Cincom

Cincom Smalltalk™

Web Service Developer's Guide

P46-0142-04

SIMPLIFICATION THROUGH INNOVATION®

Part Number: P46-0142-04

Software Release 7.7

Cincom Systems, Inc.

55 Merchant Street

Cincinnati, Ohio 45246

Phone: (513) 612-2300

Fax: (513) 612-2000

World Wide Web: http://www.cincom.com

Contents

About This Book

This guide describes the VisualWorks web services libraries and frameworks for building both client and server applications. Web services support is an integral part of the VisualWorks technologies that enable you to build applications for the internet and e-business.

In addition to web services, VisualWorks also includes these related components:

- The VisualWorks Application Server, for building web applications using server pages, servlets, Seaside, and VisualWave.
- The Net Clients framework, which provides support for widely-used internet protocols, such as HTTP, FTP, and email protocols POP3, SMTP, IMAP, and MIME.

For details, see the *Web Application Developer's Guide* and the *Internet Client Developer's Guide*.

Audience

This document is intended for new and experienced developers who want to quickly become productive developing applications using the web services capabilities of VisualWorks.

It is assumed that you have a beginning knowledge of programming in a Smalltalk environment, though not necessarily with VisualWorks. For introductory-level documentation, you may begin with the on-line *VisualWorks Tutorial* (http://www.cincom.com/smalltalk/tutorial), and the *Application Developer's Guide*.

Organization

This guide begins with a general overview of web services, and the VisualWorks framework that support this new technology. The following chapters then describe the tools and libraries provided by the web services framework and how to use them when building your applications.

The chapters are as follows:

Chapter 1, "Introduction to Web Services." Briefly describes what web services are, the architecture of the VisualWorks implementation, the parcels making up VisualWorks Web Services support, and the included example applications.

Chapter 2, "Web Services Wizard." Describes the web services wizard for building applications from a WSDL schema, and vice versa.

Chapter 3, "Building Clients." Describes support for Web Service Description Langauge (WSDL), how to write and access WSDL documents in VisualWorks, and use builders to assist in creating classes from WSDL documents.

Chapter 4, "Document Processing." A more detailed discussion of WSDL document processing in the VisualWorks web services framework.

Chapter 5, "SOAP Exchanges." Describes SOAP messaging services, how to create a SOAP request, with or without a WSDL document, and how to manipulate SOAP headers.

Chapter 6, "Building Web Services." Describes how to build server applications, either from Smalltalk service classes, or a WSDL schema. Also describes how to produce a WSDL schema for use by other clients, and explores the use of Opentalk SOAP extensions when building a server application.

Chapter 7, "XML to Object Binding Wizard." Describes the XML-to-Object binding wizard, used for ascribing types to domain classes, creating X2O bindings, and XML schemas.

Chapter 8, "XML to Smalltalk Mapping." Describes the XML-to-Smalltalk mapping framework.

Conventions

We have followed a variety of conventions, which are standard in the VisualWorks documentation.

Typographic Conventions

The following fonts are used to indicate special terms:

Example	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`cover.doc`	Indicates filenames, pathnames, commands, and other constructs to be entered outside VisualWorks (for example, at a command line).
`filename.xwd`	Indicates a variable element for which you must substitute a value.
windowSpec	Indicates Smalltalk constructs; it also indicates any other information that you enter through the VisualWorks graphical user interface.
Edit menu	Indicates VisualWorks user-interface labels for menu names, dialog-box fields, and buttons; it also indicates emphasis in Smalltalk code samples.

Special Symbols

This book uses the following symbols to designate certain items or relationships:

Examples	Description
File > New	Indicates the name of an item (New) on a menu (File).
<Return> key <Select> button <Operate> menu	Indicates the name of a keyboard key or mouse button; it also indicates the pop-up menu that is displayed by pressing the mouse button of the same name.
<Control>-<g>	Indicates two keys that must be pressed simultaneously.
<Escape> <c>	Indicates two keys that must be pressed sequentially.
Integer>>asCharacter	Indicates an instance method defined in a class.
Float class>>pi	Indicates a class method defined in a class.

Mouse Buttons and Menus

VisualWorks supports a one-, two-, or three-button mouse common on various platforms. Smalltalk traditionally expects a three-button mouse, where the buttons are denoted by the logical names <Select>, <Operate>, and <Window>:

<Select> button	*Select* (or choose) a window location or a menu item, position the text cursor, or highlight text.
<Operate> button	Bring up a menu of *operations* that are appropriate for the current view or selection. The menu that is displayed is referred to as the *<Operate> menu*.
<Window> button	Bring up the menu of actions that can be performed on any VisualWorks *window* (except dialogs), such as **move** and **close**. The menu that is displayed is referred to as the *<Window> menu*.

These buttons correspond to the following mouse buttons or combinations:

	3-Button	2-Button	1-Button
<Select>	Left button	Left button	Button
<Operate>	Right button	Right button	<Option>+<Select>
<Window>	Middle button	<Ctrl> + <Select>	<Command>+<Select>

Note: This is a different arrangement from how VisualWorks used the middle and right buttons prior to 5i.2. If you want the old arrangement, toggle the **Swap Middle and Right Button** checkbox on the **UI Feel** page of the Settings Tool.

Getting Help

There are many sources of technical help available to users of VisualWorks. Cincom technical support options are available to users who have purchased a commercial license. Public support options are available to both commercial and non-commercial license holders.

Commercial Licensees

If, after reading the documentation, you find that you need additional help, you can contact Cincom Technical Support. Cincom provides all customers with help on product installation. For other problems there are several service plans available. For more information, send email to supportweb@cincom.com.

Before Contacting Technical Support

When you need to contact a technical support representative, please be prepared to provide the following information:

- The *version id,* which indicates the version of the product you are using. Choose **Help > About VisualWorks** in the VisualWorks main window. The version number can be found in the resulting dialog under **Version Id:**.
- Any modifications (*patch files*) distributed by Cincom that you have imported into the standard image. Choose **Help > About VisualWorks** in the VisualWorks main window. All installed patches can be found in the resulting dialog under **Patches:**.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, select **copy stack** in the error notifier window (or in the stack view of the spawned Debugger). Then paste the text into a file that you can send to technical support.

Contacting Technical Support

Cincom Technical Support provides assistance by:

Electronic Mail

To get technical assistance on VisualWorks products, send email to supportweb@cincom.com.

Web

In addition to product and company information, technical support information is available on the Cincom website:

http://supportweb.cincom.com

Telephone

Within North America, you can call Cincom Technical Support at (800) 727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time.

Outside North America, you must contact the local authorized reseller of Cincom products to find out the telephone numbers and hours for technical support.

Non-Commercial Licensees

VisualWorks Non-Commercial is provided “as is,” without any technical support from Cincom. There are, however, on-line sources of help available on VisualWorks and its add-on components. Be assured, you are *not* alone. Many of these resources are valuable to commercial licensees as well.

The University of Illinois at Urbana-Champaign very kindly provides several resources on VisualWorks and Smalltalk:

- A mailing list for users of VisualWorks Non-Commercial, which serves a growing community of VisualWorks Non-Commercial users. To subscribe or unsubscribe, send a message to:

  vwnc-request@cs.uiuc.edu

  with the SUBJECT of "subscribe" or "unsubscribe". You can then address emails to vwnc@cs.uiuc.edu.

- A Wiki (a user-editable web site) for discussing any and all things VisualWorks related at:

  http://www.cincomsmalltalk.com/CincomSmalltalkWiki

The Usenet Smalltalk news group, comp.lang.smalltalk, carries on active discussions about Smalltalk and VisualWorks, and is a good source for advice.

Additional Sources of Information

This is but one manual in the VisualWorks library. The Cincom Smalltalk publications website:

http://www.cincomsmalltalk.com/documentation/

is a resource for the most up to date versions of VisualWorks manuals and additional information pertaining to Cincom Smalltalk.

1

Introduction to Web Services

The Internet has become a widely-used and trusted source of information. Increasingly, it is also a medium for performing commerce. A family of new protocols known as web services are now being used to extend the usefulness of the Internet as a general service provider.

This chapter gives a brief, general overview of web services, describes the architecture of the VisualWorks implementation, its compliance with industry standards, and offers some common usage scenarios that can guide you to the sections of this guide that are most relevant to your application.

For developers new to VisualWorks, Smalltalk, or web services, we strongly recommend starting with some of the example applications described at the end of this chapter.

In brief, this chapter describes:

About Web Services

Web services provides a fine-grained approach to building web applications. Services are typically envisioned as small building-blocks, such as authentication (e.g., Microsoft® *Passport*), phrase translation, currency conversion, or shipping status lookup. But, there is no restriction on the size of the application, and large business processes can also be presented as web services.

Web services provide a mechanism for companies to make proprietary software publically available without distributing software and data outside their organization.

According to the W3 organization "A web service is a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A web service supports direct interactions with other software agents using XML based messages exchanged via internet-based protocols" (Web Services Architecture Requirements).

To make a service available, a provider designs its API and expresses it using WSDL, then implements the service.

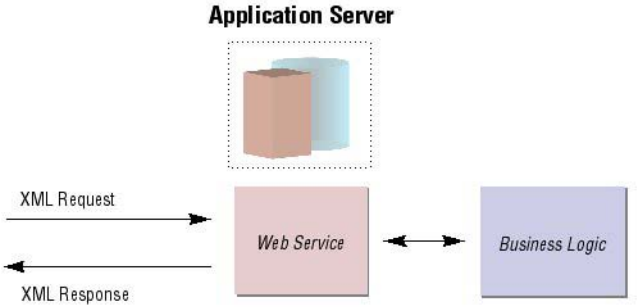
The VisualWorks web services framework makes it easy to interoperate with remote services, or to make Smalltalk applications available on the Internet as services for others. VisualWorks supports web services by providing rich implementions of the WSDL and SOAP, class builders, and wizards to simplify application development. UDDI standards are not much is use, and so are not supported at this time.

The following pages explore the high-level architecture of the VisualWorks web services framework, and offer some general suggestions about how you can find the discussions in this guide that are most relevant to your particular development tasks.

Architecture

Web services are implemented following the general model of a web application: a client sends a message to a service provider, which in turn performs some operation and sends a response. Typically, HTTP is used as a transport mechanism.

In the case of an application using a web service, the message from the client is expressed as XML. Here, a XML document is used to describe more complex interactions between client and server. The web services framework provides a way to specify access to the business logic of the web application, and handles the translation of objects in the business model to and from the XML representation used for messaging.



For an application to use a web service, the programmatic interface of the service must be precisely described. This is accomplished with WSDL (Web Services Description Language), a XML grammar for describing web services. In this sense, WSDL plays a role analogous to the Interface Definition Language (IDL) used in describing CORBA services.

A WSDL document represents the public interface of a web service as a collection of “endpoints,” or ports, that receive and handle documents. The port description includes such details as the protocol bindings, host and port number used, the operations that can be performed, the formats of the input and output messages, and the exceptions that can be raised.

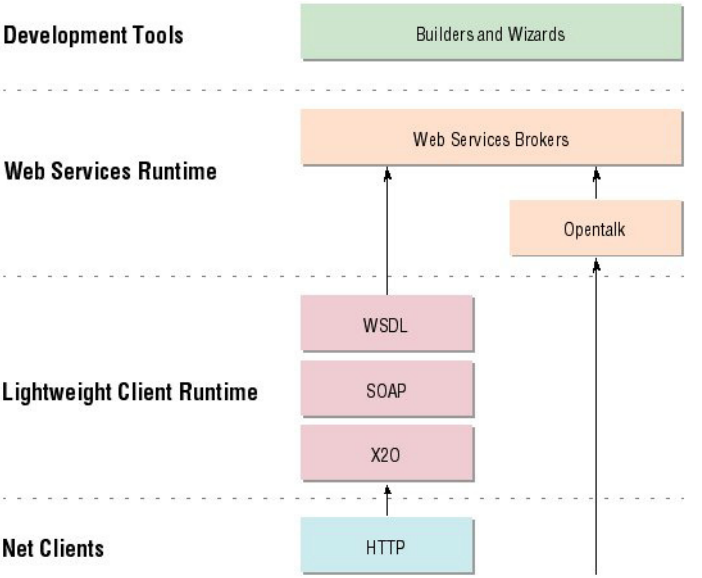
The SOAP protocol is often used with WSDL to provide a web service. WSDL provides the interface description of the service, and SOAP is then used to actually call the service over the network. SOAP provides a way to represent messages to be executed by a remote service provider. SOAP facilitates the exchange of structured and typed information in a distributed, heterogeneous environment.

VisualWorks Implementation

The VisualWorks web services framework is a layered, modular implementation that enables you to build both clients and servers. Lightweight client applications use the core support for WSDL and SOAP. For server applications, the VisualWorks Opentalk framework is used in conjunction with web services.

To help automate the construction of your application, the development environment also includes a number of class builders and wizards.

The organization of the web services framework is illustrated below.



The VisualWorks implementation of web services has been architected to simplify the design of your application, while giving you complete control over lower-level interfaces as needed.

The following sections describe each component of the web services framework in more detail.

XML and HTTP Support

The web services platform requires rich support for XML and HTTP. XML is used as the basis for the higher-level protocols such as WSDL and SOAP. VisualWorks provides a complete XML support library that includes both DOM and SAX APIs.

The NetClients package for VisualWorks provides basic HTTP client support, including authenticated HTTP and HTTPS. A VisualWorks HTTP server can be used in a stand-alone configuration or, for higher performance, behind a commercial server such as Apache or IIS.

The VisualWorks Opentalk framework provides a full-featured HTTP server. When building server applications that support WSDL and SOAP, this is augmented with the Opentalk-SOAP package.

For a complete discussion of the Opentalk framework, refer to the Opentalk Communication Layer Developer's Guide.

XML to Object Mapping

To create XML messages based on a schema description or to unmarshal XML messages into Smalltalk objects, VisualWorks includes the XMLSchemaMapping, XMLObjectMarshalers, and XMLObjectBindingTool packages.

Given an XML document containing a <types> description (or given an XML document containing a <schema> description), an XMLTypesParser is used to generate a binding specification. This specification describes the mapping of schema types into Smalltalk objects (also called an "X2O specification", this is in fact a XML document). Complex XML types may be mapped into custom Smalltalk domain classes (the web services framework can also build these classes from a binding specification).

To actually marshal Smalltalk objects into XML and vice versa, class BindingBuilder uses the XMLTypesParser to create a binding specification. From this, the builder creates Smalltalk classes, then XMLObjectBinding is used to load a binding specification, creating and registering marshalers in the XMLBindingRegistry.

WSDL

VisualWorks WSDL support provides simple mechanisms for:

- Loading and parsing WSDL documents
- Creating classes from user-defined object types in a WSDL document

- Programmatically invoking a web service based on the port information in a WSDL document

For building simple clients programmatically, most of the WSDL API is provided by class WsdlClient. This class can load a WSDL schema and invoke client operations without creating any additional classes. To use more complicated features (e.g. SOAP header processing) you may want to use the OpenTalk client.

VisualWorks also provides several builders to generate Smalltalk classes or schemas for use in your application: WsdlClassBuilder and WsdlBuilder.

Class WsdlClassBuilder can programmatically generate service classses from schemas. Given a WSDL document, WsdlClassBuilder generates Smalltalk classes for complex types from the X2O specification created by an XMLTypesParser.

Class WsdlBuilder can generate a WSDL schema from a Smalltalk application. Given a service class with a pragma description for web service operations, the WsdlBuilder can create a WSDL schema.

SOAP

VisualWorks provides support for SOAP document exchange using HTTP as the underlying transport mechanism. The SOAP API allows for both RPC-style services, where the unit of interaction is a WSDL operation, and also a service-oriented architecture (SOA) that is message oriented.

Support for WSDL greatly simplifies SOAP document exchange in VisualWorks by automatically producing the appropriate SOAP message using transformations based on a WSDL schema, making SOAP programming almost trivial.

SOAP support in VisualWorks is provided by class SoapRequest and its subclasses. The XML-to-Object mapping mechanism described above is used when making SOAP requests.

Wizards

The VisualWorks web services framework also includes two wizards to simplify the task of building your application:

WSDL Wizard

This wizard may be used when building both client applications or web services. When building a client application, the wizard can automatically generate binding classes from a WSDL

schema (i.e., the classes generated from a X2O (XML to Object) specification that correspond to XML complex types). When building a server application, the wizard can generate a WSDL schema from Smalltalk classes. The wizard also generates workspace code that may be used during testing and construction of your application.

X2O Wizard

This wizard may be used to build X2O bindings and XML schemas from classes. These may be used in web service applications, but the wizard is designed to be general purpose. It may be useful for any application that requires code to translate between Smalltalk objects and XML.

Compatibility with Standards

The VisualWorks web services implementation is compliant with the following industry-standard protocols:

Protocol	Version
WSDL	1.1 (SOAP via HTTP bindings only)
SOAP	1.1
Basic Profile	1.1
HTTP	1.1 (RFC 2616)
XML	1.0

VisualWorks interoperates naturally with web services and clients written for other platforms. For example, a web service implemented in VisualWorks and deployed on the network is immediately visible to .NET developers, and is indistinguishable from a native .NET service. Likewise, .NET services are easily discovered and accessed from VisualWorks.

Interoperability with Java web services and clients can be tested using the tools available at: http://ws.apache.org/axis/index.html.

Common Usage Scenarios

This guide includes detailed discussions of several common usage scenarios, involving both clients and servers. The code examples in these discussions may provide useful patterns as you build your application.

Creating Web Services Clients

The following scenarios presuppose that your application makes client requests and that you have a WSDL schema for a remote web service.

1 If you want to send simple client requests via an API, see the discussion of class WsdlClient (Building Clients).

2 If your WSDL schema uses complex data types, and you want to use a wizard to generate the binding classes, see the wizard guide (Web Services Wizard).

3 If your WSDL schema uses complex data types, and you want to use an API to programmatically generate binding classes, see the discussion of class WsdlClassBuilder (WSDL Class Builder).

Creating Web Services

The following scenarios presuppose that your application provides web services to clients running remotely (i.e., a server).

1 If you want to use a wizard to create services for your application, see the discussion of Creating Service Classes using the Web Services Wizard.

2 To automatically generate classes for the server application using the class builder API, see Creating Service Classes using the WsdlClassBuilder.

3 To build a Smalltalk service from a WSDL schema using a wizard, see Building Servers from a WSDL schema.

4 To create a WSDL schema for a Smalltalk service using a wizard, see Generating a Schema using the Web Services Wizard.

5 If you prefer to create the server programmatically via an API, see the discussion of Using the Opentalk Request Broker.

Loading Support for Web Services

The VisualWorks web services framework is provided in a collection of parcels that load successive layers of support. Higher-level parcels load lower level ones as prerequisites.

To load a web services parcel, open the Parcel Manager (select **Parcel Manager** from the **System** menu in the Launcher window), and select the **Web Services** category in the list on the left side of the Parcel Manager.

The main parcels and their functionality are as follows:

WSDL

Installs basic support for WSDL and SOAP.

WSDLTools

A builder for generating classes from a WSDL schema and vice versa. For details, refer to Building Clients.

WSDLWizard

Installs a tool for creating Smalltalk classes from a WSDL schema and a WSDL schema from a service class. For an example illustrating its use, see: Web Services Wizard.

SOAP

Installs the SOAP bindings for XML-to-Object marshaling. For details, refer to SOAP Exchanges.

Opentalk-SOAP

One of either Opentalk-HTTP or Opentalk-CGI must also be loaded. This parcel adds server support to WSDL, enabling the use of Web Services as a viable distributed computing environment. For details, refer to Building Web Services.

XMLObjectBindingTool

A builder for generating Smalltalk classes from an XML-to-Object binding.

XMLObjectBindingWizard

Installs a wizard for creating a XML-to-Object binding specification. For an example illustrating its use, see XML to Object Binding Wizard.

XMLObjectMarshalers

The basic XML-to-Object marshaling machinery, required by all other web service support (also useful for developing protocols other than SOAP). For details, refer to XML to Smalltalk Mapping.

You can browse the dependencies between these components by using the Parcel Manager and selecting the **Prerequisite Tree** tab.

Web Services Settings

The Web Services section in the System Settings tool provides options for controlling aspects of both class and schema building, affecting the behavior of WsdlClassBuilder, WsdlBuilder, and the web services wizards.

To open the Settings Tool, select **System** > **Settings** in the Visual Launcher.

Web Services Examples

Several example applications are provided to quickly illustrate the use of the VisualWorks web services framework. We recommend starting with these examples to learn about the framework, builders, and tools.

The examples are provided in two parcels:

WebServicesTimeDemo

The WebServicesTimeDemo provides simple client and server applications to illustrate some basic features of the VisualWorks framework.

WebServicesDemo

The WebServicesDemo is a more advanced example that illustrates the use of communication features. The basic application is contained in the Protocols-LibraryDemo package, which is extended by the WebServicesDemo. You may use it to exercise much of the advanced functionality described in this guide.

To load the demo parcels, use the Parcel Manager as described above (see Loading Support for Web Services). All prerequistes are loaded automatically.

Time Demo

The WebServicesTimeDemo is a simple time service that responds to client requests with an object reporting the current time (i.e., an instance of class Time). The demo includes three simple applications to illustrate how web service clients interact with servers.

Each of the three applications is simply a pair of two classes (client and server), illustrating a particular style of binding the message to the underlying protocol. All of these example classes are located in the WebServices.* name space.

Three clients are provided in the demo:

Class name	Description
TimeClient	Uses standard document/literal style
TimeClientRPC	Uses RPC (Remote Procedure Call) style
HTimeClient	Uses SOAP header processing

Three server classes are provided, one for each of these clients: TimeServer, TimeServerRPC, and HTimeServer.

Using the Time Demo

To test the examples, simply start one of the servers and then send requests using the corresponding client class. This can be done on a single workstation, with both server and client running in the same VisualWorks image.

For example, to test the TimeServer, evaluate the following code in a Workspace window using **Inspect It** (from the <Operate> menu):

```
| client value |
“Start the default TimeServer”
WebServices.TimeServer defaultStart.
“Create and start a client”
client := WebServices.TimeClient new.
client start.
“Request the current time from the server”
value := client timeNow.
“Stop the client and server”
client stop.
WebServices.TimeServer defaultStop.
```

The result in the variable "value" should be an OrderedCollection containing a Time object.

These simple applications are used as examples later in this guide. For additional details on the applications and their implementation, see to the package comment for WebServicesTimeDemo, or browse the client and server classes.

Library Demo

The Library Demo is a web service that models a public library. Based upon the Protocols-LibraryDemo package, it provides services to patrons, maintains holdings that can be loaned out, and provides other services like copying, searching, a weather forecast, etc.

