Eén van de voordelen van Microsoft .NET is dat het verschillende technologieën onder één dak brengt. Gezien de verscheidenheid van de verschillende onderdelen is dat op zich een geweldige prestatie. Twee onderdelen van .NET, XML en object oriëntatie (OO) werken elkaar soms echter tegen op een punt waar het makkelijk over het hoofd gezien wordt: webservices. Webservices ontwikkelen in .NET is eenvoudig, maar uit het directe zicht van de ontwikkelaar doet het raamwerk heel veel moeite alle data foutloos tussen gebruiker en webservice te transporteren. Wanneer die data eigen objecten bevatten, werken OO en XML soms slecht samen.

.NET webservices, een object te ver?

Complexe functies leggen beperkingen bloot

Twee zaken die problemen kunnen opleveren zijn de XML-transformatie van een object en de classificatie van een webservice proxy. Het .NET-raamwerk levert voor simpele webservicefuncties alle code aan die nodig is om met HTTP-GET, HTTP-POST of HTTP-SOAP de functie aan te roepen. Sterker nog, met IE kan een testpagina worden opgeroepen waarmee via het HTTP-GET protocol getest kan worden. Voor simpele functies werkt het allemaal inderdaad prachtig, de 'Hello World' variant staat al snel op de browser. Voor complexe functies is de ondersteuning echter aan wat beperkingen onderhevig. Laten we, voordat we ingaan op de beperkingen, beginnen met de vraag: wat is een complexe webservicefunctie in deze context? We kunnen de volgende definitie hanteren:

Een complexe webservicefunctie is een webservicefunctie die een instantie van een eigen class als parameter gebruikt, of een instantie van een eigen class retourneert.

PERSISTEREN VAN OBJECT In een complexe webservicefunctie vervallen de protocollen HTTP-GET en HTTP-POST, omdat die protocollen *User-Defined Types* (UDT's) niet aankunnen. Met SOAP kan dat wel, zodat dat protocol overblijft. Een UDT wordt in SOAP in een WSDL beschreven, een beschrijving die veel weg heeft van een XML schema definitie. De twee .NET-talen die het meest gebruikt worden, VB.NET en C#, ondersteunen het OO-paradigma. De voorbeelden zijn in C#, maar voor iedereen te begrijpen. Enige bekendheid met OO strekt tot voordeel.

Een object bestaat uit variabelen en bewerkingen. De data geven de toestand weer, de bewerkingen (methoden) en de mogelijke veranderingen. De data zijn verborgen, zodat deze alleen aangepast kan worden door de methoden aan te roepen (encapsulatie). De toestand en de methoden zijn te scheiden, onafhankelijk van elkaar te transporteren en weer samen te voegen tot het originele object. Dit principe wordt vaak gebruikt om objecten in een database te bewaren. Dit noemen we ook wel het persisteren van een object.

> Wanneer de data eigen objecten bevatten, werken OO en XML soms slecht samen

OBJECT ALS SOM DER DELEN Allereerst een voorbeeld om duidelijk te maken dat de overdracht in XML wezenlijk verschilt van het OO-gedachtegoed. Voor een complexe functie is een UDT nodig (zie Listing 1).

```
public class Persoon
{
        public Persoon ()
{
this.m_Name = "";
this.m_Age  = 0;
}
        public string Naam
        {
              get
              {
                    retrurn this.m_Name;
              }
              set
              {
                    this.m_Name = value;
              }
        }
        public int Leeftijd
        {
              get
              {
                    return this.m_Age;
              }
              set
              {
                    this.m_Age = value;
              }
        }
        private string m_Name;
        private int m_Age;
}
```

Listing 1

Een functie die een object van deze class retourneert is weergegeven in Listing 2.

De UDT is geen standaard XML-datatype en wordt daarom vertaald naar XML-structuur. Deze omzetting wordt geheel door het .NET-raamwerk verzorgd, de uiteindelijke XML zoals de functie die naar de aanroeper stuurt, is te zien in Listing 3.

