Kapitel 3

Erster Entwicklungszyklus mit der WCF

In diesem Kapitel:

Die Architektur der WCF	86
Ein erster Entwicklungszyklus	91
Vorbereiten der Visual Studio-Lösung	94
Erstellen der Verträge	96
Implementierung des Servers	98
Erstellen des Hosts	99
Erstellen des Clients	102
Ausführen des Codes	104

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

Bevor wir uns mit den Details der Windows Communication Foundation vertraut machen, wollen wir uns an der effektiven Architektur und einem ersten Entwicklungszyklus hinsichtlich der Erstellung einer Client/Server-Anwendung mit der WCF orientieren. Dabei spielen die verwendeten WCF-Typen eine untergeordnete Rolle. Es geht vielmehr darum, die Zusammenhänge in Bezug auf Modularisierung und Projektorganisation zu finden. Das Ganze wird an einem Beispiel aufgebaut, das sehr einfach verständlich ist und keine Besonderheiten beinhaltet.

Die Architektur der WCF

Die Architektur der WCF ist darauf ausgerichtet, dass die Implementierung der Dienste von der Kommunikationstechnik getrennt wird. Damit wird das Ziel verfolgt, einen bestimmten Dienst nach seiner Implementierung mittels unterschiedlicher Protokolle und Konfigurationen für die Datencodierung anzusprechen. Selbstverständlich müssen zu diesem Zweck die zu benutzenden Protokolle und Datencodierungen auch wirklich konfiguriert werden. Hier stellt die WCF sowohl deklarative als auch die imperative Konfiguration zur Verfügung. In der Praxis werden Sie sehr oft auf die deklarative Konfiguration zurückgreifen. Das hat den Vorteil, dass die Details der Kommunikation für eine Lösung bei der Installation festgelegt werden, und nicht bereits bei der Implementierung bekannt sein müssen.

Ein weiterer Orientierungspunkt der WCF-Architektur sieht vor, dass sich die Kommunikation an die vom W3C definierten Webdienstregeln hält. Damit bleibt sich Microsoft zum einen selbst treu, indem das Unternehmen die eigenen Entwicklungen wie RPC und SOAP in einem strategischen Produkt einsetzt, zum anderen stellt der Hersteller sicher, dass die plattformübergreifende Verbindung von verteilten Anwendungen möglich wird. Sie müssen allerdings wissen, dass nicht alle Kombinationen der möglichen Dienstkonfigurationen so kompatibel sind, dass Sie plattformübergreifend eingesetzt werden können – doch dazu später mehr.

Ebenen der WCF-Architektur

Die Implementierung der WCF orientiert sich an einem Modell mit vier Softwareebenen (siehe auch Abbildung 3.1):

- Ebene der Verträge
- Ebene der Laufzeitumgebung für Dienste
- Ebene des Nachrichtenflusses
- Ebene der Aktivierung und des Hostings

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

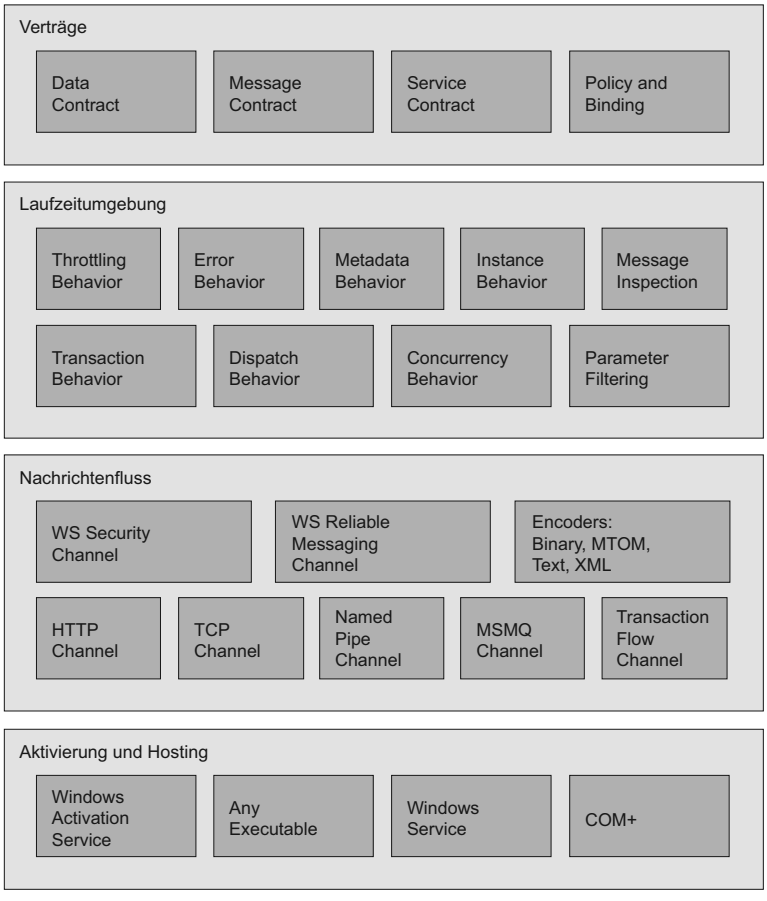

Abbildung 3.1 Architektur der WCF (Quelle: MSDN)

Verträge

Die Vertragsebene definiert die verwendeten Daten, Dienste und Nachrichten. Diese Ebene kennt eine Wechselbeziehung zwischen den WSDL-Dokumenten und den entsprechenden konkreten Beschreibungen in einer .NET-Programmiersprache.

Die Verträge eines Diensts werden öffentlich definiert. Sowohl der Client als auch der Server einer verteilten Anwendung müssen auf die Vertragsdaten zugreifen können. Die Technik erlaubt es allerdings, dass beide Seiten keinen identischen Zugriff auf die Inhalte der definierten Elemente erhalten. Speziell bei Datenverträgen kommt es immer wieder vor, dass der Dienst auf Elemente Schreibzugriff erhält, während der Client Daten nur lesen kann.

HINWEIS Beachten Sie auch das Kapitel 2 zum Thema Webdienste. Dort ist der Begriff WSDL-Dokument im Detail erklärt. Für die Umwandlung der Verträge von der Ausdrucksform WSDL in die Ausdrucksform Code stellt Visual Studio das Werkzeug *SvcUtil.exe* zur Verfügung.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

Laufzeitumgebung

In der Ebene der Laufzeitumgebung stellt die WCF Mechanismen zur Verfügung, die es erlauben, das Verhalten der Dienste zu steuern. Dazu gehören zum Beispiel folgende Verhalten:

- Die Herstellung der Dienstinstanzen (siehe auch Abschnitt »Dienstinstanziierung auf der Serverseite« in Kapitel 1)
- Das Verhalten in Bezug auf Threads des Diensts
- Das Verhalten in Bezug auf Transaktionssteuerung
- Das Verhalten in Bezug auf Fehler

Die Einstellungen für die Laufzeitumgebung werden auf der Clientseite und der Serverseite separat definiert.

HINWEIS Da der Umfang der möglichen Einstellungen für die Laufzeitumgebung enorm groß ist, macht es Sinn, dass Sie einzelne Standardeinstellungen für das Verhalten kennen. Nur wenige davon müssen Sie auswendig beherrschen, aber viele sollten Sie annähernd kennen. In den weiteren Erklärungen für die Einstellungen werde ich Sie auch auf die aktuellen Standardwerte aufmerksam machen.

Nachrichtenfluss

Die Ebene des Nachrichtenflusses definiert die Verwendung der Kommunikationsprotokolle und der Datencodierungen. Besonders interessant an der WCF ist die Tatsache, dass in einer bestimmten Installation ein Dienst gleichzeitig mit mehreren Kommunikationsprotokollen und Datenbindungen konfiguriert werden kann.

Es versteht sich von selbst, dass dabei die Clients mit denselben Kommunikationsprotokollen und Datencodierungen arbeiten müssen wie die Dienste. Obschon somit die Definitionen quasi öffentlich sind, werden sie nicht gemeinsam festgelegt, sondern individuell pro Kommunikationsseite konfiguriert. Es ist unter bestimmten Voraussetzungen möglich, dass die Kommunikationseinstellungen vom Server zum Client übernommen werden.

HINWEIS Auch die möglichen Einstellungen für Protokolle und Datencodierungen sind nicht klein geraten, und es ist wichtig, die einzelnen Standardeinstellungsmöglichkeiten zu kennen. In den jeweiligen Erklärungen hierzu werde ich Sie auf die aktuellen Standardwerte aufmerksam machen.

Aktivierung und Hosting

Die Ebene der Aktivierung und des Hostings kümmert sich um die Herausforderung, einzelne Dienste auf einer Maschine zur Verfügung zu stellen. Konkret geht es darum, die serverseitige Bildung der Prozesse und darin die Dienste zu definieren. Dazu haben wir in der heutigen Betriebssystemumgebung mehrere Möglichkeiten:

- **Windows Activation Service (WAS)** Die Windows Activation Services sind eine neue Technik, die ab Windows Vista respektive Windows Server 2008 verfügbar ist, und dazu dient, Komponenten zu hosten, ohne dafür selbst einen Host schreiben zu müssen. Die Windows Activation Services arbeiten eng mit den IIS und deren Konfiguration zusammen, ohne dabei aber das HTTP-Protokoll nutzen zu müssen. Ebenfalls ist es nicht notwendig, dass die IIS-Pipeline für die Verarbeitung der Anforderungen benutzt wird.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

- **Internet Information Services** Die IIS erlauben die einfache Integration von WCF-Diensten, ohne dass dafür ein separater Hostprozess erstellt wird. Die Nutzung der IIS für die WCF schränkt allerdings ein, dass nur mit dem HTTP- respektive dem HTTPS-Protokoll gearbeitet wird. Andere Protokolle können mit dem oben erwähnten Windows Activation Service benutzt werden.
- **Selbst hergestellte Programme (Self-Hosting)** Das Self-Hosting bedingt, dass die Dienste mit einem selbst geschriebenen Programm gehostet werden. Dabei kann das eigene Programm ein beliebiger Anwendungstyp wie eine Konsolenanwendung, eine WPF-Anwendung oder ein Windows Service sein.
- **COM+ (Wird in diesem Buch nicht behandelt)** Die althergebrachten Komponentendienste COM+ unterstützen auch die WCF. Da diese Technik ein bisschen in die Jahre gekommen ist und ich der Meinung bin, dass künftige Anwendungen auf einer der drei oben erwähnten alternativen Techniken basieren sollten, beschreibe ich diese Art des Hostings in diesem Buch nicht näher.

