Understanding WSDL

Aaron Skonnard
Northface University

October 2003

Applies to:
- Web Services
- Web Services Description Language (WSDL) 1.1
- WS-I Basic Profile Version 1.0
- XML Messaging
- XML Schema

Summary: See the importance of WSDL in the overall Web services architecture, as it describes the complete contract for application communication. Make Web services widely approachable by using WSDL definitions to generate code that knows precisely how to interact with the Web service described, and hides tedious details in sending and receiving SOAP messages over different protocols. (24 printed pages)

Contents

Overview

XML makes it possible for developers to expose valuable resources in a highly interoperable fashion, where a resource is any type of application or data store used within an organization. The XML Web services architecture defines a standard mechanism for making resources available via XML messaging. Being able to access a resource by simply transmitting XML messages over standard protocols like TCP, HTTP, or SMTP greatly lowers the bar for potential consumers. The term "Web service" (or simply "service") typically refers to the piece of code implementing the XML interface to a resource, which may otherwise be difficult to access (see Figure 1).

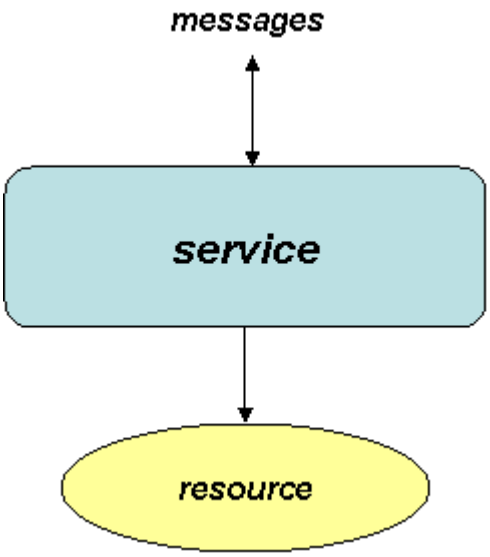

Figure 1: Resources and services

This architecture makes it possible for any consumer with XML support to integrate with Web service applications. However, in order to accomplish this, consumers must determine the precise XML interface along with other miscellaneous message details a priori. XML Schema can partially fill this need because it allows developers to describe the structure of XML messages. XML Schema alone, however, can't describe the additional details involved in communicating with a Web service.

A schema definition simply tells you what XML messages may be used but not how they relate to each other. For example, if there's an XML element named **Add** and another named **AddResponse**, it's likely they're related to each other but there's no way to indicate that in the schema. Hence, in addition to being aware of the messages, consumers must also be aware of the possible message exchanges supported by the Web service (e.g., if you send an **Add** message, you get an **AddResponse** message back).

A message exchange is also referred to as an operation. Operations are what consumers care about most since they're the focal point of interacting with the service (see Figure 2). Whenever I approach a new Web service, I first inspect its list of supported operations to get an overall feel for what it offers.

