

## VW 7.1 SOAP Tutorial (Draft)

By Roger Whitney  
whitney@cs.sdsu.edu

Service Class to Server.....	2
Hello World Example .....	2
Step 1. Creating the Service Class .....	2
Step 2. Marking the Service Methods .....	3
Step 3. Generate the WSDL for the server. ....	4
Step 4. Using the WSDL to generate the server (and client). ....	4
Step 5. Running the server and client. ....	4
Example With Parameters .....	5
Step 1. Creating the Service Class .....	5
Step 2. Marking the Service Methods .....	6
Step 3. Generate the WSDL for the server. ....	10
Step 4. Using the WSDL to generate the server (and client). ....	11
Step 5. Running the server and client. ....	12
SOAP and Smalltalk Types.....	13
List of Automatic Conversions .....	13
Collections as Parameters.....	15
Things I wish I know How to do .....	15
Generating SOAP Servers and Clients from WSDL.....	16
Three Clients.....	18
Some Useful Items.....	20

This is a short tutorial on how to create a SOAP server in VisualWorks 7.1. SOAP is one of a number of systems that allow a client on one machine to call a method on a remote machine. SOAP converts these remote method calls into XML and sends them via HTTP to the server on the remote machine. Since SOAP uses XML the client and server can be in different languages.

VW 7.1 provides several different ways to generate SOAP clients and servers. One can start with Smalltalk class that performs operations and generate the server and client. One can start with WSDL (Web Service Description Language) that defines the service to be provided by a SOAP server. With this WSDL VW tools will generate SOAP client, server and method stubs that will perform the operations. VW 7.1 supports two different types of SOAP clients: Opentalk based and non-opentalk based. The non-opentalk base client can be ad-hoc using the WSDL to dynamically generate calls or hard coded to a particular server.

The tutorial first shows how to start with a Smalltalk service class and generate the WSDL and clients and servers.

## ReadMe First

Make sure to load the following parcels before running the tutorial: Opentalk-SOAP, WebServicesServer, and WebServicesClient. The tutorial contains some Smalltalk code showing how to perform some tasks. This code runs in a Workspace, so variables are not declared and namespaces of existing classes are not imported. The example classes are generated in the Smalltalk namespace.

## Service Class to Server

The basic steps in generating clients and server from a Smalltalk service class are:

1. Create a service class that is a class that performs the operations we want the SOAP server to perform
2. Add special pragmas to methods in the service class to allows us to generate the WSDL for the SOAP server
3. Generate the WSDL for the server.
4. Using the WSDL to generate the server (and client).
5. Running the server and client.

First we will do the standard Hello World example then a more complex example.

## Hello World Example Step 1. Creating the Service Class

First we need to create a class that has a method that returns 'Hello World'. Here is the class I used:

'From VisualWorks® NonCommercial, Release 7 of March 21, 2003 on June 30, 2003 at 7:33:52 pm'

```
Smalltalk defineClass: #Greetings
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: "
  classInstanceVariableNames: "
  imports: "
  category: ""Web Services""!
```

```
"-----"!
```

```
!Greetings methodsFor: 'public api'
```

```
helloWorld
  ^Hello World' !
```

## Step 2. Marking the Service Methods

### Basic Pragmas

We need to mark the methods in the service class that we want to expose to SOAP clients. This is done by adding special pragmas to the methods. The pragmas are used to generate the WSDL for the service. Since these pragmas are no standard we have to register them with the class. This is done by adding the following class method. This class method is commonly put in the "ws pragmas" protocol.

```
Greetings class>>operationPragmas
  <pragmas: #instance>
  ^OrderedCollection new
    add: #addException:type;;
    add: #addParameter:type;;
    add: #documentation;;
    add: #operationName;;
    add: #result;;
    yourself
```

Once this is done we can add the pragmas to the helloWorld method. Edit the method so it looks like:

```
Greetings>>helloWorld
  <operationName: #HelloWorld>
  <documentation: #'Returns Hello World'>
  <result: #String>

  ^Hello World'
```