Der interne Aufbau der WCF

Zugegeben, die Darstellung der vier Ebenen in der vorangehenden Betrachtung der Architektur ist eher theoretischer Art. Allerdings ergibt diese Auslegung doch einen praktischen Überblick über die Teile, die die WCF als WCF-Infrastruktur für eine verteilte Anwendung zur Verfügung stellt respektive nutzt.

Unsere nächste Betrachtung geht nun mehr in die Richtung Praxis, weil wir versuchen werden, die involvierten Bestandteile in zwei Schritte zu unterteilen. Mit Abbildung 3.2 lernen wir zuerst die Hauptakteure in einer verteilten Anwendung kennen.

Wir erkennen auf der linken Seite der Abbildung einen Clientprozess. Dieser enthält ein Proxyobjekt, das die Schnittstelle eines Diensts enthält. Das Proxyobjekt wird von der clientseitigen Programmierung für die Kommunikation mit dem Dienst verwendet. Das Proxyobjekt selbst kommuniziert mit dem Dienst, indem es die WCF-Infrastruktur nutzt.

Der serverseitige Dienst (rechte Seite der Abbildung 3.2) wird vom Client angesprochen, indem die clientseitige Infrastruktur eine Nachricht an den Server sendet. Diese Nachricht löst eine Kommunikation aus und wird deshalb auch Anforderung genannt (request). Damit Daten an den Dienst gesendet werden können, müssen diese zuerst vom Speicher des Clients in die eigentliche Nachricht serialisiert und nach Eintreffen auf dem Server wieder von der Nachricht in den Speicher des Servers deserialisiert werden. Nach dem Erhalt einer Nachricht ruft die serverseitige WCF-Infrastruktur die gewünschten Methode des Diensts auf und sendet die erhaltene Rückgabe der Methode nach dem gleichen Prinzip in Form von Daten wieder zurück an den Client.

HINWEIS Beachten Sie zum Thema auch den Abschnitt »Ablauf eines Datenaustauschs« in Kapitel 1.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

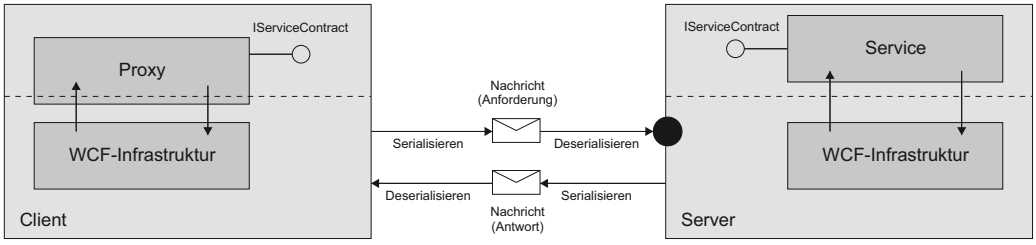

Abbildung 3.2 Prinzip der involvierten Hauptkomponenten einer verteilten Anwendung

In Abbildung 3.2 ist die WCF-Infrastruktur als Blackbox gezeichnet. Aufgrund dessen ist eine Zuordnung der Aufgaben innerhalb der WCF zu den bereits bekannten vier Ebenen kaum möglich. Mit Abbildung 3.3 brechen wir diese Blackbox auf und nehmen einen Drilldown in die Innereien derselben vor.

Die Lesart von Abbildung 3.3 beginnt wiederum im Client im Anwendungscode (links oben). Wir nehmen wieder an, dass der Anwendungscode eine Methode auf der Serverseite in einem Dienst ansprechen will. Der Aufruf geht zunächst wiederum zum Proxyobjekt. Dieses taucht in die WCF-Infrastruktur ein, wo der Weg über verschiedene Ebenen von Objekten bis hinunter auf die Transportebene führt. Unterwegs werden die Daten zu einer SOAP-Nachricht zusammengestellt. Die Transportebene übernimmt die Übermittlung der Nachricht an die Gegenstelle, wo die Nachricht nach dem Empfang durch einen gleichartigen Stapel bis zum Endpunkt-Dispatcher weitergeleitet wird und dabei vom SOAP-Format wieder in ein CLR-Objekt umgewandelt wird. Der Endpoint-Dispatcher ist nun dafür zuständig, mit dem entsprechenden Thread die Methode der Dienstimplementierung anzusprechen und die Antwort der angesprochenen Methode entgegen zu nehmen. Liegt die Antwort vor, wird diese in eine SOAP-Nachricht umgewandelt und zurück an den Client gesendet. Dieser deserialisiert die SOAP-Nachricht und liefert das resultierende CLR-Objekt an die aufrufende Methode.

Standardmäßig läuft die hier aufgezeigte Kommunikation vom Client zum Server komplett synchron ab. Dazu wird auf der Clientseite der Thread nach dem Übermitteln der Anfrage an den Server bis zum Erhalt der Antwort blockiert. Die WCF sorgt zusätzlich auf der Serverseite standardmäßig für einen threadsicheren Aufruf des Diensts, indem mehrfache Aufrufe zum selben Dienstobjekt von der WCF automatisch synchronisiert werden.

HINWEIS Mehr zu den Themen synchrone und asynchrone Aufrufe und Threadingverhalten lesen Sie in den Abschnitten »Das Threadingverhalten steuern« im sechsten Kapitel und »Einen Dienst asynchron benutzen« im neunten Kapitel.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

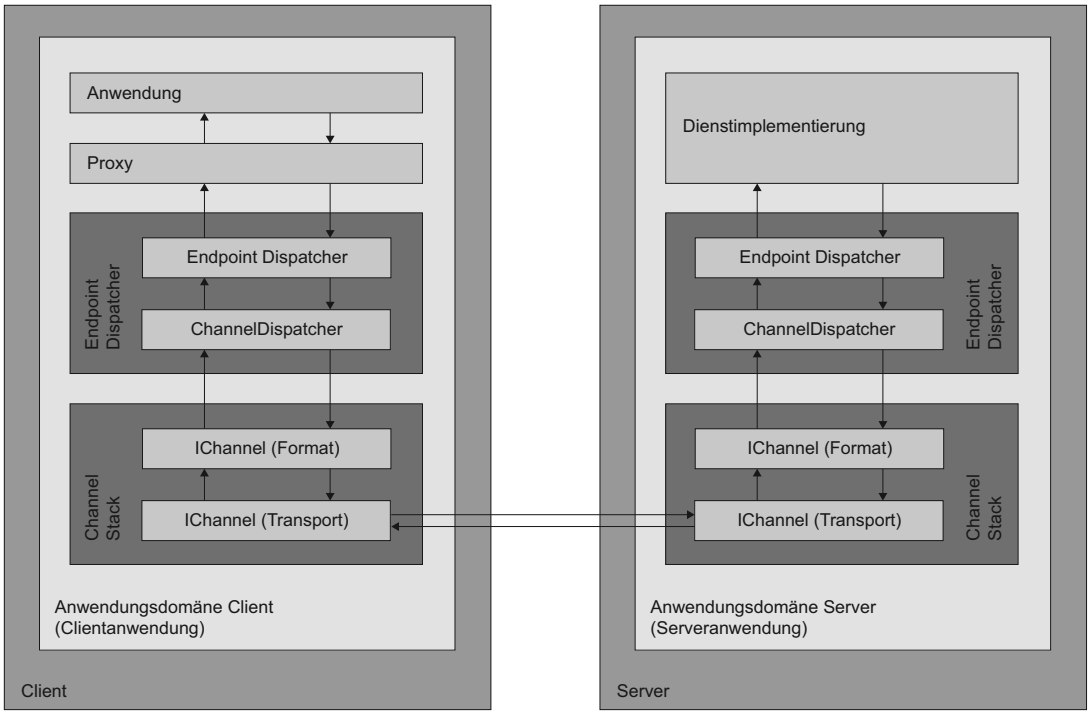

Abbildung 3.3 Prinzipieller innerer Aufbau der WCF-Infrastruktur

Ein erster Entwicklungszyklus

Wie wir bereits im ersten Kapitel in diversen Abschnitten gelesen haben, treten bei einer verteilten Anwendung immer wieder Standardaufgaben wie das Senden und Empfangen von Daten oder das Instanziieren von Dienstobjekten auf. Für die Erledigung dieser Standardaufgaben ist die WCF zuständig. Da die WCF so aufgebaut ist, dass sie unter unterschiedlichsten Bedingungen betrieben werden kann, ist eine anwendungsspezifische Konfiguration unumgänglich. Diese Konfiguration muss von den jeweiligen Prozessräumen auf der Client- und der Serverseite vorgenommen werden.

Damit das Verhalten der WCF für diese Dienste und Daten konkret konfiguriert werden kann, müssen wir unsere Dienste und Daten beschreiben. Hier kommt dazu, dass die Beschreibungen auf beiden Seiten der verteilten Anwendung bekannt sein müssen. Der Client muss wissen, welche Dienste er ansprechen kann, und welche Daten er einer Methode des Diensts senden, respektive von diesem empfangen kann. Der Server wiederum muss diese Dienste implementieren und den Umgang mit den entsprechenden Daten definieren.

Aus den oben beschriebenen Aufgaben lassen sich die Bestandteile für eine verteilte Anwendung wie folgt definieren:

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

- Eine Assembly für die Dienst- und Datenverträge
- Eine Assembly für die serverseitige Dienstimplementierung
- Eine Assembly für Aufbau des Clients
- Eventuell eine Assembly für das Hosting des Diensts (Serverprozessraum)

Abbildung 3.4 veranschaulicht, welche Assembly auf welcher Seite verwendet wird. Dabei ist insbesondere zu erkennen, dass der Client die Assembly für die Dienstimplementierung nicht kennen muss, und auch der Server muss für die Clientimplementierung die Assembly nicht kennen.