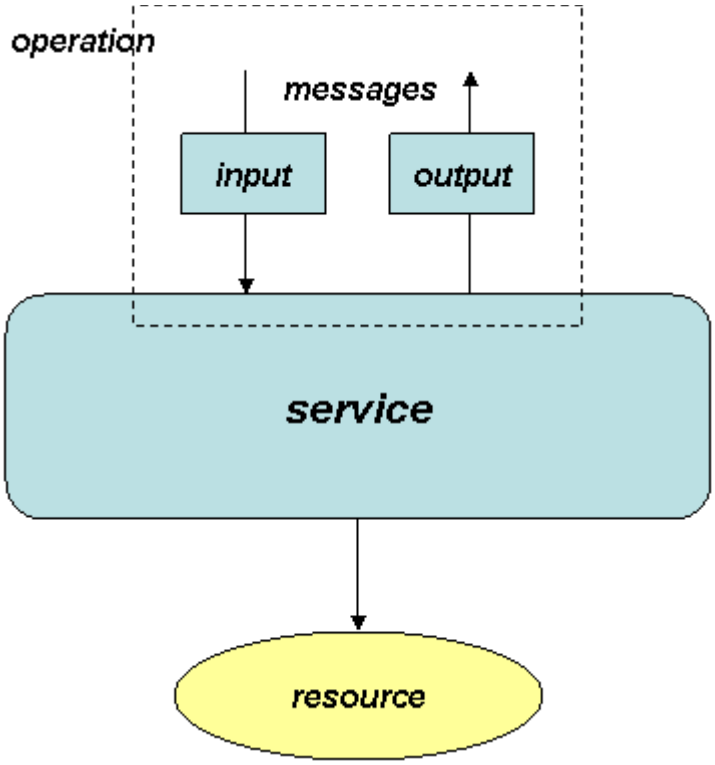

Figure 2: Messages and operations

It's common for developers to group related operations into interfaces. Consumers must be aware of these groupings since it impacts the way they write their code. This is especially important to developers working with Web services in object-oriented environments since the XML interfaces can map to programmatic interfaces (or abstract classes) in their language of choice.

Consumers must also know what communication protocol to use for sending messages to the service, along with the specific mechanics involved in using the given protocol such as the use of commands, headers, and error codes. A binding specifies the concrete details of what goes on the wire by outlining how to use an interface with a particular communication protocol. A binding also influences the way abstract messages are encoded on the wire by specifying the style of service (document vs. RPC) and the encoding mechanism (literal vs. encoded). Check out Understanding SOAP for more background on these concepts.

A service can support multiple bindings for a given interface, but each binding should be accessible at a unique address identified by a URI, also referred to as a Web service endpoint (see Figure 3).

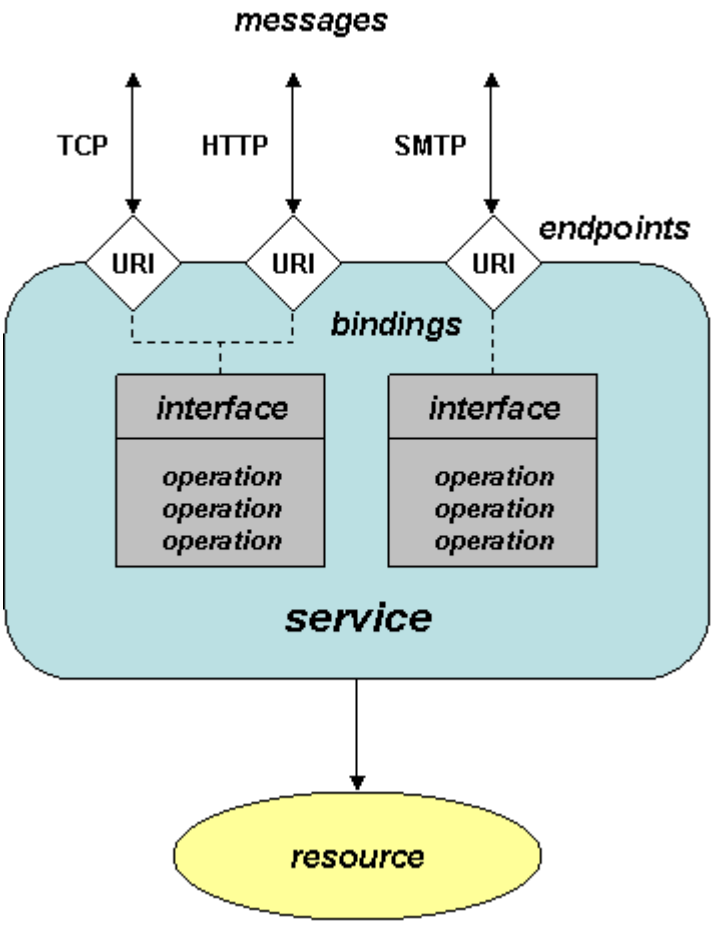

Figure 3: Interfaces and bindings

Consumers must discover all of the details described above before they can interact with a Web service. The Web Services Description Language (WSDL) provides an XML grammar for describing these details. WSDL picks up where XML Schema left off by providing a way to group messages into operations and operations into interfaces. It also provides a way to define bindings for each interface and protocol combination along with the endpoint address for each one. A complete WSDL definition contains all of the information necessary to invoke a Web service. Developers that want to make it easy for others to access their services should make WSDL definitions available.

WSDL plays an important role in the overall Web services architecture since it describes the complete contract for application communication (similar to the role of IDL in the DCOM architecture). Although other techniques exist for describing Web services, the WS-I Basic Profile Version 1.0 mandates the use of WSDL and XML Schema (see Figure 4) for describing Web services. This helps ensure interoperability at the service description layer.