The demo includes a simple application called LibraryServer that can be used to illustrate how web service clients interact with servers.

Using the Library Demo

To test the Library Demo, you may start the LibraryServer and then send requests using the corresponding client classes. This demo has been designed to be run using two separate VisualWorks images, one for the server and another for the client.

To start the Library Demo, load the WebServicesDemo and evaluate:

```
LibraryServer defaultStart.
```

For details on the actual use of the Library Demo, see the package comment for WebServicesDemo.

Unit Tests

Extensive SUnit tests are available for the VisualWorks web services framework. These are not included with the distribution CD, but may be loaded directly from the Cincom public Store repository.

2

Web Services Wizard

The VisualWorks web services framework includes wizards and builders that can automatically generate Smalltalk classes for use in your application. Given a WSDL schema, the WSDL wizard can generate custom client classes to access the service described by the schema.

The web services wizard is the simplest way to create the Smalltalk code you need to access a web service from within VisualWorks. However, for very lightweight applications that do not require custom client classes, you might also consider using the WsdlClient, as described in Building Clients.

This chapter shows how to use the WSDL wizard to build a client-side application. To generate a schema from classes and prepare an existing Smalltalk application for presentation as a web service (i.e., to build a server application), see Building Web Services.

Creating Classes using the Web Services Wizard

Given a WSDL schema, the wizard can create the Smalltalk code and client classes that are needed to access a web service from within VisualWorks. The wizard allows you to select processing options, and then it generates the <schemaBindings> section and supporting code.

To illustrate the use of the wizard, we shall use the WebServicesTimeDemo (for details, see: Web Services Examples). To begin, first load the demo parcel and then start the time server by evaluating the following code:

```
WebServices.TimeServer defaultStart.
```

Before using the wizard, you must load the WSDLWizard package (for instructions, see Introduction to Web Services).

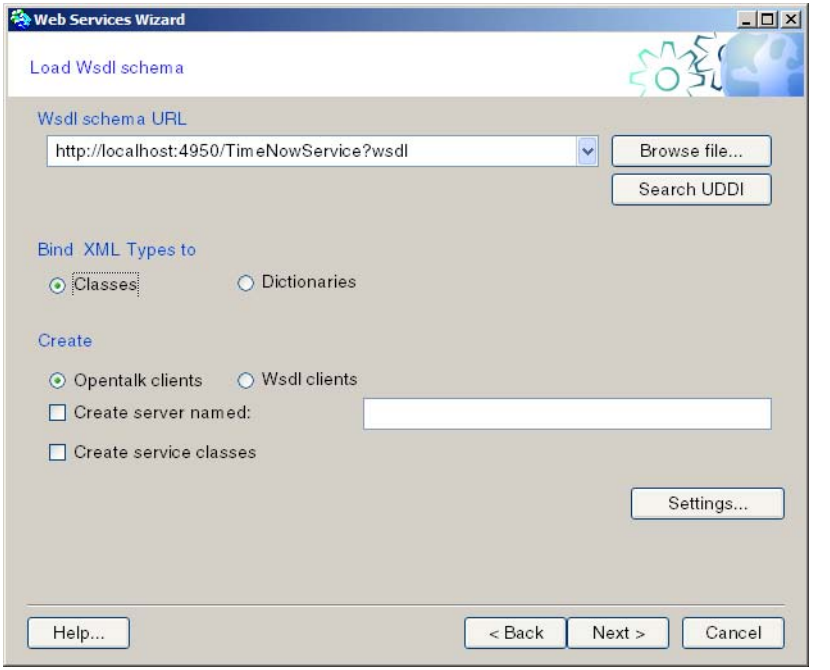
Generating a WsdlClient

The wizard provides a number of options for generating Smalltalk classes and code. The simplest is to generate a subclass of WsdlClient.

1 To launch the wizard, select **Web Services Wizard** on the **Tools** menu of the Visual Launcher.

2 On the first page of the wizard, select **Create an application from a WSDL schema**, and click **Next**.

3 On the next page, specify a schema to load.



The demo TimeServer provides a WSDL schema at this URL:

http://localhost:4950/TimeNowService?wsdl

In general, you can either enter a URL in the **WSDL schema URL** field, or load a WSDL document from a file (click on **Browse file…**).

4 In the **Bind XML Types to** section of the wizard, select Classes.

This option specifies how to handle complex data types (for details, refer to Generating XML-to-object Bindings):

- **Classes** creates a Smalltalk class for each complex type
- **Dictionaries** maps each complex type to a Dictionary

Since we are using the WebServicesTimeDemo as the target service, we select Classes.

5 The **Create** option tells the wizard what kind of code to generate for the client. The two options are:

- **WSDL clients** generates code for issuing a standard SOAP request
- **Opentalk clients** generates code for issuing requests via an Opentalk proxy (refer to the Opentalk Protocol Layer).

Select the **WSDL clients** option, which is simpler.

Leave the **Create Opentalk server name** and **Create service class** options unselected for this procedure.

6 For the WebServicesTimeDemo, ignore the **Settings...** button. This button opens a dialog for setting the package and/or name space for the generated code.

7 Click **Next** to generate the Smalltalk support code.

8 A dialog opens to show the list of response classes that will be generated. Click **OK**.

9 Once the code is generated, the final wizard page is displayed. This page displays a workspace with Smalltalk expressions that exercise the newly-generated client code.

When using the WebServicesTimeDemo, the following code should appear in the workspace:

```
client := TimeNowServiceWsdlClient new.
value := client timeNow.
```

You can select this code and evaluate it directly in the workspace with **Inspect It**. The result should be an OrderedCollection containing a single instance of TimeNowResponse. This is the result passed from the TimeServer running on your workstation.

To save the workspace, copy its contents and paste them into another workspace.

10 Click **Finish** to close the wizard.

At this point, the wizard has generated a client class that is suitable for use in an application (TimeNowServiceWsdlClient). The class is located in the package WSDefaultPackage, but may be moved elsewhere.

Generating an Opentalk client

When building a client, the wizard can generate either a new subclass of WsdlClient, or a subclass of the Opentalk client (see step 5, above). If you opt for an Opentalk client, the wizard can also create service classes.

The generated service classes include stub methods for all the operations defined in the WSDL schema, but no implementation. This is because WSDL defines the operations, but not their implementation. To use these service classes, you must add an implementation for each operation.

The wizard can be used to simplify this step as well. To illustrate how this is done, we can use the WebServicesTimeDemo again, following roughly the same steps described in the previous section (Generating a WsdlClient).

1 To begin, check that the demo parcel has been loaded (WebServicesTimeDemo should appear in the package view of a System Browser) and, if you have not already done so, start the time server by evaluating the following code:

```
WebServices.TimeServer defaultStart.
```

2 Launch the wizard, by selecting **Web Services Wizard** on the **Tools** menu of the Visual Launcher.

3 On the first page of the wizard, select **Create an application from a WSDL schema**, and click **Next**.

4 On the next page, specify the **WSDL schema URL**:

```
http://localhost:4950/TimeNowService?wsdl
```

The demo TimeServer provides a WSDL schema at this address.

5 In the **Bind XML Types to** section of the wizard, select Classes.

6 In the **Create** section of the wizard, select **Opentalk clients**, and select the **Create service class** option.

7 Click **Next** to generate the Smalltalk support code.

8 A dialog prompts to show the response classes that will be generated, with the option to change their names. Simply click **OK**.

9 Once the code is generated, the final wizard page is displayed. This page displays a workspace with Smalltalk expressions that exercise the newly-generated client code.

When using the WebServicesTimeDemo, the following code appears in the workspace:

```
client := TimeNowServiceClient new.
client start.
value := client timeNow.
client stop.
```

You can select this code and evaluate it directly in the workspace with **Inspect It**. The result in the variable value should be an OrderedCollection containing a single instance of TimeNowResponse. This is the result passed from the TimeServer running on your workstation.

To save the workspace, copy its contents and paste them into another workspace.

10 Click **Finish** to close the wizard.

3

Building Clients

The VisualWorks web services framework provides a basic API for making client requests, and also APIs for building client classes. To make client requests, the simplest of these is class WsdlClient, which provides an easy way to get started using web services.

The class-builder API (WsdlClassBuilder) may be used to create classes that are specific to your application. Provided with a suitable WSDL document, the WsdlClassBuilder can help automate the development of your application by generating classes. Since these are subclasses of WsdlClient, their operation is similar.

For applications that require more complicated features (e.g. SOAP header processing), an Opentalk client must be used instead of a WsdlClient or its subclasses (for details on Opentalk-based clients, see: Building Web Services)

In brief, this chapter describes:

- WSDL Support Services
- Loading WSDL Support
- Authentication
- WSDL Class Builder

WSDL Support Services

Class WsdlClient provides most of the WSDL client API. This class:

- Loads a WSDL schema using URI, and from schema parts if they are represented by an <import> element in the schema, such as:

  ```
  <import location="http://www.whitemesa.com/interop/
  InteropTest.wsdl"
  namespace="http://soapinterop.org/" />
  ```

- Parses the schema and automatically creates a default <schemaBindings> element (for details on schema parsing, refer to XML to Smalltalk Mapping).
- Creates a WsdlConfiguration, which consists of WSDL schema objects such as WSDL services, operations and bindings. At that time the <types> section is completely ignored. The <schemaBindings> element is used to create all marshalers.
- Based on the WsdlConfiguration, constructs a SOAP message, sends it via HTTP, and unmarshals the response from the SOAP body.

Loading WSDL Support

To send client requests with WsdlClient (or specific clients), use the Parcel Manager to load the WSDL parcel (for step-by-step instructions, see Loading Support for Web Services). To create specific clients during application development, load either the WSDLTool or WSDLWizard parcels.

Using WsdlClient

Generally, an instance of class WsdlClient is configured for a particular web service, and then used to send requests to a server.

As a simple example, we can host both client and server in a single VisualWorks development image. Here, we query the TimeServer in the WebServicesTimeDemo. To begin, load the demo parcel (see Loading Support for Web Services), open a Workspace, and evaluate the following code to start the demonstration server:

```
WebServices.TimeServer defaultStart.
```

With the server running on your local workstation, evaluate the following code in a Workspace using Inspect It:

```
wsdlClient := WsdlClient new loadFrom:
    'http://localhost:4950/TimeNowService?wsdl' asURI.
value := wsdlClient executeSelector: #TimeNow.
```

The result in the variable "value" should be an OrderedCollection containing a web services Struct object. The Struct contains the current time.Use the Settings Manager, to set the correct time zone for the VisualWorks image. Select Settings from the System menu in the Launcher window, and use Do It to evaluate code on the right-hand side of the tool to specify your time zone.

In the code example shown above, the message loadForm: causes the client to load and parse a WSDL schema, and create several registries and a WsdlConfiguration. Then, executeSelector: causes the client to actually send a request and get a response.

You can examine the client's configuration by evaluating:

```
wsdlClient config
```

Based upon this configuration object, the WsdlClient can also create a script of Smalltalk code that illustrates the full interface defined by the schema. This may be useful for testing the remote service.

To see the script, evaluate the following using Inspect It:

```
wsdlClient createScript
```

When you are finished with the example, don't forget to stop the server:

```
WebServices.TimeServer defaultStop.
```

For a more detailed discussion of WsdlClient and its use, see Document Processing.

Class Struct

The response to a query using WsdlClient generally includes a Struct. Instances of class Struct are used to represent the 'struct' object from C-like languages. The elements of a Struct can be accessed using a basic subset of Dictionary protocol (at: and at:put:, etc.), but unlike a Dictionary the order of elements in a Struct is maintained, with new elements being added at the end. Similarly, a Struct defines equivalence in terms of structural comparison.Starting in release 7.5, WebServices.Struct is a subclass of Protocols.Struct, rather than of Dictionary. Since this change may affect your application code, we recommend careful testing and allowing time for revision as required.

To provide backward compatibility, WebServices.Struct remains in the image. However, this class is obsolete and will likely be removed in a future release.

Authentication

Class WsdlClient supports all standard forms of HTTP authentication (e.g., Basic, Digest and NTLM schemas). Instances of WsdlClient use an HttpClient object as a transport mechanism, and the latter provides the actual support for authentication.HTTP authentication is presently available only for applications using the WsdlClient. This functionality is under active development for Opentalk and will be included in a future release.

Generally, authentication is set up by sending username:password: to the WsdlClient instance before executing any remote operations.

For example:

```
myClient := WsdlClient new.
myClient username: 'myUser' password: 'myPassword'.
myClient loadFrom: 'http://myCompany.com/mySchema.wsdl' asURI.
myClient executeSelector: #setString args: (Array with: 'myString').
```

At the transport level, the server at myCompany.com returns a challenge if it requires authentication for mySchema.wsdl. The HttpClient responds to the challenge with a token that authorizes the schema request.

It is also possible that executeSelector:args: will also require authentication. In this case, HttpClient adds authorization to the SOAP request. If a different user token is required, the user name and password can be reset before sending the SOAP request.

Another example would be to only set the username and password in the event of an error. In this case, an exception handler may be used to process HttpUnauthorizedError, i.e.:

```
[client := WsdlClient url: 'http://myCompany.com/mySchema.wsdl']
    on: HttpUnauthorizedError
    do:   [:ex |
            client username: 'anotherUser' password: 'somePassword'.
            ex retry].
[client executeSelector: #setString args: (Array with: 'myString')]
    on: HttpUnauthorizedError
    do:   [:ex |
            client username: 'anotherUser' password: 'somePassword'.
            ex retry].
```

WSDL Builders

Two builder classes are provided to assist in creating classes and WSDL schemas. Using these builders, you can create service classes from a WSDL schema, or vice versa.

WsdlClassBuilder

Defines Smalltalk classes using the requirements specified in a WSDL schema. A schema describes objects and the messages they respond to, which can be represented in Smalltalk as classes and instance messages. WsdlClassBuilder can also generate classes for user-defined types.

WsdlBuilder

Generates a WSDL specification document from service provider classes, simplifying the task of making a service written in VisualWorks available for web access. For details, see Building Web Services.

WSDL Class Builder

WsdlClassBuilder performs the work of WsdlClient and more. In addition to reading and parsing a WSDL document, it builds Smalltalk classes representing the services described in the WSDL document. Classes are built according to the <schemaBindings> section of the WSDL document.

The WsdlClassBuilder can generate the following classes:

- Smalltalk classes from user-defined data types.
- The client class that includes methods for accessing the services (operations) of the service classes.
- The service classes from the WSDL operation description. These classes implement the services. The service class names are the same as given in the WSDL specification binding element.
- The Opentalk server class that implements the code to set up, start and shutdown a server.

In this chapter we will examine only the client side functionality, although there is little difference for defining the service classes for a SOAP server. Refer to XML to Smalltalk Mapping for discussion of server-related uses.

Running WsdlClassBuilder

The class builder has a very simple API, very much like WsdlClient. Minimally, to build client classes from a WSDL schema, you may evaluate the following:

```
| builder |
builder := WsdlClassBuilder readFrom:
    'http://live.capescience.com/wsdl/AirportWeather.wsdl' asURI.
builder createClientClasses.
```

Classes are created in the default package, category, and name space specified in the Web Services **Class Builder** page of the Settings Manager (to open this tool, choose **Settings** from the **System** menu in the Launcher window).

To programmatically specify a package where the classes are generated, configure the builder by sending a package: message to the builder before generating the classes:

```
| builder |
builder := WsdlClassBuilder readFrom:
    'http://live.capescience.com/wsdl/AirportWeather.wsdl' asURI.
builder package: 'AirportWeather'.
builder createClientClasses.
```

The API for WsdlClassBuilder is summarized below, under Class-generating API.

Loading and Saving a Schema

Schemas may be loaded from and saved to files. The protocol for classes WsdlClassBuilder and WsdlClient is the same. For details, see: Saving a Schema with its Binding and Load and Use a WSDL Schema.

Overwriting Class names

When generating classes, it is possible that a class name specified by the schema is already in use in the Smalltalk image. These name clashes are handled according to the settings of the WsdlClassBuilder useExistingClassName options, as follows:

- If set to true and a class with this name is already in the system, the class is not generated; the existing class is used for the binding. This is the default option.
- If set to false, new classes are always generated, and they are unique in the namespace where the classes are defined.

If set to false, then to ensure uniqueness the newly class name is appended with a number, if the class already exists. For example, if the target name space already has the class Document and the XML attribute is name="Document", then the binding object name will be set to “Document1”. If Document1 already exists, then it is named “Document12”, and so on.

Cleaning the Binding Registry

Every time you run WsdlClassBuilder to generate classes, it registers the object binding in XMLBindingRegistry, which provides each object marshaler with a reference to a Smalltalk class. If you re-run WsdlClassBuilder on a schema, you need to clean up the registry; otherwise you can end up with unexpected results and obsolete class references.

To remove a specific registry entry, you can evaluate:

```
XMLObjectBinding registry removeKey:
    someTargetNamespaceFromTypesSchema
```

If you do not need to preserve the current bindings, you can reconfigure the entire registry:

```
XMLObjectBinding configure
```

Then regenerate the classes.

Moving a client Class to another Image

When you generate classes from a WSDL schema, all information about access points is registered in WsdlPort.PortRegistry. The client class knows its binding, which helps it find correct port in the registry.

To move your client class in to another VisualWorks image (via file-out or publishing as a package), the port registry needs to be updated. You can either initialize the port registry, or evaluate:

```
WebServices.WsdlBinding loadWsdlBindingFrom:
        MyWsdlClient wsdlSchema readStream.
```

Class-generating API

The WsdlClassBuilder protocol relevant for building a SOAP client from a WSDL document is described below. Additional protocol is described in Building Servers from a WSDL schema.

Instance creation class methods

readFrom: *aDataSource*

Create an instance of WsdlClassBuilder and loads the Wsdl schema from a data source, and creates a <schemaBinding> section for user defined data types. The *dataSource* may be a URI, a Filename (a String will be treated as a Filename), or an InputSource.

Environment setting methods

package: *aPackageName*

If *aPackageName* does not exist in the system, the package will be created and all classes created in it.

category: *aString*

Creates all classes in the specified category.

namespace: *aString*

Creates all generated classes in the specified namespace. By default, classes are created in the Smalltalk namespace.

Class generation methods

createBindingClasses

Creates binding classes from the schema <types> element.

createClasses

Creates binding, client, and service classes.

createClientClasses

Creates client classes, and binding classes if not already generated.

createServiceClasses

Creates stub service classes, and binding classes if not already generated. Service classes are the minimum required to implement a web service. For details, see Building Web Services.

createServerClass

Creates the Opentalk server class. For details, see Building Web Services.

Inside the WsdlClassBuilder

The WsdlClassBuilder can create the following:

- Schema Bindings
- Binding Classes
- Client Classes
- Service Classes

The following sub-sections describe each of these elements in more detail.

Schema Bindings

As described above (Load and Use a WSDL Schema) when a WSDL schema includes user-defined data types described in the <types> element, WsdlClient creates bindings for these, mapping them to Smalltalk data types, according to default type mappings. WsdlClassBuilder takes this a step further and identifies Smalltalk classes for these types.

For example, the WSDL schema for the Library Demo describes the complex type Book as (see LibraryServer class method wsdlSchema):

```
<types>
    <schema targetNamespace="urn:webservices/demo/libraryServices"
      elementFormDefault="qualified"
      xmlns:tns="urn:webservices/demo/libraryServices"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns="http://www.w3.org/2001/XMLSchema">
      <complexType name="Protocols.Library.Book">
        <sequence>
          <element name="publicationYear" type="xsd:short"/>
          <element name="publisher" type="xsd:string"/>
          <element name="acquisitionCost" type="xsd:decimal"/>
          <element name="pages" type="xsd:short"/>
          <element name="acquisitionNumber"
            type="xsd:positiveInteger"/>
          <element name="braille" type="xsd:boolean"/>
          <element name="collectionId" type="xsd:string"/>
          <element name="language" type="xsd:string"/>
          <element name="statusId" type="xsd:string"/>
          <element name="libraryName" type="xsd:string"/>
          <element name="title" type="xsd:string"/>
          <element name="catalogNumber"
            type="tns:Protocols.Library.CatalogNumber"/>
```

```
                <element name="authors"
                    type="tns:CollectionOfAuthorialName"/>
                <element name="acquisitionDate" type="xsd:date"/>
                <element name="dueDate" type="xsd:date" minOccurs="0"/>
                <element name="largePrint" type="xsd:boolean"/>
            </sequence>
        </complexType>
...
    </schema>
</types>
```

WsdlClient would create a Struct from this complex type, which would then be marshaled as a Dictionary. WsdlClassBuilder creates a X2O binding with XML complex types mapped to objects.

Binding Classes

From the bindings defined in the <schemaBindings> section, WsdlClassBuilder generates corresponding binding classes. In the above example from the Library Demo, the class is named Holding. Based on the aspect attribute, the instance variable accessors will also be created. The set accessor will include a pragma that describes parameter types. For example:

```
acquisitionCost
    ^acquisitionCost

acquisitionCost: aFixedPoint
    "Generated by WS Tool on #(January 3, 2003 11:41:36 am)"
    <addAttribute: #acquisitionCost  type: #FixedPoint>
    acquisitionCost := aFixedPoint
```

Binding classes and their method definitions are used by the class builder when defining the client and service classes, to provide the information needed to properly construct those classes. Binding classes are also used for creating a WSDL schema from service classes, as described in Building Web Services.

For a client application, the binding classes are not needed once the client classes are generated, and can be safely removed from the development image. For convenience, you can define them in a separate package from the client classes, as follows:

```
builder := WsdlClassBuilder readFrom: 'LibraryDemo.wsdl' asFilename.
builder package: 'LibraryDemoBindings'.
builder createBindingClasses.
builder package: 'LibraryDemoClient'.
builder createClientClasses.
```

Client Classes

Client classes contain the protocol necessary for requesting a service described in the WSDL document. In general, you can simply create an instance of the client and request the service using the supplied API.

The client classes are created using the WSDL operation description, and include methods that allow it to load the schema binding and invoke services from the remote server.

