

## Windows Communication Foundation, ehemals Indigo

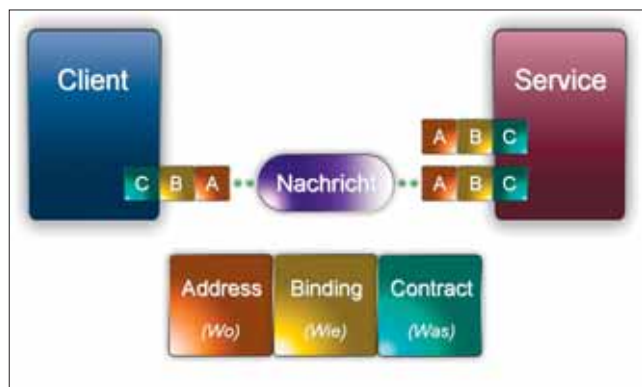
# Abgemacht!

Vereinbarungen sind wichtig, ja essenziell. Im richtigen Leben wie in der Softwareentwicklung. Auch mit der Windows Communication Foundation wird der Kontrakt zur wichtigsten Vereinbarung bei der Kommunikation zwischen Anwendungen. Ohne eine explizite Einigung aller Kommunikationspartner geht gar nichts – wenn es auch unterschiedliche Sicht- und Herangehensweisen gibt.

**D**ie Windows Communication Foundation, WCF, wird sich in den nächsten Jahren zum unumgänglichen Medium für die Erstellung verteilter Applikationen unter Windows entwickeln. In [5] wurden bereits die notwendigen Grundlagen besprochen. Dieser Beitrag erläutert die erste wichtige Säule beim Entwurf verteilter Anwendungen mit WCF, die Kontrakte.

Die vergangenen Jahre haben es gezeigt: nicht immer war die Art und Weise, wie verteilte Anwendungen konzipiert und erstellt wurden, besonders gelungen. Oftmals haben sich Softwarearchitekten und Entwickler fast blind auf die vom Hersteller oder der Basisplattform zur

**Abbildung 1**  
Beziehung zwischen Services und deren Konsumenten.



Verfügung gestellten Technologien zur Kommunikation verlassen. Zu aufreizend einfach ist es, einen entfernten Methodenaufruf zu implementieren, anstatt ein Objekt lokal über den Call Stack aufzurufen, dieses Objekt einfach in einer anderen Application Domain oder gleich auf einem anderen Computer aufzurufen. Es ist einfach, jeder macht es und es scheint zu funktionieren.

Remote Procedure Calling (RPC) scheint eine natürliche Erweiterung eines lokalen Programmiermodells zu sein. Sie ist einfach zu verstehen und scheint einfach anzuwenden, ist es aber nicht wirklich. Ein entfernter Methodenaufruf kann nicht mit den gleichen Voraussetzungen und Rahmenbedingungen funktionieren wie ein lokaler Aufruf. Einige Gründe, warum dies so ist:

- Die Koordination von Client und Server funktioniert prinzipiell anders.
- Was nach dem Anstoß einer Operation im Server passiert, wird anders geregelt.
- Client und Server haben einen anderen Kontext, keinen gemeinsamen Speicherbereich.
- Die Übermittlung von Daten zwischen Client und Server bedingt längere Latenzzeiten.

- Übertragungsweg oder notwendige Infrastruktur können ausfallen.
- Es gibt veränderte Sicherheitsaspekte.

Diesen Umstand haben übrigens schon die Technologie-Päpste aus der Java-Welt Jim Waldo & Co. festgestellt [6]. RPC ist einfach so prinzipiell anders als ein lokaler Methodenaufruf, dass es wirklich schwer zu verstehen ist, warum RPC-Kommunikation überhaupt in so hohem Ansehen steht. Die offensichtliche Alternative heißt entkoppelte Kommunikation über den Austausch von Nachrichten.

Dieser Beitrag kann nur die Oberfläche dieser Diskussion streifen. Ähnliche Argumente wurden bereits in früheren Beiträgen in der dotnetpro gebracht. Zudem ist jeder Leser eingeladen, eine Diskussion zwischen Ralf Westphal und mir zu diesem Thema online nachzulesen [4].

### Meta, Meta, Meta

Geht es um die Kommunikation zwischen Anwendungen, stehen nicht mehr Objekte, sondern Services im Mittelpunkt der Überlegungen. Damit ein Client, also ein Konsument eines Services, genau weiß, was er wie an einen Service schi-

### Auf einen Blick

#### Autor

**Christian Weyer** ist Mitbegründer von thinkecture, einer Firma, die Softwarearchitekten und Entwickler bei der Verwendung von .NET- und Web-Service-Technologien unterstützt. Er ist Mitglied im Indigo-Digerati-Team, das sehr frühen Zugriff auf Indigo Builds eingeräumt bekommt und Feedback an das Entwicklungsteam liefert. Sie erreichen ihn über [www.thinkecture.com/staff/christian](http://www.thinkecture.com/staff/christian).



dotnetpro.code  
A0511IndigoModell

Sprachen C#, XML

Technik Serviceorientierte Programmierung

Voraussetzungen Windows Communication Foundation Beta 1

cken muss, bedarf es für den Client notwendiger und hinreichender Informationen. Wie man in Abbildung 1 sehen kann, tauschen Clients und Services Nachrichten aus. Ein Service stellt seine Dienste über Kommunikationsendpunkte (Endpoints) zur Verfügung. Diese Endpunkte lassen sich in drei essentielle Artefakte gliedern

- Kontrakt (Contract),
- Bindung (Binding),
- Adresse (Address).