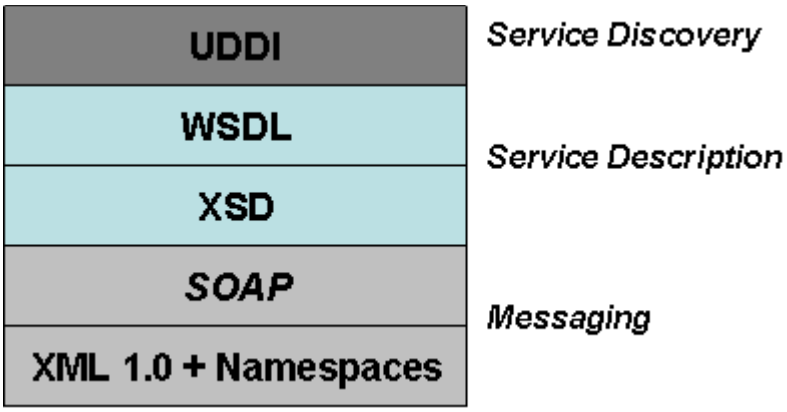

Figure 4: WS-I Basic Profile 1.0 technologies

Since WSDL is a machine-readable language (e.g., it's just an XML file), tools and infrastructure can be easily built around it. Today developers can use WSDL definitions to generate code that knows precisely how to interact with the Web service it describes. This type of code generation hides the tedious details involved in sending and receiving SOAP messages over different protocols and makes Web services approachable by the masses.

The Microsoft® .NET Framework comes with a command-line utility named wsdl.exe that generates classes from WSDL definitions. Wsdl.exe can generate one class for consuming the service and another for implementing the service. (Apache Axis comes with a similar utility named WSDL2Java that performs the same function for Java classes.) Classes generated from the same WSDL definition should be able to communicate with each other through the WSDL-provided interfaces, regardless of the programming languages in use (see Figure 5).

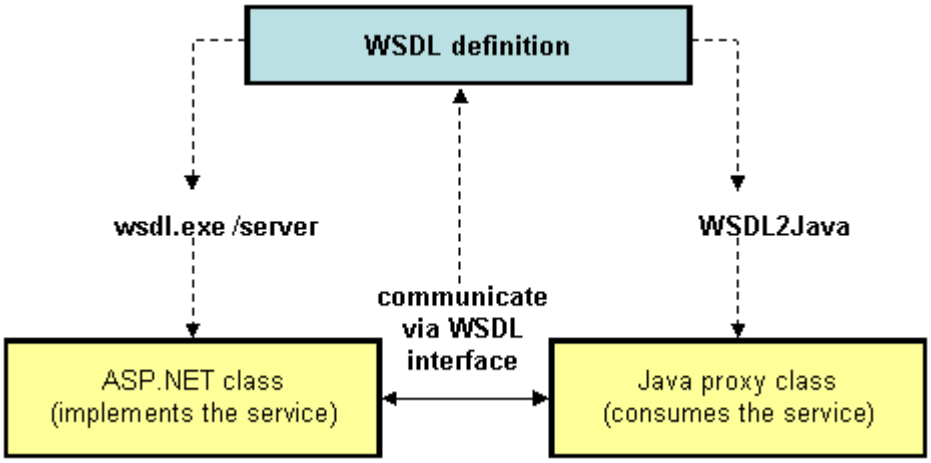

Figure 5: WSDL and code generation

WSDL 1.1 is considered the de-facto standard today because of it's industry-wide support. Most Web services toolkits support WSDL 1.1, but there have been some interoperability problems across the different implementations. Many developers believe that the extensive flexibility of WSDL (and the resulting complexity) is the fundamental source of these problems. The WS-I has helped resolve some of these issues by encouraging developers to use certain parts of the specification and discouraging them from using others.

The W3C is actively working on the next "official" version of WSDL, WSDL 1.2, but it's currently only a Working Draft and not supported by the mainstream toolkits, if any. The remainder of this article discusses the details of a WSDL 1.1 definition and highlights some of the WS-I basic profile suggestions along the way.

