Cincom

Cincom Smalltalk™ VisualWorks®

Web Services Guide

VisualWorks 8.3

P46-0142-11

SIMPLIFICATION THROUGH INNOVATION

Notice

This product contains copyrighted third-party software.

Part Number: P46-0142-11

Software Release: 8.3

This document is subject to change without notice.

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 VisualWorks *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 a set of on-line VisualWorks Tutorials, and for an overview of Smalltalk, the VisualWorks development environment and its application architecture, see 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:

"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.

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

"Building Web Service 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.

"Document Processing" provides a more detailed discussion of WSDL document processing in the VisualWorks web services framework.

"SOAP Messages" describes SOAP messaging services, how to create a SOAP request, with or without a WSDL document, and how to manipulate SOAP headers.

"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.

"XML to Object Binding Tool" describes the XML-to-Object binding wizard, used for ascribing types to domain classes, creating X2O bindings, and XML schemas.

"XML to Smalltalk Mapping" describes the XML-to-Smalltalk mapping framework.

Conventions

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

Typographic Conventions

The following fonts are used to indicate special terms:

Examples	Description
template	Indicates new terms where they are defined, emphasized words, book titles, and words as words.
`c:\windows`	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 tools.
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 > **Open...**	Indicates the name of an item (**Open...**) 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>

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 personal-use 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 issues, send email to: helpna@cincom.com.

Before contacting Technical Support, please be prepared to provide the following information:

- The release number, which is displayed in the Welcome Workspace when you start VisualWorks.
- Any modifications (patch files, auxiliary code, or examples) distributed by Cincom that you have loaded into the image. Choose **Help > About VisualWorks** in the VisualWorks Launcher window. All installed patches can be found in the resulting dialog under **Patches** on the **System** tab.
- The complete error message and stack trace, if an error notifier is the symptom of the problem. To do so, use **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.
- The hardware platform, operating system, and other system information you are using.

You can contact Technical Support as follows:

E-mail	Send questions about VisualWorks to: helpna@cincom.com.
Web	Visit: http://supportweb.cincom.com and choose the link to Support.
Telephone	Within North America, call Cincom Technical Support at +1-800-727-3525. Operating hours are Monday through Friday from 8:30 a.m. to 5:00 p.m., Eastern time. Outside North America, contact the local authorized reseller of Cincom products.

Personal-Use Licensees

VisualWorks Personal Use License (PUL) 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 an important public resource for VisualWorks developers:

- A mailing list for users of VisualWorks PUL, which serves a growing community of 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.

The Smalltalk news group, comp.lang.smalltalk, carries on general discussions about different dialects of Smalltalk, including VisualWorks, and is a good source for advice.

Online Help

VisualWorks includes an online help system.

To display the online documentation browser, open the **Help** pull-down menu from the VisualWorks main window's menu bar (i.e., the Launcher window), and select one of the help options.

Additional Sources of Information

Cincom provides a variety of tutorials, specifically for VisualWorks.

The guide you are reading is but one manual in the VisualWorks documentation library.

Complete documentation is available in the `/doc` directory of your VisualWorks installation.

Chapter

1

Introduction to Web Services

Topics

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.

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" (from: Web Services Architecture Requirements).

To make a service available, the 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 widely used, and so are not supported at this time.

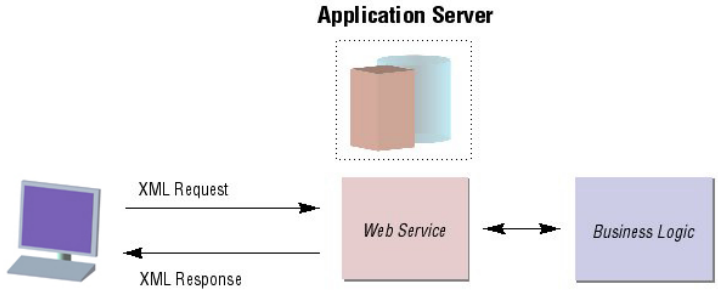
The following pages explore the high-level architecture of the VisualWorks web services framework, and offer suggestions to help you find the discussions in this guide that are most relevant to your specific development tasks.

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 then 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 numbers, the operations that can be performed, the formats of the input and output messages, and the exceptions that can be raised.

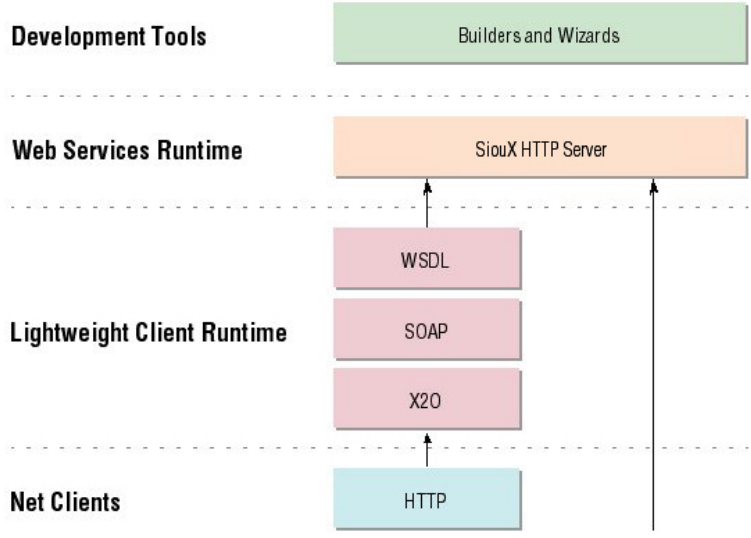
The SOAP protocol is often used with WSDL to provide a web service. WSDL provides the interface description of the service, and SOAP is then used to actually call the service over the network. SOAP provides a way to represent messages to be executed by a remote service provider. It facilitates the exchange 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 of this guide 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 SiouX framework provides a full-featured HTTP server. When building server applications that support WSDL and SOAP, this is augmented with the SOAP-Server package.

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 X2O specification. This specification (in fact, a XML document) describes the mapping of schema types into Smalltalk objects. Complex XML types may be mapped into custom Smalltalk domain classes (the web services framework can also build these classes from a X2O specification).

To actually marshal Smalltalk objects into XML and vice versa, class BindingBuilder uses the XMLTypesParser to create a X2O specification. From this, the builder creates Smalltalk classes, then class XMLObjectBinding is used to load the X2O specification, creating and registering marshalers in the XMLBindingRegistry.

WSDL

VisualWorks' WSDL support provides simple mechanisms for:

- Loading and parsing WSDL documents

- Programmatically invoking a web service based on the port information in a WSDL document
- Generating a WSDL document from Smalltalk classes
- Creating Smalltalk classes from user-defined object types 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 document and invoke client operations without creating any additional classes.

VisualWorks also provides class WsdlClassBuilder to generate Smalltalk classes. Both WSDL 1.1 and 2.0 documents are supported, and for backward compatibility class WsdlBuilder can generate WSDL from a service class with pragma descriptions for web service operations.

The Web Services framework can also generate a WSDL document from Smalltalk classes that belong to your application. For this, a subclass of WsdlConfigurationDescriptor is used to programmatically create the document from classes. Both WSDL 1.1 and 2.0 are supported. For backward compatibility, WsdlConfigurationDescriptor can also read older-style pragma descriptions for web services operations and types (the API is in class WsdlBuilder).

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

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 document, 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 document (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 document from Smalltalk classes. The wizard also generates workspace code that may be used during testing and construction of your application.

X2O Tool

This tool may be used to build X2O bindings and XML documents from Smalltalk 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	2.0, 1.1 (SOAP via HTTP bindings only)
SOAP	1.2
Basic Profile	1.2
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 document for a remote web service.

1. If you want to send simple client requests via an API, see the discussion of Building Web Service Clients, using the default Web Services client class: `WsdlClient`.
2. If your WSDL document uses complex data types, and you want to use a wizard to generate the binding classes, see: Web Services Wizard.
3. If your WSDL document uses complex data types, and you want to use an API to programmatically generate binding classes, see the discussion of the 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: Creating Responder 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 document using a wizard, see: Building Responders from a WSDL Schema.
4. To create a WSDL document for a Smalltalk service using a wizard, see: Generating WSDL Specifications.
5. If you want to send and receive messages with SOAP headers, see: Sending SOAP Messages with Header Entries.

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.

WSDLTool

A builder for generating classes from a WSDL document and vice versa. For details, refer to: Building Web Service Clients.

WSDLWizard

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

XMLObjectBindingTool

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

XMLObjectBindingWizard

Installs a wizard for creating a X2O specification. For an example illustrating its use, see: XML to Object Binding Tool.

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 Launcher window.

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 four simple applications to illustrate how web service clients interact with servers.

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

Four 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
LoginTimeClient	Uses SOAP header processing when a SOAP header is not described by WSDL

Four responder classes are provided, one for each of these clients: TimeResponder, TimeResponderRPC, and LoginTimeResponder. For details on the usage of TimeResponder and LoginTimeResponder, see the discussion of SOAP Headers.

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 responder and client running in the same VisualWorks image.

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

```
| client value |
TimeResponder addToServer.
client := WebServices.TimeClient new.
"Request the current time from the server"
value := client timeNow.
TimeResponder flushResponders.
```

The result in the variable "value" should be 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 LibraryResponder 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:

```
LibraryResponder addToServer.
```

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 VisualWorks distribution, but may be loaded directly from the Cincom public Store repository.

Chapter

2

Web Services Wizard

Topics

The VisualWorks web services framework includes wizards and builders that can automatically generate Smalltalk classes for use in your application. Given a WSDL document, the WSDL wizard can generate specialized client classes to access the service described by the document. Similarly, given a set of Smalltalk classes, the wizard can be used to generate a WSDL specification document.

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 the default Web Services client — class WsdlClient — as described in Building Web Service Clients.

The WSDL wizard can be used to build a client-side application, and to generate a WSDL document from Smalltalk classes. To prepare an existing Smalltalk application for presentation as a web service (i.e., to build a server application), see: Building Web Services.

Creating Classes from a WSDL Document

Given a WSDL document, 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.

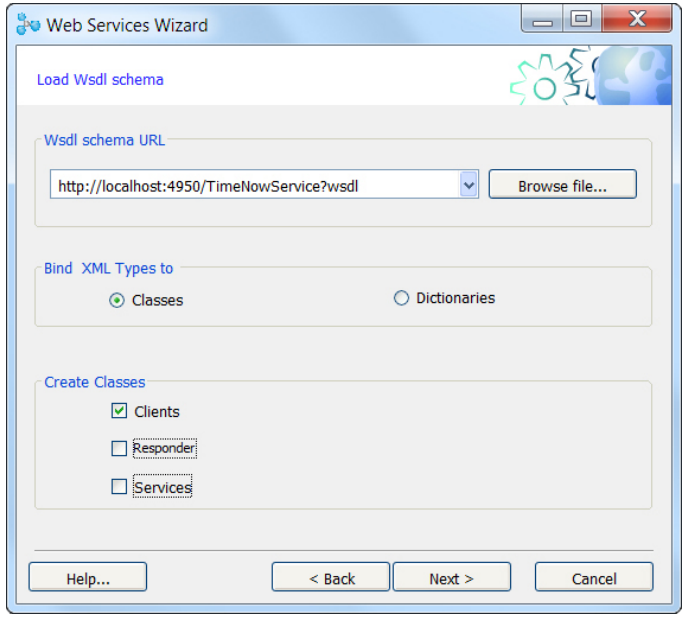
To illustrate, we shall use the WebServicesTimeDemo (for details, see: Web Services Examples). To begin, load the demo and WSDLWizard parcels, and start the time server by evaluating the following code:

```
TimeResponder addToServer.
```

Generating a Web Service Client

The wizard provides a number of options for generating Smalltalk classes and code.

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 document**, and click **Next**.
3. On the next page, specify a document to load.



The demo TimeServer provides a WSDL document at this URL:

```
http://localhost:4444/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):

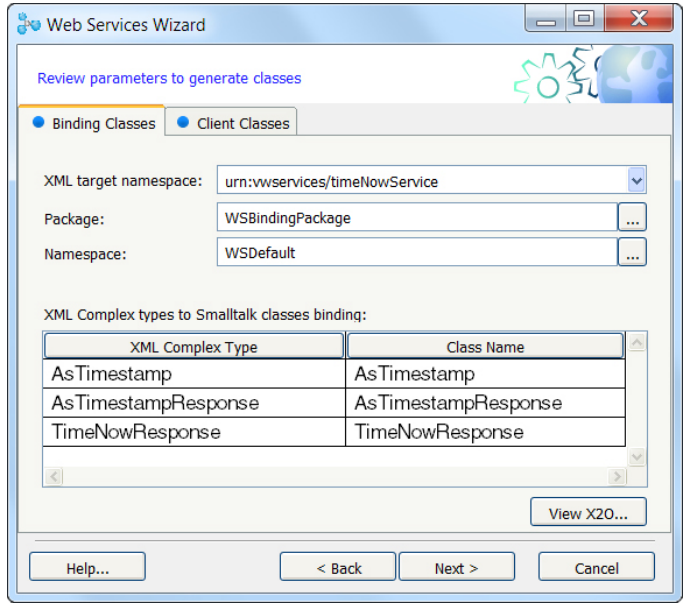
   - **Classes** creates a Smalltalk class for each complex type
   - **Dictionaries** maps each complex type to a Struct class

   Since we are using the WebServicesTimeDemo as the target service in this example, we select **Classes**.

5. The **Create Classes** options tell the wizard which kinds of classes to generate for the client.

   For this example, select **Clients** to generate code for issuing a standard SOAP request, and deselect the **Responder** and **Services** options.

6. Click **Next** to review the parameters for generating the Smalltalk support code.
7. The next page of the wizard has two tabs. The **Binding Classes** tab shows the list of response classes that will be generated, and the options that specify where they will be created.



The **XML target namespace** dropdown list box selects the XML Schema. The XML Schema complex types will be displayed in the right-hand column of the table, below.

Use the **Package** and **Namespace** input fields to specify where the Smalltalk classes will be generated. The left-hand column in the table provides suggested Smalltalk class names. This column is editable.

The **Client Classes** tab shows a list of WS Clients to generate. The default values of the **Package**, **Namespace** and **Client class name** can be changed.

Click **OK**.

8. 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 in the variable value should be a single instance of TimeNowResponse. This is the result passed from the TimeResponder running on your workstation.

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

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

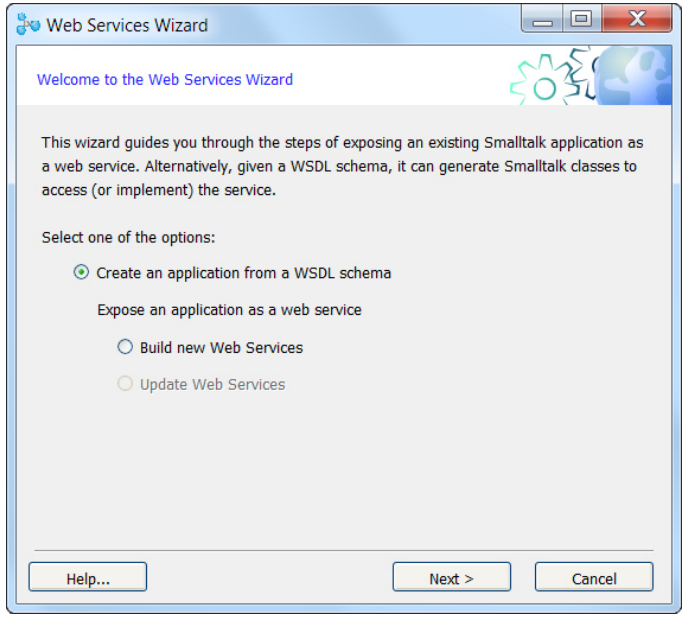
At this point, the wizard has generated a client class that is suitable for use in an application (TimeNowServiceWsdlClient). The client class is located in the package WSServicePackage and the response classes are located in WSBindingPackage, but they may be moved elsewhere.

Generating a Document from Smalltalk Classes

The web services wizard also enables you to create a WSDL document from Smalltalk code that provides a service.

Before creating a WSDL specification, you need to create and register an X2O binding using the X2O Tool (for details, refer to: Generating XML-to-object Bindings).

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



2. Select **Build new Web Services**, and click **Next**.
3. On the **Select WSDL Version** page, select **WSDL 2.0** or **WDSL 1.1** and enter the WSDL **Target Namespace** (e.g., `urn:vwservices`). The latter specifies the `targetNamespace` attribute (in the `<schema>` section of the WSDL document). When finished, press **Next**.
4. On the **Add domain type description** page, you will see the list of all X2O bindings registered in the global registry. Select all bindings that you need for your WSDL specification, and press **Next**.
5. On the **Describe interface** page, you'll need to identify the application and methods to represent in the WSDL document.

   This step involves filling 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 document. 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. For this option

to have an effect, the WSDL Builder settings in the Settings Tool must also be set to **Add the service super class methods** (for details, see Web Services Settings).

6. If your WSDL document specifies any faults or uses SOAP headers, you need to describe them before you begin describing the interface operations.

   To specify faults, use the buttons on the **Faults** tab.

   - Use **Add Fault**... to define a fault.

     The dialog displays only the X2O complex types whose classes are derived from class `Exception`. Select the exception, change its name if you don't like the suggested default, and click **Accept**.

     For WSDL 2.0 the dialog has options to set a fault **Code** and **Subcode**. Select a **SOAP Code** from the dropdown list box and enter the user subcode in the **Subcode** input field.
   - Use **Remove** to remove the selected interface fault from all operations, as well as any references to it.
   - Use **Properties** (WSDL 2.0 only) to display details of the fault. The dialog is the same as that opened by the **Add Fault**... button, and allows resetting the **SOAP Code** and **Subcode**.
   - Use **Add Header**... to add a SOAP header to the selected fault (WSDL 2.0 only).

   To specify headers, use the buttons on the **SOAP Headers** tab.

   - Use **Add Header**... to specify the header type and name. For WSDL 2.0, there are options to set the `#mustUnderstand` and `#require` header attributes.
   - Use **Remove** to remove the selected header from all operations.Use **Add Fault**... (WSDL 1.1 only) to create `<headerfault>` descriptions. The **Select Fault Type** dialog displays complex types derived from class `Exception`. Select the desired fault type, and click **Accept**.
   - Use **Properties** (WSDL 2.0 only) to display details of the fault. The dialog is the same as that opened by the **Add Fault**... button, and allows you to change the `#mustUnderstand` and `#require` header attributes.

7. Next, set the descriptions for the service methods.

   Once the class (and superclass option) has been selected, the methods that will be defined in the WSDL document are listed under the **Operations** tab.

   For each of these methods, select the method name and click **Description**... to open a dialog for specifying the description.

   1. In the **Operation Description** dialog, provide a **Name** and **Description** for the operation.
   2. By default, the operation is described as Request-Response (**In-Out**) and expects that you specify a return value. Selecting the **Robust-In-Only** (i.e., One-Way) option will disable the row containing the return value in the table of message parameters.

      For WSDL 1.1, the One-Way option means that the output message won't be created for this operation. For WSDL 2.0 the operation the interface operation pattern will be set to `http://www.w3.org/ns/wsdl/robust-in-only`. For more information about interface operation MEP, see:

      ```
      http://www.w3.org/TR/2007/REC-wsdl20-adjuncts-20070626/#robust-in-only
      ```

   3. For each parameter in the message, specify a **Name** and **Type** in the table that appears in the lower part of the dialog.

   To describe the operation parameters and return type select a row and right-click to open the <Operate> menu.

   1. Use the menu to specify the **Type**, choosing either **Simple**, **Complex**, **Choice**, **Struct**, or **Collection**.

      Your selections for types will depend on the requirements of the operation. Refer to Using X2O Bindings for help selecting types.
   2. To specify a fault for the operation, select **Add Fault**... from the menu. A sub-menu displays a list of interface faults that were created from the **Faults** tab of the wizard. For WSDL 2.0, an `<outfault>` is added for the operation.

3. Use **Remove Fault** to remove the selected fault from the operation.
4. To add a header to the operation binding input message, select **Add Input Soap Header**, from the menu. A dialog displays a list of headers created in the wizard's **Soap Headers** tab.
5. Use **Add Output Soap Header** to add a header to the operation binding output message.
6. Use **Remove Soap Header** to remove the specified header.
7. When you are finished with the **Operation Description** dialog, **Accept** the description.

The **Operations** tab includes several other buttons:

- Use **Add**... to add a new operation to the WsdlBinding and a method to the service class. You must provide the operation name, select the service class, and enter a new method signature or select an existing one from the **Method signature** dropdown box. If the method is a new, it is opened in the System Browser so you can implement the service.
- Use **Remove**... to remove the selected operation from the service map.
- Use **Input Header**... to add an input header to selected operations. Select a few operations or press Ctrl+A to select all of them, and click on the **Input Header**... button. The dialog displays all SOAP headers. To assign a header to the operation(s), select it and press **OK**.
- Use **Output Header**... to add an output header to the selected operations, follow the same sequence of steps as for **Input Header**....
- Use **Fault**... to add a fault to selected operations.

8. When all method descriptions are complete, click **Next**.
9. If you want to generate a Responder for the service being described, check the **Generate class** option on the **Generate SOAP Responder 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 a client class for the service being described, check the **Generate class** option on the **Generate 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 **Web Services Workspace** page displays a Workspace script you can use to exercise the responder 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** produces only the document, without the `<schemaBindings>` section.
    - **XML to object binding** produces only the `<schemaBindings>` section.
14. In the **Destination** section, select the output destination for the WSDL document:

    - **Method** writes the document to a class method. Provide a method name (`wsdlSchema` by default) and select the class.
    - **File out** writes the document to an external file in plain text format.
    - **POST url** posts the document on the HTTP server using the specified URL.
15. Click **Next** to generate the document.
16. Click **Finish** to close the wizard.

Updating a Binding and its Document

The web services wizard also provides a facility for updating a `WsdlBinding` and its document.

Using the wizard, you can update an older document that uses pragmas, add and remove operations or change their parameter types, and add SOAP headers and exceptions to existing operations.

1. To launch the wizard, select **Tools > Web Services Wizard** in the Visual Launcher.
2. On the first page select **Update Web Services**, and click **Next**.
3. On the **Select to update** page, choose a WSDL binding (e.g., `urn:vwservice/timeNowService#TimeNowService`) and click **Next**.

   At this point, the wizard validates the selected binding, checking its service map for (1) a service class, and (2) that all binding operations have corresponding selectors from the service class.
4. If no service class can be identified, you will be prompted to specify it using a dialog (e.g., `WebServices.TimeNowService`).
5. If any operations lack corresponding selectors, you are prompted with the **Operations with Missing methods** dialog.

   1. To specify a method in this dialog, click on a cell in the **Methods** column, and choose one from the drop-down menu.

      As soon as an operation has been assigned to a method, it will be removed from the dialog.
   2. When all operations have been specified, press **Proceed**.

      At this point, only the operations with selectors assigned from the service class will be updated, and any operations without selectors will be removed from the `WsdlBinding` and WSDL document.

From the **Add Domain Type Description** page, you can proceed from Step 4 in the procedure for Generating a Document from Smalltalk Classes.

Chapter

3

Building Web Service Clients

Topics

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 the default Web Service client (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.

WSDL Support Services

For invoking web services as a client, VisualWorks provides two general options. Either (1) you can use the default class WsdlClient, or (2) you use a specialized subclass of WsdlClient that has been created by the Web Services Tool (for details, see: Creating Classes from a WSDL Document).

The default Web Service client uses the default X2O binding and can send arguments as a Struct object. This client is good for testing purposes. For a production application, you should consider creating a Web Service client specific to a WSDL document; that is, using a specialized subclass.

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

- Loads a WSDL document using a URI, and from schema parts if they are represented by an <import> element in the document, 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 document 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.
- Creates a code script that can be used in an application to make web services requests.
- Allows for a policy to process SOAP headers using interceptors. For details, see the discussion of SOAP Headers.

Loading WSDL Support

To send client requests, load the WSDL parcel (for step-by-step instructions, see: Loading Support for Web Services).

To create specialized clients during application development (i.e. subclasses of WsdlClient), load either the WSDLTool or WSDLWizard parcels.

Using a Web Service Client

Generally, an instance of the client class (WsdlClient or a specialized subclass) is configured for a particular web service, and then used to send requests to a server.

To illustrate with a simple example, we can host both client and server in a single VisualWorks development image. Here, we can query the TimeResponder service in the WebServicesTimeDemo.

To begin, load the demo parcel (see: Loading Support for Web Services), open a Workspace, and evaluate the following code to add the TimeResponder to an existing server or create a new one:

```
TimeResponder addToServer.
```

The server is now running on your workstation. If it is created anew, it will start automatically. When you are finished with the examples that follow (below), don't forget to release the TimeResponder from the server:

```
TimeResponder flushResponders.
```

Now, let's look at the two different approaches for making web service requests.

Tip: 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.

Using a WsdlClient

The most general approach for invoking a web service is to simply use class WsdlClient.

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

```
client := WsdlClient
    url: 'http://localhost:4444/HTimeNowService?wsdl' asURI.
client executeSelector: #TimeNow.
```

The result in the variable `value` should be an instance of `Struct` that contains a `Timestamp`.

In this code example (above), the message `url:` causes the client to load and parse a WSDL document, to create several registries as well as a `WsdlConfiguration`. Then, `executeSelector:` causes the client to actually send a request and get a response.

If you wish to make another request, you don't need to load the WSDL document again. Simply set the port from the `Port` registry:

```
client := WsdlClient new.
client setPortNamed: (XML.NodeTag new
 qualifier: ''
 ns: 'urn:vwservices/timeNowService'
 type: 'TimeNowService' ).
client executeSelector: #TimeNow.
```

Class `WsdlClient` provides two methods for sending requests:

```
executeSelector: aSymbol
 ^self executeSelector: aSymbol args: OrderedCollection new

 executeSelector: aSymbol args: aCollection
 request := self newRequest.
 request smalltalkEntity:
    (Message selector: aSymbol arguments: aCollection).
 ^self executeRequest
```

For document style requests, the second parameter to `#executeSelector:args:` should be a `Collection` with one item. For RPC style, this collection can include more than one item. The web services framework detects RPC calls and will create a `Struct` from the arguments. Upon return, the result is unwrapped from a collection, per the WSDL style. The return value can also be a `Collection`, if RPC output parameters are involved.

Using a Specialized Subclass of WsdlClient

Alternately, you can use a specialized subclass of WsdlClient, which has been created using the Web Services Tool.

In this case, the subclass encapsulates the X2O binding and service maps, and the methods in the subclass' "public api" category encapsulate the services described by the WSDL document. So, using a specialized subclass, you would make a client request like this:

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

Once again, the result in the variable value should be an instance of Timestamp that contains the current time.

Subclasses of WsdlClient implement the following methods:

wsdlSchema

The WSDL document from which the client was created.

x2oBinding

The XML to Object specification that maps XML Complex types to Smalltalk objects.

serviceMap

Maps a specific WSDL interface to a service class that performs web services, and maps interface operations to a service class selectors.

bindingName

The WSDL document's binding element name.

bindingTargetNamespace

The WSDL document's binding target namespace.

processingPolicy

This method is defined only if the WSDL document requires SOAP headers.

When instantiated, subclasses of WsdlClient use these methods to initialize registries. If WSDL and X2O bindings are already registered in the image, the initialization will be skipped.

Creating a Script

Class 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**:

```
client createScript.
```

Or, when you are just using class WsdlClient:

```
(WsdlClient
 url: 'http://localhost:4950/TimeNowService?wsdl' asURI) createScript
```

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.

Authentication

Class WsdlClient and its subclasses support all standard forms of HTTP authentication (e.g., Basic, Digest and NTLM schemas). Instances of the Web Services client class use an HttpClient object as a transport mechanism, and the latter provides the actual support for authentication.

Generally, authentication is set up by sending username:password: to the client object 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 documents. Using these builders, you can create service classes from a WSDL document, or vice versa.

WsdlClassBuilder

Defines Smalltalk classes using the requirements specified in a WSDL document. A document 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.

WsdlConfigurationDescriptor

Generates a WSDL specification document. 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.
- The responder class that implements the code to process web services messages. A responder knows how to register itself with a web server (or create one if there is no server for the specified WSDL address), and how to deregister itself.

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.

Using WsdlClassBuilder

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

```
| builder |
builder := WsdlClassBuilder
 url: 'http://localhost:4444/TimeNowService?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).

The class builder uses WsdlClassBuildOptions to manage all necessary options for creating classes. To specify the options, use message protocol such as:

```
builder clientPackage: 'MyClientPackage".
```

To manipulate the #clientClassMap, you can send #buildOptions, e.g.:

```
builder buildOptions clientClassMap.
```

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

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
 url: 'http://localhost:4444/TimeNowService?wsdl' asURI.
builder package: 'TimeTest'.
builder createClientClasses.
```

You can also set Smalltalk and XML document target name spaces programmatically.

Specifying XML Document Name Spaces

Class WsdlClassBuilder provides methods that allow you to set the Smalltalk name spaces and packages for each XML document target name space. The mappings between them are specified using Dictionary objects.

To set these mappings use the #bindingPackageMap: and #bindingNamespaceMap: methods, as follows:

```
(builder := WsdlClassBuilder new)
 package: 'CustomerServices';
 classNamespace: 'CustomerServices';
 bindingPackageMap:
  (Dictionary new
   at: 'urn:customer' put: 'CustomerPackage';
   at: 'urn:address' put: 'AddressPackage';
   yourself);
 bindingNamespaceMap:
  (Dictionary new
   at: 'urn:customer' put: 'CustomerNS';
   at: 'urn:address' put: 'AddressNS';
   yourself);
 readFrom: wsdl readStream;
 createClientClasses.
```

Here, the domain classes for the target name space 'urn:customer' will be created in the package named CustomerPackage, and in the CustomerNS name space. The domain classes for 'urn:address' target name space will be created in the package named AddressPackage, and in the AddressNS name space.

Specifying Web Services Client and Service Class Names

When you create Web Service client and service classes, the names specified in the <portType> or <interface> elements will be used by default. If you wish to provide different names, you can set the client or service name map, e.g.:

```
builder := WsdlClassBuilder
 readFrom: 'http://server/someservice' asURI
 inNamespace: 'Smalltalk'.

builder buildOptions clientClassMap
 at:'ServiceTest1' put: 'FirstClient';
 at:'ServiceTest2' put: 'SecondClient'.
builder buildOptions serviceClassMap
 at: 'ServiceTest1' put: 'FirstService';
 at: 'ServiceTest2' put: 'SecondService'.
```

In this example 'ServiceTest1' and 'ServiceTest2' are the name attributes of the <portType> elements:

```
<portType name='ServiceTest1'../>
<portType name='ServiceTest2' ../>
```

Loading and Saving a Document

WSDL documents 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 Loading and Using a WSDL Schema.

Overwriting Class Names

When generating classes, it is possible that a class name specified by the document 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.

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.

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 Responders from a WSDL Schema.

Instance creation class methods

readFrom: aDataSource

Create an instance of WsdlClassBuilder and loads the WSDL document 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.

bindingPackageMap: aDictionary

Specify the Smalltalk packages for XML schema target name spaces. Each key in aDictionary indicates an XML schema target

name space, while the values specify the corresponding package names (as strings).

bindingNamespaceMap: aDictionary

Specify the Smalltalk name spaces for XML schema target name spaces. Each key in `aDictionary` indicates an XML schema target name space, while the values specify the corresponding name spaces (as strings).

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.

createResponderClass

Creates the responder class. For details, see: Building Web Services.

Inside the WsdlClassBuilder

The `WsdlClassBuilder` can create the following:

- Schema Bindings
- Binding Classes
- Service Classes

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

Schema Bindings

As described above (Loading and Using a WSDL Schema) when a WSDL document 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 document 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="WSBook">
   <sequence>
    <element name="title" type="xsd:string"/>
    <element
     name="acquisitionNumber"
     type="xsd:positiveInteger"/>
    <element name="statusId" type="xsd:string"/>
   </sequence>
  </complexType>
...
 </schema>
</types>
```

The default Web Services client object maps the complex type WSBook as a <struct>, which will be marshaled as a WebServices.Struct. A 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 WSBook. Based

on the aspect attribute, the instance variable accessors will also be created. For example:

```
title
 ^title
```

```
title: aString
 title := aString
```

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 document from service classes, as described in Building Web Services.

For convenience, you can define the binding classes in a separate package from the client classes, as follows:

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

Service Classes

Service classes represent the web service ports, from a service provider's perspective. They are named using the binding name specified in the <portType> element for WSDL 1.1, or <interface> for WSDL 2.0.

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 include scripts with return type descriptions.

Chapter

4

Document Processing

Topics

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

The default Web Services client, 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.

Working with WSDL Documents

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

Generating XML-to-object Bindings

If a WSDL document declares user-defined types in a <types> section, these are used to create XML-to-object (X2O) bindings. VisualWorks supports two types of binding:

- default, which maps complex XML data type to WebServices.Struct objects
- object, which maps complex XML data type to Smalltalk objects

The default Web Services client (class WsdlClient) uses the default mapping; WsdlClassBuilder and the Wizard give you the option or using either. WsdlConfigurationDescriptor can be invoked directly for these mappings by sending, for the default mappings:

```
WsdlConfigurationDescriptor defaultReadFrom: aDataSource.
```

and for the object mappings:

```
WsdlConfigurationDescriptor
 objectReadFrom: aDataSource
 inNamespace: aString.
```

WsdlConfigurationDescriptor 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:

```
<complexType name="WSBook">
 <sequence>
  <element name="title" type="xsd:string" />
  <element name="acquisitionNumber" type="xsd:positiveInteger" />
  <element name="statusId" type="xsd:string" />
 </sequence>
```

```
</complexType>
```

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

```
<struct name="WSBook">
 <sequence>
  <element name="title" ref="xsd:string" />
  <element name="acquisitionNumber" ref="xsd:positiveInteger" />
  <element name="statusId" ref="xsd:string" />
 </sequence>
</struct>
```

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

The object mapping, for comparison, would be:

```
<object name="WSBook" smalltalkClass="WSBook">
 <sequence>
  <element name="title" ref="xsd:string" />
  <element name="acquisitionNumber" ref="xsd:positiveInteger" />
  <element name="statusId" ref="xsd:string"
      conversionId="ByteSymbol" />
 </sequence>
</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 WSDL document, as shown in the following sections.

While loading the WSDL document, the `XMLObjectBinding` can raise a notification if there are any XML elements that are not resolved.

Saving a WSDL Document with its Bindings

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

```
wsdlClient saveSchemaDocuments
```

This opens a file dialog prompting for a file name (e.g., Library.wsdl), and writes the document 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 document into a file, or:

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

to save the document 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, Library.wsdl1, Library.wsdl12, and so forth.

Loading and Using a WSDL Document

To reuse a saved document, load it from the file, e.g.:

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

or from a stream:

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

While loading a WSDL document, 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 FictionBook. This simply involves adding a smalltalkClass attribute, with the class name enclosed in quotation marks, e.g.:

```
<object name="WSBook" smalltalkClass="FictionBook">
 <sequence>
  <element name="title" ref="xsd:string"/>
  <element name="acquisitionNumber" ref="xsd:positiveInteger"/>
  <element name="statusId" ref="xsd:string"
      conversionId="ByteSymbol"/>
 </sequence>
 </object>
```

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

Generating X2O Specifications without a WSDL Document

Sometimes you don't have a full WSDL document, but only a <types> section, and still want to generate an X2O specification 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 Struct objects or specific Smalltalk classes. The resulting bindings are processed differently to generate the Structs 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:

```
xmlSchema := '<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="WSBook">
   <sequence>
    <element name="title" type="xsd:string" />
    <element name="acquisitionNumber"
      type="xsd:positiveInteger"/>
    <element name="statusId" type="xsd:string" />
   </sequence>
  </complexType>
</schema>'.
```

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:

```
x2oSpec := XMLTypesParser readFrom: xmlSchema readStream.
```

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

```
<schemaBindings>
 <xmlToSmalltalkBinding
  elementFormDefault="qualified"
  targetNamespace="urn:webservices/demo/libraryServices"
  xmlns="urn:visualworks:VWSchemaBinding"
  xmlns:tns="urn:webservices/demo/libraryServices"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <struct name="WSBook">
   <sequence>
```

```
    <element name="title" ref="xsd:string" />
    <element
     name="acquisitionNumber"
     ref="xsd:positiveInteger" />
    <element name="statusId" ref="xsd:string" />
   </sequence>
  </struct>
 </xmlToSmalltalkBinding>
</schemaBindings>
```

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

Creating XML Object Binding Marshalers

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: x2oSpec)) first
```

The result is an XMLObjectBinding defining marshalers for the XML complex type WSBook, with a Struct as the representative Smalltalk class. To browse the binding, open a browser on XMLBindingRegistry, a shared variable of class XMLObjectBinding, and select urn:webservices/demo/libraryServices.

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 WebServices.Struct. You may construct this as follows:

```
book := WebServices.Struct new
 at: #title put: 'Smalltalk Best Practice Patterns';
 at: #acquisitionNumber put: 123456;
 at: #statusId put: 'inStock';
 yourself.
```

To marshal this Struct as a WSBook XML type, create an instance of XMLObjectMarshalingManager on the binding, get the marshaler for the WSBook type, and then marshal the object, e.g.:

```
manager := XMLObjectMarshalingManager on: binding.
marshaler := manager
 marshalerForType: 'WSBook'
 ns: 'urn:webservices/demo/libraryServices'.
xmlres := manager marshal: book with: marshaler.
```

The result is an XML element:

```
<ns:WSBook xmlns:ns="urn:webservices/demo/libraryServices">
 <ns:title>Smalltalk Best Practice Patterns</ns:title>
 <ns:acquisitionNumber>123456</ns:acquisitionNumber>
 <ns:statusId>inStock</ns:statusId>
</ns:WSBook>
```

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

```
wsBook := manager unmarshal: xmlres.
```

which returns the corresponding Struct:

```
{WSBook} {#title->'Smalltalk Best Practice Patterns' #acquisitionNumber->123456
 #statusId->'inStock'}
```

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 Structs, 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: xmlSchema readStream
 inNamespace: 'Smalltalk'.
```

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

```
<schemaBindings>
 <xmlToSmalltalkBinding
  defaultClassNamespace="Smalltalk"
  elementFormDefault="qualified"
  targetNamespace="urn:webservices/demo/libraryServices"
  xmlns="urn:visualworks:VWSchemaBinding"
  xmlns:tns="urn:webservices/demo/libraryServices"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <object name="WSBook" smalltalkClass="WSBook">
   <sequence>
    <element name="title" ref="xsd:string" />
    <element
     name="acquisitionNumber"
     ref="xsd:positiveInteger" />
    <element name="statusId" ref="xsd:string" />
   </sequence>
  </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 getting and setting the values of the classes' attributes.

Some XML documents might include complex types whose names match those of the Smalltalk core classes. For example, the XML complex type `Error`:

```
<xsd:complexType name="Error">
  <xsd:sequence>
     <xsd:element name="description"  type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

By default, the method `XMLTypesParser >> useObjectBindingReadFrom:inNamespace:` creates the following X2O specification:

```
<object name="Error" smalltalkClass="Error">
  <sequence>
     <element name="description" ref="xsd:string"></element>
  </sequence>
</object>
```

To avoid class name conflicts, you can set an option to rename any duplicate class names when an X2O specification is created.

The following code creates an X2O specification with class names that do not duplicate those of existing classes:

```
XMLObjectBindingPolicy class >> useExistingClassName: false.
x2oElement := XMLTypesParser
    useObjectBindingReadFrom: schemaWithErrorType readStream
    inNamespace: 'MyNS'.
```

The `x2oElement` specifies the `Error` type as: `<object name="Error" smalltalkClass="Error0">...</object>`.

```
builder := BindingClassBuilder new.
builder package: 'MyPackage'.
builder createClassesFromBinding: x2oElement elements.
```

The builder creates class Error0.

This option can also be selected using the Settings Tool. On the **Web Services > Class Builder** tab, specify the option to **Use the existing class names**.

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 WSBook, e.g.:

```
book := Smalltalk.WSBook new
 title: 'Smalltalk Best Practice Patterns';
 acquisitionNumber: 123456;
 statusId: 'inStock';
 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: book
   atNamespace: 'urn:webservices/demo/libraryServices'.
```

The result is an XML element representing the object:

```
<ns:WSBook xmlns:ns="urn:webservices/demo/libraryServices">
 <ns:title>Smalltalk Best Practice Patterns</ns:title>
 <ns:acquisitionNumber>123456</ns:acquisitionNumber>
 <ns:statusId>inStock</ns:statusId>
</ns:WSBook>
```

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

```
wsBook := XMLObjectMarshalingManager
 unmarshal: xmlres printString readStream
 atNamespace: 'urn:webservices/demo/libraryServices'
```

which returns the corresponding object.

Chapter

5

SOAP Messages

Topics

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. In historical terms, SOAP may be understood as a successor to XML-RPC.

Whereas WSDL is concerned with service descriptions (which are embodied in WSDL documents), 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.

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 document, making web service development almost trivial.

WSDL Support

VisualWorks supports WSDL 1.1 with SOAP bindings. Two general forms of SOAP messaging are available: RPC and Document style.

WSDL 2.0 is also supported with SOAP bindings. The messages can be encoded in IRI and RPC operation styles.

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 WSDL Support

To load WSDL support, use the Parcel Manager to load the `WSDL` parcel. This will automatically load 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.

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 and 1.2. This supports processing a WSDL document 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 `SoapHeader` instance, with the header entry collection where an item is an instance of `SoapHeaderEntry`. 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

Header entries can be added to SOAP messages by using header interceptors. The VisualWorks implementation uses interceptor classes as a mechanism for adding headers to client requests, and also for servers to validate incoming requests.

To add SOAP headers to messages, both client and responder should provide a processing policy with interceptors.

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

Defining a Client Processing Policy

The client processing policy class can be ClientProcessingPolicy or a class derived from WebServices.ClientProcessingPolicy. The client interceptors classes are derived from class WebServices.ClientMessageInterceptor.

Interceptors can be added in two ways, either using #processingPolicy: or by defining a client class method #procesingPolicy, e.g.:

```
HTimeClient class>>processingPolicy
 ^ClientProcessingPolicy new
  interceptorClasses:
   (OrderedCollection with: HPasswordClientInterceptor);
  yourself
```

Defining a Responder Processing Policy

The responder processing policy class can be SOAPProcessingPolicy or a class derived from it. The responder interceptors are derived from class SOAPMessageInterceptor. To add interceptors, use the #interceptors method, e.g.:

```
HTimeResponder class>>interceptors
 ^OrderedCollection with: HPasswordServerInterceptor
```

Testing the Header Processing Policies

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

```
HTimeResponder addToServer.
client := WebServices.HTimeClient new.
result := client timeNow.
HTimeResponder flushResponders.
```

The result from HTimeClient (stored in the variable result) should be a Time object. You can examine this variable using **Inspect It**.

Behind the Scenes

Let's explore how this works behind the scenes. The HTimeClient class method #processingPolicy specifies the ClientProcessingPolicy and its interceptor: HPasswordClientInterceptor. Similarly, the server responder is created with a processing policy. The responder configuration is created with an HPasswordResponderInterceptor interceptor class. Class HPasswordClientInterceptor implements several callbacks that are invoked at various points during the message processing.

On the client side, the header entry is attached to the SOAP request using the #sendingRequest:in: callback in two classes: HPasswordClientInterceptor and HPasswordResponderInterceptor, e.g.:

```
sendingRequest: aSOAPRequest in: aTransport
 "Client should add the header entry to the request"
 "Add the #password header to the request"
 (aSOAPRequest needsHeader: #password)
   ifTrue: [(aSOAPRequest headerFor: #password) value: 'password'].
```

Here, we attach a header entry to the client request by sending headerFor:, with the name of the new header entry as a symbol. The object to be marshaled (in this case, the password) is assigned to the header by sending value:. On the server side, the message can be validated in the receivingRequestEnvelope:in: callback in class HPasswordResponderInterceptor:

```
receivingRequestEnvelope: aSOAPRequest in: aTransport
 | headerEntry |
 (aSOAPRequest needsHeader: #password)
  ifFalse: [Error
     raiseSignal: 'HTimeServer: The password is required'].
 headerEntry := (aSOAPRequest
       headerAt: #password
       ifAbsent: [^Error raiseSignal: 'HTimeResponder: The request does not include a
 password header']) value.
 headerEntry = 'password'
  ifTrue: [Dialog warn: 'HTimeResponder: The password is accepted']
  ifFalse: [Error raiseSignal: 'HTimeResponder: Incorrect password']
```

Here, the object to be marshaled, is assigned to the header by sending value:.

After evaluating the code shown above, the result from HTimeClient (stored in the variable result) should be a Time object. You can examine this variable using **Inspect It**.

As a second example, let's consider the message exchange between LoginTimeClient and LoginTimeResponder, and look more closely at how we can add a SOAP header using a LoginInterceptor. (The classes in this example are also predefined in the WebServicesTimeDemo package.)

First, note that the Login header is not described in the WSDL document for LoginTimeClient (see: LoginTimeClient class>wsdlSchema). To be able to add this header to a SOAP message, we need to describe its binding specification. A sample is provided in LoginTimeClient class>>x2oLoginSpec. The Login specification describes the header entry like this:

```
<element name="Login" ref="tns:LoginType" />
```

To create an XMLObjectBinding from this specification, evaluate the following code in a Workspace:

```
XMLObjectBinding
 loadFrom: LoginTimeClient x2oLoginSpec readStream.
```

For the purposes of this example, we shall use the predefined client interceptor class LoginClientInterceptor. For your own applications, create client interceptors as subclasses of ClientProcessingPolicy. The responder interceptor classes should be subclassed from SOAPMessageInterceptor.

At execution time, the interceptor has to be specified when the client and responder are created. For examples of how this is done, see the methods: LoginTimeResponder class>>interceptors and LoginTimeClient class>>processingPolicy.

The interceptor callback adds the SOAP header to a client request:

```
sendingRequest: aRequest in: aTransport
 "Client: Using a processing policy create
  and add Soap headers to the request"
```

```
| tag |
tag := NodeTag qualifier: '' ns: 'urn:LoginDescription' type: 'Login'.
 (aRequest headerFor: tag)
  value: (Login new
    username: 'username';
    password: 'password';
    yourself).
```

Since the SOAP header is not described in the WSDL document, the header entry must be created as a `NodeTag` object:

```
tag := NodeTag qualifier: '' ns: 'urn:LoginDescription' type: 'Login'.
```

On the server, `LoginResponderInterceptor>>receivingRequestEnvelope:in:` receives the request and validates the `Login` header information.

To test this example, evaluate this code in a Workspace:

```
LoginTimeResponder addToServer.
client := WebServices.LoginTimeClient new.
(result := client timeNow) inspect.
LoginTimeResponder flushResponders.
```

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

Alternately, if the WSDL document for a service does include a header description, the header can be added to a client request using its name as a `Symbol`.

For example, `HTimeClient class>>wsdlSchema` includes a header description for `password`, and in `HPasswordClientInterceptor>>sendingRequest:in:` the header can be added using `#headerFor:`, e.g.:

```
sendingRequest: aSOAPRequest in: aTransport
 (aSOAPRequest needsHeader: #password)
 ifTrue: [(aSOAPRequest headerFor: #password) value: 'password']
```

If `#headerFor:` receives a string parameter, it creates a header entry with the name as an instance of `NodeTag`, whose namespace is empty. The correct marshaler will be located in the operation's WSDL binding.

Processing Policies and Message Interceptors

The processing policy enables you to customize SOAP message processing. Its main responsibility is to provide a set of *message interceptors* (subclasses of SOAPMessageInterceptor) for each incoming/outgoing message.

Interceptors will receive callbacks when:

- The request has a SOAP body, and the request is about to be sent (client).
- When the requst arrives at the server and the SOAP header blocks are unmarshaled but not the message body.
- When the reply body is marshalled and the reply is about to be sent (server).
- When the reply arrives at the client, and the SOAP header blocks and the body are unmarshalled.

Dedicated interceptors can be instantiated for each message to allow for stateful interception (i.e., carrying state from one interception point to another), or they can be global objects for stateless interception.

Interceptors include the following callbacks:

sendingRequest: aRequest in: aTransport

This method is called on the client side when the request is ready to send. The SOAP headers should be added in this method. The callback will be sent to all operations, even if they don't need headers. It is the application developer's responsibility to provide headers.

receivingRequestEnvelope: aSOAPRequest in: aTransport

Called on the server side when a request arrives. Your application can validate and process the headers here. In the event that validation fails, the application can raise exceptions from this callback.

sendingReply: aReply in: aTransport

Called on the server side when the reply is ready to be sent to a client. Your application should add response or failed headers here.

receivingReply: aReply in: aTransport

Called on the client side when a reply arrives. Your application can process reply headers here and raise exceptions in case of failed messages.

Accessing SOAP Headers from Service Methods

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

```
(server interfaces at: 'localhost:4444')
 server transport environmentWithHeaders: true.
```

The service method can reach the header object like this (see class WSAuthenticatedSearchService):

```
authenticatedSearchByTitle: aString
 | header |
 header := (ProcessEnvironment current
  at: #SoapHeader
  ifAbsent: [self error: 'There is no such header in the
   current ProcessEnvironment'])
  headerAt: 'AuthenticationToken' ifAbsent: [nil].
header := header value.
(header userID = 'UserID' and: [header password = 'password'])
  ifFalse: [^(AuthenticationTokenException new
      userID: header userID;
      password: header password;
      yourself) raise].
```

Using SOAP Header Entries with the Default Web Service Client

When using the default Web Service client (class 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:

```
HTimeResponder addToServer.
client := WsdlClient url:
 'http://localhost:4444/TimeNowServiceWithHeaders?wsdl'.
(client headerFor: #password) value: 'myPassword'.
result := client executeSelector: #TimeNow.
HTimeResponder flushResponders.
```

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:.

Creating a SOAP Header

A SOAP header is represented by a SoapHeader instance, which holds an ordered collection of SoapHeaderEntry instances. Each header entity holds three 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.) If the header entry is described in a X2O binding that is not part of the WSDL, the name has to be provided as a NodeTag where the name space is the X2O target name space.

value

Return the object to marshal.

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.

mustUnderstand: aBoolean

Sets mustUnderstand to aBoolean.

name: aString

Sets the name attribute.

SoapHeaders are held in the header variable of a SoapMessage instance. Use the following SoapHeader 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.

The SoapHeader is registered with the environment as a whole, under the key #SoapHeader.

```
soapHeader := (ProcessEnvironment current
  at: #SoapHeader ifAbsent: [nil])
   headerEntry :=  soapHeader
      headerAt: 'AuthenticationToken'
      ifAbsent: [nil].
```

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.

```
TimeResponder addToServer.
client := WsdlClient url: 'http://localhost:4444/TimeNowService?wsdl'.
value := client executeSelector: #AsTimestamp
    args: (Array with: Time now).
value := client executeSelector: #TimeNow args: Array new.
TimeResponder flushResponders.
```

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:

```
TimeResponder addToServer.
wsdlClient := WsdlClient
    url: 'http://localhost:4444/TimeNowService?wsdl'.

"connect to the HTTP server"
[wsdlClient connectToHost: 'localhost' port: 4444]
 on: Exception
 do: [:ex | wsdlClient reconnect. ex return].

"execute a few requests and close the HTTP connection"
[value := wsdlClient executeSelector: #AsTimestamp
      args: (Array with: Time now).
value := wsdlClient executeSelector: #TimeNow
     args: Array new]
 ensure: [wsdlClient close].
TimeResponder flushResponders.
```

SOAP Exception Handling

These exception classes are specific to the implementation of SOAP 1.1 and 1.2:

```
SoapFault
 Soap11Fault
  SoapClientFault
  SoapServerFault
  SoapMustUnderstand
 Soap12Fault
  DataEncodingUnknown
  MustUnderstandFault
  ReceiverFault
  SenderFault
  VersionMismatchFault
```

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:.

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 or ReceiverFault usually indicate that the request was malformed or lacked needed information to process the request. SoapServerFault or SenderFault indicate that processing failed for reasons not directly attributable to the request, such as an upstream server failing to respond.

In WSDL 1.1, SoapFault is raised if a version mismatch (invalid namespace) or a “must understand” requirement was not obeyed by the processor.

In WSDL 2.0:

- VersionMismatchFault is raised if the faulting node found an invalid element information item instead of the expected Envelope element information item.
- MustUnderstandFault is raised if an immediate child element information item of the SOAP Header element information

item targeted at the faulting node, that was not understood by the faulting node contained a SOAP `mustUnderstand` attribute information item with a value of "`true`".

- `DataEncodingUnknownFault` is raised if a SOAP header block or SOAP body child element information item targeted at the faulting SOAP node is scoped (see 5.1.1 on the SOAP `encodingStyle` attribute), with a data encoding that the faulting node does not support.

Note that SOAP and WSDL communications can suffer any number of other general communication 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 application as general exceptions.

Chapter

6

Building Web Services

Topics

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 server applications. The wizards and class builders included in the VisualWorks framework can also help simplify this task.

Server Implementation Overview

The server implementation for Web Services (class SOAPResponder) provides support for processing multiple SOAP requests and replies, as is needed in distributed environments. The responder uses SiouX server classes, which implement the server side of HTTP 1.1.

You can also use the Web Services Tool to create custom responder classes, which inherit the behavior of class SOAPResponder. Each responder is created for a specific port and path. For details, see: Creating Responder Classes using the Web Services Wizard.

Loading Server Support

The SOAP responder classes are in the WebServicesServer package. 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 **Web Services** category, in the list on the left-hand side of the tool.
3. In the upper-right list, select and load **WebServicesServer** (select the **Parcel > Load** menu command).

Working with SOAP Responders

Class SOAPResponder parses a WSDL document, creates, and registers WSDL and X20 bindings. For each <soap:address location= ... > element in a WSDL document, VisualWorks creates an instance of SOAPResponder. Responder instances are cached in the class variable #responders. Each SOAPResponder instance knows its WSDL binding, service class, processing policy and URL path.

Since the responder instances are cached, any changes to the WSDL document require flushing the #responders collection. Evaluating SOAPResponder flushResponders removes the responders from servers and empties the responder cache. To reinitialize the collection, use SOAPResponder responders.

To activate the responder, it must be added to an existing server, or a new server will be created:

```
TimeResponder addToServer.
```

When the #addToServer method is called for the first time, the WSDL document is parsed and an instance of TimeResponder is created and cached. If a suitable server (e.g., serving the specified port) doesn't yet exist. this method will create and start a new one.

To deactivate a responder, send #release to it. You can also remove the responder from the server and flush the class-side responder cache as follows:

```
TimeResponder flushResponders.
```

Note that releasing the responder does not otherwise affect the server that was hosting it. If you wish to deactivate the server, that has to be done explicitly.

Building Responders from a WSDL Document

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

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

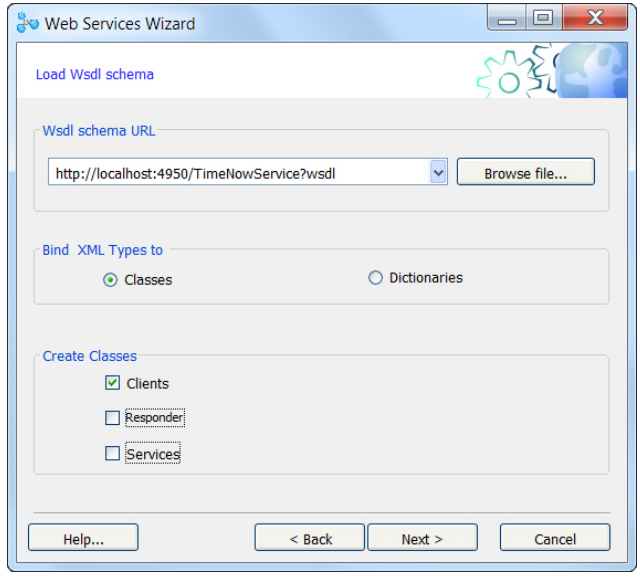
Creating Responder Classes using the Web Services Wizard

Given a WSDL document, 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 of the document, 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 document, open a Browser, locate class TimeClient in the WebServicesTimeDemo package, 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 document to load:



For example, specify the document URL:

```
http://localhost:4444/TimeNowService?wsdl
```

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

In general, you can enter a URL in the **WSDL schema URL** field (by 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** directs the wizard to create a Smalltalk class for each complex type
   - **Dictionaries** directs the wizard to map each complex type to a Dictionary

6. In the **Create Classes** section of the wizard, select the **Responder** and **Services** option.

   The **Services** option creates a stub class and methods to provide the implementation for the service.

7. Click **Next** to generate the Smalltalk support code.
8. A dialog appears 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 includes a workspace with Smalltalk code to exercise the newly-generated service classes. However, before we can use these new classes, we must add implementation code.

   Using the document for the TimeServer, the wizard has created the following classes in the WSServerPackage package: TimeNowService (the service class), TimeNowResponder (the server-side class), and TimeNowServiceClient (the client class).

   The service class TimeNowService includes stub methods for the operations defined in the document.

10. 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.

11. 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
 ^self
 "Add implementation here"
 "Result object:
 TimeNowResponse new
  result: ('Time');
  yourself"
```

    This method contains a suggestion about the implementation code.

12. To add the implementation code, edit the method as follows:

```
timeNow
 ^TimeNowResponse new
  result: Time now;
  yourself
```

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

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

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

14. In the workspace, start the server by evaluating:

```
TimeNowResponder addToServer.
```

15. To test the server, evaluate the following using **Inspect It**:

```
client := TimeNowServiceClient new.
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.

16. To deregister the responder from the server and flush any cached responders, evaluate this code (using **Do It**):

```
TimeNowResponder flushResponders.
```

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

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

Class SOAPResponder

The Web Services wizard can be used to create specialized subclasses of SOAPResponder. Subclasses must implement following class methods:

wsdlSchema

The WSDL document may be returned either as a string or a URL. If the WSDL document includes <import> statements, #wsdlSchema should return a URI. This can be used as a base URI to resolve import locations.

x2oBinding

XML-to-Object binding specification. The Web Services tools generate this method from the WSDL <schema> elements.

serviceMap

The Web Services tools generate this method from <portType> elements in the WSDL document. The service map provides information about a services class, its selectors and corresponding web service interfaces and operations.

interceptors

The Web Services tools generate this method and interceptor classes when WSDL document has SOAP headers. These classes provide additional interception points between SOAP header and body marshaling and unmarshaling. For an example, see class HTimeServer in the WebServicesTimeDemo.

wsdlQueryFor

Sets WSDL document that will be returned in response to a ?WSDL query

Creating a Web Services Responder from a WSDL document

A responder class includes all of the methods necessary to load bindings, to add itself to a server, and to process SOAP messages.

To see the Web Services responder generation example, load the WebServicesDemo parcel and browse the methods test11CreateClientResponderClasses or test20CreateClientResponderClasses in class TestCreateWSApplication.

The responder class is generated with the class methods: #wsdlSchema (for the WSDL specification), #x2oBinding, and #serviceMap (this method maps service class methods to interface operations).

Creating Service Classes using the WsdlClassBuilder

The service classes are generated by WsdlClassBuilder from the WSDL specification operations. The service classes names are created from the <portType> element names for WSDL 1.1, or <interface> names for WSDL 2.0.

The service class methods that are derived from the operations go into the public api protocol. 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.

For RPC style, in WSLDSrvcGeneralPublicRpc class:

```
holdingByAcquisitionNumber: aLDHolding_acquisitionNumber
```

```
providesServices
```

For Document style, in WSLDSrvcGeneralPublicDoc class:

```
holdingByAcquisitionNumberDoc: aHoldingByAcquisitionNumber
```

```
providesServices
```

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 object.

Generating WSDL Specifications

WSDL specifications can be generated from the information found in WsdlConfigurationDescriptor classes that provides a web service interface description.

Note: In release 7.8, class WsdlBuilder has been marked as a deprecated API. It is still functional, but its implementation has been delegated to a subclass of WsdlConfigurationDescriptor.

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

1. Describe user domain types for the service description using a X2O specification (i.e., <schemaBindings>).

   The types can be described using the X2O Tool (for details, see XML to Object Binding Tool).
2. Describe the operation of the interface, its parameters and result types. The service provider class must have descriptions for all methods that are to be represented as external interfaces.
3. Describe operation exceptions or SOAP headers, if any. Their types should be also described in one of the <xmlToSmalltalkBinding> elements.
4. Finally, provide descriptions for access points, a port description, and a SOAP address.

These steps are explained at length in an example, below.

Backward Compatibility with Operation Pragmas

Prior to version 7.8, the VisualWorks Web Service framework used method pragmas to describe operations. For backward compatibility, class WsdlConfiguration in version 7.8 can still read the old operation description pragmas. However, it neither creates new nor updates existing pragmas.

To create a WSDL 1.1 specification from service class pragmas:

```
descriptor := Wsdl11ConfigurationDescriptor new.
descriptor
 buildFromServices:
  (OrderedCollection with: WSSearchServices)
    classNamespace: 'WebServices'
    targetNamespace: 'urn:test'.
```

WSDL Specifications and Name Spaces

To hold operation parameter types the descriptor creates a new <xmlToSmalltalkBinding> binding with a #targetNamespace attribute as its WSDL targetNamespace. This attribute has a _WSDLOperations suffix. Later, if you need to change your WSDL specification operations, the descriptor will use the target name space with this suffix to find the operation type descriptions for updating the specification.

For example, creating a WSDL specification with urn:test as its name space creates an X2O binding for operation parameters with the target name space urn:test_WSDLOperations.

WSDL 1.1 Example

A WSDL 1.1 specification may be generated programmatically using an instance of WsdI11ConfigurationDescriptor.

1. To build a WSDL specification for the search service in the Library Demo:

```
descriptor := WsdI11ConfigurationDescriptor new.
descriptor targetNamespace: 'urn:LibraryDemo\testing'.
```

   By default the specification will be created in document/literal style. To set different options, send #buildOptions. For details, see Specifying WSDL 1.1 Descriptor Options.
2. Before describing interface operations you need to apply an X2O specification that includes domain types, exception and SOAP header descriptions represented as XMLObjectBinding objects. E.g.:

```
descriptor readImportedX2OBindingFrom: self x2OSpec readStream.
```

   For a working example of this XML specification, see: TestCreateWSDLSchema>>x2OSpec in the WebServicesDemo package.
3. Create an interface description for the service class using #addInterfaceForServiceClass:.

```
searchServicesInterface := descriptor
  addInterfaceForServiceClass: WSSearchServices.
```

   This method takes a service class as a parameter and returns an instance of WsdlInterfaceDescriptor.
4. Describe the operation of the interface:

```
(interfaceOperation := descriptor
 addInterfaceOperationNamed: 'SearchByTitle'
 toInterface: searchServicesInterface)
  selector: #searchByExactTitle:includeAffiliatedLibraries:;
```

```
 documentation: 'The SearchByTitle operation returns a collection of holdings or
empty collection if no holdings found'.
```

Here, the interface operation descriptor is created by `#addInterfaceOperationNamed:toInterface:`, which takes two parameters: a `String` that names the interface operation, and an interface descriptor (an instance of `Wsdl11InterfaceOperationDescriptor`). This method returns an interface operation descriptor.

5. Describe the operation's parameters and return types:

```
(parameters := descriptor newParameters)
 addParameterNamed: 'searchByExactTitle' type: String;
 addParameterNamed: 'includeAffiliatedLibraries' type: Boolean.
```

```
descriptor
 addInputParameters: parameters
 toOperation: interfaceOperation.
```

For this, use `#newParameters`, which returns an instance of `OperationParameters`. For details on working with this class, see: Specifying Operation Parameters.

6. Describe the return type.

Here, we'll specify a collection of `Protocols.Library.Book` as the return type:

```
parameters := descriptor newParameters.
parameters addReturnCollectionType: Protocols.Library.Book.
descriptor
 addOutputParameters: parameters
 toOperation: interfaceOperation.
```

7. Describe the interface operation faults, and add them to the interface operation:

```
descriptor
 addFaultNamed: 'holdingNotFound'
 type: Protocols.Library.HoldingNotFound
 toOperation: interfaceOperation.
```

8. Create a default binding descriptor for the interface descriptor:

```
searchServicesBinding := descriptor
 buildDefaultBindingDescriptorFor: interface.
```

9. Create SOAP headers, and add them to operation binding descriptors:

```
authTokenHeader := descriptor
 createSoapHeaderNamed: 'AuthenticationToken'
 type: AuthenticationToken.
```

10. Create and add faults to the header:

```
headerFault := descriptor
 createSoapHeaderFaultNamed: 'AuthenticationTokenException'
 type: AuthenticationTokenException.
authTokenHeader addFault: headerFault.
headerFault := descriptor
 createSoapHeaderFaultNamed: 'WrongPasswordException'
 type: WrongPasswordException.
authTokenHeader addFault: headerFault.
```

11. Find the operation binding descriptor, and add the input header:

```
bindingOperationDescriptor := interfaceBinding
 operationNamed: 'SearchByTitle'
 ifNone: [nil].
bindingOperationDescriptor addInputSoapHeader: authTokenHeader.
```

12. Set parameters to create endpoints for the service class:

```
service := descriptor addServiceNamed: 'TestLibraryServices'.
```

13. Create port description, and set the SOAP address for the service:

```
service addPort: (descriptor
  createEndpointNamed: 'WSSearchServices'
  address: 'http://localhost:3331/testSearchServices'
  binding: searchServicesBinding).
```

14. When you finish describing the WSDL specification, you must evaluate the following expression:

```
descriptor complete.
```

    The method #complete validates the descriptor, and then registers the XMLObjectBinding for operation requests, responses, and WsdlBinding.

15. To view the descriptor as a WSDL document:

```
descriptor asWsdl inspect.
```

Specifying WSDL 1.1 Descriptor Options

Class Wsdl11BuildOptions provides a simple API for specifying descriptor options:

soapStyle: aSymbol

Specify: #document or #rpc

soapUse: aSymbol

Specify: #literal or #encoded

Customizing WSDL 1.1 Part Names

For WSDL 1.1 specifications in document style, you can customize the <part> element's name attribute in input and output messages.

By default, the Web services tool generates WSDL 1.1 messages like this:

```
<message name="foobarSoapIn">
 <part name="parameter" element="tns:foobar" />
</message>
<message name="foobarSoapOut">
 <part name="return" element="tns:foobarResponse" />
</message>
```

Here, the input part name is "parameter" and the output part name is "return".

To change these default names in generated schemas, open the Settings Tool from the Visual Launcher, and select **Web Services > Wsdl Builder > WSDL 1.1 Binding**. On this page, you can change the deafults for **Input part name** and **Output part name**. Click **OK** to accept your modifications.

These settings can also be changed programmatically, using class Wsdl11BuildOptions, e.g.:

```
Wsdl11BuildOptions inputPartName: 'parameters'.
Wsdl11BuildOptions outputPartName: 'return'.
```

Setting Operation Parameters in WSDL 1.1

When generating a WSDL 1.1 document in document style using class Wsdl11ConfigurationDescriptor, you may optionally set nillable attributes for the operation parameters. For example, to create a WSDL specification for a service named CustomerServices:

```
x2o := '<?xml version="1.0" encoding="utf-8" ?>
<schemaBindings>
 <xmlToSmalltalkBinding targetNamespace="urn:CustomerDescription"
 defaultClassNamespace="WebServices" xmlns="urn:visualworks:VWSchemaBinding"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="urn:CustomerDescription">
  <object name="Customer" smalltalkClass="Customer">
   <element name="name" ref="xsd:string" />
   <element name="id" ref="xsd:int" />
  </object>
 </xmlToSmalltalkBinding>
</schemaBindings>'.
```

```
descriptor := Wsdl11ConfigurationDescriptor new.
descriptor targetNamespace: 'urn:testNillable'.
descriptor readImportedX2OBindingFrom: x2o readStream.
interface := descriptor
    addInterfaceNamed: 'CustomerService'
    serviceClass: CustomerService.
(interfaceOperation := descriptor
        addInterfaceOperationNamed:
         'SetCustomer'
        toInterface: interface) selector: #setCustomer:.
```

Create the input parameter descriptor for the operation #SetCustomer:

```
parameterDescriptor := (descriptor
        addInputParameterType:
         WebServices.Customer
        toOperation: interfaceOperation) first.
```

Set the input parameter to be nillable:

```
(descriptor
 documentStyleOperationParametersFor: parameterDescriptor)
  do: [:marshaler | marshaler beNillable].
```

Create the result descriptor for the operation #SetCustomer:

```
parameterDescriptor := descriptor
        addOutputParameterType:
         WebServices.Customer
        toOperation: interfaceOperation.
```

Set the output parameter to be not nillable (this is the default):

```
(descriptor
 documentStyleOperationParametersFor: parameterDescriptor)
  do: [:marshaler | marshaler dontBeNillable].
```

Create a binding for the interface:

```
interfaceBinding := descriptor
        buildDefaultBindingDescriptorFor: interface.
```

Create a <service> component:

```
service := descriptor addServiceNamed: 'CustomerServices'.
service
 addPort:
  (descriptor
   createEndpointNamed: 'CustomerEndpoint'
   address: 'http://localhost:3333/customerServices'
   binding: interfaceBinding).
```

Validate and complete the descriptor:

```
descriptor complete.
```

Create a WSDL specification:

```
descriptor asWsdl inspect.
```

If a WSDL specification was already built by the Web Services Wizard, it is also possible to find the operation parameter marshalers and reset the nillable options.

For example, after building the WSDL document with target namespace: ‘urn:testNillable’, the operation binding is registered with a target namespace 'urn:w110_WSDLOperations'. This holds the operation marshalers. To change the input operation marshaler parameters to be nillable:

```
binding := XMLObjectBinding bindingAtNamespace:
 'urn:testNillable_WSDLOperations'.
binding marshalers
 do: [:operationMarshaler |
  "Find the operation"
  operationMarshaler tag type = 'SetCustomer'
   ifTrue: [operationMarshaler typeMarshaler elementMarshalers
      do: [:parameterMarshaler |
       "Change the nillable default to true"
       parameterMarshaler beNillable]]].
wsdlBinding := (WsdlBinding
       bindingsAtNamespace: 'urn:testNillable') first.
wsdlBinding asWsdlSchema inspect.
```

WSDL 2.0 Example

A WSDL 2.0 specification may be generated programmatically using an instance of Wsdl20ConfigurationDescriptor.

1. To build a WSDL specification for the search service in the Library Demo:

```
descriptor := Wsdl20ConfigurationDescriptor new.
descriptor targetNamespace: 'urn:LibraryDemo\testSoapHeaders'.
```

By default the specification will be created in IRI style. To set different options, send `#buildOptions`. For details, see Specifying WDSL 2.0 Descriptor Options.

2. Before describing interface operations you need to apply an X2O specification that includes domain types, exception and SOAP header descriptions represented as `XMLObjectBinding` objects. E.g.:

```
descriptor readImportedX2OBindingFrom: self x2OSpec readStream.
```

3. Create the interface descriptor:

```
interface := descriptor
 addInterfaceForServiceClass: WSAuthenticatedSearchService.
```

4. Add a fault description to the interface:

```
descriptor
 addFaultNamed: 'AuthenticationTokenException'
 type: AuthenticationTokenException
 toInterface: interface.
```

5. Add operation descriptors to the interface.

   Here, we add details about the `AuthenticatedSearchByTitle` operation:

```
(interfaceOperation := descriptor
 addInterfaceOperationNamed: 'AuthenticatedSearchByTitle'
 toInterface: interface)
  selector: #authenticatedSearchByTitle:;
  documentation: 'The AuthenticatedSearchByTitle operation returns a collection
 of holdings or an empty collection if no holdings are found'.
```

6. Describe the input parameters:

```
descriptor
 addInputParameterType: String
 toOperation: interfaceOperation.
```

7. Describe the return type:

```
books := descriptor newParameters.
books addReturnCollectionType: Protocols.Library.Book.
descriptor
```

```
 addOutputParameters: books
 toOperation: interfaceOperation.
```

8. Describe the faults associated with this operation:

```
descriptor
 addOutfaultNamed: 'AuthenticationTokenException'
 toOperation: interfaceOperation.
```

9. Create the default binding descriptor for the interface.

   This descriptor will include a binding description for all interface operations and faults.

```
interfaceBinding := descriptor
 buildDefaultBindingDescriptorFor: interface.
```

10. Create a descriptor for the fault header:

```
header := descriptor
 createSoapHeaderNamed: 'AuthenticationTokenExceptionHeader'
 type: AuthenticationTokenExceptionHeader.
```

    Here, the message `#createSoapHeaderNamed:type:`returns an instance of `Wsdl20SoapHeaderDescriptor` with `mustUnderstand` and `required` both set to `true` by default.

11. Create a SOAP fault:

    To create a SOAP fault, its type must be described in the interface descriptor.

```
(soapFault := descriptor
 addSoapFaultNamed: 'AuthenticationTokenException'
 toBinding: interfaceBinding)
  addSoapFaultCodeNamed: 'Sender';
  addSoapFaultSubcodeNamed: 'WrongAuthenticationToken'.
soapFault addSoapHeader: header.
```

    The method `#addSoapFaultNamed:toBinding:` returns an instance of `Wsdl20SoapFaultDescriptor` with a default SOAP fault code of `#any`, and fault subcode of `#any`.

#addSoapFaultCodeNamed: accepts one of the following code types: #VersionMismatch, #MustUnderstand, #DataEncodingUnknown, #Sender, #Receiver, #any.

SOAP fault codes are described in detail here:

```
http://www.w3.org/TR/2007/REC-soap12-part1-20070427/#faultcodes
```

The method #addSoapFaultSubcodeNamed: accepts any customer code in the form of a String.

In the WSDL specification, the binding fault will be described as:

```
<fault ref="tns:AuthenticationTokenException"
 wsoap:code="SOAP-ENV:Sender"
 wsoap:subcodes="tns:WrongAuthenticationToken">
  <wsoap:header
   element="ns:AuthenticationTokenExceptionHeader"
   mustUnderstand="true"
   required="true"/>
</fault>
```

12. Create and add SOAP headers to the operation (e.g., AuthenticationToken):

```
authTokenHeader := descriptor
 createSoapHeaderNamed: 'AuthenticationToken'
 type: WebServices.AuthenticationToken.
operation := interfaceBinding
 operationNamed: 'AuthenticatedSearchByTitle'
 ifNone: [nil].
operation addInputSoapHeader: authTokenHeader.
```

Again, the message #createSoapHeaderNamed:type: returns an instance of Wsdl20SoapHeaderDescriptor with mustUnderstand and required both set to true by default. The methods addInputSoapHeader: and addOutputSoapHeader: both expect an instance of Wsdl20SoapHeaderDescriptor as a parameter.

13. Set parameters to create end points for the service class:

```
service := descriptor
 addServiceNamed: 'TestLibraryServicesWithHeaders'
```

```
 interface: interface.
service addPort: (descriptor
 createEndpointNamed: 'WSAuthenticatedSearchService'
 address: 'http://localhost:4450/testSoapHeadersServices'
 binding: interfaceBinding ).
```

14. Compile and validate the descriptor:

```
descriptor complete.
```

15. Inspect the WSDL specification:

```
descriptor asWsdl inspect.
```

The WDSL service description should be:

```
<service
 name="TestLibraryServicesWithHeaders"
 interface="tns:WSAuthenticatedSearchService">
 <endpoint
  name="WSAuthenticatedSearchService"
  binding="tns:WSAuthenticatedSearchService"
  address="http://localhost:4450/testSoapHeadersServices"/>
</service>
```

Specifying WDSL 2.0 Descriptor Options

For WDSL 2.0, use `Wsdl20ConfigurationDescriptor`. To provide the descriptor options, use its instance method `#buildOptions`, which returns an instance of class `Wsdl20BuildOptions`.

Class `Wsdl20BuildOptions` provides the following API:

bindingType: aString

By default, this is: `http://www.w3.org/ns/wsdl/soap`

httpVerb: aString

By default: `'POST'`

mep: aString

By default: `http://www.w3.org/ns/wsdl/in-out`

soapMep: aString

By default: http://www.w3.org/2003/05/soap/mep/request-response/

styleDefault: aString

By default: http://www.w3.org/ns/wsdl/style/iri

transportProtocol: aString

By default: http://www.w3.org/2003/05/soap/bindings/HTTP/

Specifying Operation Parameters

Class OperationParameters provides the following API:

addParameterNamed: aString type: aClass

Add the specified parameter as a simple or complex type. aString is the parameter name, and aClass is the type described by #x2OSpec. The simple type classes are listed in XMLObjectBinding class>>simpleTypes. For complex types, the aClass parameter must be fully a qualified class name, e.g.: Protocols.Library.Book.

addParameterNamed: aString collectionType: aClass

Add the specified parameter as a collection, where aClass is the type of the collection item.

addReturnType: aClass

Specify a return type.

addReturnCollectionType: aClass

Describes a return type as a collection with the aClass type.

Printing a WSDL Specification

Class WsdlConfigurationDescriptor includes the following protocol for writing WSDL specifications to a Stream object:

printSpecOn: aStream

Print the WSDL specification on aStream.

printX2OSpecificationOn: aStream

Print the X2O specification on aStream.

printXMLSchemaSpecificationOn: aStream

Print the XML schema on aStream.

Mapping SOAP Operations to Smalltalk Messages

When the Web Service wizard creates client and service classes it also creates a #serviceMap class-side method for them. This method provides a map for service class methods and WSDL interface operations.

For example, here is a map for WebServices.TimeNowService:

```
serviceMap
 ^'<ns:serviceMap
  xmlns:ns="urn:visualworks:serviceMap"
  xmlns:ns0="urn:vwservices/timeNowServiceRPC">
 <ns:serviceClass>WebServices.TimeNowService</ns:serviceClass>
 <ns:interface>ns0:TimeNowService</ns:interface>
 <ns:operation name="TimeNow" selector="timeNow" />
 <ns:operation name="AsTimestamp" selector="asTimestamp:" />
</ns:serviceMap>'
```

This map links:

- The service class WebServices.TimeNowService to the WSDL interface ns0:TimeNowService.
- The Smalltalk message #timeNow to the WSDL interface operation TimeNow.
- The Smalltalk message #asTimestamp: to the WSDL interface operation AsTimestamp.

Without this service map method, 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 this selector is not defined in the WSDL operation binding, then,

1. the operation name is transformed using the algorithm in OperationBinding>>equivalentSmalltalkMessageName,

2. the target object 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. When WsdlBindings are built automatically from publicly available X2O specifications, the `#serviceMap` will be created automatically.

Exceptions and SOAP Faults

In general, any exceptions sent to the client are sent as `SoapFaults`. `SoapFault` instances can be generated by the server 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`. The client 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.

HTTP Transport Extensions

HTTP is the only transport currently supported for web services in the VisualWorks implementation. This is provided by class `HTTPTransport`, which provides standard HTTP request handling.

HTTPTransport

The SiouX framework provides the basic HTTP server infrastructure for Web Services. SiouX allows that multiple instances of a server can coexist and be active in the same image, as long as they listen on unique ports. The server instances are managed through a shared variable Server.Registry.

When a server instance is created, it is automatically registered with an ID, that is either explicitly specified or auto-generated. There can be only one server with given ID at a time. Server activity is controlled with the #start and #stop messages. When a server instance is to be discarded, it must be sent the #release message to remove it from the registry.

A server has one or more listeners, each listening for incoming connections. Once established, a connection waits for incoming requests, which are then passed to the server for execution. A server is configured with one or more responders. For each incoming request, the server asks its responders to handle it, one by one in their registered order. A responder can reject a request, in which case the next responder is given the opportunity. This iteration ends with the first responder that handles the request.

Class Responder is the superclass for all responders. Its main responsibility is to evaluate requests and selectively produce replies. The responder class verifies the request URL in in the method: #dispatchRequest:method:version:connection:.

Servers and Responders

Instances of class SOAPResponder are responsible for processing a SOAP message for a specified host, port and WSDL binding. The responder accepts an instance of Net.HttpRequest using the method #dispatchRequest:method:version:connection: and checks if there is a registered service for the specified URL. The service is registered by a key created from the WSDL binding port address and path. If the responder accepts the request, it returns an HttpResponse object with the marshaled SOAP response. If there is no corresponding service, the responder returns nil and the server passes the HttpRequest to the

next responder. Each SOAPResponder can define its own processing policy.

By dispatching to several different responders, the web server can serve not only web service requests but other types of requests too (e.g. Seaside).

SOAP Responder Example

The following example illustrates how to set up a Time SOAP responder.

First, load the WSDL document and register its bindings:

```
binding := (WsdlConfiguration
    defaultReadFrom: TimeClient wsdlSchema readStream)
     bindings first.
```

Specify the TimeNow service class:

```
binding serviceClass: TimeNowService.
```

Create SOAP responder for TimeNow service:

```
url := binding portAddress.
soapResponder := SOAPResponder new.
soapResponder
 host: url host;
 port: url port;
 addBinding: binding.
```

Create the TimeNow server:

```
server := SiouX.Server id: 'Experiments'. server
addResponder: soapResponder;
listenOn: 8000 for: SiouX.HttpConnection;
start.
```

Once started, the server is ready to respond to client requests, e.g.:

```
'http://localhost:8000/test' asURI get.
```

To stop the server and remove it from the global registry:

```
server release.
```

Chapter

7

XML to Object Binding Tool

Topics

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

The XML-to-Object Binding tool 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.

XML-to-Object Bindings

XML-to-Object (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 document for a given application, the types in an X2O binding element appear in the <types> element of the document.

XML-to-Object bindings can be created from:

- An existing XML Schema, using class XMLTypesParser, or from Smalltalk classes, using the X2O Tool.

X2O Binding Registry

All X2O specification objects (instances of XMLObjectBinding) are held in a special registry, located in your development image (the shared variable XMLObjectBinding.XMLBindingRegistry). Typically, the X2O tool is used to maintain this registry.

An instance of XMLObjectBinding creates X2O binding and XML Schema specifications using the following API:

#asX2OSpecification

Returns a <xmlToSmalltalkBinding> element with the X2O specification.

#asXMLSchemaSpecification

Returns a <schema> element with an XML Schema specification.

#asSchemaBindingsSpecification

Returns a <schemaBindings> element with a <xmlToSmalltalkBinding> sub-element.

Using the XML-to-Object Binding Tool

The XML to Object Binding tool helps to create X2O binding and XML schema specifications for your domain classes. Specifically, the tool simplifies the task of assigning classes as types for the operation parameters that will be passed to service classes.

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

The tool maintains all X2O specification objects in the global X2O binding registry, located in your development image.

This tool enables you to create new X2O specification objects, or modify existing ones in the registry. You can export XML representations of these objects to files, post them to a server, and automatically create scripts for marshaling and unmarshaling domain objects.

An Example Application

To illustrate the use of the tool, 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 are classes that belong to the Library Demo:

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

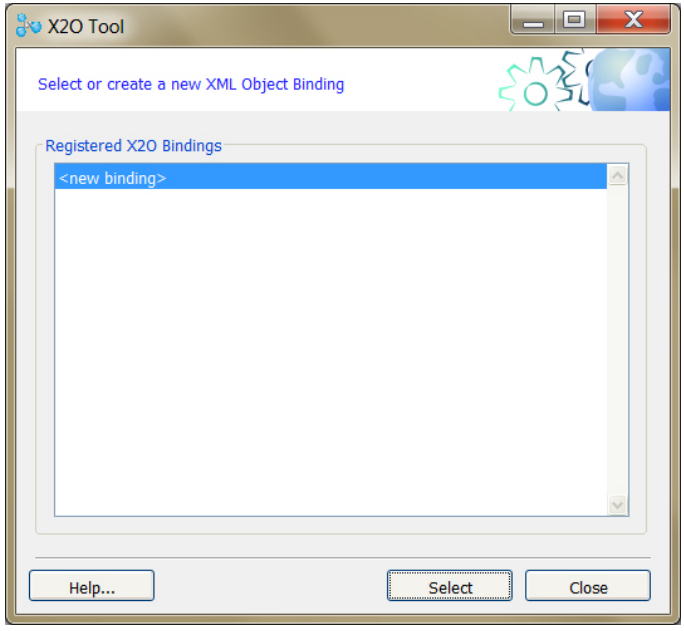
Creating an XML to Object Binding

To illustrate the use of the X2O Binding tool, the following steps show how to create a binding description for the sample class Customer, as defined previously.

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

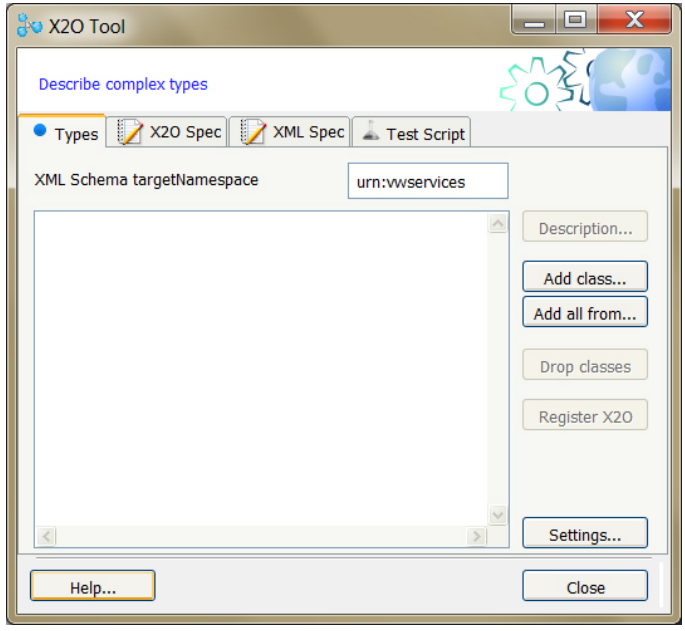
1. To launch the tool, select **X2O Tool** from the **Tools** menu in the Visual Launcher.

   Use the first page of the tool to either create a new binding description, or to modify an existing one.



Since we are creating a new binding, choose **<new binding>**, and click on **Select**:

2. The **Describe Complex Types** page of the tool provides a tabbed view of a new binding description. Initially it is empty, and we shall describe its elements by building up a list of classes.



3. To begin, set the **XML Schema targetNamespace** attribute.

   For this example, use **urn:CustomerSchema**.

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

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

   If the selected class is defined in a name space other than `Smalltalk.*`, the dialog **Select Default Class Name space** prompts you to specify a default class name space. This name space will be added to the X2O binding as a `defaultClassNamespace` attribute, so that later complex marshalers can later resolve the `smalltalkClass` attribute by using this default name space.

   If the selected class includes type pragmas that were generated by previous versions of the X2O tool, these pragmas are used to create type marshalers for the class instance variables.

   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**....

Note: If a class is missing all type descriptions, a blue, empty circle appears next to its name. If the class has partially-

resolved types, a half-circle will appear. A class can have descriptions for some, all, or none of the instance variables, but in general all should be defined to produce a correct X2O specification.

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

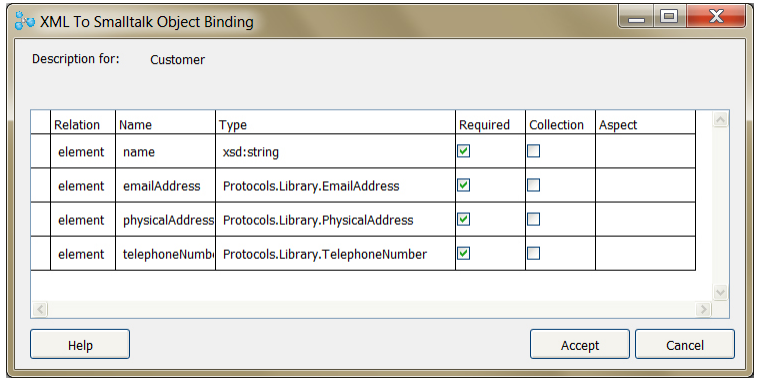
   In the **Description** dialog, use the <Operate> menu in the **Type** field to set the following values:

   1. For **name**, select **Simple Types > xsd:string**.

      The menu lists XML Schema Datatypes (for details, see: http://www.w3.org/TR/xmlschema-2/).

      The order of the types is defined in the X2O specification: XMLObjectBinding class>>defaultXsdBindingMap.
   2. For **emailAddress**, select **Complex Type**..., and enter **Protocols.Library.EmailAddress** (start by entering the name of the class, e.g. **EmailAddress**).
   3. For **physicalAddress**, select **Complex Type**..., and enter **Protocols.Library.PhysicalAddress**.
   4. For **telephoneNumbers**, select the **Collection** check-box, and in the **Type** column, select **Complex Type**..., and enter **Protocols.Library.TelephoneNumber**.

   The completed description should appear as follows:



   When you have finished specifying all the types, click **Accept**.

If the class doesn't have accessor methods, they are created by the tool. The names of the accessor methods are used in the X2O binding's aspect attribute.

Repeat this step as necessary, selecting a class and filling out its description, until all classes have been described.

6. At this point, the following list of classes appears in the X2O Tool:

   AbstractRandom

   Customer

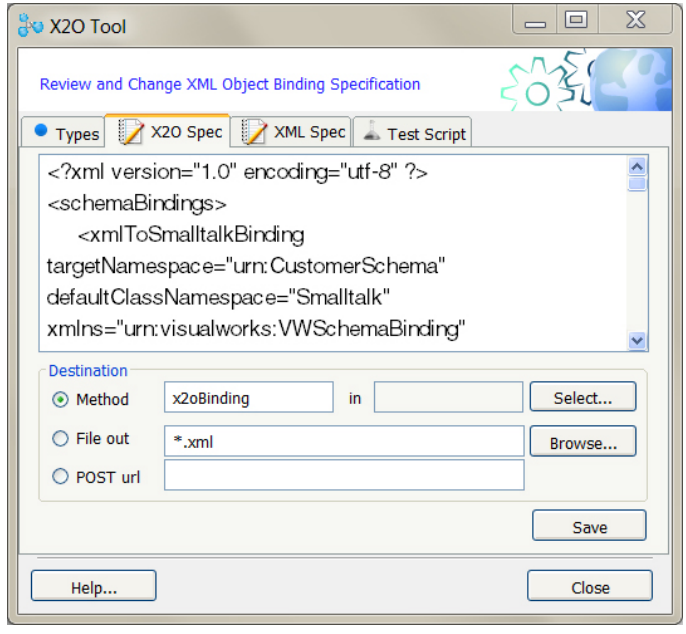
   EmailAddress

   PhysicalAddress

   TelephoneNumber

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

7. When all classes are described, you can review the X2O specification by clicking on **X2O Spec** tab in the Tool, and the XML Schema specification by clicking on the **XML Spec** tab.



Note that you can also edit the XML that appears in the X2O tab. If you modify the specification, the X2O Tool will ask if you wish to accept the changes. Answer **Yes** to modify the binding description.

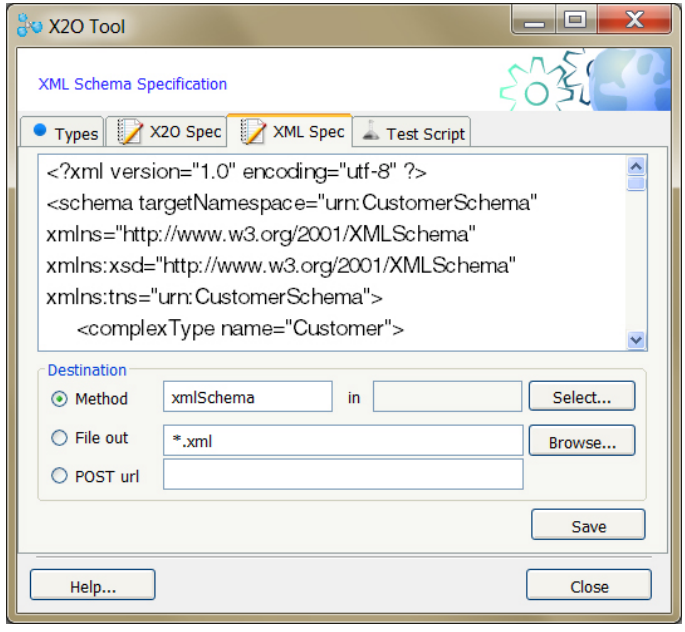
Note that you cannot modify XML Schema specification using the X2O Tool.

8. When the specification has been finalized, click **Register X2O**.

   This loads and validates the schema, and then registers it in the XMLObjectBinding registry. You can view the registry Dictionary by evaluating this code:

```
WebServices.XMLObjectBinding.XMLBindingRegistry inspect.
```

9. On the **XML Schema Specification** page (click on the **XML Spec** tab), you can save the XML Schema to a particular destination.



To save the schema, first 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.

An xmlSchema method can be referenced as an external resource for a WSDL specification (i.e., in an <import> element).

10. When the destination has been specified, click **Save**.
11. The **Testing XML To Object Binding** page (click on the **Test Script** tab), provides a script to test the newly created X2O bindings. Before running a script, the X2O binding must be registered.

    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**.

Upgrading Binding Descriptions

With VisualWorks 7.8, X2O binding descriptions are represented as instances of class XMLObjectBinding, stored in the registry XMLObjectBinding.XMLBindingRegistry.

In previous releases, the X2O specifications were represented using pragmas attached to methods in the domain classes.

For backward compatibility, the X2O tools in version 7.8 can still read these pragmas. If you add a domain class in the X2O tool that includes type pragmas, they will be read and used to create marshalers. However, if you change the classes or any of the type descriptions, the X2O tool will no longer write or update the pragmas.

In this way, you can easily upgrade an existing application to use the new form of X2O specification objects stored in the registry.

Chapter

8

XML to Smalltalk Mapping

Topics

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 support this use of XML in VisualWorks, a mechanism is required for mapping XML elements and attributes to Smalltalk objects, and back again. VisualWorks includes a XML-to-Object (X2O) marshaling mechanism that performs this mapping, and is the foundation for the rest of VisualWorks web service support.

Marshaling Objects to XML

An X2O specification object is generated using a *binding specification*, which is a XML document written with a special syntax. Before describing this syntax, let's look at how the VisualWorks implementation performs XML to Smalltalk mapping, and some of the options that are available to application developers.

To map between XML and Smalltalk, the VisualWorks implementation begins with the binding specification document. Using this specification, class `BindingBuilder` creates marshalers that are stored in a global registry. The registry maps these marshalers to the tag names in the binding specification document. The marshalers are subsequently used to marshal Smalltalk objects to XML documents and unmarshal XML documents to Smalltalk objects.

The X2O mechanism supports marshaling for:

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

The X2O framework supports the common XML primitive types, including strings, numbers, token lists, URI references, name space-qualified names and name references, etc.

For example, a Smalltalk `String` object can be mapped to a `xsd:string` XML type, and XML types with attributes can be mapped to Smalltalk objects, where each attribute is mapped to an instance variable.

X2O Marshaling Example

A simple example can illustrate how to marshal Smalltalk domain objects to XML.

Given two domain classes `Customer` and `Address`, the following code may be used to marshal their instances (for convenience, these classes are already defined in the `WebServicesDemoModels` package).

First, evaluate the following code to create a X2O specification object from a XML binding specification document:

```
x2OSpec := '<?xml version="1.0" encoding="utf-8" ?>
<schemaBindings>
 <xmlToSmalltalkBinding
  targetNamespace="urn:customer"
  defaultClassNamespace="WebServices"
  xmlns="urn:visualworks:VWSchemaBinding"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:tns="urn:customer">
  <object name="Customer" smalltalkClass="Customer">
   <sequence>
    <element name="address" ref="tns:Address" />
    <element name="name" ref="xsd:string" />
    <element name="id" ref="xsd:int" />
   </sequence>
  </object>
  <object name="Address" smalltalkClass="Address">
   <sequence>
    <element name="state" ref="xsd:string" />
    <element name="street" ref="xsd:string" />
    <element name="zip" ref="xsd:int" />
   </sequence>
  </object>
 </xmlToSmalltalkBinding>
</schemaBindings>'.
x2oBinding := XMLObjectBinding loadFrom: x2OSpec readStream.
```

Next, create a domain object, an instance of `Customer`. Note that both classes are defined in the `WebServices.*` name space:

```
customer := WebServices.Customer new
    name: 'John Doe';
    address: (WebServices.Address new
        state: 'OH';
        street: '55 Merchant street';
        zip: 45245;
        yourself);
    id: 123456789;
    yourself.
```

Lastly, marshal the customer object into XML, and then unmarshall it into another Smalltalk object:

```
xml := x2oBinding marshal: customer.
unmarshaledObject := x2oBinding unmarshal:
        (XML.Document new addNode: xml) root.
```

The unmarshaledObject and customer objects should be equivalent.

Marshaling Dictionaries with Dynamic Keys

Typically, a Dictionary object can be described in a X2O binding specification using <struct> elements, where the dictionary keys are element tags.

For example, let's create X2O binding specification for a simple dictionary:

```
aDictionary := Dictionary new  at: #firstKey put: 'value1';  at: #secondKey put: 'value2';
  yourself.
```

Here, the X2O binding specification can be:

```
<struct name="Dictionary">
 <element name="firstKey" ref="xsd:string" />
 <element name="secondKey" ref="xsd:string" />
</struct>
```

Describing dictionaries with dynamic keys may require representing them as a collection of associations. For example, we can create a DictionaryHolder class, whose single instance variable #dictionary is a dictionary with dynamic keys.

```
Smalltalk defineClass: #DictionaryHolder
 superclass: #{Core.Object}
 indexedType: #none
 private: false
 instanceVariableNames: 'dictionary'
 classInstanceVariableNames: ''
 imports: ''
 category: ''
```

We need to add methods to this class that return a collection of associations from #dictionary and likewise set the dictionary from a collection of associations. We will use these methods in the X2O specification.

```
getDict
 ^dictionary associations
```

```
setDict: aColl
 dictionary := Dictionary new.
 aColl do: [:assn |  dictionary at: assn key put: assn value].
```

Now we can describe the object in a X2O binding specification:

```
x2o := '<?xml version="1.0" encoding="utf-8" ?>
<schemaBindings>
 <xmlToSmalltalkBinding
  targetNamespace="urn:DictionaryHolder"
  defaultClassNamespace="Smalltalk"
  xmlns="urn:visualworks:VWSchemaBinding"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <object name="DictionaryHolder" smalltalkClass="DictionaryHolder">
  <sequence>
   <element name="dictionary"
    setSelector="setDict:"
    getSelector="getDict" maxOccurs="unbounded">
    <object name="dictionary" smalltalkClass="Association">
     <sequence>
      <element name="key" ref="xsd:string" />
      <element name="value" ref="xsd:string" />
     </sequence>
    </object>
   </element>
  </sequence>
 </object>
 </xmlToSmalltalkBinding>
</schemaBindings>'.
```

Notice that the #dictionary element is represented by an Association, where the #key and #value elements can hold any string. The #dictionary element has special accessor methods to convert a dictionary to a

Collection of associations and to create a dictionary from a Collection of associations.

To test marshaling:

```
binding := WebServices.XMLObjectBinding loadFrom: x2o readStream.
object := DictionaryHolder new
    dictionary: (Dictionary new
        at: #firstKey  put: 'value1';
        at:  #secondKey put: 'value2';
        yourself );
    yourself.
xml := binding marshal: object.
```

This code example creates:

```
<ns:DictionaryHolder xmlns:ns="urn:DictionaryHolder">
 <dictionary>
  <key>secondKey</key>
  <value>value2</value>
 </dictionary>
 <dictionary>
  <key>firstKey</key>
  <value>value1</value>
 </dictionary>
</ns:DictionaryHolder>
```

To test unmarshaling:

```
dictionaryHolder := binding unmarshal: xml.
dictionaryHolder dictionary isKindOf: Dictionary.
```

Marshaling Options

In addition to support for marshaling simple and complex objects, the X2O mechanism also supports some important options for object marshaling. This functionality is useful when you wish to marshal objects for which no specific domain class has yet been defined (e.g. Collection or Dictionary objects).

For instance, XML can be marshaled either as an object with aspects or as a Struct. That is, it can be marshaled to a Dictionary, handling aspects using at: and at:put: messages. Collections can also

be marshaled in this way. By using a Struct, you can marshal objects for which no specific domain class has yet been defined, permitting you to start developing your application with an intermediate implementation that you can gradually refine later.

Mapping with Aspects

In a X2O specification, *aspects* represent accessors to instance variables. For example, this mapping uses the aspect attribute to set default accessors to the instance variable name as #name and #name:

```
<element name="name" aspect="name" ref="xsd:int" />
```

Aspects can be defined easily. The VisualWorks implementation supports both scalar aspects and repeating groups (collection-valued aspects).

To illustrate how aspects are used to map between XML and a X2O specification, let's say we have a XML schema for a complex type Customer:

```
<xsd:complexType name="Customer">
 <xsd:sequence>
  <xsd:element name="name" type="xsd:string" />
 </xsd:sequence>
</xsd:complexType>
```

This type would be described in an X2O specification like this:

```
<object name="Customer" smalltalkClass="Customer">
 <sequence>
  <element name="name" aspect="name" ref="xsd:int" />
 </sequence>
</object>
```

The corresponding Smalltalk class for Customer would be defined as:

```
Smalltalk.WebServices defineClass: #Customer
 superclass: #{Core.Object}
 instanceVariableNames: 'name '
```

This class is expected to have accessor methods #name and #name: which correspond to the XML attribute #aspect.

Alternately, the XML complex type Customer can be mapped to a Struct:

```
<struct name="Customer">
 <sequence>
  <element name="name" aspect="name" ref="xsd:int" />
 </sequence>
</struct>
```

When a Struct object is marshaled its aspects are handled using the at: and at:put: messages rather than accessor methods.

A Struct object may be created like this:

```
WebServices.Struct new
 at: #name put: 'Winston Smith';
 yourself.
```

Core Framework Classes

The primary marshaling classes are implemented as subclasses of XMLTypeMarshaler:

```
Object
 BindingBuilder
 XMLMarshaler
  BindingImport
  XMLObjectBinding
   SoapBinding
   WsdlBinding
  XMLTypeMarshaler
   CompositorMarshaler
    SequentialMarshaler
    UnorderedMarshaler
   HrefMarshaler
   ObjectMarshaler
    CollectionObjectMarshaler
    ComplexObjectMarshaler
     ModelGroupMarshaler
      ModelAttributeGroupMarshaler
    StructMarshaler
```

```
KeyObjectMarshaler
 KeyRefObjectMarshaler
RelationMarshaler
AnyRelationMarshaler
 AnyAttributeMarshaler
 AnyCollectionMarshaler
AttributeMarshaler
ChoiceMarshaler
ElementMarshaler
GroupMarshaler
 AttributeGroupMarshaler
ImplicitMarshaler
 ImplicitAttributeMarshaler
TextMarshaler
RestrictionMarshaler
SimpleTypeMarshaler
 ListMarshaler
 SimpleObjectMarshaler
 UnionMarshaler
```

Most of these classes are marshalers for various XML types, and are described in a separate topic (see: XML Marshalers).

`BindingBuilder` is responsible for generating bindings from X2O specification documents, maintaining a registry of mappings from XML tags to marshaler classes, and blocks for resolving primitive types.

`XMLObjectBinding` maintains a separate registry of bindings for specific schemas. An instance of `XMLObjectBinding` holds marshalers for a specific target name space. It responds to requests to find a marshaler for a specific XML tag or Smalltalk object.

Using X2O Bindings

XML-to-object translation is bidirectional. Given a Smalltalk object to be sent to a web service, the object must be marshaled as a XML data element. And, given a XML element, it must be unmarshaled to an 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.

In the VisualWorks implemention, this document is referred to as a *binding specification*.

Using this specification document, you create a set of X2O specification objects that tell the XML-to-object engine how to map the XML content to Smalltalk objects. Each X2O specification object determines two items: the marshaler for an element or attribute, and the Smalltalk class to represent the item.

Binding Specifications

Each binding specification object is an instance of `XMLObjectBinding`, and has a unique target name space (URI) as well as an optional name (tag). The main purpose of this class is to serve requests to its marshaler according to its name. The bindings are held in a registry that exists as a shared variable in class `XMLObjectBinding`. They can be fetched from the registry by referencing their target name space.

A binding specification is itself a XML document that is constructed using element tags stored in the `BindingBuilder.Registry` dictionary, a shared (class) variable. This registry associates element tags with marshaler classes. A number of default marshalers are defined for common data types (e.g. '`object`' and '`struct`', '`sequence_of`', '`element`', '`attribute`', '`text`', etc.), and you can create additional marshalers (for details, see: XML Marshalers).

For example, here are a few of the default entries in the `BindingBuilder.Registry` dictionary:

<object>

For an `<object>` element, the builder creates an instance of `ComplexObjectMarshaler` with a XPath expression of '`self::`'.

<element>

For an `<element>` node, the builder creates an instance of `ElementMarshaler` with a XPath expression of '`child::`'.

<attribute>

For an `<attribute>` node, the builder creates an instance of `AttributeMarshaler` with a XPath expression of '`@`'.

<enumeration>, <length>

For an <enumeration> or <length> element, the builder creates an instance of RestrictionMarshaler.

These are not only preconfigured, but implicitly create XPath expressions based on the values of other attributes.

When the schema is being parsed to create binding specification objects, class BindingBuilder creates new instances of marshalers using the #newProxyFor: method. The method parameter is the node tag registered in the BindingBuilder.Registry (e.g. 'object'). In this fashion, it is possible to register several different nodes with the same marshaler.

For example, consider this simple binding specification:

```
schema := '<schemaBindings>
 <xmlToSmalltalkBinding name="Generator"
 xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
 targetNamespace="urn:test">
  <sequence_of name="numbers">
   <element ref="xsd:float" />
  </sequence_of>
  <struct name="Document">
   <attribute name="id" ref="xsd:integer" />
  </struct>
 </xmlToSmalltalkBinding>
</schemaBindings>'.
```

Here, there are two elements, introduced by the <sequence_of> and <struct> tags. The binding registry contains predefined entries for each of these, mapping the tags to CollectionObjectMarshaler and StructMarshaler, respectively. Class BindingBuilder reads the binding specification, and creates an instance of the marshaler registered under the tag. Only tag types are compared, ignoring name spaces.

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

Tag type	Marshaler class
any	AnyRelationMarshaler
anyAttribute	AnyAttributeMarshaler

Tag type	Marshaler class
attribute	AttributeMarshaler
bindingImport	BindingImport
choice	ChoiceMarshaler
element	ElementMarshaler
group	GroupMarshaler
implicit	ImplicitMarshaler
key	KeyObjectMarshaler
keyRef	KeyRefObjectMarshaler
object	ComplexObjectMarshaler
sequence_of	CollectionObjectMarshaler
simple	SimpleObjectMarshaler
struct	StructMarshaler
text	TextMarshaler
union	UnionMarshaler
sequence	SequentialMarshaler
all	UnorderedMarshaler

At the implementation level, all registered marshalers are described using class-side methods in XMLObjectBinding. These specify the X2O bindings for creating X2O and XML Schema specifications:

xml_SchemaSpecification

This schema describes the mappings for marshalers to create an XML schema specification from an instance of XMLObjectBinding.

xml_BindingSpecification

This schema describes the mappings for marshalers to create an X2O specification from an instance of XMLObjectBinding.

Creating a X2O Specification

X2O specification objects are created by parsing a XML binding specification document. The relationships between the bindings in this document and Smalltalk classes are specified using the smalltalkClass attribute for each <object> element.

For details on the syntax of the binding specification document, see: Binding Schema Syntax.

Before you load a binding specification document, all referenced Smalltalk classes need to be present in the image. Alternately, you can use the BindingClassBuilder in the XMLObjectBindingTool to create X2O object specifications and to create Smalltalk classes for each <object> element (for details, see: Creating the Binding Classes).

The general steps to create a X2O specification (an instance of XMLObjectBinding), are as follows:

1. Create the X2O specification from a XML Schema using the XMLTypeParser.
    1. To create a default X2O specification:

```
x2oSpec := XMLTypesParser
 readFrom: xmlSchemaString readStream.
```

    2. Alternately, to create an <object> binding specification:

```
x2oSpec := XMLTypesParser
 useObjectBindingReadFrom: xmlSchemaString readStream
 inNamespace: 'Smalltalk'.
```

2. Review the X2O specification, and save it.
3. Load the X2O specification, which will create and register an instance of XMLObjectBinding.

```
x2oBinding := XMLObjectBinding loadFrom: x2oSpecString readStream.
```

    While creating marshalers from X2O specifications, class BindingBuilder can raise an UnresolvedReferenceSignal if there are any unresolvable types in the specification. If this occurs, you must fix the specification and repeat step 3.

Class BindingBuilder creates marshalers for each X2O specification element based on the X2O element tag.

To test the new X2O specifications, you can marshal Smalltalk objects to an XML document. E.g.:

```
customer := WebServices.Customer new
  name: 'Smith';
  yourself.
xmlElement := x2oBinding marshal: customer.
```

or:

```
XMLObjectBinding
 marshal: customer
 atNamespace: 'x2oTargetNamespace'.
```

Conversely, to unmarshal the xmlDocument object:

```
customer := x2oBinding unmarshal:
     (XML.Document new addNode: xmlElement) root.
```

The registry for holding the marshalers is in the shared (class) variable BindingBuilder.Registry (for details, see: BindingBuilder class>>initializePrototypeMarshalers). The registry is a dictionary that maps tags to marshaler class names.

Binding Schema Syntax

To create a X2O specification, generally you begin with a XML binding specification document. This document uses a special syntax, and may be an external file or the return value of a method (for an example of the latter, see the method: XMLObjectBinding class>>defaultSoapBindingLocation).

The binding specification document uses the basic datatype structures of XML Schema, with some additional element and attribute definitions that are needed to support X2O mapping.

For details on XML Schema and its syntax, see:

```
http://www.w3.org/TR/xmlschema-0/
```

The binding specification document begins with the usual prefix information, identifying the XML version:

```
<?xml version ="1.0"?>
```

The body of the document is an `xmlToSmalltalkBinding` element (or some other X2O binding):

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

Name spaces are declared in this tag as shown, including the name space for your service's schema and any others, including the XML Schema name space. The reference to XML Schema (`xmlns:xsd`) is required. This is used to reference simple types, and to encode or decode them using XML Schema datatypes.

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

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

Here, the value "`SoapBinding`" specifies the name of the imported binding (in `XMLObjectBinding.BindingRegistry`). The bindingImport element can also include information about the name space of the imported schema, e.g.:

```
<bindingImport namespace="http://schemas.xmlsoap.org/soap/envelope/" />
```

The rest of the document, before the closing `</xmlToSmalltalkBinding>` tag, consists of specifications for individual bindings. The following examples elaborate the binding specifications for the most common object types.

Simple Objects

Simple objects are objects that have an opaque structure, and include common primitive data types, such as String or Float:

```
<simple name="string" id="String" />
<simple name="float" id="Float" />
```

In the binding named float, the XML simple type (xsd:float) will be encoded and decoded using the Smalltalk Float class.

Simple objects are identified by their conversion ID and are resolved using serialization/deserialization blocks that are defined for several Smalltalk classes (for details, see: Mashaling XML <simpleType> Elements). The value comes from or is put into the element's character data.

You can also define your own simple types by applying restrictions, e.g.:

```
<s:element name="Gender" type="s0:Gender" />
<!-- User-defined simple type -->
<s:simpleType name="Gender">
 <s:restriction base="s:string">
  <s:enumeration value="Male" />
  <s:enumeration value="Female" />
 </s:restriction>
</s:simpleType>
```

The bindings for simple types are defined in XMLObjectBinding class>>defaultXsdBindingMap. These mappings name the Smalltalk classes used to encode or decode the data. The encoders and decoders are defined in BindingBuilder class>>initializeSerializationBlocks where the key is the id attribute from the binding specification.

While loading the XMLToObjectBinding class, the following 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 VisualWorks provides support only for the "enumeration" facet, which constrains the value space to a specified set of values.

For details on enumeration constraints, see:

http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/#dt-enumeration.

Complex Objects

Complex objects are objects with relations. These objects are described by a type and set of relations with other objects. Relations can be implemented in a number of different ways, and are a generalization of notions such as attributes, aspects, members of a collection, etc. A complex object knows how to get/set the 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>
```

For details on complex types, see:

http://www.w3.org/TR/xmlschema-1/#Complex_Type_Definitions.

Attributes

Complex objects may contain *attributes*, which are themselves always simple types such as string, integer, date, or boolean, e.g.:

```
<attribute name="ID" type="xsd:string" />
```

Attributes may also declare a default value, to be used when no value has been specified, or a fixed value, to be used at all times, e.g.:

```
<attribute name="lang" type="xsd:string" default="EN" />
```

By default, the VisualWorks implementation maps attribute definitions to X2O binding specifications, using the ref attribute as a type resolver. That is, the ID attribute shown above is mapped to the X2O specification as:

```
<attribute name="ID" ref="xsd:string" />
```

For historical reasons and for backward compatibility, VisualWorks handles attributes in the schema that use ref in a different fashion. For example, if the schema declares attributes like this:

```
<attribute name="backgroundColor" type="int" />
```

```
<attribute ref="color" />
```

They will be represented in the binding specification like this:

```
<attribute name="backgroundColor" ref="int" />
```

```
<implicitAttribute ref="color" />
```

Note that the <implicitAttribute ... > tag is specific to the VisualWorks implementation, and not used in XML schema. Attributes that use ref are marshaled/unmarshaled using an ImplicitAttributeMarshaler.