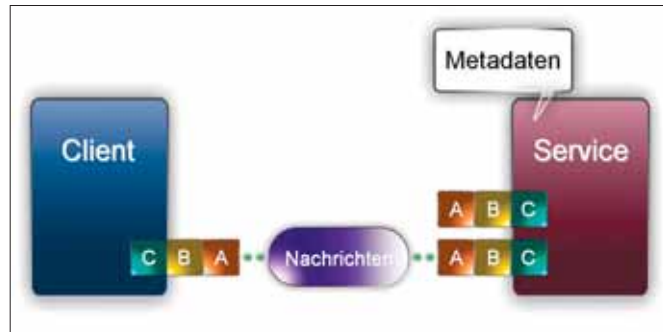
Liest man diese drei Konzepte in umgekehrter Reihenfolge, ergibt sich aus den Anfangsbuchstaben ABC. Microsoft nutzt diesen Umstand gern auch zur Marketingaussage, dass WCF so einfach sei wie das ABC. Wie viele Marketingaussagen ist auch diese etwas übertrieben. Schließlich kann, wer nur das ABC beherrscht, noch lange nicht schreiben und ist schon gar kein Schriftsteller.

Die Adresse eines Service-Endpunktes gibt das *Wo* an. Hier wird spezifiziert, an welche konkrete Adresse Nachrichten gesendet werden sollen. Das Binding gibt an, *wie* Nachrichten an einen Endpunkt geschickt werden, also beispielsweise ob über TCP oder HTTP, verschlüsselt oder im Klartext. Das eigentliche Herz, der Ursprung eines jeden Services, ist der Kontrakt. Hier wird überhaupt erst die Gestalt des Services definiert. *Was* kann ein Service bereitstellen und *was* akzeptiert er? Der Kontrakt ist somit das öffentliche Gesicht eines Services.

Doch wie sieht ein solcher Kontrakt aus? Weiter unten wird auf die exakte Ausprägung von Kontrakten vor allem im WCF-Umfeld eingegangen. Allgemein bleibt zunächst festzuhalten, dass ein Service oder ein Service-Anbieter den Kontrakt – oder die Kontrakte – über Metadaten modelliert und kommuniziert, wie in Abbildung 2 skizziert.

Von den klassischen Web Services ist diese Vorgehensweise bereits bekannt. Hier ist es üblich, dass Metadaten eines Web Services in Form von XML Schema (XSD) und Web Services Description Language (WSDL) formuliert werden. Nicht immer muss die Modellierung der Metadaten über XML und spitze Klammern geschehen. Man kann mit WCF Kontrakte auch recht komfortabel in .NET-Code erstellen. Allerdings sind Kontraktdateien nicht alles, was zur effektiven Kommunikation erforderlich ist. Zusätzlich zu den abstrakten Metadaten der Service-Schnittstelle sind noch Meta-

**Abbildung 2**  
Kontrakte werden über Metadaten modelliert und bekannt gemacht.



informationen für die tatsächliche Kommunikation notwendig. Diese werden über Bindings festgelegt. Mit Bindings beschäftigt sich dieser Beitrag noch nicht, sie werden in einem künftigen Artikel genauer beleuchtet.

Auf Basis der Metadaten können alle beteiligten Parteien sich einigen, welche Nachrichten zwischen Service-Konsument und Service-Anbieter ausgetauscht werden können.

Im Folgenden geht es zunächst um die Erstellung und Nutzung von Kontrakten mit Microsofts neuer Kommunikationsplattform WCF.

### Kontrakt gleich Kontrakt?

Anstatt einfach platt die neuen Features von WCF im Bereich Kontrakte herunterzubeten, soll versucht werden, WCF als Werkzeug zu betrachten. WCF als Mittel zur Umsetzung einer Idee, wie man Services und Service-Schnittstellen unabhängig von Technologie und Hersteller gut modelliert. Hier gibt es sicherlich unterschiedliche Sichtweisen. Dieser Beitrag begutachtet eine Sichtweise, die sich in der Praxis sehr gut bewährt hat, Details dazu sind in [7] zu finden.

Eine einfache Strategie zur Modellierung von Service-Schnittstellen beinhaltet drei einfache und klare Schritte, nämlich die Modellierung von

- Daten,
- Nachrichten,
- Schnittstelle.

Services arbeiten mit Daten, sie erhalten Daten von konsumierenden Anwendungen und schicken bei Bedarf Daten an interessierte Applikationen. Diese Daten wandern innerhalb von Nachrichten zwischen Services und Konsumenten, die Nachricht ist das zentrale Kommunikationsparadigma. Operationen innerhalb von Service-Schnittstellen fassen eine

oder mehrere Nachrichten zusammen. Man spricht hier auch von unterschiedlichen Nachrichtenaustauschmustern (*Message Exchange Patterns*). Daten, Nachrichten, Schnittstellen – ganz einfach zu verstehen und eine durchaus natürliche Vorgehensweise.