WSDL Basics

A WSDL definition is an XML document with a root definition element from the http://schemas.xmlsoap.org/wsdl/ namespace. The entire WSDL schema is available at http://schemas.xmlsoap.org/wsdl/ for your reference. The definitions element may contain several other elements including types, message, portType, binding, and service, all of which come from the http://schemas.xmlsoap.org/wsdl/ namespace. The following illustrates the basic structure of a WSDL definition:

```
<!-- WSDL definition structure -->
<definitions
    name="MathService"
targetNamespace="http://example.org/math/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
>
   <!-- abstract definitions -->
    <types> ...
    <message> ...
   <portType> ...

   <!-- concrete definitions -->
   <binding> ...
   <service> ...
</definition>
```

Notice that you must specify a target namespace for your WSDL definition, just like you would for an XML Schema definition. Anything that you name in the WSDL definition (like a message, portType, binding, etc.) automatically becomes part of the WSDL definition's target namespace defined by the **targetNamespace** attribute. Hence, when you reference something by name in your WSDL file, you must remember to use a qualified name.

The first three elements (**types**, **message**, and **portType**) are all abstract definitions of the Web service interface. These elements constitute the programmatic interface that you typically interface with in your code. The last two elements (**binding** and **service**) describe the concrete details of how the abstract interface maps to messages on the wire. These details are typically handled by the underlying infrastructure, not by your application code. Table 1 provides brief definitions for each of these core WSDL elements and the remaining sections discuss them in more detail.

Table 1. WSDL Elements

Element Name	Description
types	A container for abstract type definitions defined using XML Schema
message	A definition of an abstract message that may consist of multiple parts, each part may be of a different type
portType	An abstract set of operations supported by one or more endpoints (commonly known as an interface); operations are defined by an exchange of messages
binding	A concrete protocol and data format specification for a particular portType

service	A collection of related endpoints, where an endpoint is defined as a combination of a binding and an address (URI)

Types

The WSDL types element is a container for XML Schema type definitions. The type definitions you place here are referenced from higher-level message definitions in order to define the structural details of the message. Officially, WSDL 1.1 allows the use of any type definition language, although it strongly encourages the use of XML Schema and treats it as its intrinsic type system. The WS-I enforces this by mandating the use of XML Schema in the Basic Profile 1.0.

The types element contains zero or more schema elements from the http://www.w3.org/2001/XMLSchema namespace. The basic structure of the types element (with namespaces omitted) is as follows (***** means zero or more):

```
<definitions .... >
    <types>
        <xsd:schema .... />*
    </types>
</definitions>
```

You can use any XML Schema construct within the schema element, such simple type definitions, complex type definitions, and element definitions. The following WSDL fragment contains an XML Schema definition that defines four elements of type **MathInput** (**Add**, **Subtract**, **Multiply**, and **Divide**) and four elements of type **MathOutput** (**AddResponse**, **SubtractResponse**, **MultiplyResponse**, and **DivideResponse**).

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/"
>
  <types>
    <xs:schema
      targetNamespace="http://example.org/math/types/"
      xmlns="http://example.org/math/types/"
    >
     <xs:complexType name="MathInput">
       <xs:sequence>
      <xs:element name="x" type="xs:double"/>
      <xs:element name="y" type="xs:double"/>
       </xs:sequence>
     </xs:complexType>
     <xs:complexType name="MathOutput">
       <xs:sequence>
         <xs:element name="result" type="xs:double"/>
       </xs:sequence>
     </xs:complexType>
     <xs:element name="Add" type="MathInput"/>
     <xs:element name="AddResponse" type="MathOutput"/>
     <xs:element name="Subtract" type="MathInput"/>
     <xs:element name="SubtractResponse" type="MathOutput"/>
     <xs:element name="Multiply" type="MathInput"/>
     <xs:element name="MultiplyResponse" type="MathOutput"/>
```

```
     <xs:element name="Divide" type="MathInput"/>
     <xs:element name="DivideResponse" type="MathOutput"/>
    </xs:schema>
  </types>
   ...
</definitions>
```

If you're new to XML Schema and need to ramp up, check out Understanding XML Schema. Once you have your XML Schema types defined, the next step is to define the logical messages that will constitute your operations.

Messages

The WSDL message element defines an abstract message that can serve as the input or output of an operation. Messages consist of one or more part elements, where each part is associated with either an **element** (when using document style) or a **type** (when using RPC style). The basic structure of a message definition is as follows (***** means zero or more and **?** means optional):