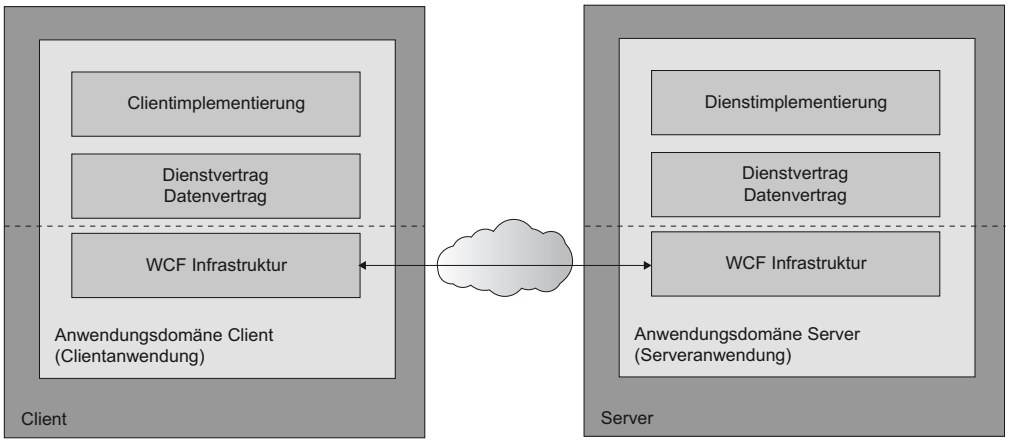

Abbildung 3.4 Übersicht der verwendeten Komponenten bei einer Client/Server-Anwendung mit der WCF

HINWEIS Selbstverständlich ist es technisch möglich, alle notwendigen Typen für eine Client/Server-Anwendung in nur einer Assembly unterzubringen. Ich wähle aber hier den Ansatz der modularisierten Darstellung, weil dieser Ansatz auch die betrieblichen und sicherheitstechnischen Aspekte abdeckt.

Trotzdem sei erwähnt, dass, wenn Sie einfach mal rasch in einem Projekt eines kleinen Prototyps ein technisches Detail der WCF studieren wollen, diese Aufteilung natürlich einen Overkill darstellt. In diesem Spezialfall lege ich oft die Verträge, die Dienstimplementierung und die Erstellung des Proxys in nur eine Assembly.

Die zu erstellenden Assemblys besitzen untereinander Abhängigkeiten. Zudem verwenden die Assemblys Typen aus der WCF-Infrastruktur von .NET. Die daraus entstehenden Gesamtabhängigkeiten zeigt Abbildung 3.5. Beachten Sie, dass die Assemblys für Client und Server aufgrund der Verwendung der Assembly für Verträge auch von den entsprechenden .NET-Assemblys abhängig werden. Diese indirekten Abhängigkeiten habe ich aus Gründen der Übersichtlichkeit in Abbildung 3.5. nicht eingezeichnet.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

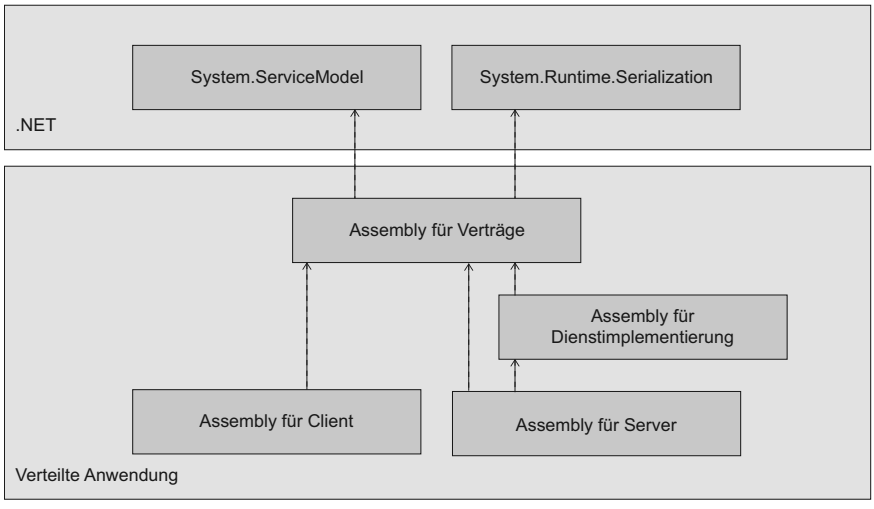

Abbildung 3.5 Prinzipielle Abhängigkeiten der Assemblys einer verteilten Anwendung

HINWEIS Selbstverständlich ist ein Projekt in der Realität vielschichtiger als dieses einfache Beispiel aufgebaut. Aber auch in der Praxis ist es vorteilhaft, die Verträge von den Dienstimplementierungen zu trennen. Die Komplexität der Zusammenstellung der Assemblys für Client und Server werden weniger durch die WCF als durch die verwendete Technik geprägt. So hat zum Beispiel in der WPF-Technologie ein Client, der das MVVM-Entwurfsmuster nutzt, oft einen vielschichtigen Aufbau. Für die effektiven WCF-Clients ist es in dieser Situation von Vorteil, in einer eigenständigen Assembly losgelöst vom GUI-Code verfügbar zu sein.

Im vierten Kapitel werden wir zudem die verschiedenen Techniken für das Hosting eines Diensts kennenlernen. Dort werden Sie feststellen, dass ein Dienst auch mithilfe der Microsoft Internetinformationsdienste (IIS) verfügbar gemacht werden kann. In diesem Fall fällt in dem in Abbildung 3.5 vorgestellten Modell die Assembly für den Server weg, da diese Aufgabe vollständig von den IIS wahrgenommen wird.

Wie bereits erwähnt, wollen wir in diesem ersten Entwicklungszyklus nicht ein möglichst kompliziertes Beispiel erstellen, sondern uns an der Einfachheit orientieren und die Modularisierung und das Vorgehen verstehen. Trotzdem soll das Beispiel den grundlegenden Anforderungen einer Dienstimplementierung entsprechen. Wir orientieren uns zu diesem Zweck an der UML-Darstellung in Abbildung 3.6.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

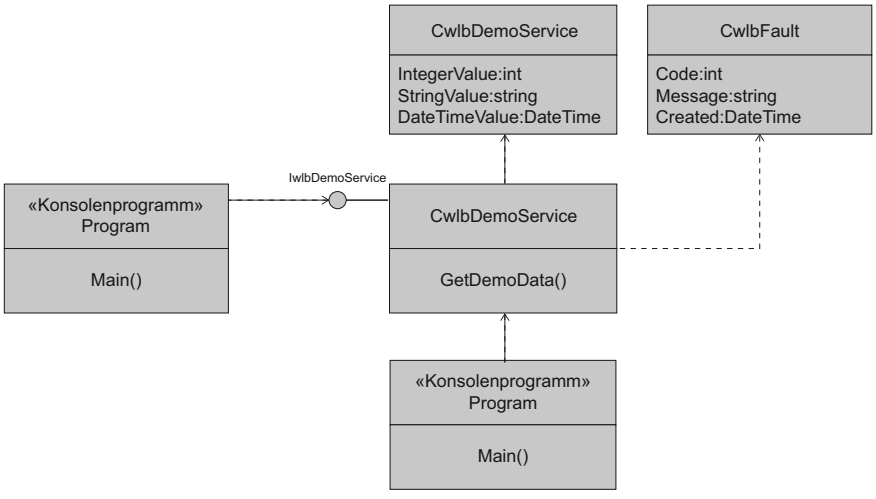

Abbildung 3.6 Verwendetes Klassenmodell für den ersten Entwicklungszyklus

Vorbereiten der Visual Studio-Lösung

Für den ersten Entwicklungszyklus sehen wir vor, die im vorangehenden Abschnitt vorgestellten Assemblys in einer Lösung zu realisieren. Es ist zu erwähnen, dass das in einem realen Projekt eher nicht der Fall sein wird, weil oft mehrere Dienste implementiert werden und sowohl die Server als auch die Clients in der Praxis komplizierter aufgebaut sind. Zudem hat ein praxisorientiertes Projekt auch Elemente, die allgemeingültig zum Einsatz gelangen. Diese sollten besser zwecks Wiederverwendung ebenfalls in separaten Lösungen untergebracht werden.

Nach dem Start des Assistenten für die Erstellung eines neuen Projekts in Visual Studio, stehen uns verschiedene Möglichkeiten zur Verfügung. Da fällt sicher zuerst die Gruppe der Vorlagen für die Erstellung von spezifischen WCF-Projekten auf (siehe Abbildung 3.7). In dieser Gruppe interessiert uns zunächst die Vorlage *WCF Service Library*. Diese Vorlage hat aber zwei Nachteile:

- Sie erstellt die Verträge und die Implementierung in der gleichen Assembly
- Wie alle Vorlagen erstellt sie eine Lösung und darin enthalten ein gleichnamiges Projekt

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

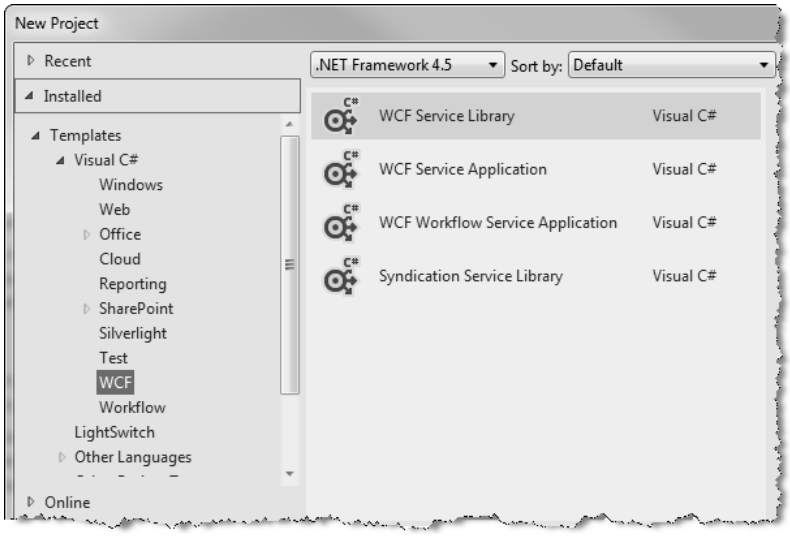

Abbildung 3.7 Visual Studio stellt Vorlagen für die Erstellung von WCF-Projekten zur Verfügung

Beide Nachteile gelten auch für unser Vorhaben, weshalb ich für die Erstellung der ersten WCF-Lösung einen anderen Weg empfehle. Dieser Weg sieht vor, zuerst eine leere Lösung zu erzeugen, und anschließend in dieser leeren Lösung, Schritt für Schritt, die einzelnen Assemblys zu ergänzen.