Damit ein Entwickler oder Architekt diese Grundidee mit WCF umsetzen kann, besitzt WCF Konstrukte und Artefakte, die den oben vorgestellten Artefakten in den drei Modellierungsphasen entsprechen.

Um für eine Schnittstelle und somit einen Kontrakt eine Struktur definieren zu können, bietet WCF die beiden Konstrukte Data Contract und Message Contract. Ein Data Contract repräsentiert die identifizierten und modellierten Daten einer Service-Kommunikation in der WCF-Welt.

Ein Message Contract hingegen kann optional verwendet werden, um exakt, sehr detailliert und spezifisch die Struktur und das Aussehen der tatsächlichen Nachricht, gemäß dem Prinzip des SOAP Envelopes, festlegen und manipulieren zu können. Letzteres Konstrukt hat sich bereits in den frühen Betaphasen als sehr speziell und nur für wenige Entwickler als hilfreich oder überhaupt sinnvoll herauskristallisiert. Denn in rund 90 Prozent aller Fälle genügt es, die Nachrichten ebenfalls als Data Contract festzulegen. Doch dazu gleich noch mehr.

Um nun zusätzlich zu den zu übertragenden Daten und auszutauschenden Nachrichten auch noch das Verhalten in Form einer Service-Schnittstelle bekommen zu können, bietet WCF den Service Contract an. Mit dem Konstrukt Service Contract lassen sich vor allem die Operationen und die Benennung der Schnittstellenelemente in .NET-Programmcode vornehmen.

Somit bietet WCF drei Konstrukte, die man entweder 1:1 umsetzen oder aber in-

### Contract First

Immer häufiger berichten Experten über so genannte Best Practices bei der Softwareentwicklung. Sei es Kollege und .NET Twin Ralf Westphal [1], [2] oder aber Wolfgang Pleus [3], alle scheinen derzeit fast aus dem Nichts heraus die Wichtigkeit von Schnittstellen neu entdeckt zu haben. „Schnittstelle“ oder „Contract First“ heißen die Schlagworte. Auch dem Autor der WCF-Artikel geht es so. Die Ideen und Konzepte von schnittstellenbasierter Programmierung werden mit zunehmender Softwarekomplexität immer wichtiger. Dies gilt für kleine Projekte ebenso wie für große, allein auf .NET basierende Systeme, ganz zu schweigen von unternehmensweiten Applikationen, die plattformübergreifend agieren. Die Schnittstelle, der Kontrakt, ist essenziell. Auch und vor allem bei verteilten Applikationen mit WCF.

direkt auf die oben vorgestellte Modellierungsstrategie anwenden kann:

- Data Contract beschreibt die Struktur.
- Message Contract beschreibt die Struktur.
- Service Contract beschreibt das Verhalten.

### Data Contract

Wie schon oben ausgeführt, geht es hier nicht um die Kommunikation zwischen Objekten und auch nicht um den Austausch von Objekten. Daten machen Anwendungen aus und Daten tauschen Anwendungen untereinander aus. Hier ein konkretes Beispiel: Ziel ist es, einen Service zu modellieren und bereitzustellen, über den man gewisse Aspekte einer Konferenz abbilden kann. Es soll beispielsweise möglich sein, alle verfügbaren Konferenz-Sessions in Erfahrung zu bringen oder aber Detailinformationen über Sprecher und Referenten abzurufen. Hierfür gilt es zunächst das Herzstück, den Kontrakt des Services zu entwerfen. Alles andere sind Details. Ob HTTP oder TCP genutzt werden, wird zum Zeitpunkt des Deployments entschieden. Andere Aspekte wie Transaktionen, Sicherheit oder zuverlässiges Messaging sind hingegen zweischneidige Schwerter. Sicherheit kann man im Normalfall nicht einfach hinterher, also nach der eigentlichen Design- und Entwicklungsphase hinzufügen, quasi noch als Stempel aufdrücken. Ebenso verhält es sich in manchen Situationen mit Transaktionen. Daher gehören gewisse Aspekte durchaus mit in den Kontrakt. In diesem Beitrag wollen wir uns jedoch auf die Grundlagen und somit zunächst auf das Wesentliche konzentrieren.

Weiter oben wurden die drei notwendigen Schritte aufgezeichnet, um eine Service-Schnittstelle zu entwerfen. Der

erste Schritt, die Modellierung der Daten, wird in der WCF-Welt mit dem Data Contract erledigt. Besonders einfach kann man sich den Data Contract als optimale Umsetzung und Implementierung des Data Transfer Object Patterns [8] vorstellen. In WCF werden hierfür .NET-Klassen verwendet, die aber eigentlich nicht mehr darstellen als simple Datenstrukturen. Simpel deswegen, weil sich keine oder kaum Logik in diesen Klassen befindet. Datenklassen als Data Contract sind, mit Verlaub, dumme Datencontainer.