```
<definitions .... >
    <message name="nmtoken"> *
        <part name="nmtoken" element="qname"? type="qname"?/> *
    </message>
</definitions>
```

The messages and parts must be named making it possible to refer to them from elsewhere in the WSDL definition. If you're defining an RPC style service, the message parts represent the method's parameters. In this case, the name of the part becomes the name of an element in the concrete message and its structure is determined by the supplied type attribute. If you're defining a document style service, the parts simply refer to the XML elements that are placed within the body (referenced by the **element** attribute). The following example contains several message definitions that refer to elements by name:

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/"
>
   ...
   <message name="AddMessage">
      <part name="parameter" element="ns:Add"/>
   </message>
   <message name="AddResponseMessage">
      <part name="parameter" element="ns:AddResponse"/>
   </message>
   <message name="SubtractMessage">
      <part name="parameter" element="ns:Subtract"/>
   </message>
   <message name="SubtractResponseMessage">
      <part name="parameter" element="ns:SubtractResponse"/>
   </message>
   ...
</definitions>
```

Although the message parts typically refer to XML types or elements, they can also refer to non-XML types. This allows WSDL to represent a wide range of messages that contain a mixture of data formats, as is the case with multi-part MIME.

Note The message and type definitions in WSDL are considered to be abstract definitions. This means you don't know how they'll appear in the concrete message format until you've applied a binding to them. For example, if you use one abstract message with two different bindings, it's possible that the two concrete messages will look different. Only with 'literal' bindings are the abstract definitions guaranteed to accurately describe the concrete message format. See the Bindings section for more details.

Interfaces (portTypes)

The WSDL **portType** element defines a group of operations, also known as an interface in most environments. Unfortunately, the term "portType" is quite confusing so you're better off using the term "interface" in conversation. WSDL 1.2 has already removed "portType" and replaced it with "interface" in the current draft of the language.

A portType element contains zero or more operation elements. The basic structure of a portType is as follows (* means zero or more):

```
<definitions .... >
    <portType name="nmtoken">
        <operation name="nmtoken" .... /> *
    </portType>
</definitions>
```

Each portType must be given a unique name making it possible to refer to it from elsewhere in the WSDL definition. Each operation element contains a combination of **input** and **output** elements, and when you have an output element you can also have a **fault** element. The order of these elements defines the message exchange pattern (MEP) supported by the given operation.

For example, an input element followed by an output element defines a request-response operation, while an output element followed by an input element defines a solicit-response operation. An operation that only contains an input element defines a one-way operation, while an operation that only contains an output element defines a notification operation. Table 2 describes the four MEP primitives defined by WSDL.

Table 2. Message Exchange Patterns

MEP	Description
One-way	The endpoint receives a message.
Request-response	The endpoint receives a message and sends a correlated message.
Solicit-response	The endpoint sends a message and receives a correlated message.
Notification	The endpoint sends a message.

The input, output, and fault elements used in an operation must refer to a message definition by name. The following example defines a **portType** named **MathInterface** that consists of four math operations: **Add**, **Subtract**, **Multiply**, and **Divide**.

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/"
>
   ...
   <portType name="MathInterface">
      <operation name="Add">
         <input message="y:AddMessage"/>
         <output message="y:AddResponseMessage"/>
      </operation>
      <operation name="Subtract">
         <input message="y:SubtractMessage"/>
         <output message="y:SubtractResponseMessage"/>
      </operation>
      <operation name="Multiply">
         <input message="y:MultiplyMessage"/>
         <output message="y:MultiplyResponseMessage"/>
      </operation>
      <operation name="Divide">
         <input message="y:DivideMessage"/>
         <output message="y:DivideResponseMessage"/>
      </operation>
   </portType>
    ...
</definitions>
```

A portType is still considered an abstract definition because you don't know how its messages are represented on the wire until you apply a binding.

Bindings

The WSDL **binding** element describes the concrete details of using a particular portType with a given protocol. The binding element contains several extensibility elements as well as a WSDL **operation** element for each operation in the portType it's describing. The basic structure of a binding element is as follows (***** means zero or more and **?** means optional):