### Step 3. Generate the WSDL for the server.

These steps will be discussed a bit more in the complex example. For now just do them.

```

serviceClass := Greetings.
wsdlBuilder := WsdlBuilder new.
wsdlBuilder useRPC.
wsdlBuilder
    buildFromService: serviceClass
    classNamespace: 'Smalltalk'
    targetNamespace: 'urn:Hellonamespace'.
wsdlBuilder
    setPortAddress: 'http://localhost:8889/Hello'
    forBindingNamed: serviceClass name asString
    wsdlServiceNamed: serviceClass name asString.

stream := (String new: 2048) writeStream.
wsdlBuilder printSpecWithSmalltalkBindingOn: stream.
wsdl := stream contents.

WsdlBinding loadWsdlBindingFrom: wsdl readStream.

```

### Step 4. Using the WSDL to generate the server (and client).

```

classBuilder := WsdlClassBuilder readFrom: wsdl readStream.
classBuilder
    package: 'SampleServicePackage';
    opentalkServerName: 'HelloWorldServer';
    serviceClasses: #( Greetings);
    createOpentalkServerClass;
    createOpentalkClientClasses.

```

This code will put the client and server classes in the package 'SampleServicePackage'.

### Step 5. Running the server and client.

```

"Start the server and client"
opentalkServer := HelloWorldServer new startServers.
client := OpentalkClientGreetings new start.

"Using the client"
client helloWorld.

```

Don't forget to stop the client and server when done. Opentalk clients use a RequestBroker which does listen on a port. Seems a bit odd for a client, but does require it to be stopped.

```

opentalkServer stopServers.
client stop.

```

## Example With Parameters

### Step 1. Creating the Service Class

Here is the definition of the class we will use. It is a normal Smalltalk class. Not all the methods in the class will be accessed directly by the SOAP server. We will mark the methods

----

'From VisualWorks® NonCommercial, Release 7 of March 21, 2003 on June 7, 2003 at 9:20:37 am!'

```
Smalltalk defineClass: #SampleService
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'accessCount '
  classInstanceVariableNames: "
  imports: "
  category: "'Web Services'"!

"-----"!

!SampleService methodsFor: 'accessing'!

accessed
  accessCount ifNil: [accessCount := 0].
  accessCount :=accessCount + 1! !

+ aNumber
  "This method is here to point out that you can not use a binary message as a SOAP method"
  ^aNumber + 2!

!SampleService methodsFor: 'public api'!

accessCount
  self accessed.
  ^accessCount!

add: anInteger to: anotherInteger
  self accessed.
  ^anInteger + anotherInteger!

complexIncrease: aCounter
  self accessed.
  ^aCounter increase!
```

```
greetings
  self accessed.
  ^'Hello World'!
```

```
increase: anInteger
  self accessed.
  ^anInteger + 1!!
```

----

There are a number different methods to show methods with different amount of arguments and different types of arguments: basic types and types you define. Each one of these must be handled slightly differently. The class uses the class Counter. Here is its definition.

----

'From VisualWorks® NonCommercial, Release 7 of March 21, 2003 on June 5, 2003 at 10:25:49 am!'

```
Smalltalk defineClass: #Counter
  superclass: #{Core.Object}
  indexedType: #none
  private: false
  instanceVariableNames: 'count '
  classInstanceVariableNames: "
  imports: "
  category: "'Web Services'!"
```

```
!Counter methodsFor: 'accessing'!
```

```
count
  ^count!
```

```
count: anInteger
  count := anInteger!
```

```
increase
  count :=count + 1!!
```

----

## Step 2. Marking the Service Methods

### Basic Pragmas

We need to mark the methods in the service class that we want to expose to SOAP clients. This is done by adding special pragmas to the methods. The pragmas are used to generate the WSDL for the service. Since these pragmas are no standard we have to register them with the class. This is done by adding the following class method. This class method is commonly put in the "ws pragmas" protocol.

SampleService class>>operationPragmas

```
<pragmas: #instance>
^OrderedCollection new
  add: #addException:type;;
  add: #addParameter:type;;
  add: #documentation;;
  add: #operationName;;
  add: #result;;
  yourself
```

If you forget this step later you will get an error message stating you have unrecognized pragmas.