Deze vertaling wordt verzorgd door een object van de class XmlSerializer. Uit listing 3 valt af te leiden hoe deze werkt. Alle publieke elementen die zowel leesbaar

```
[WebMethod]
public Persoon VindPersoon (string naam)
{
        Persoon gevonden = null;
        ........ het vinden .......
        return gevonden;
}
```

Listing 2

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/en
velope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <soap:Body>
          <VindPersoonResponse
          xmlns="http://tempuri.org/">
           <VindPersoonResult>
            <Naam>gevonden</Naam>
           <Leeftijd>25</Leeftijd>
          </VindPersoonResult>
         </VindPersoonResponse>
        </soap:Body>
</soap:Envelope>
```

Listing 3

als schrijfbaar zijn worden XML-elementen. De .NET documentatie van de class XmlSerializer beschrijft het proces in detail. Tevens valt op dat onze encapsulatie verdwenen is. De toestand van het object staat niet meer in m_Name en m_Age, maar in Naam en Leeftijd. Voor XML telt niet de toestand van een object, maar de waarneembare toestand. Voor vele classes is dat een verschil. Dit gegeven vormt de bron van de problemen. Wanneer we deze functie aanroepen vanuit een ander programma, moet het proces omgekeerd worden. XML wordt een object.

WAARNEEMBAAR EN WERKELIJK Door in Visual Studio .NET een webservice referentie op te nemen, wordt automatisch een webservice proxy aangemaakt. Die proxy zorgt ervoor dat de functie er voor ons uitziet als een reguliere functie. De proxy zorgt ook voor juiste protocol afhandeling. Met WSDL.EXE kan ook een proxy aangemaakt worden, waarbij dan wat meer opties zelf in te stellen zijn. Voor beide geldt: op basis van de WSDL worden de functies en data definities in code omgezet. Zo ook de persoon class, die er nu als volgt uitziet:

```
public class Persoon {

        /// <remarks/>
        public string Naam;

        /// <remarks/>
        public int Leeftijd;
}
```

Als we dat vergelijken met het origineel, dan is de ene persoon duidelijk de andere niet. In dit voorbeeld is

de toestand van een class transparant. De omzetting is één op één, zodat er afgezien van het verlies van encapsulatie niet zo heel veel verandert. Nu een voorbeeld van een class waarbij er wel een verschil tussen de waarneembare en werkelijke toestand is:

```
public class MyDateTime
        {
        public MyDateTime () {}
        public DateTime UTC
        {
              get
              {
                    return this.m_utc;
              }
              set
              {
                    this.m_utc = value;
              }
        }
        public DateTime Local
        {
              get
              {
                    return this.m_utc.Add
                    (this.m_localdelta);
              }
              set
              {
                    this.m_utc = value.ToUni-
                    versalTime();
                    this.m_localdelta =
                    this.m_utc.Subtract(value)
                    ;
              }
        }
        public int MinutesToUTC
        {
              get
              {
                    return this.m_local-
                    delta.Minutes;
              }
              set
              {
                    this.m_localdelta = new
                    TimeSpan(0,0,value);
              }
        }
              private DateTime m_utc;
              private TimeSpan m_localdelta;
        }
```

In dit object leidt het wijzigen van één kenmerk tot een verandering in minstens één ander kenmerk. De XML die ontstaat tijdens gebruik in een webservice is te zien in Listing 4.

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/en
velope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <soap:Body>
         <GeefTijdResponse xmlns="http://tem-
         puri.org/">
          <GeefTijdResult>
           <UTC>2003-03-
           26T15:12:54.4775000+01:00</UTC>
           <Local>2003-03-
           26T14:12:54.4775000+01:00</Local>
           <MinutesToUTC>-60</MinutesToUTC>
          </GeefTijdResult>
         </GeefTijdResponse>
         </soap:Body>
</soap:Envelope>
```

Listing 4

Het object had één datum/tijd veld, de XML heeft er twee. In dit voorbeeld wijkt dus de waarneembare toestand af van de werkelijke toestand. Bij complexe objecten doet zich dit met enige regelmaat voor. De generatie van de proxy code kijkt zuiver naar de WSDL en dat levert de volgende code op:

```
public class MyDateTime {

        /// <remarks/>
        public System.DateTime UTC;

        /// <remarks/>
        public System.DateTime Local;

        /// <remarks/>
        public int MinutesToUTC;

}
```

Nu is het gedrag van het gegenereerde object wezenlijk anders dan dat van het origineel. Daar waar dit problemen oplevert, moet een situatie gecreëerd worden, waarin een 1 op 1 vertaling optreedt. Dit kan gerealiseerd worden door bijvoorbeeld eliminatie van gedrag, delegatie, het herstellen van de toestand, eliminatie van data of het delen van de originele class code.

Elimineer gedrag

Soms is het mogelijk de class zodanig te definiëren dat de gedragskenmerken die voor problemen zorgen niet meer nodig zijn. Zo kan in het bovenstaande voorbeeld één van de twee datum/tijden helemaal ver-

dwijnen. De verantwoordelijkheid van de class vermindert. Het verlies van gedragselementen kan vervelend zijn, maar is onvermijdelijk als we alleen de toestand opsturen. Dit is een optie die vaak goed werkt met 'read-only' of 'return-only' objecten waarbij 'luxe' gedrag verdwijnt.