```
<wsdl:definitions .... >
    <wsdl:binding name="nmtoken" type="qname"> *
        <-- extensibility element providing binding details --> *
        <wsdl:operation name="nmtoken"> *
           <-- extensibility element for operation details --> *
           <wsdl:input name="nmtoken"? > ?
               <-- extensibility element for body details -->
           </wsdl:input>
           <wsdl:output name="nmtoken"? > ?
               <-- extensibility element for body details -->
           </wsdl:output>
           <wsdl:fault name="nmtoken"> *
               <-- extensibility element for body details -->
           </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
</wsdl:definitions>
```

A binding must be a given a unique name so you can refer to it from elsewhere in the WSDL definition. The binding must also specify which portType it's describing through the type attribute. For example, the following binding is called **MathSoapHttpBinding** and it describes the concrete details for the **MathInterface** portType:

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/"
>
   ...
   <binding name="MathSoapHttpBinding" type="y:MathInterface">
          ... <-- extensibility element -->
      <operation name="Add">
         ... <-- extensibility element -->
         <input>
            ... <-- extensibility element -->
         </input>
         <output>
            ... <-- extensibility element -->
         </output>
      </operation>
      ...
    </binding>
    ...
</definitions>
```

The WSDL binding element is generic. It simply defines the framework for describing binding details. The actual binding details are provided using extensibility elements. This architecture allows WSDL to evolve over time since any element can be used in the predefined slots. The WSDL specification provides some binding elements for describing SOAP bindings, although they're in a different namespace. The following example illustrates a SOAP/HTTP binding for the **MathInterface** portType:

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/"
>
   ...
   <binding name="MathSoapHttpBinding" type="y:MathInterface">
      <soap:binding style="document"
             transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="Add">
         <soap:operation
                  soapAction="http://example.org/math/#Add"/>
         <input>
            <soap:body use="literal"/>
         </input>
         <output>
            <soap:body use="literal"/>
         </output>
```

```
      </operation>
          ...
   </binding>
   ...
</definitions>
```

The **soap:binding** element indicates that this is a SOAP 1.1 binding. It also indicates the default style of service (possible values include **document** or **rpc**) along with the required transport protocol (HTTP in this case). The **soap:operation** element defines the **SOAPAction** HTTP header value for each operation. And the **soap:body** element defines how the message parts appear inside of the SOAP **Body** element (possible values include **literal** or **encoded**). There are other binding-specific details that can be specified this way.

Using **document** style in SOAP indicates that the body will contain an XML document, and that the message parts specify the XML elements that will be placed there. Using **rpc** style in SOAP indicates that the body will contain an XML representation of a method call and that the message parts represent the parameters to the method.

The **use** attribute specifies the encoding that should be used to translate the abstract message parts into a concrete representation. In the case of 'encoded', the abstract definitions are translated into a concrete format by applying the SOAP encoding rules. In the case of 'literal', the abstract type definitions become the concrete definitions themselves (they're 'literal' definitions). In this case, you can simply inspect the XML Schema type definitions to determine the concrete message format. For example, the **Add** operation for the above **document/literal** binding looks like this on the wire:

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
>
   <SOAP-ENV:Body>
      <m:Add xmlns:m="http://example.org/math/types/">
         <x>3.14159265358979</x>
         <y>3.14159265358979</y>
      </m:Add>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice that the SOAP **Body** simply contains an instance of the **Add** element defined in the schema —this is what makes document/literal so attractive. Now let's see what the message would look like using an **rpc/encoded** binding. The following WSDL fragment contains a revised rpc/encoded binding and the message definitions use **type** instead of **element**:

```
   ...
   <message name="AddMessage">
      <part name="parameter" type="ns:MathInput"/>
   </message>
   <message name="AddResponseMessage">
      <part name="parameter" type="ns:MathOutput"/>
   </message>
   ...
   <binding name="MathSoapHttpBinding" type="y:MathInterface">
      <soap:binding style="rpc"
             transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="Add">
         <soap:operation
```