Registering a Binding

Before you can marshal or unmarshal objects, you need to create an instance of XMLObjectBinding and register it. To parse and register

the bindings in a binding specification document, send a loadFrom: message to the XMLObjectBinding class. For example, if the specification were defined in the myBindingSpec class method in your application, the minimal code would be:

```
WebServices.XMLObjectBinding
 loadFrom: self myBindingSpec readStream.
```

After evaluating #loadFrom:, the binding is added to the XMLObjectBinding.BindingRegistry, and is ready for use.

Each time your application uses #loadFrom:, the VisualWorks implementation will update its registry. Depending on the size of your X2O specifications, it may be time consuming to create marshalers too often. To avoid this overhead, you can optimize your code to only load new X2O specifications.

For example, to probe for a X2O specification that is registered with #targetNamespace as 'urn:visualworks:operationSelectors', and then to load it if there is no such binding:

```
WebServices.XMLObjectBinding
 bindingAtNamespace: 'urn:visualworks:operationSelectors'
 ifAbsent: [WebServices.XMLObjectBinding
     loadFrom: SelectorMap x2oBinding readStream].
```

For an example of #loadFrom: with condition checking, see the XMLObjectBinding class method #loadVWBinding and related methods.

XML Marshalers

Depending upon the requirements of your application, it may be necessary to create new marshaler classes. For this, you need to implement the following behavior.

Marshalers convert XML elements and/or their parts to Smalltalk objects, and vice versa. Each marshaler matches an XML node or collection of nodes. Therefore, each marshaler has an associated XPath expression and XPath parser for this expression. Since a XPath is a kind of object in itself, there is a bridge between the 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. They must include an initialize method where additional attributes are set up, and must support the following protocols:

setNodeAndMarshalFrom: marshalingContext

Marshal a Smalltalk object as XML.

acceptNodesAndUnmarshal: marshalingContext do: aBlock

Accept the marshaling context nodes that match the marshaler tag, unmarshal selected nodes as a Smalltalk object, and remove the accepted nodes from marshaling context nodes. The tag corresponds to an XML element (e.g., <customer ...>), and is defined in the tag instance variable of class XMLTypeMarshaler.

decodeAndUnmarshalFrom: marshalingContext

Using a XPath expression, collect nodes and unmarshal them as Smalltalk objects.

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.

Compositor Marshalers

These marshalers hold element marshalers in ComplexObjectMarshalers and define how the XML nodes are processed while unmarshaling.

X2O Bindings

These are high-level repositories of binding info. They are usually root elements in the binding specification. An X2O binding contains the list of all element marshalers and can

find a marshaler for a specific XML tag or Smalltalk object. Bindings can import other bindings.

Marshaling XML Entity Types

The mappings between simple Smalltalk objects and primitive XML datatypes are defined in the method `defaultSmalltalk2XMLMap` of class `XMLObjectBinding`. For example, an instance of class `String` is mapped to to the XML type `"xsd:string"`.

The VisualWorks X2O framework maps complex types with attributes to Smalltalk objects with instance variables, where each instance variable represents an attribute.

For a more detailed discussion of XML types:

```
http://www.w3.org/TR/xmlschema-1/#Type_Definition_Summary
```

XML complex types with simple content are discussed here:

```
http://www.w3.org/TR/xmlschema-1/#element-simpleContent
```

Marshaling XML <simpleType> Elements

Simple XML types like strings and numbers are marshaled using instances of `SimpleObjectMarshaler`. This marshaler handles XML primitive types that are defined as XML Schema datatypes.

When it is time to marshal between XML and Smalltalk objects, simple primitive types are resolved using serialization, deserialization and initializer blocks, stored in `BindingBuilder` class registries (i.e., shared variables).

Primitive type marshalers implement the methods `serialize:` and `deserialize:`, and have two blocks that actually perform the conversion between a XML string and a Smalltalk object. A `BindingBuilder` instance maintains dictionaries of serialization and deserialization blocks keyed by `id` (e.g. `'String'`, `'date'`, `'binary'`).

For example, the mapping for the XML data type `dateTime` is:

```
<simple name="dateTime" id="dateTime" />
```

The serialization and deserialization blocks are:

```
Deserializers at: 'date'
 put: [[:mv :string :aclass | self decodeDateFrom: string]].
```

```
Serializers at: 'date'
 put: [[:mv | self encodeDate: mv value]].
```

Similarly, to encode a `Double` object:

```
serializer := WebServices.BindingBuilder serializers at: 'Double'.
(serializer
 value: (MarshalingContext new value: 1.2 )
 value: Double) = '1.200000047683716e0'
```

To decode a XML string to an instance of `Double`:

```
deserializer := BindingBuilder deserializers at: 'Double'.
(deserializer
 value: (MarshalingContext new value: '1.200000047683716e0')
 value: Double) = 1.2
```

It is also possible to use custom-defined simple types derived from XML primitive types. For example, this is a primitive string type in which the only values allowed are "Male" and "Female":

```
<s:simpleType name="Gender">
 <s:restriction base="s:string">
  <s:enumeration value="Male" />
  <s:enumeration value="Female" />
 </s:restriction>
</s:simpleType>
```

This element would be described in the X2O binding as follows:

```
<simple baseType="xsd:string" name="Gender">
 <enumeration value="Male" />
 <enumeration value="Female" />
</simple>
```

The following example code illustrates this in detail:

```
x2oSpec := '<schemaBindings>
 <xmlToSmalltalkBinding targetNamespace="urn:schemaWithSimpleTypes"
 xmlns="urn:visualworks:VWSchemaBinding"
 xmlns:tns="urn:schemaWithSimpleTypes" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
 <struct name="GetLifeExpectancy">
  <element name="age" ref="xsd:int" />
  <element name="gender" ref="tns:Gender" />
 </struct>
 <simple baseType="xsd:string" name="Gender">
  <enumeration value="Male"></enumeration>
  <enumeration value="Female"></enumeration>
 </simple>
</xmlToSmalltalkBinding>
</schemaBindings>'.
x2oBinding := XMLObjectBinding loadFrom: x2oSpec readStream.
struct := WebServices.Struct new.
struct
 age: 25;
 gender: 'Male'.
xml := x2oBinding marshal: struct with: x2oBinding marshalers first.
```

Evaluating this example code should return:

```
<GetLifeExpectancy xsi:type="ns:GetLifeExpectancy"
 xmlns="urn:schemaWithSimpleTypes" xmlns:ns="urn:schemaWithSimpleTypes"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <age>25</age>
 <gender>Male</gender>
</GetLifeExpectancy>
```

While marshaling or unmarshaling the `Gender` element, the `SimpleObjectMarshaler` validates the string value and raises an `EnumerationValueError` if the value does not equal `'Male'` or `'Female'`.

Marshaling XML <complexType> Elements

Complex elements may contain other elements, attributes, and/or text.

For example, given a complex type defined as:

```
<xsd:complexType name="CommentString">
 <xsd:simpleContent>
  <xsd:extension base="xsd:string">
   <xsd:attribute name="id" type="xsd:string" />
  </xsd:extension>
 </xsd:simpleContent>
</xsd:complexType>
```

We can use the definition of CommentString to unmarshal this XML:

```
<CommentString id="123">some comments</CommentString>
```

This complex type with simple content will be mapped to a Struct or to an object with an instance variable named id.

Class ObjectMarshaler is the superclass for marshaling/unmarshaling complex objects, such as structs, objects, and collections. Class ComplexObjectMarshaler and its subclasses marshal objects with parts, into XML content with tags 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. This is the default mapping for XML complex types.

For example, a XML schema for the complex type Person might be:

```
<complexType name="Person">
 <sequence>
  <element name="name" type="xsd:string" />
  <element name="birthDate" type="xsd:string" />
 </sequence>
 <attribute name="age" use="required" type="xsd:int" />
```

```
</complexType>
```

Given this definition, the default X2O specification would be:

```
<struct name="Person">
  <sequence>
    <element name="name" ref="xsd:string" />
    <element name="birthDate" ref="xsd:string" />
  </sequence>
  <attribute name="age" use="required" ref="xsd:int" />
</struct>
```

And a XML element described by this struct could be:

```
WebServices.Struct new
 at: #name put: 'Smith';
 at: #birthDate put: '2000';
 at: #age put: 11;
 yourself.
```

Alternately, the XML schema could be defined using a complex type with simple contents:

```
<element name="Person">
<complexType>
   <simpleContent>
     <extension base="xsd:string">
       <attribute name="name" type="xsd:string" />
       <attribute name="birthDate" type="xsd:string" />
     </extension >
   </simpleContent>
  </complexType>
<element>
```

In this case, the default X2O specification would be:

```
<element name="Person">
<struct name="Person">
      <text value="xsd:string" />
      <attribute name="name" ref="xsd:string" />
      <attribute name="birthDate" ref="xsd:string" />
  </struct>
</element>
```

This struct would be unmarshaled as:

```
WebServices.Struct new
 at: #name put: 'Smith';
 at: #birthDate put: '2000';
 at: #value put: 11;
 yourself.
```

A <struct> element is, by default, marshaled by the StructMarshaler as an instance of WebServices.Struct (a subclass of ProtoObject), with its constituent objects as entries. The dictionary entries are specified by the aspect attribute.

Marshaling XML Complex Types as Objects

The XML <complexType> element information items can be mapped to <object> binding elements.

During the conversion process, an <object> element in the binding specification is marshaled by the ComplexObjectMarshaler as an instance of the class whose name is specified by its smalltalkClass attribute.

To illustrate how this works, let's consider several different ways that we could marshal a domain class called Person.

The XML schema for the complex type Person might be:

```
<complexType name="Person">
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="birthDate" type="xsd:string" />
  </sequence>
  <attribute name="age" use="required" type="xsd:int" />
</complexType>
```

In this case, the X2O specification would be:

```
<object name="Person" smalltalkClass="Person">
  <sequence>
    <element name="name"
  setSelector="setName:" getSelector="getName" ref="xsd:string" />
     <element name="birthDate" ref="xsd:string" />
  </sequence>
  <attribute name="age" use="required" ref="xsd:int" />
```

```
</object>
```

Now, given this definition of a `Person` object:

```
Person new
 setName: 'Smith';
 birthDate: '2000';
 age: 11;
 yourself.
```

Using the X2O binding shown above, this `Person` object will be marshaled to XML as:

```
<person age="11">
 <name>Smith</name>
 <birthDate>2000</birthdate>
</person>
```

Alternately, here's the XML schema for the complex type `Person`, with simple contents:

```
<element name="Person">
  <complexType>
    <simpleContent>
      <extension base="xsd:string">
        <attribute name="name" type="xsd:string" />
        <attribute name="birthDate" type="xsd:string" />
      </extension>
    </simpleContent>
  </complexType>
<element>
```

In this case, the default X2O specification would be:

```
<element name="Person">
  <object name="Person" smalltalkClass="Person">
      <text value="xsd:string" />
      <attribute name="name" ref="xsd:string" />
      <attribute name="birthDate" ref="xsd:string" />
  </object>
</element>
```

Now, given this definition of a Person object:

```
Person new
 name: 'Smith';
 birthDate: '2000';
 value: 11;
 yourself.
```

Using the alternate X2O binding shown above, this Person object will be marshaled to XML as:

```
<person name="Smith" birthDate="2000">11</person>
```

During the conversion process, an <object> binding element is marshaled by the ComplexObjectMarshaler as an instance of the class whose name is specified by the smalltalkClass attribute.

Specifying Name Spaces and Superclasses

A Smalltalk name space can be set in the binding specification as a default for all classes:

```
<xmlToSmalltalkBinding
 targetNamespace="urn:vwservices:test"
 defaultClassNamespace="WebServices">
```

Alternately, the class' name space can be specified using the smalltalkClass attribute:

```
<object name="Person" smalltalkClass="WebServices.Person">
```

In addition, the superclass can be defined with the baseType attribute:

```
<object
 name="Male" smalltalkClass="WebServices.Male" baseType="Person">
```

Handling Exceptions

When the X2O binding is loaded, the Smalltalk class must be defined in the development environment; otherwise, the binding builder raises an ClassIsNotDefinedSignal exception.

This exception is resumeable. It is raised with an instance of ComplexObjectMarshaler as its parameter. Using this mechanism, it is possible to specify a Smalltalk class for the ComplexObjectMarshaler and then resume the exception, e.g.:

```
[XMLObjectBinding loadFrom: x2oSpec]
 on: ClassIsNotDefinedSignal
 do:  [:ex | ex parameter smalltalkClass: Person.
  ex proceed].
```

Complex Type Elements and XPath

When we unmarshal a document, XPath attributes define the way that we select an XML element. For example, consider the following X2O specification:

```
<object name="Male" smalltalkClass="Male" baseType="Person">
  <element
 name="Name" aspect="name"
 setSelector="setName:" getSelector="getName" ref="xsd:string" />
```

By default, we would create an XPath "child::name" for the element name, where the child's parent is a <person> complex type.

It is also possible to change the default XPath attributes in the X2O specification. For example:

```
x2o := '<?xml version ="1.0"?>
 <xmlToSmalltalkBinding
  name="testEmbededText"
  targetNamespace="urn:XMLMarshalingTests#testEmbededText"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <struct name="tspan" smalltalkClass="WebServices.Struct">
  <attribute name="font-style" aspect="fontStyle" ref="xsd:string" />
  <text aspect="text" ref="xsd:string" />
 </struct>
 <object name="text" smalltalkClass="WebServices.MyText">
  <attribute name="font-style" aspect="fontStyle" ref="xsd:string" />
  <all>
   <element
    aspect="text"
    ref="xsd:string"
    xpathPrefix="" xpath= "//child::text()" />
```

```
   <element name="tspan" ref="tspan" />
  </all>
 </object>
</xmlToSmalltalkBinding>'.
WebServices.XMLObjectBinding loadFrom: x2o readStream.
text := '<text xmlns="urn:XMLMarshalingTests#testEmbededText" font-style="normal">
  This text is normal,
    <tspan font-style="bold">but this is bold,</tspan>
  while this is normal again.
</text>'.
text := XMLObjectBinding
   unmarshal: (ReadStream on: text)
   atNamespace: 'urn:XMLMarshalingTests#testEmbededText'.
text text ='  This text is normal,    but this is bold,  while this is normal again.'.
text fontStyle = 'normal'.
text tspan fontStyle = 'bold'.
text tspan text = 'but this is bold,'.
```

Here, the X2O specification declares specific XPath attributes, which can be accessed using Smalltalk methods.

Using Derived Types

Support for derived types follows the model defined by XML Schema. It defines a mechanism to force substitution for a particular type. Just as it is possible to derive a complex from a simple type, so too you can derive a new complex type from an existing one.

When a type is declared to be abstract, it cannot be used in an instance document. When an element's corresponding type definition is declared as abstract, all instances of that element must use `xsi:type` to indicate a derived type that is not abstract (`xsi:type` is part of the XML Schema instance name space).

For example, let's say we want to map instances of class `Library`, `Holding`, `Book`, and `ShipPlan` (a library contains holdings, which may be books or ship plans). The following code and binding specification creates a suitable binding object:

```
binding := XMLObjectBinding
    loadFrom: '<?xml version="1.0" encoding="utf-8" ?>
<schemaBindings>
 <xmlToSmalltalkBinding
  targetNamespace="urn:derivedTypes"
```

```
 defaultClassNamespace="Protocols.Library"
 xmlns="urn:visualworks:VWSchemaBinding"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="urn:derivedTypes">
 <object name="Holding" abstract="true" smalltalkClass="Holding">
  <sequence>
   <element name="libraryName" ref="xsd:string" />
  </sequence>
 </object>
 <object name="Book" smalltalkClass="Book" baseType="tns:Holding"
constraint="extension">
  <sequence>
   <element name="catalogNumber" ref="xsd:int" />
   <element name="title" ref="xsd:string" />
  </sequence>
 </object>
 <object name="ShipPlan" smalltalkClass="ShipPlan" baseType="tns:Holding"
constraint="extension">
  <sequence>
   <element name="scale" ref="xsd:float" />
   <element name="shipName" ref="xsd:string" />
  </sequence>
 </object>
 <object name="Library" smalltalkClass="Library">
  <sequence>
   <element name="ownedHoldings" maxOccurs="*" ref="tns:Holding" />
  </sequence>
 </object>
</xmlToSmalltalkBinding>
</schemaBindings>' readStream.
```

Here, we define the `Holding` type as an abstract type, which is referenced by the `ownedHoldings` of the `Library` object. In this way, to encode and decode the `Library` object, we can use a new type derived from type `Holding`, but not the `Holding` type itself. The `Book` and `ShipPlan` types are encoded with the `xsi:type` attribute to indicate which derived type is used.

To illustrate the use of this binding, let's say we have a `Book` object stored in the `Library`, e.g.:

```
book := Protocols.Library.Book new
 libraryName: 'City Public Library';
 catalogNumber: 12345;
```

```
 title: 'Distributed Algorithms';
 yourself.
library := Protocols.Library.Library new
 ownedHoldings: (OrderedCollection with: book);
 yourself.
```

To marsal this as XML, evaluate the following:

```
xml := binding marshal: library.
```

The resulting XML should be:

```
<Library xmlns="urn:derivedTypes" xmlns:ns="urn:derivedTypes" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
 <ownedHoldings xsi:type="ns:Book">
  <libraryName>City Public Library</libraryName>
  <catalogNumber>12345</catalogNumber>
  <title>Distributed Algorithms</title>
 </ownedHoldings>
</Library>
```

To unmarshal this XML, evaluate:

```
unmarshaledObject := binding unmarshal: xml.
```

Note, however, that using this binding specification, we cannot insert a `Holding` object into the library and marshal that to XML, e.g.:

```
holding := Protocols.Library.Holding new
 libraryName: 'City Public Library';
 yourself.
library := Protocols.Library.Library new
 ownedHoldings: (OrderedCollection with: holding);
 yourself.
binding marshal: library.
```

Attempting to marshal the `library` object will raise a `WrongObjectType` exception.

In general, the following invariants apply when encoding derived types:

- An instance of ComplexObjectMarshaler specified as abstract expects to marshal an instance of a class inherited from its Smalltalk class, or raises a WrongObjectException.
- If an instance of ComplexObjectMarshaler is not specified as abstract, it can marshal instances derived from its Smalltalk class
- All derived types are encoded with an inline type attribute

In general, when decoding derived types:

- An instance of ComplexObjectMarshaler specified as abstract expects an inline type attribute to decode the XML element or raises a MissingTypeAttributeSignal exceptionIf an instance of ComplexObjectMarshaler specified as abstract cannot find a marshaler for an element or the marshaler found is not derived from the object marshaler, an NoMarshalerSignal exception is raised.

The following implementation details are also worth noting: if the binding specification includes complex types whose derivation is specified as #extension, or #restriction, or #abstract, and if the binding maps complex types to a Struct (the default binding), the encoding for derived types may be incorrect. Under these circumstances, the XMLTypesParser will raise a NotSupportedDerivation exception.

Note: In VisualWorks 7.9, the #useDerivedTypes option has been deprecated and is not used for encoding or decoding. This option has likewise been removed from the Web Services wizard. In addition, the default value for #useInlineType is now set to false. This option was created for the first version of the VisualWorks Web Services implementation, when the principle style was RPC/encoded. The #literal style now causes a xsi:type attribute to be added, as required by the XML Schema specification. RPC/encoded still uses an inline type to encode messages.

Using Unbounded Sequences

To illustrate the use of unbounded sequences of elements, let's assume the following complex type:

```
<complexType name="CategoryType">
 <attribute name="attr" type="xsd:string" />
 <sequence maxOccurs="unbounded">
```

```
  <element name="count" type="xsd:long" />
  <element name="delimiter" type="xsd:string" />
 </sequence>
</complexType>
```

The object X2O specification for this complex type would be defined as:

```
<object name="CategoryType" smalltalkClass="CategoryType">
 <attribute name="attr" ref="xsd:string"></attribute>
 <sequence maxOccurs="unbounded">
  <element name="count" ref="xsd:long"></element>
  <element name="delimiter" ref="xsd:string"></element>
 </sequence>
</object>
```

This `<object>` binding element is marshaled by `ComplexObjectMarshaler` as an instance of the Smalltalk class `CategoryType`. For this example, class `CategoryType` has to be defined as a collection of `Struct` with `#count` and `#delimiter` keys, while the XML attribute `#attr` is defined as an instance variable. E.g.:

```
Smalltalk.WebServices defineClass: #CategoryType
 superclass: #{Core.Object}
 indexedType: #objects
 private: false
 instanceVariableNames: 'attr'
```

For convenience, this class is already defined in the `WebServicesDemoModels` package.

In the VisualWorks implementation, XML complex types with unbounded sequences can only be mapped to `<object>` elements. Due to their ambiguous semantics (e.g., attribute names can collide), there is no support for the default `<struct>` mapping. If the X2O Binding Tool is being used, it detects unbounded sequences and converts them to `<object>` elements.

When the `XMLTypesParser` creates a default X2O specification for an unbounded sequence it raises the `MappingUnboundedSequenceToStruct` exception and the default resume action creates an `<object>` mapping for the type instead of a `<struct>`.