Now to mark the methods we wish to expose to the SOAP client. The convention is to place all such methods in the protocol "public api".

First lets mark the method greetings. Edit the method to be:

```
SampleService>>greetings
  <operationName: #Greetings>
  <documentation: #Returns Hello World'>
  <result: #String>

  self accessed.
  ^'Hello World'
```

We will look at the pragmas one at a time.

"<operationName: #Greetings>" Clearly this pragma provides the name of the method. Note that the pragma starts the name with a capital letter. This seems to be the way to do it. I don't know if or why this is the case. I do not know what happens if you use the Smalltalk convention and use lower case.

"<documentation: #Returns Hello World'>" This pragam is optional. Is adds the comment to the WSDL, which can be useful to people trying to write clients to your server from the WSDL.

"<result: #String>" This pragma indicates the return type of the method. More about return types later.

The impatient can skip to steps 3-5 to test this method. Actually it is a good idea to do this, as it will give you more practice doing it.

With this information adding the pragmas to accessCount is easy. We get:

```

SampleService>>accessCount
  <operationName: #AccessCount>
  <documentation: #Returns the number of times a SOAP method has been called'>
  <result: #Integer>

self accessed.
^accessCount

```

The point of this method is to allow you to test how the server handles the state of the service object. That is does the server create a new object for each request or does it create one service object and reuse it.

Try doing steps 3-5 to find the answer.

### Handling Arguments

More interesting is handling methods with arguments. Here are the increase and add:to: methods

```

SampleService>>increase: anInteger
  <operationName: #Increase>
  <documentation: #'Adds one to the argument'>
  <addParameter: #increase type: #Integer>
  <result: #Integer>

self accessed.
^anInteger + 1

```

```

add: anInteger to: anotherInteger
  <operationName: #Add>
  <documentation: #'Adds the two arguments'>
  <addParameter: #add type: #Integer>
  <addParameter: #to type: #Integer>
  <result: #Integer>

self accessed.
^anInteger + anotherInteger

```

We have added the pragma describing the argument for the methods. The pragma contains the name and the type of the argument. The order of the pragmas is important, since it will be used to generate the order of the parameters in the client and in the server method. Note that name of the each argument is the same as the corresponding keyword of the method. For the first argument of a keyword message one can use any name one wants for the parameter name. When Smalltalk code (client and server) is generated from the pragma (or the WSDL generated from the pragmas) the operation name is always used as the first keyword of the message. For all other arguments the name of the argument becomes the corresponding keyword of the message. Keep in mind that one can use these pragmas to generate the WSDL, which is used

to generate the server and client, or one can start with the WDSL and generate the client, server and the service class. Also others may use the WDSL to generate clients in other languages. A Java tool might generate a method stub like

```
Integer Add( Integer add, Integer to){};
```

This may have an impact on how you want to name the arguments, particularly the first one.

You might wish to go through steps 3-5 to test these messages out using the server.

### Handling New Types

The next method is complexIncrease, which has a Counter object as a parameter. Here is the method with the pragma:

```
complexIncrease: aCounter
  <operationName: #ComplexIncrease>
  <documentation: #Increases the value of aCounter by one'>
  <addParameter: #aCounter type: #Counter>
  <result: #Counter>

  self accessed.
  ^aCounter increase
```

While this method does not use any new pragmas it does make us deal with types. SOAP uses XML Schema types, which defines a set of basic types and allows creation of new types. For an overview of XML Schema types see XML Schema Part 0: Primer at <http://www.w3.org/TR/xmlschema-0/>. For a list of the basic (or Simple) types in XML Schema see <http://www.w3.org/TR/xmlschema-0/#CreatDt>. Any parameter or return value sent between a SOAP client and server must be converted to/from XML. VW like most SOAP systems handles converting basic XML Schema types. To pass other types between a client and server we need to:

- a. Register a marshaler for the type in the image
- b. Make sure that the WSDL contains the XML schema definitions for the type so other clients know about the new type.

To do this in VW we need to add pragmas to the class that will map to the new type. Each instance variable of the class needs the standard accessor methods. In each setter method one adds a pragma defining the type of the argument. One can do this by hand or use the WsdIClassBuilder to do it for you.