```
                    soapAction="http://example.org/math/#Add"/>
         <input>
            <soap:body use="encoded"/>
         </input>
         <output>
            <soap:body use="encoded"/>
         </output>
      </operation>
          ...
   </binding>
```

With these definitions in place, the **Add** operation looks much different as you can see here:

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:m0="http://example.org/math/types/"
>
   <SOAP-ENV:Body>
      <m:Add xmlns:m="http://example.org/math/">
         <parameter xsi:type="m0:MathInput">
            <x xsi:type="xsd:double">3.14159265358979</x>
            <y xsi:type="xsd:double">3.14159265358979</y>
         </parameter>
      </m:Add>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Using literal definitions is much cleaner and easier for the tools to get right. Using the encoding rules has led to significant interoperability problems across toolkits. It also leads to weird situations like not being able to validate the wire-level message against the original schema definition (since it's abstract and not a true representation of the message). To help alleviate the confusion and facilitate better interoperability, the WS-I prohibits the use of encodings altogether, including the SOAP encoding, in the Basic Profile 1.0. This means that only literal definitions may be used if you care about Basic Profile 1.0 compliance. For more information on this topic, check out The Argument Against SOAP Encoding.

The most common combination of SOAP **style/use** attributes is **document/literal** (what I used in the first example above). It's the default in most toolkits today and the one that comes with the fewest interoperability issues. The second most common combination is **rpc/encoded** but now that the WS-I has banned the use of 'encoded', it's no longer a viable option. The only other combination that exists is **rpc/literal**. Check out RPC/Literal and Freedom of Choice for a discussion on rpc/literal and why document/literal is superior.

In addition to the SOAP binding, the WSDL specification defines two other bindings: one for HTTP GET & POST and another for MIME. Check out the examples in the specification for more details.

Services

The WSDL **service** element defines a collection of **ports**, or **endpoints**, that expose a particular binding. The basic structure of the service element is as follows:

```
<definitions .... >
    <service .... > *
        <port name="nmtoken" binding="qname"> *
           <-- extensibility element defines address details -->
        </port>
    </service>
</definitions>
```

You must give each port a name and assign it a **binding**. Then, within the port element, you use an **extensibility** element to define the address details specific to the binding. For example, the following sample defines a service called **MathService** that exposes the **MathSoapHttpBinding** at the **http://localhost/math/math.asmx** URL:

```
<definitions
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/"
>
   ...
   <service name="MathService">
       <port name="MathEndpoint" binding="y:MathSoapHttpBinding">
         <soap:address
            location="http://localhost/math/math.asmx"/>
       </port>
   </service>
</definitions>
```

The service element completes the WSDL definition. You could now take this definition and easily generate code that implements the complete interface in your language of choice. I've provided the complete WSDL definition that we've been walking through in Appendix A.

WSDL Editors

I generated this WSDL file using XML Spy 5.0 (see Figure 6). There are several editors available today that nearly make authoring WSDL pleasant. You can check out some reviews from one of my previous articles in MSDN Magazine.

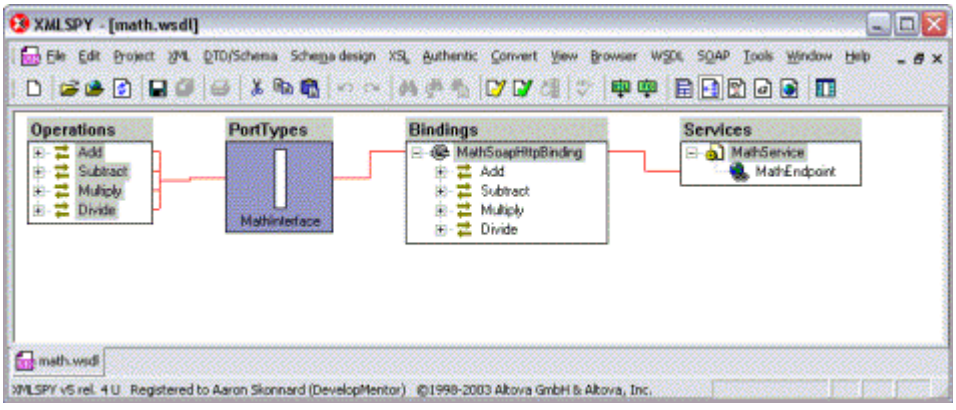

Figure 6. MathService WSDL definition generated using XML Spy 5.0

Where Are We?

WSDL builds on XML Schema by making it possible to fully describe Web services in terms of messages, operations, interfaces (portTypes), bindings, and service endpoints. WSDL definitions make it possible to generate code that implements the given interface, on either the client or the server, making Web services accessible to the masses. WSDL is capable of much more than what I've described here but if you truly care about interoperability, you should limit yourself to the WS-I Basic Profile 1.0 guidelines that we've covered here.

References

Essential XML Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More by Aaron Skonnard and Martin Gudgin, DevelopMentor Books, Addison-Wesley Pub. Co., 2001

Web Services Description Language (WSDL) 1.1

Web Services Description Language (WSDL) Version 1.2 Part 1: Core Language

WS-I Basic Profile Version 1.0

XML Schema Part 1: Structures

XML Schema Part 2: Datatypes

Appendix: WSDL

