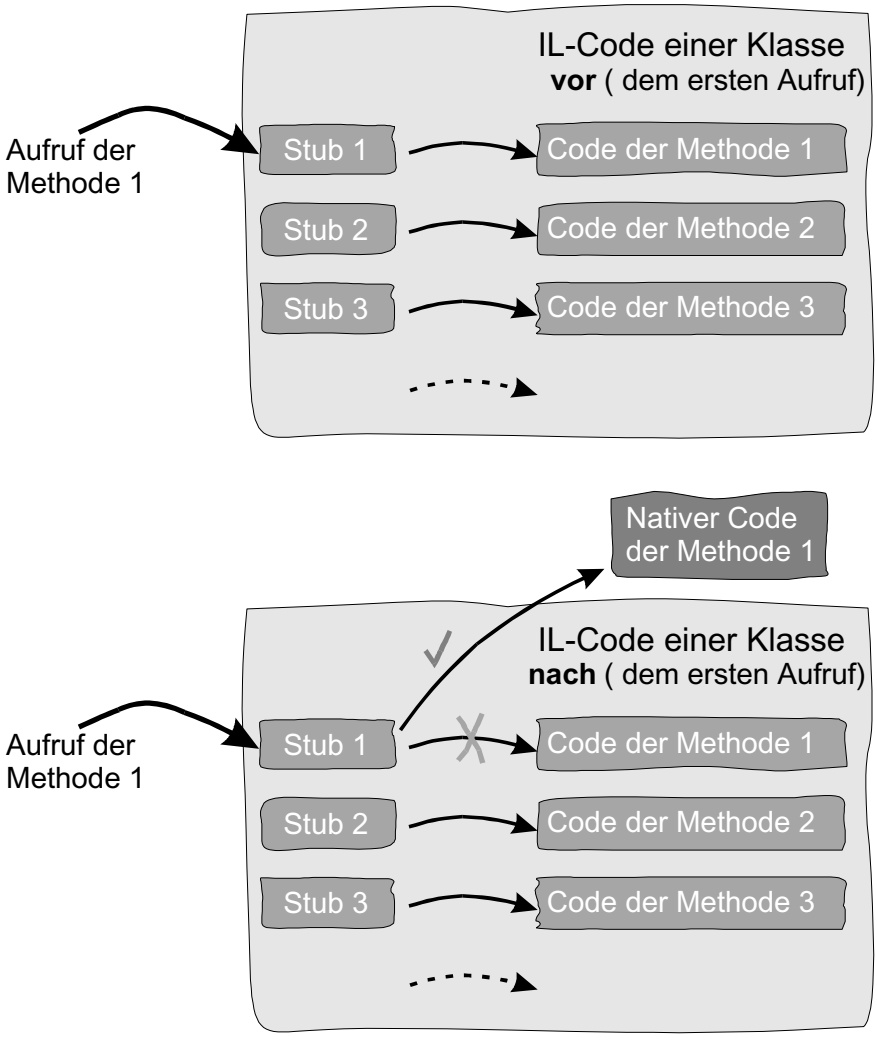

Abbildung 5.11: Vor dem ersten Aufruf verweist ein Stub auf den entsprechenden Code der Methode. Nach erfolgter JIT-Kompilierung leitet ein Stub jeden Aufruf direkt durch an den nativen Code der entsprechenden Methode.

Der Klassenlader erzeugt einen *Stub* für jeden Methodenaufruf eines Typs und heftet diesen an die entsprechende Stelle im Code. Der *Stub* ist ein Objekt, das speziell für eine Schnittstelle verwendet wird. Er entpackt bei einem Methodenaufruf die Parameter und leitet den Aufruf an die entsprechende Methode weiter. Das Entpacken ist dann notwendig, wenn der Aufruf zu einer entfernten Methode stattfindet und dieser Aufruf entsprechend der Übertragung zum entfernten Objekt formatiert wurde (*Marshalling*).

Beim ersten Aufruf einer Methode übergibt der Stub die Kontrolle an den JIT-Compiler, der den IL-Code für diese Methode in den nativen Code übersetzt. Anschließend wird der Stub entsprechend modifiziert, so dass alle nachfolgenden Aufrufe direkt zu dem bereits transformierten Code der Methode umgeleitet werden.

Bevor auch nur ein Zeichen von IL in nativen Code transformiert wird, wird dieser einem Verifikations-Prozess unterzogen. Dabei werden der IL-Code und die Metadaten daraufhin überprüft, ob sie typensicher sind. Typensicherheit ist notwendig, um sicherzustellen, dass Objekte mit Sicherheit voneinander isoliert sind. Ist der Code typensicher, kann die CLR von folgenden Annahmen ausgehen:

- Ein Verweis auf einen Typ ist kompatibel mit dem referenzierten Typen selbst.
- Nur vorher wohl definierte Operationen werden auf einen Typen ausgeführt.
- Identitäten sind gesichert.

Während des Verifikations-Prozesses wird der IL-Code ebenfalls daraufhin überprüft, ob er korrekt generiert worden ist, da inkorrekter IL-Code zu Verletzungen der Typensicherheit führen kann

Der Code, der vom JIT-Compiler erzeugt wird, wird lediglich im Speicher gecached und niemals auf eine Festplatte geschrieben oder in anderer Art und Weise persistiert – mit einer einzigen Ausnahme:

Wer einen sorgfältigen Blick auf den Assembly-Cache geworfen hat, konnte feststellen, dass dort einige Assemblies als *Native Images* gekennzeichnet sind. Das sind Assemblies, die bereits in nativer Form vorliegen und von einem JIT-Compiler aus IL-Code generiert worden sind.

Das Kommandozeilen-Tool *ngen.exe* kann dazu genutzt werden, ein Assembly manuell von IL in den ausführbaren Code zu transformieren. Allerdings wird das Assembly dabei direkt in den GAC installiert und kann nicht in dieser Form auf der Platte abgelegt werden.

Manche Assemblies, wie z. B. *System.dll* oder *System.Windows.Forms.dll* liegen in IL und *gejitet* vor. In manchen Situationen wird der bereits gejitete Code nicht verwendet:

- Wenn irgendein Identifizierer eines abhängigen Moduls nicht korrekt ist oder sich von dem im IL-Image unterscheidet.

- Wenn ein Administrator die Regeln zur Auflösung der entsprechenden Verweise auf einer Maschine geändert hat.
- Wenn Unterschiede in der referenzierten Version und der Referenz erkannt werden.
- Wenn eine neue Version der Runtime installiert worden ist.

5.2 Konfiguration von .NET Applikationen

Es wurde bereits mehrfach angedeutet, dass die Möglichkeit besteht, eine .NET Applikation in ihrem Verhalten durch Konfiguration zu beeinflussen. Damit kann das Binden einer Anwendung an verschiedene Assemblies dynamisch gesteuert werden. Die Konfiguration geschieht sowohl auf Maschinen- als auch auf Applikations-Ebene. Für beide Varianten müssen Dateien erstellt werden, die auf XML basieren.

5.2.1 Maschinenweite Konfiguration

Bei der Installation des .NET Frameworks wird eine Konfigurationsdatei mit dem Namen *machine.config* mit installiert. Sie liegt im Verzeichnis *%Windows%\Microsoft.NET\Framework\<version>\CONFIG*. Die Datei lässt sich – da XML – mit dem Internet Explorer öffnen und anschauen:

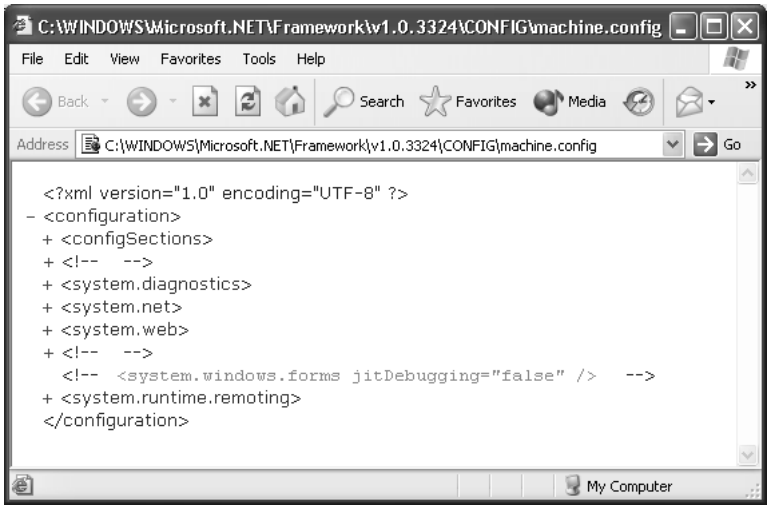

Abbildung 5.12: Die Datei machine.config zur Einstellung der maschinenweiten Konfiguration im Internet Explorer

Die Datei besteht aus mehreren Abschnitten, in denen für unterschiedliche Sub-Systeme des Frameworks Konfigurationseinstellungen gemacht werden. Diese Abschnitte werden im Folgenden etwas näher betrachtet.

Prinzipieller Aufbau der Konfigurationsdatei *machine.config*

Die Datei *machine.config* ist in XML geschrieben, wodurch sie leicht bearbeitet werden kann. Sowohl manuell als auch automatisiert (z.B. per Skript) lassen sich Einträge löschen, hinzufügen oder ergänzen.