Client class names are created from the WSDL port element attribute name, plus the suffix "Client". So, from the port element:

```
<port name=" SrvcSearchRPC"../>
```

the client class is named SrvcSearchRPCClient and the class is derived from the WsdlClient class. Based on the WSDL operation description above, the following methods would be generated in WSClient:

RPC style:

```
initialize
    self setPortNamed: 'SrvcSearchRPC'.

searchByExactTitle: aString includeAffiliatedLibraries: aBoolean
"Generated by WS Tool on #(February 21, 2003 6:50:37 am)"
    "operationName: #SearchByExactTitle"
    "documentation: #'The SearchByExactTitle operation returns a
        collection of holdings or empty collection if no holdings found'"
    "addParameter: #holding_title type: #'String'"
    "addParameter: #includeAffiliatedLibraries type: #'Boolean'"
    "result: #( #Collection #'WSLDHoldingBook' )"
    | args |
    args := Array new: 2.
    args at: 1 put: aString.
    args at: 2 put: aBoolean.
    ^self executeSelector: #'searchByExactTitle:includeAffiliatedLibraries:'
        args: args.
```

Document-literal style:

```
initialize
    self setPortNamed: 'SrvcSearchDoc'.
```

```
searchByExactTitle: aStruct
"Generated by WS Tool on #(February 21, 2003 6:50:31 am)"
    "operationName: #SearchByExactTitle"
    "documentation: #'The SearchByExactTitle operation returns a
        collection of holdings or empty collection if no holdings found'"
    "addParameter: #holding_title type: #'String'"
    "addParameter: #includeAffiliatedLibraries type: #'Boolean'"
    "result: #( #Collection #'WSLDHoldingBookDoc' )"
    | args |
    args := Array new: 1.
    args at: 1 put: aStruct.
    ^self executeSelector: #'searchByExactTitle:' args: args.
```

Service Classes

Service classes represent the web service ports, from a service provider perspective. They are named using the binding name specified in the <binding name=... >.

The generated service classes include stub methods for the protocol defined in the specification, but no implementation for those methods (because the WSDL does not include implementation). The methods also include pragmas, which, along with the binding classes, are used by WsdlBuilder when generating a WSDL schema from the implementation.

4

Document Processing

This chapter explores and elaborates the lower-level details of WSDL document processing in the VisualWorks web services framework.

The WsdlClient, the web services Wizard, and the WsdlClassBuilder can all load and parse WSDL documents to generate a Smalltalk binding schema, which describes a mapping between Smalltalk objects and the elements described in the WSDL document.

This chapter explores the web services APIs for:

- Loading a WSDL Schema
- Saving a Schema with its Binding
- Generating XML-to-object Bindings
- Customizing Mappings
- Generating Bindings without a WSDL Document

Working with WSDL Schemas

The VisualWorks web services framework provides some lower-level messaging protocol in WsdlClient for loading and parsing WSDL schemas.

Loading a WSDL Schema

To load a WSDL schema and generate XML-to-object bindings, send an url: instance creation message to WsdlClient, with its URL as argument:

```
wsdlClient := WsdlClient url:
    'http://www.capeclear.com/AirportWeather.wsdl'.
```

WsdlClient uses WsdlSchemaLoader to load the schema and schema parts.

Generating XML-to-object Bindings

If the schema declares user defined types in a <types> section, these are used to create XML-to-object bindings. VisualWorks supports two types of binding:

- default, which maps complex XML data type to Smalltalk dictionaries
- object, which maps complex XML data type to Smalltalk objects

WsdlClient uses the default mapping; WsdlClassBuilder and the Wizard give you the option or using either. WsdlSchemaLoader can be invoked directly for these mappings by sending, for the default mappings:

```
WsdlSchemaLoader defaultReadFrom: aDataSource.
```

and for the object mappings:

```
WsdlSchemaLoader objectReadFrom: aDataSource
```

WsdlSchemaLoader creates the <schemaBindings> element and writes default mappings of XML elements to elements that will be used to create XML marshalers. (The default mappings are described in XML to Smalltalk Mapping)

For example, consider this element from the <schema> section of the WSDL document retrieved above:

```
<xsd:complexType name="WeatherSummary">
   <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" name="location"
         nillable="true" type="xsd:string" />
      <xsd:element maxOccurs="1" minOccurs="1" name="wind"
         nillable="true" type="xsd:string" />
      <xsd:element maxOccurs="1" minOccurs="1" name="sky"
         nillable="true" type="xsd:string" />
      <xsd:element maxOccurs="1" minOccurs="1" name="temp"
         nillable="true" type="xsd:string" />
      <xsd:element maxOccurs="1" minOccurs="1" name="humidity"
         nillable="true" type="xsd:string" />
      <xsd:element maxOccurs="1" minOccurs="1" name="pressure"
         nillable="true" type="xsd:string" />
      <xsd:element maxOccurs="1" minOccurs="1" name="visibility"
         nillable="true" type="xsd:string" />
   </xsd:sequence>
</xsd:complexType>
```

The default mapping maps this complexType element to a <struct> element in the <schemaBindings> section:

```
<struct name="WeatherSummary">
   <element maxOccurs="1" minOccurs="1" name="location"
      nillable="true" ref="xsd:string"/>
   <element maxOccurs="1" minOccurs="1" name="wind"
      nillable="true" ref="xsd:string"/>
   <element maxOccurs="1" minOccurs="1" name="sky"
      nillable="true" ref="xsd:string"/>
   <element maxOccurs="1" minOccurs="1" name="temp"
      nillable="true" ref="xsd:string"/>
   <element maxOccurs="1" minOccurs="1" name="humidity"
      nillable="true" ref="xsd:string"/>
   <element maxOccurs="1" minOccurs="1" name="pressure"
      nillable="true" ref="xsd:string"/>
   <element maxOccurs="1" minOccurs="1" name="visibility"
      nillable="true" ref="xsd:string"/>
</struct>
```

The default Smalltalk marshaler for the <struct> element marshals this element as a Dictionary, as explained in XML to Smalltalk Mapping.

The object mapping, for comparison, would be:

```
<object name="WeatherSummary" smalltalkClass="WeatherSummary">
   <element maxOccurs="1" minOccurs="1" name="location"
nillable="true"
      ref="xsd:string" aspect="location"/>
   <element maxOccurs="1" minOccurs="1" name="wind" nillable="true"
      ref="xsd:string" aspect="wind"/>
   <element maxOccurs="1" minOccurs="1" name="sky" nillable="true"
      ref="xsd:string" aspect="sky"/>
   <element maxOccurs="1" minOccurs="1" name="temp" nillable="true"
      ref="xsd:string" aspect="temp"/>
   <element maxOccurs="1" minOccurs="1" name="humidity"
nillable="true"
      ref="xsd:string" aspect="humidity"/>
   <element maxOccurs="1" minOccurs="1" name="pressure"
nillable="true"
      ref="xsd:string" aspect="pressure"/>
   <element maxOccurs="1" minOccurs="1" name="visibility"
nillable="true"
      ref="xsd:string" aspect="visibility"/>
</object>
```

When bindings have been created, they should be reviewed, and possibly custom mappings created. In general, you would save the specification, make any customizations, and then load the schema, as shown in the following sections.

While loading the WSDL schema, the XMLObjectBinding can raise a notification if there are any XML elements that are not resolved. The notifications are collected in the BindingReport objects and should be reviewed before sending a SOAP request.

```
BindingReport report inspect
```

The result may be list of warnings, such as:

```
"Warning in: AnyRelationMarshaler>>unmarshalFrom:do:
There is no defined marshaler for the node tag: <suds:class>"
```

Any warnings about the XML elements that will not be used in the SOAP request can be ignored.

Saving a Schema with its Binding

To preserve the generated bindings with the schema, save it with the binding. The schema can be saved to a file, a method by evaluating:

```
wsdlClient saveSchemaDocuments
```

This opens a file dialog prompting for a file name (e.g., AirportWeather.wsdl), and writes the schema to the file, including the bindings in the <schemaBindings> section at the end.

Alternatively, send saveDocumentsIntoFile: to the client:

```
wsdlClient saveDocumentsIntoFile: filename
```

to save the schema into a file, or:

```
wsdlClient saveSchemaDocumentsIntoMethod: aMethodSelector
    class: aClassName withComment: aString
```

to save the schema in a class method.

In both cases, if there are more than one WSDL document, the first document is named as specified, and successive documents are given a suffix, for example, AirportWeather.wsdl1, AirportWeather.wsdl12, ...

Load and Use a WSDL Schema

To reuse a saved schema, load it from the file:

```
wsdlClient := WsdlClient fileName: 'AirportWeather.wsdl'.
```

or from a stream:

```
wsdlClient := WsdlClient readFrom:
    'AirportWeather.wsdl' asFilename readStream lineEndTransparent.
```

While loading a WSDL schema, the following registries are created:

XMLObjectBinding registry

Includes XML to object marshalers.

WsdlBinding wsdlBindings

Includes WSDL configurations for all web services.

WsdlPort portRegistry

Includes all port descriptions.

WsdlService registry

Includes all web services.

Customizing Mappings

After saving the schema with the bindings, you should review the default mappings and possibly create custom mappings. For example, instead of the default mapping for <struct> to a Dictionary, an element can be created mapping it to a Smalltalk class, such as WeatherSummary. This simply involves adding a smalltalkClass= attribute, with the class name enclosed in quotation marks, e.g.:

```
<object name="WeatherSummary" smalltalkClass="WeatherSummary">
  <element maxOccurs="1" minOccurs="1" name="location"
    nillable="true" ref="xsd:string" aspect="location"/>
  <element maxOccurs="1" minOccurs="1" name="wind"
    nillable="true" ref="xsd:string" aspect="wind"/>
  <element maxOccurs="1" minOccurs="1" name="sky"
    nillable="true" ref="xsd:string" aspect="sky"/>
  <element maxOccurs="1" minOccurs="1" name="temp"
    nillable="true" ref="xsd:string" aspect="temp"/>
  <element maxOccurs="1" minOccurs="1" name="humidity"
    nillable="true" ref="xsd:string" aspect="humidity"/>
  <element maxOccurs="1" minOccurs="1" name="pressure"
    nillable="true" ref="xsd:string" aspect="pressure"/>
  <element maxOccurs="1" minOccurs="1" name="visibility"
    nillable="true" ref="xsd:string" aspect="visibility"/>
</object>
```

Once the schema is updated and saved, you can reuse it to make requests.

Making a Request with a WSDL Schema

The WSDL schema may specify several services. To make a request, specify the service and port:

```
wsdlClient port:
    (wsdlClient getPortAt: wsdlClient config targetNamespace
    fromService: wsdlClient services first).
```

If a port is not specified, the default port will be used, which is the first port from the first service.

```
wsdlClient port: client config anyPort
```

Using this information, as well as the message definitions in the WSDL schema, you can create a SOAP request (see SOAP Exchanges):

```
| wsdlClient soapRequest  |
    wsdlClient := WsdlClient fileName: 'AirportWeather.wsdl'.
    wsdlClient port: client config anyPort.
    soapRequest := SoapRequest new.
    soapRequest port: wsdlClient config anyPort.
    soapRequest smalltalkEntity:
        (Message selector: #getHumidity argument: 'KSNA').
    ^wsdlClient executeRequest: soapRequest.
```

Generating Bindings without a WSDL Document

Sometimes you don't have a full WSDL document, but only a <types> section, and still want to generate XML object bindings and even generate marshaler classes. This can be done using the facilities described in this section.

You can invoke the XMLTypesParser directly to generate the <xmlToSmalltalkBinding> element of a binding schema. The way you invoke the parser depends on how you want complex types to be mapped, whether to dictionaries or specific Smalltalk classes. The resulting bindings are processed differently to generate the dictionaries or classes, and then accessed differently for marshaling and unmarshaling. This section gives separate descriptions for these two approaches to mapping.

For the code examples given below, consider this fragment, containing only a <types> section:

```
xmlTypes := '<?xml version="1.0" encoding="UTF-8"?>
<wsdl:types xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" >
    <schema targetNamespace="urn:vwservices"
```

```
        xmlns="http://www.w3.org/2001/XMLSchema"
        elementFormDefault="qualified"
        xmlns:ns="urn:vwservices"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <complexType name="Customer">
          <sequence>
            <element name="name" minOccurs="1" maxOccurs="1"
              type="xsd:string" />
            <element name="id" minOccurs="1" maxOccurs="1"
              type="xsd:int" />
            <element name="address" minOccurs="1" maxOccurs="1"
              type="ns:Address" />
          </sequence>
        </complexType>
        <complexType name="Address">
          <sequence>
            <element name="street" minOccurs="1" maxOccurs="1"
              type="xsd:string" />
            <element name="state" minOccurs="1" maxOccurs="1"
              type="xsd:string" />
            <element name="zip" minOccurs="1" maxOccurs="1"
              type="xsd:int" />
          </sequence>
        </complexType>
      </schema>
    </wsdl:types>'.
```

Complex Type to Dictionary Bindings

The default binding for complex types is to map them to instances of Dictionary, as specified in a <struct> section in the binding schema.

Generating the Binding Schema

To generate a schema mapping complex types to dictionaries, send a readFrom: message to the parser class with a ReadStream on the types fragment text:

```
xmlToObjElement := XMLTypesParser readFrom: xmlTypes readStream.
```

This returns an Element holding the following <schemaBindings> section:

```
<schemaBindings>
  <xmlToSmalltalkBinding elementFormDefault="qualified" name=""
    targetNamespace="urn:vwservices"
      xmlns="urn:visualworks:VWSchemaBinding"
      xmlns:ns="urn:vwservices"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
        <struct name="Customer">
            <element maxOccurs="1" minOccurs="1" name="name"
                ref="xsd:string"></element>
            <element maxOccurs="1" minOccurs="1" name="id"
                ref="xsd:int"></element>
            <element maxOccurs="1" minOccurs="1" name="address"
                ref="ns:Address"></element>
        </struct>
        <struct name="Address">
            <element maxOccurs="1" minOccurs="1" name="street"
                ref="xsd:string"></element>
            <element maxOccurs="1" minOccurs="1" name="state"
                ref="xsd:string"></element>
            <element maxOccurs="1" minOccurs="1" name="zip"
                ref="xsd:int"></element>
        </struct>
    </xmlToSmalltalkBinding>
</schemaBindings>
```

Note that the complex types from the types document are represented as <struct> elements.

Creating the Binding Dictionaries

Given the bindings schema, you build the bindings, including the relevant dictionaries, by sending a buildBindings: message, with the schema as argument, to class XMLObjectBinding:

```
binding := (XMLObjectBinding buildBindings:
    (Array with: xmlToObjElement )) first.
```

The result is an XMLObjectBinding defining marshalers for Customer and Address, each with a Struct as the representative Smalltalk class. To browse the binding, you can browse XMLBindingRegistry, a shared variable of class XMLObjectBinding, and select urn:vwservices.

Marshaling and Unmarshaling a Struct

The object binding provides the mechanism necessary to marshal (represent a Smalltalk object in XML) and unmarshal (represent an XML object in Smalltalk) objects.

To marshal an object of a complex type, you first construct the object, which is an instance of Dictionary. In the current example, there are two complex objects, Customer and Address, and an Address is a component of a Customer. Both are structs, so each is represented as

a Dictionary, in this case a Customer dictionary holding an Address dictionary in its address entry. You might construct this structure as follows:

```
cust := Dictionary new.
cust
   at: #name put: 'bob';
   at: #id put: 123;
   at: #address put:
         (Dictionary new
            at: #street put: 'street';
            at: #state put: 'state';
            at: #zip put: 123456;
            yourself).
```

To marshal this dictionary as a Customer, create an XMLObjectMarshalingManager on the binding, get the marshaler for the Customer data type, and then marshal the object:

```
manager := XMLObjectMarshalingManager on: binding.
marshaler := manager marshalerForType: 'Customer' ns:
'urn:vwservices'.
xmlres := manager marshal: cust  with: marshaler.
```

The result is an XML element:

```
<ns:Customer xsi:type="ns:Customer" xmlns:ns="urn:vwservices"
   xmlns:ns0="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <ns:name xsi:type="ns0:string">xxx</ns:name>
   <ns:id xsi:type="ns0:int">123</ns:id>
   <address xsi:type="ns:Address">
      <ns:street xsi:type="ns0:string">street</ns:street>
      <ns:state xsi:type="ns0:string">state</ns:state>
      <ns:zip xsi:type="ns0:int">123456</ns:zip>
   </address>
</ns:Customer>
```

Conversely, given an Customer XML element, unmarshaling it into a dictionary can be done by sending:

```
cust1 := manager unmarshal: xmlres.
```

which returns the corresponding Struct.

Complex Type to Object Bindings

For extensive processing within a Smalltalk application, it is frequently better to represent complex types as instances of corresponding Smalltalk classes. Instead of mapping to dictionaries, you can map complex types to Smalltalk objects.

Generating the Binding Schema

To generate a schema binding for mapping to objects, send a useObjectBindingReadFrom:inNamespace: message to XMLTypesParser. A ReadStream on the contents of the <types> section fragment is the first argument; the second argument is a String specifying the Smalltalk name space:

```
xmlToObjElement := XMLTypesParser
    useObjectBindingReadFrom: xmlTypes readStream
    inNamespace: 'Smalltalk'.
```

This returns an Element object containing a <schemaBindings> section:

```
<schemaBindings>
    <xmlToSmalltalkBinding defaultClassNamespace="Smalltalk"
        elementFormDefault="qualified"
        name="" targetNamespace="urn:vwservices"
        xmlns="urn:visualworks:VWSchemaBinding"
        xmlns:ns="urn:vwservices"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <object name="Customer" smalltalkClass="Customer">
            <element aspect="name" maxOccurs="1" minOccurs="1"
                name="name" ref="xsd:string"></element>
            <element aspect="id" maxOccurs="1" minOccurs="1" name="id"
                ref="xsd:int"></element>
            <element aspect="address" maxOccurs="1" minOccurs="1"
                name="address" ref="ns:Address"></element>
        </object>
        <object name="Address" smalltalkClass="Address">
            <element aspect="street" maxOccurs="1" minOccurs="1"
                name="street" ref="xsd:string"></element>
            <element aspect="state" maxOccurs="1" minOccurs="1"
                name="state" ref="xsd:string"></element>
            <element aspect="zip" maxOccurs="1" minOccurs="1" name="zip"
                ref="xsd:int"></element>
        </object>
    </xmlToSmalltalkBinding>
</schemaBindings>
```

Note that the complex types from the types document are represented as <object> elements.

Creating the Binding Classes

Given the bindings schema, you build the bindings, including the relevant classes, by sending a createClassesFromBindings: message, with a collection of schemas as argument, to an instance of BindingClassBuilder. The builder should be given a package name in which to put the generated classes:

```
builder := BindingClassBuilder new.
builder package: 'WSTest'.
builder createClassesFromBinding:
   (OrderedCollection with: xmlToObjElement).
```

The result is a collection of classes built for the complex types. The classes include accessor methods for setting and getting the values of the classes' attributes.

Marshaling and Unmarshaling the Objects

The object binding provides the mechanism necessary to marshal (represent a Smalltalk object in XML) and unmarshal (represent an XML object in Smalltalk) objects.

To marshal an object of a complex type, you first construct instances of the representing class, which in this case are instances of Address and Customer, and an Address is a component of a Customer. You might construct a Customer instance as follows:

```
cust := Smalltalk.Customer new.
cust
   name: 'bob';
   id: 123;
   address: (Smalltalk.Address new
            street: 'street';
            state: 'state';
            zip: 123456;
            yourself).
```

To marshal the object as an XML element, send a marshal:atNamespace: message to XMLObjectMarshalingManager, with the object and name space as arguments:

```
xmlres := XMLObjectMarshalingManager
      marshal: cust
      atNamespace: 'urn:vwservices'.
```

The result is an XML element representing the object:

```
<ns:Customer xsi:type="ns:Customer" xmlns="urn:vwservices"
   xmlns:ns="urn:vwservices"
   xmlns:ns0="http://www.w3.org/2001/XMLSchema"
```

```
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <ns:name xsi:type="ns0:string">xxx</ns:name>
        <ns:id xsi:type="ns0:int">123</ns:id>
        <address xsi:type="ns:Address">
            <ns:street xsi:type="ns0:string">street</ns:street>
            <ns:state xsi:type="ns0:string">state</ns:state>
            <ns:zip xsi:type="ns0:int">123456</ns:zip>
        </address>
    </ns:Customer>
```

Conversely, given an Customer XML element, unmarshaling it into a Customer instance can be done by sending:

```
cust1 := XMLObjectMarshalingManager
    unmarshal: xmlres printString readStream
    atNamespace: 'urn:vwservices'
```

which returns the corresponding object.

5

SOAP Exchanges

While many applications using web services can be built solely with WSDL, there are occasions when you may need to use SOAP as well. SOAP (Simple Object Access Protocol) is an XML-based protocol for exchanging structured and typed messages in a distributed and heterogeneous environment.

While it may be compared with technologies like CORBA and DCOM, SOAP is a vendor-neutral technology for exchanging messages via HTTP. Historically, SOAP may be understood as a successor to XML-RPC.

Where WSDL is concerned with a service description (which is embodied in a WSDL schema), SOAP is concerned with the actual details of exchanging messages. That is, it specifies the envelope (message formats for data exchange), request/response handshaking, and protocol binding.

WSDL was designed to use HTTP/REST or MIME as messaging protocols, but it generally uses SOAP and HTTP. Thus, most web services applications use WSDL and SOAP together.

This chapter discusses:

- Building a SOAP Request using a WSDL Schema
- SOAP Messaging without WSDL
- SOAP Headers
- Sending Requests over Persistent HTTP
- SOAP Exception Handling

VisualWorks Implementation

Support for WSDL in the VisualWorks web services framework greatly simplifies SOAP messaging by automatically producing the appropriate SOAP message. Generally, the web services framework produces SOAP messages transparently, by using transformations based on a WSDL schema, making web service development almost trivial.

However, for situations where SOAP is needed without a WSDL schema, you can still write directly to the SOAP protocols. Both approaches are described in this chapter.

When WSDL is used in conjuction with SOAP, two general forms of SOAP messaging are available: RPC- and Document-literal style.

At the core of SOAP support in VisualWorks is the XML-to-object mapping mechanism (described in XML to Smalltalk Mapping). Using a SOAP binding, Smalltalk messages are marshaled into a SOAP/XML representation for communication to a service provider, and the XML response is unmarshaled back into Smalltalk.

Loading SOAP Support

To load SOAP support, use the Parcel Manager to load the WSDL parcel. This will automatically load the SOAP parcel, as well as the XML-to-Object support parcels (for details, see Loading Support for Web Services).

SOAP Messaging Framework

A SOAP message is an XML document with a mandatory envelope, an optional header, and a mandatory body. The structure of such a message is described in detail in the SOAP specifications, and is not covered here.

The three parts of a SOAP message are modeled in VisualWorks using SoapEnvelope, SoapBodyStruct, and SoapHeaderStruct. You seldom need to deal with instances of these classes directly, since they are intermediate objects used by the SOAP marshaler.