For example:

```
schema := '<xs:schema
  targetNamespace="urn:testDefaultMappingUnboundedSequence"
  xmlns="urn:testDefaultMappingUnboundedSequence"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xs:complexType name="CategoryType">
   <xs:sequence maxOccurs="unbounded">
    <xs:element name="count" type="xs:int" />
    <xs:element name="delimiter" type="xs:string" />
   </xs:sequence>
  </xs:complexType>
  <xs:complexType name="EventType">
   <xs:sequence>
    <xs:element name="id" type="xs:int" minOccurs="0" />
   </xs:sequence>
  </xs:complexType>
</xs:schema>'.
[x2o := XMLTypesParser readFrom: schema readStream]
 on: MappingUnboundedContentToStruct
 do: [:ex | ex resume].
```

This code snippet (above) creates the following X2O specification:

```
<schemaBindings>
 <xmlToSmalltalkBinding
  defaultClassNamespace="WSDefault"
  elementFormDefault="qualified"
  targetNamespace="urn:testDefaultMappingUnboundedSequence"
  xmlns="urn:visualworks:VWSchemaBinding"
  xmlns:tns="urn:testDefaultMappingUnboundedSequence"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <object name="CategoryType" smalltalkClass="CategoryType">
   <sequence maxOccurs="unbounded">
    <element name="count" ref="xsd:int"></element>
    <element name="delimiter" ref="xsd:string"></element>
   </sequence>
  </object>
  <struct name="EventType">
   <sequence>
    <element minOccurs="0" name="id" ref="xsd:int"></element>
   </sequence>
  </struct>
```

```
 </xmlToSmalltalkBinding>
</schemaBindings>
```

To marshal an unbounded sequence:

```
schema := '<xs:schema
 targetNamespace="urn:schemaWithUnboundedMandatorySequence"
 xmlns="urn:schemaWithUnboundedMandatorySequence"
 xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified" >
  <xs:complexType name="CategoryType">
   <xs:attribute name="attr" type="xs:string" />
   <xs:sequence maxOccurs="unbounded">
    <xs:element name="count" type="xs:long" />
    <xs:element name="delimiter" type="xs:string" />
   </xs:sequence>
  </xs:complexType>
</xs:schema>'.
x2o := XMLTypesParser
    useObjectBindingReadFrom: schema readStream
    inNamespace: 'WebServices'.
binding := (XMLObjectBinding buildBindings: x2o realElements) first.
manager := XMLObjectMarshalingManager on: binding.
ns := 'urn:schemaWithUnboundedMandatorySequence'.
(category := CategoryType new: 2)
      attr: 'attribute';
      at: 1 put: (WebServices.Struct new
          at: #count put: 12;
          at: #delimiter put: 'delimiter1';
          yourself);
      at: 2 put: (WebServices.Struct new
          at: #count put: 34;
          at: #delimiter put: 'delimiter2';
          yourself).
xml := manager marshal: category atNamespace: ns.
```

The `CategoryType` object will be marshaled and assigned to the variable `xml` as:

```
<ns:CategoryType
 attr="attribute"
 xmlns:ns="urn:schemaWithUnboundedMandatorySequence">
 <ns:count>12</ns:count>
 <ns:delimiter>delimiter1</ns:delimiter>
```

```
 <ns:count>34</ns:count>
 <ns:delimiter>delimiter2</ns:delimiter>
</ns:CategoryType>
```

Using Nested <sequence> Elements

It is also possible to nest <sequence> elements in a complex type. For example, the following binding specification maps instances of class CategoryType to an <object> element:

```
<schemaBindings>
 <xmlToSmalltalkBinding
  defaultClassNamespace="WebServices"
  targetNamespace="urn:schemaWithNestedMandatorySequence"
  xmlns="urn:visualworks:VWSchemaBinding"
  xmlns:tns="urn:schemaWithNestedMandatorySequence"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <object name="CategoryType" smalltalkClass="CategoryType">
  <sequence>
   <element minOccurs="0" name="id" ref="xsd:int"></element>
   <sequence aspect="sequenceValue" minOccurs="1">
    <element name="count" ref="xsd:int"></element>
    <element name="delimiter" ref="xsd:string"></element>
   </sequence>
  </sequence>
 </object>
 </xmlToSmalltalkBinding>
</schemaBindings>
```

Alternately, a <complexType> with nested sequences can be mapped with the default binding as:

```
<schemaBindings>
 <xmlToSmalltalkBinding
  defaultClassNamespace="WebServices"
  targetNamespace="urn:schemaWithNestedMandatorySequence"
  xmlns="urn:visualworks:VWSchemaBinding"
  xmlns:tns="urn:schemaWithNestedMandatorySequence"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <struct name="CategoryType">
 <sequence>
  <element minOccurs="0" name="id" ref="xsd:int"></element>
   <sequence aspect="sequenceValue" minOccurs="1">
```

```
    <element name="count" ref="xsd:int"></element>
    <element name="delimiter" ref="xsd:string"></element>
   </sequence>
 </sequence>
 </struct>
 </xmlToSmalltalkBinding>
</schemaBindings>
```

For this nested sequence, class CategoryType would need to be defined as:

```
Smalltalk.WebServices defineClass: #CategoryType
 superclass: #{Core.Object}
 indexedType: #none
 private: false
 instanceVariableNames: ' id sequenceValue'
 classInstanceVariableNames: ''
```

Here, the instance variable sequenceValue holds a Struct object with the keys #count and #delimiter. To marshal this nested sequence:

```
binding := XMLObjectBinding loadFrom: '<schemaBindings>
 <xmlToSmalltalkBinding
  defaultClassNamespace="WebServices"
  elementFormDefault="qualified"
  targetNamespace="urn:schemaWithNestedMandatorySequence"
  xmlns="urn:visualworks:VWSchemaBinding"
  xmlns:tns="urn:schemaWithNestedMandatorySequence"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <object name="CategoryType" smalltalkClass="CategoryType">
 <sequence>
  <element minOccurs="0" name="id" ref="xsd:int"></element>
  <sequence aspect="sequenceValue" minOccurs="1">
   <element name="count" ref="xsd:int"></element>
   <element name="delimiter" ref="xsd:string"></element>
  </sequence>
 </sequence>
</object>
</xmlToSmalltalkBinding>
</schemaBindings>' readStream.
manager := XMLObjectMarshalingManager on: binding.
ns := 'urn:schemaWithNestedMandatorySequence'.
(c := CategoryType new)
```

```
   id: 123;
   sequenceValue: (Struct new
         at: #count put: 12;
         at: #delimiter put: 'delimiter1';
         yourself).
xml := manager marshal: c atNamespace: ns.
```

The result, in the variable `xml`, should be:

```
<ns:CategoryType xmlns:ns="urn:schemaWithNestedMandatorySequence">
 <ns:id>123</ns:id>
 <ns:count>12</ns:count>
 <ns:delimiter>delimiter1</ns:delimiter>
</ns:CategoryType>
```

To marshal a nested unbounded sequence:

```
binding := XMLObjectBinding loadFrom: '<schemaBindings>
 <xmlToSmalltalkBinding
  defaultClassNamespace="WebServices"
  elementFormDefault="qualified"
  targetNamespace="urn:schemaWithNestedMandatorySequence"
  xmlns="urn:visualworks:VWSchemaBinding"
  xmlns:tns="urn:schemaWithNestedMandatorySequence"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <object name="CategoryType" smalltalkClass="CategoryType">
 <sequence>
  <element minOccurs="0" name="id" ref="xsd:int"></element>
  <sequence aspect="sequenceValue"
   minOccurs="1" maxOccurs="unbounded">
   <element name="count" ref="xsd:int"></element>
   <element name="delimiter" ref="xsd:string"></element>
  </sequence>
 </sequence>
 </object>
 </xmlToSmalltalkBinding>
</schemaBindings>' readStream.
manager := XMLObjectMarshalingManager on: binding.
ns := 'urn:schemaWithNestedMandatorySequence'.
(c := CategoryType new)
   id: 123;
   sequenceValue: (OrderedCollection with:
         (Struct new
          at: #count put: 12;
```

```
            at: #delimiter put: 'delimiter1';
            yourself)).
xml := manager marshal: c atNamespace: ns.
```

The result, in the variable xml, should be:

```
<ns:CategoryType xmlns:ns="urn:schemaWithNestedMandatorySequence">
 <ns:id>123</ns:id>
 <ns:count>12</ns:count>
 <ns:delimiter>delimiter1</ns:delimiter>
</ns:CategoryType>
```

Mapping XML <union> Elements

A union defines a collection of simple types. For example, we could define a #size type as a union of an integer and a token, such that size can be a number or the string "small" or "large":

```
<union name="size">
 <simple baseType="xsd:integer" />
 <simple baseType="xsd:token">
  <enumeration value="small" />
  <enumeration value="large" />
 </simple>
</union>
```

Alternately, we could define it using the memberTypes attribute:

```
<union name="size" memberTypes="tns:integerType tns:booleanType" >
 <simple baseType="xsd:token">
  <enumeration value="small" />
  <enumeration value="medium" />
  <enumeration value="large" />
 </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 name="color" ref="xsd:string" />
 </struct>
```

The UnionMarshaler object 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 is added first to the marshalers collection. The evaluation order can be overridden with the use of xsi:type.

To illustrate, consider the following X2O specification:

```
<element name="Coat">
 <struct name="Coat">
  <sequence>
   <element name="size" ref="tns:size" />
   <element name="color" ref="xsd:string" />
  </sequence>
 </struct>
</element>
<union name="size">
 <simple baseType="xsd:integer" />
 <simple baseType="xsd:token">
  <enumeration value="small" />
  <enumeration value="medium" />
  <enumeration value="large" />
 </simple>
</union>
```

If we use this specification to marshal a Struct:

```
WebServices.Struct new
 size: 6;
 color: 'red';
 yourself.
```

we should get the following XML document:

```
<ns:Coat
 xmlns:ns="urn:unionType"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ns:size xsi:type="xsd:integer">6</ns:size>
  <ns:color>red</ns:color>
</ns:Coat>
```

or, using this Struct:

```
WebServices.Struct new
 size: 'small';
 color: 'red';
 yourself.
```

we should get the following XML document:

```
<ns:Coat
 xmlns:ns="urn:unionType"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ns:size xsi:type="xsd:token">small</ns:size>
  <ns:color>red</ns:color>
</ns:Coat>
```

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 a XPath expression defined as 'child::xxx', where xxx is the value of the named attribute.

For example, the XML Schema element:

```
<element name="birthdate" type="xsd:string" />
```

is going to be described in an X2O specification as:

```
<element name="birthdate" ref="xsd:string" />
```

Here, the ElementMarshaler created for the element includes the XPath expession "child::birthdate" which is used for marshaling and unmarshaling.

To marshal a Smalltalk object, the ElementMarshaler uses its resolver (either a simple or a complex marshaler) to encode the object. This resolver is defined by the #ref attribute. The aspect attribute is used to get the Smalltalk object from the element marshaler's parent.

For example, let's say we have a class Customer with an instance variable name, and getter and setter methods name and name:. A simple X2O specification for this class might be:

```
<object name="Customer">
 <element name="name" aspect="name" ref="xsd:string" />
</object>
```

The #name element will be encoded using a SimpleObjectMarshaler with an encoder for String objects, and the value of #name will be retrieved from a Customer object using the #name method.

Since relations may be either one-sided or many-sided, an instance of a subclass of Relation is used to represent a relation between an object and its parts. A relation has a descriptive name, getter and setter selectors, and cardinality constraints.

For example, an element with a one-sided relation might be:

```
<element ref="personName" />
```

and an element with a many-sided relation might look like this:

```
<element ref="email" minOccurs="0" maxOccurs="*" aspect="emailList" />
```

In the latter case, the cardinality constraints are expressed using the minOccurs and maxOccurs attributes.

An ElementMarshaler object contains a Relation object that represents the element's cardinality. This relation object can be an instance of OneRelation, if maxOccurs="1", or ManyRelation if maxOccurs is more than one. The relation object also holds information about the aspect, setter and getter methods of the ElementMarshaler.

For example, an element marshaler that uses OneRelation:

```
<element name="birthdate" ref="xsd:string:/>
```

will marshal the value 'June 1, 2000' as:

```
<birthdate>June 1, 2000</birthdate>
```

Whereas an element marshaler that uses ManyRelation, e.g.:

```
<element
 name="id" minOccurs="0" maxOccurs="unbounded" ref="xsd:string" />
```

will marshal a collection of strings #('1' '2' '3') as:

```
<id>1</id>
<id>2</id>
<id>3</id>
```

If an element marshaler has a relation with minOccurs of 0, we call this element optional and the marshaler is allowed to skip these values. If an element marshaler is created with minOccurs set to 1, the marshaler considers the value to be mandatory and will raise a MissingValueSignal for any nil values.

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 Struct aspects these are keys for at: and at:put: methods.

If an XML Schema element is defined using the reference ref="name", e.g.:

```
<element name="name" type="xsd:string" />
<compexType name="Person">
  <element ref="name" />
</complexType>
```

In this case, the X2O specification maps the element to an ImplicitMarshaler object:

```
<element name="name" ref="xsd:string" />
<object name="Person" smalltalkClass="Person">
  <implicit ref="name" />
</object>
```

The ImplicitMarshaler inherits the element description (the type resolver and XPath) from the global element (that is, <element name="name" type="xsd:string" />).

Marshaling XML Attributes

An <attribute…/> binding defines relations for XML attributes.

For example, for the 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:

```
SampleDocument new
 id: '1234';
 yourself.
```

Marshaling and unmarshaling are performed using the XPath expression '@xxx', where xxx is the value of name attribute. The marshaler is an instance of class RelationMarshaler.

Attribute Name Spaces and White Space

Attributes may be scoped using XML name spaces, including special attributes such as xml:lang and xml:space (the latter may be used to control white space processing).

To illustrate, we can write a binding specification for a simple struct that also defines an xml:space attribute:

```
x2o :='<xmlToSmalltalkBinding name="Test1Binding"
   targetNamespace="urn:test:Test1Binding"
   xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
   xmlns:tns="urn:test:Test1Binding">
   <struct name="MyObject">
    <implicitAttribute ref="xml:space" />
   <text ref="xsd:string" aspect="data" />
   </struct>
  </xmlToSmalltalkBinding>'.
```

Here, the implicitAttribute definition is necessary because ref is used instead of type.

To generate a binding specification object from this, and to unmarshal XML for <MyObject ...>, we can evaluate the following:

```
manager := XMLObjectMarshalingManager
      on: (XMLObjectBinding loadFrom: x2o readStream).
document := ((XMLParser on: '<MyObject xml:space="default"
 xmlns="urn:test:Test1Binding">some  text  </MyObject>' readStream)
      validate: false;
      scanDocument)
      root.
aStruct := manager
     unmarshal: document
     with: manager binding marshalers first.
```

By default, the white space is preserved.

Similarly, to marshal a Struct object into an XML element, and explicitly direct VisualWorks to preserve the white space:

```
aStruct := WebServices.Struct new
  data: 'testing  some text';
  space: 'preserve';
  yourself.
xml := manager
    marshal: aStruct
    with: manager binding marshalers first.
```

For details on the xml: name space, see the class-side method #xmlBinding1998Namespace in XMLObjectBinding, and:

```
http://www.w3.org/XML/1998/namespace
```

Marshaling XML Values

A <text.../> binding defines relations for a XML value. Marshaling/unmarshaling is done using the XPath expression 'child::text()'. The marshaler is an instance of class TextMarshaler.

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:

```
SampleDocument new
 value: 'some text';
 yourself.
```

Marshaling XML Wildcard Elements

The wildcard bindings <any.../> and <anyAttribute.../> may be used to specify mappings between arbitrary XML elements and attributes. These bindings are especially useful when you need to accommodate loosely-structured data inside elements of your content model.

Both <any.../> and <anyAttribute.../> bindings may include namespace, minOccurs, maxOccurs, and processContents attributes. These are used to constrain the allowable content matched by the wildcard. Each attribute is explained below.

The value of the namespace attribute is generally a single name space, or a whitespace-separated list of name spaces. Matching elements are expected to be defined in these name spaces. In addition, several reserved values may be used with the namespace attribute to include or exclude element content:

Namespace Attribute	Allowable Content
##any	Any well-formed XML from any name space (default, if no name space is specified).
##local	Any well-formed XML that is not qualified, i.e. not declared to be in a name space.
##other	Any well-formed XML that is from a name space other than the target name space of the type

Namespace Attribute	Allowable Content
	being defined (unqualified elements are not allowed).

Both <any.../> and <anyAttribute.../> bindings may also specify a processContents attribute to indicate whether the XML content matched by the wildcard should be validated. This attribute can have one of three possible values: skip, lax, or strict. By default, the value is strict.

If processContents is specified as skip, the X2O marshaler accepts and returns XML elements (e.g., instances of XML.Element) without any attempt to marshal or unmarshal them.

If processContents is specified as lax, the X2O marshaler tries to find an ElementMarshaler. If it cannot be found, an instance of XML.Element is returned as a result of unmarshaling.

If processContents is specified as strict, the X2O marshaler tries to find an instance of ElementMarshaler to marshal/unmarshal the XML node. If none can be found, the NoMarshalerSignal exception will be raised.

The minOccurs attribute may be zero or a positive number. The maxOccurs attribute may be zero, positive, or unbounded. By default, both minOccurs and maxOccurs are 1.

Wildcard bindings are processed using either an AnyRelationMarshaler, or an AnyAttributeMarshaler. For marshaling, the marshaler takes a tag and a value. The AnyRelation/Attribute marshaler finds the specific ElementMarshaler class that matches the tag, validates that it is defined in the scope of its name space, and then uses the ElementMarshaler to encode the value.

For unmarshaling, a complex type marshaler processes the subnodes of the content model sequentially, using an AnyRelationMarshaler to finds a specific ElementMarshaler for it. If the ElementMarshaler is in the scope of the node's name space, it will be used to unmarshal the node. Otherwise, an exception is raised.

Marshaling XML <any> Elements

An <any.../> binding specifies that any well-formed XML element can appear in the content model. This binding functions as a wildcard

schema component, and is often used for more flexibility in content models, or for extending complex types.

The marshaling/unmarshaling of <any> starts with an instance of AnyRelationMarshaler, which searches for an appropriate marshaler based on the element's name space and type.

For example, assume that we have the 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 the XML element:

```
<GetPerson>
 <Person1>
  <name>any name</name>
 </Person1>
</GetPerson>
```

or the XML element:

```
<GetPerson>
 <Person2>
  <age>10</age>
 </Person2>
</GetPerson>
```

The unmarshaled object will be a Struct with entry #contents and its value set to an instance of Person1 or Person2, e.g.:

```
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 a 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 a Struct with entry #person1 and a value instance of Person1, or entry #person2 and a 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).
```

Non-Deterministic Content

If a complex type marshaler includes an <any> element the complex marshaler will be checked that it represents a deterministic content model.

For more information see:

W3C XML Schema determinism definition

http://www.w3.org/TR/xml/#determinism

W3C XML Schema determinism rules

http://www.xml.com/pub/a/2004/10/27/extend.html?page=6

Loading a XML Object Binding with a non-deterministic content model raises a NonDeterministicContentModel exception. This exception, however, is proceedable. If you proceed the exception, the binding is created and registered. If you use this binding, though, you should be aware that unmarshaling can produce an erroneous result.

For example:

```
schema := '<?xml version="1.0" encoding="utf-8" ?>
<schema targetNamespace="urn:AnyTypeTests#testAnyOthersStrict_AnyAspect"
 xmlns="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="urn:AnyTypeTests#testAnyOthersStrict_AnyAspect" xmlns:xsd="http://
www.w3.org/2001/XMLSchema">
  <complexType name="Library" >
   <sequence>
    <element
     name="services" minOccurs="1" maxOccurs="unbounded"
     type="xsd:string" />
    <any namespace="##any" processContents="strict" />
   </sequence>
  </complexType>
</schema>'.
[binding := XMLObjectDriver loadFrom:
 (XMLTypesParser
```

```
useObjectBindingReadFrom: schema readStream
inNamespace: 'Protocols.Library') printString readStream]
on: NonDeterministicContentModel
do: [:ex | ex proceed].
```

Specifying Input and Output for AnyRelationMarshaler

By default, an AnyRelationMarshaler expects an object wrapped in a TaggedObject. The unmarshaled value will be also returned as a TaggedObject. The TaggedObject provides information about an object to marshal and a tag for finding the appropriate ElementMarshaler.

This information in the TaggedObject must be provided when <any> is specified with processContents set to #strict and may optionally be provided when processContents is set to #lax. If processContents is #lax or #skip, the TaggedObject can pass a XML element to AnyRelationMarshaler to insert it in a XML document.

If an <any> wildcard is described with maxOccurs more than 1, AnyRelationMarshaler expects an arbitrary collection of tagged objects, where each tagged object can provide information about a different marshaler.

For backward compatibility AnyRelationMarshaler can also accept non-tagged objects. A type marshaler for this object will be found by the object class name. To be able to send non-tagged objects you need to change the binding option using: #useTaggedObject:.

For example:

```
binding := XMLObjectDriver loadFrom: x2o readStream.
binding useTaggedObject: false.
```

If #useTaggedObject is set to false, the AnyRelationMarshaler object returns the unmarshaled value but not a TaggedObject.

If an <any> element is described with the name space ##any, it can be time consuming to search all global registry entries for a marshaler. To narrow the search scope, you can set #wildcardScope: in the marshaling manager.

```
manager := XMLObjectMarshalingManager on: binding.
manager
 addToWildcardScope: 'urn:AnyTypeTests';
```

```
addToWildcardScope: 'urn:anotherNS'.
```

The method #addToWildcardScope: accepts a binding name space represented as a String. The binding must be registered in the XMLObjectBinding.XMLBindingRegistry registry.

Using a TaggedObject

An instance of TaggedObject provides information about an object to marshal and a type tag for finding the appropriate ElementMarshaler. Class TaggedObject includes the following class-side protocol:

ns: namespaceString type: aString value: anObject

Return a new instance whose marshaler tag name space is set to namespaceString, whose type tag is set to aString, and whose value (i.e., the object to marshal or unmarshal) is set to anObject.

element: xmlElement

Return a new instance that wraps xmlElement, which should be an instance of XML.Element.

For example, to wrap an instance of class Book in a TaggedObject:

```
TaggedObject
 ns: 'urn:AnyTypeTests'
 type: 'loanedHolding'
 value: (Book new catalogNumber: 'XR67890'; yourself).
```

To wrap a XML.Element in a TaggedObject:

```
TaggedObject
 element:
  ((XMLParser
   on:  '<holdings xmlns="urn:testing/lax">
      <catalogNumber>XR12345</catalogNumber>
    </holdings>' readStream)
   validate: false; scanDocument) root.
```

Instances of XMLObjectBinding can also create TaggedObject objects, using XMLObjectBinding>>tag: anObject withType: aString, where anObject is the object to marshal, and aString is the binding marshaler tag type.

Passing Parameters for AnyRelationMarshaler

This following code illustrates how to create objects with TaggedObject for an X2O specification that includes <any>. For example, let's say we have a complex object type called Library, defined as:

```
<complexType name="Library">
  <sequence>
    <element name="libraryName" minOccurs="1" type="xsd:string" />
    <any namespace="##any" processContents="strict" />
  </sequence>
</complexType>
```

By specifying namespace="##any" and processContents="strict", the name space of the element node can be any string, and the marshaling framework will accept any marshaler (that was specified by TaggedObject) from the global X2O registry.