Die bei der Installation erstellten Abschnitte für die maschinenweite Konfiguration sind:

- **system.diagnostics**: Mit der *Trace*-Klasse werden Nachrichten speziell über die Ausführung des Codes einer installierten Applikation behandelt. Indem so genannte *Switches* im Code platziert werden, kann Kontrolle darüber ausgeübt werden, wann und in welchem Umfang Meldungen protokolliert werden. In einer komplexen Business-Applikation kann auf diese Weise der Status überwacht werden. Wie sich die im Code eingebrachten Switches verhalten, kann über die Konfiguration eingestellt werden. Nach erfolgter Installation einer .NET-Applikation auf einem Computer kann über die Konfiguration eingestellt werden, ob die im Code stehenden Switches Meldungen in eine Datei schreiben oder nicht, die dann im Anschluss zu Debugging- und Diagnose-Zwecken genutzt werden können. Bei der Installation ist das Tracing in der Konfiguration ausgeschaltet.
- **system.net**: In diesem Abschnitt können Einstellungen bezüglich der Verwendung der Klassen aus dem Namespace *System.Net* gemacht werden. Bei der Installation werden Einstellungen für den Default-Proxy, benutzerdefinierte (HTTP) Request-Module, Authentifizierungs-Module für Web-Requests und die maximale Anzahl an Verbindungen zum Server eingetragen.
- **system.web**: Konfiguration für ASP .NET. Die Einstellungen, die hier gemacht werden können, sind zum Teil im Abschnitt über ASP .NET in diesem Buch beschrieben.
- **system.runtime.remoting**: Hier werden Einstellungen für entfernte (*remote*) Objekte und Kanäle (*channels*) gemacht. Dies betrifft u.a. entfernte Objekte, die von einer Applikation konsumiert bzw. exponiert werden, und Informationen über die Kanäle, die zur Kommunikation mit entfernten Objekten benutzt werden.

Es können noch weitere Abschnitte hinzugefügt werden. Das Schema für die Konfigurationsdatei enthält zusätzlich die folgenden Sektionen:

Schema-Sektionen	Beschreibung
Startup	Festlegung der Version der CLR, die von der Applikation zur Laufzeit benötigt wird.
Cryptography	Einstellungen bezüglich der kryptografischen Dienste.
Runtime	Einstellungen bezüglich des Garbage Collectors und der Assemblies der Applikation.

Tabelle 5.2: Auswahl an Sektionen aus dem Schema der maschinenweiten Konfigurationsdatei

In der Regel wird die maschinenweite Konfigurationsdatei nicht editiert, da hier immerhin Einstellungen für alle (!) .NET-Anwendungen gemacht werden, die auf einem Computer installiert sind. Wenn spezifische Angaben bezüglich einer einzigen Applikation gemacht werden sollen, gibt es die Möglichkeit applikationsspezifische Konfigurationsdateien anzulegen.

5.2.2 Applikationsspezifische Konfiguration

Die Vorteile und der Nutzen von applikationsspezifischen Konfigurationsdateien werden anhand eines Beispiels erläutert:

Der EuroRechner

Aufgrund der aktuellen Situation zum Jahreswechsel 2001/2002 soll eine kleine Anwendung als Demonstrationsobjekt dienen, die eine Währungsumrechnung von DM in Euro (und umgekehrt) vornimmt.

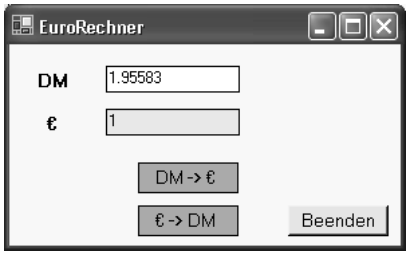

Abbildung 5.13: Die Beispiel-Applikation EuroRechner

Bei der Eingabe von Werten in die erste (obere) Textbox (siehe Abbildung 5.13) wird dynamisch die zweite Textbox mit dem umgerechneten Wert aktualisiert. Es ist nur die Eingabe von Ziffern, dem Backspace und dem Dezimal-Punkt ('.') zugelassen. Zur eigentlichen Berechnung werden Methoden eines zweiten Assemblies aufgerufen.

Damit besteht die Anwendung aus den zwei Assemblies: *EuroRechner.exe* und *EuroRechnerLib.dll*. Hier zunächst der (wesentliche) Code der Bibliothek *EuroRechnerLib.dll*:

Listing 5.5: Ausschnitt aus der Bibliothek EuroRechnerLib.dll der Anwendung EuroRechner

```
 1:  using System;
 2:
 3:  namespace EuroRechnerLib
 4:  {
 5:     public class myMathLib
 6:     {
 7:        public static double calculateEuro(double betrag)
 8:        {
 9:           [...]
10:        }
11:
12:        public static double calculateDM(double betrag)
13:        {
14:           [...]
15:        }
16:     }
17: }
```

Die Bibliothek (dargestellt in Listing 5.5) enthält zwei Methoden *calculateEuro* (Zeile 7) und *calculateDM* (Zeile 12) zur Umrechnung von DM in Euro bzw. umgekehrt. Beide Methoden übernehmen einen double-Wert als Parameter und geben einen entsprechenden Wert als Ergebnis der Berechnung zurück. Die Methoden sind als statische Methoden (*static*) angelegt. Damit ist es für einen Client nicht notwendig, eine Instanz der Klasse *myMathLib* anzulegen.

Nun der Code des Clients *EuroRechner.exe*. Der Client ist eine Windows Forms-Anwendung, die lediglich eine einzige Klasse implementiert:

Listing 5.6: Erster Ausschnitt aus der Klasse CurrencyForm des Clients EuroRechner.exe

```
1:  public class CurrencyForm : System.Windows.Forms.Form
2:  {
3:     private bool mord = false; /* Berechne Euro */
4:     [...]
5:     public bool MOrD
6:     {
7:        get
8:        {
9:           return mord;
```

```
10:       }
11:       set
12:       {
13:          mord = value;
14:       }
15:    }
16:
17:    public CurrencyForm()
18:    {
19:       InitializeComponent();
20:
21:       this.textBox1.KeyPress += new
22:             System.Windows.Forms.KeyPressEventHandler(
23:             this.textBox1_KeyDown);
24:    }
25:    [...]
26: }
```

Die Klasse *CurrencyForm* enthält eine Eigenschaft (*property*) *MOrD* (Zeilen 5-15 und Zeile 3), die den Status der Anwendung speichert. Hier wird abgelegt, ob eine Berechnung von DM in Euro (*mord = false*) oder von Euro in DM (*mord = true*) durchgeführt werden soll.

In den Zeilen 21-23 wird eine Ereignisbehandlungsroutine für das Ereignis *KeyPress* registriert. Diese Routine ist ebenfalls in der Klasse CurrencyForm implementiert:

Listing 5.7: Die Ereignisbehandlungsroutine textBox1_KeyDown der Klasse CurrencyForm der Anwendung EuroRechner.exe

```
 1:  private void textBox1_KeyDown(object sender,
 2:        System.Windows.Forms.KeyPressEventArgs ke)
 3:  {
 4:     string searchMe = "0123456789";
 5:     if( (searchMe.IndexOf(ke.KeyChar) == -1) &&
 6:        (ke.KeyChar != '\x8') && /* Backspace */
 7:        (ke.KeyChar != '\x2e') ) /* . */
 8:     {
 9:        ke.Handled = true;
10:    }
11: }
```

Die Methode *textBoxKexDown* überprüft, ob in der Textbox tatsächlich nur Zahlen (Zeilen 4/5), der Backspace (Zeile 6) oder ein Dezimal-Punkt (Zeile 7) eingegeben worden sind. Wenn keines der spezifizierten Zeichen eingegeben wurde, wird die Ereigniskette abgebrochen und die Anwendung erwartet die Eingabe weiterer Zeichen bzw. Aktionen des Benutzers (z.B. Klick auf den Beenden-Button).

Um den *EuroRechner* von einem Berechnungsmodus in den anderen umzustellen, gibt es zwei entsprechende Methoden als Ereignisbehandlungsroutinen für die Buttons (siehe Abbildung 5.13):

Listing 5.8: Die Ereignisbehandlungsroutine button1_Click zur Umstellung des EuroRechners von Modus Euro->DM in DM->Euro

```
 1:  private void button1_Click(object sender,
 2:        System.EventArgs e)
 3:  {
 4:     label1.Text = "DM";
 5:     label2.Text = "_";
 6:     MOrD = false;
 7:     textBox1.Text = "";
 8:     textBox2.Text = "";
 9:     textBox1.Focus();
10: }
```

Die zweite Methode *button2_Click* ist dieser recht ähnlich und wird hier nicht wiedergegeben.

Wichtiger ist eine weitere Methode zur Behandlung des Ereignisses *TextChanged*:

Listing 5.9: textBox1_TextChanged behandelt das Ereignis TextChanged der textBox1 und ruft dynamisch in Abhängigkeit von Status (MOrD) die Berechnungsmethoden der Bibliothek auf.

```
 1:  private void textBox1_TextChanged(object sender,
 2:        System.EventArgs e)
 3:  {
 4:     try
 5:     {
 6:        if(textBox1.Text != "")
 7:        {
 8:           if(!MOrD)
 9:           {
10:             textBox2.Text = EuroRechnerLib.myMathLib
11:                .calculateEuro(
12:                double.Parse(textBox1.Text)).ToString();
13:          }
14:          else
15:          {
16:             textBox2.Text = EuroRechnerLib.myMathLib
17:                .calculateDM(
18:                double.Parse(textBox1.Text)).ToString();
19:          }
20:       }
21:       else
```

```
22:         {
23:             textBox2.Text = "";
24:         }
25:     }
26:     catch
27:     {
28:         textBox2.Text = "0";
29:     }
30: }
```

Die Methode aus Listing 5.9 behandelt das Ereignis *TextChanged*, das dann auftritt, wenn sich in der Textbox *textBox1* der Wert ändert. Dann wird zunächst überprüft, ob die Textbox überhaupt einen Wert enthält (Zeile 6). Wenn dies der Fall ist, wird je nach Status der Applikation (Überprüfung der Eigenschaft *MOrD* in Zeile 8) die entsprechende Berechnungsroutine in der Bibliothek aufgerufen (Zeilen 10-12 bzw. Zeile 16-18). Der Wert, der von diesem Methoden zurückgeliefert wird, wird gleichzeitig der Textbox *textBox2* zugewiesen (Zeile 10 bzw. Zeile 16).