Since complexIncrease: uses a new type (Counter) we need to add a pragma to the counter class. The pragma will be used to generate WSDL with XML schema definitions for the type and XML that will be used to register a marshaler for the type in VW. The following method will open an editor on the setter methods in the Counter class. Edit the count: method so it is as below.

```
WsdIClassBuilder setTypePragmasForClasses:
  (OrderedCollection
   with: Counter).
```

Doing this results in one new method and the setter method modified as:

```
Counter class>>typePragmas
"Generated by WS Tool on #(June 2, 2003 5:19:19 pm)"
<pragmas: #instance>
^OrderedCollection new
  add: #addAttribute:type;;
  yourself
```

```
Counter>>count: anInteger
"Generated by WS Tool on #(June 6, 2003 7:06:04 pm)"
<addAttribute: #( #count ) type: #Integer>
count := anInteger
```

Note that count is not optional, so you need to remove the #optional attribute. Also the WsdIClassBuilder uses the type string, which must be changed to Integer in this case.

Once you have seen the result it is clear how to perform the modifications by hand.

### Binary Messages

The VW tools will not generate WSDL for binary messages. It is not clear if they are allowed in SOAP. Even if they were some languages would not support them (Java), so clients in these languages could not

#### Step 3. Generate the WSDL for the server.

The following code will generate the WSDL and register it where it is needed.

```
serviceClass := SampleService.
wsdlBuilder := WsdIBuilder new.

"Generate RPC style soap messages"
wsdlBuilder useRPC.

"The first two arguments are clear. I am not sure what difference the third makes"
wsdlBuilder
  buildFromService: serviceClass
  classNamespace: 'Smalltalk'
  targetNamespace: 'urn:SampleServicenamespace'.
```

"The first argument is the address of the server. Local host works for tests and examples, but needs to have an actual url for a real server. The end of the url 'SampleService' can be anything you want. "

```
wSDLBuilder
  setPortAddress: 'http://localhost:8889/SampleService'
  forBindingNamed: serviceClass name asString
  wSDLServiceNamed: serviceClass name asString.
```

"Create the wSDL"

```
stream := (String new: 2048) writeStream.
wSDLBuilder printSpecWithSmalltalkBindingOn: stream.
wSDL := stream contents.
```

"Register the WSDL with the image. Do this on both the client and server image"

```
WSDLBinding loadWSDLBindingFrom: wSDL readStream.
```

"Register the standard XML marshaller. This is only needed if you are passing new types between the client and the server. Needed in both the client and server image"

```
smalltalkBinding := wSDL readStream
  upToAll: '<xmlToSmalltalkBinding';           "No typo here, the tag has attributes"
  throughAll: '</xmlToSmalltalkBinding>'.
```

```
smalltalkBinding isEmpty ifFalse: [XMLObjectBinding loadFrom: smalltalkBinding readStream]
```

#### **Step 4. Using the WSDL to generate the server (and client).**

Generating the server and client is easy. Here is the code. The package is the name of the Smalltalk package you want the classes to be placed in.

```
classBuilder := WSDLClassBuilder readFrom: wSDL readStream.
classBuilder
  package: 'SampleService';
  opentalkServerName: 'SampleServiceServer';
  serviceClasses: #( SampleService);           "The class already exists, so no need to create it"
  createOpentalkServerClass;
  createOpentalkClientClasses.
```

There is a bug in the generation of methods that have two arguments of the same type. You need to hand edit all such method in the generated client. The generated code in the client is below. Note that the two arguments have the same name. There is not such method in the server. Be sure to edit all such methods in you client before you use it, otherwise you will get rather odd results.

```
OpentalkClientSampleService>>add: aInteger to: aInteger
"Generated by WS Tool on #(June 6, 2003 7:32:19 pm)"
"operationName: #Add"
"documentation: #Adds the two arguments"
"addParameter: #add type: #Integer"
"addParameter: #to type: #Integer"
"result: #Integer"
```

```
^proxy add: aInteger to: aInteger
```