Delegeren

Een optie die doorgaans goed werkt, is het verplaatsen van gedrag. We ontwerpen nieuwe classes die elk een deel van het originele gedrag implementeren. De originele class bundelt de objecten en regelt de aansturing. Deze methode werkt vaak goed omdat het de oplossing zoekt op het vlak waar XML sterk is: hiërarchische datastructuur. Waakzaamheid blijft geboden, het model mag niet vollopen met een hele batterij kleine vrijwel nietszeggende classes.

> Tot de verbazing van onze ontwikkelaar blijkt de code niet foutloos te compileren

Repareren

Als het object als invoer dient kunnen we ook een methode bedenken die na ontvangst de toestand controleert en herstelt. Hoewel dit in eerste instantie vaak aantrekkelijk lijkt, kent deze optie veel nadelen. Een dergelijke methode is vaak moeilijk te implementeren. Maar belangrijker, het feit dat de methode implementeerbaar is, houdt in dat we de inhoud van de variabelen anders waarderen. Dit duidt er vaak op dat een objectmodel nog verbeterd kan worden.

Elimineren van data

In het gegeven voorbeeld kunnen op eenvoudige wijze één van de twee data worden weggelaten. Als we bijvoorbeeld de lokale tijd van het voorbeeld willen weglaten, dan volstaat het opnemen van een attribuut:

```
 [XmlIgnoreAttribute]
public DateTime Local
{
……
}
```

Hiermee verdwijnt de variabele Local uit listing 7, waardoor de eenduidigheid weer hersteld wordt. Wel moet er naar naamgeving gekeken worden. Nu blijft over de UTC datum/tijd en de *MinutesToUTC* variabele. Zo is het wellicht handiger om die dan MinutesToLocal te noemen en de algoritmen enigszins aan te passen.

Anders dan bij het elimineren van gedrag blijft de class hier ongewijzigd. We dragen nu niet de waarneembare toestand over, maar de werkelijke. Helaas moeten we erop vertrouwen dat de 'ander' in staat is dat weggevallen deel opnieuw op te bouwen. Door het delen van verantwoordelijkheid lopen we het risico dat er twee classes ontstaan met hetzelfde gedrag. In OO-termen is dat een even zware 'overtreding' als het niet encapsuleren van data. Ondanks de risico's kan data-eliminatie vaak toegepast worden, zeker als de bewerking 'elementair' is.

DELEN VAN DE CLASS Een optie die alle problemen lijkt op te lossen is die waarbij we niet alleen de toestand opsturen, maar ook de methoden. Dubbele elementen leveren geen complicaties meer op, omdat het object zelf te vertrouwen is. Nu kan zonder bezwaar volstaan worden met het opsturen van de werkelijke toestand. Toch zijn er ook nadelen; er ontstaan snel afhankelijkheden tussen systemen die op termijn problemen opleveren.

Toch wil ik op deze optie verder ingaan omdat het laat zien hoe .NET-objecten (denk bijvoorbeeld aan een dataset) er wel in slagen ongeschonden de lijn over te komen.

De kunst is om de XmlSerializer de bestaande class te laten gebruiken in plaats van de gegenereerde versie. Dit krijgen we voor elkaar door de namespace waarin de originele class zit op te nemen in de proxycode en de gegenereerde definitie te verwijderen.

Met WDSL.EXE maken we de broncode voor onze nieuwe proxy. Deze broncode wordt dan handmatig aangepast zodat tijdens compilatie de XmlSerializer met de juiste datadefinitie verbonden wordt. De aan te brengen wijzigingen:

- Zorg ervoor dat de namespace van de class die de werkelijke definitie levert beschikbaar is.
- Verwijder alle data-definities die in die namespace beschikbaar zijn.
- Neem de gegenereerde en aangepaste code op in een project waarin aan die DLL gerefereerd wordt.

Wanneer de code vertaald wordt, zal de code niet meer een uit XML afgeleide versie construeren, maar een instantie van de originele class. Omdat het objectmodel hiermee ook aan de consumer kant beschikbaar komt, is dat een aantrekkelijke optie. Dankzij .NET is het zeker dat de DLL ook door alle mede .NET'ers gebruikt kan worden. Voor de niet .NET'ers is dit echter nooit een oplossing.