Debuggen und Ausführen der Applikation

Um die Anwendung in Visual Studio .NET ausführen und debuggen zu können muss die Referenz in der Windows Forms-Anwendung *EuroRechner* auf die Bibliothek *EuroRechnerLib* entsprechend eingestellt werden. Dazu muss im Eigenschaften-Fenster der Referenz die Eigenschaft COPY LOCAL auf *true* gesetzt werden.

Dies hat den einfachen Grund, dass dann die Bibliothek im selben Verzeichnis liegt, wie die Anwendung.

Im nächsten Schritt werden die Client-Anwendung und die Bibliothek in ein gemeinsames Verzeichnis kopiert: (z.B.) *C:\EuroRechner*.[4]

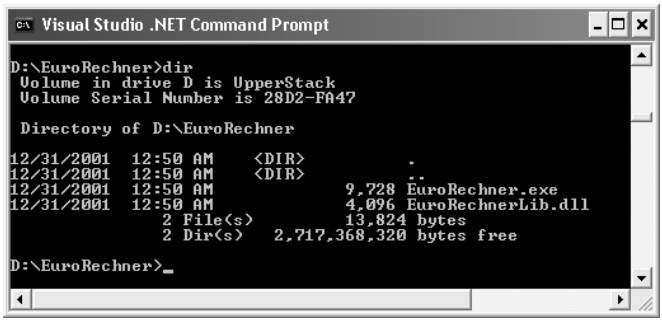

Abbildung 5.14: Beide Assemblies *EuroRechner.exe* und *EuroRechnerLib.dll* werden in ein gemeinsames Verzeichnis kopiert.

4 Dies wird hier lediglich zur Erläuterung der folgenden Konfiguration der Applikation vorgenommen. Selbstverständlich kann in der IDE auch der Pfad angegeben werden, in dem die zu erzeugende Anwendung (eines Projekts) abgelegt werden soll.

Wird das Programm *EuroRechner.exe* von der Kommandozeile aus ausgeführt, kann die Bibliothek *EuroRechnerLib.dll* von der CLR gefunden, geladen und ausgeführt werden. Dabei handelt es sich um eine einfache Referenz eines Assemblies (*EuroRechner.exe*) auf ein anderes externes Assembly (*EuroRechnerLib.dll*).

Bei einer komplexeren Anwendung mit mehreren externen Assemblies ist es jedoch meistens unangebracht und unübersichtlich, alle Assemblies in das Hauptverzeichnis der Applikation zu legen. Zur besseren Strukturierung sollten die Assemblies in entsprechenden Unterverzeichnissen untergebracht werden. Die Bibliothek *EuroRechnerLib.dll* wird deshalb in ein Unterverzeichnis mit dem Namen *EuroRechnerLib* verschoben.

Auch bei einer erneuten Ausführung der Anwendung findet die Runtime das Assembly und kann es ohne Fehlermeldung laden.

Aber auch hier ist es wieder nicht besonders sinnvoll, jedes einzelne Assembly einer komplexeren Anwendung in jeweils ein eigenes Unterverzeichnis zu verschieben. Deshalb wird hier im Beispiel das Verzeichnis *EuroRechnerLib* in *meineAssemblies* umbenannt und die Anwendung wieder ausgeführt:

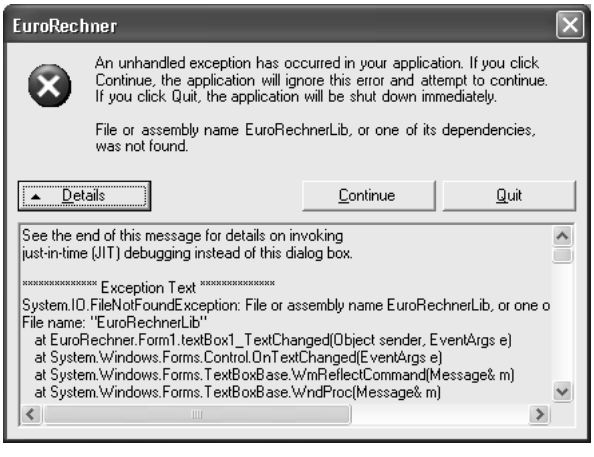

Abbildung 5.15: Wenn die CLR ein Assembly nicht finden und laden kann, wird ein entsprechender Ausnahmefehler erzeugt.

Die CLR kann nun das Assembly *EuroRechnerLib.dll* nicht mehr finden, und es wird ein Ausnahmefehler erzeugt.

Damit die Runtime das Assembly aber zur Laufzeit finden kann, gibt es die Möglichkeit, eine applikationsspezifische Konfigurationsdatei zu schreiben und diese im Verzeichnis der Anwendung abzulegen. Die Datei muss bestimmten Namenskonventionen folgen:

```
NameDerAnwendung.exe.config
```

Damit lautet der Name der Konfigurationsdatei in diesem Beispiel *EuroRechner.exe.config*.

Auch diese Datei wird in XML erstellt:

Listing 5.10: Einfache Konfigurationsdatei für die Anwendung EuroRechner.exe

```
1:  <configuration>
2:     <runtime>
3:        <assemblyBinding xmlns=
4:              "urn:schemas-microsoft-com:asm.v1">
5:           <probing privatePath="meineAssemblies"/>
6:        </assemblyBinding>
7:     </runtime>
8:  </configuration>
```

Diese einfache Konfigurationsdatei enthält lediglich einen wichtigen Hinweis für die CLR, wo das referenzierte Assembly zu finden ist. Das Tag *probing* bzw. dessen Attribut *privatePath* wird dazu verwendet, eine Liste von Verzeichnissen anzugeben, in denen die CLR nach entsprechenden Assemblies suchen soll. Dies wird dann genutzt, wenn die Assemblies nicht im Hauptverzeichnis der Anwendung oder in Unterverzeichnissen mit den passenden Namen gefunden werden konnten.

Die Einstellungen sind eingebettet in das *configuration*- und das *runtime*-Tag und erweitern die Konfiguration aus der maschinenweiten Datei *machine.config*.

Eine Möglichkeit, ohne Konfigurationsdatei zu arbeiten und gleichzeitig die Bibliothek für andere Anwendungen zur Verfügung zu stellen ist, sie im GAC zu installieren. Dazu benötigt die Bibliothek jedoch einen *strong name* (siehe *Abschnitt Ein starker Name*). Es ist eine Schlüsseldatei zu generieren und die Bibliothek damit erneut zu kompilieren. Anschließend kann die Bibliothek mit dem Befehl

```
gacutil.exe /i EuroRechnerLib.dll
```

im Global Assembly Cache installiert werden.

Nun könnte die Konfigurationsdatei gelöscht oder umbenannt werden und die Anwendung würde dennoch erfolgreich ausgeführt werden, da das Assembly von der CLR im GAC gefunden werden kann.

Damit jedoch sichergestellt ist, dass auch genau das Assembly mit einem fest vorgegebenen öffentlichen Schlüssel im GAC referenziert wird, wird die Konfigurationsdatei erweitert. Dazu muss das Token des öffentlichen Schlüssels mit dem Befehl

```
sn.exe -T EuroRechnerLib.dll
```

aus der Schlüssel-Datei entnommen werden.

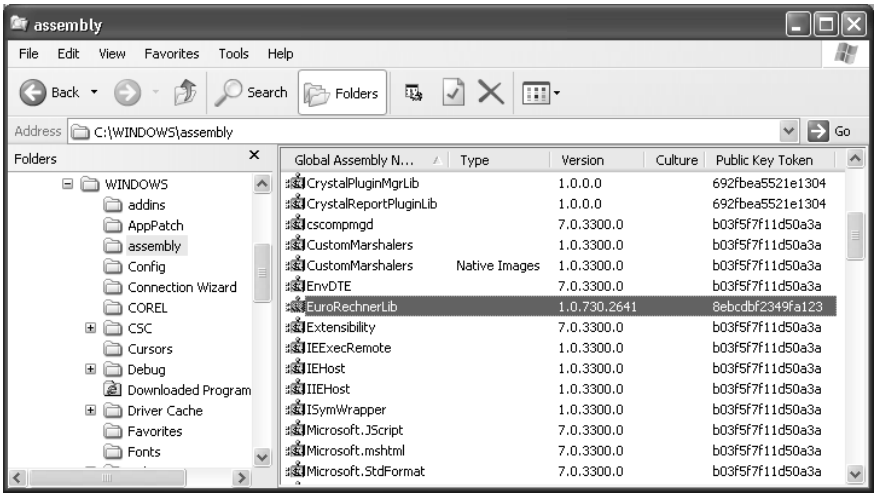

Abbildung 5.16: Das Assembly EurorechnerLib.dll kann nach Generierung eines strong name im GAC installiert werden.

