SERVICE STATION

Techniques for Contract-First Development

Aaron Skonnard

In my May 2005 column, I discussed contract-first development and appropriate times to use it (see Service Station: Contract-First Service Development). In this second part, I'm going to focus on some techniques for contract-first development within the ASMX framework. I'll look at various tools that come with the Microsoft® .NET Framework and Visual Studio® .NET, as well as some useful third-party tools. I'll also discuss an interesting hybrid technique that offers a nice compromise between interoperability and productivity given the current tool situation.

Contract-First in .NET Today

If you use a contract-first development model with Visual Studio .NET and the ASMX framework, you'll probably need third-party tools for ease-of-use and overall productivity. With contract-first, you typically go through the development cycle that is shown in **Figure 1** and follow these steps:

1. Define XML messages using XSD
2. Define operations using WSDL
3. Generate ASMX code from WSDL
4. Implement operations
5. Iterate

I'll walk you through each of these steps and explain the different tool options available for assisting you. Then I'll give you a few ideas on simplifying the process.

Author XSD

When using contract-first, you begin by identifying the data exchanged by the service and then model it using XML Schema Definition (XSD). During this step, focus on naming and structuring the data in the messages that interact with the service. When using the document/literal style, the resulting XSD contains all the information about what goes in the SOAP envelopes on the wire. This is often referred to as the service's data contract.

There are a variety of ways to author XSD, starting with opening Notepad and typing. Most developers, however, prefer to use a more sophisticated XSD editor. Visual Studio .NET makes authoring XSD easy through its built-in schema

designer. The designer makes it possible to drag and drop schema constructs from the toolbox onto the design surface, shielding developers from the angle brackets and other complexities of the language. You can always switch between the design view and the XML view and edit within either.

Let's look at an example. You want to design a purchase order (PO) service that will have two operations: Submit and GetPending. Using contract-first development, you create a new XSD file in Visual Studio .NET and then begin authoring the XML messages that will be needed by these operations (see **Figure 2**). In this example, Submit will be a one-way operation so it needs only a single XML element. GetPending needs to be a request/response operation so it will need two elements.

The XSD shown in **Figure 3** illustrates what a sample schema might contain for the messages needed by the PO service. Notice that there are three elements defined: SubmitRequest, GetPendingRequest, and GetPendingResponse. The associated complex type definitions define the structure of these elements.

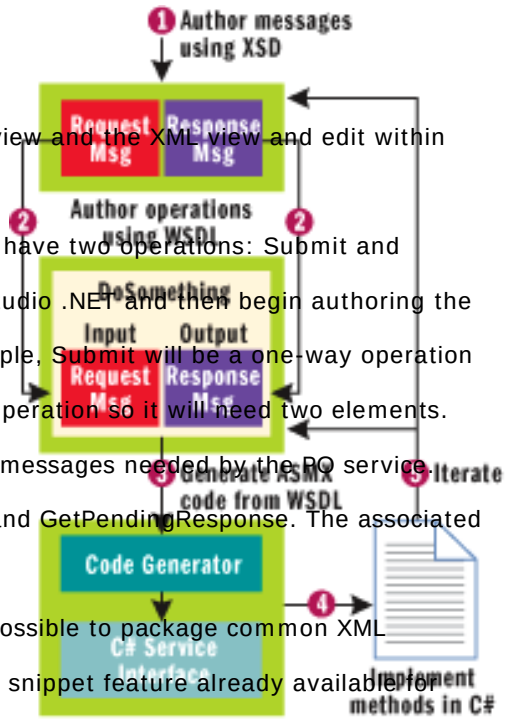

Figure 1 Contract-First Development

Visual Studio 2005 comes with a new XML code snippet feature, which will make it possible to package common XML constructs and tie them to short aliases for automatic expansion (similar to the code snippet feature already available for C#), though as always with beta code, things are subject to change before the final release. Microsoft is planning to ship numerous XSD code snippets and expansions to make hand-authoring XSD more efficient for those who prefer to work directly with angle brackets. This feature alone could greatly increase the number of developers who feel comfortable authoring raw XSD.

In addition to Visual Studio .NET, there are also a variety of third-party tools that can be used to author XSD. Altova's XML Spy is one of the industry favorites that also integrates directly with Visual Studio .NET. Several others can be used standalone. The important thing is to find one that your developers will use.

Assuming a document/literal service, the resulting schema defines the valid XML elements that are used when interacting with the service. What is still not known is how the different elements map to logical operations or how they're correlated with each other (for example, GetPendingResponse follows a GetPendingRequest, and so on). This is where Web Services Description Language (WSDL) comes in.

Starting with XSD helps avoid most interoperability problems. It forces you to define your messages in the language that everyone understands, and that means you avoid using types specific to your environment that don't have well-known schema equivalents (like DataSet or Hashtable). Thinking more about the XML will improve interoperability. Because Visual Studio .NET has a built-in designer for XSD, most developers today have at least tinkered with it. Unfortunately, the next step is a bit harder.

Author WSDL

Once you have your schemas in place, you then define your service's operations and transport bindings using WSDL. A WSDL operation simply defines an exchange of XSD messages. The operations are then grouped into a WSDL interface (or portType). WSDL also defines transport binding details that are needed to transmit messages to the service. WSDL is a fairly thin layer over XSD, but it's a necessary one. It contains all the information required by code generators to produce implementation code on either side of the wire.

Authoring WSDL is the most confounding part of the contract-first model. Developers get frustrated by this step because the language is complex and confusing and there isn't much in the way of decent tool support. Writing WSDL in Notepad is a long shot—it takes a strong understanding of XML namespaces, XSD, SOAP, and of course, the WSDL language itself. Even most

Web services experts have a hard time hand-authoring a bug-free WSDL definition on the fly.

