SERVICE STATION

Contract-First Service Development

Aaron Skonnard

In one of my previous columns on Service Orientation (SO), I introduced the concept of "contract-first" service development (see Service Station: SOA: More Integration, Less Renovation). Over the next two installments of this column, I'm going to cover the topic in depth. In this first part, I'll build the case for contract-first development and discuss the ASMX programming model. As a follow-up, I'll walk you through a practical contract-first development process using today's tools and illustrate a hybrid technique that can give you a good balance between interoperability and productivity.

Lessons Learned from COM

The shift from object-orientation to component development made it possible to build more loosely coupled and flexible systems, in which individual components could evolve independently without affecting the rest of the system. Component technology made this possible by enforcing strict boundaries and restricting access to objects through explicit interface definitions. An interface-based development approach focuses on sharing explicit interface definitions, or "contracts," expressed in a language easily understood by others, removing dependencies on implementation details and offering programming language interoperability. There are valuable design lessons to be learned from the experience with components as SO evolves. One of the most important of these lessons revolves around contract design.

In order to use a class found inside a COM component, you ask the runtime to activate an instance through a class factory. The runtime then returns an interface reference for communicating with the object instance. All further interactions with the instance occur through the supplied interface reference. This interface-based model shields the consumer from all class implementation details. The only dependency the consumer takes on in this situation is the binding to the interface definition. As long as the interface definition doesn't change, the implementation can be updated with bug fixes or other necessary improvements.

The decoupling provided by interface-based development also makes it possible to distribute components across different contexts without impacting the consumer's programming model. As far as the programming model is concerned, it doesn't matter to the consumer whether the instance is local (on the same machine) or remote (on a separate machine), although there are some obvious design considerations. The DCOM infrastructure handles this through an interception layer that takes care of serializing method calls into the appropriate wire-level messages for transmission between machines. It handles recreating the call stacks on the remote machine, invoking the instance, and returning any values. The interface definition essentially tells the DCOM infrastructure what to do and therefore drives the wire-level contract of the system.

COM relied on a common language for defining interface definitions known as the Interface Definition Language (IDL). IDL can be implemented in a variety of different programming languages, such as C++ and Visual Basic®. In fact, IDL is typically used to generate header files and type libraries that can be used from these different languages during development (see **Figure 1**).

Figure 1 IDL Contracts and Language Interoperability

Components authored in different programming languages can successfully interact with each other and achieve interoperability, since they share an understanding of the interface definition. It has been quite common for Visual Basic applications to rely on C++ components and vice-versa. This type of cross-language interoperability was one of COM's greatest strengths.

Contract-First Component Development

While it's true that COM offered cross-language interoperability, it required discipline to achieve it in practice. Despite the interface-based architecture, it was common for problems to arise when trying to achieve interoperability across Visual Basic and C++ components due to the differences in their type systems, their capabilities, and how the IDL contracts were authored by developers. There was a big difference between how the two camps thought about, authored, and used IDL.

Developers using C++ typically authored their IDL interface definitions by hand, focusing on the contract before the implementation—a contract-first approach. After authoring IDL, C++ developers ran it through midl.exe, a code generator that produced header files, type libraries, and other source files needed by the COM interception layer. The header file contained C++-style interface definitions that the developer could implement in a C++ class. C++ clients used the same exact files to write code that consumed the interface.

The Visual Basic support for COM attempted to hide IDL from the developer as much as possible because it was generally considered hard to use. Hence, most developers using Visual Basic didn't start with IDL during design. Instead, they started by authoring Visual Basic classes—a code-first approach. During compilation, Visual Basic 6.0 automatically made the class into a COM object and produced an interface definition behind the scenes. The generated interface became part of the component's type library, which was referenced by clients wanting to use the component.

The code-first approach made it much easier for developers using Visual Basic to build COM applications, but it came with its own set of problems. Although it increased productivity, this code-first approach complicated versioning. Since developers were shielded from the official interface definitions, they became completely dependent on the IDE to manage them properly. The translation layer (from Visual Basic to IDL) made it harder for developers to understand the consequences of modifying their class definitions during development. As a result, the Visual Basic IDE came with several compatibility modes, which provided the needed assistance and warnings when making breaking changes, perhaps unintentionally, during development. Using the compatibility model effectively in real-world long-term projects became one of the more complex areas of Visual Basic COM development.

The other problem with the Visual Basic code-first approach was that it hurt interoperability in mixed-language environments. The self-centered model didn't encourage developers to think about the types they were using in their interface definitions, and the problems that such types might cause in other language environments. Hence, it was common for developers using Visual Basic to naively produce method signatures that became extremely difficult, if not impossible, to work with in C++. Now to be fair, it was also possible for developers using C++ to produce interfaces that were unfriendly to Visual Basic, even though they started with IDL, but in general, they were more aware of the issues given their focus on the contract. An important aspect to designing a shared contract is considering the needs of all possible implementation environments before writing code.