Listing 5.11: Erweiterte Konfigurationsdatei für die Anwendung EuroRechner, um eine feste Referenz auf ein Assembly mit einem bestimmten öffentlichen Schlüssel-Token zu erreichen.

```
 1: <configuration>
 2:    <runtime>
 3:       <assemblyBinding xmlns=
 4:             "urn:schemas-microsoft-com:asm.v1">
 5:          <publisherPolicy apply="no"/>
 6:          <dependentAssembly>
 7:             <assemblyIdentity name="EuroRechnerLib"
 8:                   publicKeyToken="8ebcdbf2349fa123"
 9:                   culture=""/>
10:            <publisherPolicy apply="no"/>
11:         </dependentAssembly>
12:      </assemblyBinding>
13:   </runtime>
14: </configuration>
```

Die Erweiterung der Konfiguration umfasst die Zeilen 5 und 10 sowie die Zeilen 7 bis 9. In den Zeilen 5 und 10 wird die Verwendung einer so genannten *publisher policy* ausgeschaltet. Dabei handelt es sich um eine spezielle Art von Konfigurationsdatei, die auch im GAC installiert werden kann. Dort werden die Konfigurationseinstellungen dann für alle Anwendungen gültig, die auf das dazugehörige Assembly verweisen.

In den Zeilen 7 bis 9 wird eindeutig die Identität des Assemblies festgelegt, auf das hier verwiesen wird. Das Assembly wird nur dann von der CLR geladen, wenn es den entsprechenden Namen und das passende Token besitzt.

Im letzten Schritt geht es nun darum, mehrere Versionen des Assemblies *EuroRechnerLib.dll* im GAC zu installieren und auf eine entsprechende Variante mit einer bestimmten Versionsnummer zu verweisen.

Die Bibliothek *EuroRechnerLib.dll* wird in einer neuen Version so umgeschrieben, dass die Methoden mit Werten des Typs *decimal* arbeiten und auch solche Werte als Ergebnis zurückgeben. Die neue Version wird ebenfalls im GAC installiert:

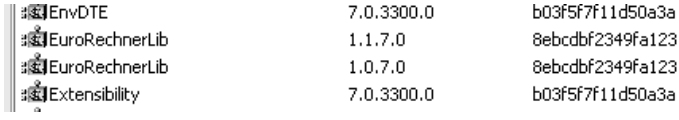

Abbildung 5.17: Das Assembly EuroRechnerLib.dll kann in zwei Versionen im GAC installiert werden.

Das Assembly *EuroRechner.exe* hat einen Verweis zur Versionsnummer *1.0.7.0* der Bibliothek, weil es zusammen mit dieser Version kompiliert worden ist.

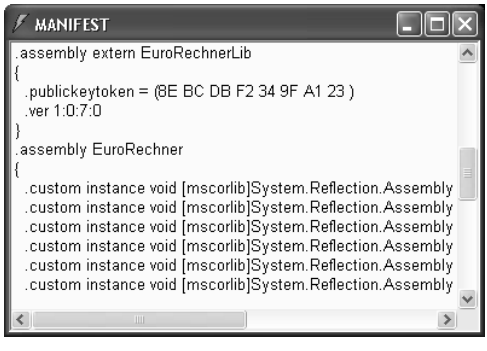

Abbildung 5.18: Im Manifest der Client-Anwendung EuroRechner.exe ist der Verweis zur Version 1.0.7.0 des Assemblies EuroRechnerLib.dll fest verdrahtet.

Die folgende Konfigurationsdatei erzeugt einen Ausnahmefehler in der Anwendung:

Listing 5.12: Konfigurationsdatei für die Anwendung EuroRechner.exe, die den fest verdrahteten Verweis auf eine bestimmte Versionsnummer umleitet.

```
1:  <configuration>
2:     <runtime>
3:        <assemblyBinding xmlns=
4:              "urn:schemas-microsoft-com:asm.v1">
```

```
 5:          <publisherPolicy apply="no"/>
 6:          <dependentAssembly>
 7:             <assemblyIdentity name="EuroRechnerLib"
 8:                    publicKeyToken="8ebcdbf2349fa123"
 9:                    culture=""/>
10:            <publisherPolicy apply="no"/>
11:            <bindingRedirect oldVersion="1.0.7.0"
12:                  newVersion="1.1.7.0"/>>
13:         </dependentAssembly>
14:       </assemblyBinding>
15:    </runtime>
16: </configuration>
```

Der Ausnahmefehler tritt auf, weil durch die Konfigurationsdatei der Verweis zur Versionsnummer *1.0.7.0* auf die neue Version *1.1.7.0* umgeleitet wird. In dieser neuen Version der Bibliothek kann die aufgerufene Methode nicht gefunden werden, da sich die Signatur geändert hat.

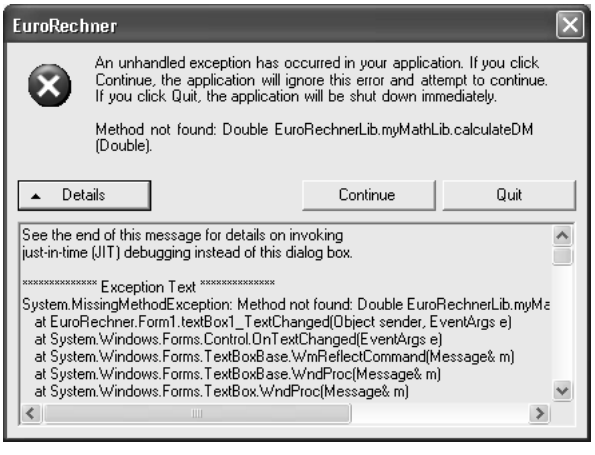

Abbildung 5.19: Die Details des Ausnahmefehlers sagen aus, dass eine Methode in der referenzierten Version des Assemblies nicht gefunden werden konnte.

Anhand dieses Beispiels wird sehr schön deutlich, wie ein Verweis per Konfiguration auf eine andere Version eines Assemblies umgelenkt werden kann, ohne dass die Client-Anwendung neu kompiliert werden muss. Damit kann per Konfiguration genau eingestellt werden, welche Assemblies in welcher Version referenziert werden sollen. Die Angabe der Versionsnummern in der Konfigurationsdatei kann auch mit Wildcards gefüllt werden.

```
11:      <bindingRedirect oldVersion="1.0.*"
12:            newVersion="1.1.7.0"/>>
```

Es kann aber auch ein ganzer Nummernbereich angegeben werden:

```
11:      <bindingRedirect oldVersion="1.0.0.0 - 1.0.8.0"
12:            newVersion="1.1.7.0"/>>
```

5.2.3 Der vollständige Algorithmus zum Auffinden von Assemblies

Die Common Language Runtime (CLR) verfolgt ein festgelegtes Suchmuster beim Auflösen von Verweisen auf externe Assemblies. Dieses Verfahren ist eine Kombination aus von vornherein fest vorgegebenem Verhalten und dessen Beeinflussungen durch eine eventuell vorhandene Konfigurationsdatei, die entsprechende Informationen enthält.

Die Abbildung 5.20 zeigt eine grafische Darstellung des Algorithmus in Form eines Flussdiagramms.

Zu Beginn des Vorgangs werden die Informationen – sofern vorhanden – aus der maschinenweiten Konfigurationsdatei *machine.config* eingelesen. Dies geschieht immer, wenn die Datei im Konfigurationsverzeichnis des Frameworks existiert.

Im zweiten Schritt wird überprüft, ob eine applikationsspezifische Konfigurationsdatei vorhanden ist und im positiven Fall die dort eventuell vorhandenen Informationen bezüglich der aktuellen Referenz eingelesen.

Ob die Datei vorhanden ist oder nicht, der nächste Schritt besteht darin, dass die CLR kontrolliert, ob das zu suchende Assembly bereits früher schon einmal geladen worden ist. Dies ist besonders in dem Fall wichtig, wenn ein externes Assembly über das Internet geladen werden soll (siehe Erläuterungen zum <*codebase*>-Tag). Ist das Assembly noch vorhanden wird es nun geladen und ausgeführt. Wenn es nicht vorhanden ist, wird im GAC gesucht.

Konnte die Runtime das Assembly im GAC finden (entsprechend den eventuell vorhandenen Informationen aus der Konfigurationsdatei), wird es ausgeführt. Wenn nicht, holt sich die CLR weitere Informationen aus der maschinenweiten (was eher selten der Fall ist) oder applikationsspezifischen Konfigurationsdatei.

Dort wird überprüft, ob das Tag <*codebase*> Hinweise auf den Verbleib des zu suchenden Assemblies geben kann. Dieser Eintrag kann dazu verwendet werden, einen Verweis auf ein Assembly zu machen, das per URI-Referenz geladen werden kann. Dies kann beispielsweise eine URL zu einem Server sein, der das Assembly (und evtl. noch weitere) zur Benutzung zur Verfügung stellt.

Ist das Assembly am spezifizierten Ort vorhanden, kann es geladen (eventuell über das Internet heruntergeladen) und ausgeführt werden. Ist es nicht vorhanden, wird ein Ausnahmefehler generiert und der Suchvorgang an dieser Stelle abgebrochen.