Visual Studio .NET does not provide the XSD designer equivalent for WSDL. So unless you use a third-party tool, you're basically stuck doing it by hand. However, as with XSD, Visual Studio 2005 comes with several canned WSDL code snippets and expansions that will provide a valuable crutch for developers hand-authoring WSDL.

Another way that you can get around hand-authoring WSDL is to prototype the contract with ASMX and then use the generated WSDL as a starting point. You can inspect the generated WSDL, make necessary changes, and iterate until you've produced the contract you desire. This approach requires you to know enough about WSDL to make good decisions and guide the outcome during the process. Many developers take this approach, but it can still cause some shops to fumble.

There are a handful of third-party editors that can be used to author WSDL definitions—examples include Altova's XML Spy and Cape Clear's SOA Editor. Most of these tools make a noble attempt at an intuitive WSDL designer, but they all make the mistake of attempting full coverage of the language. As a result, most tools in this camp end up being hard to use, not because of flaws in the product necessarily, but because WSDL itself is too hard. Because the language is so flexible and complex, there are many decisions to be made while authoring a WSDL definition. What most developers need is a simple wizard model that helps them produce a WS-I compliant definition, avoiding the parts of the language they don't need to know about.

There are a few third-party tools that make this possible. My personal favorite is Thinktecture's free WsContractFirst tool. WsContractFirst integrates with Visual Studio .NET and provides a step-by-step wizard for generating WS-I-compliant WSDL definitions around existing XSD files, letting you avoid editing the WSDL altogether. It's easy to use and can help you produce WSDL without having to get knee-deep in the complexities of the language.

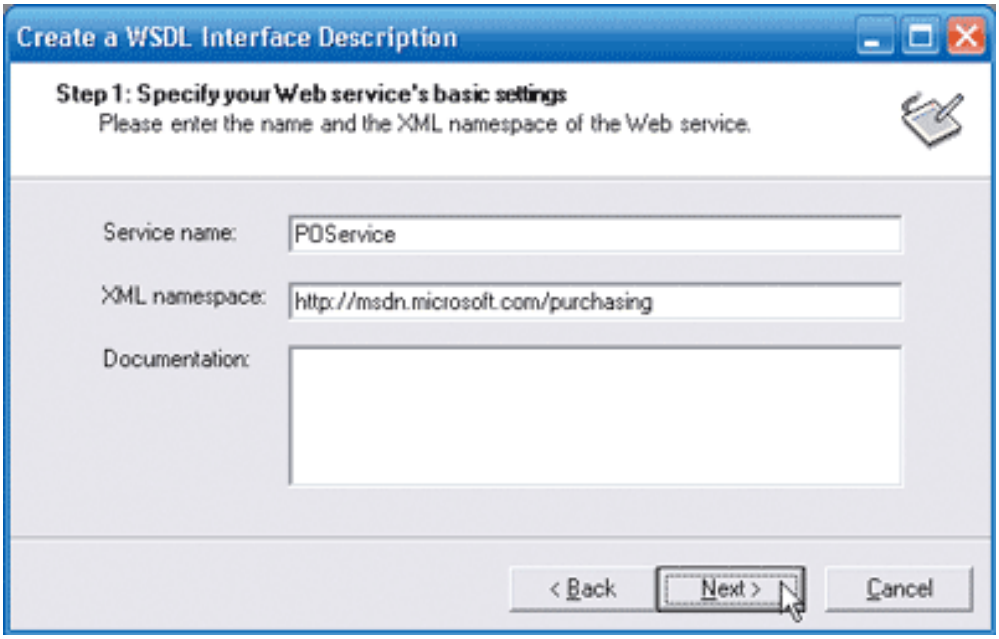

Figure 4 WsContractFirst Step 1

Once you've installed WsContractFirst, you can right-click on any .xsd file in your project and select "Create WSDL Interface Description." Doing so will launch a wizard that allows you to fill in the WSDL details. On the first step, specify the name and namespace of your service (see **Figure 4**). Then you specify the names of the operations your service will contain along with the message exchange pattern (see **Figure 5**). And finally you specify which XSD elements will be used by the operations (see **Figure 6**). Then WsContractFirst generates a WSDL definition and adds it to your project (see **Figure 7**).