Im Namensraum *System.Runtime.Serialization*, respektive *System.Runtime.Serialization.dll* in der neuen Assembly, findet der geneigte Entwickler das Attribut *DataContractAttribute*, also *[DataContract]* für C#- beziehungsweise *<DataContract>* für VB-Programmierer. Mit *[DataContract]* weist man die Serialisierungsinstanz von WCF an, dass es sich hier um einen bestimmten Typen handelt, der eine spezielle Behandlung erfordert. Der Standardserialisierer für WCF ist der so genannte *XmlFormatter*. Wo bei klassischen ASMX Web Services der *XmlSerializer* und bei .NET-Remoting-basierten Anwendungen der *SoapFormatter* oder *BinaryFormatter* zum Einsatz kamen, ist nun der *XmlFormatter* für hochperformanten und interoperablen Datenaustausch komplett neu geschrieben worden.

Listing 1 beinhaltet die Definition der für das Beispielszenario notwendigen Daten. Mit *SessionData* soll die Metainformation über einen Konferenzvortrag modelliert werden. Es lohnt sich, den Code etwas genauer zu betrachten.

Auffällig ist zunächst die unterschiedliche Verwendung von .NET-Codestrukturen und den Attributen für WCF. Die öffentliche Klasse *SessionData* besitzt fünf private Datenfelder. Diese privaten Datenfelder werden als öffentliche Eigen-

### Listing 1

#### Data-Transfer-Objekte für die Kommunikation über Service-Grenzen hinweg.

```
[DataContract(
    Name = "ConferenceSessionDetails",
    Namespace = "urn:thinktecture-com:
        conferences:data")]
public class SessionData
{
    [DataMember(Name = "SessionID")]
    private int id;
    [DataMember(Name = "SpeakerName")]
    private string speakerName;
    [DataMember(Name = "SessionTitle")]
    private string sessionTitle;
    [DataMember(Name = "SessionAbstract")]
    private string sessionAbstract;
    [DataMember(Name = "SessionTime")]
    private DateTime sessionTime;

    public int ID
    {
        get { return id; }
        set { id = value; }
    }

    public string SpeakerName
    {
        get { return speakerName; }
        set { speakerName = value; }
    }

    public string SessionTitle
    {
        get { return sessionTitle; }
        set { sessionTitle = value; }
    }

    public string SessionAbstract
    {
        get { return sessionAbstract; }
        set { sessionAbstract = value; }
    }

    public DateTime SessionTime
    {
        get { return sessionTime; }
        set { sessionTime = value; }
    }
}
```

schaften der Klasse von außen zugreifbar gemacht. Da es sich um schlichte Datencontainer handelt, soll in *Property-Set* und *Property-Get* allenfalls einfache Validierungslogik implementiert werden, aber kein Code, der als echte Geschäftslogik erscheinen könnte.

Die öffentliche Sicht der Daten im Sinne von serviceorientierter Programmie-

## Listing 2

### Definition der Nachrichten als spezieller Data Contract zum Austausch zwischen Services und Konsumenten.

```
[DataContract(
    Name = "SpeakerRequestMessage",
    Namespace = "urn:thinktecture-com:conferences:messages")]
public class SpeakerRequest
{
    [DataMember("SpeakerID")]
    private int speakerId;

    public int SpeakerId
    {
        get { return speakerId; }
        set { speakerId = value; }
    }
}

[DataContract(
    Name = "SpeakerResultMessage",
    Namespace = "urn:thinktecture-com:conferences:messages")]
public class SpeakerResponse
{
    [DataMember("SpeakerName")]
    private string speakerName;

    public string SpeakerName
    {
        get { return speakerName; }
        set { speakerName = value; }
    }
}

[DataContract(
    Name = "SessionRequestMessage",
    Namespace = "urn:thinktecture-com:conferences:messages")]
public class SessionRequest
{
    [DataMember("SessionID")]
    private int sessionId;

    public int SessionId
    {
        get { return sessionId; }
        set { sessionId = value; }
    }
}

[DataContract(
    Name = "SessionResultMessage",
    Namespace = "urn:thinktecture-com:conferences:messages")]
public class SessionResponse
{
    [DataMember("SessionInformation")]
    private SessionData session;

    public SessionData Session
    {
        get { return session; }
        set { session = value; }
    }
}
```

ung erhält man durch die Verwendung von speziellen Attributen. Wie bereits erwähnt, ist *[DataContract]* das wichtigste Attribut in diesem Kontext. *[DataContract]* besitzt die Eigenschaften *Name* und *Namespace*. Doch damit nicht genug: Das Konzept von *DataContract* verlangt das explizite Bekanntmachen der Daten, die nach außen gestellt werden sollen – ganz im Gegensatz beispielsweise zum *XmlSerializer*. Daher ist noch das Attribut *[DataMember]* erforderlich. Über *[DataMember]* wird angezeigt, dass exakt dieses zugehörige Datenfeld als Datum öffentlich gemacht werden soll. Daher wurden sämtliche *DataMember*-Attribute in Listing 1 auch auf die privaten Felder und nicht auf die öffentlichen Eigenschaften der Klasse angewendet. Damit auch wirklich klar wird, dass die interne, mit .NET verbundene Darstellung nicht identisch ist mit der äußeren Sicht der Daten, wurden sowohl der Datenstruktur selbst als auch den einzelnen Datenelementen eindeutige Namen gegeben, die sich von den .NET-Bezeichnern klar unterscheiden. Damit ist der erste Schritt erfolgreich abgeschlos-