Wie Abbildung 3.8 zeigt, steht in Visual Studio für das Erstellen einer leeren Lösung auch eine Vorlage zur Verfügung. Diese ist in den Vorlagenkategorien unter *Other Project Types/Visual Studio Solutions* untergebracht.

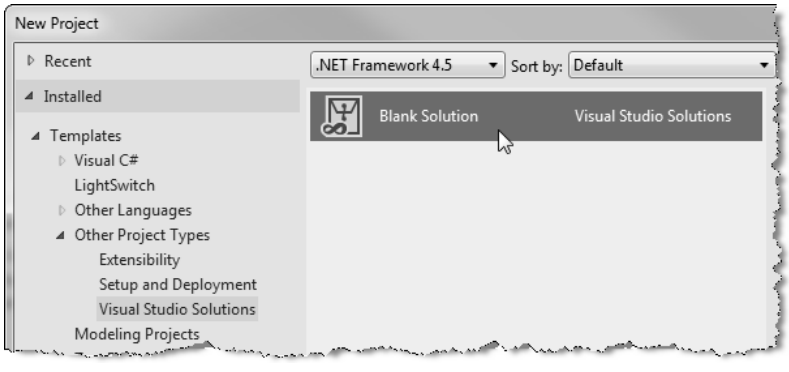

Abbildung 3.8 Erstellen einer leeren Lösung in Visual Studio

Die leere Lösung ergänzen wir Schritt für Schritt mit den notwendigen Projekten. Aufgrund der Abhängigkeiten (siehe Abbildung 3.5) ist es sinnvoll, die Realisierung auch in der Reihenfolge dieser Abhängigkeiten und zusätzlich unter Berücksichtigung der funktionellen Abhängigkeiten (Client ist abhängig von Server) zu realisieren. Daraus ergibt sich folgende Realisierungsreihenfolge:

1. Assembly für Verträge
2. Assembly für Dienstimplementierung
3. Assembly für Server
4. Assembly für Client (nutzt den Server funktionell, es besteht keine Abhängigkeit auf der Stufe der Assembly zum Server)

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

Erstellen der Verträge

Nach dem Erstellen der leeren Lösung ergänzen wir das erste Projekt um die Erstellung der Assembly für die Verträge. Auch hier können Sie wieder zwischen der Vorlage *WCF Service Library* und der Vorlage *Class Library* für eine normale Bibliothek entscheiden. Der Unterschied ist nur marginal. Die normale Bibliothek hat noch keine Referenzierung auf die notwendigen .NET-Assemblys *System.ServiceModel* und *System.Runtime.Serialization*. Demzufolge müssen diese im Fall einer normalen Bibliothek manuell hinzugefügt werden, während die Vorlage *WCF Service Library* diese Referenzierungen generiert. In beiden Fällen werden Klassen und im Fall einer *WCF Service Library* auch noch eine Schnittstelle generiert. Da der generierte Code allerdings in jedem Fall komplett verändert werden muss, führt der schnellste Weg unweigerlich über das Löschen der generierten Dateien und das Neuerstellen von Dateien für die eigenen Typen.

In unserem ersten Projekt benötigen wir für die Beschreibung der Verträge eine Schnittstelle, eine Datenklasse für Nutzdaten und eine Datenklasse für eine Fehlerbeschreibung. Listing 3.1 zeigt zunächst die Schnittstelle des Diensts. Achten Sie darauf, dass die Schnittstelle öffentlich definiert wird. Ferner wird der Typ mit dem Attribut `ServiceContractAttribute` ausgestattet.

Die Methoden der Schnittstelle, die auch tatsächlich von außen erreichbar sein sollen, werden mindestens mit dem Attribut `OperationContractAttribute` ausgestattet. Für die Handhabung von Fehlern wird die Methode mit einem oder mehreren Attributen des Typs `FaultContractAttribute` versehen.

```
// Definiert die Schnittstelle des Diensts.
[ServiceContract(Namespace = "http://WeroSoft.net/Samples/LearningWcf")]
public interface IwlbDemoService
{
    // Liefert ein Array von Daten.
    [OperationContract]
    [FaultContract(typeof(CwlbFault))]
    CwlbDemoData[] GetDemoData(CwlbDemoData demoData, int iParam);
}
```

Listing 3.1 Band_3\Kapitel_03\LearningWCF\WeroSoft.Samples.LearningWcfContract\IwlbDemoService.cs

HINWEIS Beachten Sie, dass Dienstverträge in der WCF nur mit Methoden ausgestattet werden. Eigenschaften, Indexer und Ereignisse werden in der WCF nicht unterstützt. Ebenso können wegen der Definition von Webdiensten durch das W3C auch keine Methodenüberladungen unterstützt werden.

Die Auswirkung und die weiteren Möglichkeiten der Attribute `ServiceContractAttribute` und `OperationContractAttribute` sind im Abschnitt »Der Dienstvertrag« im vierten Kapitel beschrieben. Die Möglichkeiten und Auswirkungen für das Attribut `FaultContractAttribute` entnehmen Sie dem Abschnitt »Der Fehlervertrag« im vierten Kapitel und im Abschnitt »Fehlerhandhabung in einem Dienst« im sechsten Kapitel.

Die Methode `GetDemoData()` benutzt für die Eingabe den eigenen Typ `CwlbDemoData` und für die Rückgabe ein Array desselben Typs. Dieser Typ wird ebenfalls in der Assembly für Verträge definiert. Listing 3.2 zeigt die Definition des Typs. Beachten Sie die Anwendung der Attribute `DataContractAttribute` für die Definition des Typs und `DataMemberAttribute` für die Kennzeichnung der Daten, die übermittelt werden sollen.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

```
// Definiert die Daten, die zwischen Server und Client ausgetauscht werden.
[DataContract(Namespace = "http://WeroSoft.net/Samples/LearningWcf")]
public class CwlbDemoData
{
    // Liefert oder definiert einen Integerwert.
    [DataMember]
    public int IntegerValue { get; set; }

    // Liefert oder definiert eine Zeichenfolge.
    [DataMember]
    public string StringValue { get; set; }

    // Liefert oder definiert einen Datumswert.
    [DataMember]
    public DateTime DateTimeValue { get; set; }

    // Liefert den Inhalt der Klasse.
    public override string ToString() {
            return string.Format("{0,5}. {1} - {2}",
                IntegerValue, DateTimeValue.ToShortDateString(), StringValue);
    }
}
```

Listing 3.2 Band_3\Kapitel_03\LearningWcf\WeroSoft.Samples.LearningWcfContract\CwlbDemoData.cs

HINWEIS Ein Datenvertrag enthält in aller Regel vor allem Daten. Es spielt dabei keine Rolle ob Sie Felder oder Eigenschaften verwenden. Die Definition von Methoden ist in reinen .NET-Projekten möglich und selbstverständlich auf beiden Seiten der verteilten Anwendung nutzbar. Werden die Daten von einem .NET- an einen nicht .NET-Teil der verteilten Anwendung übermittelt, geht die Funktionalität in der Datenklasse der Seite, die nicht .NET-basiert ist, verloren.

Grundsätzlich kann die WCF auch Daten übermitteln, die nicht mit den erwähnten Attributen versehen sind. Die Details zu den Attributen für Datenverträge entnehmen Sie dem Abschnitt »Der Datenvertrag« im vierten Kapitel.

Die Verwendung von wohldefinierten Fehlern ist in der WCF ein wichtiges Konzept für die erfolgreiche Nutzung der Verbindungen. Dazu stellt die WCF einige Standardtypen zur Verfügung. Um aber selbst definierte Fehlerinformationen zu benutzen, müssen eigene Fehlerklassen definiert werden. Im Gegensatz zu den normalen CLR-Ausnahmen verwendet die WCF jedoch keine Ableitungen des Basistyps `Exception`, sondern einfache Datenklassen, die im Zusammenhang mit einer speziellen generischen WCF-Fehlerklasse verwendet werden.