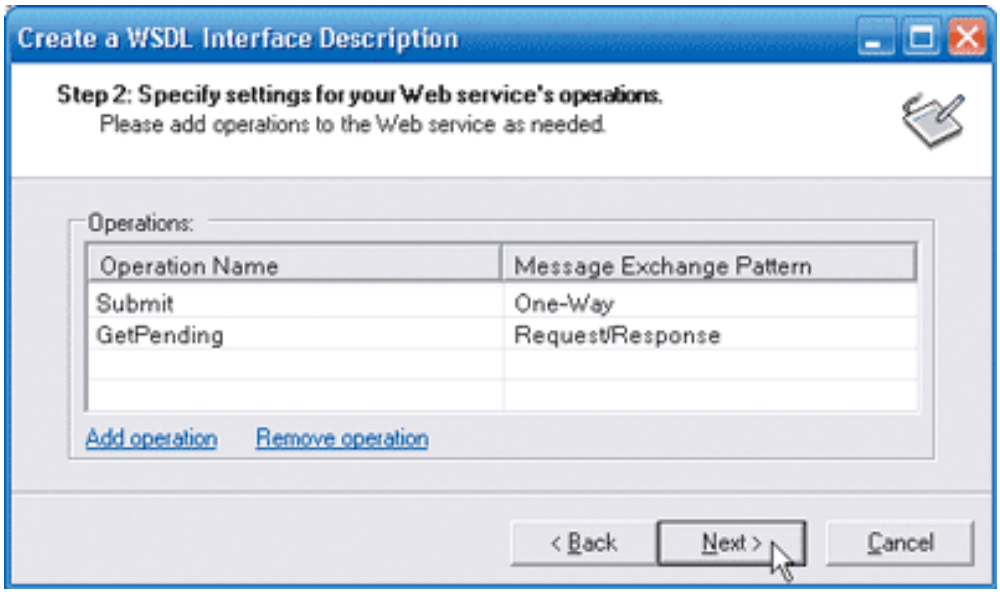

Figure 5 WsContractFirst Step 2

WsContractFirst is a great example of how simple contract-first development can be with the proper tool support. It's unfortunate that WSDL support isn't deeply integrated in Visual Studio .NET. That turns many developers away from the model despite the obvious benefits. However, thanks to emerging tools like WsContractFirst, it's definitely becoming more feasible. Shortly I'll show you a hybrid mechanism that works around this problem if you would still prefer to avoid WSDL altogether.

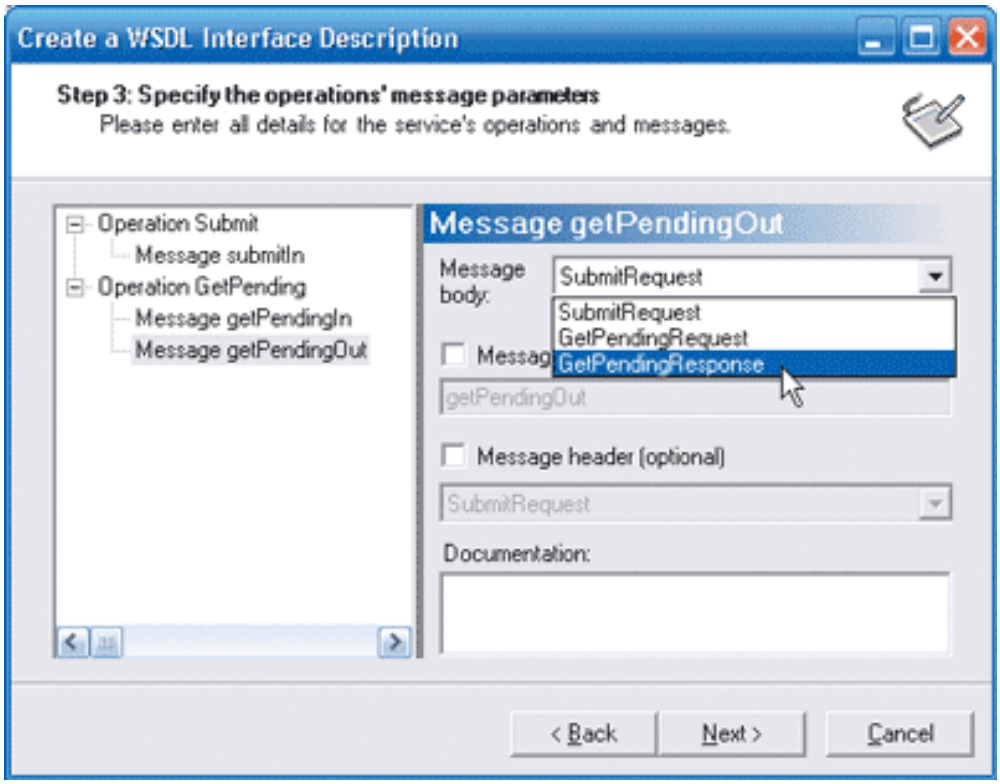

Figure 6 WsContractFirst Step 3

Once you've produced the WSDL contract for your service, you're ready to generate code and implement the operations.

Generate ASMX Code

With contract-first, you do the heavy lifting in the first two steps. This third step, code generation, is the easiest part of the whole process because you get to use a code generator to produce code that conforms to the contract, for either side of the wire.

If you're building code to consume the service, simply run it through wsdl.exe and it will generate a proxy class for you. If you need to implement the service, run "wsdl.exe /server" and it will generate an abstract class for you as a starting point—all you have to do is finish implementing it.

Figure 8 illustrates how to run wsdl.exe from the command line. It first shows how to generate a client-side proxy class. Depending on how you run wsdl.exe, you may have to specify the location of the XSD file as well as the WSDL file, as in this example. After generating this class, you can simply add it to your project and begin using it. The generation of client proxy code is also fully integrated with Visual Studio .NET via the "Add Web Reference" feature. After you point it to the WSDL contract, it generates the code, and automatically adds the files to your project. Since most developers are used to the client side, I'm going to focus attention on implementing a service.

