Artix™

Getting Started with Artix Encompass

Version 1.2, September 2003

Making Software Work Together™

Updated: 17-Oct-2003

M 3 1 1 0

Contents

List of Figures v

Preface vii

Chapter 1 Artix Encompass Concepts 1
- **Introduction to Artix Encompass** 2
- **The Elements of Artix** 4
  - The Artix Bus 5
  - Artix Service Access Points 6
  - Artix Contracts 7
- **The Artix Designer** 10

Chapter 2 Using Artix Designer to Build a C++ Web Service 21
- **The Web Service Project** 23
- **Using Artix Designer** 24
- **Starting Artix Designer** 28
- **Creating an Artix Designer Project** 31
- **Building the Widget Web Server** 36
- **Building the Widget Web Service Client** 39
- **Testing the Application** 42

Chapter 3 Using Artix Command Line Tools to Build a C++ Web Service 45
- **The Web Service Project** 46
- **Using Artix Encompass Tools** 47
- **Building the Widget Web Server** 51
- **Building the Widget Web Service Client** 53
- **Testing the Application** 55

Appendix A Implementation Code for the Widget Server and Client 57
- **Server Implementation Code** 58
- **Client Implementation Code** 60

Glossary 65

Index 69

List of Figures

Figure 1: The Artix Bus 4
Figure 2: Client-Server System Diagram 11
Figure 3: Artix Contract Editor 12
Figure 4: Editing a complexType 13
Figure 5: Adding Parts to a Message 14
Figure 6: Editing a PortType 15
Figure 7: Editing an Operation 16
Figure 8: Artix Service Editor 17
Figure 9: Editing the Properties of an HTTP Port 18
Figure 10: Development Tool 19
Figure 11: Deployment Tool 20
Figure 12: Welcome Screen 29
Figure 13: Artix Designer 30
Figure 14: Select Project Type 31
Figure 15: New project details 32
Figure 16: System Configuration 33
Figure 17: WSDL File Selection 34
Figure 18: Widget Service Starting Point 35
Figure 19: Widget Server Development Screen 37
Figure 20: Widget Client Development Screen 40

Preface

Overview

Getting Started with Artix Encompass gives a brief overview of Artix Encompass and provides a simple example of how to use Artix Encompass to solve a real world problem.

Audience

Getting Started with Artix Encompass is for anyone who needs to understand the concepts and terms used in IONA's Artix Encompass product, as well as anyone who needs to install Artix or maintain installed Artix systems.

Organization of this guide

This guide is divided as follows:

- "Artix Encompass Concepts" provides general information about Artix and how it is used.
- "Using Artix Designer to Build a C++ Web Service" presents a walk through of how to create a C++ Web service with the Artix Designer.
- "Using Artix Command Line Tools to Build a C++ Web Service" presents a walk through of the same scenario using the Artix command line tools.

Related documentation

The document set for IONA Artix includes the following:

- *Getting Started With Artix*
- *Artix User's Guide*
- *Artix Installation Guide*
- *Artix Tutorial*

- *Artix C++ Programming Guide*
- *Artix Security Guide*

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs/artix/1.2/index.xml.

Online help

Artix includes comprehensive online help, providing:

- Detailed step-by-step instructions on how to perform important tasks.
- A description of each screen.
- A comprehensive index and glossary.
- A full search feature.
- Context-sensitive help.

The **Help** menu in Artix Designer provides access to this online help.

Reading path

If you are new to Artix, you should read the documentation in the following order:

1. *Getting Started with Artix Encompass*

   The getting started book describes the basic concepts behind Artix. It also provides details on installing the system and a detailed walk through for developing a C++ Web Service.

2. *Artix Tutorial*

   The tutorial guides you through programming Artix applications against all of the supported transports.

3. *The Artix Users' Guide*

   The users. guide describes the development pattern for designing and deploying Artix enabled systems. It provides detailed examples for a number of typical use cases.

4. *GUI Online Help*

   The Artix design tools have context sensitive on-line help the provides information specific to the tools that you are using.

5. *Artix C++ Programmer's Guide*

   The programmer's guide discusses the technical aspects of programming applications using the Artix C++ API.

Additional resources

The IONA knowledge base contains helpful articles, written by IONA experts, about Artix and other products. You can access the knowledge base at the following location:

The IONA update center contains the latest releases and patches for IONA products:

If you need help with this or any other IONA products, contact IONA at support@iona.com. Comments on IONA documentation can be sent to docs-support@iona.com .

Typographical conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`* **Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with *italic* words or characters.

Keying conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS or Windows command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

CHAPTER 1

Artix Encompass Concepts

Artix Encompass extends enterprise Quality of Service features to Web services applications and enables the rapid creation and deployment of EAI solutions using Web services technology.

In this chapter

This chapter discusses the following topics:

Introduction to Artix Encompass	page 2
The Elements of Artix	page 4
The Artix Designer	page 10

Introduction to Artix Encompass

Overview

Artix Encompass enables developers to expose existing application logic as Web services without changing the underlying middleware upon which they run or developers can write new C++ Web services. In addition, it provides enterprise levels of service such as session management, service look-up, security, and transaction propagation.

Encompass does this by leveraging IONA's proven Adaptive Runtime Technology (ART) platform to provide a high-speed, stable backbone for your Web service deployments. In addition, Encompass extends the ART platform and the Web service metaphor by using the Artix Bus, IONA's transport and payload format switching technology. The Artix Bus enables the creation of Web services that communicate using protocols other than SOAP over HTTP; Web services can be developed and deployed using proven enterprise quality communication mechanisms such as TIBCO Rendevous™, CORBA, and IBM Websphere MQ (formerly MQSeries).

Artix Encompass Features

Artix Encompass has the following unique features:

- Support for C++ Web service development
- Routing
- Transaction support for Web services
- Support for Asynchronous Web services
- Mainframe support for Web services
- Support for Web services to use multiple transports and message data formats
- Security support for Web services
- Support for stateful Web services
- Leasing for Web services
- Load-balancing
- Look-up services

Supported transports

Artix supports the following message transports:

- HTTP

- BEA Tuxedo
- IBM Websphere MQ
- TIBCO Rendezvous™
- IIOP Tunnel

Supported payload formats

Artix can automatically transform between the following payload formats:

- G2++
- FML – Tuxedo format
- CORBA (GIOP) – CORBA format
- FRL – fixed record length
- VRL – variable record length
- SOAP
- TibrvMsg - TIBCO Rendezvous format

The mapping of logical data items between payload formats is supported by Artix tools.

The Elements of Artix

Overview

Artix's unique features are implemented by a number of plug-ins to IONA's ART platform. These plug-ins form the core of Artix, the Artix Bus. Applications that make use of Artix connect to the Bus using Artix Services Access Points (SAPs). Service Access Points are described by Artix Contracts.

Figure 1 shows how all of the Artix elements fit together.

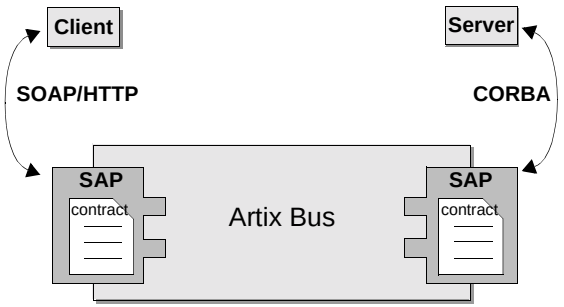

Figure 1: *The Artix Bus*

In this Section

This section discusses the following topics:

The Artix Bus	page 5
Artix Service Access Points	page 6
Artix Contracts	page 7

The Artix Bus

Overview

The Artix Bus is a set of plug-ins that work in much the same way as the simultaneous translators at the United Nations. The plug-ins read data that can be in a number of disparate formats, the Bus directly translates the data into another format, and the plug-ins write the data back out to the wire in the new format. In this way Artix enables all of the applications in your company to communicate over the Web without needing to understand SOAP or HTTP. It also means that clients can contact Web services without understanding the native language of the server handling requests.

Benefits

While other Web service suites provide some ability to expose enterprise applications as Web services, they frequently require a good deal of coding. The Artix bus eliminates the need to modify your applications or write code by directly translating between the enterprise application's native communication protocol and SOAP over HTTP, the prevalent protocol for Web services. For example, by deploying an Artix instance with a SOAP over Websphere MQ SAP and a SOAP over HTTP SAPoint, you can expose a Websphere MQ application directly as a Web service. The Webshpere MQ application would not need to be altered or made aware that it was being exposed using SOAP over HTTP.

The Artix Bus' translation ability also makes it a powerful integration tool. Unlike Enterprise EAI applications, Artix translates directly between different middlewares without first translating into a canonical format. This saves processing and increases the speed at which messages are transmitted through the Bus.

Artix Service Access Points

Overview

An Artix Service Access Point (SAP) is where a service provider or service consumer connects to the Artix Bus. SAPs are described by a contract describing the services offered and the physical representation of the data on the network.

Reconfigurable connection

In essence, an SAP provides an abstract connection point between applications. The benefit of using this abstract connection is that it allows you to change the underlying communication mechanisms without recoding any of your applications. You simply need to modify the contract describing the SAP. For example, if one of your backend service providers is a Tuxedo application and you want to swap out Tuxedo for a CORBA implementation, you would simply change the SAP's contract to contain a CORBA connection to the Bus. The clients accessing the backend service provider never need to be aware that the application has changed.

Artix Contracts

Overview

The Web Services Definition Language (WSDL) is used to describe the characteristics of the Service Access Points (SAPs) of an Artix connection. By defining characteristics like service operations and messages in an abstract way -- independent of the actual transport or protocol used to implement the SAP -- these characteristics can be bound to a variety of a specific protocols and formats. In fact, Artix allows an abstract definition to be bound to multiple specific protocols and formats. This means that the same definitions can be reused in multiple implementations of a service.

Artix contracts define the services exposed by a set of systems, the payload formats and transports available to each system, and the rules governing how the systems interact with each other. The most simple Artix contract defines a set of systems with a shared interface, payload format, and transport. Artix contracts, however, can define very complex integration scenarios.

WSDL concepts

Understanding Artix contracts requires some familiarity with WSDL, including the definitions of the following terms:

WSDL types provide data type definitions used to describe messages.

A WSDL message is an abstract definition of the data being communicated and each part of a message is associated with defined types.

A WSDL operation is an abstract definition of the capabilities supported by a service, and is defined in terms of input and output messages.

A WSDL portType is a set of abstract operation descriptions.

A WSDL binding associates a specific protocol and data format for operations defined in a portType.