sen. Für den nächsten Schritt gilt es, sich Gedanken zu machen, wie die soeben erhaltenen Daten in Nachrichten zu verpacken sind, um sie später in einer oder mehreren Service-Schnittstellen verwenden zu können. Wie bereits weiter oben angedeutet, sollen die Nachrichtenstrukturen ebenfalls mithilfe des *DataContract*-Konstrukts modelliert werden. Dafür gilt es zu überlegen, welche Datenfragmente in den Nachrichten zwischen Endpunkten hin und her wandern sollen. Wenn man möchte, kann man Nachrichten als Verpackungsmechanismus für Daten sehen. Ein Blick auf Listing 2 lässt erkennen, dass keine Magie hinter der Gestaltung von Nachrichten steckt. Die Nachrichtenklassen *SessionRequest*, *SpeakerRequest* und *SpeakerResponse* tragen beispielsweise jeweils einfache Datentypen wie *String* oder *Integer*. *SessionResponse* hingegen überträgt eine konkrete Ausprägung des zuvor modellierten Datenkontraktes *SessionData*. Analog zur Erstellung des *DataContracts* weiter oben wird auch hier für die Nachrichten eine Trennung von Innen- und Außensicht ex-

plizit durch die Nutzung von *Name* und *Namespace* hervorgehoben. Ein wichtiges Detail ist hierbei, dass sich die Daten in einem anderen XML-Namensraum befinden als die Nachrichten. Denn die Daten könnten ja durchaus in anderen Nachrichten und eventuell in ganz anderen Services eingesetzt werden.

So weit zu den Grundlagen von *[DataContract]*. Das anfangs erwähnte Konstrukt des Message Contracts, manifestiert über die Nutzung des Attributs *[MessageContract]* von WCF, wird in diesem Beitrag nicht detailliert betrachtet. Wie schon angemerkt, ist es in vielen Situationen nicht wirklich brauchbar. Mit ihm kann man die Struktur der Nachricht exakt festlegen, kann also genau bestimmen, welche Daten in den Header und welche in den Body wandern. Die häufigsten Situationen jedoch, in denen Header benötigt werden, findet man im Bereich von Infrastrukturanforderungen. So möchte man etwa Authentifizierungsinformationen oder Transaktions-IDs in Header-Feldern zwischen Kommunikationspartnern übertragen. Dafür bietet

WCF aber vollständige Unterstützung in Form von öffentlichen Standards. Spezifikationen wie *WS-Security* oder *WS-AtomicTransaction* bedienen sich intensiv des Konzepts der Übertragung von Out-of-Band-Informationen über Nachrichten-Header, sodass man sich häufig darüber gar keine Gedanken mehr machen muss.

### Service Contract

Bislang wurden im Konferenz-Service-Beispiel die Daten und die Nachrichten entworfen, bleibt also noch die eigentliche Schnittstelle festzulegen. Dieser vorerst letzte Schritt umfasst das Identifizieren und Festlegen der Operationen und somit der Nachrichtenaustauschmuster in der Schnittstelle. Dabei hat ein Designer unterschiedliche Möglichkeiten, wie sich eine Operation im Sinne des Kommunikationsverhaltens präsentiert. WCF bietet in der ersten Version folgende drei grundlegende Kommunikationsmuster:

- Einweg (*OneWay*),
- Anfrage/Antwort (*Request/Reply*),
- Duplex.

Besonders vertraut wirkt *Request/Reply*, nicht nur wegen der Nähe zu *Request/Response*, das vom RPC bekannt ist. Doch der eigentliche Kern von Messaging ist die *OneWay*-Message. Obwohl üblich und häufig in Gebrauch, ist der Begriff Einwegnachricht im Grunde genommen falsch. Denn Nachrichten gehen immer nur in eine Richtung. Gemeint sind Einwegoperationen, also Operationen, bei denen es nur eine Nachricht vom Sender zum Empfänger gibt. In einer Messaging-

Welt ist es eigentlich auch nicht mehr zeitgemäß, von Client und Service zu sprechen, jedoch hat sich Microsoft auf genau diese Nomenklatur für WCF festgelegt. Viel sinnvoller und passender sind die Begriffe Sender und Empfänger. Denn spätestens mit der dritten Art der von WCF unterstützten Kommunikation verwischen die Grenzen zwischen Client und Service vollends, bei der ja auch immer irgendwie eine Art Abhängigkeit oder Hierarchie suggeriert wird. *Duplex* heißt, dass alle beteiligten Kommunikationspartner gleichberechtigt sind und sich gegenseitig Nachrichten gemäß ihrer spezifizierten Schnittstellen zusenden können. Ein konkretes Beispiel hierzu wird weiter unten vorgestellt.

Zurück zum Code. Nun gilt es, die Schnittstelle des Konferenz-Services zu modellieren. Nur nebenbei soll hier erwähnt werden, dass auch mit WCF schemabasiertes Contract First möglich ist. Man muss nicht immer über .NET Code den Weg zu den Schnittstellen-Metadaten begehen.