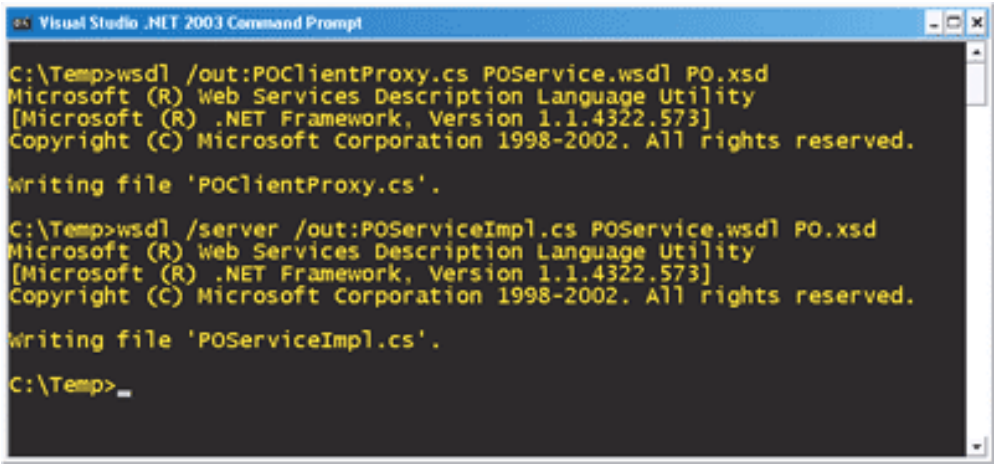

Figure 8 Running wsdl.exe

Figure 8 also shows how to use wsdl.exe to generate a service implementation from a WSDL file. This time specify "/server" on the command line.

Figure 9 shows part of the generated file. Notice that it creates a class that can be used with the ASMX infrastructure—the class in this case is called POService. POService is automatically decorated with all the necessary Web service attributes to ensure that the service implementation exposes the original WSDL contract (POService.wsdl). Wsdl.exe also generates all the data classes that model the original XSD types used by the WSDL definition (PO.xsd). These classes are also decorated with the appropriate XML serialization attributes to conform with the original schema.

To implement the service, add the generated files to your project and begin writing code (see the next step). Unfortunately, Visual Studio .NET does not provide any built-in support (similar to "Add Web Reference" on the client) for implementing services from WSDL, but Visual Studio 2005 Team System introduces support for this via the new Application Connection Designer (ACD).

If you want Visual Studio .NET integration today, you can use WsContractFirst. WsContractFirst also generates code from the WSDL contract. Assuming you have the tool installed in Visual Studio .NET, you can right-click on any WSDL definition in your project and select "Generate Web Service Code." This brings up the dialog shown in **Figure 10**. Options there include whether you want to generate a client-side proxy or a server-side interface (as in wsdl.exe /server). Other handy options are generating properties instead of fields, marking the classes serializable, and enabling XSD validation for incoming messages. And when you use this tool to generate a server-side interface, it greatly simplifies what you have to do in the next step, as you'll see shortly.

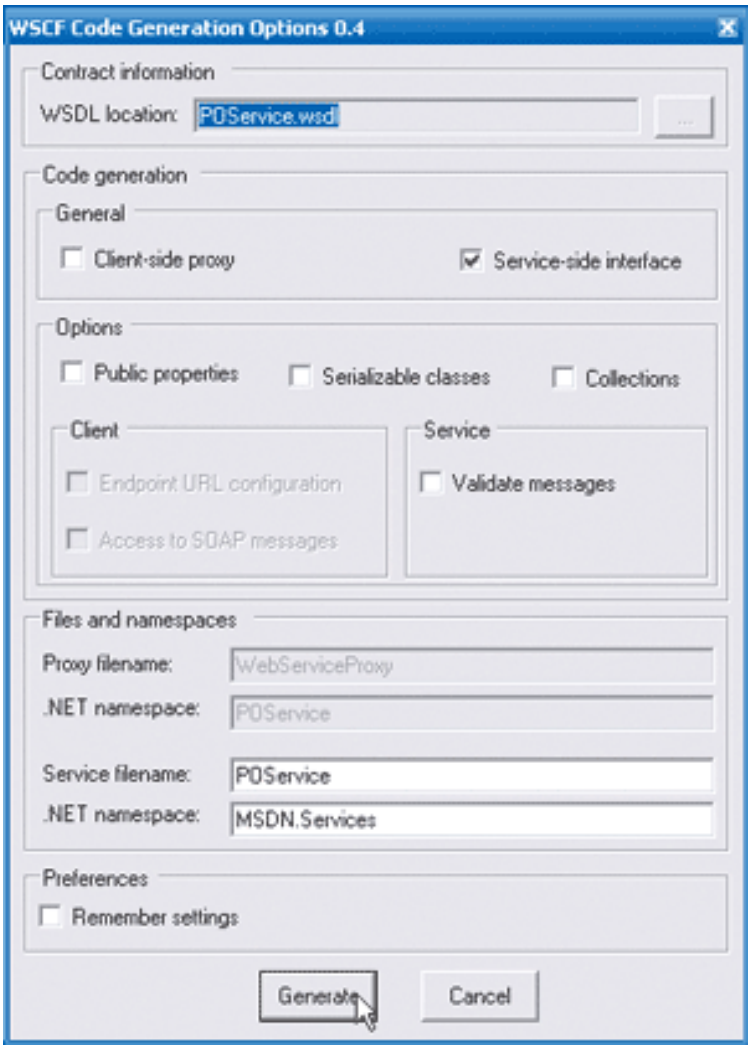

Figure 10 WsContractFirst Code Generator

When you generate .NET classes from WSDL, they'll contain all the attributes needed to ensure conformance to the predefined contract, shielding you from having to learn much about the .NET attributes themselves and letting you focus your time and energy on learning more about XSD and WSDL. When the ASMX framework is asked to generate WSDL for this class, it will produce a WSDL definition that is equivalent to the original. It's important for you to note that it doesn't return the original WSDL file—it actually generates a new one from the attributes found on the class. You can, however, tell ASMX to return the original WSDL file by specifying its location through the Location property on the [WebServiceBinding] attribute. Now that you have the service framework code generated, the next step is to implement the service operations.

Implement Service Operations

In this step your focus is on taking the generated code and implementing your business logic. You don't have to worry about modifying attributes since they've all been generated for you in the previous step. You should only have to add code within the method blocks. Then you simply need to bind the class to an .asmx file.