Listing 3.3 zeigt die Definition der in diesem Beispiel verwendeten Fehlerklasse. Ein Unterschied in der Syntax zu der vorhin definierten Klasse für Nutzdaten ist nicht vorhanden. Lediglich der Typenname verrät uns hier, dass es sich wohl um eine Fehlerklasse handelt. Beachten Sie, dass die Eigenschaft `Created` schreibgeschützt definiert ist. Das heißt in diesem Fall, dass der Client diese Information nicht verändern kann. Die gehaltene Information der Eigenschaft wird beim Erzeugen des Objekts auf der Serverseite erstellt und in der Eigenschaft gespeichert.

```
// Definiert ein Fehlerdetail im Fall des Auftretens eines Fehlers auf der Serverseite.
[DataContract(Namespace = "http://WeroSoft.net/Samples/LearningWcf")]
public class CwlbFault
{
    // Liefert oder definiert einen Detailfehlercode.
    [DataMember]
    public int Code { get; set; }
```

```
    // Liefert oder definiert einen detaillierten Text zur Fehlersituation.
    [DataMember]
    public string Message { get; set; }

    // Liefert oder definiert den Zeitpunkt der Erzeugung des Fehlers.
    // Diese Eigenschaft dient lediglich der Demonstration der Allgemeingültigkeit und
    // ist nicht zwingend von der WCF oder SOAP vorgeschrieben.
    [DataMember]
    public DateTime Created { get; private set; }

    // Initialisiert eine neue Instanz des Typs CwlbFault.
    // Diese Eigenschaft dient lediglich der Demonstration der Allgemeingültigkeit und
    // ist nicht zwingend von der WCF oder SOAP vorgeschrieben.
    public CwlbFault() {
        Created = DateTime.UtcNow;
    }
}
```

Listing 3.3 Band_3\Kapitel_03\LearningWcf\WeroSoft.Samples.LearningWcfContract\CwlbFault.cs

Implementierung des Servers

Nach der Definition der Verträge kann im zweiten Projekt unserer Lösung bereits die serverseitige Funktionalität implementiert werden. Zu diesem Zweck erstellen wir in der Lösung eine zweite Assembly. Am einfachsten gehen Sie dazu genauso vor, wie bereits bei der Assembly für die Verträge.

Listing 3.4 zeigt die dazu notwendige Lösung. Speziell zu erwähnen ist hier, dass der Typ nicht mit einem Attribut versehen ist (siehe dazu auch Hinweis nach Listing 3.4). Ferner prüft die Methode `GetDemoData()` zuerst die Gültigkeit der beiden Argumente. Wird dabei ein Verstoß der Regeln festgestellt, wird eine Ausnahme ausgelöst. Die Ausnahme ist vom Typ `FaultException<T>`. Dieser Typ ist in der WCF vordefiniert und wird zusammen mit unserem Fehlervertrag `CwlbFault` verwendet.

Die Erstellung der Nutzdaten dient lediglich dem Beispiel und hat keine codetechnischen Highlights.

```
// Implementierung des Diensts IwlbDemoService.
public class CwlbDemoService : IwlbDemoService
{
    // Liefert ein Array von Daten
    public CwlbDemoData[] GetDemoData(CwlbDemoData objDemoData, int iParam)
    {
        // Prüfen der Eingabeparameter
        if (objDemoData == null) {
            CwlbFault objFault = new CwlbFault();
            objFault.Code = 1;
            objFault.Message = "Die Vorlage muss angegeben werden.";
            throw new FaultException<CwlbFault>(
                objFault,
                new FaultReason("Parameterprüfung"),
                new FaultCode("1"));
        }
        if (iParam < 1) {
            CwlbFault objFault = new CwlbFault();
            objFault.Code = 2;
```

```
            objFault.Message = "Die Anzahl der angeforderten Elemente darf nicht kleiner als 1 sein.";
            throw new FaultException<CwlbFault>(
                objFault,
                new FaultReason("Parameterprüfung"),
                new FaultCode("1"));
        }

        // Herstellen der Daten
        List<CwlbDemoData> cobjData = new List<CwlbDemoData>();
        for (int iCount = 0; iCount < iParam; iCount++) {
            CwlbDemoData objData = new CwlbDemoData();
            objData.IntegerValue = iCount + 1;
            objData.StringValue = objDemoData.StringValue + " - Antwort";
            objData.DateTimeValue = new DateTime(2012, 1, 1) + TimeSpan.FromDays(10 * iCount);
            cobjData.Add(objData);
        }
        return cobjData.ToArray();
    }
}
```

Listing 3.4 Band_3\Kapitel_03\LearningWCF\WeroSoft.Samples.LearningWcfImplementation\CwlbDemoService.cs

HINWEIS Die Implementierung eines Diensts kann mit dem Attribut `ServiceBehaviorAttribute` oder `CallbackBehaviorAttribute` ausgestattet werden. Die Anwendungsfälle und die Wirkung dieser beiden Attribute entnehmen Sie dem Abschnitt »Einen Dienst implementieren« im sechsten Kapitel.

Selbstverständlich ist es möglich, nach der Dienstimplementierung einen Unit-Test für die Qualitätssicherung der Implementierung zu erstellen. In der Praxis empfiehlt sich dies in der Regel auch. Sie sollten sich aber immer bewusst sein, dass die Prüfung eines Diensts ohne effektive Nutzung der WCF nicht dieselbe Qualitätssicherung ist, wie die Nutzung über die WCF. Ich empfehle Ihnen deshalb, einen Dienst immer über die WCF zu testen. Mehr zum Testen von WCF-Diensten lesen Sie im dreizehnten Kapitel.

Erstellen des Hosts

Mit der Fertigstellung des Diensts können wir uns nun um die Erstellung des Serverprozesses für die Ausführung des Diensts kümmern. Der künftige Serverprozess wird zwei Hauptaufgaben wahrnehmen:

- Die Konfiguration der serverseitigen Infrastruktur der WCF
- Die Ausführung der Dienstobjekte

Während die Konfiguration der WCF-Infrastruktur eine Aufgabe ist, die normalerweise am Anfang des Lebenszyklus des Prozesses steht, ist die Ausführung während der ganzen Lebensdauer sicherzustellen. Damit wird auch deutlich, dass der Serverprozess sehr lange laufen muss, damit der Dienst über lange Zeit von den Clients angesprochen werden kann.

Für die Erledigung dieser Aufgabe benutzen wir in unserem ersten Beispiel ein Konsolenprogramm. Konsolenprogramme haben die Eigenschaft, dass Sie nach der Ausführung automatisch beendet werden und somit der Prozessraum auch wieder geschlossen wird. Dieses Problem lösen wir, indem der Hauptthread des

Programms einfach mit einer Konsoleneingabe beschäftigt wird. Die Nutzung der Dienstobjekte benötigt den Hauptthread nicht, denn die Dienstobjekte werden von der WCF mit Threads aus dem .NET-Threadpool bearbeitet.

Erstellen Sie nun in der aktuellen Lösung ein weiteres Projekt. Der Projekttyp soll jetzt *Console Application* sein. Nach der Erstellung referenzieren Sie die beiden Assemblys *System.ServiceModel* und *System.Runtime.Serialization*. Anschließend referenzieren Sie noch die beiden eigenen Assemblys der Lösung (Vertrag und Implementierung). Nach diesen vorbereitenden Schritten kann die Methode `Main()` der generierten Klasse `Program` an unsere Bedürfnisse angepasst werden.

Listing 3.5 zeigt das einfache Konsolenprogramm, das in unserem Fall die Aufgabe des so genannten Hostings des Diensts übernimmt. Der Code sieht eine Ausnahmebehandlung vor. Diese dient einfach dem Festhalten des Fehlers, sofern etwas während der Initialisierung schief geht. Nehmen Sie bereits hier zur Kenntnis, dass ein Fehler beim Aufruf des Diensts durch einen Client diese Ausnahmebehandlung nicht nutzen kann, da der Aufruf vom Client in einem anderen Thread stattfinden wird.

Für die Ausführung des Diensts benutzen wir ein Objekt der Klasse `ServiceHost` von .NET Framework. Die Konfiguration der WCF-Infrastruktur beginnt mit dem Erstellen des Servicehost-Objekts. Der angesprochene Konstruktor der Klasse wird mit dem Typ des Diensts bekannt gemacht, den das entstehende Servicehost-Objekt zur Verfügung stellen soll.

Der zweite Schritt der Konfiguration besteht darin, das Servicehost-Objekt mit einem Kommunikationsendpunkt zu verbinden. Damit wird die WCF konfiguriert, auf einer oder mehreren IP-Adressen mit einem definierten Port einen Abhörthread zu etablieren. Dieser Abhörthread übernimmt die Aufgabe, ankommende Daten entgegenzunehmen und, sofern passend, an den Dienst weiterzuleiten. Das kann die WCF allerdings nur tun, wenn sie weiß, mit welchen Kommunikationsprotollen und welcher Datencodierung gearbeitet werden soll. Beide Angaben werden mit der so genannten Bindung festgelegt. Alle diese Definitionen werden mithilfe der Methode `AddServiceEndpoint()` an das Servicehost-Objekt übermittelt.

Nach der abgeschlossenen Konfiguration durch die beiden soeben besprochenen Schritte, kann der Servicehost zur Arbeit aufgefordert werden, indem die Methode `Open()` aufgerufen wird. Da der Servicehost nun selbständig ist, braucht unser Programm nur noch dafür zu sorgen, dass der Prozessraum vorerst bestehen bleibt. Diese Aufgabe löst die Zeile `Console.Readkey()`.

Wird dann später der Prozessraum geschlossen, wird der Servicehost im `finally`-Block entweder geschlossen oder, für den Fall, dass er einen fehlerhaften Status aufweist, mit der Methode `Abort()` in den Zustand `Closed` überführt.

```
// Implementiert den Servicehost.
class Program
{
    // Einsprungpunkt in das Programm.
    static void Main()
    {
        // Definiert den Servicehost.
        ServiceHost objHost = null;

        try
        {
            // Erstellen des Servicehosts für den Dienst CwlbDemoService
            objHost = new ServiceHost(typeof(CwlbDemoService));
```

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

```
            // Defineren eines Kommunikationsendpunkts
            objHost.AddServiceEndpoint(
                typeof(IwlbDemoService),
                new NetTcpBinding(SecurityMode.None),
                string.Format("net.tcp://localhost:4711/{0}", GwlbLearningWcf.WcfDemoServiceName));
            // Dienst starten
            objHost.Open();
            // Verhindern, dass Prozess beendet wird.
            Console.WriteLine("Servicehost ist gestartet. Mit einem Tastendruck kann er beendet werden ...");
            Console.ReadKey(true);
        }
        catch (Exception excObject)
        {
            // Ausnahmebehandlung
            Console.WriteLine(excObject.Message);
        }
        finally
        {
            // Beenden des Diensts im Normalfall
            if (objHost != null && objHost.State == CommunicationState.Opened)
            {
                objHost.Close();
            }
            // Beenden des Diensts im Ausnahmefall
            if (objHost != null && objHost.State == CommunicationState.Faulted)
            {
                objHost.Abort();
            }
        }
    }
}
```

Listing 3.5 Band_3\Kapitel_03\LearningWCF\WeroSoft.Samples.LearningWcfHost\Program.cs

Nach der erfolgreichen Kompilierung des Codes des Hostprojekts können Sie den Server ruhig einmal starten. Sie sollten im entstehenden Konsolenfenster lediglich die Ausgabe sehen, dass der Service Host gestartet ist und das Programm mit einem Tastendruck beendet werden kann.