A WSDL Port specifies a network address for a binding, and defines a single communication endpoint.

A WSDL service specifies a set of related ports.

The Artix contract

An Artix contract is specified in WSDL and conceptually divided into logical and physical components.

The logical contract specifies things that are independent of the underlying transport and wire format; it fully specifies the data structure and the possible operations or interactions with the interface. It allows Artix to generate skeletons and stubs without having to define the physical characteristics of the connection (wire format and transport).

The physical component of an Artix contract defines:

- The wire format, middleware transport, and service groupings
- The connection between the PortType 'operations' and wire formats
- Buffer layout for fixed formats
- Artix extensions to WSDL

Example 1: *Artix WSDL Contract Elements*

```
Logical Contract:

<Schema>
<Type>          (analogous to typedefs)
<Message>       (analogous to parameter)
<PortType>      (analogous to class or CORBA interface definition)
<Operations>    (analogous to methods)

Physical Contract:

<Binding>       (payload format)
<Services>      (groups of ports)
<Port>          (transport addressing information)
<Route>         (rules governing system interaction)
```

Payload Formats

A payload format controls the layout of a message delivered over a transport. The WSDL definition of a Port and its binding together associate a payload format with a transport. A binding can be specified in the logical

portion of an Artix contract (`portType`), which allows for a logical contract to have multiple bindings and thus allow multiple on-the-wire formats to use the same contract.

The Artix Designer

Overview

The Artix Designer is a tool for creating and managing Artix contracts. It provides editors for creating contracts from standard WSDL files as well as from CORBA IDL files. The Designer also makes it easy to define new data types, logical interfaces, payload bindings, and transports by providing editors to walk you through each step.

The Artix Designer generates all of the Artix components you need to complete your project. These components include:

- Artix contracts describing each of the services in your system.
- An Artix contract describing how Artix integrates your services.
- Any Artix stub and skeleton code needed to write Artix application code.
- The needed configuration information to deploy your Artix instances.

In addition, the Artix Designer can also generate CORBA IDL from any contracts that have a CORBA binding.

System Diagram

The first screen you see when using the Artix Designer is the system diagram. The system diagram displays all of the services in your system and the Artix instances deployed to integrate the services. This diagram is updated as you add services and Artix instances to your system. Figure 2 shows a system diagram containing a client and server being integrated

using a standalone Artix instance.

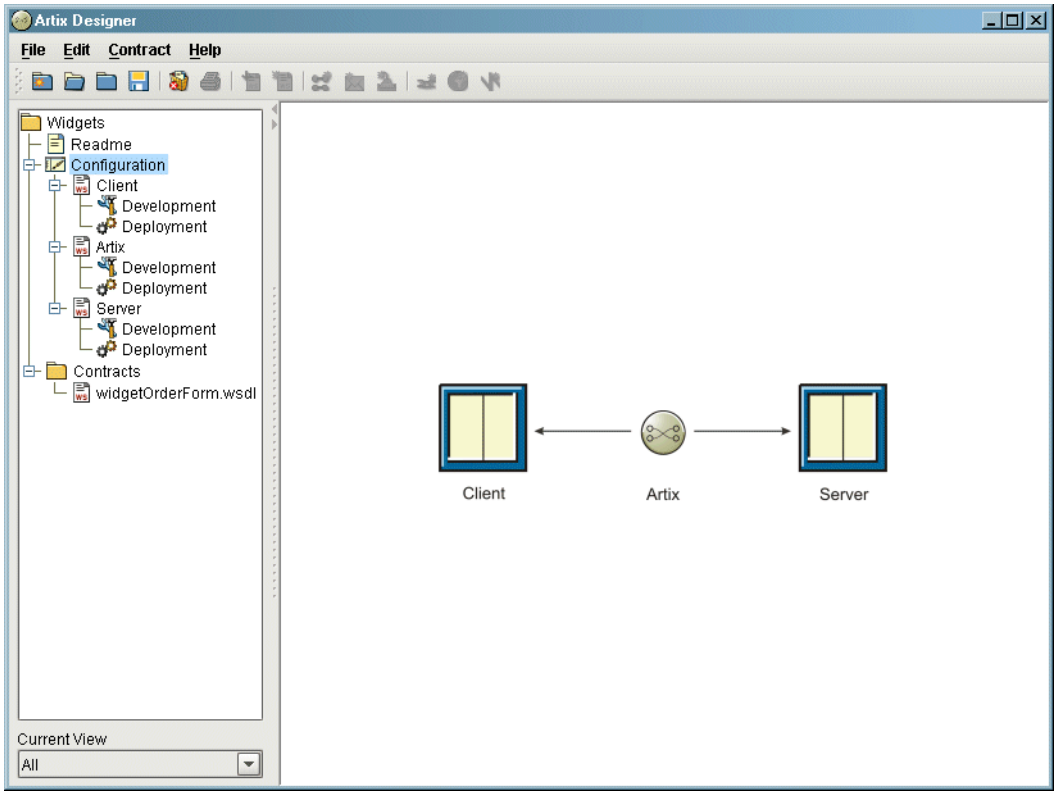

Figure 2: *Client-Server System Diagram*

Project Tree

To the left of the Designer's editor panel is the project tree. The project tree lists all of system diagram components with nodes for generating code, generating deployment information, and, if you are using CORBA, generating IDL. The project tree also lists all of the contracts imported into your project..

The drop down list at the bottom of the project tree panel controls the amount of detail shown in the tree at a time. The default is to show all the information about the project. You can chose to view only the contracts imported into the project or just the system components.

Contract Editor

The contract editor of the Artix Designer is where most of the work is done when developing an Artix project. As shown in Figure 3, the contract editor presents you with a graphical representation of an Artix contract. By selecting the different nodes in the diagram you bring up editors that allow you to add to or edit each of the parts of an Artix contract.

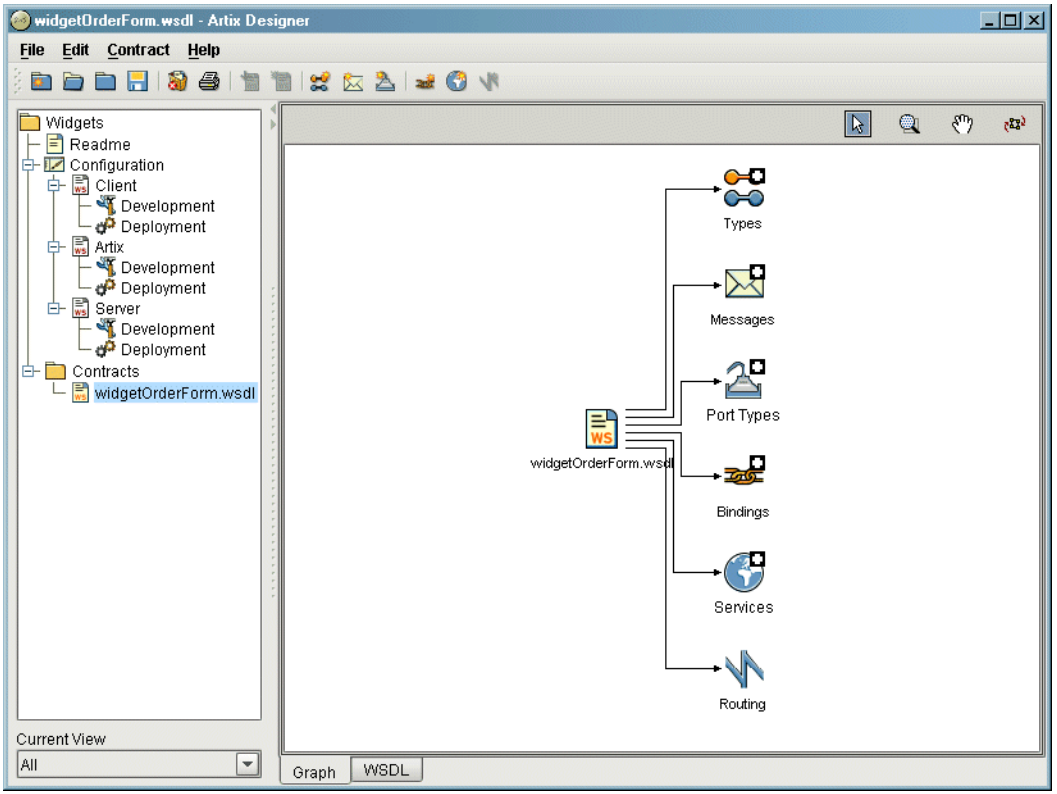

Figure 3: *Artix Contract Editor*

Type Editor

The type editor is invoked from the contract editor and allows you to create new logical types in your contract or modify existing types. When editing existing types, the editor screen is tailored to match the kind of data type you are editing. Figure 4 shows the screen for editing a `complexType`.

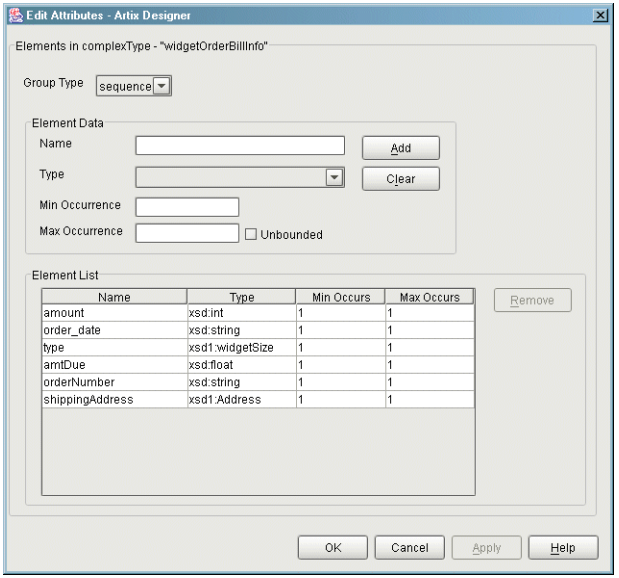

Figure 4: *Editing a complexType*

When adding a new type the editor walks you through the creation of your data type.

Message Editor

The message editor is invoked from the contract editor and allows you to add new messages to your contract and to edit existing messages. Using the editor you can add new parts to existing messages from the types existing in your contract and the editor ensures that there are no naming conflicts. Figure 5 shows the message editor's main dialog.