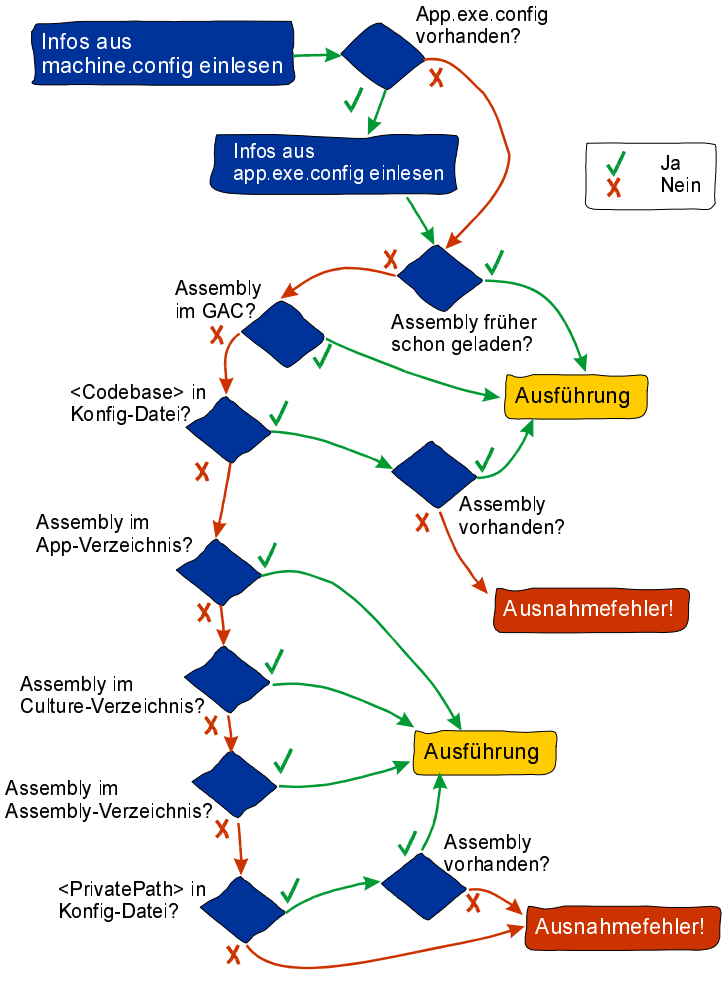

Abbildung 5.20: Der Suchalgorithmus der CLR zum Auffinden von referenzierten Assemblies.

Wenn das *codebase*-Tag nicht vorhanden ist, werden verschiedene Verzeichnisse in einer vorgegebenen Reihenfolge durchsucht. Sobald das Assembly gefunden ist, kann es ausgeführt werden, wenn nicht, wird das nächste Verzeichnis in der Kette untersucht. Zuerst ist da das Applikationsverzeichnis selbst, in welchem auch die Client-Anwendung liegt, gefolgt vom *Culture*-Verzeichnis. Dieses Verzeichnis kann explizit angelegt werden und ist ein Unterverzeichnis des Applikationsverzeichnisses. Wenn eine Bibliothek beispielsweise nur eine bestimmte Sprache (Thema: *Internationalisierung*) unterstützt, kann für jede Sprache ein Verzeichnis angelegt (z.B. *\en-us*) und dort die entsprechenden Versionen der Assemblies abgelegt werden.

Kann das Assembly auch im Culture-Verzeichnis nicht gefunden werden, wird als nächstes ein Unterverzeichnis gesucht, das denselben Namen trägt wie das Assembly selbst (ohne Dateiendung). Wird es nicht gefunden oder ist das Assembly in dem Verzeichnis nicht vorhanden, werden wieder Informationen in der Konfigurationsdatei gesucht.

Ist dort das Tag <*probing*> inklusive Attribut *privatePath* nicht vorhanden, wird nun ein Ausnahmefehler generiert und der Suchvorgang erfolglos abgebrochen. Ist das Tag vorhanden, werden die dort spezifizierten Verzeichnisse nach dem gesuchten Assembly durchforstet. Bleibt auch diese Suche erfolglos, gibt es für die CLR an diesem Punkt nun wirklich keine Möglichkeit mehr, und es wird ein Ausnahmefehler generiert.

5.3 Deployment – was ist das?

Deployment bezeichnet den Vorgang der Verteilung einer Applikation. Dazu gehören Aufgaben wie das Erstellen einer Installationsversion der Applikation inklusive eventuell notwendiger Konfigurationsdateien und das Versenden zum bzw. Zur-Verfügung-Stellen für einen Benutzer.

Das wichtigste Element in diesem Zusammenhang ist das *Assembly.* Was das ist und wie es erstellt wird, wurde in den beiden vorhergehenden Abschnitten detailliert erläutert.

5.3.1 XCopy-Deployment

Eines der Hauptziele des .NET Framework ist die Vereinfachung der Verteilung von Programmen und Bibliotheken. Mit dem so genannten *XCopy-Deployment* in .NET wurde dieses Ziel erreicht. Eine komplette Applikation inklusive aller Unterverzeichnisse kann am Hauptverzeichnis der Anwendung genommen und einfach in ein anderes Verzeichnis, ja sogar auf einen anderen Computer kopiert werden.

Alle Verweise innerhalb dieser Verzeichnisstruktur sind relativ und werden beim Kopieren der Verzeichnisstruktur nicht aufgebrochen.

Sollten jedoch einige Assemblies aus dem GAC referenziert werden, müssen diese beim Verschieben der Anwendung auf einen anderen Rechner dort ebenfalls zur Verfügung stehen.

Eine .NET-Anwendung kann damit recht einfach auf einem zentralen Datei-Server zum Download bereitgestellt werden. In diesem Fall ist es besonders einfach für ein Update oder die Bereinigung von Fehlern zu sorgen – es muss lediglich eine Applikation – nämlich die auf dem Datei-Server – gepflegt werden.

Eine wichtige Voraussetzung, die jedoch für jede .NET-Applikation unbedingt zu beachten ist, ist das Vorhandensein des .NET Framework auf einem Computer, auf dem die Applikation ausgeführt werden soll. Da das .NET Framework bislang noch kein Bestandteil des Betriebssystems geworden ist, ist es erforderlich, das

Framework selbst zusammen mit einer .NET Anwendung zu verteilen und zu installieren. Im Laufe der Zeit ist davon auszugehen, dass mehrere Versionen des Frameworks existieren werden, und da eine .NET-Anwendung eventuell kompatibel zu einer bestimmten Version des Frameworks ist, muss diese mitgeliefert werden.

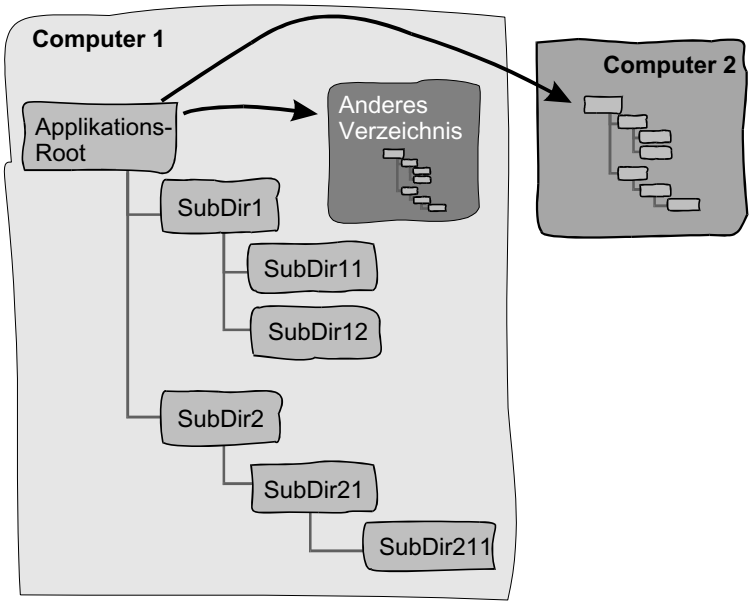

Abbildung 5.21: In .NET kann der Aufwand für das Verteilen einer kompletten Anwendung auf ein einfaches XCopy-Deployment reduziert werden.

Auch die Deinstallation gestaltet sich so einfach wie die Installation: Sofern nicht zusätzliche Assemblies aus dem GAC zu entfernen sind, reicht ein simples Löschen des Anwendungsverzeichnisses (und aller Unterverzeichnisse) aus, um die Anwendung von einem Computer wieder zu deinstallieren. Es werden hierbei keine unliebsamen Einträge in der Registrierungsdatenbank (*Registry*) vergessen, weil dort keine zu löschen sind. Es können auch keine gemeinsam mit anderen Programmen benutzten Bibliotheken versehentlich gelöscht werden (außer vielleicht im GAC!).

5.3.2 Verpacken einer .NET Anwendung

Um eine aufwändigere Anwendung, die eventuell aus mehreren Assemblies besteht, entsprechend leicht zu verteilen und zu installieren, sind einige Bearbeitungsschritte notwendig.

Da die Assemblies in .NET selbstbeschreibende Komponenten sind und die Verbindungen zu anderen Komponenten sauber definieren, ist die Verteilung und Installation von .NET-Applikationen recht einfach. .NET-Anwendungen benötigen beispielsweise keine Eintragungen in der Windows-Registry. Die CLR kann eine Anwendungen starten und die Verweise auf .NET Komponenten über das Dateisystem auflösen. Bei Verwendung von COM- bzw. ActiveX-Komponenten muss dies auch weiterhin über die Einträge in der System-Registrierung geschehen.

Um ein solches .NET-Programm zu verteilen, können beispielsweise Microsoft Installer-Dateien (*msi*-Dateien) und Merge Modules (*msm*-Dateien) zur Nutzung mit dem Windows Installer erzeugt werden. Eine andere Möglichkeit ist die Erstellung von *cab*-Dateien. Näheres hierzu siehe im letzten Abschnitt Projekte für Setup und Verteilung.