If you inspect the generated class definition shown in **Figure 9** you'll notice it is marked "abstract" and it is missing the [WebService] attribute. It was designed to have you derive a new class from the generated abstract class and implement the operations there. The [WebService] attribute belongs on the implementation class, which is why it's left off the generated abstract class. Given this approach, if you need to modify the WSDL contract down the road and regenerate the service interface, you won't accidentally overwrite your implementation code. The flaw is that the Web service attributes don't actually propagate during derivation. This ultimately means that you have to copy and paste all Web service attributes to the

new derived class regardless.

A simpler approach is to simply make the generated class concrete, removing the abstract keyword from the class and each method. You will also need to add a [WebService] attribute to the class (specifying at least the service namespace). Then you can add an implementation block to each method and begin writing your business logic. The only problem here, of course, is that you have to be careful if you need to modify the WSDL contract and regenerate the service interface. Make sure always to use a different file name via the /out switch so you don't accidentally overwrite your implementation code. Then after you generate the new abstract class, you'll have to repeat the process, making the class concrete, adding the [WebService] attribute, and then copying over your existing implementation code.

WsContractFirst shields you from some of these complexities. When you use the WsContractFirst code generator it will produce the typical abstract base class as well as a derived class with all the Web service attributes copied over. It also injects the [WebService] attribute for you on the derived class, so all you have to do is begin writing your code.

Figure 11 shows a sample derived class generated using WsContractFirst. You could still overwrite your implementation using this tool, so be careful when making changes. And if you generate a new service interface, you'll still need to copy over your existing method implementations.

There is a new feature coming in the .NET Framework 2.0 that will give developers a little more flexibility when making changes to the contract during development. The new /serverInterface switch tells the code generator to produce an interface definition instead of an abstract base class. The ASMX framework in the .NET Framework 2.0 now supports decorating interface definitions with Web service attributes in order to specify WSDL contract details. Then when you derive a class from the interface, all of the attributes propagate down as they should have before. The only attribute that you need to have on the derived class is [WebService].

Using an interface to hold all of the WSDL contract information makes a lot of sense because it cleanly separates the contract from the implementation code. If you need to make changes to the WSDL later in the development cycle, you can rerun wsdl.exe with /serverInterface and let it override the previous interface definition. Then you just need to make sure that your existing implementation still works—and the compiler can help you with that. You don't have to worry about overwriting your implementation code, since all of the WSDL metadata is cleanly factored into a separate artifact. For more details on this new feature, check out my column in the April 2005 issue of *MSDN®Magazine* at Service Station: Developing .NET Web Services with Beta 2.

Iterate

In the last step, one of the issues that comes up is how to deal with changes to the WSDL contract during the development cycle. For example, when you need to change the WSDL contract after you're already well into development, you also need to regenerate the proxy and service interfaces, rationalize those changes with your existing implementation code, and make sure that everything is still working.

Support for an iterative development process is crucial to long-term success with contract-first development. Obviously it's rare to get the contract definition right the first go-around, so you need the ability to go back to the original XSD and WSDL, make modifications, and keep everything in sync. This is the other place where the current tools let you down.

In the previous step I discussed all of the issues surrounding wsdl.exe and regenerating ASMX code. Today you have to manually copy code (either the attributes or the method implementations) in order to make this work. Even third-party tools like WsContractFirst don't currently address this issue.

With a fully integrated iterative development environment, you shouldn't have to care about where the contract metadata is stored (whether on the class or an interface). It should let you make changes to the contract and keep everything in sync automatically without losing changes to your existing code. This is the direction Microsoft is headed with future versions of Visual Studio.

Visual Studio 2005 Team System introduces an ACD that allows you to author service contracts before implementing them. You can model service contracts and specify which applications implement them and which consume them. Once the application framework has been completely modeled, the designer will generate all the service-related code behind the scenes, adding proxy code to consuming applications and implementation code to services implementing the contract. But the real beauty of the Application Connection Designer is how it lets you make changes to the contract while keeping all the generated code in sync, throughout all applications using the contract, without causing you to worry about losing implementation code.

The main problem with the ACD, from a contract-first perspective, is that it doesn't support modeling XML messages with XSD and service operations using WSDL. It still focuses on methods and parameters as opposed to schema types and elements. The good news is Microsoft plans to provide better support for a full message-oriented contract-first design experience in a future release of Visual Studio. Ill explore developing services using the new ACD in an upcoming column.

The overall lack of tool support for an iterative and integrated contract-first development process is why contract-first requires more discipline today. However, there is a hybrid technique that can provide a simpler process given today's tools.

Hybrid Approach

The two biggest problems with the current tool support are lack of WSDL support and lack of support for an iterative development process. The current support for XSD design, however, is quite reasonable. It turns out that using contract-first for XSD message design addresses most of the interoperability problems developers run into. One way to compromise between your interoperability and productivity needs is to use a hybrid approach, where you use contract-first to model XSD and .NET code to model WSDL.

In other words, you model the XSD as I showed you here (for your request and response messages) and then you run it through xsd.exe with the /classes switch (instead of layering WSDL and using wsdl.exe). This generates the .NET serialization classes that will produce the XML format described by the XSD. Then you can use those classes in your [WebMethod] signatures.