To illustrate, the following code creates an X2O specification from an XML schema, loads an XMLObjectBinding, creates an instance of class Library, and then marshals it to XML:

```
schema := '<?xml version="1.0" encoding="utf-8" ?>
<schema targetNamespace="urn:AnyTypeTests" xmlns="http://www.w3.org/2001/
XMLSchema" xmlns:tns="urn:AnyTypeTests" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
 <complexType name="Library" >
  <sequence>
   <element name="libraryName" minOccurs="1" type="xsd:string" />
   <any namespace="##any" processContents="strict" />
  </sequence>
 </complexType>
 <element name="loanedHoldings" minOccurs="0" type="tns:Book" />
 <element name="ownedHoldings" minOccurs="0" type="tns:Book" />
 <complexType name="Book" >
  <sequence>
  <element name="catalogNumber" minOccurs="0" type="xsd:string" />
  </sequence>
 </complexType>
</schema>'.
binding := XMLObjectDriver loadFrom:
  (XMLTypesParser
    useObjectBindingReadFrom: schema readStream
    inNamespace: 'Protocols.Library') printString readStream.
```

```
manager := XMLObjectMarshalingManager on: binding.
(library := Library new)
  libraryName: 'Port Nowhere Public Library';
  any: (binding
    tag: (Book new catalogNumber: 'XR12345'; yourself)
    withType: 'loanedHoldings').
xml := manager marshal: library.
```

After evaluating this code, the marshaled XML document should be:

```
<Library>
  <libraryName>Port Nowhere Public Library</libraryName>
   <loanedHoldings>
      <catalogNumber>XR12345</catalogNumber>
   </loanedHoldings>
</Library>
```

Passing Collections

In the example described above, the schema element `loanedHoldings` is described as a collection:

```
<element name="loanedHoldings"
 minOccurs="0" maxOccurs="unbounded" type="tns:Book" />
```

The parameter to the `any:` method is prepared as:

```
(library := Library new)
  libraryName: 'Port Nowhere Public Library';
  any: (binding
    tag: (OrderedCollection
      with: (Book new catalogNumber: 'XR12345'; yourself)
      with: (Book new catalogNumber: 'XR67890'; yourself))
    withType: 'loanedHoldings').
```

The `<any>` element in the schema is decribed as a collection:

```
<any namespace="##any"
 maxOccurs="unbounded" processContents="strict" />
```

The parameter to the `any:` method is prepared as:

```
taggedObjects := OrderedCollection new.
```

```
taggedObjects add: (TaggedObject
    ns: 'urn:AnyTypeTests'
    type: 'ownedHoldings'
    value: (Book new catalogNumber: 'XR12345'; yourself)).
taggedObjects add: (TaggedObject
    ns: 'urn:AnyTypeTests'
    type: 'loanedHolding'
    value: (Book new catalogNumber: 'XR67890'; yourself)).
(library := Library new)
 libraryName: 'Port Nowhere Public Library';
 any: taggedObjects.
```

Passing XML Elements

Let's assume that the schema defined `<any>` as:

```
<any namespace="##any" processContents="lax" />
```

The parameter to the `any:` method could then be passed a XML element, as follows:

```
(library := Library new)
 libraryName: 'Port Nowhere Public Library';
 any: (TaggedObject element:
   ((XMLParser on: '<holdings xmlns="urn:testing/lax"><catalogNumber>XR12345</
catalogNumber></holdings>' readStream)
    validate: false; scanDocument) root).
```

Extending a Schema

The `<any>` binding may also be used to extend a schema rather than redefining its elements. For example, let's say we have a `Library` object that holds an `OrderedCollection` of `Book` objects. To represent this in XML, we could define the following two binding specifications:

```
<complexType name="Library">
 <sequence>
  <any namespace="##any" maxOccurs="unbounded" />
 </sequence>
</complexType>
<complexType name="Book">
 <sequence>
  <element name="title" minOccurs="0" type="xsd:string" />
 </sequence>
```

```
</complexType>
<element name="ownedHoldings" type="tns:Book" />
```

Here, the content of Library is specified to be <any>. In this way, we could later add a new binding for ShipPlan, and also add these to the Library.

```
<complexType name="ShipPlan">
 <sequence>
  <element name="shipName" minOccurs="0" type="xsd:string" />
 </sequence>
</complexType>
<element name="retainedHolding" type="tns:ShipPlan" />
```

Marshaling XML <anyAttribute> Elements

The <anyAttribute ...> element can appear only within a <complexType> or an <attributeGroup> binding. Class AnyAttributeMarshaler is used to process this element.

The AnyAttributeMarshaler marshaler allows the occurence of attributes from specified name spaces into a content model. The marshaler accepts and returns a collection of tagged objects as input or output values. A real marshaler for an object expected to be AttributeMarshaler with a tag that matches TaggedObject tag.

For example, to pass an object to an AnyAttributeMarshaler:

```
schema := '<?xml version="1.0" encoding="utf-8" ?>
 <schema targetNamespace="urn:AnyTypeTests" xmlns="http://www.w3.org/2001/
XMLSchema" xmlns:tns="urn:AnyTypeTests" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <complexType name="Library" >
   <sequence>
    <element name="account" minOccurs="0" type="xsd:string" />
   </sequence>
   <anyAttribute namespace="##any" processContents="strict" />
  </complexType>
  <attribute name="libraryName" type="xsd:string" />
  <attribute name="serviceApprovals" type="xsd:string" />
 </schema> '.
binding := XMLObjectDriver loadFrom:
    (XMLTypesParser
```

```
      useObjectBindingReadFrom: schema readStream
      inNamespace: 'Protocols.Library') printString readStream.
manager := XMLObjectMarshalingManager on: binding.
manager addToWildcardScope: 'urn:AnyTypeTests'.
(library := Library new)
 account: '12345';
 anyAttribute: (OrderedCollection
       with: (TaggedObject
          ns: 'urn:AnyTypeTests'
          type: 'libraryName'
          value: 'Port Nowhere Public Library')).
xml := manager marshal: library.
```

To pass arbitrary attributes to an AnyAttributeMarshaler:

```
schema := '<?xml version="1.0" encoding="utf-8" ?>
 <schema targetNamespace="urn:AnyTypeTest" xmlns="http://www.w3.org/2001/
XMLSchema" xmlns:tns="urn:AnyTypeTest" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
  <complexType name="Library" >
   <anyAttribute namespace="##any" processContents="skip" />
  </complexType>
 </schema>'.
binding := XMLObjectDriver loadFrom:
     (XMLTypesParser
      useObjectBindingReadFrom: schema readStream
      inNamespace: 'Protocols.Library') printString readStream.
manager := XMLObjectMarshalingManager on: binding.
(library := Library new)
 anyAttribute: (OrderedCollection
   with: (TaggedObject element:
    (XML.Attribute
     name: (NodeTag
        qualifier: ''
        ns: 'urn:AnyTypeTest'
        type: 'libraryName')
     value: 'Port Nowhere Public Library'))).
xml := manager marshal: library.
```

Wildcard Element Example

To illustrate the use of <any> and <anyAttribute> bindings, consider the following binding specification for the complex type Library:

```
schema := '<?xml version="1.0" encoding="utf-8" ?>
 <schema targetNamespace="urn:AnyTypeTest" xmlns="http://www.w3.org/2001/
XMLSchema" xmlns:tns="urn:AnyTypeTest" xmlns:xsd="http://www.w3.org/2001/
XMLSchema">
 <complexType name="Library">
  <sequence>
   <any namespace="##targetNamespace" />
  </sequence>
  <anyAttribute namespace="##targetNamespace" />
 </complexType>
 <attribute name="libraryName" minOccurs="1" type="xsd:string" />
 <attribute name="libraryId" required="true" type="xsd:string" />
 <element name="ownedHolding" type="tns:Book" />
 <element name="loanedHolding" type="tns:Book" />
 <complexType name="Book">
  <sequence>
   <element name="catalogNumber"
    minOccurs="0" type="xsd:string" />
   <element name="title" minOccurs="0" type="xsd:string" />
  </sequence>
 </complexType>
</schema>'.
```

With this binding specification, an XML instance of the complex type Library can include any element and attribute from the target name space.

Creating an X2O specification for the XML schema will map the <any> element to an instance of class AnyRelationMarshaler and the <anyAttribute> element to class AnyAttributeMarshaler.

```
x2o := XMLTypesParser
 useObjectBindingReadFrom: schema readStream
 inNamespace: 'Protocols.Library'.
```

The value of x2o should be:

```
<schemaBindings>
```

```
<xmlToSmalltalkBinding defaultClassNamespace="Protocols.Library"
targetNamespace="urn:AnyTypeTest" xmlns="urn:visualworks:VWSchemaBinding"
xmlns:tns="urn:AnyTypeTest" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<attribute name="libraryId" ref="xsd:string" />
<attribute name="libraryName" type="xsd:string" />
<element name="ownedHolding" ref="tns:Book" />
<element name="loanedHolding" ref="tns:Book" />
<object name="Library" smalltalkClass="Library">
 <sequence>
  <any name="any" namespace="##targetNamespace" />
 </sequence>
 <anyAttribute aspect="anyAttribute"
  namespace="##targetNamespace" />
</object>
<object name="Book" smalltalkClass="Book">
 <sequence>
   <element minOccurs="0" name="catalogNumber" ref="xsd:string" />
   <element minOccurs="0" name="title" ref="xsd:string" />
 </sequence>
</object>
</xmlToSmalltalkBinding>
</schemaBindings>
```

With the X2O specification, we can build marshalers:

```
binding := XMLObjectBinding loadFrom: x2o printString readStream.
```

When we marshal an instance of the `Library` object, we can provide `#loanedHolding` or `#ownedHolding` for the `#any` aspect, because an `AnyRelationMarshaler` accepts only the `ElementMarshalers` defined in this target name space. The `#anyAttribute` aspect will be marshaled by an `AnyAttributeMarshaler` that always accepts a collection. In the case of this schema, the attribute collection can include the two attributes `#libraryId` and `#libraryName`.

We can marshal the `Library` object with the `#ownedHolding ElementMarshaler` and `#libraryName` with an `AttributeMarshaler`:

```
book := Book new
    catalogNumber: 'XR67890';
    title: 'Distributed Algorithms';
    yourself.
attributes := OrderedCollection
```

```
 with: (TaggedObject
    ns: 'urn:AnyTypeTest'
    type: 'libraryName'
    value: 'Port Nowhere Public Library')
 with: (TaggedObject
    ns: 'urn:AnyTypeTest'
    type: 'libraryId'
    value: 'ABC123').
library := Protocols.Library.Library new.
library
 any: (binding tag: book withType: 'ownedHolding');
 anyAttribute: attributes.
xml := binding marshal: library.
```

Where `xml` is:

```
<Library ns:libraryId="ABC123" ns:libraryName="Port Nowhere Public Library"
 xmlns="urn:AnyTypeTest" xmlns:ns="urn:AnyTypeTest">
 <ownedHolding>
  <catalogNumber>XR67890</catalogNumber>
  <title>Distributed Algorithms</title>
 </ownedHolding>
</Library>
```

Alternatively, we can provide the `#any` aspect with loaned holdings:

```
library any: (binding tag: book withType: 'loanedHolding').
```

The marshaled xml will be:

```
<Library ns:libraryId="ABC123" ns:libraryName="Port Nowhere Public Library"
 xmlns="urn:AnyTypeTest" xmlns:ns="urn:AnyTypeTest">
 <loanedHolding>
   <catalogNumber>XR67890</catalogNumber>
   <title>Distributed Algorithms</title>
  </loanedHolding>
</Library>
```

Marshaling XML <choice> Elements

The `<choice>` binding is similar to `<any>`, except that the choice relation is limited to a set of types. The marshaler is `ChoiceMarshaler`, which holds a registry of available marshalers.

For example, consider the following 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>
```

Using this schema, marshaling and unmarshaling will be done based on three types: `Person1Type`, `Person2Type` and `string`.

Marshaling XML <group> and <attributeGroup> Elements

The `<group .../>` and `<attributeGroup .../>` bindings specify a group of elements, and a set of attribute declarations, to be used in complex type definitions.

When we create a X2O 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 X20 binding above should be prepared as follows:

```
anObj := WebServices.Struct new.
anObj
 group1:'astring';
 aaa: (WebServices.Struct new
   grAttr1: 'grAttr1';
   grAttr2: 'grAttr2';
   aaaName: 'aaName';
   yourself).
```

Marshaling a <group> with Unbounded Cardinality

A `<group .../>` binding may have unbounded cardinality, e.g.:

```
<xsd:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns="urn:schemaWithUnboundedGroups"
 targetNamespace="urn:schemaWithUnboundedGroups">
  <xsd:group name="type">
   <xsd:sequence>
    <xsd:element name="count" type="xsd:int" />
    <xsd:element name="delimiter" type="xsd:string" />
   </xsd:sequence>
  </xsd:group>
  <xsd:group name="aa">
   <xsd:sequence>
    <xsd:element name="a1" type="xsd:string" />
    <xsd:element name="a2" type="xsd:string" />
   </xsd:sequence>
  </xsd:group>
  <xsd:complexType name="CategoryType">
   <xsd:attribute name="attr" type="xsd:string" />
   <xsd:sequence>
    <xsd:group minOccurs="0" maxOccurs="unbounded" ref="aa" />
    <xsd:group minOccurs="0" maxOccurs="unbounded" ref="type" />
   </xsd:sequence>
  </xsd:complexType>
```

For this schema, the `XMLTypesParser` creates a X2O binding specification, and assigns aspects for each `<group>` element; in this case, `groupValue` and `groupValue1`:

```
x2o := XMLTypesParser
    useObjectBindingReadFrom:
```

```
     XMLSchemas schemaWithUnboundedGroups readStream
    inNamespace: 'WebServices'.
```

Where the X2O specification `#schemaWithUnboundedGroups` is:

```
x2o := '<schemaBindings>
 <xmlToSmalltalkBinding
  defaultClassNamespace="WebServices"
```

```
 targetNamespace="urn:schemaWithUnboundedGroups"
 xmlns="urn:visualworks:VWSchemaBinding"
 xmlns:tns="urn:schemaWithUnboundedGroups"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <modelGroup name="type">
  <sequence>
   <element name="count" ref="xsd:int" />
   <element name="delimiter" ref="xsd:string" />
  </sequence>
 </modelGroup>
 <modelGroup name="aa">
  <sequence>
   <element name="a1" ref="xsd:string" />
   <element name="a2" ref="xsd:string" />
  </sequence>
 </modelGroup>
 <object name="CategoryType" smalltalkClass="CategoryType">
  <attribute name="attr" ref="xsd:string" />
  <sequence>
   <group
    aspect="groupValue"
    maxOccurs="unbounded" minOccurs="0"
    ref="tns:aa" />
   <group
    aspect="groupValue1"
    maxOccurs="unbounded"
    minOccurs="0"
    ref="tns:type" />
  </sequence>
 </object>
</xmlToSmalltalkBinding>
</schemaBindings>'.
binding := XMLObjectBinding loadFrom: x2o readStream.
manager := XMLObjectMarshalingManager on: binding.
```

Unbounded groups are mapped to a collection of `Struct` objects.

To marshal the `CategoryType` object, we should prepare it as follows:

```
category := WebServices.CategoryType new
 groupValue:
  (OrderedCollection
   with: (WebServices.Struct new
     at: #a1 put: 'a1';
```

```
      at: #a2 put: 'a2';
      yourself)
   with: (WebServices.Struct new
      at: #a1 put: 'a11';
      at: #a2 put: 'a22';
      yourself));
 groupValue1:
  (OrderedCollection
   with: (WebServices.Struct new
      at: #count put: 123;
      at: #delimiter put: 'delimiter';
      yourself));
 attr: 'attr';
 yourself.
document := manager marshal: category.
```

Finally, the XML document is encoded as follows:

```
<ns:CategoryType attr="attr" xmlns:ns="urn:schemaWithUnboundedGroups">
 <a1>a1</a1>
 <a2>a2</a2>
 <a1>a11</a1>
 <a2>a22</a2>
 <count>123</count>
 <delimiter>delimiter</delimiter>
</ns:CategoryType>
```

Marshaling Collections

Class CollectionObjectMarshaler marshaler converts a sequence or XML elements and/or their parts to a collection of Smalltalk objects, and vice versa. The sequence_of tag introduces a homogeneous collection of items with known types or type ANY.

Describing a 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 description2</details>
</document>
```

the unmarshaled Smalltalk object would be:

```
doc := SoapDocument new.
doc detailsCollection:
  (OrderedCollection
    with: 'some description1'
    with: 'some description2').
```

Describing a 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 description2</item>
 </details>
</document>
```

the unmarshaled smalltalk object would be:

```
doc := SoapDocument new.
doc detailsCollection:
  (OrderedCollection
    with: 'some description1'
    with: 'some description2').
```

Describing a 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 document, the soapArray is usually described as a 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 a X2O 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 description2</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>

Class KeyObjectMarshaler marshals and unmarshals between a XML <key> element and a Smalltalk object.

In some cases we would like to relate a 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 that of XML Schema, 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 name space is the target name space 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 a 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 the 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. "resolved to association key (keyObject)"
```

Preserving Object Identity

The VisualWorks web services implementation provides a means to preserve object identity when marshalling. This is useful or necessary when working with complex objects that have circular references, or a single object which is referenced by several others, or a collection in which several items are identical.

VisualWorks provides several different options for encoding identical objects: id/href, id/ref and customer specified attributes.

WSDL supports XML schema encoding (#literal) and SOAP encoding (#encoded). If a WSDL document uses the latter #encoded style, identical objects can be encoded using id and href attributes. In SOAP 1.1, even identical strings can be encoded in this fashion.

Class XMLObjectBinding provides a class-side method useReference:, which enables your application to use the id and href attributes for this encoding. To enable it, send #useReference: true to an instance of XMLObjectBinding.

Note: In release 7.9 of VisualWorks, the default value of this option has been changed to false. Note that with this default, objects that contain cyclical structures can cause an infinite loop during the process of marshaling to XML.

In SOAP 1.2, XML Schema encoding is used for identical objects. For details, see the following:

http://www.w3.org/TR/2007/REC-soap12-part2-20070427/#uniqueids

To illustrate this mechanism, let's say that we want to marshal a collection of Book objects, such that several books are written by the same author. For this example, we use some of the predefined classes in the Library Demo, and the following schema:

```
schema := '<schemaBindings>
<xmlToSmalltalkBinding name="BookEample"
 xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
 useReference="true"
 xmlns:tns="urn:bookExample"
 targetNamespace="urn:bookExample"
 defaultClassNamespace="Protocols.Library">
<object name="authorialName" smalltalkClass="AuthorialName">
 <element name="name" type="xsd:string" />
</object>
<object name="Holding" abstract="true" smalltalkClass="Holding">
 <sequence>
  <element name="libraryName" ref="xsd:string" />
 </sequence>
</object>
<object name="book" smalltalkClass="Book" baseType="tns:Holding"
 constraint="extension">
 <sequence>
  <element name="title" ref="xsd:string" />
  <element name="authors" aspect="authors" minOccurs="1" maxOccurs="*" />
  <element name="libraryName" ref="xsd:string" />
  <element name="catalogNumber" ref="xsd:int" />
```

```
 </sequence>
</object>
<object name="library" smalltalkClass="Library">
 <sequence>
  <element name="ownedHoldings" maxOccurs="*" ref="tns:Holding" />
 </sequence>
</object>
</xmlToSmalltalkBinding>
</schemaBindings>'.
```

Next, we prepare the collection of `Book` objects and place them in a `Library` object:

```
author := AuthorialName new name: 'Alfred Jarry'; yourself.
firstBook := (Book new)
     title: 'Days and Nights';
     authors: (Array with: author);
     libraryName: 'City Public Library';
     catalogNumber: 100;
     yourself.
secondBook := (Book new)
      title: 'King Ubu';
      authors: (Array with: author);
      libraryName: 'City Public Library';
      catalogNumber: 101;
      yourself.
library := Protocols.Library.Library new
   ownedHoldings: (OrderedCollection with: firstBook with: secondBook);
   yourself.
```

Finally, we parse the schema and marshal the `Library` object:

```
manager := WebServices.XMLObjectMarshalingManager on:
     (WebServices.XMLObjectDriver
       loadFrom: schema readStream).
element := manager marshal: library.
```

If we unmarshal the XML and compare the author of each book, they should be idential (note that a book may have several `authors`, so we only select the `first`):

```
library := manager unmarshal: element.
firstBook := library ownedHoldings first.
```

```
secondBook := library ownedHoldings last.
firstBook authors first == secondBook authors first.
```

More examples can be found in the package WebServices-Core-Tests. Specifically, in the #testing id-href-ref category of class XMLMarshalingTests. For a different example that uses Microsoft encoding, look at the method #testMicrosoftDefinedIdRef in class WsdlClassBuilderTest in the WSDLToolTest package.

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.

Class 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.

The method add: is implemented using double-dispatch as:

```
add: child
 child addTo: self
```

The method 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 marshaler registers itself with its parent by sending register: to the builder. The builder then sends add: to its parent (the top element on the stack).

Registering a Marshaler

Once you have created a new marshaler, you need to add it to the marshaler registry, BindingBuilder.Registry. This is a Dictionary with element tag names as keys and marshaler class names 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 using the tag "foo". Register it by sending:

```
BindingBuilder.Registry
 at: 'foo'
 put: Bar.
```

The marshaler class Bar needs to initialize its XPath axis in an initialize method. If the marshaler Bar is defined for a few XML elements the method newProxyFor: should be used to initialize the marshaler instance. The parameter to newProxyFor: is the key value from BindingBuilder.Registry. In the example above, the parameter is 'foo'.

As an example, see the instance-side initialize method in class RestrictionMarshaler, and how multiple XML schema components are registered with RestrictionMarshaler.

Marshaling Exceptions

The following exceptions may be raised during marshaling and/or unmarshaling, to allow higher-level code decide what to do.

```
XMLObjectBindingSignal
 ClassNotDefinedSignal
 MissingRequiredHeader
 MissingValueSignal
 NilValueNotAllowed
 NoMarshalerSignal
 ObjectNotResolvedSignal
 UnresolvedReferenceSignal
 ValidationError
 WrongObjectType
 XMLDatatypeError
  XMLDecodingError
   DecodedIntegerOutOfRange
   DecodedInvalidString
  XMLEncodingError
   EncodingIntegerOutOfRange
   EncodingInvalidString
```

ClassNotDefinedSignal

Raised by the compex object marshaler to signal that there is no Smalltalk class for the XML attribute `smalltalkClass`.

MissingRequiredHeader

Raised if a required Soap header is missing.

MissingValueSignal

Indicates that there is no value for a relation marshaler to marshal or unmarshal.

NoMarshalerSignal

Indicates that no known marshaler is available for marshaling or unmarshaling.

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 this exception to indicate that it cannot resolve type elements.

ValidationError

Raised by the simple object marshaler to signal failed validation.

XMLDatatypeError

Is the superclass for errors 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.

XMLDecodingError

Raised while decoding XML strings into objects (simple types: arithmetic, boolean, date, time, url, byte) if there is a problem to decode a XML string into a corresponding Smalltalk object.

XMLEncodingError

Raised while encoding objects (simple types: arithmetic, boolean, date, time, URL, byte) into XML strings if an object type is different than a XML schema type.

DecodedIntegerOutOfRange

Raised if a XML string decoded into an integer that is out of the expected XML datatype range. The default action for DecodedIntegerOutOfRange returns a parameter which is a XML string with encoded integer value and the decoded integer value.

EncodingIntegerOutOfRange

Raised if the integer value to be encoded is outside of the range of the expected XML datatype. The default action for EncodingIntegerOutOfRange returns an integer encoded as a XML string.

DecodedInvalidString

Raised if a decoded string has invalid characters (violating the restrictions of the XML string type). The error is resumable

and the default action returns the XML string as it was received.

EncodingInvalidString

Raised if a string to be encoded includes invalid characters (violating the restrictions of the XML string datatype). The error is resumable and expects a corrected string as the resumption value.

Index