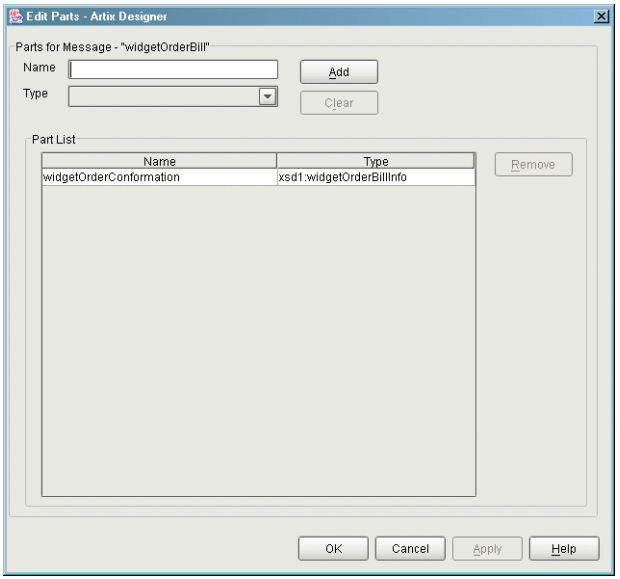

Figure 5: *Adding Parts to a Message*

Interface Editor

The interface editor, or PortType editor, is invoked from the contract editor and allows you to edit existing logical interfaces or add new logical interfaces. Logical interfaces are referred to as `portTypes` in a WSDL

document and the editor dialogs rely on WSDL terminology. The output of this editor will be entered in a `portType` element in your contract. Figure 6 shows the interface editor.

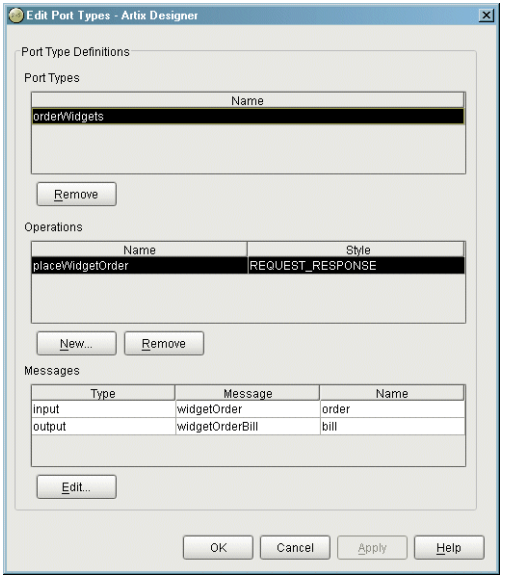

Figure 6: *Editing a PortType*

Operation Editor

The operation editor is part of the interface editor. It allows you to modify existing operations defined on the interface or to add new operations to the interface. When adding messages to an operation, the editor will only allow you to select from messages already defined in the contract. The editor also

checks for any naming conflicts. Figure 7 shows the operation editor.

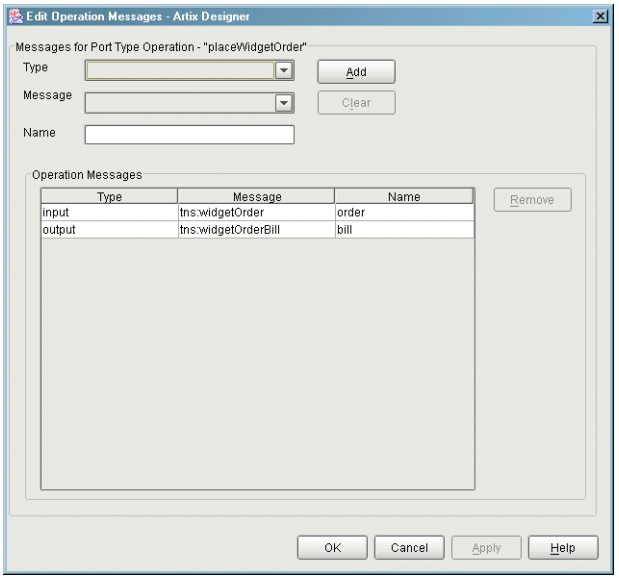

Figure 7: *Editing an Operation*

Binding Editor

The binding editor is invoked from the contract editor and allows you to map any interface described in your contract to one of the payload formats supported by Artix. The editor asks you to select the payload format and the interface. It then performs the mapping automatically.

Service Editor

The service editor is invoked from the contract editor and allows you to edit existing WSDL service definitions in your contract and to add new WSDL service definitions in your contract. As shown in Figure 8, the editor shows

you the name of service, the ports defined as part of the service, the transport used by the selected port, and any properties set on the selected port.

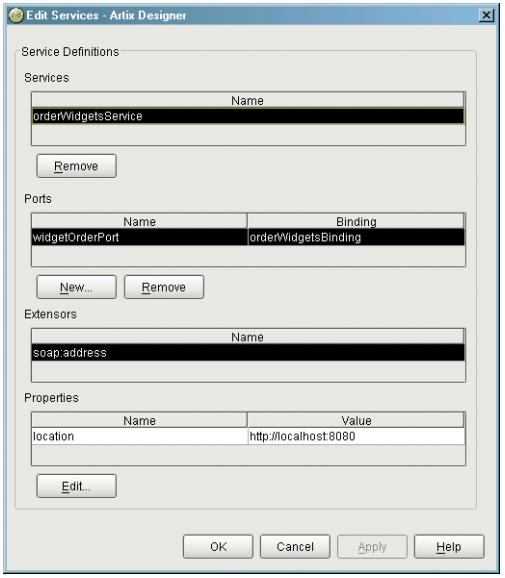

Figure 8: *Artix Service Editor*

Port Editor

The port editor is part of the service editor and it allows you to modify the properties of an existing port or add a new port to an existing service. It provides you with a list of properties you can set on each type of port Artix supports and ensures that the required values are supplied. Figure 9 shows the properties for an Artix HTTP port.

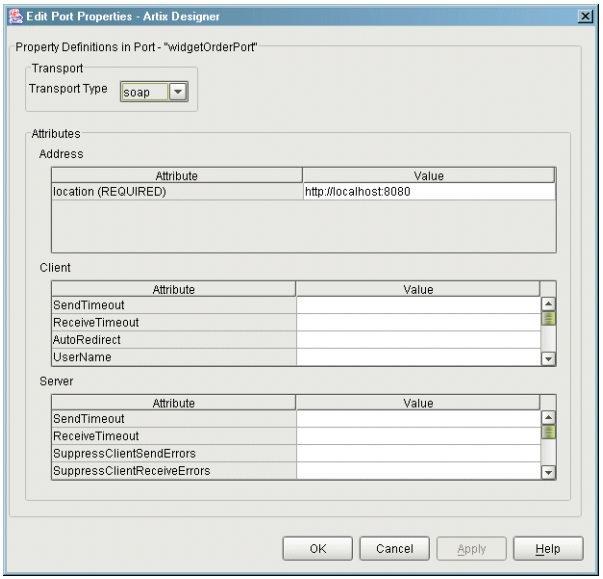

Figure 9: *Editing the Properties of an HTTP Port*

Routing Editor

The routing editor is invoked from the contract editor and allows you to create routes between compatible ports. For this editor to be used, your contract must have more than one port defined and the ports must be compatible. For a detailed discussion on port compatibility and routing see the *Artix Users' Guide*.

Development Tool

The development tool is invoked by selecting the **Development** icon under one of the services in the project tree. Using this tool, shown in Figure 10, you can generate Artix C++ stub and skeleton code for the interfaces defined by the selected service's contract. The tool will also generate a make file and sample server and client mainlines for you.

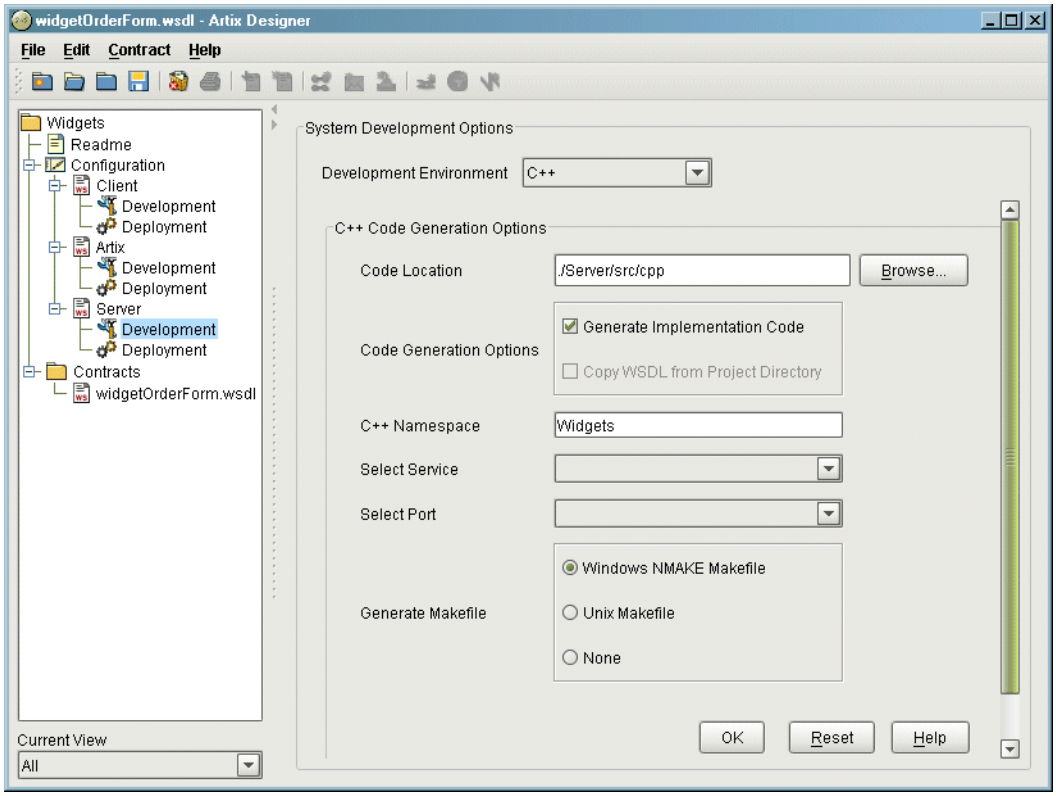

Figure 10: *Development Tool*

If the service's contract contains a CORBA binding, the development tool will also generate IDL describing the service's interfaces.

Deployment Tool

The deployment tool is invoked by selecting the **Deployment** icon under one of the services in the project tree. The deployment tool, show in Figure 11, generates an Artix configuration file that is optimized for the selected service, a script for setting up your Artix runtime environment, and a composite Artix contract that is suitable for deployment into a runtime system. The generated configuration file contains all of the information needed to deploy your service using Artix. In the case of a standalone Artix service the deployment tool also generates start and stop scripts for the Artix service.