EEN NAMESPACE MEER Een ander probleem waar veel projecten op stuiten heeft te maken met de manier waarop de ontwikkelomgeving de proxy code genereert. Wanneer een referentie naar een webservice aangemaakt wordt met de VS.NET IDE, wordt de namespace van de webservice proxy als bepaald door standaard project namespace en de service naam. In die namespace wordt de code gegenereerd voor alle data en functies. Gebruik je WSDL.EXE, dan bepaal je zelf de namespace.

Dit scenario is niet waterdicht, zoals blijkt uit het volgende voorbeeld, waarin het 'lek' wordt toegelicht. In het fictieve bedrijf ACME zijn twee afdelingen: de personeelsadministratie en de salarisadministratie. Beide afdelingen willen diensten op het intranet aanbieden en gaan daar natuurlijk .NET webservices voor gebruiken. ACME standaards dwingen het gebruik van de algemene Medewerker class af omdat het wiel in ACME niet tweemaal uitgevonden hoeft te worden.

AANROEPER De personeelsadministratie ontwerpt de webservice *Inlichtingen* met de functie *ZoekMedewerker*. Deze wordt aangestuurd met een personeelsnummer en levert de aanroeper een instantie van de bovengenoemde Medewerker class op. Ondertussen bouwt de salarisadministratie een webservice *SalarisInfo*, met de functie *DatumLaatsteStorting*. Die wordt aangestuurd met een instantie van Medewerker en levert de datum waarop het salaris voor die medewerker het laatst gestort is. De ontwikkelaar ziet de geboden kans en maakt vanuit het project webreferenties aan naar beide services met de uiteindelijke bedoeling de uitkomst van *ZoekMedewerker* te gebruiken voor *DatumLaatsteStorting*. Tot de verbazing van onze ontwikkelaar blijkt deze code niet foutloos te compileren.

In de ACMEClient namespace vinden we de twee services als namespaces terug, dus ACMEClient.Inlichtingen en ACMEClient.SalarisInfo. De ACMEClient.Inlichtingen namespace bevat de classes ZoekMedewerker en Medewerker, de ACMEClient.SalarisInfo namespace bevat de classes DatumLaatsteStorting en Medewerker. Wacht eens even, twee Medewerkers? Zolang deze in andere namespaces zitten levert dit geen fouten op, maar wanneer je iets doet als ACMEClient.SalarisInfo.Medewerker ik = new ACMEClient.Inlichtingen.Medewerker(); ontstaan de problemen die onze ontwikkelaar ook ontdekt heeft. De uitvoer van de personeelsadministratie is niet te gebruiken als invoer voor de salarisadministratie.

DEFINITIE VERWIJDEREN Als alle afdelingen hun functies in één grote ACMEIntraNet service zouden aanbieden, verdwijnt het probleem. Maar dat biedt geen structurele oplossing. Omdat onze ontwikkelaar die wel zoekt, besluit deze vervolgens met WSDL.EXE de proxy's in dezelfde namespace te genereren. Nu vertaalt weer niet alle broncode, maar de foutmelding verschilt. Onze Medewerker class is tweemaal in één namespace gedefinieerd. Wanneer we één definitie verwijderen en opnieuw vertalen werkt alles zoals het hoort. In plaats van WSDL was het ook mogelijk om met reflection een adapter class te ontwikkelen die het object van de ene namespace in de andere overneemt.

CONCLUSIES Het gebruik van eigen classes in .NET webservices is een kwestie van 'eerst denken dan doen'. Door op voorhand alert te zijn op de mogelijke problemen kan met relatief simpele correcties een robuust OO-model ontstaan dat zonder problemen via SOAP met anderen gedeeld kan worden. Ook moet vooraf bedacht worden hoe webservices en webservicefuncties ingedeeld worden. Omdat de IDE de proxy code generatie verbergt, is het verstandig om altijd gebruik te maken van WSDL.EXE. De webservices- en functieindeling, alsmede het gebruik van WSDL.EXE wordt vastgelegd in standaards en richtlijnen zodat er een voor iedereen herkenbare werkwijze ontstaat. Tevens voorkomt dit dat een project vertraging oploopt doordat pas in een latere fase de code alsnog aangepast moet worden. Omdat proxy code van nature redelijk onveranderlijk is, is het handmatig aanpassen niet erg bezwaarlijk. Desgewenst is het aanpassen van de proxy code met een script te automatiseren. De macro's kunnen dan als onderdeel van de tooling ter beschikking gesteld worden, zodat de kwaliteit van de oplossing verder verhoogd wordt.

> Wanneer we één definitie verwijderen en opnieuw vertalen werkt alles zoals het hoort

Chris Widdows is werkzaam als software architect bij Cap Gemini Ernst & Young.