When you use this approach, the [WebMethod] should take a single argument (representing the request message) and specify a return type (representing the response message). Both should be decorated with an [XmlElement] attribute specifying the correct element names for the request and response messages (ASMX uses the argument name by default). The [WebMethod] should also be marked with [SoapDocumentMethod], specifying a ParameterStyle of SoapParameterStyle. Bare. This prevents ASMX from generating additional wrapper elements for the messages. This general style is referred to as document/literal/bare. Check out **Figure 9**, **Figure 11**, and **Figure 12**, all of which use document/literal/bare signatures. You can also use this technique with the new .NET Framework 2.0 interface model, which probably makes the most sense down the road.

As far as the iterative experience goes, a free third-party tool from www.sellsbrothers.com/tools called XsdClassesGen integrates with Visual Studio .NET and greatly simplifies this approach. This tool integrates running xsd.exe /classes within Visual Studio .NET. All you have to do is set the Custom Tool property for each .xsd file in your project to

SBXsdClassesGenerator and the add-in will automatically generate the classes behind the scenes and keep them attached to your .xsd files. When you modify the XSD contracts, the tool automatically runs and the classes are updated.

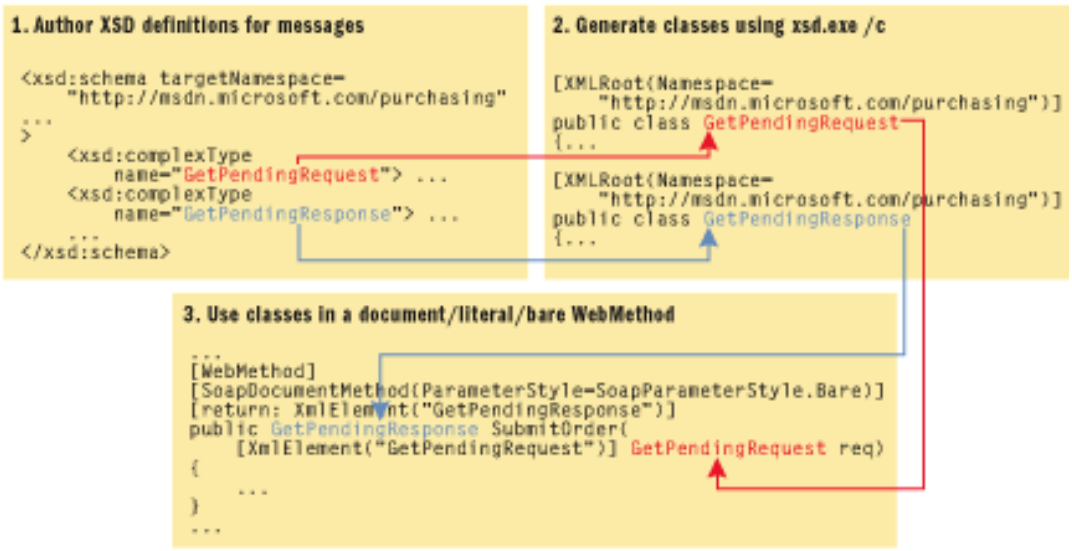

Figure 13 Document/literal/bare WebMethod

When you develop ASMX this way, as shown in **Figure 13**, you're using XSD to define the message formats, and you're using the ASMX interface to correlate the XSD messages into operations. The focus on XSD helps you avoid most interoperability concerns, and authoring the WSDL operations via the ASMX class gives you a more pleasant and productive experience.

Where Are We?

There are a variety of ways to employ contract-first design when building Web services with .NET. Whether you bite off authoring the complete WSDL definition by hand or you prefer to stop at XSD using the hybrid technique I described, your systems will benefit from increased interoperability and simplified versioning.

Additional Resources

- "Contract First Web Services Interoperability between Microsoft .NET and IBM WebSphere"
- Service Station, "SOA: More Integration, Less Renovation," MSDN Magazine
- MSDN Web Services Interoperability and Integration
- Service Station, "Improving Web Services Interoperability," MSDN Magazine

Send your questions and comments for Aaron to sstation@microsoft.com.

Aaron Skonnard is a co-founder of Pluralsight, a premier Microsoft .NET training provider. Aaron is the author of Pluralsight's *Applied Web Services*, *Applied BizTalk Server 2004*, and *Introducing Indigo* courses. Aaron has spent years developing courses, speaking at conferences, and teaching professional developers. Reach him at http://pluralsight.com/aaron.

From the June 2005 issue of MSDN Magazine.
Get it at your local newsstand, or better yet, *subscribe*.