### Step 5. Running the server and client.

Once just needs to start the server and client and call the methods you are interested in. When you are testing the server the following method can be useful. It will stop all existing Opentalk servers and clients. It prevents problems that occur when you try to start a server on a port that is already in use.

```
RequestBroker allInstances do: [ :each | each stop].
```

```
"Start the client and server"
opentalkServer := SampleServiceServer new startServers.
client := OpentalkClientSampleService new start.
```

```
client greetings.
client increase: 5.
client accessCount.
client add: 2 to: 3.
aCounter :=Counter new count: 0.
client complexIncrease: aCounter
```

```
input :=Foo new count: 2.
client complexAdd: input
```

```
opentalkServer stopServers.
client stop.
```

### **SOAP and Smalltalk Types**

In order to send arguments and return values between a client and a server they are converted to XML Schema types encoded in XML. This means all values need to be mapped to a XML Schema type. For an overview of XML Schema types see XML Schema Part 0: Primer at <http://www.w3.org/TR/xmlschema-0/>. For a list of the basic (or Simple) types in XML Schema see <http://www.w3.org/TR/xmlschema-0/#CreatDt>.

There are several important issues relating to types:

1. Range and size restrictions of types
2. Which Smalltalk classes are automatically mapped to existing XML Schema types and which classes do we have to generate new XML Schema types for
3. Adding new Types

#### **Range and size restrictions**

In general there is no restriction on the range or size of such XML Schema types as Integer or String. However, the client or server you are talking to may impose a range or size restriction. In particular Smalltalk integers (if one includes the subclass of Integer) have no size restriction where as most other languages do. XML Schema does not seem to impose any restriction on Integers. So any size restrictions on integers come from the language of the client/server your Smalltalk code is interacting with.

#### **List of Automatic Conversions**

The following table contains a list of existing Smalltalk classes that are automatically mapped to XML Schema types. This means that you can use any of these types as parameters or return values without having to generate a type and register a marshaler (as was done for Counter in the example above.) These values were determined by looking at `BindingBuilder class>>initializeSerializationBlocks` and trying all types listed.

## Existing Mapping From Smalltalk classes to XML

<b>Smalltalk Class</b>	<b>XML Schema type</b>
Boolean	boolean
ByteSymbol	string
Character	string
Date	date
Double	double
FixedPoint	decimal
Float	float
Integer	int
LargeInteger	Not Handled
SmallInteger	short
String	string
Time	time
Timestamp	dateTime
URI	Not Handled

The following table contains the mapping for the simple XML Schema types and a few common XML types. These values were determined experimentally trying all types listed to see what Smalltalk class they were mapped to.

## Existing Mapping from XML Schema to Smalltalk Class

<b>XML Schema Type</b>	<b>Smalltalk Class</b>
anyType	Object
anyURI	URI
base64	ByteArray
base64Binary	Object
boolean	Boolean
byte	Integer
date	Date
dateTime	Timestamp
decimal	FixedPoint
double	Double
duration	String
float	Float
gDay	String
gMonth	String
gMonthDay	String
gYear	String
gYearMonth	String
hexBinary	Object
id	Not Handled
idrefs	Not Handled
int	Integer

integer	Integer
language	String
long	LargeInteger
NCName	Object
negativeInteger	Integer
NMTOKENS	Object
nonNegativeInteger	Integer
nonPositiveInteger	Integer
normalizedString	String
positiveInteger	Integer
QName	Object
short	SmallInteger
string	String
time	Time
unsignedByte	Integer
unsignedInt	LargeInteger
unsignedLong	LargeInteger
unsignedShort	SmallInteger

### Collections as Parameters

Collection can be used as parameters and return types of a SOAP method. The following method gives the syntax for returning an array. A VW Smalltalk SOAP client converts the SOAP return type to an `OrderedCollection`. It appears that you can send any subclass of `SequenceableCollection` and the other side will convert it to an `OrderedCollection` (assuming VW Smalltalk on the other side).

greetings

```
<operationName: #greetings>
<result: #( #Collection #String) >
```

```
^#('Hello' 'World')
```

### Things I wish I know How to do

1. How to mark a parameter or return value of a service method as a dictionary
2. Automatically provide the WSDL of a service via http

## Generating SOAP Servers and Clients from WSDL

In creating SOAP servers in VW we can start either WSDL or a service class (a class that provides the service we wish to make available via the SOAP server). Here we will start with the WSDL. This is more common when creating clients for existing servers. SOAP servers provide the WSDL, which describes the operations you can call on the server. I created the WSDL by modifying the WSDL from a different server. Perhaps those with a better grasp of and more experience with SOAP & WSDL than I have find it easy to generate valid WSDL for a server and would prefer this method over generating WSDL from a service class.