Damit man als Designer in WCF Services in .NET notieren kann, steht die Attributklasse *[ServiceContract]* zur Verfügung. Ähnlich wie bei *[DataContract]* sind für einfache, also herkömmliche *Request/Reply*-Schnittstellen meist nur die Eigenschaften *Name* und *Namespace* von Bedeutung.

Listing 3 zeigt eine solche Variante des Konferenz-Service-Interfaces. In der .NET-Welt wird ein Service Contract als .NET Interface notiert. Seltsamerweise hat das WCF-Team bei Microsoft es auch erlaubt, direkt Klassen als Schnittstellen für einen Service zu verwenden, was aber

nicht nur beim Lesen äußerst seltsam anmutet.

Die .NET-Notation der Schnittstelle genügt den in .NET üblichen Namenskonventionen, etwa *IConferenceService*. Die Methoden im .NET-Code verwenden als Eingabe- und Ausgabeparameter die zuvor entworfenen Nachrichtendatenstrukturen, zum Beispiel *SessionRequest* und *SessionResponse*. Damit nun die externe Sicht auf die Schnittstelle von internen Änderungen weitestgehend unberührt bleiben kann – etwa bei der Umbenennung von Typen – wird die Schnittstellenbeschreibung in einem eigenen XML-Namespace mit einem eigenen Namen gesetzt. Andererseits erhalten auch die Operationen in der Service-Schnittstelle eindeutige Namen. Dies geschieht unter Verwendung eines weiteren .NET-Attributs namens *[OperationContract]*. Denn auch hier gilt wieder analog zum *[DataContract]*: Es wird explizit entschieden, welche Operation öffentlich gemacht wird und welche nicht. Eine Methode muss über das Attribut *[OperationContract]* nach außen gestellt werden. Diese Vorgehensweise wird als Opt-In-Modell bezeichnet.

Bisher sieht das Programmiermodell aus wie ASMX. Jetzt soll ein Bogen geschlagen werden hin zu Einwegoperationen und weiter zu Duplex. Damit eine Operation als Einwegoperation veröffentlicht wird, nutzt man die Eigenschaft *IsOneWay* im *OperationContract*-Attribut. Dies hat zur Folge, dass diese Operation eine eingehende Nachricht erwartet, aber keine Antwort verschicken wird. Als Voraussetzung darf eine in .NET-Code formulierte Einwegoperation keinen Rückgabewert und keine *ref*- oder *out*-Parameter besitzen. Und *OneWay* bedeutet, dass gemäß dem Prinzip „Fire and Forget“ gehandelt wird. Dies heißt insbesondere, dass keine Fehler an den Aufrufenden übermittelt werden können. Ferner lassen sich aus Schnittstellen mit Einwegoperationen prinzipiell, unabhängig von WCF, sehr flexible und wirklich entkoppelte Kommunikationsszenarien umsetzen.

Auch mit bisherigen Technologien konnte man Duplex-Kommunikation implementieren, etwa über Microsoft Message Queuing (MSMQ) und *System.Messaging*. Doch ist MSMQ nur mäßig interoperabel. Mit WCF wird die Duplex-Kommunikation befördert in den Status eines der drei wichtigen und zentralen Kommunikationsmuster. Daher lohnt sich

### Listing 3

#### Schnittstelle eines einfachen Services für Konferenzdaten.

```
[ServiceContract(
    Name = "ConferenceService",
    Namespace = "urn:thinkecture-com:conferences:interfaces")]
public interface IConferenceService
{
    [OperationContract(
        Name = "ListAllSpeakers")
    SpeakerResponse RetrieveSpeaker(SpeakerRequest
        requestMessage);

    [OperationContract(
        Name = "ListSession")
    SessionResponse RetrieveSessionDetails(SessionRequest
        requestMessage);
}
```

## Listing 4

### Zwei zusammengehörige Schnittstellen zur expliziten Duplex-Kommunikation.

```
[ServiceContract(
    Name = "ConferenceDuplexService",
    Namespace = "urn:thinktecture-com:
    conferences:interfaces",
    CallbackContract = typeof(IConferenceServiceResults))]
public interface IConferenceDuplexService
{
    [OperationContract(IsOneWay = true)]
    void RetrieveSpeaker(SpeakerRequest requestMessage);

    [OperationContract(IsOneWay = true)]
    void RetrieveSessionDetails(SessionRequest
        requestMessage);
}

[ServiceContract(
    Name = "ConferenceDuplexServiceCallback",
    Namespace = "urn:thinktecture-com:conferences:interfaces")]
public interface IConferenceServiceResults
{
    [OperationContract(IsOneWay = true)]
    void RetrieveSpeakerResult(SpeakerResponse
        resultMessage);

    [OperationContract(IsOneWay = true)]
    void RetrieveSessionDetailsResult(SessionResponse
        resultMessage);
}
```

hier ein Blick auf Duplex-Kontrakte und wie sie umgesetzt werden.