Many developers didn't realize that they too could enjoy the benefits of contract-first development by authoring the IDL first, compiling the IDL into a type library, and then referencing the type library to bring the interface definition into the project. Then it was as simple as implementing the interface explicitly in a new Visual Basic class and, by so doing, avoiding an auto-generated interface and the problematic translation layer. It didn't take long for some organizations to catch on to this secret and to encourage its use, especially in environments with collaborating teams of developers using C++ and Visual Basic.

Now, not all organizations using COM were concerned with mixed-language support or long-term versioning and were more interested in improving short-term development productivity. Such organizations were perfectly content, and even successful, taking the Visual Basic code-first route despite some of the issues I've outlined. Ironically, the new world of SO is heading down the same path, dealing with a very similar set of issues.

Fast-Forward to Service Orientation

A service is very similar to a distributed component, only the component runtime (COM) is no longer required by those who want to consume the service. The service contract shields consumers from all implementation details, including the platform, development framework, and programming language in use. Anyone who understands the service contract—the message formats and transport protocols—can have access to the service; this significantly increases reuse.

As with component development, the interfaces or contracts become the building blocks of an SO application. However, instead of IDL, there are new languages that are used to express service contracts, all of which are now widely understood throughout the industry. XML Schema Definition (XSD) is used to define the data contracts—the precise format of the XML messages that travel on the wire via SOAP. Web Services

Description Language (WSDL) is layered over XSD and specifies how the XSD messages are correlated to define operations—the focus here is on sequencing, message exchange patterns, and transport details. These languages together provide an interoperable contract language for services. (In addition to XSD and WSDL, Web service policies can be used to define additional rules and behavior such as security constraints, which can also be considered to be part of the behavioral contract.)

When working with a service, the only dependency you have to enter into is the binding to the XSD and WSDL definitions in order to produce the correct wire-level protocol. And like IDL, XSD and WSDL can be mapped to any programming language and used in a variety of different development environments, promoting great flexibility and a wide range of programming options (see **Figure 2**).

Figure 2 WSDL Contracts

Like the component days, most of today's Web services frameworks make it possible to design services using either code-first or contract-first techniques; however, most Web service development frameworks push developers towards the former. Frameworks that provide a code-first approach provide a translation layer to map between the native programming language and XSD and WSDL.

Code-First ASMX

Like Visual Basic 6.0, the ASP.NET Web services framework, also known as ASMX, enlists a code-first development process that hides the XSD and WSDL contract from developers. This approach makes it easy to get Web service endpoints up and running by simply annotating classes with predefined Web service attributes and binding the class to an .asmx endpoint.

The code in **Figure 3** shows a simple ASMX service for submitting and retrieving purchase orders (po.asmx). You should notice that the OrderService class looks like a regular .NET class with a few extra Web service attributes, specifically [WebService] and [WebMethod]. These attributes tell the ASMX framework to expose the class as a Web service and to expose the marked methods as Web service operations, respectively.

ASMX automatically maps the class and the [WebMethod] signatures to a WSDL contract. The ASMX approach encourages a remote procedure call (RPC) programming style, where developers are focused more on methods and parameters than XML messages and elements. ASMX uses the method and parameter names to generate the XML message used on the wire. By default, ASMX generates a wrapper element with the same name as the method. It also generates an element for each parameter and nests it within the wrapper element. Other attributes, such as [SoapDocumentService] and [SoapDocumentMethod], can be used to customize the precise WSDL mapping, and to control whether or not the method wrapper element is used.

ASMX also takes care of mapping .NET data types and other user-defined types to an appropriate XSD type within the generated WSDL. You can also use various serialization attributes (like [XmlType], [XmlRoot], [XmlElement], and [XmlAttribute] found in System.Xml.Serialization) to customize the XSD mapping if you need something other than the default transformation.

If you browse to order.asmx you'll see that ASMX automatically generates a human-readable documentation page along with the official WSDL contract that others can use. If you tweak your code or modify the attributes in use and refresh the ASMX page, you'll notice the WSDL contract change, illustrating that the ASMX code is driving the contract details. **Figure 4** illustrates what the SOAP request looks like for SubmitPO (that you saw in **Figure 3**), as taken from the documentation page.

Like Visual Basic, ASMX embraces a code-first development model through a sophisticated translation layer. At run time, the translation layer provides all the necessary dispatching and serialization support to map between the worlds of SOAP and .NET. For many developers, the translation layer is a godsend because the familiar development process increases productivity. However, like before, the translation layer can become troublesome and usually causes developers to lose sight of the real service contract (the XSD and WSDL) they are producing and forcing consumers to use. Like before, this can cause problems with interoperability and long-term versioning.