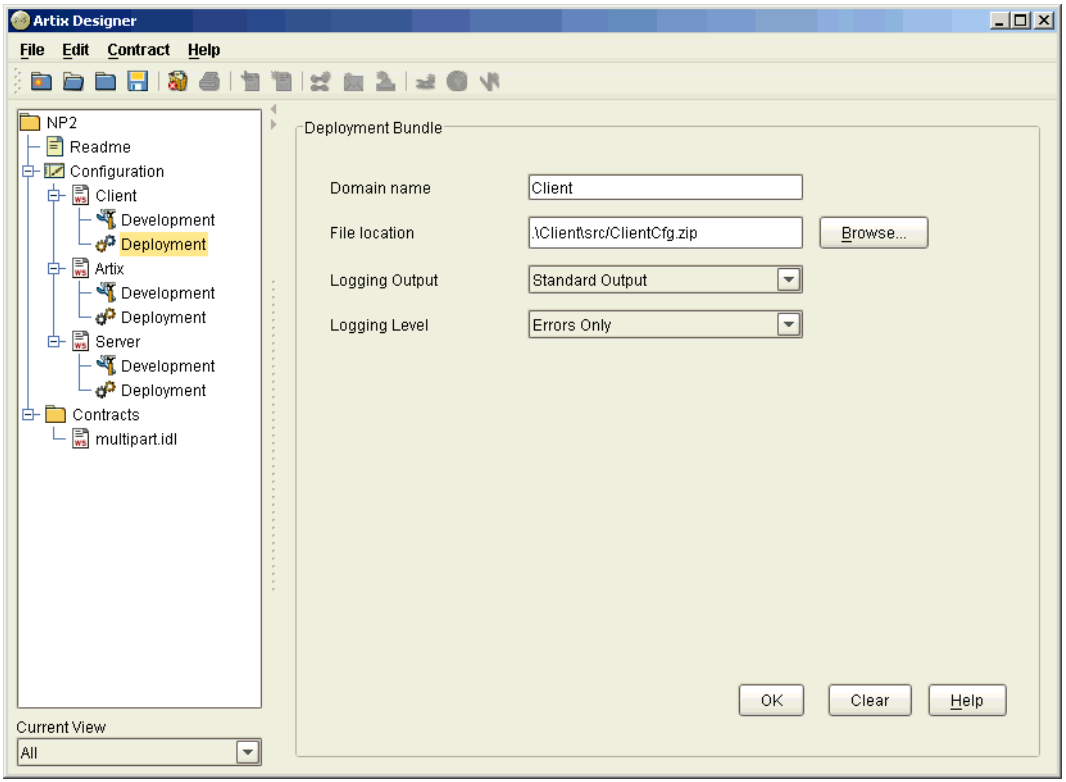

Figure 11: *Deployment Tool*

CHAPTER 2

Using Artix Designer to Build a C++ Web Service

Artix Encompass is a world class utility for building C++ Web services using simple, standard C++ programming techniques. It provides all the tools needed to edit WSDL contracts and generate starting point code for Web servers and their clients.

In this chapter

This chapter discusses the following topics:

The Web Service Project	page 23
Using Artix Designer	page 24
Starting Artix Designer	page 28
Creating an Artix Designer Project	page 31
Building the Widget Web Server	page 36

Building the Widget Web Service Client	page 39
Testing the Application	page 42

The Web Service Project

The problem scenario

Your company produces widgets and has decided to automate its ordering system to cut labor costs and reduce turnaround time. The new system will allow the company's customers to submit their orders electronically, will generate and send electronic bills to the customer, and generate a work order for your manufacturing system.

Your company's CIO has determined he wants this new system to be implemented using a Web service and that the development of both the server and the client will be done in-house. Unfortunately, your IT department doesn't have anybody with solid Java or Web services skills and there is no money or time to hire a new developer for this project.

How Artix simplifies solving the problem

Artix simplifies the solution to this problem by providing automated generation of the following:

- C++ server skeletons which allow developers to program using standard C++ metaphors
- C++ server implementation object method shells
- C++ client stubs which allow developers to program using standard C++ metaphors
- C++ server mainline starting point code
- C++ client mainline starting point code
- makefiles for Unix or Windows
- deployment descriptors for Web services

Using Artix Designer

Overview

Artix Designer provides a graphical environment in which to define your Web service's interfaces and the transports it will use. In this case, the problem is to define a service that receives an order and returns a bill. A full description of this service includes:

- The structure of the data the service sends and receives
- The operations offered by the service
- The order in which the data is encoded
- The payload format the service uses
- The transport the service uses
- The location of the service.

A Web service is defined in a WSDL document. Artix can import WSDL directly, and convert it into Artix contracts (which are themselves WSDL files that may include IONA-specific extensions). Even if a service description is less formal than an existing WSDL file (e.g., in the case where a service is under development), Artix Designer provides a series of wizards to guide you through the process of creating an Artix contract based on the information available.

The Web service description

For the purposes of this example we will use a predefined Widget service defined in Example 2.

Example 2: *Vendor WSDL document*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
 targetNamespace="http://widgetVendor.com/widgetOrderForm"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://widgetVendor.com/widgetOrderForm"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://widgetVendor.com/types/widgetTypes">
  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

Example 2: *Vendor WSDL document*

```
    <xsd:simpleType name="widgetSize">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="big"/>
        <xsd:enumeration value="large"/>
        <xsd:enumeration value="mungo"/>
        <xsd:enumeration value="gargantuan"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="Address">
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street1" type="xsd:string"/>
        <xsd:element name="street2" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="zipCode" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="widgetOrderInfo">
      <xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="shippingAddress" type="xsd1:Address"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="widgetOrderBillInfo">
      <xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="amtDue" type="xsd:float"/>
        <xsd:element name="orderNumber" type="xsd:string"/>
        <xsd:element name="shippingAddress" type="xsd1:Address"/>
      </xsd:sequence>
    </xsd:complexType>
  </schema>
</types>
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
</message>
```

Example 2: *Vendor WSDL document*

```
  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order"/>
      <output message="tns:widgetOrderBill" name="bill"/>
    </operation>
  </portType>
  <binding name="orderWidgetsBinding" type="tns:orderWidgets">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="placeWidgetOrder">
        <soap:operation soapAction="" style="rpc"/>
        <input name="widgetOrder">
          <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                     namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
        </input>
        <output name="widgetOrderBill">
          <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                     namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
        </output>
    </operation>
  </binding>
  <service name="orderWidgetsService">
    <port name="widgetOrderPort" binding="tns:orderWidgetsBinding">
      <soap:address location="http://localhost:8080"/>
    </port>
  </service>
</definitions>
```

This WSDL document completely describes the interface exposed by the Web service and the data that is passed to and from the server. Artix Designer can import this file directly and use it in the Artix contract that describes the entire integrated system you are building.

The major sections of the WSDL description are interpreted as follows:

`<types>`	Defines the complex data types used by the service. This service uses an enumerated type, `widgetSize`, to describe the widgets, a structure, `Address`, to hold the shipping address, and two structures, `widgetOrderInfo` and `widgetOrderBillInfo`, for the data needed to process the order.
`<message>`	Defines the messages by which the service communicates.
`<portType>`	Defines the operations offered by the service.

`<binding>`	Describes how the service expects its data to be formatted. In this case, it formats the data using SOAP.
`<service>`	Defines the address where the service can be contacted.

Starting Artix Designer

Overview

Artix Designer is a suite of tools for developing Artix solutions and managing Artix projects.

Windows

On a Windows system you can start Artix Designer from the **Start** menu. Select **Programs|IONA|Artix|Artix Designer**. You can also start Artix Designer from the command line with the following command:

```
start_designer
```

The executable for this command is installed in the following directory:

```
%IT_PRODUCT_DIR%\artix\1.2\bin
```

UNIX

On a UNIX system you must start Artix Designer from the command line. To start Designer, complete the following steps:

1. Run `%IT_PRODUCT_DIR%\artix\1.2\bin\artix_env` to source the Artix environment.
2. Run `%IT_PRODUCT_DIR%\artix\1.2\bin\start_designer` to start the GUI.

Once the GUI is running

1. Select **Go straight to designer** on the welcome screen shown in Figure 12.

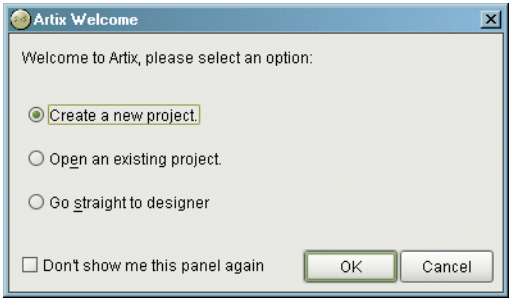

Figure 12: *Welcome Screen*

2. You will see a screen like Figure 13.

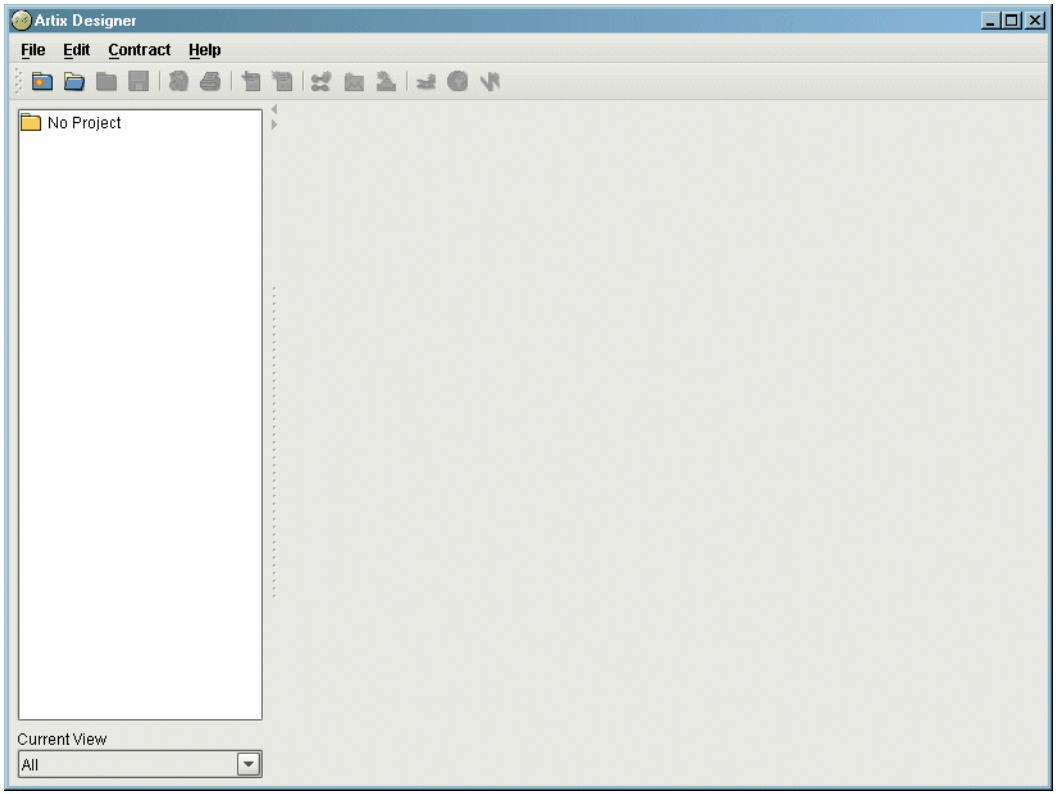

Figure 13: *Artix Designer*

Creating an Artix Designer Project

Overview

An Artix project consists of one or more Artix contracts, a system design diagram, and a number of source code files. Artix Designer creates a special directory and project structure to manage these artifacts.