Duplex bedeutet technisch gesehen nichts anderes, als dass der ursprüngliche Empfänger – also im klassischen Jargon der Service – dem ursprünglichen Sender – dem Client – eine Nachricht schicken kann. In unzähligen Diskussionen und Entwürfen haben Experten versucht, dies in ASMX nachzubilden beziehungsweise eine vom Aufwand her vertretbare Lösung zu ersinnen. Mit WCF ist Microsoft dies nun gelungen. Die Grundidee ist dabei einfach: einer Service-Schnittstelle wird eine Rückrufschnittstelle zugeordnet. Als logische Konsequenz dieser Idee müssen alle Operationen in den beteiligten Schnittstellen Einwegoperationen sein. Ein prima Beispiel, bei dem Einwegoperationen zu einem neuen Kommunikationsmuster zusammengesetzt werden.

Mithilfe von WCF-Duplex-Kontrakten lässt sich die Kommunikation zwischen einem Client und dem Konferenz-Service von der zeitlichen Abhängigkeit her entkoppeln. Listing 4 zeigt zwei Interfaces in C#. Diese beiden Interfaces *ConferenceDuplexService* respektive *ConferenceDuplexServiceCallback* sind WCF-konforme Service Contracts. Es fällt auf, dass beide Interfaces ausschließlich aus Einwegoperationen bestehen, also die beste Voraussetzung bieten für eine Duplex-Kommunikation. Nur die Korrelierung der beiden Schnittstellen fehlt noch. Dies geschieht über die Eigenschaft *CallbackContract* der Klasse *ServiceContractAttribute*. Über

```
CallbackContract = typeof(IConferenceService-
Results)
```

wird exakt festgelegt, dass die Schnittstelle *ConferenceDuplexServiceCallback*

ein Rückrufkontrakt zur Schnittstelle *ConferenceServiceDuplex* ist. Dies ist tatsächlich erst einmal alles, was man über den Duplex-Datenaustausch mit WCF wissen muss. Im Zuge des Ausbaus des Beispiels in künftigen Ausgaben der *dotnetpro* wird auch auf die exakte Implementierung eingegangen, vor allem auf Client-Seite.

### Services implementieren Kontrakte

Zu guter Letzt müssen sämtliche Kontrakte auch verwendet und Service-Kontrakte implementiert werden. Ein WCF Service ist eine .NET-Klasse, die einen oder mehrere Service Contracts umsetzt. Das folgende Codefragment zeigt eine Service-Implementierung, die beide weiter oben eingeführten Kontrakte implementiert.

```
public class ConferenceServiceImplementation :
    IConferenceService,
    IConferenceDuplexService
```

Ein Service muss also alle nachrichtenbasierten Operationen eines Schnittstellenkontraktes umsetzen. Der Service muss beim Aufruf einer Operation, die eine Operation auf dem Callback-Interface aufrufen soll, wissen, wohin diese Rückrufoperation gerichtet sein soll. Es muss also in der Service-Implementierung bekannt sein, welcher aufrufende Konsument diese Rückrufschnittstelle implementiert und was die Adresse des Kommunikationsendpunktes des Aufrufenden ist.

Dies hat zur Folge, dass der ursprüngliche Client auch einen Endpunkt anbieten muss. Wie das für die unterschiedlichen Transportprotokolle genau geschieht, wird in einem zukünftigen Artikel erklärt. Hier wird angenommen, dass der Client einfache Möglichkeiten hat, diesen Endpunkt anzubieten. Damit die Service-Implementierung in einem Duplex-Szenario auch mit mehreren Clients korrekt umgehen kann, muss für von einem Client gesendete Nachrichten in ei-

### Nachrichten versus Methodenaufrufe

Es wurde bereits in früheren Artikeln zu Indigo/WCF angedeutet: WCF ist eine nachrichtenorientierte Kommunikationsplattform. Dies bezieht sich vor allem auf den Umstand, dass intern immer mit dem Artefakt der Nachricht gearbeitet wird und alles, was über die Leitung verschickt wird, eine Nachricht ist und keine Objekte im .NET-Sinne im Spiel sind. Jedoch hat sich Microsoft entschlossen, das Programmiermodell sowohl nachrichtenorientiert zu gestalten als auch die vertrauten Methodenaufrufe nachzubilden. Die WCF-Artikel der *dotnetpro* werden ausschließlich mit der Grundidee der Nachricht hantieren und versuchen, die alten – und meistens schlechten – Gewohnheiten von Remote Procedure Classes (RPC) außen vor zu lassen. Eine etwas tiefer gehende Diskussion zum Thema RPC, Nachrichtenorientierung und WCF finden Sie online unter [4].

### Listing 5

#### Initialisierung der Callback-Funktionalität in der Service-Implementierung.

```
public class ConferenceServiceImplementation:
    IConferenceService,
    IConferenceDuplexService
{
    IConferenceServiceResults callback = null;

    public ConferenceServiceImplementation()
    {
        try
        {
            callback =
                OperationContext.Current.
                GetCallbackChannel<
                    IConferenceServiceResults>();
        }
        catch (Exception) { }
    }
}
```

ner Duplex-Kommunikation eine Sitzung, also eine Session, geöffnet werden. Da WCF nicht zuletzt unterhalb des Programmiermodells ausschließlich mit Nachrichten arbeitet, werden diese Informationen auch in eben diesen Nachrichten gehalten.