Wenn die Verteilung von einem Web Server aus geschehen soll, können Assemblies über die Konfiguration auf dem Client-Computer per <*codeBase*>-Tag integriert werden. Die so konfigurierten Assemblies werden dann dynamisch von einem Server zum Computer, auf dem die Anwendung läuft, heruntergeladen. Diese Art minimiert die Größe der Haupt-Anwendung. Damit kann ein Benutzer sich lediglich die exe-Datei herunterladen und benötigt bei einem Start des Programms eine aktive Internet-Verbindung, damit die referenzierten Assemblies gefunden und geladen werden können. Nachteil hierbei: Dieses Vorgehen ist nicht geeignet für zeitkritische Anwendungen, da das Herunterladen von Assemblies über das Internet unter Umständen ein wenig Zeit in Anspruch nehmen kann. Außerdem ist jedes Assembly, das in den Download-Cache auf die lokale Maschine heruntergeladen worden ist, nur für die Anwendung nutzbar, die für das Herunterladen des Assemblies verantwortlich ist.

Installations-Komponenten

Im Framework bestehen Anwendungen nicht nur aus einer einzigen Programm-Datei, sondern auch aus dazugehörigen Ressourcen wie z.B. Message-Queues, Log-Dateien für Ereignisse oder Performance-Zähler. Diese Ressourcen müssen auf einem Ziel-Computer, auf dem die Anwendung installiert werden soll, ebenfalls vorhanden sein und gegebenenfalls dort erst generiert werden.

Eine Anwendung kann so konfiguriert werden, dass die erforderlichen Ressourcen bei der Installation angelegt werden. Dies kann natürlich nur in einem gewissen Rahmen geschehen – wenn für den Computer kein Drucker vorhanden ist, kann auch keiner bei der Installation generiert werden. Bei der Deinstallation müssen die angelegten Ressourcen auch wieder entfernt werden.

Diese Aufgaben können von *Installationskomponenten* übernommen werden. Eine Anwendung, die auf die Computer zahlreicher Kunden verteilt werden soll, kann zum Beispiel in eine Datenbank auf einem SQL-Server schreiben. Diese Datenbank muss auf dem Ziel-Rechner vorhanden und entsprechend eingerichtet sein. Mit einer Installationskomponente können die notwendigen Arbeitsschritte zur Einrichtung einer entsprechenden Datenbank und der darin enthaltenen Tabellen automatisiert durchgeführt werden.

Nachdem eine Installationskomponente zu einer Applikation hinzugefügt worden ist, muss noch eine Installations-Klasse erzeugt werden. Jede weitere Installationskomponente für die entsprechenden Installationsaufgaben wird dieser Klasse hinzugefügt. Im Beispiel für einen Windows-Dienst im *Kapitel 3* ist die Vorgehensweise zur Erzeugung einer Installationskomponente beschrieben.

Installationskomponenten sind auf einer 1-zu-1 Basis mit den Komponenten verbunden, für die sie zuständig sind. Im obigen Beispiel, in dem eine Anwendung zur Laufzeit eine entsprechende Datenbank benötigt, muss also eine Installationskomponente vorhanden sein, die die Datenbank einrichtet. Wenn nun eine spezielle Komponente zur Protokollierung von Ereignissen (EventLog-Komponente) verwendet werden soll, muss auch hierfür wieder eine Installationskomponente explizit erstellt werden.

Alle Installationskomponenten in einem Projekt befinden sich innerhalb von speziellen Klassen, die mit dem Attribut *[RunInstaller=true]* versehen sind. Für eine so gekennzeichnete Klasse wird das Tool *InstallUtil.exe* aktiv werden, wenn die Anwendung installiert wird. Standardmäßig wird eine Klasse mit dem Namen *ProjectInstaller* erzeugt, wenn eine Installationskomponente generiert wird. Auch diese Klasse trägt das Attribut, es kann jedoch auch eine andere Klasse verwendet oder einfach nur der Name der generierten Klasse geändert werden.

.NET liefert fünf vorgefertigte Installationskomponenten mit. Diese sind über die Toolbox zu erreichen und können per Drag&Drop von dort aus auf eine Anwendung gezogen werden. Über das Eigenschaften-Fenster der jeweiligen Komponente kann der Dialog zur Einrichtung einer Installer-Komponente aufgerufen werden – z. B. für einen PerformanceCounter:

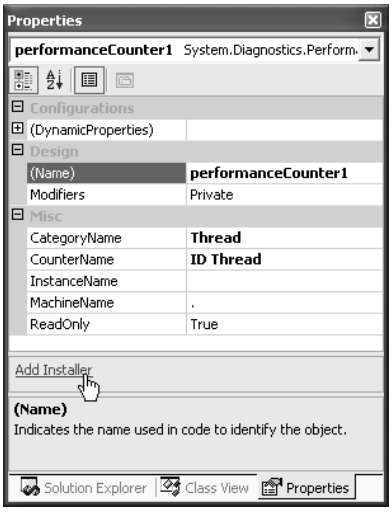

Abbildung 5.22: Über den Dialog Eigenschaften kann der Assistent zur Erstellung eines Installers für die jeweilige Komponente aufgerufen werden.

- **EventLog**: Der Installer für einen EventLog basiert auf der Klasse *System.Diagnostics.EventLogInstaller*. und ermöglicht die Installation und Konfiguration eines benutzerdefinierten Eventlogs.
- **MessageQueue**: Dieser Installer basiert auf der Klasse *System.Messaging.MessageQueueInstaller*. Mit ihm kann eine Queue installiert und eingerichtet werden, die eine Anwendung zur Laufzeit benötigt.
- **PerformanceCounter**: Der PerformanceCounter-Installer basiert auf der Klasse *System.Diagnostics.PerformanceCounterInstaller*. Mit ihm können Performance-Counter installiert und konfiguriert werden.
- **Service** und **ServiceProcess**: Diese beiden Installer basieren auf den Klassen *ServiceInstaller* und *ServiceProcessInstaller* im Namespace *System.ServiceProcess*. Sie werden gemeinsam benutzt, um Dienste zu installieren und zu konfigurieren.

Während der Installation einer Komponente erzeugt die entsprechende Klasse (z.B. ProjectInstaller) auf dem Installations-Rechner eine Datei mit der Endung *.InstallState*. In diese Datei werden Informationen bezüglich des System-Status vor der Installation der Komponenten geschrieben. Außerdem werden dort die vorgenommenen Änderungen am System protokolliert.

Die Installation von Ressourcen läuft transaktionell ab. Dabei iteriert der Installationsprozess über alle Installationskomponenten und überprüft deren Status. Die Zwischenstände werden in den jeweiligen Status-Dateien abgelegt. Entsprechende Methoden fragen diesen Status zu einem bestimmten Zeitpunkt ab, wenn alle Komponenten erfolgreich installieren konnten, wird ein abschließendes »*O.K.*« ausgerufen und alle Änderungen werden fest übernommen. Wenn während dieses Vorgangs in irgendeiner Komponente ein Fehler auftauchen sollte, werden alle vorgenommenen Änderungen der anderen Komponenten wieder zurückgenommen (deshalb werden sie protokolliert).

Der Windows Installer

Der Windows Installer unterstützt alle Leistungsmerkmale von .NET (speziell Assemblies) ab der Version 1.5. Das bedeutet, dass alle hilfreichen Leistungsmerkmale des Windows Installers zur Installation einer .NET-Anwendung genutzt werden können:

- Lokale Installation
- Installation auf einem entfernten Computer
- Reparatur von Applikationen
- Installation-on-demand
- Installation zu Wartungszwecken

Alle Leistungsmerkmale können zusammen mit dem *Microsoft System Management Server* (*SMS*) und der *Microsoft IntelliMirror* Technologie (ab Windows 2000) genutzt werden.

5.3.3 Projekte für Setup und Verteilung

Für das Deployment von .NET-Komponenten gibt es in Visual Studio .NET verschiedene Projektvorlagen. Je nach Typ der Komponente ist die entsprechende Vorlage auszuwählen. In Kapitel 4 wurden die Vorlagen bereits kurz erwähnt. Hier folgt nun eine ausführlichere Erläuterung.

Cab-Projekt

cab-Dateien sind Setup-Dateien für ActiveX-Komponenten. Diese Dateien können in eine Web-Anwendung integriert werden, damit anschließend die Komponente von der Web-Seite auf den Computer eines Besuchers heruntergeladen wird.

Die Vorlage für ein entsprechendes Projekt ist in Visual Studio .NET im Dialog NEW PROJECT im Ordner SETUP AND DEPLOYMENT PROJECTS zu finden. Nach Auswahl wird ein (fast) leeres cab-Projekt angelegt. Über die PROPERTY PAGES des Projekts können entsprechende Dateien für eine *Authenticode-Signierung* angewählt werden.