HINWEIS Das erstmalige Starten des Hostprojekts kann es zu einer Anfrage der lokalen Firewall führen, die wissen möchte, ob das soeben gestartete Programm in die Liste der erlaubten Programme für die Nutzung von Ports aufgenommen wird und der Port geöffnet werden soll. Bestätigen Sie diese Anfrage, ansonsten wird später die Lösung nicht funktionieren. Diese Anfrage erfolgt einmal für den Start ohne Debugger und einmal für den Start mit Debugger. Der Grund hierfür besteht darin, dass Visual Studio Debugcode über die spezielle projektspezifische Datei <*Projekt* >.*vshost.exe* ausführt.

Dienste in einem Konsolenprogramm zu hosten, ist nicht praxisorientiert. Für die Ausbildung und die Laborumgebung im Alltag sind sie aber ein praktisches und einfach zu handhabendes Mittel. Die Konfiguration der WCF-Infrastruktur kann zudem sowohl imperativ (wie in diesem Beispiel) als auch deklarativ erfolgen.

Mehr zu den verschiedenen Verfahren und Varianten der Konfiguration des Hostings von Diensten entnehmen Sie bitte dem siebten Kapitel.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

Erstellen des Clients

Der letzte Akt in der Erstellung des ersten Entwicklungszyklus beginnt. Die Generierung des Clients besteht ebenfalls darin, ein Konsolenprogramm zu erstellen. Das Programm dient dazu, den serverseitigen Dienst einmal mit gültigen Argumenten und dann mit ungültigen Argumenten anzusprechen.

Erstellen Sie nun in der aktuellen Lösung ein weiteres Projekt. Der Projekttyp soll erneut vom Typ *Console Application* sein. Nach der Erstellung referenzieren Sie die beiden Assemblys *System.ServiceModel* und *System.Runtime.Serialization*. Anschließend referenzieren Sie die eigenen Assemblys für Verträge. Nach diesen vorbereitenden Schritten kann die Methode `Main()` der generierten Klasse `Program` an unsere Bedürfnisse angepasst werden.

Listing 3.6 zeigt den dazu notwendigen Code. Dieser definiert als erstes eine Ausnahmebehandlung. Diese ist für Clients von verteilten Anwendungen besonders wichtig, denn wir sprechen externe Ressourcen an. Bei diesem Vorgang ist die Vielfalt der möglichen Fehler naturgemäß größer als beim Ansprechen von internen Ressourcen.

Das eigentliche Ansprechen des Diensts erfolgt in unserem ersten Beispiel auf der Basis der definierten Dienstschnittstelle. Das dazu notwendige Objekt `objProxy` wird mithilfe der WCF-Infrastruktur erzeugt. Zu diesem Zweck nutzen wie die WCF-Klasse `ChannelFactory<T>`. Diese Klasse stellt uns die statische Methode `CreateChannel()` zur Verfügung, mit deren Hilfe wir ein Proxyobjekt zum Dienst erzeugen können.

Die Methode `CreateChannel()` erfordert für die Konfiguration der clientseitigen WCF-Infrastruktur wiederum verschiedene Angaben. Dazu gehören die Definition der zu verwendenden Kommunikationsprotokolle, der Codierung der Daten und die anzusprechende Zieladresse im Netzwerk. Die beiden ersten Angaben werden wiederum mit der Bindung definiert. Die Adresse wird mithilfe einer Zeichenfolge und der Klasse `EndpointAddress` als zweites Argument an die Methode `CreateChannel()` übergeben.

WICHTIG Ein Proxyobjekt eines Diensts definiert die clientseitige Instanz, über die ein Dienst angesprochen werden kann. Das Proxyobjekt implementiert den Dienst nicht. Es arbeitet mit der WCF-Infrastruktur zusammen und übermittelt die Daten des Methodenaufrufs an den Dienst auf dem Server und empfängt dessen Antwort, die als normale CLR-Antwort wieder an den Aufrufer zurückgereicht wird.

Beachten Sie, dass die Portangabe des Servers und der WCF-interne Adressierungsname sowie die verwendete Bindung und Codierung auf dem Server und dem Client identisch sind. Sind diese Angaben nicht auf Client und Server gleich definiert, wird die Anwendung nicht funktionieren.

Nach der Erstellung des Proxyobjekts kann dieses direkt benutzt werden, um den entfernten Dienst anzusprechen. In unserem Beispiel in Listing 3.6 geschieht dies durch den zweimaligen Aufruf der Methode `GetDemoData()`. Der erste Aufruf benutzt zwei gültige Werte, somit ist das Resultat verwendbar. Die zwanzig auf der Serverseite erzeugten Objekte werden in der Konsole aufgelistet.

Der zweite Methodenaufruf verwendet einen ungültigen zweiten Parameter, was dazu führt, dass serverseitig ein Fehler ausgelöst und an den Client übermittelt wird. Der Fehlertyp ist in diesem Fall `FaultException<T>`. `FaultException<T>` ist eine Spezialisierung der allgemeinen Ausnahme `CommunicationException` und muss demzufolge in der Behandlung führend erwähnt werden. Eine `FaultException<T>` kann mehrfach genannt werden, sofern die generischen Typenparameter unterschiedlich sind. In unserem Fall ist der generische Typenparameter die eigene Fehlerklasse. Genau dieser Typ wird im Fehlerfall auf der Serverseite produziert und demzufolge hier wieder abgefangen und ausgewertet.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

Den Abschluss des clientseitigen Programms bildet auch hier das korrekte Beenden des Proxyobjekts, das entweder den Proxykanal schließt oder, sofern der Proxy in einem fehlerhaften Zustand ist, den Proxy unabhängig von laufenden Aktionen schließt.

```
// Implementiert den Client.
class Program
{
    // Einsprungpunkt in das Programm.
    static void Main()
    {
        IwlbDemoService objProxy = null;

        try {
            // Zieladresse
            string strHostIPAddress = "localhost";

            // Normalen Client erstellen
            objProxy = ChannelFactory<IwlbDemoService>.CreateChannel(
                new NetTcpBinding(SecurityMode.None),
                new EndpointAddress(
                    string.Format("net.tcp://{0}:4711/{1}",
                        strHostIPAddress,
                        GwlbLearningWcf.WcfDemoServiceName)));

            // Aufruf ohne Fehler.
            Console.WriteLine("Aufruf ohne Fehler ...");
            CwlbDemoData objData = new CwlbDemoData { DateTimeValue = DateTime.Now, StringValue = "Anfrage" };
            CwlbDemoData[] aobjData = objProxy.GetDemoData(objData, 20);
            foreach (CwlbDemoData objItem in aobjData) {
                Console.WriteLine(objItem);
            }

            // Aufruf mit Fehler
            Console.WriteLine();
            Console.WriteLine("Aufruf mit serverseitigem Fehler ...");
            aobjData = objProxy.GetDemoData(objData, -2);
        }
        catch (FaultException<CwlbFault> excObject) {
            // Verarbeitung des Serverfehlers
            Console.WriteLine("Eigener Fehler gefangen:");
            Console.WriteLine(string.Format("   Fehlertext:          {0}", excObject.Message));
            Console.WriteLine(string.Format("   Fehlergrund:         {0}", excObject.Reason));
            Console.WriteLine(string.Format("   Fehlercode:          {0}", excObject.Code.Name));
            Console.WriteLine(string.Format("   Text Fehlerdetail:   {0}", excObject.Detail.Message));
            Console.WriteLine(string.Format("   Code Fehlerdetail:   {0}", excObject.Detail.Code));
            Console.WriteLine(string.Format("   Datum Fehlerdetail:  {0}", excObject.Detail.Created));
        }
        catch (CommunicationException excObject) {
            // Verarbeitung einer allgemeinen Kommunikationsausnahme
            Console.WriteLine(excObject.Message);
        }
        catch (Exception excObject) {
            // Verarbeitung aller anderen Ausnahmen
            Console.WriteLine(excObject.Message);
        }
```

```
        finally {
            // Beenden des Clients im Normalfall
            if (objProxy != null && ((IChannel)objProxy).State == CommunicationState.Opened) {
                ((IChannel)objProxy).Close();
            }
            // Beenden des Clients im Ausnahmefall
            if (objProxy != null && ((IChannel)objProxy).State == CommunicationState.Faulted) {
                ((IChannel)objProxy).Abort();
            }
        }
    }
}
```

Listing 3.6 Band_3\Kapitel_03\LearningWCF\WeroSoft.Samples.LearningWcfClient\Program.cs

HINWEIS Die WCF definiert unterschiedliche Möglichkeiten der Generierung clientseitiger Aufrufe von Diensten. Mehr dazu lesen Sie im achten Kapitel.

Ausführen des Codes

Nach dem Aufbau der Lösung wollen wir diese natürlich auch ausführen und den Umgang in Visual Studio mit mehreren Projekten kennenlernen. Dazu gehören auch das Debugging und die Spezialitäten des Debuggings einer verteilten Anwendung.

Eine Lösung als Multistart-Lösung definieren

Da die Lösung über zwei ausführbare Dateien für die Erstellung der beiden Prozesse auf der Client- und der Serverseite verfügt, müssen für eine erfolgreiche Ausführung beide auch gestartet werden. Im *Solution Explorer* kann über das Kontextmenü ein Startprojekt festgelegt werden. In unserer aktuellen Situation ist das nicht genug. Wir müssen in der Lage sein, zwei Projekte gleichzeitig zu starten. Diese Möglichkeit bietet der Eigenschaftseditor auf der Ebene der Lösung.