SOAP messages and their components are modeled by the following classes:

```
SoapMessage
    SoapRequest
    SoapResponse
```

For placing a service request, you build and execute a SoapRequest. The response, if successful, comes as a SoapResponse, though the request execution generally returns a more useful object, as illustrated below.

Building a SOAP Request using a WSDL Schema

The easiest way to use SOAP in VisualWorks is by requesting services for which there is a WSDL document describing those services. Because there is a SOAP binding for WSDL, VisualWorks can provide bindings to map a Smalltalk object (specifically, instances of Message) to the appropriate SOAP messages. By specifying the mappings between a WSDL schema and Smalltalk in a transformation method (or other mechanism), the task of constructing the appropriate SOAP messaging is automated.

For example, the VisualWorks WebServicesTimeDemo includes services that each provide a WSDL document. When running the demo on your workstation, the schema is available at this URL:

http://localhost:4950/TimeNowService?wsdl

(For a selection of example services with WSDL schemas, see http://xmethods.net.

From this document we discover that the service supports a request message named TimeNow which takes no arguments.

To prepare and send a request, we can evaluate the following code:

```
| wsdlClient soapRequest soapResponse |
    wsdlClient := WsdlClient new loadFrom:
        'http://localhost:4950/TimeNowService?wsdl' asURI.
    soapRequest := SoapRequest new.
    soapRequest port: wsdlClient config anyPort.
    soapRequest smalltalkEntity:
        (Message selector: #TimeNow).
    soapResponse := soapRequest value.
```

Among other things, the WSDL schema document defines the ports providing the available services. In the code example shown above, the first line creates a WsdlClient containing a configuration, an instance of WsdlConfiguration, from the WSDL document.

The next line creates a new, empty SOAP request:

```
soapRequest := SoapRequest new.
```

Then, we add the parts to the request that are needed to send it. First, assign a port to identify a service. The anyPort message retrieves the first port of the first service in the WsdlConfiguration, which is often sufficient:

```
soapRequest port: wsdlClient config anyPort.
```

Next, the Smalltalk message to be marshaled into the SOAP request is added, by sending a smalltalkEntity: message to the SoapRequest. The Smalltalk entity will be a Message, which is created by sending the selector: instance creation method:

```
soapRequest smalltalkEntity:
    (Message selector: #TimeNow).
```

The selector must match *exactly* the operation name in the interface definition. To get the possible message selectors, you can inspect:

```
wsdlClient config interfaces first operations
```

This gives an OrderedCollection of WsdlOperationDescriptor instances. If the message takes arguments, you can dig down into an operation to find its input. Consult the service's documentation for information about what the argument values should be.

With that information provided, we can now ask for the value of the request (the message #value actually causes the request to be sent to the server):

```
soapResponse := soapRequest value.
```

The result will be an instance of SoapResponse.

In summary, this code example marshalls the request as a SOAP message, sending the request, and returning the value of the response.

You can inspect the result of this example (evaluate with **Inspect it**).

Messages with Arguments

Most service messages require argument values. To provide arguments, you may use executeSelector:args:. Alternately, you can create the soapEntity for the request by sending a selector:arguments: instance creation message to class Message. In each case, the arguments are passed in an Array with the appropriate content.

The content of this array depends on the request binding style; SOAP requests can be in either an RPC or a Document style.

RPC-style Message Arguments

For an RPC-style binding, the soap message can have zero or many arguments that are sent in an Array of elements. For example, the WebServicesTimeDemo includes a service that accepts a Time object as an argument, returning a Timestamp object. The following code snippet illustrates this style of binding.

After starting the TimeServerRPC, evaluate the following using **Inspect It**:

```
| wsdlClient arguments value |
wsdlClient := WsdlClient new loadFrom:
    'http://localhost:4952/TimeNowServiceRPC?wsdl' asURI.
arguments := Array with: Time now.
value := wsdlClient executeSelector: #AsTimestamp args: arguments.
```

(This code may be found in the class-side method wsdlClientScript of TimeServerRPC.)

We can also illustrate RPC-style binding with a real-world example.

The BabelFish translation service may be used with RPC-style binding. The request takes two arguments: a translation code and a string to translate, both as strings. Thus, to translate from English to French, we use the directive string **'en_fr'**. This example also shows how to use a SoapRequest object directly:

```
| wsdlClient soapRequest |
wsdlClient := WsdlClient new loadFrom:
    'http://www.xmethods.net/sd/BabelFishService.wsdl' asURI.
soapRequest := SoapRequest new.
soapRequest port: wsdlClient config anyPort.
soapRequest smalltalkEntity:
    (Message selector: #BabelFish arguments:
        #('en_fr' 'this is a test')).soapRequest value.
```

Document-style Message Arguments

For a Document binding style, the SOAP message body must have either zero or one parts. To pass arguments to a document-style service, the arguments should be provided as Dictionary or a web services Struct.

The result of a document style request is an OrderedCollection with one element.

For example, the demo TimeServer may be invoked with document-style binding by evaluating the following code:

```
wsdlClient := WsdlClient new loadFrom:
    'http://localhost:4950/TimeNowService?wsdl' asURI.
struct := WebServices.Struct new.
arguments := Array with: struct.
struct at: #asTimestamp put: Time now.
value := wsdlClient executeSelector: #AsTimestamp args: arguments.
```

SOAP Messaging without WSDL

A SOAP message is an XML document, so you can, in principle, create a message simply by constructing the XML document and sending it. VisualWorks simplifies that, even without the use of a WSDL schema, by allowing you to create the SoapRequest. You need to specify the binding, and explicitly provide input and output marshalers, but even without the schema this is much simpler than creating raw XML.

For example, suppose we want to make the SOAP echo request:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
    xmlns:m="http://soapinterop.org/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
        <m:echoInteger>
            <inputInteger>123</inputInteger>
        </m:echoInteger>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

First, we must build an object binding. Since an integer is being passed and echoed, the marshaler for type int is required.

```
m := XMLObjectBinding soapBinding
    marshalerForTag:
        (XML.NodeTag new qualifier: ''
            ns: 'http://www.w3.org/2001/XMLSchema'
            type: 'int')
        ifAbsent: [nil].
```

A SoapOperationBinding provides marshalers for the SOAP operation:

```
soapOperBinding  := SoapOperationBinding new
    name: 'echoInteger';
    style: 'rpc'.
```

Next, create the input marshaler and assign it to the SoapOperationBinding. The marshaler will be an instance of SoapParameterMarshaler.

```
(inputMarshaler := SoapParameterMarshaler new)
    structName: (XML.NodeTag new
        qualifier: 'm'
        ns: 'http://soapinterop.org/'
        type: 'echoInteger');
    style: 'rpc'.
inputMarshaler
    from: (Array with:
        ((XML.NodeTag new qualifier: '' ns: '' type: 'inputInteger') -> m ))
    use: ''
    encodingStyle: ''
    order: nil.
soapOperBinding inputMarshaler: inputMarshaler.
```

The from:use:encodingStyle:order: message assigns the arguments to the SOAP message, as an Array as was the case when using WSDL.

Similarly, create and assign the output marshaler:

```
outMarshaler := SoapParameterMarshaler new
    structName: (XML.NodeTag new
        qualifier: ''
        ns: 'http://soapinterop.org/'
        type: 'echoIntegerResponse');
    style: 'rpc'.
outMarshaler
    from: (Array with:
        ((XML.NodeTag new qualifier: '' ns: '' type: 'return') -> m))
    use: ''
    encodingStyle: ''
    order: nil.
soapOperBinding outputMarshaler: outMarshaler.
```

Fault marshalers also must be specified, but can be empty:

```
soapOperBinding faultMarshalers: OrderedCollection new.
```

Given this, we can build a SOAP request, as follows:

```
req := SoapRequest new
    binding: XMLObjectBinding soapBinding;
    transport: (SoapHttpBindingDescriptor
        verb: 'POST' soapAction: 'http://soapinterop.org/');
    accessPoint:
        'http://www.dolphinharbor.org/services/interop2001' asURI;
    operation: soapOperBinding;
    smalltalkEntity: (Message selector: #echoInteger arguments: #(123));
    execute.
resp := req value.
```

SOAP Headers

SOAP Header blocks are used to extend an application with additional features. Extensions that can be implemented as header entries include authentication, transaction management, payments, and so on.

Currently, support for SOAP headers in VisualWorks is compliant with SOAP 1.1. This supports processing a WSDL schema with SOAP headers, and marshaling and unmarshaling SOAP messages with headers.

The SoapMessage class holds its header entries in the header instance variable, which holds a Dictionary (a SoapHeaderStruct), with the header entry name as the key, and an instance of SoapHeaderEntry as its value. This corresponds to the SOAP specification in which a header is the first immediate child element of a SOAP envelope, and header entries are immediate child elements of the header.

Sending SOAP Messages with Header Entries

The WebServicesTimeDemo includes a simple example that illustrates the use of SOAP headers: class HTimeServer. This is similar to class TimeServer, which was described previously (for details, see Web Services Examples), but it also includes additional code to invoke a header processor class.

For example, to test the HTimeServer using an instance of HTimeClient, evaluate the following code in a Workspace window using **Do It** (from the <Operate> menu):

```
WebServices.HTimeServer defaultStart.
client := WebServices.HTimeClient new.
client start.
(client headerFor: #password) value: 'myPassword'.
“Request the current time from the server”
result := client timeNow.
“Stop the client and server”
client stop.
WebServices.HTimeServer defaultStop.
```

After starting both the server and client, we attach a header entry to the client's requests by sending headerFor:, with the name of the new header entry as a symbol. A value, the object to be marshaled, is assigned to the header by sending value:.

The result in the variable "result" should be an OrderedCollection containing a Time object (you can examine this variable using **Inspect It**).

Using SOAP Header Entries with WsdlClient

When using WsdlClient to make SOAP requests, you can add header entries using the same API, or assemble the SoapMessage in your application code. To illustrate, we can again query HTimeServer, but this time using class WsdlClient to make and send the request with a SOAP header:

```
WebServices.HTimeServer defaultStart.
client := WsdlClient url:
    'http://localhost:4954/TimeNowServiceWithHeaders?wsdl'.
(client headerFor: #password) value: 'myPassword'.
result := client executeSelector: #TimeNow.
WebServices.HTimeServer defaultStop.
```

Here, the header is again created in the client by sending headerFor:, and a value assigned, but this time in the WsdlClient. The request is then constructed and sent using executeSelector:.

Using Soap Header Entries with an Opentalk Client

To illustrate, we will query AuthSearchServer from WebServicesDemo. The examples below can be found in the TestSoapHeaders class.

The WSAuthenticatedSearchService class provides some services that require SOAP headers for authentication. The operation pragmas should include a description (inputHeader/outputHeader) for the header processors. The header processors must provide the header type description in the class methods: headerTag and header. The header processors must implement methods to process header objects.

```
client := ClientForAuthenticatedSearchService new.
    client start.
```

Add the #AccessLevel header to the request

```
( client headerFor: #'AccessLevel')value: '12345'.
```

Add the AuthenticationToken header to the request. The provided user ID and password are accepted by the header processor on the server.

```
( client headerFor: #'AuthenticationToken')
    value: ( WebServices.AuthenticationToken new
        userID: 'UserID';
        password: 'password';
        yourself).
value := (client authenticatedSearchByWord: 'and') first.
```

When the Opentalk server receives a request with headers it unmarshals the headers and then sends the header objects to the corresponding processor. For an example, see processInputHeader:transport: in class HdPrAuthenticationToken.

Handling a Requst with Wrong Parameters.

Add the #AuthenticationToken header to the request. The provided user ID and password are NOT accepted by the header processor on the server.

```
( client headerFor: #'AuthenticationToken')
    value: ( WebServices.AuthenticationToken new
        userID: 'ID#ABC';
        password: 'password';
        yourself).
```

The server returns a reply with

```
Confirmation header value: <not confirmed>
```

The client header processor raises the WrongConfirmationExc exception. See processOutputHeader:reply:transport: in HdPrConfirmation.

```
[client authenticatedSearchByWord: 'and']
    on: WrongConfirmationExc
    do: [ :ex | 'do some processing' ]
```

Setting the Result Type

The Opentalk client can be set to return different result types. The default return type is specified by the WSDL schema. The requests from the testServiceWithHeaders return the default types.To return an instance of WebServices.SoapEnvelope set the returned type to #envelope:

```
client returnedObject: #envelope.value := client
authenticatedSearchByWord: 'and'.
```

To return an instance of WebServices.SoapResponse set the returned type to #response:

```
client returnedObject: #response.value := client
authenticatedSearchByWord: 'and'.
```

Accessing Soap Headers from Service Methods

The SOAP headers are accessible in services methods. The server can set an option and the header object will be added to the ProcessEnviroment. For example:

```
broker := (opentalkServer interfaces at: 'localhost:4922') broker.broker
marshalerConfiguration environmentWithHeaders: true.
```

The service method can reach the header object as:

```
WSAuthenticatedSearchService>>authenticatedSearchByTitle:
   | header |
   header := ProcessEnvironment current
      at: #AuthenticationToken
      ifAbsent: [self error: 'There is no such header in the
         ProcessEnvironment current'].
```

Creating a SOAP Header

A SOAP header consists of one or more SOAP header entries, each of which is an instance of SoapHeaderEntry. Each header entity holds four values in its instance variables:

name

The local name of the entry element. The name space path is taken from the WSDL document. (Note that this variable is often not set because it is not used by the marshaler, which uses the keys of the SoapHeaderStruct instead.)

value

The object to marshal.

actor

The actor attribute specifies the recipient SOAP node of a header element. The value of the SOAP actor attribute is a URI. The special URI "http://schemas.xmlsoap.org/soap/actor/next" indicates that the header element is intended for the very first SOAP application that processes the message. If no actor is specified, the actor is understood to be the ultimate destination of the SOAP message.

mustUnderstand

The mustUnderstand attribute Indicate whether a header entry is mandatory or optional for the recipient to process. The value of the mustUnderstand attribute is either true or false. The absence of the SOAP mustUnderstand attribute is semantically equivalent to false.

The following SoapHeaderEntry instance creation methods set these values:

value: *anObject*

Creates new header entry with the specified value. The actor and mustUnderstand attributes are not set.

value: *anObject* **mustUnderstand:** *aBoolean* **actor:** *aString*

Creates new header entry with the specified value, and actor and mustUnderstand attributes.

targetRecipientValue: *anObject*

Creates new header entry with the specified value, the actor as "http://schemas.xmlsoap.org/soap/actor/next", and mustUnderstand as nil.

targetRecipientMU1Value: *anObject*

Creates new header entry with the specified value, actor set to "http://schemas.xmlsoap.org/soap/actor/next", and mustUnderstand set to true.

targetReceipientMU0Value: *anObject*

Creates new header entry with the specified value, actor set to "http://schemas.xmlsoap.org/soap/actor/next", and mustUnderstand set to false.

Header entries are held in the header variable (in a SoapHeaderStruct dictionary) of a SoapMessage instance. Use the following SoapMessage (SoapRequest or SoapResponse) accessor messages to get or set header entries:

headerFor: *aSymbol*

Creates new header entry with the specified name aSymbol

header

Returns a dictionary of all header entries

headerAt: *aSymbol* **put:** *aSoapHeaderEntry*

Adds new header entry to the message

headerAt: *aSymbol* **ifAbsentPut:** *aBlock*

Returns the header entry aSymbol, or adds new header entry to the message header if it does not already exist.

headerRemoveKey: *aSymbol* **ifAbsent:** *aBlock*

Removes the header entry from the message.

This API is duplicated in class WsdlClient (implemented in SoapClient). The WsdlClient holds its own header dictionary. When the client sends the request, the header entries are added to the request as follows:

1 If the header entry exists in both WSDL client and SOAP request dictionaries, the SOAP request header entry is used.

2 If the header entry exists in WSDL client but not in the SOAP request, the WSDL client header entry is added to the request.

Sending Requests over Persistent HTTP

The typical SOAP session over HTTP sends several requests to the same server, opening a new HTTP connection for each request.

```
wsdlClient := WsdlClient url:
    'http://www.dolphinharbor.org/services/interop2001/service.wsdl'.
value := wsdlClient executeSelector: #echoInteger args: #(123).
wsdlClient executeSelector: #echoString args: #('ABc 46').
```

If you are using a secure HTTP connection this is not satisfactory, because it can take a fairly long time to negotiate the connection. To handle this, WsdlClient provides an API to open a persistent connection and reuse it. Send a connectToHost: message to establish the connection, and then send your requests.

For example:

```
wsdlClient := WsdlClient url:
    'http://www.dolphinharbor.org/services/interop2001/service.wsdl'.
"connecting to the Http server"
[wsdlClient connectToHost: 'www.dolphinharbor.org']
    on: Exception do: [:ex | wsdlClient reconnect. ex return].
"executing a few requests and close the Http connection"
[intValue := wsdlClient executeSelector: #echoInteger args: #(123).
    strValue := wsdlClient executeSelector: #echoString args: #('ABc 46')]
        ensure: [wsdlClient close].
```

An alternative approach is to create the connection using the normal HTTP mechanisms, and send the SoapRequest. For example:

```
wsdlClient := WsdlClient new loadFrom:
    'http://services.pagedownweb.com/ZipCodes.asmx?WSDL' asURI.
http := HttpClient host: 'services.  pagedownweb.com'.
[http connect] on: Exception
    do: [:ex | ex inspect].
request := SoapRequest new.
request port: wsdlClient config anyPort.
request transportClient: http.
dict := Dictionary new.
dict at: #zip_IN put: '45069'.
request smalltalkEntity:
    (Message selector: #rtnZipInfoCSV
        arguments: (Array with: dict)).
http close.
request value.
```

SOAP Exception Handling

There are five exception classes specific to SOAP support: SoapFault, SoapClientFault, SoapServerFault, SoapException, and SoapEmptyBodyException:

```
Exception
   Error
      SoapException
         SoapEmptyBodyException
   SoapFault
      SoapClientFault
      SoapServerFault
```

These SOAP exceptions are not intended for use by user applications. When building your web services you should create and use exceptions that are specific to your application, creating new classes as necessary. As a example, in the WebServicesDemo, see WSLDSrvcGeneralPublicDoc holdingByAcquisitionNumberDoc:.

SoapException indicates a server or socket error. SoapBodyException is signaled specifically if, after unmarshaling a response, the body is empty.

SoapFault and its subclasses are returned as the body element of a response, indicating that the server found some problem with the request or its processing. SoapClientFault usually indicates that the request was malformed or lacked needed information to process the request. SoapServerFault indicates that processing failed for reasons not directly attributable to the request, such as an upstream server failing to respond. SoapFault is raised if a version mismatch (invalid namespace) or a “must understand” requirement was not obeyed by the processor.

SOAP and WSDL communications can suffer any number of other general communications errors as well, such as server performance and schema relocation errors.

Any time your application fetches a schema from a remote server, various communication errors may be raised (e.g. HTTP Not Found) that are not specifically covered by SOAP and WSDL support. These errors may be raised by the host OS or the VisualWorks protocols framework, and will be sent to your aplication as general exceptions.

6

Building Web Services

The VisualWorks web services framework may be used to build both client and server applications. Where the previous chapters focused on building client applications, this chapter is devoted to building servers. The wizards and class builders included in the VisualWorks framework can also help simplify the task of developing server applications.

The facilities described in the previous chapters are appropriate for a client that needs to make an occassional SOAP request. For a server application, though, it is inconvenient and inefficient to set up a new connection for each request. The same holds for both client and server when web services are used in a distributed computing environment. In both of these cases, it is better to use Opentalk.

The follow pages explore the Opentalk extensions to VisualWorks Web services support and their use in developing servers.

In brief, this chapter discusses:

- Web Services and Opentalk
- Building Servers from a WSDL schema
- Generating a Schema from Smalltalk Classes
- Generating a Schema using the WsdlBuilder
- Using the Opentalk Request Broker
- SOAP Messaging
- HTTP Transport Extensions

Web Services and Opentalk

Opentalk is a VisualWorks component that adds a rich and extensible framework for developing, deploying, maintaining, and monitoring distributed applications. For web services applications, Opentalk provides rich support for developing brokers to process multiple SOAP requests and replies, as is needed for servers and distributed environments.

By integrating web services with the Opentalk framework, VisualWorks simplifies the task of developing request brokers and provides other support services, such as the Opentalk naming service.

Loading Opentalk-SOAP

The Opentalk extensions to SOAP and XML protocol support are in the Opentalk-SOAP parcel. To load the support required for the discussion in this chapter:

1 Open the Parcel Manager (select **System > Parcel Manager** in the VisualWorks Launcher window).

2 Select the **Distributed Computing** category, in the list on the left-hand side of the tool.

3 In the upper-right list, select and load **Opentalk-SOAP** (**Parcel > Load** menu command).

4 Select and load **Opentalk-HTTP** or **Opentalk-CGI**, to provide the transport for SOAP messaging.

Parcel contents

Loading Opentalk-SOAP also loads Opentalk-Core, its prerequisites, and Opentalk-XML. The contents of Opentalk-Core are described in the Opentalk Communication Layer Developer's Guide.

Opentalk-XML contains two main extensions to the client XML support:

- XMLRequest and XMLReply classes handle general XML messages, and fit the messages in the Opentalk RemoteMessage framework.
- XMLMarshaler represent an XML binding in the Opentalk framework.

See XML Messaging below for more information.

Opentalk-SOAP further extends the XML extensions covering specific SOAP requirements for XML messaging:

- SOAPRequest and SOAPReply provide for special SOAP requirements, including message structure and SOAP error responses.
- SOAPMarshaler provides for marshaling and unmarshaling support.

See SOAP Messaging below for more information.

Opentalk-HTTP provides the HTTP transport infrastructure, as required by Opentalk. Two classes are of particular interest.

- HTTPClientTransport simply wraps Net.HttpClient to do the actual HTTP work. It is only useful for request clients.
- HTTPTransport implements both an HTTP client and an HTTP server. However, it is not currently a full-featured HTTP server, so might not be useful in some circumstances.

See HTTP Transport Extensions for more information.

Building Servers from a WSDL schema

In Web Services Wizard we described how to use the web services wizard and WsdlClassBuilder to create classes based on a WSDL document to access a web service as a client from Smalltalk. These tools provide additional features for creating a server, which we discuss here.

Before using the wizard, you must also load the WSDLWizard package (for instructions, see: Loading Support for Web Services).

Creating Service Classes using the Web Services Wizard

Given a WSDL schema, the wizard can create the Smalltalk code and service classes that are needed to build a web service in VisualWorks. The wizard allows you to select processing options, and then it generates the <schemaBindings> section and produces the supporting code.