Procedure

To create a new Artix Designer project complete the following steps:

1. Create a new Artix project by selecting **New|Project** from the designer's **File** menu.
2. You will see a screen like Figure 14.

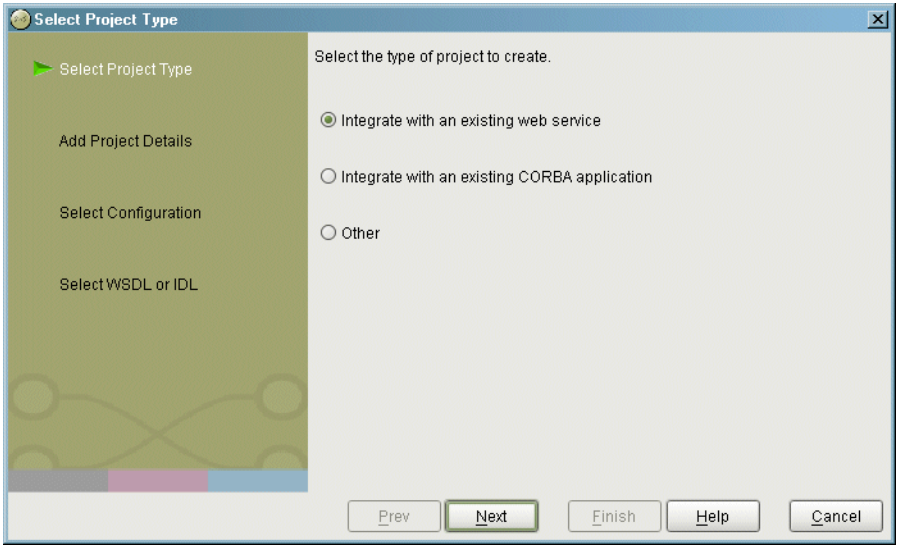

Figure 14: *Select Project Type*

3. Select **Integrate with an existing web service**.
4. Click **Next**.

5. You will see a screen like Figure 15.

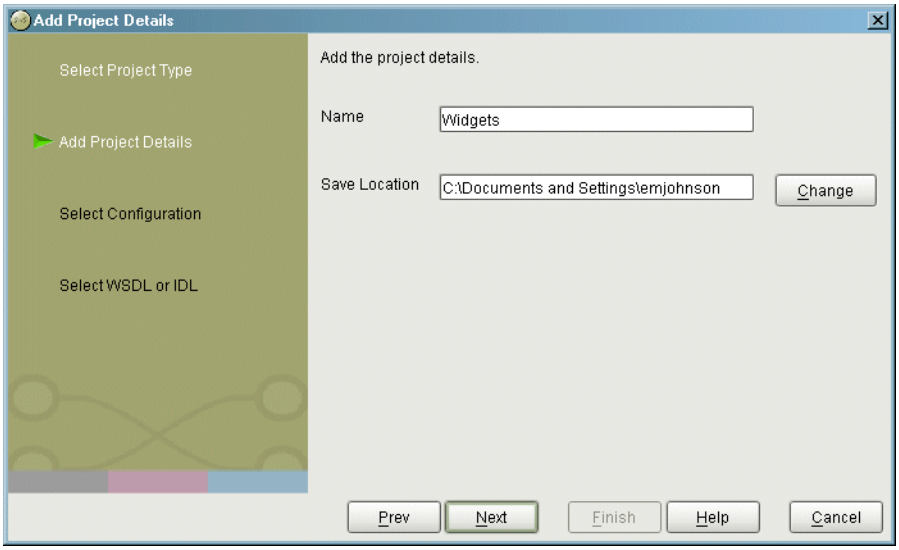

Figure 15: *New project details*

6. Type **`Widgets`** in the **Name** field.
7. Click **Change**.
8. Using the file navigation dialog box, navigate to your home directory and click **Select Project Directory**.
9. Click **Next**.

10. A screen like that shown in Figure 16 appears:.

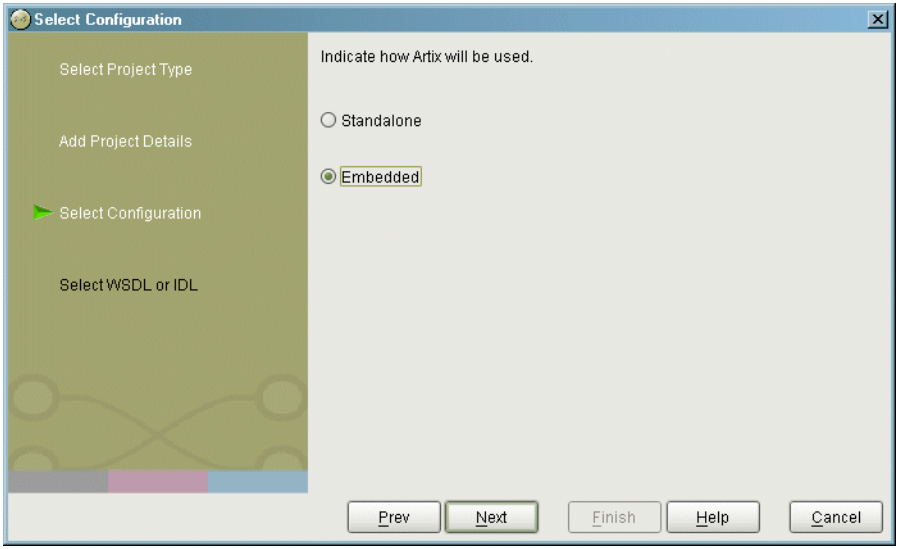

Figure 16: *System Configuration*

11. Select **Embedded**.
12. Click **Next**.

13. You will see a screen like Figure 17.

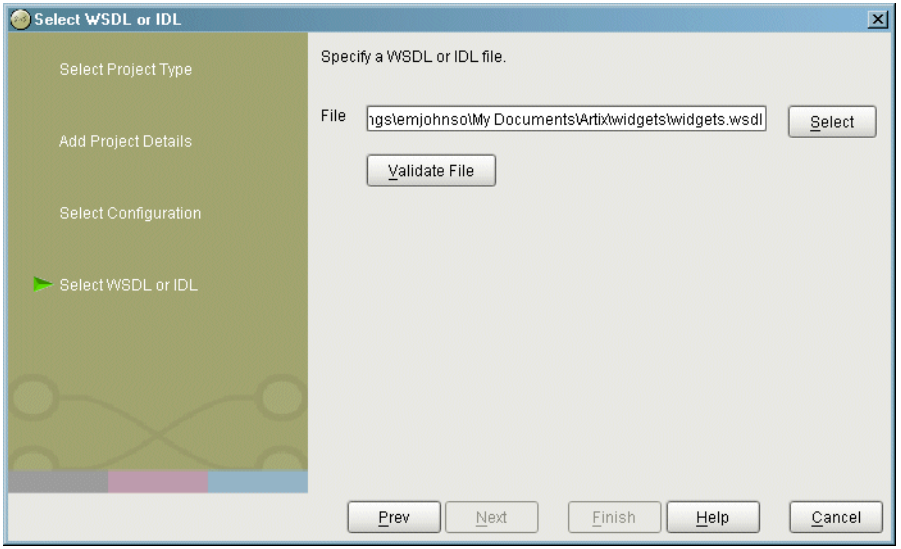

Figure 17: *WSDL File Selection*

14. Click the **Select** button.
15. Using the file navigation dialog box, navigate to your Artix installation directory.
16. Under your Artix installation directory, locate the `demos/widgets` directory.
17. Select `widgets.wsdl` from the file selection box.
18. Click the **Validate File** button.
19. When **Finish** becomes available, click it to create your project.
20. The Designer screen now looks like Figure 18.

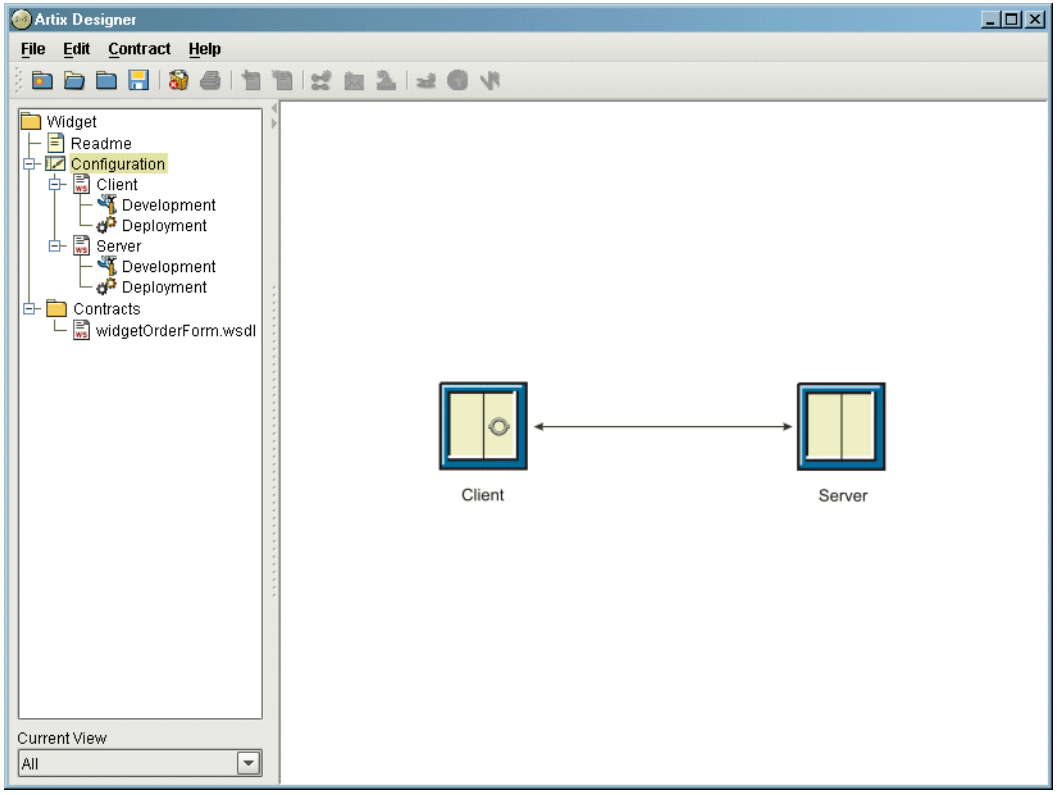

Figure 18: *Widget Service Starting Point*

Building the Widget Web Server

Overview

Artix Designer generates server stubs for any of the contracts used to describe a component of your integration project. In addition, the designer generates a sample server mainline, and generates a makefile to build the server.