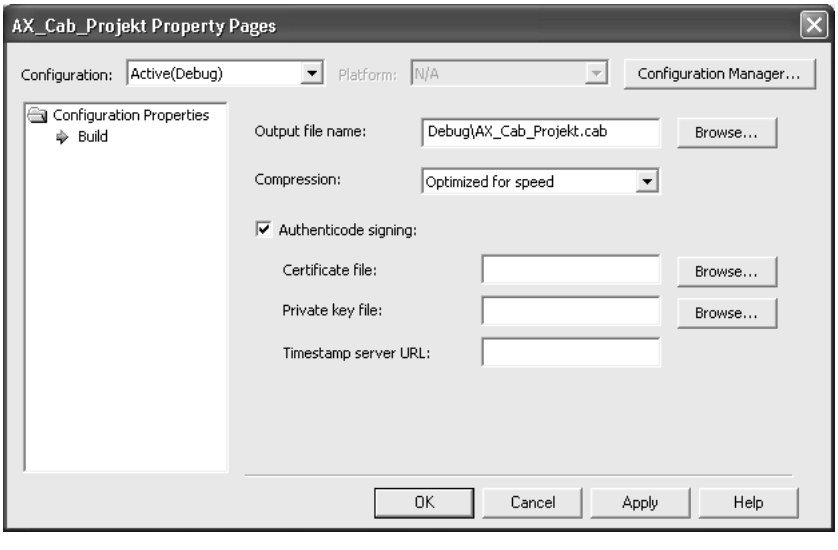

Abbildung 5.23: Im Eigenschaften-Fenster (Property Pages) eines cab-Projekts können Dateien für eine Authenticode-Signierung angegeben werden.

Dem Projekt muss nun über den Menü-Befehl PROJECT\ADD\FILE eine ActiveX-Komponente hinzugefügt werden. Anschließend kann das Projekt gebaut werden. Standardmäßig wird im Debug-Verzeichnis des Projekts eine CAB-Datei mit dem Namen des Projekts erzeugt.

Diese Datei kann mit einem Zip-Tool geöffnet werden. Wurde beispielsweise das ActiveX-Steuerelement *MSCOMCT2.OCX* ausgewählt, enthält die cab-Datei das ActiveX-Control selbst und eine XML-Datei *OSD5A.OSD*.

Listing 5.13: Die Datei OSD5A.OSD, die bei der Erstellung einer cab-Datei generiert wird und selbst Teil der cab-Datei wird.

```
<?XML version="1.0" ENCODING='UTF-8'?>
<!DOCTYPE SOFTPKG SYSTEM
"http://www.microsoft.com/standards/osd/osd.dtd">
<?XML::namespace
href="http://www.microsoft.com/standards/osd/msicd.dtd"
as="MSICD"?>
<SOFTPKG NAME="AX_Cab_Projekt" VERSION="1,0,0,0">
        <TITLE> AX_Cab_Projekt </TITLE>
        <MSICD::NATIVECODE>
            <CODE NAME="MSCOMCT2">
                <IMPLEMENTATION>
                    <CODEBASE FILENAME="MSCOMCT2.OCX">
                    </CODEBASE>
                </IMPLEMENTATION>
            </CODE>
        </MSICD::NATIVECODE>
</SOFTPKG>
```

OSD steht für *Open Software Description*, und diese Datei enthält u.a. Referenz-Informationen für die in der cab-Datei enthaltenen ActiveX-Komponenten. Kommen weitere Komponenten hinzu, wird ein weiterer *<MSICD::NATIVECODE>*-Abschnitt generiert.[5]

Eine solche cab-Datei kann dann zusammen mit der enthaltenen ActiveX-Komponente in eine Web-Applikation integriert werden. Kommt ein Besucher auf die entsprechende Web-Seite, wird die Komponente auf den lokalen Computer heruntergeladen (wenn die Sicherheitseinstellungen des Client-Browsers dies zulassen). Ist sie dort noch nicht vorhanden, wird sie automatisch installiert und registriert.

5 Für weitere Informationen über OSD siehe *SDK-Dokumentation* in der *MSDN-Bibliothek* von Microsoft.

Mergemodul-Projekt

Es können nicht nur wieder verwendbare Komponenten für Programme wie Windows- oder Web-Applikationen erstellt werden. Auch für Setup-Projekte lassen sich Teil-Komponenten erstellen, die in verschiedenen Installations-Programmen eingesetzt werden können.

Eine *Mergemodul*-Datei hat die Endung *.msm*. Sie beinhaltet alle Dateien und dazugehörige Ressourcen, um eine Software-Komponente auf einem Ziel-Rechner installieren zu können.

Als Beispiel wird die Bibliothek *EuroRechnerLib2* verwendet. Der Solution ist ein neues Setup-Projekt MERGE MODULE SETUP hinzuzufügen.

Im Editor-Fenster von Visual Studio .NET kann über das Kontextmenü des Ordners MODULE RETARGETABLE FOLDER der Dialog ADD PROJECT OUTPUT GROUP ausgewählt werden. Dort sind (wie beim Setup-Projekt) das entsprechende Projekt und die zu integrierenden Ressourcen anzugeben. Im Beispiel wird *Primary Output* gewählt, um die DLL- und EXE-Dateien des Projekts zu integrieren. Nun kann das Projekt gebaut werden.

Diese Setup-Komponente kann nun in anderen Setup-Projekten verwendet werden. Im Beispiel wird in der Solution noch zusätzlich ein Setup-Projekt (*EuroRechnerLib2_Installer*) erstellt. Es kann nun so weitergemacht werden, wie im Abschnitt über *Setup-Projekte* gezeigt wird (siehe nächsten Abschnitt). Es muss jedoch im Dialog ADD PROJECT OUTPUT GROUP das Projekt *EuroRechnerLib2_Setup* ausgewählt werden.

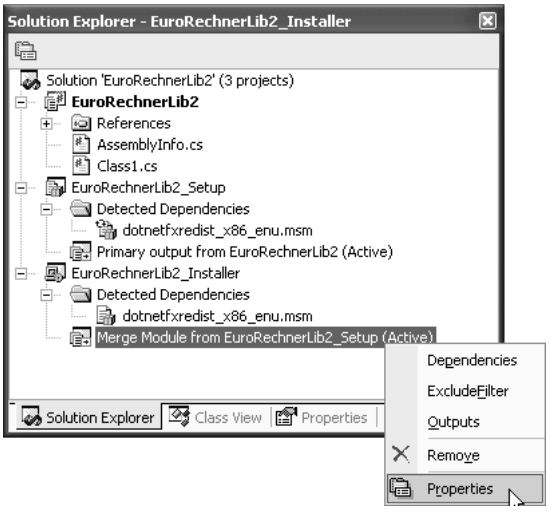

Abbildung 5.24: Öffnen der Eigenschaften der Mergemodul-Datei des Installer-Projekts

Nun muss nur noch die Eigenschaft *Module Retargetable Folder* des Mergemoduls angepasst werden. Dazu ist das Eigenschaften-Fenster der Mergemodul-Datei des Installer-Projekts ausgewählt werden.

In den Eigenschaften befindet sich unter KEYOUTPUT\MERGEMODULPROPERTIES der Eintrag für *Module Retargetable Folder*. Diese Eigenschaft muss auf *Application Folder* gesetzt werden; damit wird einfach nur der Ziel-Ordner für die Installation gesetzt. Bei einem Build wird nun eine msi-Datei erstellt.

Setup-Projekt

Die Projektvorlage *Setup Project* dient zur Erstellung einer *msi*-Datei. Eine solche Datei wird zusammen mit dem Windows Installer verwendet und dient zur Installation einer Anwendung und den dazugehörenden Ressourcen. Die msi-Datei führt alle Aufgaben aus, die notwendig sind, um eine Anwendung auf einem Computer zu installieren. Dies beinhaltet auch die Erzeugung von Verzeichnissen und Unterverzeichnissen oder Generierung von Einträgen in der System-Registrierung. Sollte aus irgendeinem Grund bzw. Fehler die Installation nicht vollständig durchgeführt werden können, werden alle bis dahin von der msi-Datei vorgenommenen Änderungen wieder zurückgenommen (*roll-back*).

Als Beispiel wird eine kleine msi-Datei für die Windows-Anwendung *EuroRechner* erstellt.

Der Solution *EuroRechner* wird ein neues Setup-Projekt hinzugefügt: *Setup_EuroRechner*. Im Editor-Fenster ist über den Eintrag APPLICATION FOLDER der Dialog ADD PROJECT OUTPUT GROUP auszuwählen. Dort müssen das entsprechende Projekt und die zu integrierenden Ressourcen ausgewählt werden.

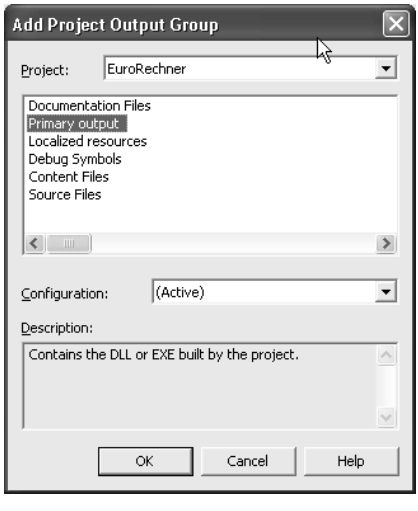

Abbildung 5.25: Im Dialog Add Project Output Group muss das entsprechende Projekt ausgewählt werden.

Anschließend werden automatisch die Abhängigkeiten entdeckt. Dies sind bei der Anwendung *EuroRechner* die Bibliothek *EuroRechnerLib.dll* und das .NET Framework. Wenn das Setup-Projekt zu diesem Zeitpunkt gebaut wird, beträgt die Größe der msi-Datei (setzen Sie sich bitte) etwa 20 MB – und das deshalb, weil die notwendigen Dateien des .NET Framework in der Datei enthalten sind.

Die Installation wird durch Ausführung der msi-Datei gestartet. Der Benutzer wird dann durch mehrere Dialoge geführt, deren Reihenfolge und Inhalte auch über Visual Studio .NET manipuliert werden können. Standardmäßig werden folgende Dialoge angelegt:

- **Welcome**: Ein Begrüßungsdialog, dessen Text über die Eigenschaften abgeändert werden kann.