Nachrichten in WCF sind SOAP-Nachrichten. Dies bedeutet übrigens nicht automatisch, dass jede versendete Nachricht in jeder WCF-Anwendung XML und spitze Klammern bedeutet – dieser Irrtum wurde bereits in [5] aus der Welt geschafft. In diesen SOAP-Nachrichten wird als Header eine Session ID mitgeschickt. Damit eine Chance auf Plattform übergreifende Interoperabilität besteht, hat sich das

WCF-Team von Microsoft entschieden, die Duplex-Kommunikation auf Basis der öffentlichen Spezifikation WS-Reliable-Messaging zu implementieren. Dies bedeutet, dass bestimmte Teilfunktionalitäten von WS-ReliableMessaging genutzt werden, um eine Session auf der Nachrichtenebene zu initiieren, zu betreiben und wieder zu beenden. Damit aber der Code in der Service-Implementierung tatsächlich funktionieren kann, werden nun über den *OperationContext* alle notwendigen Informationen über den Rückruf ausgelesen. Der *OperationContext* ist eine Sammlung von Eigenschaften, die aus der Nachricht befüllt werden, vornehmlich aus den Headern der Nachricht. Damit kann man mit dem Code aus Listing 5 den Service vorbereiten, damit die Implementierungen der tatsächlichen Operationen die *callback*-Instanz benutzen können, um die Gegenseite asynchron in einer Duplex-Manier zu benachrichtigen.

Existiert erst einmal ein Callback-Kontext, fällt die Umsetzung der Operationen sehr leicht. In Listing 6 wird in der Duplex-Variante einfach die synchrone Ausprägung aus dem *Request/Reply*-Interface aufgerufen. Sender und Empfänger sind somit zeitlich komplett voneinander entkoppelt. Die Abarbeitung der Operation auf der Service-Seite dürfte auch mehrere Minuten dauern. Der Client wäre nicht blockiert und würde über einen Rückruf initiiert vom Service das Ergebnis der Operation geliefert bekommen.

Wie man auf der Konsumentenseite Duplex Contracts implementiert, ist teilweise trickreich. Vor allem wenn man mit Windows Forms Clients arbeitet, kann man sich mit lokalen Delegates und unterschiedlichen Threads schnell in die Breddouille bringen. Diese Feinheiten und

### Listing 6

#### Exemplarische Implementierung einer Callback-Operation im Service.

```
void IConferenceDuplexService.RetrieveSpeaker(SpeakerRequest requestMessage)
{
    [...]

    void IConferenceDuplexService.RetrieveSessionDetails(SessionRequest requestMessage)
    {
        if (callback != null)
        {
            SessionResponse result = RetrieveSessionDetails(requestMessage);
            callback.RetrieveSessionDetailsResult(result);
        }
    }
}
```

### WCF

Die Codenamen für Indigo und Avalon wurden pünktlich für die Veröffentlichung der Beta 1 des WinFX bekannt gegeben. Avalon heißt jetzt Windows Presentation Foundation und Indigo wird künftig nur noch Windows Communication Foundation oder kurz WCF genannt.

Eigenheiten werden in kommenden Ausgaben noch detailliert besprochen.

### Fazit

Die Windows Communication Foundation, ehemals Indigo, soll Ende des Jahres 2005 als Beta 2 vorliegen. Spätestens dann sollte sich jeder Entwickler, der heute mit ASMX, WSE, Remoting, Enterprise Services oder MSMQ hantiert, einen Blick darauf werfen. Dieser Artikel hat das Konzept der Kontrakte zur expliziten Beschreibung von Schnittstellen unter die Lupe genommen. Kontrakte sind das A und O von Services, und somit auch von WCF. |||||

- [1] Ralf Westphal, Am Anfang war der Vertrag, Contract First Design und Microkernel-Frameworks, dotnetpro 6/2005, Seite 20 ff.
- [2] Ralf Westphal, Spicken nicht erlaubt, Contract First Design und Microkernel-Frameworks, dotnetpro 9/2005, Seite 124 ff.
- [3] Wolfgang Pleus, Dienste zusammenklicken, BizTalk Server 2004 als Web-Service-Plattform, dotnetpro 9/2005, Seite 134 ff.
- [4] thinktecture-Gespräche, Indigo/WCF, [www.thinktecture.com/Gespraech/Indigo\\_CW\\_RW.htm](http://www.thinktecture.com/Gespraech/Indigo_CW_RW.htm)
- [5] Christian Weyer, Blaues Service-Wunder, Indigo Beta 1, dotnetpro 7/8/2005, Seite 130 ff.
- [6] [research.sun.com/techrep/1994/abstract-29.html](http://research.sun.com/techrep/1994/abstract-29.html)
- [7] Christian Weyer, Introducing Contract-First, [www.code-magazine.com/Article.aspx?quickid=0507061](http://www.code-magazine.com/Article.aspx?quickid=0507061)
- [8] Martin Fowler, Data Transfer Object Pattern, [www.martinfowler.com/eaCatalog/dataTransferObject.html](http://www.martinfowler.com/eaCatalog/dataTransferObject.html)
- [9] WinFX SDK (mit WCF als Bestandteil) auf MSDN, [winfx.msdn.microsoft.com/library/](http://winfx.msdn.microsoft.com/library/)