Once Artix Designer generates the stub code, you must write the implementation logic using the C++ development environment of your choice.

Procedure

To develop the widget web server using Artix Designer complete the following steps:

1. Select the **widgetOrderForm** contract from the **Contracts** folder of the project tree.
2. Drag the contract to the **Server** node under the **Configuration** folder on the project tree.
3. A copy of the contract will appear under the **Server** node.
4. Select the **Development** icon under the **Server** node in the project tree.

5. You will see a screen similar to Figure 19.

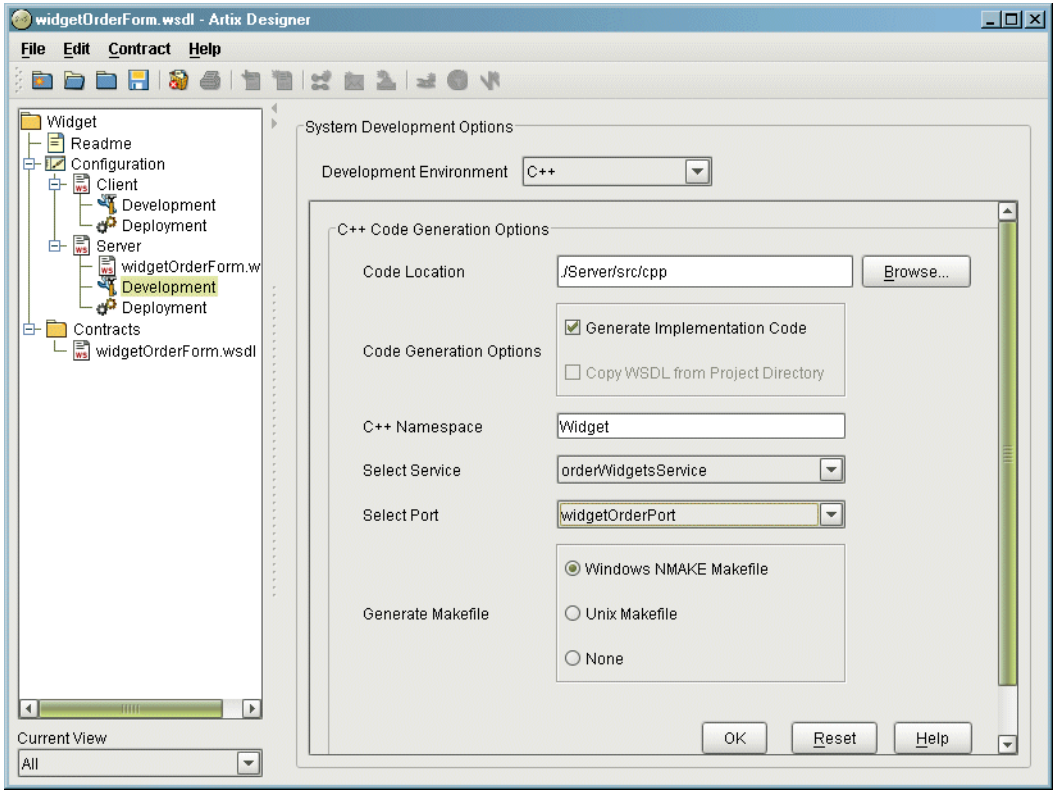

Figure 19: *Widget Server Development Screen*

6. Select **C++** from the **Development Environment** pull-down list.
7. Enter **`WidgetServer`** for the **C++ Namespace**.
8. Select the appropriate type of makefile generation for your platform.
9. Select **orderWidgetsService** from the **Select Service** pull-down list.
10. Select **widgetOrderPort** from the **Select Port** pull-down list.
11. Click **OK**.

12. The following files are generated in the `Server/src/cpp` directory of your project folder:

`orderWidgets.h`	`orderWidgetsClient.cxx`
`orderWidgetsClient.h`	`orderWidgetsImpl.cxx`
`orderWidgetsImpl.h`	`orderWidgetsServer.cxx`
`orderWidgetsServer.h`	`SampleClient.cxx`
`SampleServer.cxx`	`Makefile`
`Server_wsdlTypesFactory.cxx`	`Server_wsdlTypesFactory.h`
`widgets_wsdlTypes.cxx`	`widgets_wsdlTypes.h`

    For the purposes of generating a Web server to implement the widget ordering system, you do not need any of the client, `*Client.*`, source files.

13. Insert the highlighted code shown in "Server Implementation Code" on page 58, to `orderWidgetsImpl.cxx` to add the application logic to the server.
14. Build the server.

UNIX

```
make server.exe
```

Windows

```
nmake server.exe
```

Building the Widget Web Service Client

Overview

Artix-generated proxy classes integrate smoothly into a standard C++ application. To use an Artix proxy you must initialize the Artix Bus and then instantiate an instance of the proxy class. Once instantiated the proxy object provides all of the functionality of the server through standard invocations of its methods.

Procedure

To develop the widget web service client using Artix Designer complete the following steps:

1. Select the **widgetOrderForm** contract from the **Contracts** folder of the project tree.
2. Drag the contract to the **Client** node under the **Configuration** folder on the project tree.
3. A copy of the contract will appear under the **Client** node.
4. Select the **Development** icon under the **Client** node in the project tree.

5. You will see a screen similar to Figure 20.

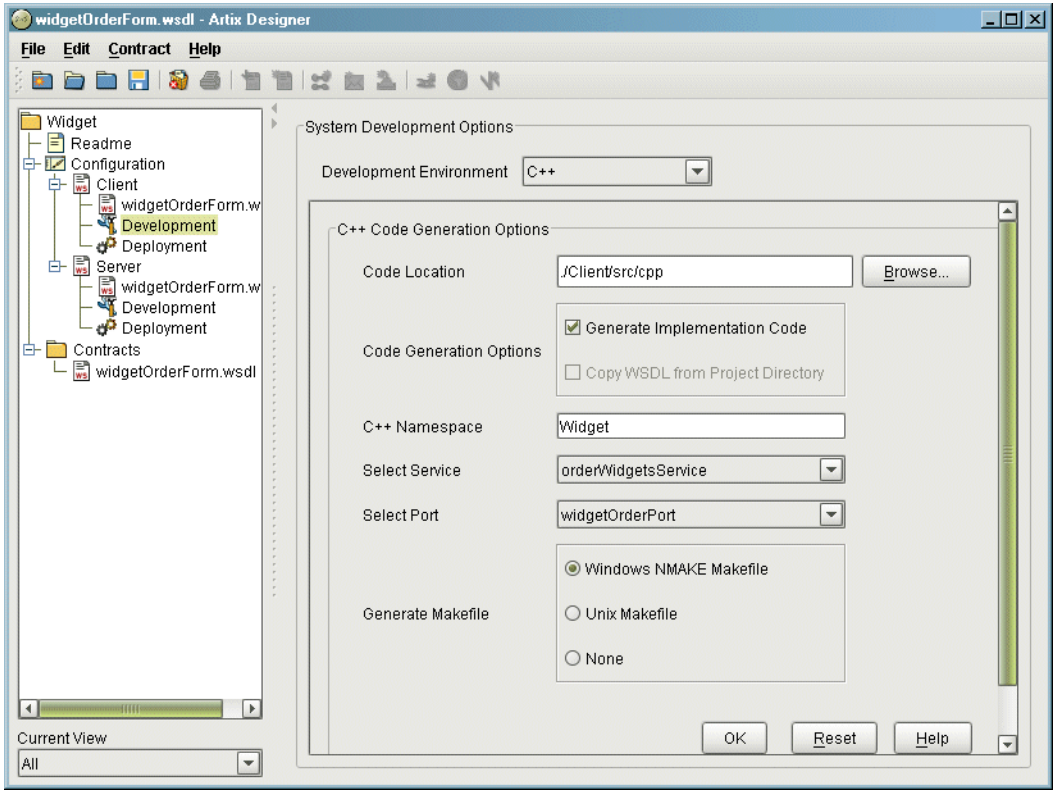

Figure 20: *Widget Client Development Screen*

6. Select **C++** from the **Development Environment** pull-down list.
7. Enter **`WidgetClient`** for the **C++ Namespace**.
8. Select the appropriate type of makefile generation for your platform.
9. Select **orderWidgetsService** from the **Select Service** pull-down list.
10. Select **widgetOrderPort** from the **Select Port** pull-down list.
11. Click **OK**.

12. The following files are generated in the `Clinet/src/cpp` directory of your project folder:

`orderWidgets.h`	`orderWidgetsClient.cxx`
`orderWidgetsClient.h`	`orderWidgetsImpl.cxx`
`orderWidgetsImpl.h`	`orderWidgetsServer.cxx`
`orderWidgetsServer.h`	`SampleClient.cxx`
`SampleServer.cxx`	`Makefile`
`Client_wsdlTypesFactory.cxx`	`Client_wsdlTypesFactory.h`
`widgets_wsdlTypes.cxx`	`widgets_wsdlTypes.h`

    For the purposes of generating a web service client to interact with the widget web server, you do not need any of the server, `*Server.*` and `orderWidgetsImpl.cxx`, source files.