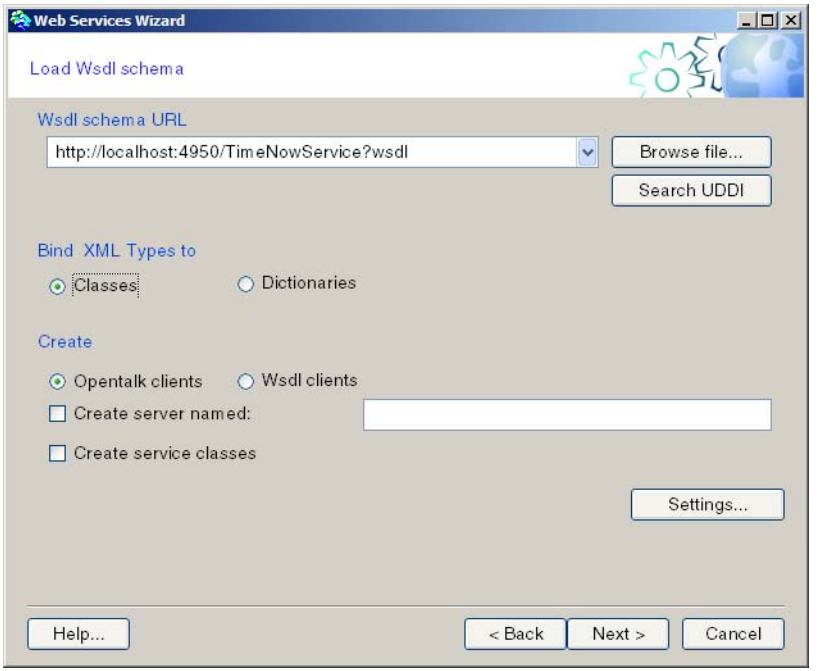
To illustrate this use of the wizard, we will use WebServicesTimeDemo (for details, see: Web Services Examples). If you haven't already done so, load the WebServicesTimeDemo parcel, but do not start any of the demonstration servers.

1 To prepare a WSDL schema, open a Browser, locate class TimeServer in the WebServicesTimeDemo parcel, and save the XML contents of the class-side method wsdlSchema in a file (e.g. TimeServer.wsdl). Only save the contained XML, not the Smalltalk code.

2 Launch the wizard by selecting **Web Services Wizard** from the **Tools** menu of the Visual Launcher.

3 On the first page of the wizard, select **Create an application from a WSDL schema**, and click **Next**.

4 On the next page, specify a schema to load:



Click on **Browse file…** and locate the file TimeServer.wsdl that we created in the first step. We'll use this schema to build the service.

In general, you can enter a URL in the **WSDL schema URL** field (including a file: URL).

5 In the **Bind XML Types to** section of the wizard, select Classes.

This option specifies how to handle complex data types (for details, refer to Generating XML-to-object Bindings):

- **Classes** creates a Smalltalk class for each complex type
- **Dictionaries** maps each complex type to a Dictionary

6 In the **Create** section of the wizard, select **Opentalk clients**.

7 Check the **Create Opentalk server named:** option, and provide a name for the server. For this example, enter TimeNowServer as the server name.

8 Check the **Create service classes** option.

This tells the wizard to generate the stub class and methods to provide the implementation for the service.

9 By default, the server, and binding, service, and server classes will all be created in the package named WSDefaultPackage, and they will belong to the Smalltalk.* name space.

To change the package and/or name space for the generated code, click **Settings…** and enter different names.

For this example, use the default package and name space.

10 Click **Next** to generate the Smalltalk support code.

11 A dialog appears to show the response classes that will be generated, with the option to change their names. Simply click **OK**.

12 Once the code is generated, the final wizard page is displayed. This page displays a workspace with Smalltalk expressions that exercise the newly-generated server code. However, before we can use the server code, we must add implementation code.

Using the schema for the TimeServer, the wizard has created the following classes in the WSDefaultPackage package: TimeNowService (the service class), TimeNowServer (an Opentalk server), and TimeNowServiceClient (an Opentalk client class).

The service class TimeNowService includes stub methods for the operations defined in the schema. These methods contain pragmas that define the operation, but no implementation in Smalltalk.

13 To add implementation code to the service class TimeNowService, the workspace includes code to open a browser on it. Evaluate the following using Do It:

```
RefactoringBrowser newOnClass: TimeNowService.
```

You may be prompted for the full name of the class. If so, choose: Smalltalk.TimeNowService. The class WebServices.TimeNowService (note the different name space) belongs to the WebServicesTimeDemo, and we do not want to change it.

14 In the browser on class Smalltalk.TimeNowService, select the protocol named public api, and the method named **timeNow**.

The stub method generated by the wizard now appears in the code pane of the browser:

```
timeNow
    <operationName: #TimeNow>
    <result: #TimeNowResponse>
    ^self  "Add implementation here"
"Result object:
TimeNowResponse new
    result: ('Time');
    yourself"
```

This method contains a suggestion about the implementation code.

15 To add the implementation code, edit the method as follows:

```
timeNow
    <operationName: #TimeNow>
    <result: #TimeNowResponse>
    ^TimeNowResponse new
        result: Time now;
        yourself
```

The effect of this code is to return a new instance of class TimeNowResponse that contains a Time object.

16 Select Accept from the <Operate> menu to compile the code.

You may now test this code in the wizard's workspace.

17 In the workspace, start the server by evaluating:

```
server := TimeNowServer new.
server start.
```

18 To test the server, evaluate the following using Inspect It:

```
client := TimeNowServiceClient new.
client start.
```

```
value := client timeNow.
```

The result in value should be an OrderedCollection that contains a single TimeNowResponse object. This is the result passed from the newly-created TimeServer running on your workstation.

19 Stop the server and client by evaluating this code (using Do It):

```
client stop.
server stop.
```

To save code from the Workspace, copy its contents and paste them into another workspace.

20 Click **Finish** to close the wizard.

Creating an Opentalk Server from a WSDL schema

An Opentalk server class includes all of the methods necessary to create Opentalk brokers, load bindings, start brokers, and shut down brokers.

To see the Opentalk server generation example, load the WebServicesDemo parcel and browse testCreateOpentalkClientServerClasses in TestCreateWSApplication.

The Opentalk server class is generated with port implementation information in a portDescription method, in class OpentalkServerSrvcGenPublicDoc:

```
portDescription
    "Generated by WS Tool on #(January 31, 2003 10:29:21 am)"
    <wsdlServiceImplementation: #'ServiceImplementationName'>
    <serviceClass: #'WebServices.WSLDSrvcGeneralPublicDoc' address:
        #'http://localhost:3931/generalDoc' bindingType: #'soap'>
```

The port description describes the server access point. The wsdlServiceImplementation name is the WSDL specification <service> element name.

For more information about Opentalk server support, refer to the Opentalk Communication Layer Developer's Guide.

Creating pragma templates

The Web Services Tool API can also create templates for pragmas and the Opentalk server class. For an example, see the LibraryServer class (jn the WebServicesDemo package).

To create pragma templates for interfaces, for example, use the following:

```
WsdlClassBuilder setInterfacePragmasForClasses:
    (Array with: WSLDSrvcGeneralPublic with:
        WSLDSrvcGeneralPublicRpc)
```

This creates a pragma for the methods located in “public api” category:

```
searchByExactTitle: aStruct
"Generated by WS Tool on #(February 2, 2003 7:39:01 pm)"
    <operationName: #'SearchByExactTitle'>
    <addInputParameter: #'searchByExactTitle' type: #String>
    <result: #String>
```

the default type String should be changed to the correct type.

To create pragma templates for the instance variables, evaluate:

```
WsdlClassBuilder setTypePragmasForClasses:
    (Array with: WSLDDataPersonName with: WSLDHoldingBook)
```

This creates “set” accessors and a pragma for the variable type, e.g.:

```
authorialType: aString
    "Generated by WS Tool on #(February 2, 2003 7:39:01 pm)"
    <addAttribute: #authorialType type: #String>
    authorialType := aString
```

To create an Opentalk server class template, evaluate:

```
WsdlClassBuilder new
        serverClassName: 'OpentalkServerSrvcGenPublicRpcx';
        namespace: 'WebServices';
        serviceClasses: (Array with: WSLDSrvcGeneralPublicRpc);
        package: 'WebServicesDemo';
        createServerClass.
```

Creating Service Classes using the WsdlClassBuilder

The service classes are generated by WsdlClassBuilder from the WSDL specification operations. The WSDL specification binding names are used for the service class names. The service class methods that are derived from the operations go into the “public api” protocol, and contain pragmas that describe operation parameters. For example, the Library Demo defines services such as HoldingByAcquisitionNumber and ProvidesServices.

The HoldingByAcquisitionNumber service has an input parameter that is a positive integer, and the ProvidesServices service has no input parameters.

```
<message name="HoldingByAcquisitionNumber">
   <part name="holding_acquisitionNumber" type="xsd:positiveInteger"/>
</message>
<portType name="SrvcGeneralPublicAllTypesPortType">
   <operation name="ProvidesServices">
       <documentation>The ProvidesServices operation checks if the
          library has some services. Returns true or false </
documentation>
      <input message="tns:ProvidesServices"/>
      <output message="tns:ProvidesServicesResponse"/>
   </operation>
   <operation name="HoldingByAcquisitionNumber">
      <documentation parameterOrder="holding_acquisitionNumber">The
         HoldingByAcquisitionNumber operation returns a holding or
         exception if the holding not found</documentation>
      <input message="tns:HoldingByAcquisitionNumber"/>
      <output message="tns:HoldingByAcquisitionNumberResponse"/>
      <fault message="tns:HoldingByAcquisitionNumberFault"/>
   </operation>
….
</portType>
```

Based on this description and the specification style, the WsdlClassBuilder generates the WSService service class with the following methods.

RPC style, in WSLDSrvcGeneralPublicRpc class:

holdingByAcquisitionNumber: aLDHolding_acquisitionNumber

```
"Generated by WS Tool on #(February 19, 2003 9:36:44 am)"
   <operationName: #HoldingByAcquisitionNumber>
   <documentation: #'The HoldingByAcquisitionNumber operation
      returns a holding or exception if the holding not found'>
   <addParameter: #'holdingByAcquisitionNumberDoc' type:
      #'LargePositiveInteger'>
   <result: #'WSLDHoldingBook'>
   <addException: #holdingNotFound type: #'LDExcHoldingNotFound'>
```

providesServices

```
"Generated by WS Tool on #(January 24, 2003 10:33:21 am)"
   <operationName: #'ProvidesServices'>
   <documentation: #'The ProvidesServices operation checks if the
```

```
        library has some services. Returns true or false '>
    <result: #'Boolean'>
```

Document style, in WSLDSrvcGeneralPublicDoc class:

```
holdingByAcquisitionNumberDoc: aStruct
"Generated by WS Tool on #(February 19, 2003 9:36:44 am)"
    <operationName: #HoldingByAcquisitionNumber>
    <documentation: #'The HoldingByAcquisitionNumber operation
        returns a holding or exception if the holding not found'>
    <addParameter: #'holdingByAcquisitionNumberDoc' type:
        #'LargePositiveInteger'>
    <result: #'WSLDHoldingBook'>
    <addException: #holdingNotFound type: #'LDExcHoldingNotFound'>


providesServices"Generated by WS Tool on #(January 24, 2003 10:33:21
am)"
    <operationName: #'ProvidesServices'>
    <documentation: #'The ProvidesServices operation checks if the
        library has some services. Returns true or false '>
    <result: #'Boolean'>
```

If style='document' and use='literal' are specified at the SOAP binding level, a description must have zero or one part in a wsdl:message element. For this reason, the document-style parameters are always placed in a Struct.

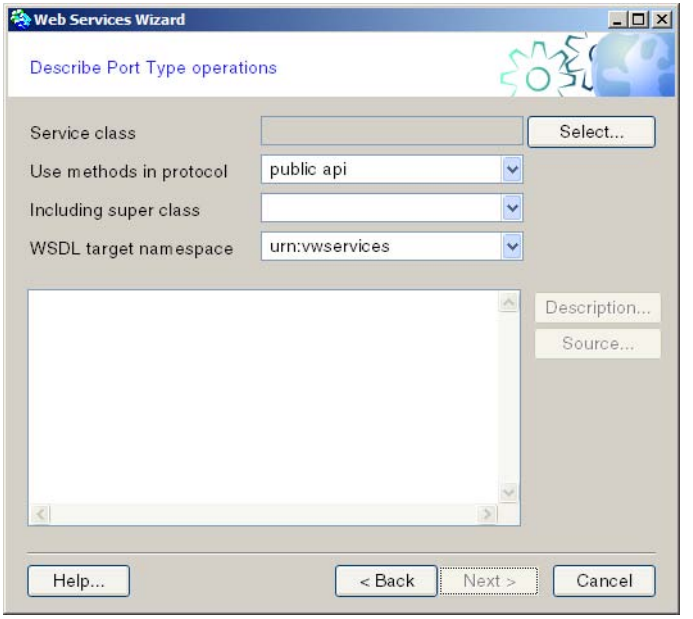
Generating a Schema from Smalltalk Classes

Generating a Schema using the Web Services Wizard

The web services wizard also provides a simple way to create a WSDL schema document from Smalltalk code that provides a service.

1 To launch the wizard, select **Tools > Web Services Wizard** in the Visual Launcher.

2 On the first page select **Expose an application as a web service**, and click **Next**.



3 The **Describe Port Type operations** page is used to identify the application and methods to represent in the WSDL schema.

Fill in the following fields:

- **Service class** is the application class providing the service. Click the **Select...** button to browse the image for the desired class.
- **Use methods in protocol** selects the method category, or protocol, that contains the methods to expose in the schema. All methods must be in the same protocol, which is **public api** by default.
- **Including super class** specifies the superclass up to which method and type specifications are included. The WSDL Builder settings in the Settings Tool must also be set to **Add the service super class methods** for this option to have effect. (for details, see Web Services Settings).
- **WSDL target namespace** specifies the targetNamespace attribute in a WSDL schema (in the <schema> section).

4 Next, set the descriptions for methods.

Once the class (and superclass option) has been selected, the methods that will be defined in the WSDL schema are listed. Any methods with **Caution!** symbols need additional information in their descriptions. For each of these methods, select the method and click **Description…**

In the **Operation Description** dialog, provide a **Name**, **Type**, and **In/Out** value for each operation parameter and the return value. Your selections for types will depend on the requirements of the operation. Refer to Creating XML-to-Object bindings for help selecting types.

When you are finished, **Accept** the settings.

5 When all method descriptions are complete, click **Next**.

6 On the **Add Header Processors to Operations** page, you may optionally include header processors for each operation in the service.

A header processor serves two purposes:

- It maps between a Smalltalk type and SOAP header node
- The procesor methods are invoked to process header entries for the operation by the Opentalk server or client classes.

7 When finished, click **Next**.

8 On the **Describe complex types** page, set the descriptions for complex types.

If the method descriptions involve complex types, they are listed on this page. Any incomplete descriptions are marked with a **Caution!** symbol. Complete the descriptions, as in one of the previous steps.

When all complex type descriptions are complete, click **Next**.

9 If you want to generate a Opentalk server for the service being described, check the **Generate class** option on the **Generate Opentalk-SOAP server class** page; otherwise leave it unchecked.

If you check the option, provide a class name and other parameters as needed.

When ready, click **Next**.

10 If you want to generate an Opentalk client class for the service being described, check the **Generate class** option on the **Generate Opentalk client class** page; otherwise leave it unchecked.

If you check the option, provide a class name and other parameters as needed.

When ready, click **Next**

11 The **Testing Opentalk server** page displays a Workspace script you can use to exercise the broker and client classes generated by the wizard. You should review these results.

To correct problems and regenerate, click **Back**. To proceed, click **Next**.

12 To generate a WSDL document for this service, check **Generate** on the **Generating WSDL schemas** page

13 In the **Schemas** section, select the schema type:

- **WSDL schema with XML to object binding** produces a schema including the <schemaBindings> section.
- **WSDL schema** produces only the schema, without the <schemaBindings> section
- **XML to object binding** produces only the <schemaBindings> section
- **Service interface** produces the abstract description of web services operations that includes messages and port types.

14 In the **Destination** section, select the output destination for the schema:

- **Method** writes the schema to a class method. Provide a method name (wsdlSchema by default) and select the class.
- **File out** writes the schema to an external file in plain text format.
- **POST url** posts the schema on the HTTP server using the specified URL.

15 Click **Next** to generate the schema.

16 Click **Finish** to close the wizard.

Generating a Schema using the WsdlBuilder

WsdlBuilder generats a basic WSDL schema from the information found in either a service class that provides a web service interface description or an Opentalk server class.

There are a few steps required before we can generate the WSDL:

1 Provide descriptions for service interfaces.

2 Provide descriptions for interface parameters, and for result and exception types.

3 Optionally, provide descriptions for access points or service implementations

Providing a description for service interfaces

The service provider class must have descriptions for all methods that are to be represented as external interfaces. The methods should include pragmas that describe the operation name, input/output parameters, return types, exceptions, and documentation.

Pragmas can be added to methods manually, or WsdlClassBuilder can create pragma templates for service class methods in the “public api” category. The method creates or updates all methods in “public api” category and opens a browser on all updated methods.

The pragmas can be added to the service class and its super classes.

For an interface description the following pragmas are supported:

#operationName:

The operation name to be used for the <operation> element in the WSDL <portType> element.

#documentation:

The operation description.

#addParameter: *parameterDescription* **type:** *typeDescription*

parameterDescription is either:

- #parameterName
- #(#parameterName #parameterType)

where #parameterName is the parameter name, as a Symbol, and the #parameterType is either #in or #out.

typeDescription is either:

- #type - describes simple or complex type, e.g., #String or #LDHolding.

- #(#Collection #type #min #max) - describes a collection of type #type. #min and #max specify the cardinality of #Collection.
- #(#Array #type #dimension) - describes a Soap array type. #dimension specifies array dimension.
- #(#Choice #(#name #type) …#(#name #type)) - describes choice type. The choice type will be represented as a Struct object, where the key and value of a choice type are #(#name #type).

#addException: *exceptionName* **type:** *typeDescription*

Operation exception name and type. Each operation can have zero or more exceptions. Corresponds to the WSDL <fault> elements for the operation.

#result: *typeDescription*

Result type. Corresponds the WSDL operation response message.

Note: If data types are defined in your own namespace or in Smalltalk, the types can be described by their class names alone. Types defined outside of your namespace or Smalltalk, in order to be resolved, must be fully qualified as MyNamespace.HoldingBook.

For example:

```
holdingByAcquisitionNumber: aLDHoldingAcquisitionNumber
"Generated by WS Tool on #(January 24, 2003 10:33:21 am)"
   <operationName: #'HoldingByAcquisitionNumber'>
   <documentation: #'The HoldingByAcquisitionNumber operation
      returns a holding or exception if the holding not found'>
   <addParameter: #'holdingByAcquisitionNumber'
      type: #'LargePositiveInteger'>
   <result: #'WSLDHoldingBook'>
   <addException: #'holdingNotFound' type: #'LDExcHoldingNotFound'>
```

The above method will be represented in the WSDL <portType> and <message> elements:

```
<message name="HoldingByAcquisitionNumber">
   <part name="holdingByAcquisitionNumber"
      type="xsd:positiveInteger"/>
</message>
<message name="HoldingByAcquisitionNumberResponse">
```

```
        <part name="return" type="ns:WSLDHoldingBook"/>
    </message>
    <message name="HoldingByAcquisitionNumberFault">
        <part name="holdingNotFound" type="ns:LDExcHoldingNotFound"/>
    </message>
<portType name="WSLDSrvcGeneralPublicRpc">
    <operation name="HoldingByAcquisitionNumber">
    <documentation>The HoldingByAcquisitionNumber operation returns
a
        holding or exception if the holding not found</documentation>
        <input message="ns:HoldingByAcquisitionNumber"/>
        <output message="ns:HoldingByAcquisitionNumberResponse"/>
        <fault message="ns:HoldingByAcquisitionNumberFault"/>
    </operation>….
</portType>
```

Providing a description for interface parameters, result, and exception types

All data type definitions used to describe the parameters in the messages exchanged have to be defines in the WSDL <types> element in the form of an XML Schema.

The current implementation supports complex, simple, and collection types.

- Complex types define a set of attribute declarations. For example, the complex type named WSLDHoldingBook contains descriptions of holding book attributes as authors, coverPhoto, language, largePrint, pages, and so on. In the WSDL <schema> element, this type will be mapped to the <complexType> element.
- Simple types have no child elements. For an example, browse the LargePositiveInteger type. Simple types are encoded using XML Schema: Datatypes
- Collection types define collection of complex or simple types.

Each complex object must have attribute type descriptions for the attributes that are to participate in the message exchange. The attribute description should be provided in the set accessor pragma.

If a complex object has references to other complex objects that are going to participate in the message exchange, these complex objects must have attribute descriptions.

If a superclass of the complex object has an attribute description, the XML schema will include the <complexType> elements for the complex object and its superclass.

The following pragmas are supported to describe the attributes:

#addAttribute: *attributeDescription* **type:** *typeDescription*

attributeDescription is either:

- #attributeName - element name in complex type
- #(#attributeName #options) - #options is either #optional or #required. These values specify if this XML element will be optional or required. The XML attribute minOccurs will be 0. #required is the default option

typeDescription - as described above for #addParameter:type:

To create a pragma template for complex objects, you can send setTypePragmasForClasses: to WsdlClassBuilder. This message creates or updates setter accessors with pragmas for all instance variables, and opens a selector browser on all updated methods.

For example

```
WsdlClassBuilder setTypePragmasForClasses:
   (OrderedCollection
      with: WSLDHoldingBook
      with: WSLDDataCatalogNumber
      with: WSLDDataAuthorialName)
```

will generate these pragma templates in WSDLHoldingBook:

acquisitionDate: aDate

```
"Generated by WS Tool on #(January 3, 2003 11:41:36 am)"
   <addAttribute: #acquisitionDate  type: #String>
   acquisitionDate := aDate
```

authors: aCollOfWSLDDataAuthorialName

```
"Generated by WS Tool on #(January 3, 2003 11:41:36 am)"
   <addAttribute: #authors  type: #(#Collection #String)>
   authors := aCollOfWSLDDataAuthorialName
```

The default String type should be replaced by the correct attribute type:

```
acquisitionDate: aDate
"Generated by WS Tool on #(January 3, 2003 11:41:36 am)"
   <addAttribute: #acquisitionDate  type: #Date>
   acquisitionDate := aDate

authors: aCollOfWSLDDataAuthorialName
"Generated by WS Tool on #(January 3, 2003 11:41:36 am)"
   <addAttribute: #(#authors #optional)
      type: #(#Collection #WSLDDataAuthorialName)>
   authors := aCollOfWSLDDataAuthorialName
```

Also, in WSLDDataAuthorialName:

```
name: aWSLDDataPersonName
"Generated by WS Tool on #(January 3, 2003 11:41:35 am)"
   <addAttribute: #name  type: #'WSLDDataPersonName'>
   name := aWSLDDataPersonName
```

Providing descriptions for service access points

There are two options to provide access points for a service: either by specifying the service URL in the Opentalk server class, or by using the WsdlBuilder API. WsdlClassBuilder can generate the Opentalk server class with the service description template. The service URL should be changed to correct address.