Abbildung 3.9 zeigt das Dialogfeld mit den Eigenschaften der Lösung. Wählen Sie darin die Option *Multiple startup projects*. Definieren Sie anschließend für die zu startenden Projekte *LearningWcfHost* und *LearningWcfClient* den Eintrag *Action Start* oder *Action without Debugging*. Letztendlich ordnen Sie die Prozesse in der Liste von oben nach unten, sodass die zeitlich früher zu startenden Prozesse oberhalb der zeitlich später zu startenden Prozesse erscheinen. Sie verwenden dazu die beiden Pfeilschaltflächen am rechten Rand des Dialogfelds.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

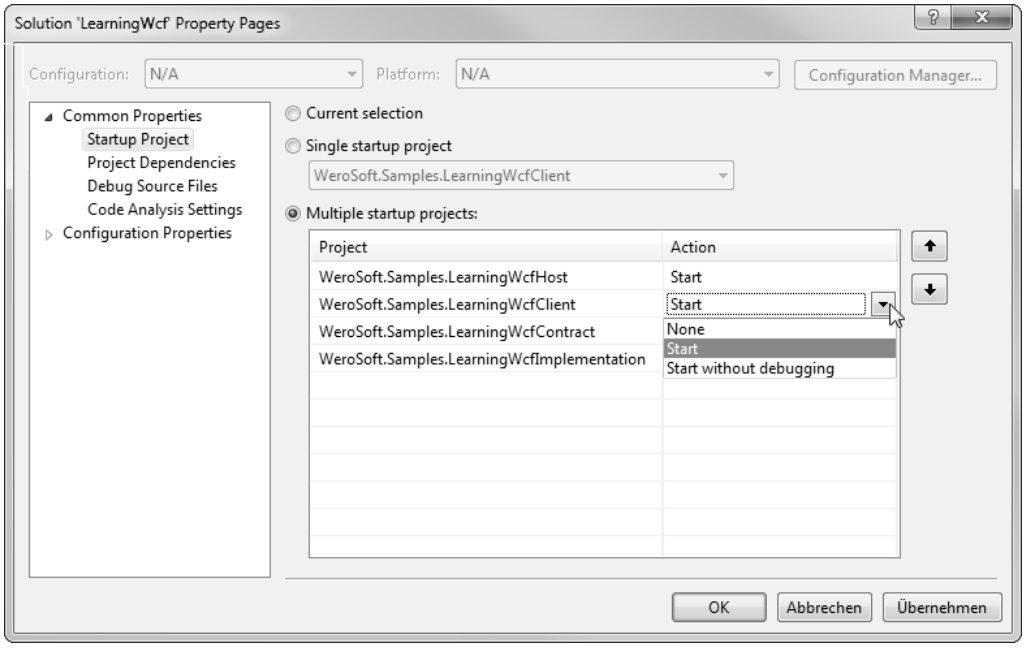

Abbildung 3.9 Einstellung als *Multiple startup project* in den Eigenschaften der Lösung

Nach diesen Einstellungen können Sie die Lösung in Visual Studio ausführen. Selbstverständlich können Sie auch zwei Konsolenfenster erstellen und darin die Anwendungsteile manuell starten. In diesem Fall müssen Sie aber jeweils daran denken, den Server zunächst zu starten.

Das Resultat der Ausführung in den Konsolenfenstern zeigt die Inhalte gemäß Abbildung 3.10 und Abbildung 3.11.

HINWEIS Das erstmalige Starten des Hostprojekts kann zu einer Anfrage der lokalen Firewall führen, die wissen möchte, ob das soeben gestartete Programm in die Liste der erlaubten Programme für die Nutzung von Ports aufgenommen wird und der entsprechende Port geöffnet werden soll. Bestätigen Sie diese Anfrage, ansonsten wird die Lösung später nicht funktionieren.

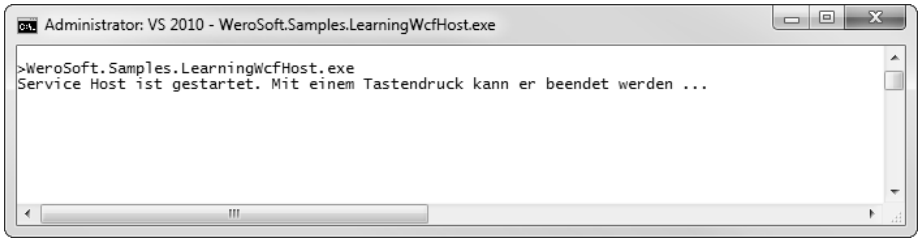

Abbildung 3.10 Ausführung des Programms *LearningWcfHost.exe*

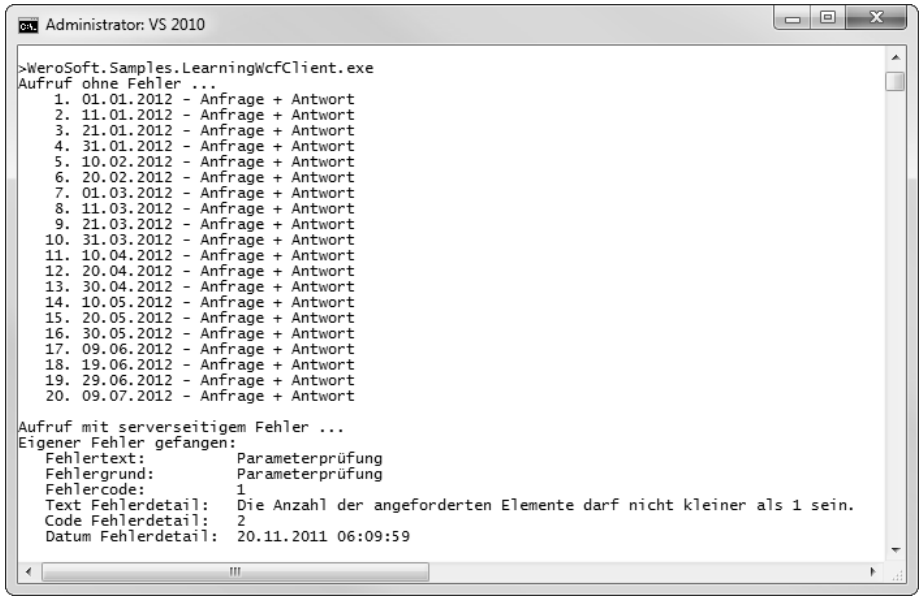

Abbildung 3.11 Ausführung des Programms *LearningWcfClient.exe*

Eine Lösung debuggen

Durch die Definition des Mehrfachstartprojekts in den Eigenschaften der Lösung, können beide Prozesse auch mit dem Debugger gestartet werden. Dabei können Sie selbstverständlich auch beliebige Haltepunkte definieren.

WICHTIG Bei der Definition von Haltepunkten in einer verteilten Anwendung kann es zu Nebeneffekten in dem Teil der Anwendung kommen, der nicht mit einem Haltepunkt versehen ist. Der Grund hierfür besteht darin, dass die WCF Aufrufe zeitlich überwacht, und nach definierten Timeouts Ausnahmen produziert. Wird also zum Beispiel auf der Serverseite ein Haltepunkt definiert und dann an dieser Stelle lange im Debugger verharrt, um den Ablauf zu verfolgen oder Werte zu kontrollieren, kommt es in unserem Beispiel nach einer Zeitspanne von zehn Minuten zu einer Timeoutausnahme im Client. Mit diesem Umstand können Sie entweder leben (die Lösung arbeitet ja beim Kunden dann viel schneller) oder durch Konfiguration der verwendeten WCF-Einstellungen das Verhalten ändern. Mehr zur Konfiguration der Zeitwerte lesen Sie in Abschnitt »Endpunkte, Protokolle und Bindungen« im siebten Kapitel.

Auf der Clientseite kann mit dem Debugger festgestellt werden, ob die Verbindung zum Server auch tatsächlich hergestellt wurde. Zu diesem Zweck erzeugen Sie einen Haltepunkt nach der Erstellung des Proxyobjekts, starten die beiden Anwendungsteile und nutzen nach dem Anhalten des Clients die Möglichkeit, die Variableninhalte zu betrachten.

Das Proxyobjekt entpuppt sich bei dieser Betrachtung als Objekt vom Typ `__TransparantProxy` aus dem Namensraum `System.Runtime.Remoting.Proxies`. Dieser Typ sagt uns unweigerlich, dass die WCF den richtigen Typ erstellt hat (siehe auch Abbildung 3.12). Ob allerdings die Verbindung mit dem Server korrekt funktioniert, ist nicht nur von der Existenz dieses Objekts abhängig, sondern auch von der exakten Konfiguration. Die Konfiguration kann dem Objekt auch entlockt werden, indem der Typ im Debugger erweitert wird und die entsprechenden Daten geprüft werden.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

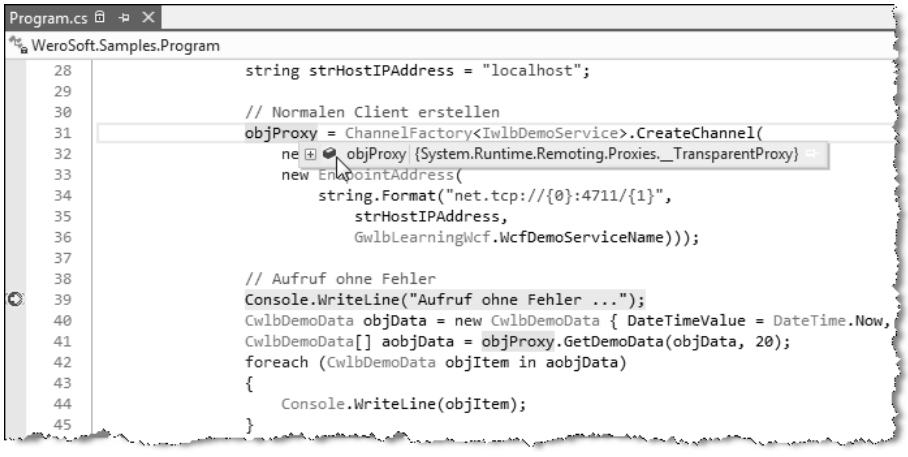

Abbildung 3.12 Clientseitiges Debuggen und Darstellen des Proxyobjekts

Die Erweiterung der Daten des Proxyobjekts im Debugger offenbart die Schnittstelle, das Objekt für die Verwaltung des Kommunikationskanals (`serviceChannel`) und vieles mehr. In den enthaltenen Daten finden Sie unter anderem die eingestellte Kommunikationsadresse, die Konfigurationswerte der WCF und den Namen der Schnittstelle, die das Proxyobjekt implementiert.