13. Insert the highlighted code shown in "Client Implementation Code" on page 60, to `sampleClient.cxx` to add the application logic to the client.
14. Build the client.

UNIX

```
make client.exe
```

Windows

```
nmake client.exe
```

Testing the Application

Overview

Once all of the components are generated, your system is ready to be tested.

Procedure

To test your Artix project complete the following steps:

1. Go to the widget project directory you created.
2. Run `artix_env`.
3. Go to the server directory.

   The server will be located in the `Server/src/cpp` folder of your project directory.
4. Start the server with the following command:

```
start server
```

5. Go to the client directory.

   The client will be located in the `Client/src/cpp` folder of your project directory.
6. Start the client with the following command:

```
start client
```

7. Answer the questions to complete the widget order form.
8. The server will return a bill containing the information you entered along with a randomly generated order number and a price for the widgets.

Sample output

Example 3 shows the output from a sample run of the Artix project.

Example 3: *Sample Widget Order*

```
C:\IONA\artix\1.2\demos\widgets>start client
```

Example 3: *Sample Widget Order*

```
orderWidgets Client
How many widgets do you want to order?123

What type of widgets do you want to order?
1 - Big
2 - Large
3 - Mungo
4 - Gargantuan
Selection [1-4]4

Enter Street Address:123 Elm Street
Enter Apt. or Suite Number:
Enter City:Walford
Enter State:CT
Enter ZIP Code:02343
Sending Widget Order
Bill for Your Widgets
Order Number: 23:12:4807/31/03
Date: 07/31/03
Quantity: 123
Type: Gargantuan
Amount Due: 123
Ship To:
123 Elm Street

Walford, CT
02343
```

CHAPTER 3

Using Artix Command Line Tools to Build a C++ Web Service

Artix Encompass is a world class utility for building C++ Web services using simple, standard C++ programming techniques. It provides all the tools needed to edit WSDL contracts and generate starting point code for Web servers and their clients.

In this chapter

This chapter discusses the following topics:

The Web Service Project	page 46
Using Artix Encompass Tools	page 47
Building the Widget Web Server	page 51
Building the Widget Web Service Client	page 53
Testing the Application	page 55

The Web Service Project

The problem scenario

Your company produces widgets and has decided to automate its ordering system to cut labor costs and reduce turnaround time. The new system will allow the company's customers to submit their orders electronically, will generate and send electronic bills to the customer, and generate a work order for your manufacturing system.

Your company's CIO has determined he wants this new system to be implemented using a Web service and that the development of both the server and the client will be done in-house. Unfortunately, your IT department doesn't have anybody with solid Java or Web services skills and there is no money or time to hire a new developer for this project.

How Artix simplifies solving the problem

Artix simplifies the solution to this problem by providing automated generation of the following:

- C++ server skeletons which allow developers to program using standard C++ metaphors
- C++ server implementation object method shells
- C++ client stubs which allow developers to program using standard C++ metaphors
- C++ server mainline starting point code
- C++ client mainline starting point code
- makefiles for Unix or Windows
- deployment descriptors for Web services

Using Artix Encompass Tools

Overview

Artix Encompass provides a full set of command line tools to take a Web service description and build the server stubs and client proxy code needed to implement the service. In this case, the problem is to define a service that receives an order and returns a bill. A full description of this service includes:

- The structure of the data the service sends and receives
- The operations offered by the service
- The order in which the data is encoded
- The payload format the service uses
- The transport the service uses
- The location of the service.

A Web service is defined in a WSDL document. Artix tools import WSDL directly and generate standard C++ code as starting point for development of the Web service.

The Web service description

For the purposes of this example we will use a predefined Widget service defined in Example 4.

Example 4: *Vendor WSDL document*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
 targetNamespace="http://widgetVendor.com/widgetOrderForm"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://widgetVendor.com/widgetOrderForm"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://widgetVendor.com/types/widgetTypes">
  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

Example 4: *Vendor WSDL document*

```
    <xsd:simpleType name="widgetSize">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="big"/>
        <xsd:enumeration value="large"/>
        <xsd:enumeration value="mungo"/>
        <xsd:enumeration value="gargantuan"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="Address">
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street1" type="xsd:string"/>
        <xsd:element name="street2" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="zipCode" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="widgetOrderInfo">
      <xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="shippingAddress" type="xsd1:Address"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="widgetOrderBillInfo">
      <xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="amtDue" type="xsd:float"/>
        <xsd:element name="orderNumber" type="xsd:string"/>
        <xsd:element name="shippingAddress" type="xsd1:Address"/>
      </xsd:sequence>
    </xsd:complexType>
  </schema>
</types>
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
</message>
```

Example 4: *Vendor WSDL document*

```
  <portType name="orderWidgets">
    <operation name="placeWidgetOrder">
      <input message="tns:widgetOrder" name="order"/>
      <output message="tns:widgetOrderBill" name="bill"/>
    </operation>
  </portType>
  <binding name="orderWidgetsBinding" type="tns:orderWidgets">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="placeWidgetOrder">
        <soap:operation soapAction="" style="rpc"/>
        <input name="widgetOrder">
          <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                     namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
        </input>
        <output name="widgetOrderBill">
          <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                     namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
        </output>
    </operation>
  </binding>
  <service name="orderWidgetsService">
    <port name="widgetOrderPort" binding="tns:orderWidgetsBinding">
      <soap:address location="http://localhost:8080"/>
    </port>
  </service>
</definitions>
```

This WSDL document completely describes the interface exposed by the Web service and the data that is passed to and from the server. Artix Designer can import this file directly and use it in the Artix contract that describes the entire integrated system you are building.

The major sections of the WSDL description are interpreted as follows:

`<types>`	Defines the complex data types used by the service. This service uses an enumerated type, `widgetSize`, to describe the widgets, a structure, `Address`, to hold the shipping address, and two structures, `widgetOrderInfo` and `widgetOrderBillInfo`, for the data needed to process the order.
`<message>`	Defines the messages by which the service communicates.
`<portType>`	Defines the operations offered by the service.

`<binding>`	Describes how the service expects its data to be formatted. In this case, it formats the data using SOAP.
`<service>`	Defines the address where the service can be contacted.

Building the Widget Web Server

Overview

Artix's wsdltocpp tool generates server stubs for any of the contracts used to describe a component of your integration project. In addition, it generates a sample server mainline, and generates a makefile to build the server.

Once wsdltocpp generates the stub code, you must write the implementation logic using the C++ development environment of your choice.

Procedure

To develop the widget web server using wsdltocpp complete the following steps:

1. Go to the Artix `bin` directory.

UNIX

```
$IT_PRODUCT_DIR/artix/1.2/bin
```

Windows

```
%IT_PRODUCT_DIR%\artix\1.2\bin
```

2. Source the `artix_env` script.
3. Go to the widgets demo directory.

UNIX

```
$IT_PRODUCT_DIR/artix/1.2/demos/widgets
```

Windows

```
%IT_PRODUCT_DIR%\artix\1.2\demos\widgets
```

4. Generate the server stubs from widget.wsdl using the `wsdltocpp` tool.

UNIX

```
wsdltocpp -sample -impl -m UNIX widgets.wsdl
```

Windows

```
wsdltocpp -sample -impl -m NMAKE widgets.wsdl
```

5. The following files are generated:

`orderWidgets.h`	`orderWidgetsClient.cxx`
`orderWidgetsClient.h`	`orderWidgetsImpl.cxx`
`orderWidgetsImpl.h`	`orderWidgetsServer.cxx`
`orderWidgetsServer.h`	`SampleClient.cxx`
`SampleServer.cxx`	`Makefile`
`widgts_wsdlTypesFactory.cxx`	`widgts_wsdlTypesFactory.h`
`widgts_wsdlTypes.cxx`	`widgts_wsdlTypes.h`

   For the purposes of generating a Web server to implement the widget ordering system, you do not need any of the client, `*Client.*`, source files.

6. Insert the highlighted code shown in "Server Implementation Code" on page 58, to `orderWidgetsImpl.cxx` to add the application logic to the server.
7. Build the server.

UNIX

```
make server.exe
```

Windows

```
nmake server.exe
```

Building the Widget Web Service Client

Overview

Artix generated proxy classes integrate smoothly into a standard C++ application. To use an Artix proxy you must initialize the Artix Bus and then instantiate an instance of the proxy class. Once instantiated the proxy object provides all of the functionality of the server through standard invocations of its methods.

Procedure

To create the widget web service client using `wsdltocpp` complete the following steps:

1. Go to the Artix `bin` directory.

UNIX

```
$IT_PRODUCT_DIR/artix/1.2/bin
```

Windows

```
%IT_PRODUCT_DIR%\artix\1.2\bin
```

2. Source the `artix_env` script.
3. Go to the widgets demo directory.

UNIX

```
$IT_PRODUCT_DIR/artix/1.2/demos/widgets
```

Windows

```
%IT_PRODUCT_DIR%\artix\1.2\demos\widgets
```

4. Generate the client proxies from `widget.wsdl` using the `wsdltocpp` tool.

UNIX

```
wsdltocpp -sample -m UNIX widgets.wsdl
```

Windows

```
wsdltocpp -sample -m NMAKE widgets.wsdl
```

5. The following files are generated:

`orderWidgets.h`	`orderWidgetsClient.cxx`
`orderWidgetsClient.h`	`orderWidgetsImpl.cxx`
`orderWidgetsServer.h`	`orderWidgetsServer.cxx`
`SampleServer.cxx`	`SampleClient.cxx`
`Makefile`	`widgts_wsdlTypesFactory.cxx`
`widgts_wsdlTypesFactory.h`	`widgts_wsdlTypes.h`
`widgts_wsdlTypes.cxx`	

   For the purposes of generating a Web service client to interact with the widget web server, you do not need any of the server, `*Server.*` and `orderWidgets.impl`, source files.

6. Insert the highlighted code shown in "Client Implementation Code" on page 60, to `sampleClient.cxx` to add the application logic to the client.
7. Build the client.

UNIX

```
make client.exe
```

Windows

```
nmake client.exe
```

Testing the Application

Overview

Once all of the components are generated, your system is ready to be tested.

Procedure

To test your Artix project complete the following steps:

1. Go to the widget project directory you created.
2. Run `artix_env`.
3. Start the server with the following command:

```
start server
```

4. Start the client with the following command:

```
start client
```

5. Answer the questions to complete the widget order form.
6. The server will return a bill containing the information you entered along with a randomly generated order number and a price for the widgets.

Sample output

Example 5 shows the output from a sample run of the Artix project.