The following pragmas are supported to describe server ports:

wsdlServiceImplementation

Specifies the WSDL <service> element name.

serviceClass: *aServiceSymbol* **address:** *accessPointSymbol*
bindingType: *bindingSymbol*

Describes the service class, its access point, and binding type. The service class name must be fully qualified (e.g., WebServices.WSLDSrvcGeneralPublicDoc), unless it is defined in the Smalltalk.* name space.

For example:

```
OpentalkServerSrvcGenPublicDoc class >> portDescription
"Generated by WS Tool on #(January 31, 2003 10:29:21 am)"
   <wsdlServiceImplementation: #'ServiceImplementationName'>
   <serviceClass: #'WebServices.WSLDSrvcGeneralPublicDoc'
      address: #'http://localhost:3931/generalDoc' bindingType: #'soap'>
```

Generating the specification

After performing the above preparatory steps, the WSDL specification can be generated.

Browse WSLD1TestCreateWSDL, in the WebServicesDemo parcel, for examples.

WsdlBuilder instance creation API

buildFromOpentalkServer: *aServerClassName*
classNamespace: *clString* **targetNamespace:** *tnsString*

(Both class and instance versions.) Builds all elements for the WSDL specification: types, portType, binding, service and XMLToSmalltalk binding. All data types will be resolved in the specified class name space. tnsNamespace contains the target namespace for the WSDL specification and XML types. The default message style is “document/literal.” The server port description provides information about service class and access point.

buildFromOpentalkServer: *aServerClass*

The same as above with default values for class name space and target name space. The default class name space is 'Smalltalk.*' and the default target namespace is 'urn:vwservices'.

buildFromService: *aServiceClass* **classNamespace:** *clString*
targetNamespace: *tnsString*

Same as buildFromService:.

buildFromService: *aServiceClass*

(Both class and instance versions.) From service class interface description builds the WSDL elements types, portType, binding and XML-to-Smalltalk binding.

Instance methods

useDocument

Sets properties to build 'document/literal' style

useRPC

Sets properties to build 'rpc/literal' style

useRPCEncoded

Sets properties to build 'rpc/encoded' style

setPortAddress: *url* **forBindingNamed:** *serviceName*
wsdlServiceNamed: *wsdlService*

Set an access point (url) for the service class (serviceName). The value of wsdlService provides name for the WSDL <service> element. The method corresponds to the service description from Opentalk server class.

Creating WSDL specification elements

createSpec

Creates a full WSDL specification with elements types, portType, binding and service. The service element created only if information about access point was provided.

createServiceInterface

Creates types, portType and binding elements.

createServiceImplementation: *locationString*

Creates an import element with interface location specified by URL (locationString) and service element.

createSmalltalkBinding

Creates a XMLToSmalltalkBinding element

createSpecWithSmalltalkBinding

Full WSDL specification and XMLToSmalltalkBinding element

Printing WSDL specification

printSpecOn: *aStream* **printServiceInterfaceOn:** *aStream*
printServiceImplementationOn: *aStream* **interfaceLocation:** *aString*
printSmalltalkBindingOn: *aStream*
printSpecWithSmalltalkBindingOn: *aStream*

Output the spec on the specified stream.

Examples

To build a WSDL 'rpc/literal' schema from an Opentalk server class:

```
builder := WsdlBuilder new.
builder
```

```
    useRPC;
    buildFromOpentalkServer: OpentalkServerSrvcGenPublicRpc
    className space: 'WebServices'
    targetNamespace: 'urn:LibraryDemo\srvcGeneral\rpc'
```

To print WSDL with a XMLToSmalltalkBinding element:

```
stream := (String new: 2048) writeStream.
builder printSpecWithSmalltalkBindingOn: stream.
```

To build a WSDL 'document/literal' schema from a service class:

```
builder := WsdlBuilder
    buildFromService: WSLDSrvcGeneralPublicDoc
    className space: 'WebServices'
    targetNamespace: 'urn:LibraryDemo\srvcGeneral\srvcdoc'.
```

Printing types, portType and binding elements:

```
stream := (String new: 2048) writeStream.
builder printSpec: stream.
```

Adding an access point for the service:

```
builder
    setPortAddress: 'http://localhost/WSSrvcGeneralPublic/
srvcGeneralDoc'
    forBindingNamed: 'WSLDSrvcGeneralPublicDoc'
    wsdlServiceNamed: 'LibraryDemoSoapDoc'.
```

Printing a service implementation:

```
builder
    printServiceImplementationOn: stream
    interfaceLocation: 'http://localhost/schemaDoc.wsdl'
```

Using the Opentalk Request Broker

By integrating the SOAP support infrastructure with Opentalk, your server and client applications gain access to the Opentalk request broker and its services.

In the Opentalk framework, your application uses a request broker to access all distribution services and their corresponding APIs. The application establishes an instance of the distribution machinery by creating an instance of the request broker.

In the preceding discussions of WSDL and SOAP clients, a broker was not needed. When using higher-level APIs, the services performed by a broker are established implicitly as part of the request.

However, for developing a web services server, or using web services as a distributed computing architecture, you need a full request broker.

The following discussion describes the APIs used to create, configure, and manipulate these broker objects.

Creating and Configuring a Broker

A request broker maintains the necessary machinery to manage access to its services. This requires an access point, an object adaptor to represent the protocol implementation, and a registry of objects that represent its services.

Several instance creation methods, defined in BasicRequestBroker, provide common broker configurations:

newSoapCgiAt: *anIPSocketAddress* **binding:** *aWsdlBinding*
newSoapCgiAt: *anIPSocketAddress* **bindingNamed:** *aString*
newSoapCgiAtPort: *aNumber* **binding:** *aWsdlBinding*
newSoapCgiAtPort: *aNumber* **bindingNamed:** *aString*
newSoapHttpAt: *anIPSocketAddress* **binding:** *aWsdlBinding*
newSoapHttpAt: *anIPSocketAddress* **bindingNamed:** *aString*
newSoapHttpAtPort: *aNumber* **binding:** *aWsdlBinding*
newSoapHttpAtPort: *aNumber* **bindingNamed:** *aString*

Create a new binding as specified.

Select the instance creation method according to the transport type (CGI or HTTP), how the access point is specified (port number or IPSocketAddress instance), and how the WSDL binding is specified (a WsdlBinding instance or a String).

For example, to create a broker using the HTTP transport on port 4242 and the TimeServerSoapBinding (see SOAP Messaging below or the Opentalk-SOAP package comment for the binding definition), use:

```
server := BasicRequestBroker
        newSoapHttpAtPort: 4242
        bindingNamed: 'TimeServerSoapBinding'
```

The access point (port number or IP socket address) is the specific location in the network where the broker receives remote messages. Access points link the brokers with the underlying network protocol, and so are expressed in terms recognized by the protocol. For example, the access points of TCP/IP based brokers are instances of IPSocketAddress.

The broker creation methods all invoke the more basic Opentalk broker creation API. For a full discussion of the Opentalk broker framework, refer to the Opentalk Communication Layer Developer's e.

Additional configuration of the broker can be performed by sending these messages to the broker:

requestType: *aClass*

Sets the request class to be used by the broker.

requestTimeout: *anInteger*

Sets the request timeout as *anInteger* milliseconds

Starting and Stopping a Broker

An instance of a broker has to be activated before it can mediate remote communication. To activate it to a "running" state, use the start message. Similarly, stop it by sending a stop message. Stopping a broker closes all the open communication channels and deactivates it. Once it has been stopped, a broker can be restarted again with start.

```
| server |
server := (BrokerConfiguration standard
    adaptor: (AdaptorConfiguration connectionOriented
        transport: (TransportConfiguration http
            marshaler: (MarshalerConfiguration soap
                bindingNamed: 'TimeServerSoapBinding')))
    ) newAtPort: 4242.
"  server start.
```

SOAP Messaging

The Opentalk-SOAP parcel integrates generic SOAP/WSDL support with Opentalk-XML (see XML Messaging below), to provide transparent messaging access to SOAP services for clients, and infrastructure for setting up SOAP servers. SOAP messages are sent via lower-level transport protocols, usually via HTTP, which has to be loaded as well in order to set up a SOAP broker (see HTTP Transport Extensions below).

The SOAP marshaler configuration method is soap, with a required parameter binding: which takes an instance of WsdlBinding. Another form of the required parameter is bindingNamed: that takes a name of a pre-loaded WSDL binding.

Let's build a simple time service as an example. It will support a single operation “Now” that will return the current time. For the service implementation we will simply use the Timestamp class, and the operation “Now” will invoke the now method. Consider the following WSDL binding definition:

```
schema := '<definitions
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
   targetNamespace="urn:time-server">
   <message name="timeNow">
   </message>
   <message name="timeNowResult">
      <part name="time" type="xsd:dateTime" />
   </message>
   <portType name="TimeServerPortType">
      <operation name="TimeNow">
         <input message="timeNow" />”
         <output message="timeNowResult" />
      </operation>
   </portType>
   <binding name="TimeServerSoapBinding" type="TimeServerPortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http" />
      <operation name="TimeNow" selector="now">
          <soap:operation
            soapAction="http://localhost/TimeServer.TimeNow" />
         <input>
            <soap:body use="encoded"
               encodingStyle=
                  "http://schemas.xmlsoap.org/soap/encoding/" />
         </input>
         <output>
            <soap:body use="encoded"
               encodingStyle=
                  "http://schemas.xmlsoap.org/soap/encoding/" />
         </output>
      </operation>
   </binding>
</definitions>'.
```

The binding instance can be built from the schema as follows:

```
(WsdlBinding loadWsdlBindingFrom: schema readStream)
   bindings first.
```

Once the binding is loaded we can create an instance of a broker for SOAP over HTTP:

```
client := (BrokerConfiguration standard
   adaptor: (AdaptorConfiguration connectionOriented
      transport: (TransportConfiguration http
         marshaler: (MarshalerConfiguration soap
            bindingNamed: 'TimeServerSoapBinding')))
) newAtPort: 4243.
client start.
```

We will need to create another SOAP broker for the server, and let's run it on port 4242. To complete the server setup we just need to register the server object implementing the service with the server broker. Registration is performed by simply exporting the server object with a preselected OID:

```
service := Timestamp.
server objectAdaptor export: service oid: 'timeserver'.
```

This simple setup allows us to send SOAP requests. SOAP requests are sent to instances of a URL instead of an ObjRef. The URL should point to the address that the server broker is listening on and include the OID as an additional path element. In our example we can run both brokers in the same image in which case the URL would be:

```
url := 'http://localhost:4242/timeserver' asURI.
```

A requests can be sent in an RPC style or in a fully transparent way. An RPC-style request invocation would look as follows:

```
reply := client sendRequest:
         (SOAPRequest
            newRequest: (Message selector: #now)
            to: url
            timeout: 20000).
```

To be able to send a message transparently we need to have a proxy for the remote service (of course the proxy can be reused for multiple message sends):

```
proxy := RemoteObject newOn: url requestBroker: client.
reply := proxy now.
```

Don't forget that when you're done using a broker it should be stopped.

```
client stop.
server stop.
```

Mapping SOAP operations to Smalltalk messages

In the WSDL binding specification above, there's one element attribute, “selector,” in the <operation> element in the <binding> section, that is not specified in the WSDL specification. This attribute is part of the Opentalk extensions, and is very important, because it is the only thing that links the WSDL operation “TimeNow” to the Smalltalk message now. Without it the framework can only make some simple guesses about which Smalltalk message to invoke when a SOAP request arrives. It is also important for clients, because it instructs the marshaler to marshal given Smalltalk message as the corresponding WSDL operation in the outgoing SOAP request. If the selector is not defined in the WSDL operation binding, then,

1 the operation name is transformed using the algorithm in SoapOperationBinding equivalentSmalltalkMessageName,

2 the target objects is checked as to whether it responds to the computed selector, and

3 if it does, that message is sent to the target object.

If the target object does not respond to either selector, it will be sent message soapPerform: with the SOAPRequest instance as the argument.

It is usually desirable to provide explicit mapping between WSDL operations and Smalltalk messages. However WsdlBindings are often built automatically from publicly available binding specifications, and it specification writers can not generally be expected to add valid smalltalk selector equivalents of operation names to it.

You may simplify this work by using the WsdlClassBuilder to add mappings to Smalltalk classes and methods to a schema, in the <schemaBindings> section. Refer to Creating Service Classes using the WsdlClassBuilder in this chapter for more information.

If simply editing the specification and filling in the “selector” attribute is not feasible, for example because the specification is still under development and changes frequently, it is possible to modify the instance of WsdlBinding directly aftet it is built from the specification

XML. For example, if the “TimeNow” operation did not have the selector attribute specified, we could use following code fragment to compensate for that:

```
(binding operations
   detect: [:op | op name asString = 'TimeNow']) selector: #now.
```

User-defined SOAP types

Only simple data types are passed along with the messages in our simple example. To handle more complex objects, the marshaling framework needs an XMLToSmalltalk binding, the same as the one used for generic XML brokers. The binding definition can be included directly in the WSDL specification as in the next example:

```
schema := '<definitions
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
   targetNamespace="urn:coordinate-converter">
   <schemaBindings >
   <xmlToSmalltalkBinding name="CoordinateConverterBinding"
      xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
      targetNamespace="urn:coordinate-converter">
      <object name="Point" smalltalkClass="Point">
         <element name="x" ref="xsd:double"/>
         <element name="y" ref="xsd:double"/>
      </object>
   </xmlToSmalltalkBinding>
   </schemaBindings >
   <message name="ConvertPolarRequest">
      <part name="radius" type="xsd:double" />
      <part name="theta" type="xsd:double" />
   </message>
   <message name="ConvertPolarResponse">
      <part name="point" type="Point" />
   </message>
   <portType name="CoordinateConverterPortType">
      <operation name="ConvertPolar">
         <input message="ConvertPolarRequest" />
         <output message="ConvertPolarResponse" />
      </operation>
   </portType>
   <binding name="CoordinateConverterBinding"
      type="CoordinateConverterPortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http" />
```

```
        <operation name="ConvertPolar" selector="r:theta:">
            <soap:operation
                soapAction=
                    "http://localhost/CoordinateConverter.ConvertPolar" />
            <input>
                <soap:body use="encoded"
                    encodingStyle=
                        "http://schemas.xmlsoap.org/soap/encoding/" />
            </input>
            <output>
                <soap:body use="encoded"
                    encodingStyle=
                        "http://schemas.xmlsoap.org/soap/encoding/" />
            </output>
        </operation>
    </binding>
</definitions>'.
```

This is a schema that allows us to set up a server supporting a ConvertPolar operation that takes polar coordinates and returns an instance of Point with (x,y) coordinates. The operation binding plugs into the Point class method r:theta:, so one can simply export the Point class itself as the service implementation. The client side can directly use the #r:theta: message to invoke the operation.

The standard way is to define custom data types using the WSDL <types> section. So instead of the <schemaBindings> section above there would be something like the following:

```
'...
<types>
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
    targetNamespace="urn:coordinate-converter">
    <complexType name="Point">
        <element name="x" type="double" />
        <element name="y" type="double" />
    </complexType>
</schema>
</types>
...'
```

This has to be converted into an <xmlToSmalltalkBinding> section for the marshaling framework. It can be done by hand, or with the help of XMLTypesParser, which is able to generate an <xmlToSmalltalkBinding> out of the WSDL <types> section.

Class WsdlClient illustrates how to use the XMLTypesParser for "on-the-fly" parsing of WSDL specifications downloaded from the Internet. Currently the easiest way to invoke this parser is through the WsdlClient, e.g.:

```
binding := (WsdlClient readFrom: schema readStream)
    config bindings first.
```

Note that the <xmlToSmalltalkBinding> generated by the parser has a limitation. Since the <types> section does not specify the Smalltalk classes to which the individual types correspond, it generates all the type marshalers as struct marshalers. This means that the marshalers don't expect to handle general objects but instances of Struct instead. This is actually useful for dynamic clients like WsdlClient, for which one cannot expect to have the domain classes to be readily implemented in the client image. Marshaling the datatypes as Struct objects avoids this obstacle.

Exceptions and SOAP Faults

In general, any exceptions sent to the client are sent as SoapFaults. SoapFault instances can be generated by Opentalk or built by the application code. Transparent handling of operation faults (faults listed in the operation declaration in the WSDL specification) is also supported. Operation faults are marshaled into the <detail> element of SoapFault. On the client side Opentalk checks if the incoming SoapFault has an operation fault in its <detail> element. If there is one, it signals the <detail> exception, otherwise it signals the generic SoapFault exception. Operation faults are expected to be implemented as subclasses of Error.

The application code generates an operation fault by signaling the corresponding Error. Application code can also generate a SoapFault explicitly by signaling a SoapFault exception. Client code is expected to use the usual exception handling constructs, with the constraint that faults are inherently not resumable.

XML Messaging

The Opentalk-XML parcel enhances the XML-Object marshaling framework (XMLObjectMarshalers parcel) to integrate it with Opentalk. It also adds a new XML marshaler and XML messages, providing a general support implementation for XML messaging. As an abstraction, this provides for future implementation of XML-based

protocols, in addition to SOAP, simply by providing a binding. The SOAP implementation is, in effect, simply an extension of the XML protocol framework.

Marshaling

As before, the protocol is defined by an XMLObjectBinding, as described in XML to Smalltalk Mapping. All entities carried by the messages, i.e., objects, exceptions, and message formats, have to be defined in a binding. Also, individual marshaler methods are defined for the various XML elements.

Opentalk-XML adds a new marshaler, XMLMarshaler, as a centralized marshaler object that invokes all the individual marshaler methods in the SOAP parcel. The marshaler is initialized with an instance of XMLObjectBinding, which is wrapped in an instance of XMLOperationMap that maps Smalltalk selectors to corresponding request/reply entity types.

The XML marshaler configuration method is xml. An XML marshaler configuration has a required parameter map: which takes an instance of the XMLOperationMap as the parameter. An instance of XMLOperationMap is created on an instance of binding using the binding: instance creation method. The selector mapping has to be built using add:request:reply: which takes the selector, request struct tag name and reply struct tag name as the parameters.

The XML protocol has to be used in combination with a lower level transport protocol, like HTTP, which must be loaded in order to set up an XML broker.

Let's build a simple random number generator service as an example. It will support a single request NextBatch that will return a sequence of random numbers. For the service implementation we will simply use an instance of Random, and the NextBatch request will invoke the #next: method on it. The corresponding binding definition would look as follows:

```
schema := '<schemaBindings >
<xmlToSmalltalkBinding name="RandomGenerator"
   xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
   targetNamespace="urn:random">
<sequence_of name="numbers">
   <implicit ref="xsd:float"/>
</sequence_of>
<struct name="NextBatch">
<element name="number" ref="xsd:integer" />
```

```
</struct>
   </xmlToSmalltalkBinding>
</schemaBindings >'.
```

The operation map has to associate the 'NextBatch' request and the 'numbers' reply with the Smalltalk message next:

```
binding := XMLObjectBinding loadFrom: schema readStream.
map := (Opentalk.XMLOperationMap binding)
          add: #next:
          request: 'NextBatch'
          reply: 'numbers';
          yourself.
```

With the operation map, we can create an instance of a broker for XML over HTTP as follows:

```
client := (BrokerConfiguration standard
   adaptor: (AdaptorConfiguration connectionOriented
      transport: (TransportConfiguration http
         marshaler: (MarshalerConfiguration xml
            map: map)))) newAtPort: 4243.
client start.
```

Of course, we'll need to setup a server to be able to run the example. Let's assume we have another XML broker called server running on port 4242. To complete the server setup we just need to register the server object implementing the service with the server broker. Registration is performed by simply exporting the server object with a preselected OID:

```
service := Random new.
server objectAdaptor export: service oid: 'random'.
```

Now we are ready to send requests. XML requests are sent to instances of a URL instead of an ObjRef. The URL should point to the address that the server broker is listening on and include the OID as an additional path element. In our case, it could be:

```
url := 'http://localhost:4242/random' asURI.
```

A request can be sent in an RPC style or in a fully transparent way. An RPC-style request invocation would look as follows:

```
reply := client sendRequest: (
   XMLRequest
      newRequest: (Message selector: #next: argument: 10)
      to: url
      timeout: 20000).
```

To be able to send a message transparently we need to have a proxy for the remote service (of course the proxy can be reused for multiple message sends):

```
random := RemoteObject newOn: url requestBroker: client.
reply := random next: 10.
```

Don't forget that when you're done using a broker it should be stopped.

```
client stop.
server stop.
```

HTTP Transport Extensions

HTTP is the only transport currently supported for web services in Opentalk. This is provided by HTTPTransport, which does standard HTTP request handling, and CGITransport, which handles requests passed via a CGI relay.

HTTPTransport

The Opentalk-HTTP parcel contains the HTTP transport infrastructure required to operate with the request broker. The HTTP transport can be used with either the XML marshaler or the SOAP marshaler.

Two versions of HTTP transport are implemented.

- HTTPClientTransport simply wraps Net.HttpClient to do the actual HTTP work, and is, therefore, only usable for request clients.
- HTTPTransport implements both an HTTP client and an HTTP server, but is not currently a full-featured HTTP server

HTTPTransport may be missing some features that are important in your application. For example, there is currently no built-in support for firewall proxies, and so will not be useful in cases where you have to use a proxy to get to an external SOAP server. HTTPClientTransport, on the other hand, does support proxies, and so is useful in this case. Also, for setting up a server, consider using the CGITransport from the Opentalk-CGI package.

HTTPTransport assumes that application messages can carry contextual information (both XML and SOAP messages do). The transport compiles an "environment" dictionary from all the header

fields of the transport message and attempts to install it into the message during unmarshaling. This allows the application to access the header fields if necessary.