Experience has shown that services implemented using code-first techniques, like the default ASMX model, are less likely to interoperate cleanly in mixed-language environments. However, the difference with SO is that mixed environments are much more common—practically the norm—making interoperability a much greater design concern. The problem is compounded by the fact there are more programming languages and type systems involved in this new world than before in the component days. Now it's even more important to consider all possible implementation

environments during the contract design stage.

Code-first can also complicate versioning because the developer isn't in direct control of the contract, but must rely on the translation layer throughout development. However, unlike Visual Basic 6.0, ASMX doesn't provide any fancy compatibility modes to help the developer know when making changes to the ASMX class may break contract compatibility with existing clients, which makes the situation even worse.

Contract-First ASMX

.NET provides the necessary tool support for contract-first ASMX development today, but it requires more discipline than the convenient code-first approach. To use a contract-first approach, you start by authoring the XSD definitions to model the messages used by the service. Next you author the WSDL to define the service operations, specifying which XSD messages are sent and received. There are a variety of applications you can use to author these documents, including Notepad. These first few steps are the hardest for most developers because they aren't familiar with XSD and WSDL and because there isn't a great deal of tool support.

Once you have the WSDL contract created, you use wsdl.exe (with the "/server" switch) to generate the ASMX service interface, leaving you with the simple task of implementing the operations. The generated service interface contains all the necessary .NET attributes to produce the same WSDL contract from which you started. All you have to do is implement this service interface using your own business logic.

When you generate the WSDL contract up front, numerous developers from different organizations, perhaps working on different platforms and in different languages, can use code generation tools from their frameworks to begin developing against the predefined contract before you've even started writing code. Development work against the predefined contract, where you may have several organizations implementing the same service contract and various client applications being developed to consume them, can happen simultaneously (see **Figure 5**).

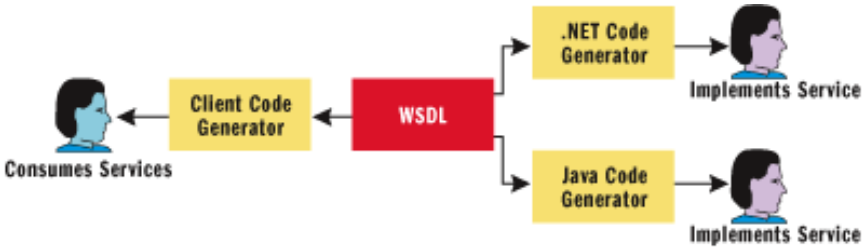

Figure 5 Contract-First Development with WSDL

Figure 6 summarizes the differences between the ASMX code-first and contract-first development models. With contract-first, you do the heavy lifting in the first few steps, authoring the contract (WSDL), after which you automate the generation of the .NET code. With code-first, you start by authoring the .NET code, and let ASMX generate the contract (WSDL). The choice you'll make will boil down to what you want to author and what you want to automate. Both approaches have their pros and cons, and it makes sense to use each one at certain times.

Contract-First or Code-First?

When designing a service contract, you can always choose to use either code-first or contract-first techniques. Ultimately the decision hinges on whether you care more about ensuring interoperability or improving productivity.

It's interesting that contract-first is generally considered the right model to follow when building clients. Virtually all frameworks start by generating code from WSDL on the client slide. You need to conform to a contract, so you start with the contract and generate the code—that makes sense. But when you start talking about using the same model for implementing services, some developers don't see it as clearly. So when should you definitely consider using the contract-first plan?

The answer is any time there is a strong emphasis on the messages exchanged or you need to ensure conformance to an existing message format. BizTalk® applications are a good example of this because they are typically designed around (existing) message formats from the ground up. Interestingly, BizTalk natively supports a contract-first development experience. When designing a BizTalk orchestration, developers start by designing the schemas that represent the input and output and everything else is designed from that starting point. Thanks to the BizTalk support for contract-first, developers commonly embrace BizTalk (perhaps instead of using ASMX) when they need to conform to existing standards. Contract-first is, without question, the most natural approach in a message-oriented world, such as that of BizTalk.

Contract-first is also especially important when you require conformance to a particular contract from third parties, or when you're collaborating with other groups to define a shared contract throughout an organization. Using contract-first here allows you to solidify the contract up front

while numerous parties implement the contract in parallel. All of these situations are common in industry today.

In a real-world SO system, it's likely that you'll have to implement and consume contracts designed by others, and you're also going to design contracts for others to use. And when you're forced to use someone else's contract, all you're going to care about is that it works with the framework you're stuck using. So when you own the contract, it is common sense to strive to make the contract as easy as possible for your consumers to use, which also begs for a contract-first model.