The steps taken here are very similar to the steps done before, so there will be less discussion.

"Now on with the example. Define a WSDL schema:"

```
helloWorldSchema := '<?xml version="1.0"?>
<definitions name="HelloWorldService"
xmlns:tns="http://localhost:9999/HelloWorldService.wsdl"
targetNamespace="http://localhost:9999/HelloWorldService.wsdl"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <message name="HelloWorldRequest">
  </message>
  <message name="HelloWorldResponse">
    <part name="return" type="xsd:string"/>
  </message>
  <portType name="HelloWorld">
    <operation name="Hello">
      <documentation>Returns Hello World string</documentation>
      <input message="tns:HelloWorldRequest" name="HelloWorldRequestFoo"/>
      <output message="tns:HelloWorldResponse" name="HelloWorldResponseFoo"/>
    </operation>
  </portType>
  <binding name="HelloWorld" type="tns:HelloWorld">
    <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="Hello">
      <soap:operation soapAction="urn:xmethodsHelloWorld#HelloWorld"/>
      <input>
        <soap:body use="encoded" namespace="urn:xmethodsHelloWorld"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
      </input>
      <output>
        <soap:body use="encoded" namespace="urn:xmethodsHelloWorld"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
      </output>
    </operation>
```

```

</binding>
<service name="HelloWorldService">
  <documentation>Just returns Hello World</documentation>
  <port name="HelloWorldExample" binding="tns:HelloWorld">
    <soap:address location="http://localhost:9998/HelloWorldAddress"/>
  </port>
</service>
</definitions>'.

```

"Now make sure the binding is loaded"

```
WSDLBinding loadWSDLBindingFrom: helloWorldSchema readStream.
```

"create the client and server classes"

```

builder := WSDLClassBuilder readFrom: helloWorldSchema readStream.
builder
  package: 'HelloWorldPackage';
  opentalkServerName: 'OpentalkHelloWorldServer';
  createClasses;          "Note the change from before"
  createOpentalkServerClass;
  createOpentalkClientClasses.

```

The above creates the classes:

- HelloWorld - the application class for the server
- OpentalkHelloWorldServer - the server for HelloWorld
- OpentalkClientHelloWorldExample - client with a broker
- HelloWorldExampleClient - another client which is subclass of WDSLClient

Now go edit the method HelloWorld>>hello so it returns Hello World

"Make sure all brokers (servers and clients) are stopped"

```
RequestBroker allInstances do: [ :ea | ea stop].
```

"start the server"

```
opentalkServer := OpentalkHelloWorldServer new startServers.
```

"Use the client to connect to the server"

```
client := OpentalkClientHelloWorldExample new start.
client hello inspect.
```

"Using the other client"

```
client2 := HelloWorldExampleClient new.
client2 hello.
```

```
opentalkServer stopServers.
client stop
```

### Three Clients

VW 7.1 supports three different clients. First one can use the class WsdIClient. Here is an example.

```

| babelClient |
babelClient := WsdIClient url: 'http://www.xmethods.net/sd/BabelFishService.wsdl'.
babelClient executeSelector: #BabelFish args: #('en_de' 'this is a test')

```

One problem with this is you have to generate the method name and argument list. The following will create a subclass of WsdIClient, BabelFishPortClient (which is put into the none package), which contains one method for each SOAP method on the server.

```

classBuilder := WsdClassBuilder
    readFrom: 'http://www.xmethods.net/sd/BabelFishService.wsdl' asURI readStream.
classBuilder createClientClasses

```

In this case there is one method babelFish:sourcedata:. Currently there are bugs in the code generation. The method generated looks like:

```

babelFish: aString sourcedata: aString
    "Generated by WS Tool on #(June 30, 2003 4:12:17 pm)"
    "operationName: #BabelFish"
    "addParameter: #translationmode type: #String"
    "addParameter: #sourcedata type: #String"
    "result: #String"

| args |
args :=Array new: 2.
args at: 1 put: aString.
args at: 2 put: aString.
^self executeSelector: #'babelFish:sourcedata:' args: args.