The configuration message for HTTPClientTransport is chttp. The transport is implemented as a datagram transport, and therefore it must be used with a connectionless adaptor. For example, assuming an xml-to-Smalltalk binding:

```
<xmlToSmalltalkBinding
   elementFormDefault="qualified"
   name=""
   targetNamespace="urn:TestingHeaders"
   defaultClassNamespace="Opentalk"
   xmlns:tns="urn:TestingHeaders"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="urn:visualworks:VWSchemaBinding">
```

you might set up a SOAP broker for it with:

```
^(BasicBrokerConfiguration new
   requestTimeout: 3000;
   adaptor: (ConnectionAdaptorConfiguration new
      isBiDirectional: false;
      soReuseAddr: true;
      transport: (TransportConfiguration http
         marshaler: (MarshalerConfiguration soap
         binding: aXMLObjectBinding)))
   ) newAtPort: aPort
```

If you have defined and registered an XML to Smalltalk binding for this (see XML to Smalltalk Mapping), named DateCalculatorBinding, this would simplify to:

```
(BrokerConfiguration standard
   adaptor: (AdaptorConfiguration connectionLess
      transport: (TransportConfiguration chttp
         marshaler: (MarshalerConfiguration soap
            bindingNamed: 'DateCalculatorBinding'))))
               newAtPort: 4242
```

To configure an HTTPTransport, use the message http. The transport is a StreamTransport, therefore it has to be used with a connection-oriented transport. Here is an example of how to setup a SOAP broker for it:

```
(BrokerConfiguration standard
    adaptor: (AdaptorConfiguration connectionOriented
        transport: (TransportConfiguration http
            marshaler: (MarshalerConfiguration soap
                bindingNamed: 'DateCalculatorBinding'))))
                    newAtPort: 4242
```

CGITransport

The Opentalk CGITransport class, which is provided in the Opentalk-CGI parcel extends HTTPTransport to support web-server mediated communication through VisualWave CGI relays. This is useful, for example, if you want to use a web server like Apache to serve static content and pass of CGI requests to a VisualWave application.

CGITransport simply listens to a socket, waiting for requests from the relay. CGITransport is obviously only usable for servers, not clients.

Note that CGITransport assumes that application messages can carry contextual information (both XML and SOAP messages do). The transport compiles an "environment" dictionary from the environment variables passed by the CGI relay and attempts to install it into the message during unmarshaling. This allows the server application to access the header fields if necessary (e.g. when trying to hook into the security features of the webserver).

The corresponding configuration message for TransportConfiguration is cgi. The transport is a StreamTransport therefore it has to be used with a connection-oriented transport. Here is an example setting up a SOAP broker using CGI:

```
(BrokerConfiguration standard
    adaptor: (AdaptorConfiguration connectionOriented
        transport: (TransportConfiguration cgi
            marshaler: (MarshalerConfiguration soap
                bindingNamed: 'DateCalculatorBinding'))))
                    newAtPort: 4242
```

Configuring the VisualWave CGI relay is described in the Web Server . Configuring the relay involves:

1 Copying the appropriate cgi2vw VisualWave CGI relay executable to the server, and renaming it appropriately.

2 Copying the cgi2vw.ini file to a VisualWave directory, and setting access permissions.

3 Editing the HTTP server configuration file appropriately to find the INI file.

4 Configuring the CGITransport to use the VisualWave CGI relay.

Refer to the Web Server Configuration Guide for detailed instructions.

7

XML to Object Binding Wizard

The VisualWorks web services framework includes wizards and builders that can automatically generate Smalltalk classes for use in your application.

The XML-to-Object Binding wizard enables you to ascribe types to domain classes, to create X2O bindings, and to test the marshaling and unmarshaling behavior of these classes.

X2O bindings may also be used by any application (not merely web services) that needs to serialize and deserialize Smalltalk classes to/from XML. For a more detailed discussion of XML to object translation, see: XML to Smalltalk Mapping.

Using the XML-to-Object Binding Wizard

The XML to Object Binding Wizard helps to create X2O binding and XML schema specifications for your domain classes. These X2O bindings may be used when building web services applications, or any other application that needs to translate between Smalltalk objects and XML documents.

In the case of a web services application, an X2O binding provides descriptions of the parameters to each operation that belongs to a service. These descriptions include type information, which is represented using Smalltalk classes (i.e., each type is actually a class). In the WSDL schema for a given application, the types in an X2O binding element apper in the <types> element of the schema.

Specifically, the wizard simplifies the task of assigning classes as types for the operation parameters that will be passed to service classes.

Before using the wizard, you must also load the XMLObjectBindingWizard package (for instructions, see: Loading Support for Web Services).

An Example Application

To illustrate the use of the wizard, consider a simple web service that defines a Customer class like this:

```
Smalltalk defineClass: #Customer
superclass: #{Core.Object}
indexedType: #none
private: false
instanceVariableNames: 'name emailAddress physicalAddress
telephoneNumbers '
classInstanceVariableNames: ''
imports: ''
category: '(none)'
```

Let's say, further, that the web service includes an operation to fetch the address of a particular Customer instance, e.g.:

```
addressFor: aCustomer
```

In order to send instances of class Customer to this service, we need to be able to translate it into XML. The web services framework does this automatically, but the application developer must first specify the types of class Customer's instance variables. In VisualWorks, these types are represented as classes.

For the purposes of this example, we'll specify these types using a mix of simple and complex types, the latter defined by the Library Demo:

Instance Variable	Type
name	String
emailAddress	Protocols.Library.EmailAddress
physicalAddress	Protocols.Library.PhysicalAddress
telephoneNumbers	Collection of Protocols.Library.TelephoneNumber

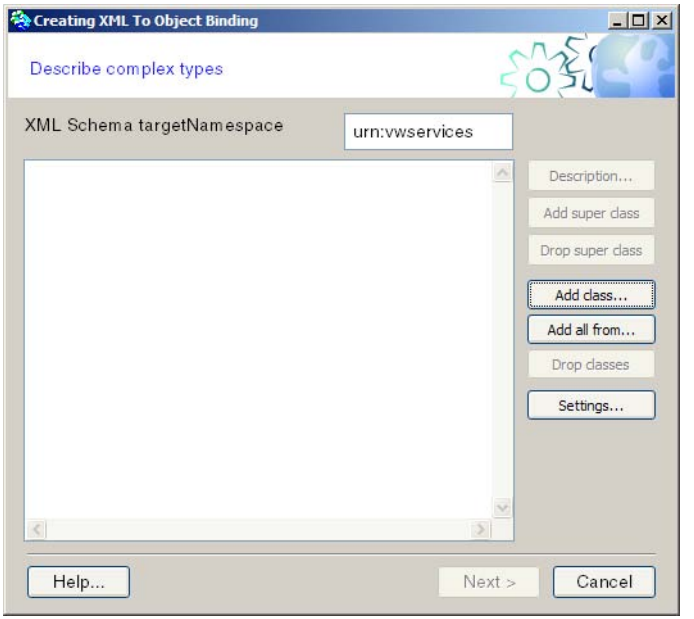
The steps described below show how to create an X2O binding for class Customer.

Creating an XML to Object Binding

To begin, first load the XMLObjectBindingWizard and WebServicesDemo parcels (for instructions, see: Loading Support for Web Services).

1 To launch the wizard, select **XML to Object Binding** from the **Tools** menu in the Visual Launcher.

Use the first page of the wizard to describe types, by building up a list of classes.



2 To begin, set the XML Schema targetNamespace attribute.

For this example, use urn:CustomerSchema.

3 Next, add classes to the binding using the **Add class...** button.

For this example, choose class Customer, as we defined it above.

(To remove a class from the list, select it and click on the **Drop Classes** button. To add all classes in a package, use **Add all from...**.)

4 After adding some classes, select one and click the **Description** button.

In the **Description** dialog, use the <Operate> menu in the **Type** field to set the following values:

a For **name**, select Simple Types • String.

b For **emailAddress**, select Complex Type.., and enter Protocols.Library.EmailAddress (start by entering the name of the class, e.g. EmailAddress).

c For **physicalAddress**, select Complex Type.., and enter Protocols.Library.PhysicalAddress.

d For **telephoneNumbers**, select Collection.., and enter Complex Type • Protocols.Library.TelephoneNumber.

When finished, click Accept.

The type description for each instance variable in the class is stored as a pragma definition in the corresponding instance setter method.

If the class doesn't have accessor methods they are created by the wizard. The names of the accessor methods are used in the X2O binding's aspect attribute.If the class doesn't have any type description, a **Caution!** symbol will appear next to its name. A class can have descriptions for some, all of none of the instance variables, but in general all should be defined to produce a correct WSDL schema.

Repeat this step as necessary, until all classes have been described.

5 At this point, the following list of classes appears:

Customer

Protocols.Library.EmailAddress

Protocols.Library.PhysicalAddress

Protocols.Library.TelephoneNumber

6 Select **Protocols.Library.PhysicalAddress** and click Add super class. The class **AbstractRandom** will be added to the list.

7 Class AbstractRandom doesn't have any instance variables and won't have a type description, so you can ignore the **Caution!** icon that appears next to its name.

8 When all classes are described, click **Next**.

9 On the **Validate and Load XML To Object Binding** page, you can edit the specification as desired.

10 When the specification has been finalized, click **Next**. This loads and validates the schema, and then registers it in the XMLObjectBinding registry.

11 On the **Create XML schema description** page, you can update and save the XML Schema to a particular destination.

To save the schema, select an option in the **Destination** section:

- **Method** writes the schema to a class-side method. Provide a method name (#wsdlSchema is the default) and specify the class by clicking on **Select...**.
- **File-Out** writes the schema to an external file in plain text format.
- **POST URL** posts the schema to an HTTP server using the specified URL.

12 When the destination has been specified, click **Next**.

13 The **Testing XML To Object Binding page** script provides a way to test the newly created X2O bindings.

Code fragments are provided to marshal Smalltalk objects into an XML document, and unmarshal an XML document into a Smalltalk object.

Here, you can create instances of the XML to Object binding classes and execute the script using Do It and Inspect It.

14 To close the tool, click **Finish**.

8

XML to Smalltalk Mapping

XML has become a standard format for data exchange over the web, and is the essential foundation for deploying Web Services via SOAP and WSDL. To accommodate this use of XML in VisualWorks, a mechanism is required for mapping XML elements and attributes to Smalltalk objects, and back again. VisualWorks includes an XML-to-Object marshaling mechanism that performs this mapping, and is the foundation for the rest of VisualWorks web service support.

The XML-to-object mapping is generated based on a binding specification, which is an XML document written using a syntax described later in this chapter. From this mapping, the marshaling engine produces prototype objects. The objects are stored in a registry that maps the prototype objects to tag names in the binding document. This registry is then used by the engine to marshal and unmarshal an XML document.

The XML-to-object mechanism supports marshaling for:

- simple types to simple types;
- simple types with attributes to complex objects;
- complex elements to complex objects;
- elements, attributes, or text to aspects of Smalltalk objects.

The XML-to-object mechanism also supports a few important object marshaling alternatives. For instance, it can marshal XML either as an object with aspects or as a “struct,” that is, marshal to a Dictionary, handling aspects by at:, at:put: messages. This allows you to marshal objects for which no specific domain object has yet been defined, permitting you to start with an intermediate implementation and then gradually refine it. Collections can also be used in this way.

Aspects can be defined easily. Both scalar aspects and repeating groups (collection-valued aspects) are supported.

The XML to object framework supports a number of common primitive types, including strings, numbers, token lists, URI references, namespace-qualified names and name references, etc. To resolve a primitive type, serialization, deserialization and initializer blocks are used. The blocks are stored in BindingBuilder class registries.

Core framework classes

The primary marshaling classes are implemented as subclasses of XMLTypeMarshaler:

```
Object
    BindingBuilder
    XMLTypeMarshaler
        BindingImport
        HrefMarshaler
        ObjectMarshaler
            ComplexObjectMarshaler
            CollectionObjectMarshaler
            KeyObjectMarshaler
                KeyRefObjectMarshaler
            SoapArrayMarshaler
        RelationMarshaler
            AnyRelationMarshaler
                BodyMarshaler
            ChoiceMarshaler
                ChoiceRelation
            GroupMarshaler
        RestrictionMarshaler
        SimpleTypeMarshaler
            SimpleObjectMarshaler
            UnionMarshaler
        XMLObjectBinding
            SoapBinding
            WsdlBinding
```

Most of these classes are marshalers for various XML types, and are described below (see XML marshalers).

BindingBuilder is responsible for generating bindings from binding specification documents, maintaining a registry of mappings from XML tags to prototype marshalers, and blocks for resolving primitive types.

XMLObjectBinding maintains another registry, a registry of bindings for specific schemas. It responds to requests to find a marshaler for a specific XML tag or Smalltalk object.

Creating XML-to-Object bindings

XML-to-object translation is bidirectional. Given a Smalltalk object to be sent to a service, the object must be marshaled as an XML data element. And, given an XML element, it must be unmarshaled as a Smalltalk object so it can then be processed by a Smalltalk application.

An XML document generally has a schema describing the structure and type of elements and attributes that the document may contain. Based on such a schema, you create a binding specification that tells the XML-to-object engine how to map the XML objects to Smalltalk objects. The binding specification determines two items: the marshaler for element or attribute, and the Smalltalk class to represent the item.

Each binding has a target namespace (URI) and an optional name. All NCNAMEs defined in this binding belong to their target namespace. Each binding is represented by an instance of XMLObjectBinding or a subclass. The main purpose of a this class is to serve requests to its marshaler according to its name (tag). The bindings with a target namespace register themselves in the XMLObjectBinding class-side binding registry. These bindings can be obtained by the target namespace.

The binding specification is itself an XML document that is constructed using element tags stored in the BindingBuilder Registry shared (class) variable. The element tag identifies the marshaler. Several marshalers are already defined, though you can create additional marshalers. Elements may take several attributes, one of which typically specifies the Smalltalk class to model the XML data, if a class other than the default class for the marshaler is required.

An application may register its own marshalers. An important point is that the registry is prototype based; it contains preconfigured instances of marshalers, not classes. One can view registry entries as a sort of macro. This allows us to have a very small set of highly configurable marshaler types, which is convenient because every marshaler is potentially a small cluster of objects: the marshaler itself, its XPath proxy and parser, and potentially a relation description,

aspect implementation, and marshaler for the far side of the relation. To create a new instance of a marshaler we, make a copy of an entry in the registry.

An instance of RelationMarshaler can be configured to marshal elements, attributes or text; the complex object marshaler can be configured to marshal objects, dictionaries, collections or streams. On top of that, a XPath expression can be specified to define rules of XML parsing/composition. To make it usable, the registry contains some of the most frequent and useful configurations (i.e. 'object' and 'struct', 'sequence_of', 'element', 'attribute', 'text', etc.). These are not only preconfigured, but implicitly create XPath expressions based on the values of other attributes.

For example, consider this simple binding specification:

```
schema := '<schemaBindings >
   <xmlToSmalltalkBinding name="RandomGenerator"
      xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
      targetNamespace="urn:random">

   <sequence_of name="numbers">
      <implicit ref="xsd:float"/>
   </sequence_of>

   <struct name="NextBatch">
            <element name="number" ref="xsd:integer" />
   </struct>
   </xmlToSmalltalkBinding>
</schemaBindings >'.
```

There are two elements, introduced by the sequence_of and struct tags. The binding registry contains entries for each of these, mapping the tags to CollectionObjectMarshaler and ComplexObjectMarhsaler, respectively.

BindingBuilder reads the binding specification, and maps each tag to a copy of the marshaler object registered under the tag. Only tag types are compared, ignoring namespaces.

The initial set of binding specification tags and their marshalers are summarized below.

Tag type	Marshaler class
any	AnyRelationMarshaler
anyCollection	AnyCollectionMarshaler
attribute	RelationMarshaler
bindingImport	BindingImport
choice	ChoiceMarshaler
choiceRelation	ChoiceRelation
element	RelationMarshaler
enumeration	RestrictionMarshaler
group	GroupMarshaler
identityStruct	ComplexObjectMarshaler
implicit	RelationMarshaler
key	KeyObjectMarshaler
keyRef	KeyRefObjectMarshaler
object	ComplexObjectMarshaler
sequence_of	CollectionObjectMarshaler
simple	SimpleObjectMarshaler
struct	ComplexObjectMarshaler
text	RelationMarshaler
union	UnionMarshaler
xmlToSmalltalkBinding	XMLObjectBinding

Creating a binding specification

To create a binding specification, create an XML binding specification document. The document may be an external file, or the return value of a method. For example, browse the binding specification in the XMLObjectBinding class method defaultSoapBindingLocation.

The document begins with the usual prefix information, identifying the XML version. You'll probably include a comment as well:

```
<?xml version ="1.0"?>
<!-- MyService to Smalltalk binding -->
```

The body of the document is an xmlToSmalltalkBinding element (or some other XML-to-object binding)

```
<xmlToSmalltalkBinding name="MyServiceBinding"
   defaultClassNamespace="MyCo"
   targetNamespace="http://www.myco.com/schemas/myservice"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
</xmlToSmalltalkBinding>
```

Namespaces are declared in this tag as shown, including the namespace for your service's schema and any others, such as the XML Schema namespace.

You can import other bindings, using the bindingImport tag, and giving the name of the binding. For example:

```
<!-- Imports -->
<bindingImport name="SoapBinding"/>
```

where "SoapBinding" specifies another binding (in XMLObjectBinding.BindingRegistry).

The rest of the document, before the </xmlToSmalltalkBinding> close tag, consists of specifications for individual bindings. Several examples are shown in the following section (see Binding specification examples, below).

Binding specification examples

The following examples illustrate binding specifications for several object types.

Simple objects

Simple objects are objects that have an opaque structure, and include the common primitive data types, such as string or float:

```
<simple name="string" id="String" />
<simple name="float" id="Float" />
```

Simple objects are identified by their conversion ID and are resolved using serialization/deserialization blocks that are defined for several Smalltalk classes (browse the BindingBuilder class method initializeSerializationBlocks). The value comes from or is put into element character data.

While loading the XMLToObjectBinding class, this XML schema binding for simple types is loaded and registered:

http://www.w3.org/2001/XMLSchema

XMLToObjectBinding class method defaultXsdBindingLocation2001.

Simple object descriptions can include constraining facets. Currently we provide support only for "enumeration" facet (enumeration constrains the value space to a specified set of values). For details, see: http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/s.

Complex objects

Complex object are objects with relations. These objects are described by their type and the set of their relations with other objects. Relations can be implemented in a number of different ways and are a generalization of such notions as attributes, aspects, members of a collection, etc. A complex object knows how to get/set value for a specific relation.

As examples of complex objects, consider the following XML complex type element information items:

```
<complexType name="LDDataPersonName">
   <sequence>
      <element name="title" type="string"/>
      <element name="firstName" type="string"/>
      <element name="lastName"  type="string"/>
   </sequence>
</complexType>
```

and

```
<complexType name="Document">
   <simpleContent>
      <extension base="string">
         <xsd:attribute name ="ID" type="string"/>
      </extension>
   </simpleContent>
</complexType>
```

Installing a binding

To install the bindings in a binding specification document, send a loadFrom: message to the XMLObjectBinding class. Minimally, the command would be, if the specification is defined in the myBindingSpec class method in your application:

```
WebServices.XMLObjectBinding
   loadFrom: self myBindingSpec readStream
```

For an example with condition checking, see the XMLObjectBinding class method loadVWBinding and similar methods.

Your binding is now loaded into XMLObjectBinding.BindingRegistry, and ready for use.

When you update Web services with a new version, it make sense to clean up the XML to object registry and reload you bindings. To reset the registry, send:

```
WebServices.XMLObjectBinding configure.
```

Then load your bindings.

XML marshalers

Marshalers convert XML elements and/or their parts to Smalltalk objects, and vice versa. Each marshaler matches some XML node or collection of nodes. Therefore, each marshaler has an associated XPath expression and XPath parser for this expression. Since XPath is a kind of object in itself, there is a bridge between marshaler and XPath parser—XMLMarshalerProxy.

XML marshalers interact with a MarshalingContext, which maintains all the context needed to perform marshaling and unmarshaling.

All marshalers are subclasses of XMLTypeMarshaler, and must support the following protocols:

marshalFrom: *marshalingContext*

Marshal Smalltalk object as XML.

unmarshalFrom: *marshalingContext*

Unmarshal XML as a Smalltalk object.

Marshalers also implement methods that facilitate marshaling of compound objects.

The marshaler hierarchy contains three main categories of marshalers

- Type marshalers - These are the “real” marshalers that marshal (parts of) XML elements to Smalltalk objects. Several are described below.
- XML-to-object bindings - These are high-level repositories of binding info. They are usually root elements in the binding spec.

An XML-to-Object binding contains the list of all element marshalers and can find a marshaler for specific XML tag or Smalltalk object. Bindings can import other bindings.

- Others - These are not real marshalers, but rather serve to configure other marshalers. An example is BindingImport which is only used to add import definitions to bindings.

Marshaling XML entity types

Mashaling XML <simpleType> elements

SimpleObjectMarshaler instances are used to marshal simple types like strings and numbers, and so implement known primitive types that are defined in XML Schema datatypes. Primitive type marshalers implement methods serialize: and deserialize:, and have two blocks that actually perform the conversion between an XML string and a Smalltalk object. BindingBuider maintains dictionaries of serialization and deserialization blocks keyed by id (i.e. 'String', 'date', 'binary').

For example, for the XML data type “dateTime” the mapping is:

```
<simple name="dateTime" id="dateTime"/>
```

The serialization and deserialization blocks are:

```
Deserializers at: 'date'
   put: [[ :mv :string | self decodeDateFrom: string ]].

Serializers at: 'date'
   put: [[ :mv | self encodeDate: mv value ]].
```

The XML simple type can be described as:

```
<s:simpleType name="Title">
   <s:restriction base="s:string">
      <s:enumeration value="Mr." />
      <s:enumeration value="Mrs." />
   </s:restriction>
</s:simpleType>
```

The element will be described in the XML to object binding:

```
<element name="Title" ref="s:string">
   <s:enumeration value="Mr." />
   <s:enumeration value="Mrs." />
</element>
```

While marshaling/unmarshaling the "Title" element, the SimpleObjectMarshaler will validate the string value and raise an EnumerationValueError exception if the value does not equal 'Mr.' or 'Mrs.'.

Marshaling XML <complexType> elements

ObjectMarshaler is superclass for marshaling/unmarshaling complex objects, such as structs, objects, and collections.

ComplexObjectMarshaler and its subclasses marshal objects with parts, such as:

struct

Marshal an object as a Dictionary, for accessing parts using an at: message.

object

Marshal objects with aspects (accessor methods).