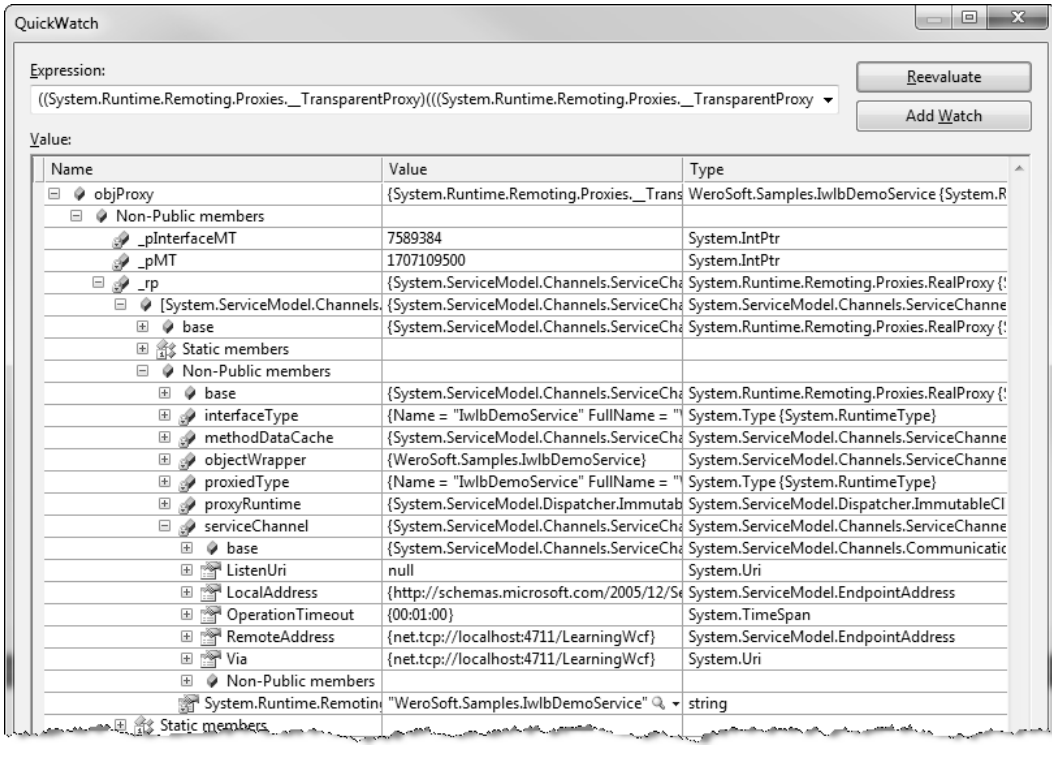

Abbildung 3.13 Erweiterung des `TransparentProxy` im *QuickWatch*-Fenster

Beim Debuggen auf der Serverseite werden Sie Ihre Arbeiten in der Regel auf die Dienstobjekte konzentrieren. Zu diesem Zweck definieren Sie an der gewünschten Stelle im Dienstobjekt einen Haltepunkt und starten die Programme. Das Debugging geschieht nun routinemäßig. Im Gegensatz zur Clientseite ist auf der Serverseite im Code innerhalb des Dienstobjekts kein Objekt vorhanden, das die Daten der WCF für das Dienstobjekt reflektiert. Es ist allerdings möglich, diese Daten über das *Watch*-Fenster, das Dialogfeld *QuickWatch* oder auch das *Immediate Window* in Visual Studio abzufragen. Zu diesem Zweck rufen Sie das Objekt des aktuellen `OperationContext` ab. Das Objekt wird mittels der statischen Eigenschaft `Current` der Klasse `OperationContext` erstellt, wie in Abbildung 3.14 dargestellt.

Mithilfe des `OperationContext`-Objekts kann der Kontext der Instanz des Dienstobjekts und damit einmal mehr die Konfiguration des Diensts zur Laufzeit geprüft werden. Ferner kann das verantwortliche Objekt der Klasse `ServiceHost` für diesen Dienst ebenfalls in Erfahrung gebracht werden.

Abbildung 3.14 Abfragen des `OperationContext` in einer serverseitigen Methode

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

Das Beispiel über Maschinengrenzen hinweg ausführen

Eine verteilte Anwendung auf einer Maschine auszuführen, ist keine große Herausforderung, da normalerweise darauf lokal alle notwendigen Ressourcen und Rechte vorhanden sind. Allerdings sind verteilte Anwendungen nicht primär dafür vorgesehen, auf einer Maschine ausgeführt zu werden, sondern es sollten mehrere Maschinen involviert sein. Ich könnte mir vorstellen, dass Sie die Arbeit dieses Kapitels auch tatsächlich auf zwei Rechner verteilen und das Resultat so testen möchten.

Damit dieses Vorhaben nicht scheitert, müssen Sie folgende Vorkehrungen treffen:

- Ändern der Zieladresse im Clientprogramm
- Kopieren des Hostcodes auf eine entfernte Maschine
- Prüfen der Einstellungen der Firewall auf der entfernten Maschine oder alternativ diese kurzeitig ausschalten

HINWEIS Achten Sie darauf, dass der ausgewählte Zielrechner auch tatsächlich das .NET 4.5-Framework installiert hat, und Sie sollten sich auch vergewissern, dass der im Beispiel verwendete Port 4711 auf dem Zielrechner noch frei ist.

Eine Liste der verwendeten Ports können Sie auf einem Windows-Rechner mit dem Konsolenbefehl *netstat –a* abfragen. Sollte der Port auf der Zielmaschine bereits verwendet sein, müssen Sie einen freien Port wählen und diesen sowohl im Client (*LearningWcfClient*) als auch im Server (*LearningWcfHost*) anstelle des aktuell verwendeten Ports 4711 eintragen.

Ändern der Zieladresse im Clientprogramm

Nachdem Sie entschieden haben, welchen Rechner Sie als zweiten Rechner verwenden, ermitteln Sie von diesem Rechner die IP-Adresse oder seinen Namen. Tragen Sie nun im Projekt eine der beiden Angaben in die Variable `strHostIPAddress` der Klasse `Program` ein und kompilieren Sie die Lösung. Listing 3.7 zeigt für meinen Versuch den Namen `"M1WRSO2"` des Servers, den ich verwendete.

```
// Implementiert den Client.
class Program
{
    // Einsprungpunkt in das Programm.
    static void Main()
    {
        IwlbDemoService objProxy = null;

        try {
            // Zieladresse
            string strHostIPAddress = "M1WRSO2";

            // Normalen Client erstellen
            objProxy = ChannelFactory<IwlbDemoService>.CreateChannel(
                new NetTcpBinding(SecurityMode.None),
                new EndpointAddress(
                    string.Format("net.tcp://{0}:4711/{1}",
                        strHostIPAddress,
                        GwlbLearningWcf.WcfDemoServiceName)));
```

Listing 3.7 Veränderte Zieladresse im Client

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)

HINWEIS Achten Sie bei diesem Schritt unbedingt darauf, dass kein Tippfehler geschieht. Es gibt keine Verifikation des Namens. Entsprechend würde bei einer Falscheingabe der Versuch einfach scheitern, weil die WCF den Rechner oder auf dem Rechner den Dienst nicht erreichen würde.

Eventuell kann Ihr Netzwerk die Verwendung eines Computernamens nicht in eine IP-Adresse umwandeln. In diesem Fall verwenden Sie direkt die IP-Adresse des Servers im Code des Clients.

Kopieren des Codes des Hosts auf eine entfernte Maschine

Im zweiten Schritt kopieren Sie die Ausgaben des Compilers des Projekts *LearningWcfHost* (Verzeichnis *Band_3\Kapitel_03\LearningWCF\WeroSoft.Samples.LearningWcfHost\bin\Debug*) auf den definierten entfernten Rechner in ein Verzeichnis Ihrer Wahl. Hier können Sie die Funktionsweise prüfen, indem das Programm gestartet wird, und dabei im Konsolenfenster keine Fehlermeldung erscheint.

Prüfen der Firewalleinstellungen auf der entfernten Maschine

Vor der Ausführung der beiden Programme sollten Sie noch sicherstellen, dass die Firewall auf dem entfernten Rechner ihre Aufgabe korrekt erfüllt. Wenn der ausgewählte Zielrechner mit einem Windows Server-Betriebssystem läuft, ist die Standardeinstellung der lokalen Firewall so, dass nicht dynamisch angefragt wird, ob das gestartete Programm die Erlaubnis erhält, den Port zu öffnen. In diesem Fall müssen Sie entweder diese Option in der lokalen Firewall einschalten oder das soeben installierte Programm manuell in den *Inbound Rules* als gültig definieren.

HINWEIS Sofern Sie alle drei Schritte korrekt befolgt haben, sollte Ihr Programm verteilt über zwei Rechner problemlos funktionieren. Ich habe beim Aufbau des Beispiels darauf geachtet, dass das Programm sowohl in einem Umfeld mit zwei Rechnern in der gleichen Domäne als auch mit zwei Rechnern in unterschiedlichen Domänen funktioniert. Verantwortlich dafür ist die Einstellung `SecurityMode = SecurityMode.None` der definierten TCP/IP-Bindung.

Rolf Wenger: Handbuch der .NET 4.0/4.5-Programmierung, Band 3, Microsoft Press 2012
(ISBN 978-3-86645-440-8)