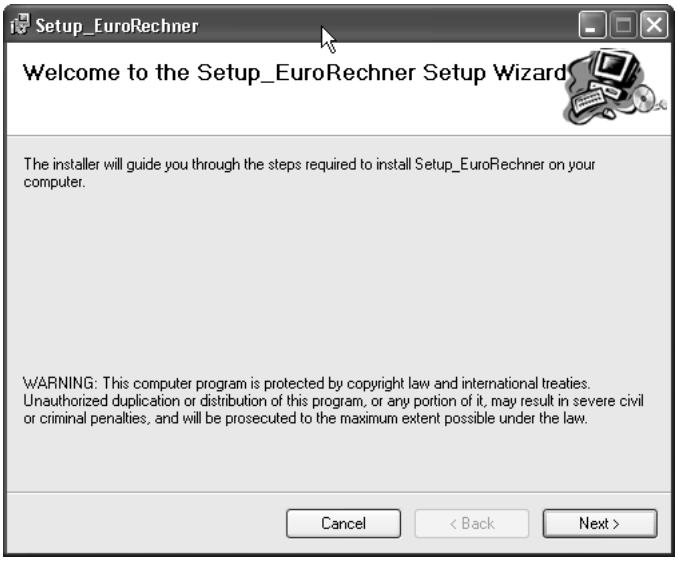

Abbildung 5.26: Erstes Dialogfenster bei der Installation mittels msi-Datei – Der hier markierte Begrüßungstext kann in Visual Studio .NET über die Eigenschaften des entsprechenden User Interface Dialog editiert werden.

- **Installation Folder**: Hier muss der Benutzer einen Installations-Ordner angeben.
- **Confirm Installation**: Bevor die Installation gestartet wird, wird einem Benutzer hier noch einmal die Möglichkeit zum Abbruch gegeben.
- **Progress**: Zeigt den Fortschritt der Installation an.
- **Finished**: Gibt eine Zusammenfassung wieder.

Websetup-Projekt

Die Vorlage für ein *Websetup-Projekt* wird ähnlich benutzt wie die Vorlage *Setup-Projekt*. Wichtig ist hierbei jedoch, dass im Editor die Eigenschaften *VirtualDirectory* und *DefaultDocument* gesetzt werden.

Mit *VirtualDirectory* wird ein Verzeichnis für die Installation auf einem Web Server spezifiziert. Der Wert *MyWebApp* führt dazu, dass die Web-Anwendung in das Verzeichnis *<Web Server>\MyWebApp* installiert wird. Dies kann z. B. *C:\inetpub\wwwroot\MyWebApp* sein.

DefaultDocument ist die Datei, die bei einem Start der Anwendung als erstes im Browser erscheint.

> **QUELLTEXT ZUM WEBSETUP-PROJEKT**
> Als Anwendungs-Beispiel dient hier das Projekt *VBWebService* aus Kapitel 3, für das ein Websetup-Projekt erstellt wird.
>
> Das Projekt *VBWebService* ist auf der Buch-CD im Verzeichnis *\Programme\Kap3\VBWebService* zu finden. Über die Datei *VBWebService.sln* können die Solution und alle darin enthaltenen Projekte in die Visual Studio .NET-Entwicklungsumgebung geladen werden. Das Setup-Projekt ist ebenfalls Teil dieser Solution und befindet sich auf der Buch-CD im Verzeichnis *\Programme\Kap3\VBWebService\VBWebService_Setup*.

Im SOLUTION EXPLORER wird der Solution *VBWebService* ein neues Projekt hinzugefügt. Da ein Websetup-Projekt erstellt werden soll, ist im Solution Explorer die Vorlage WEB SETUP PROJECT im Ordner SETUP AND DEPLOYMENT PROJECTS auszuwählen. Als Name für das Websetup-Projekt ist hier im Beispiel *VBWebService_Setup* gewählt.

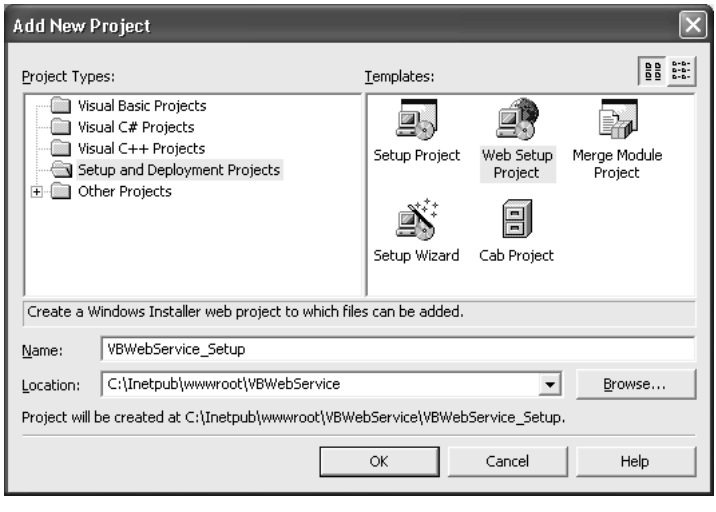

Abbildung 5.27: Erstellen eines neuen Websetup-Projekts

In dem sich daraufhin öffnenden Editor-Fenster können die gewünschten Dateien hinzugefügt werden.

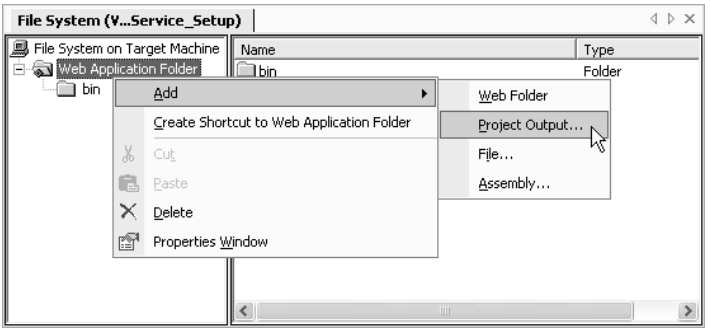

Abbildung 5.28: Hinzufügen derjenigen Dateien zu einem Websetup-Projekt, die mit ausgeliefert werden sollen.

Die Dateien, die hier in die Setup-Datei integriert werden können, sind z.B. alle Dateien des Projekts (wie .aspx- oder Konfigurations-Dateien) und/oder die in die Projekt-Unterverzeichnisse kompilierten Dateien (.exe und .dll). Es ist auch möglich, weitere externe Assemblies oder einzelne Dateien und Ressourcen hinzuzufügen.

Wenn dieses Projekt nun gebaut wird, entsteht eine Setup-Datei, die eine Größe von ca. 20MB hat (siehe auch Kapitel 3). Das liegt daran, dass standardmäßig die Dateien des .NET Framework eingebunden sind. Eine Auswahl der darin enthaltenen Dateien zeigt die *Tabelle 3.22 in Kapitel 3*.

Abbildung 5.29: In einem Setup-Projekt wird standardmäßig die Datei dotnetfxredist_x86_enu.msi eingefügt, die die Dateien zur Installation des .NET Framework enthält.

Im Solution Explorer ist im Setup-Projekt ein Unterverzeichnis zu finden, das die automatisch gefundenen Abhängigkeiten anzeigt. Dies ist generell die Datei *dotnetfxredist_x86_enu.msi*, die alle (grundlegenden) Dateien zur Installation des .NET Framework enthält.

Dieser Link kann hier getrost gelöscht[6] werden, wenn davon auszugehen ist, dass auf dem Ziel-Rechner das Framework bereits installiert worden ist. Damit reduziert sich die Größe der Installationsdatei drastisch um ca. 19MB.

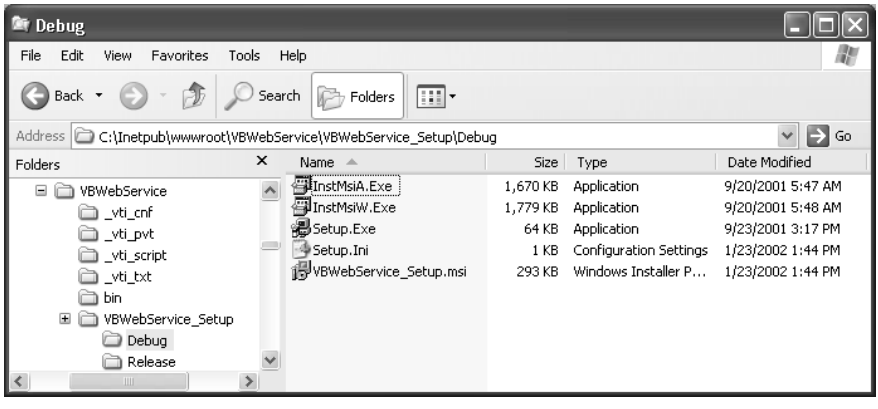

Abbildung 5.30: Die Größe einer Setup-Datei kann wesentlich (um ca. 19MB) verkleinert werden, wenn die Dateien für die Installation des .NET Framework nicht mit einbezogen werden.

Damit steht eine Setup-Datei (.msi-Datei) zur Verfügung, mit der nun der Web Service *VBWebService* auf einem Ziel-Server installiert werden kann – einfach per Doppelklick auf die Datei.

6 Der Link muss nicht gelöscht werden, sondern es ist ausreichend, diese Datei über das entsprechende Kontextmenü lediglich vom Projekt auszuschließen (*exclude*).