Marshaling XML complex types as Dictionaries

The XML <complexType> element information item can be mapped to a <struct> binding element, which is the default mapping for XML complex types.

The XMLToObjectBinding descriptions for the two XML elements above would be:

```
<struct name="LDDataPersonName">
   <element name="title" aspect="title" ref="xsd:string"/>
   <element name="firstName" aspect="firstName" ref="xsd:string"/>
   <element name="lastName" aspect="lastName"  ref:="string"/>
</struct>
```

and

```
<struct name="Document">
   <text aspect="value" ref="xsd:string"/>
   <attribute name ="ID" aspect="id" ref="xsd:string"/>
</struct>
```

A <struct> binding element is, by default, marshaled by the ComplexObjectMarshaler as an instance of WebServices.Struct (a subclass of Dictionary), with its constituent objects as entries. The dictionary entries are specified by the "aspect" attribute.

The XML element described by the first struct is unmarshaled into:

```
WebServices.Struct new
   at: #title put: 'Mr.';
   at: #firstName put: 'John'
   at: #lastName put: 'Smith'.
```

The second struct is represented as:

```
WebServices.Struct new
   at: #value put: 'some text';
   at: #id put: '1234'.
```

Marshaling XML complex types as objects

The XML <complexType> element information items can be mapped to <object> binding elements. In this case, the samples above would be described as:

```
<object name="LDDataPersonName" smalltalkClass = "PersonName">
   <element name="title" aspect="title" ref="xsd:string"/>
   <element name="firstName" aspect="firstName" ref="xsd:string"/>
   <element name="lastName" aspect="lastName"  ref:="string"/>
</object>
```

and

```
<object name="Document" smalltalkClass = "SampleDocument">
   <text aspect="value" ref="xsd:string"/>
   <attribute name ="ID" aspect="id" ref="xsd:string" setSelector="setID"
      getSelector="getID"/>
</object>
```

An <object> binding element is marshaled by the ComplexObjectMarshaler as an instance of the Smalltalk class with the name specified by the smalltalkClass attribute.

When the XML to object binding is loaded the Smalltalk class must exist; otherwise, the binding builder is raises an ClassIsNotDefinedSignal exception, and the handler builds the class. Smalltalk classes are created with any instance variables that are specified in "aspect" attributes, and with accessor methods for those instance variables.

In our examples, the first XML element described by the first "object" is would be unmarshaled to:

```
PersonName new
   Title:  'Mr.';
   firstName: 'John';
   lastName: 'Smith'.
```

The second object is unmarshaled as:

```
Doc := SampleDocument new
    value:'some text';
    setID: '1234'.
Id := doc getID.
```

Existing binding attributes

Attribute name	default	description
Relation description:		
name	no	Can be used to set default accessors.
aspect	name	Can be used to set default accessors.
setSelector	aspect	Used as set accessor for the specified aspect.
getSelector	aspect	Used as get accessor for the specified aspect.
minOccurs	0	Sets occurrence constraint.
maxOccurs	1	Sets occurrence constraint; to set unbounded, use '*'.
Resolvers		
ref	no	Used to resolve the simple or complex object identity.
smalltalkClass	no	Used to resolve complex object identity. Default class namespace is 'Smalltalk'; to specify another use:'MyNamespace.MyClass' or setdefaultClassNamespace="MyNamespace"in XMLToObjectBinding for all binding classes
Xpath description:		
xpath	tag or name	Used to construct xpath expression to define rules of xml parsing/ composition.
xpathPrefix	no	Constructs xpath expression xpathPrefix,xpath
xpathSuffix	no	Constructs xpath expression xpath, xpathSuffix

Mapping XML <union> elements

A union defines a collection of simple object definitions.

XML to olbject bindings for union datatypes can be described as:

```
<union name="size">
   <simple baseType="xsd:integer"/>
   <simple baseType="xsd:token">
      <enumeration value="small"/>
      <enumeration value="large"/>
   </simple>
</union>
```

or using memberTypes attribute:

```
<union name="size" memberTypes="tns:integerType tns:booleanType" >
<simple baseType="xsd:token">
      <enumeration value="small">
</enumeration>
      <enumeration value="medium">
</enumeration>
      <enumeration value="large">
</enumeration>
   </simple>
</union>
<simple name="booleanType" baseType="xsd:boolean"/>
<simple name="integerType" baseType="xsd:integer" maxInclusive="18"
   minInclusive="2"/>
   <struct name="Coat" tag="Coat">
         <element name="size" ref="tns:size">
</element>
         <element name="color" ref="xsd:string">
   </element>
</struct>
```

The UnionMarshaler holds a collection of union member type marshalers and tries to use them in the order in which they appear in the definition until a match is found. The marshaler for memberTypes attributes s added first to the marshalers collection.The evaluation order can be overridden with the use of xsi:type.

Marshaling XML <element> elements

An <element … /> binding specifies a relationship between a new object type and an already defined type. Marshaling/unmarshaling is done based on an XPath expression defined as 'child::xxx', where xxx is the value of the name attribute.

A RelationMarshaler handles the marshaling/unmarshaling for objects whose marshalers are specified as the same as for some already defined object type. Accordingly, it collaborates with its parent marshaler to get/set the value to the parent object.

Since relations may be either one-sided or many-sided, and implementation of marshaling is reasonably different, to represent those differences there is a separate class, a subclass of Relation, which represents a relation between an object and its parts. A relation has a descriptive name, get and set selectors, and cardinality constraints.

Note that the meaning of a get or set selector entirely depends on the aspect implementation for that relation. For object aspects, for example, it represents real selectors; for Dictionary aspects these are keys for at: and at:put: methods.

For example, a relation a single relation:

```
<element ref = "personName"/>
```

and a relational with many relations:

```
<element ref = "email" minOccurs = "0" maxOccurs = "*"
   aspect="emailList"/>
```

For example,

```
<element name="firstName" ref="string" />
```

This element specifies that any element named “firstName” is to be handled as if it were "string" and so is marshaled as a simple object to an instance of String. So,

```
<firstName>John</firstName>
```

is unmarshaled as 'John'.

As another example,

```
<element name="money" ref="float" />
```

specifies that a “money” element is to be marshaled as a Float value. So,

```
<money>50.02</money>
```

is unmarshaled to a Float value 50.02.

An <implicit … /> binding is the same as element relation, except for its XPath expression. Marshaling/unmarshaling is done based on XPath expression defined as 'child::*'. This relation is convenient to use for describing collections where the item tag is not important. The marshaler is RelationMarshaler.

For example, for binding description:

```
<sequence_of name = "contacts">
   <implicit ref="contact" />
</sequence_of>
```

and an XML element:

```
<contacts>
   <contactx>first contact</contactx>
<contactx>second contact</contactx>
</contacts>
```

will be unmarshal to

```
OrderedCollection new
   Add: 'first contact';
   Add: 'second contact'.
```

Marshaling XML attributes

An <attribute…/> binding defines relations for XML attributes. Marshaling/unmarshaling is done using the XPath expression '@xxx', where xxx is the value of name attribute. The marshaler is RelationMarshaler.

For example, for binding description:

```
<object name="document" smalltalkClass = "SampleDocument">
<attribute name ="ID" aspect="id" ref="xsd:string" />
</object>
```

and XML element:

```
<document ID='1234'/>
```

the unmarshaled value is an instance of SampleDocument:

```
Sample := SamIpeDocument new
Sample id: '1234'.
```

Marshaling XML values

A <text…/> binding defines relations for XML value. Marshaling/unmarshaling is done using the XPath expression 'child::text()'. The marshaler is RelationMarshaler.

For example, for binding description:

```
<object name="document" smalltalkClass = "SampleDocument">
<text aspect="value" ref="xsd:string" />
</object>
```

and XML element:

```
<document>some text</document>
```

the unmarshaled value is an instance of SampleDocument class:

```
Sample := SamlpeDocument new
Sample value: 'some text'.
```

Marshal XML <any> elements

AnyRelationMarshaler marshals and unmarshals the XML <any> element.

An <any…/> binding is the same as an 'element' relation. The XML to object binding should include descriptions for all objects that can occur in the XML document. If the attempt to find a marshaler fails, an exception is raised. The marshaling/unmarshaling starts with AnyRelationMarshaler which searches for an appropriate marshaler based on the node namespace and type.

For example, for binding descriptions

```
<struct name="GetPerson">
   <any aspect="contents" />
</struct>
<object name="Person1" smalltalkClass="Person1">
   <element name="name" ref="string"/>
</object>
<object name="Person2 smalltalkClass="Person2>
   <element name="age" ref="int"/>
</object>
```

and XML element:

```
<GetPerson>
   <Person1>
      <name>any name</name>
   </Person1>
</GetPerson>
```

or XML element:

```
<GetPerson>
   <Person2>
      <age>10</age>
```

```
    </Person2>
</GetPerson>
```

the unmarshaled object will be a <struct> with entry #contents and value an instance of Person1 or Person2:

```
WebServices.Struct new
    At: #contents put: (Person1 new name: 'any name'; yourself).
```

or

```
WebServices.Struct new
    At: #contents put: (Person2 new age: 10; yourself).
```

Using an <any> relation allows you to specify no aspect in a complex object, but still to use relation aspects from referenced types. For example, for binding description:

```
<struct name="GetPerson">
    <any />
</struct>
<element name="Person1" aspect="person1" ref="Person1Type"/>
<object name="Person1Type" smalltalkClass="Person1">
    <element name="name" ref="string"/>
</object>
<element name="Person2" aspect="person2" ref="Person2Type"/>
<object name="Person2Type" smalltalkClass="Person2>
    <element name="age" ref="int"/>
</object>
```

and XML element:

```
<GetPerson>
    <Person1>
        <name>any name</name>
    </Person1>
</GetPerson>
```

or:

```
<GetPerson>
    <Person2>
        <age>10</age>
    </Person2>
</GetPerson>
```

the unmarshaled object will be struct with entry #person1 and value instance of Person1, or entry #person2 and value instance of Person2:

```
WebServices.Struct new
    At: #person1 put: (Person1 new name: 'any name'; yourself).
```

or

```
WebServices.Struct new
   At: #person2 put: (Person2 new age: 10; yourself).
```

Marshal XML <choice> element

The <choice..> binding is similar to <any>, except that the choice relation is limited to some set of types. The marshaler is ChoiceMarshaler, which holds a registry of available marshalers.

For example, for binding description:

```
<struct name="GetData">
   <choice>
      <element name="Person1" aspect="person1" ref="Person1Type"/>
      <element name="Person2" aspect="person2" ref="Person2Type"/>
      <element name="Data" aspect="data" ref="string"/>
   </choice>
</struct>
<object name="Person1Type" smalltalkClass="Person1">
   <element name="name" ref="string"/>
</object>
<object name="Person2Type" smalltalkClass="Person2>
   <element name="age" ref="int"/>
</object>
```

Marshaling/unmarshaling will be done based on three types: Person1Type, Person2Type and string.

Marshaling XML <group> and <attributeGroup> elements

When we create an XML-to-Object binding we map the attributeGroup and group elements as follows:

1 elements with name are mapped to struct type

2 elements with ref are mapped to group

For example, the XML schema:

```
<xsd:group name="myGroup">
   <xsd:sequence>
      <xsd:element name="group1" type="xsd:string"/>
   </xsd:sequence>
</xsd:group>
<xsd:attributeGroup name="myAttrs">
   <xsd:attribute name="grAttr1" type="xsd:string"/>
   <xsd:attribute name="grAttr2" type="xsd:string"/>
</xsd:attributeGroup>
<xsd:element name="Item">
```

```
<xsd:complexType>
   <xsd:sequence>
   <xsd:group ref="myGroup"/>
      <xsd:element name="aaa">
         <xsd:complexType>
            <xsd:attributeGroup ref="myAttrs"/>
               <xsd:attribute name="aaaName" type="xsd:string"/>
         </xsd:complexType>
      </xsd:element>
   </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

will be mapped to:

```
<xmlToSmalltalkBinding ..>
      <struct name="myAttrs">
         <attribute name="grAttr1" ref="xsd:string"/>
         <attribute name="grAttr2" ref="xsd:string"/>
      </struct>
      <struct name="myGroup">
         <element name="group2" ref="xsd:string"/>
      </struct>
      <struct name="Item" tag="Item">
         <group name="myGroup" ref="tns:myGroup"/>
         <element ref="tns:Item_aaa" tag="aaa"/>
      </struct>
      <struct name="Item_aaa" tag="aaa">
         <group name="myAttrs" ref="tns:myAttrs"/>
         <attribute name="aaaName" ref="xsd:string"/>
      </struct>
   </xmlToSmalltalkBinding>
```

The request arguments for the XML-to-Object binding above should be prepared as:

```
anObj := WebServices.Struct new.
anObj
   group1:'astring';
   aaa: (WebServices.Struct new
   grAttr1: 'grAttr1';
         grAttr2: 'grAttr2';
         aaaName: 'aaName';
         yourself).
```

Marshaling collections

CollectionObjectMarshaler marshaler converts a sequence or XML elements and/or their parts to a Smalltalk collection of objects, and vice versa. The “sequence-of” tag introduces a homogeneous collection of items with known types or type ANY.

Describing collection using cardinality

The marshaler is RelationMarshaler. For a binding description:

```
<object name="document" smalltalkClass="SoapDocument">
   <element name="details" aspect="detailsCollection" minOccurs=1
      maxOccurs="*" />
</object>
```

and XML element:

```
<document>
   <details>some description1</details>
   <details> some description1</details>
</document>
```

the unmarshaled Smalltalk object would be:

```
Doc := SoapDocument new.
Doc detailsCollection: (OrderedCollection with: 'some description1'
   with: 'some description2').
```

Describing collection using <sequence_of>

The marshaler is CollectionMarshaler. For a binding description:

```
<object name="document" smalltalkClass="SoapDocument">
   <element name="details" aspect="detailsCollection" ref="details" />
</object>
<sequence_of name="details">
   <implicit ref="string"/>
</sequence_of>
```

and XML element:

```
<document>
   <details>
      <item>some description1</item>
      <item> some description1</item>
   </details>
</document>
```

the unmarshaled smalltalk object would be:

```
Doc := SoapDocument new.
Doc detailsCollection: (OrderedCollection with: 'some description1'
   with: 'some description2').
```

Describing collection using <soapArray>

Currently, soapArray support is limited to multi-dimensional arrays. Partially transmitted and sparse arrays are not supported. The marshaler is SoapArrayMarshaler.

In a WSDL schema, the soapArray is usually described as complexType:

```
<complexType name="ArrayOfDetails">
   <complexContent>
      <restriction base="SOAP-ENC:Array">
         <xsd:sequence>
            <xsd:element name="item" type="string"/>
         </xsd:sequence>
         <attribute ref="SOAP-ENC:arrayType"
            wsdl:arrayType="string[1,3]"/>
      </restriction>
   </complexContent>
</complexType>
```

For example, for an XML to object binding description:

```
<object name="document" smalltalkClass="SoapDocument">
   <soapArray name=" ArrayOfDetails" aspect="details" ref="xsd:string"
      dimension="1" dimSize ="3" elementTag="item"/>
</object>
```

The attributes dimension, dimSize and elementTag are optional.

Then, the XML element:

```
<document>
   < ArrayOfDetails >
      <item>some description1</item>
      <item> some description1</item>
      <item> some description3</item>
   </ ArrayOfDetails >
</document>
```

will be unmarshaled to the Smalltalk object:

```
Doc := SoapDocument new.
Doc details: (Array with: 'some description1' with: 'some description2').
```

Resolving object identity using <key> <keyRef>

KeyObjectMarshaler marshals and unmarshals between an XML <key> element and a Smalltalk object.

In some cases we would like to relate an XML node to an already existing object. This is important in case of multi-reference nodes and other scenarios. These are resolved using an approach that mimics the XMLSchema approach, using keys and key references.

A TYPE may have zero or more keys. A key may consist of one or more fields. Each field may be any XPath expression, although the current implementation limits key generation to a simple XPath expression consisting of an element or attribute name. Each key has a name, which is a qualified name whose namespace is the target namespace of the binding. Since keys consist of parts, they are similar in their structure to complex types and use the same notation.

A RELATION may have zero or one key reference marshalers. A key reference references the key it corresponds to and must have the same number of fields as the corresponding key so that the fileds match in their respective types.

In the presence of keys and key references, object identity is resolved as follows:

1 When a type is being unmarshaled, its keys are computed and registered with the marshaling manager. The manager stores each key in a dictionary where the key is an association (key marshaler -> key fields) and the value is the target Smalltalk object.

2 When a relation is unmarshaled and the marshaler contains a key reference, this key reference is unmarshaled first and then the marshaler calls the marshaling manager to find a value for the association (key reference's key, unmarshaled fields). If match is found, its value is used as a target. Otherwise, the marshaling manager places this association in the list of actions waiting for resolution. Every time a new key is registered, this list is rescanned. When a match is finally found, a callback is evaluated the same way as if match was found right away. This allows for handling forward references.

3 If the relation does not have any key references, the marshaling manager resolves object identity by invoking the method `newInstanceFor:` on the marshaler. Most marshalers would then answer a new instance of Smalltalk class stored in it. This provides for the default resolution behavior when creating new instance of the class.

For example, suppose we want to unmarshal an XML document and resolve the key reference in the GetAuthToken object as a "structToken" struct. The "structToken" object has defined a "userID" attribute as a key with name "tokenKey". The GetAuthToken object defines the "favorite" attribute as a key reference that should be resolved to an object with key "tokenKey" and value the same as "favorite".

The XML to object binding is:

```
<xmlToSmalltalkBinding >
<struct name="structToken">
      <key name="tokenKey">
         <attribute name="userID" ref="string" minOccurs="1"/>
      </key>
      <attribute name="userID" ref="string" minOccurs="1" />
      <attribute name="name" ref="string" />
      <attribute name="favorite" ref="string"/>
      <element name="cred" ref="string" />
   </struct>
   <object name="get_authToken" smalltalkClass="GetAuthToken">
      <attribute name="userID" ref="string" minOccurs="1" />
      <attribute name="name" ref="string" />
      <attribute name="favorite" ref="structToken">
         <keyRef ref="tokenKey">
            <attribute name="favorite" ref="string"/>
         </keyRef>
      </attribute>
      <element name="cred" ref="string" />
   </object>
   <object name="tokens" smalltalkClass="Association">
      <element name="key" ref="structToken"/>
      <element name="value" ref="get_authToken"/>
   </object>
</xmlToSmalltalkBinding>'
```

The XML document is:

```
<tokens xmlns="mynamespace">
   <key   name="alex"   userID="1234"  favorite="5678">
      <cred>my credentials</cred>
   </key>
   <value  name="fred"  userID="5678" favorite="1234">
      <cred>freds credentials</cred>
   </value>
</tokens>
```

Unmarshaling results in:

```
Assoc := Association key: keyObject value: valueObject.
(keyObject := WebServices.Struct new)
    at: #userID put:'1234';
    at: #favorite put: '5678';
    at: #cred put: 'my credentials';
    at: #name put: 'alex'.
ValueObject := GetAuthToken new.
ValueObject
    Name: 'fred';
    UserID: '5678';
    Cred: 'freds credentials';
    Favourite: keyObject.  ß resolved to association key
```

Invoking a marshaler

Marshalers, once registered, are invoked using a simple API defined in XMLObjectMarshalingManager:

marshal: *anObject*

Marshals anObject into its XML representation.

unmarshal: *anXmlNode*

Unmarshals anXmlNode into a Smalltalk object.

An exception is raised if an appropriate marshaler is not specified in the registry.

Adding new marshalers

The available bindings can be extended by creating new a marshaler and adding it to BindingBuilder.Registry, associated with a tag. The tag can then be used in binding specifications to invoke the new marshaler.

BindingBuilder expects prototype marshalers to support the following builder API:

add: *child*

Adds a submarshaler, child, to the receiver

setAttributesFrom: *dictOfAttributes* **in:** *builder*

Selectively sets attributes in dictOfAttributes.

add: is implemented using double-dispatch as:

```
add: child
   child addTo: self
```

addTo: is implemented polymorphically, so a child has to know how to add itself to its parent. For example, the attribute and aspect marshalers add themselves differently, as does a binding import marshaler, which is not a real marshaler.

Each prototype marshaler registers itself with its parent by sending register: to the builder. The builder then sends add: to its parent (top element on the stack).

Registering the marshaler

Once you have created a new marshaler, you need to add it to the prototype marshaler registry, BindingBuilder.Registry. This is a Dictionary with tag names as keys and instance creation expressions as values.

To add your new marshaler, you send an at:put: message to the registry, possibly from an initialization method in your application. (Browse the BindingBuilder class method initializePrototypeMarshalers for an extended example.)

For example, suppose we have a marshaler class Bar and we want to invoke it by the tag “foo”. Register it by sending:

```
Net.BindingBuilder.Registry
   at: 'foo'
   put: (Bar newProxy xpathPrefix: 'self::';yourself).
```

The argument to xpathPrefix: will be the appropriate XPath axis for the XML element your marshaler handles.

Marshaling exceptions

The following exceptions may be raised during marshaling and/or unmarshaling, to allow higher-level code decide what to do.

XMLObjectBindingSignalEnumerationValueErrorMissingValueException
  MissingValueNotificationNoMarshalerSignal
NoRelationMarshalerSignal
  ObjectNotResolvedSignal
  UnResolvedReferenceSignal
  XMLDatatypeError

NoMarshalerException indicates that no known marshaler is available for marshaling or unmarshaling. For example, we use the no-marshaler exception for unmarshaling a SOAP envelope. The content of a SOAP body is described as ANY, and is defined in a schema other than the one that describes the SOAP envelope. Sometimes we would like to be able to unmarshal both envelope and body contents all the way through, other times we would like to stop at the body contents. The missing-marshaler exception enables this mechanism.

Another use would be if the schema/namespace of the body contents is not known in advance. In this case, the missing-marshaler exception acts as a callback to locate, load, and activate the body contents schema.

MissingValueException indicates that there is no value to marshal or unmarshal. Higher-level code can then, for example, set the element to a default value, stop unmarshaling the current path, or pass exception outwards. This is useful for defining policies to handle optional elements.

ObjectNotResolvedSignal signals that the XML element was not resolved to a Smalltalk object.

UnresolvedReferenceSignal signals that the XML element reference was not mapped to a Smalltalk object. Class BindingBuilder uses the exception for resolving elements.

XMLDatatypeError is raised when the XML data type does not match a Smalltalk object, for example, if XML data defined as type int includes non-digit characters.

EnumerationValueError signals that the validation failed for the simple value ClassIsNotDefinedSignal signals when the class specified by the smalltalkClass attribute does not exist.

Index

X