```

Edit the method so it is:

```

babelFish: aString sourcedata: bString
    "Generated by WS Tool on #(June 30, 2003 4:12:17 pm)"
    "operationName: #BabelFish"
    "addParameter: #translationmode type: #String"
    "addParameter: #sourcedata type: #String"
    "result: #String"

| args |
args :=Array new: 2.
args at: 1 put: aString.
args at: 2 put: bString.
^self executeSelector: #BabelFish args: args.

```

Note that the argument list was modified so it does not have to arguments with the same name and the last line was modified to send the correct SOAP method name to the server. Now we can use the client as follows.

```
babelClient := BabelFishPortClient url: 'http://www.xmethods.net/sd/BabelFishService.wsdl'.
babelClient babelFish: 'en_de' sourcedata: 'This is a test' .
```

Actually a better name for the method would be translationMode:sourcedata:.

The third version of a SOAP client uses Opentalk. The following code creates the Opentalk client for BabelFish.

```
| classBuilder |
classBuilder := WsdIClassBuilder readFrom: 'http://www.xmethods.net/sd/BabelFishService.wsdl'
asURI readStream.
classBuilder
    package: 'SampleServicePackage';
    createOpentalkClientClasses
```

This code creates the class OpentalkClientBabelFishPort in the package SampleServicePackage. A bug in the code generation requires you to edit the method OpentalkClientBabelFishPort>>babelFish:sourcedata:. The generated method contains to arguments with the same name: aString. Edit the method so it looks like:

```
OpentalkClientBabelFishPort>>babelFish: aString sourcedata: bString
    "Generated by WS Tool on #(June 30, 2003 4:31:43 pm)"
    "operationName: #BabelFish"
    "addParameter: #translationmode type: #String"
    "addParameter: #sourcedata type: #String"
    "result: #String"

    ^proxy babelFish: aString sourcedata: bString
```

The following code shows how to use the client.

```
| babelClient |
babelClient := OpentalkClientBabelFishPort new.
babelClient start.
babelClient babelFish: 'en_de' sourcedata: 'This is a test'
```

Since this is an Opentalk client when you are done you need to stop the client. This does seem very odd (stopping a client when there are no outstanding requests) but must be done.

```
babelClient stop.
```

### Why the three different clients?

Other than the obvious differences in the code above I have no insight into why VW 7.1 has an Opentalk SOAP client and WsdIClient.

### Some Useful Items

In playing with VW 7.1 Web Services I used several code snippets frequently. I include those here, so I have a place I can find them. Hope that they might be useful to others.

Many existing SOAP servers provide access to the service WSDL via normal http, so the following works.

```
'http://www.xmethods.net/sd/BabelFishService.wsdl' asURI readStream contents
```

When dealing with Opentalk clients and servers sometimes one loses references to running servers and clients. The following code will stop all clients and servers.

```
RequestBroker allInstances do: [ :each | each stop].
```

We will have a series of useful methods on a WsdIClient object. The first thing is to create a client.

```
babelWsd := WsdIClient url: 'http://www.xmethods.net/sd/BabelFishService.wsdl'.
```

"Which operations does the service support?"

```
babelWsd config interfaces first operations
```

"We can look at some details of the operations"

```
operation := babelWsd config interfaces first operations first.
```

```
operation inspect
```

"Since there is only one operation we can use the following to see some of the Wsdl"

```
babelWsd bindDocuments first root
```

"Saving the WSDL for later use. This code will open a dialog asking for a name for the file to hold the WSDL"

```
babelWsd saveSchemaBindings.
```

"Using the saved bindings"

```
newClient := WsdIClient fileName: 'TheFilenameContainingTheWSDL'.
```

```
newClient executeSelector: #BabelFish args: #('en_es' 'this is a test').
```