Figure 2 Visual Studio .NET XSD Designer

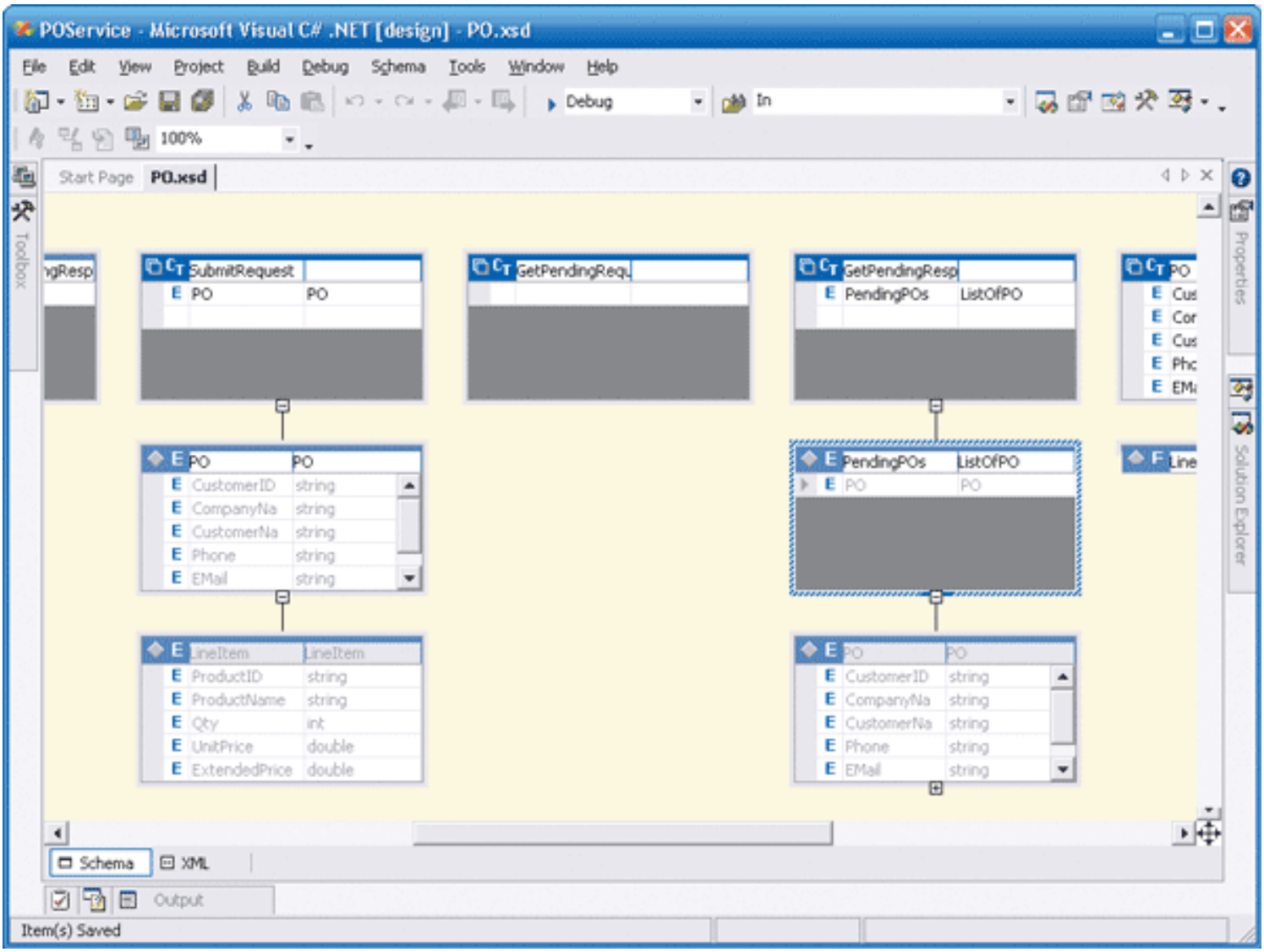

Figure 3 PO.xsd

```
<xs:schema xmlns="http://msdn.microsoft.com/purchasing"
   targetNamespace="http://msdn.microsoft.com/purchasing"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
    <xs:element name="SubmitRequest" type="SubmitRequest" />
    <xs:element name="GetPendingRequest" type="GetPendingRequest" />
    <xs:element name="GetPendingResponse" type="GetPendingResponse" />

    <xs:complexType name="SubmitRequest">
        <xs:sequence>
            <xs:element name="PO" type="PO" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="GetPendingRequest" />
    <xs:complexType name="GetPendingResponse">
        <xs:sequence>
            <xs:element name="PendingPOs" type="ListOfPO" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="PO">
        <xs:sequence>
          <xs:element name="CustomerID" type="xs:string" />
          <xs:element name="CompanyName" type="xs:string" />
          <xs:element name="CustomerName" type="xs:string" />
          <xs:element name="Phone" type="xs:string" />
```

```
          <xs:element name="EMail" type="xs:string" />
          <xs:element name="LineItem" type="LineItem"
                minOccurs="1" maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="LineItem">
        <xs:sequence>
          <xs:element name="ProductID" type="xs:string" />
          <xs:element name="ProductName" type="xs:string" />
          <xs:element name="Qty" type="xs:int" />
          <xs:element name="UnitPrice" type="xs:double" />
          <xs:element name="ExtendedPrice" type="xs:double" />
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ListOfPO">
        <xs:sequence>
          <xs:element name="PO" type="PO"
                maxOccurs="unbounded" />
        </xs:sequence>
    </xs:complexType>
  </xs:schema>
```

Figure 7 POService.wsdl

```
<definitions xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://msdn.microsoft.com/purchasing"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   targetNamespace="http://msdn.microsoft.com.com/purchasing"
   name="POService" xmlns="http://schemas.xmlsoap.org/wsdl/"
>
  <types>
    <xsd:schema>
      <xsd:import schemaLocation="PO.xsd"
        namespace="http://msdn.microsoft.com/purchasing" />
    </xsd:schema>
  </types>
  <message name="submitIn">
    <part name="messagePart" element="tns:SubmitRequest" />
  </message>
  <message name="getPendingIn">
    <part name="messagePart" element="tns:GetPendingRequest" />
  </message>
  <message name="getPendingOut">
    <part name="messagePart" element="tns:GetPendingResponse" />
  </message>
  <portType name="POServiceInterface">
    <operation name="Submit">
      <input message="tns:submitIn" />
    </operation>
    <operation name="GetPending">
      <input message="tns:getPendingIn" />
      <output message="tns:getPendingOut" />
    </operation>
  </portType>
  <binding name="POService" type="tns:POServiceInterface">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
       style="document" />
    <operation name="Submit">
      <soap:operation
         soapAction="http://msdn.microsoft.com/purchasing:Submit"
```

```
          style="document" />
      <input>
        <soap:body use="literal" />
      </input>
    </operation>
    <operation name="GetPending">
      <soap:operation
         soapAction="http://msdn.microsoft.com/purchasing:GetPending"
         style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
</definitions>
```