```
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:y="http://example.org/math/"
  xmlns:ns="http://example.org/math/types/"
  targetNamespace="http://example.org/math/">
   <types>
     <xs:schema targetNamespace="http://example.org/math/types/"
       xmlns="http://example.org/math/types/"
       elementFormDefault="unqualified" attributeFormDefault="unqualified">
         <xs:complexType name="MathInput">
            <xs:sequence>
               <xs:element name="x" type="xs:double"/>
               <xs:element name="y" type="xs:double"/>
            </xs:sequence>
         </xs:complexType>
         <xs:complexType name="MathOutput">
            <xs:sequence>
               <xs:element name="result" type="xs:double"/>
            </xs:sequence>
         </xs:complexType>
         <xs:element name="Add" type="MathInput"/>
         <xs:element name="AddResponse" type="MathOutput"/>
```

```
      <xs:element name="Subtract" type="MathInput"/>
      <xs:element name="SubtractResponse" type="MathOutput"/>
      <xs:element name="Multiply" type="MathInput"/>
      <xs:element name="MultiplyResponse" type="MathOutput"/>
      <xs:element name="Divide" type="MathInput"/>
      <xs:element name="DivideResponse" type="MathOutput"/>
   </xs:schema>
</types>
<message name="AddMessage">
   <part name="parameters" element="ns:Add"/>
</message>
<message name="AddResponseMessage">
   <part name="parameters" element="ns:AddResponse"/>
</message>
<message name="SubtractMessage">
   <part name="parameters" element="ns:Subtract"/>
</message>
<message name="SubtractResponseMessage">
   <part name="parameters" element="ns:SubtractResponse"/>
</message>
<message name="MultiplyMessage">
   <part name="parameters" element="ns:Multiply"/>
</message>
<message name="MultiplyResponseMessage">
   <part name="parameters" element="ns:MultiplyResponse"/>
</message>
<message name="DivideMessage">
   <part name="parameters" element="ns:Divide"/>
</message>
<message name="DivideResponseMessage">
   <part name="parameters" element="ns:DivideResponse"/>
</message>
<portType name="MathInterface">
   <operation name="Add">
      <input message="y:AddMessage"/>
      <output message="y:AddResponseMessage"/>
   </operation>
   <operation name="Subtract">
      <input message="y:SubtractMessage"/>
      <output message="y:SubtractResponseMessage"/>
   </operation>
   <operation name="Multiply">
      <input message="y:MultiplyMessage"/>
      <output message="y:MultiplyResponseMessage"/>
   </operation>
   <operation name="Divide">
      <input message="y:DivideMessage"/>
      <output message="y:DivideResponseMessage"/>
   </operation>
</portType>
<binding name="MathSoapHttpBinding" type="y:MathInterface">
   <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http"/>
   <operation name="Add">
      <soap:operation soapAction="http://example.org/math/#Add"/>
      <input>
         <soap:body use="literal"/>
      </input>
      <output>
         <soap:body use="literal"/>
      </output>
```

```
      </operation>
      <operation name="Subtract">
         <soap:operation soapAction="http://example.org/math/#Subtract"/>
         <input>
            <soap:body use="literal"/>
         </input>
         <output>
            <soap:body use="literal"/>
         </output>
      </operation>
      <operation name="Multiply">
         <soap:operation soapAction="http://example.org/math/#Multiply"/>
         <input>
            <soap:body use="literal"/>
         </input>
         <output>
            <soap:body use="literal"/>
         </output>
      </operation>
      <operation name="Divide">
         <soap:operation soapAction="http://example.org/math/#Divide"/>
         <input>
            <soap:body use="literal"/>
         </input>
         <output>
            <soap:body use="literal"/>
         </output>
      </operation>
   </binding>
   <service name="MathService">
      <port name="MathEndpoint" binding="y:MathSoapHttpBinding">
         <soap:address location="http://localhost/math/math.asmx"/>
      </port>
   </service>
</definitions>
```