The biggest obstacle to contract-first design today is the lack of tool support, which hurts productivity. When productivity is the overriding concern, code-first can be an acceptable approach, especially when you're not particularly concerned with mixed-environment interoperability.

Contract-First Tool Support

All the tools you need to employ contract-first development are available today, but unfortunately they're not integrated within a single, unified development environment. Visual Studio® .NET provides an intuitive XSD designer that can be used to author your data contracts. And the Microsoft® .NET Framework provides the code generation tools you need, including an XSD code generator (xsd.exe) and a WSDL code generator (wsdl.exe). The main things missing from .NET are a WSDL designer and support for an iterative development process fully integrated with Visual Studio .NET.

There are, however, numerous third-party tools that can help improve the contract-first experience today. There are sophisticated WSDL editors that give you a more user-friendly experience than Notepad. There are also tools like Thinktecture's WsContractFirst that integrate with Visual Studio .NET and fill some of the big holes. So as long as you're willing to use some third-party tools, you can pull together a complete toolset supporting the contract-first model. There are always developers who don't like stepping outside of Visual Studio .NET no matter what, and if you're one of them, you'll simply have to wait for Microsoft to fill the gap.

Microsoft has taken notice of the community's general desire for more tool support in this area and has already begun making improvements to Visual Studio .NET. In Visual Studio 2005 Team System, you'll notice a new Application Connection Designer that allows you to model service contracts and interactions with other applications. The designer allows you to model the service contracts, including the applications that are going to use them, before writing a single line of ASMX code. Once you're happy with the contracts you've authored, the designer will generate all of the service-related code for the applications using them.

The Application Connection Designer is a great step in the right direction and a good teaser for what a fully integrated and iterative contract-first experience could be like. However, it doesn't provide an integrated approach for modeling messages with XSD and operations using WSDL. It still focuses on the contracts from an RPC perspective, focusing on methods and parameters, as opposed to a message perspective, where your focus is on schema types and elements. The good news is that Microsoft plans to provide better support for a full message-oriented contract-first design experience in a future release of Visual Studio.

The reason I say this approach requires more discipline today is the overall lack of tool support for an iterative and integrated contract-first development process. In a follow-up column, I'll walk you through some practical examples of contract-first development techniques using the various tools I've mentioned here.

What's Next?

Component development teaches some important lessons about designing contracts. Embracing a contract-first development process will provide your systems with increased interoperability and better versioning control. However, if you're concerned less with interoperability and more with productivity, code-first may still be a reasonable approach. Future tools will continue to make contract-first development easier, but that doesn't mean you can't begin using it today.

My next column will show you how to get started. In the meantime, check out the following references on this topic. "Contract First Web Services Interoperability between Microsoft .NET and IBM WebSphere", "Web Services Interoperability", and my Service Station columns from February 2005 and November 2004.

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

Aaron Skonnard is a co-founder of Pluralsight, an education and content creation company, where he focuses on XML and Web services technologies. He's a contributing editor to *MSDN Magazine* and the author of several books, including *Essential XML Quick Reference* (Addison-Wesley, 2001). Reach him at www.pluralsight.com/aaron.

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

Figure 3 Sample ASMX Endpoint (po.asmx)

```
<%@ WebService Language="C#"
    CodeBehind="~/Code/Test.cs" Class="POService" %>

using System.Web.Services;

[WebService(Namespace="http://pluralsight.com/purchasing")]
public class POService : WebService {
    [WebMethod]
    public void SubmitPO(string customerId, string customerName,
        string companyName, LineItem[] items) {
        ... // process order here
    }
}

public class LineItem {
    public string productId;
    public string productName;
    public double quantity;
    public double unitPrice;
    public double extendedPrice ;
}
```

Figure 4 SOAP Request Format for SubmitPO

```
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://
schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <SubmitPO xmlns="http://pluralsight.com/purchasing">
      <customerId>string</customerId>
      <customerName>string</customerName>
      <companyName>string</companyName>
      <items>
        <LineItem>
          <productId>string</productId>
          <productName>string</productName>
          <quantity>double</quantity>
          <unitPrice>double</unitPrice>
          <extendedPrice>double</extendedPrice>
        </LineItem>
        ...
      </items>
    </SubmitPO>
  </soap:Body>
</soap:Envelope>
```

Figure 6 Code-First Versus Contract-First Development

Code-First Development	
1.	Author ASMX classes directly, defining contract and implementing operations simultaneously
2.	Use ASMX to generate XSD and WSDL
3.	Iterate
Contract-First Development	
1.	Define XML messages using XSD
2.	Define operations using WSDL
3.	Generate ASMX code from WSDL
4.	Implement operations

5.	Iterate