Figure 9 Code Generated by wsdl.exe /server

```
//------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version: 1.1.4322.573
// </autogenerated>
//------------------------------------------------------------------

using System.Diagnostics;
using System.Xml.Serialization;
using System;
using System.Web.Services.Protocols;
using System.ComponentModel;
using System.Web.Services;

/// <remarks/>
[WebServiceBinding(Name="POService",
  Namespace="http://msdn.microsoft.com/purchasing")]
public abstract class POService : WebService
{

    /// <remarks/>
    [WebMethod()]
    [SoapDocumentMethod("http://msdn.microsoft.com/purchasing:Submit",
      OneWay=true, Use= SoapBindingUse.Literal, ParameterStyle=
      SoapParameterStyle.Bare)]
    public abstract void Submit
    (
      [XmlElement(Namespace="http://msdn.microsoft.com/purchasing")]
      SubmitRequest SubmitRequest);

    /// <remarks/>
    [WebMethod()]
    [SoapDocumentMethod(
      "http://msdn.microsoft.com/purchasing:GetPending",
      Use=SoapBindingUse.Literal, ParameterStyle=
      SoapParameterStyle.Bare)]
    [return: XmlElement("GetPendingResponse",
      Namespace="http://msdn.microsoft.com/purchasing")]
    public abstract GetPendingResponse GetPending([XmlElement(
```

```
      Namespace="http://msdn.microsoft.com/purchasing")]
      GetPendingRequest GetPendingRequest);
}

/// <remarks/>
[XmlType(Namespace="http://msdn.microsoft.com/purchasing")]
public class SubmitRequest
{

    /// <remarks/>
    [XmlElement(Form=XmlSchemaForm.Unqualified)]
    public PO PO;
}
... // remaining classes omitted for brevity
```

Figure 11 Service-Interface Code

```
//------------------------------------------------------------------
// <autogenerated code>
//     This code was generated by WsContractFirst.
//     Changes to this file may cause incorrect behavior and will
//     be lost if the code is regenerated.
// </autogenerated code>
//------------------------------------------------------------------

namespace MSDN.Services
{
    using System.Diagnostics;
    using System.Xml.Serialization;
    using System;
    using System.Web.Services.Protocols;
    using System.ComponentModel;
    using System.Web.Services;

    /// <remarks/>
    [WebServiceBinding(Name="POService",
       Namespace="http://msdn.microsoft.com/purchasing")]
    [WebService(Namespace="http://msdn.microsoft.com/purchasing")]
    public class POServiceImpl : POService
    {

        /// <remarks/>
        [WebMethod()]
        [SoapDocumentMethod(
          "http://msdn.microsoft.com/purchasing:Submit", OneWay=true,
          Use=SoapBindingUse.Literal, ParameterStyle=
          SoapParameterStyle.Bare)]
        public override void Submit([XmlElement(
          Namespace="http://msdn.microsoft.com/purchasing")]
          SubmitRequest SubmitRequest)
        {
            ... // put your code here
        }

        /// <remarks/>
        [WebMethod()]
        [SoapDocumentMethod(
           "http://msdn.microsoft.com/purchasing:GetPending",
           Use= SoapBindingUse.Literal, ParameterStyle=
           SoapParameterStyle.Bare)]
```

```
        [return: XmlElement("GetPendingResponse",
           Namespace="http://msdn.microsoft.com/purchasing")]
        public override GetPendingResponse GetPending([XmlElement(
           Namespace="http://msdn.microsoft.com/purchasing")]
           GetPendingRequest GetPendingRequest)
        {
            ... // put your code here
        }
    }
}
```

Figure 12 Classes Generated Using wsdl.exe /serverinterface

```
//------------------------------------------------------------------
// <autogenerated>
//     This code was generated by a tool.
//     Runtime Version:2.0.40903.19
// </autogenerated>
//------------------------------------------------------------------
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Serialization;

/// <remarks/>
[WebService(Namespace="http://msdn.microsoft.com/purchasing")]
[WebServiceBinding(Name="POService",
   Namespace="http://msdn.microsoft.com/purchasing")]
public interface IPOService {

    /// <remarks/>
    [WebMethod()]
    [SoapDocumentMethod("http://msdn.microsoft.com/purchasing:Submit",
       OneWay=true, Use= SoapBindingUse.Literal, ParameterStyle=
       SoapParameterStyle.Bare)]
    void Submit([XmlElement(
       Namespace="http://msdn.microsoft.com/purchasing")]
       SubmitRequest SubmitRequest);

    /// <remarks/>
    [WebMethod()]
    [SoapDocumentMethod(
      "http://msdn.microsoft.com/purchasing:GetPending",
      Use= SoapBindingUse.Literal, ParameterStyle=
      SoapParameterStyle.Bare)]
    [return: XmlElement("GetPendingResponse",
      Namespace="http://msdn.microsoft.com/purchasing")]
    GetPendingResponse GetPending([XmlElement(
      Namespace="http://msdn.microsoft.com/purchasing")]
      GetPendingRequest GetPendingRequest);
}

/// <remarks/>
[Serializable()]
[XmlType(Namespace="http://msdn.microsoft.com/purchasing")]
public partial class SubmitRequest {

    private PO poField;
```

```
    /// <remarks/>
    [XmlElement(Form=XmlSchemaForm.Unqualified)]
    public PO PO {
        get {
            return this.poField;
        }
        set {
            this.poField = value;
        }
    }
}
... // remaining classes omitted for brevity
```