Example 5: *Sample Widget Order*

```
C:\IONA\artix\1.1\demos\widgets>start client
```

Example 5: *Sample Widget Order*

```
orderWidgets Client
How many widgets do you want to order?123

What type of widgets do you want to order?
1 - Big
2 - Large
3 - Mungo
4 - Gargantuan
Selection [1-4]4

Enter Street Address:123 Elm Street
Enter Apt. or Suite Number:
Enter City:Walford
Enter State:CT
Enter ZIP Code:02343
Sending Widget Order
Bill for Your Widgets
Order Number: 23:12:4807/31/03
Date: 07/31/03
Quantity: 123
Type: Gargantuan
Amount Due: 123
Ship To:
123 Elm Street

Walford, CT
02343
```

APPENDEX A

Implementation Code for the Widget Server and Client

In this appendex

This appendex contains the following:

Server Implementation Code	page 58
Client Implementation Code	page 60

Server Implementation Code

Overview

The logic of an Artix server is developed inside of an implementation class generated by the Artix tools. This implementation code can typically be written using standard C++. For more advanced functionality, like transactions or security, you may need to use Artix-specific calls.

Code

Example 6 shows the implementation code for the sample widget Web service.

Example 6: *Widget Server Implementation*

```
#include <it_cal/iostream.h>
#include <it_cal/fstream.h>
#include <it_cal/cal.h>
#include <string.h>
#include <stdlib.h>
#include "orderWidgetsImpl.h"

IT_USING_NAMESPACE_STD

orderWidgetsImpl::orderWidgetsImpl(IT_Bus::Bus_ptr bus,
   IT_Bus::Port* port) : orderWidgetsServer(bus, port)
{
}

orderWidgetsImpl::~orderWidgetsImpl()
{
}

void orderWidgetsImpl::placeWidgetOrder(
    const widgetOrderInfo & widgetOrderForm,
    widgetOrderBillInfo & widgetOrderConformation
) IT_THROW_DECL((IT_Bus::Exception))
{
  widgetOrderConformation.setamount(
     widgetOrderForm.getamount());

  widgetOrderConformation.setorder_date(
     widgetOrderForm.getorder_date());
```

Example 6: *Widget Server Implementation*

```
  widgetOrderConformation.settype(widgetOrderForm.gettype());

  widgetOrderConformation.setshippingAddress(
     widgetOrderForm.getshippingAddress());

  IT_Bus::Float amtDue = widgetOrderForm.getamount() * 1.00;
  widgetOrderConformation.setamtDue(amtDue);

  char tempOrdNum[128], tempBuf[20];
  _strtime(tempOrdNum);
  _strdate(tempBuf);
  strcat(tempOrdNum, tempBuf);
  widgetOrderConformation.setorderNumber(tempOrdNum);
}
```

Client Implementation Code

Overview

The logic of an Artix client is developed using standard C++ calls. Artix-specific code is only needed to initialize the Artix Bus in the mainline of your client. For more advanced functionality, like transactions or security, you may need to use Artix specific-calls.

Code

The client application logic code is shown in Example 7.

Example 7: *Widget Web Service Client*

```
#include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_cal/iostream.h>
#include <it_cal/fstream.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

#include "orderWidgetsClient.h"

IT_USING_NAMESPACE_STD
using namespace IT_Bus;

long get_amount()
{
  long amount;

  cout << endl;
  cout << "How many widgets do you want to order?" << flush;

  cin >> amount;

  return(amount);
}
```

Example 7: *Widget Web Service Client*

```
widgetSize get_type()
{
  widgetSize type;
  char selection;

  cout << endl;
  cout << "What type of widgets do you want to order?" << endl;
  cout << "1 - Big" << endl;
  cout << "2 - Large" << endl;
  cout << "3 - Mungo" << endl;
  cout << "4 - Gargantuan" << endl;
  cout << "Selection [1-4]" << flush;

  cin >> selection;

  switch (selection)
    {
      case '1':
          {
            type = big;
            break;
          }
      case '2':
          {
            type = large;
            break;
          }
      case '3':
          {
            type = mungo;
            break;
           }
      case '4':
          {
            type = gargantuan;
            break;
          }
      default : type = mungo;
    }

  return(type);
}
```

Example 7: *Widget Web Service Client*

```
Address get_address()
{
  Address address;
  char temp[256];

  cout << endl;
  cout << "Enter Street Address:" << flush;
  gets(temp);  // clears the buffer
  gets(temp);
  address.street1 = string_dup(temp);

  cout << "Enter Apt. or Suite Number:" << flush;
  gets(temp);
  address.street2 = string_dup(temp);

  cout << "Enter City:" << flush;
  gets(temp);
  address.city = string_dup(temp);

  cout << "Enter State:" << flush;
  cin >> temp;
  address.state = string_dup(temp);

  cout << "Enter ZIP Code:" << flush;
  cin >> temp;
  address.zipCode = string_dup(temp);

  return(address);
}

void print_bill(widgetOrderBillInfo bill)
{
  cout << "Bill for Your Widgets" << endl;
  cout << "Order Number: " << bill.orderNumber << endl;
  cout << "Date: " << bill.order_date << endl;
  cout << "Quantity: " << bill.amount << endl;
```

Example 7: *Widget Web Service Client*

```
  switch(bill->type)
    {
      case big:
         {
           cout << "Type: Big" << endl;
           break;
         }
      case large:
          {
            cout << "Type: Large" << endl;
            break;
          }
      case mungo:
          {
            cout << "Type: Mungo" << endl;
            break;
          }
      case gargantuan: cout << "Type: Gargantuan" << endl;
    }

  cout << "Amount Due: " << bill.amtDue << endl;

  cout << "Ship To:" << endl;
  cout << bill.shippingAddress.street1 << endl;
  cout << bill.shippingAddress.street2 << endl;
  cout << bill.shippingAddress.city << ", " <<
   bill.shippingAddress.state << endl;
  cout << bill.shippingAddress.zipCode << endl;
}

int main(int argc, char** argv)
{
    cout << "orderWidgets Client" << endl;

    /*
     *  Create an instance of the web service client.
     */

    try
    {
        IT_Bus::init(argc, argv);

        orderWidgetsClient client;
```

Example 7: *Widget Web Service Client*

```
        // Sample invocation calls are shown in
        // commented lines below.

        /*

         widgetOrderInfo   widgetOrderForm;   // (INPUT)
         widgetOrderBillInfo   widgetOrderConformation;   //
   (OUTPUT)
         client.placeWidgetOrder ( widgetOrderForm,
   widgetOrderConformation  );
         */

      widgetOrderInfo order_form;
      order_form.amount = get_amount();
      char date[10];
      _strdate(date);
      order_form.order_date = CORBA::string_dup(date);
      order_form.type = get_type();
      order_form.shippingAddress = get_address();

    widgetOrderBillInfo bill;

    cout << "Sending Widget Order" << endl;
    client.placeWidgetOrder(order_form, bill);
    print_bill(bill);

    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.Message()
             << endl;
        return -1;
    }

    return 0;
}
```

Glossary

A

Artix Designer
A suite of GUI tools for creating and deploying Artix integration solutions.

B

Binding
A binding associates a specific transport/protocol and data format with the operations defined in a <portType>.

Bus
See Service Bus

Bridge
A usage mode in which Artix is used to integrate applications using different payload formats.

C

Connection
An established communication link between any two Artix endpoints.

Contract
An Artix contract is a WSDL file that defines the interface and all connection-related information for that interface. A contract contains two components: logical and physical. The logical contract defines things that are independent of the underlying transport and wire format, and is specified in the <portType>, <operation>, <message>, <type>, and <schema> WSDL tags.

The physical contract defines the payload format, middleware transport, and service groupings, and the mappings between these things and portType 'operations.' The physical contract is specified in the <port>, <binding> and <service> WSDL tags.

Contract Editor
A GUI tool used for editing Artix contracts. It provides several wizards for adding services, transports, and bindings to an Artix contract.

D

Deployment Mode

One of two ways in which an Artix application can be deployed: Embedded and Standalone. An embedded-mode Artix application is linked with Artix-generated stubs and skeletons to connect client and server to the service bus. A standalone application runs as a separate process in the form of a daemon.

E

Embedded Mode

Operational mode in which an application creates a Service Access Point, either by invoking Artix APIs directly, or by compiling and linking Artix-generated stubs and skeletons to connect client and server to the service bus.

End-point

The runtime deployment of one or more contracts, where one or more transports and its marshalling is defined, and at least one contract results in a generated stub or skeleton (thus an end-point can be compiled into an application). Contrast with Service.

H

Host

The network node on which a particular service resides.

M

Marshalling Format

A marshalling format controls the layout of a message to be delivered over a transport. A marshalling format is bound to a transport in the WSDL definition of a Port and its binding. A binding can also be specified in a logical contract portType, which allows for a logical contract to have multiple bindings and thus multiple wire message formats for the same contract.

P

Payload Format

The on-the-wire structure of a message over a given transport. A payload format is associated with a port (transport) in the WSDL via the binding definition.

Protocol

A protocol is a transport whose format is defined by an open standard.

R

Routing

The redirection of a message from one WSDL binding to another. Routing rules are specified in a contract and apply to both end-points and standalone services. Artix supports port-based routing and operation-based routing defined in WSDL contracts. Content-based routing is supported at the application level.

Router

A usage mode in which Artix redirects messages based on rules defined in an Artix contract.

S

Service

An Artix service is an instance of an Artix runtime deployed with one or more contracts, but with no generated language bindings. The service has no compile-time dependencies. A service is dynamically configured by deploying one or more contracts on it.

Service Access Point

The mechanism, and the points at which individual service providers and consumers connect to the service bus.

Service Bus

The set of service providers and consumers that communicate via Artix. Also known as an Enterprise Service Bus.

Standalone Mode

An Artix instance running independently of either of the applications it is integrating. This provides a minimally invasive integration solution and is fully described by an Artix contract.

Switch

A usage mode in which Artix connects applications using two different transport mechanisms.

System

A collection of services and transports.

T

Transport

An on-the-wire format for messages.

Transport Plug-In

A plug-in module that provides wire-level interoperation with a specific type of middleware. When configured with a given transport plug-in, Artix will interoperate with the specified middleware at a remote location or in another process. The transport is specified in the `<port>` element of a contract.

Index

A
Artix Bus 5
Artix contract 8
Artix Designer 24, 28
  binding editor 16
  contract editor 12
  interface editor 14
  message editor 14
  operation editor 15
  port editor 18
  project tree 11
  service editor 16
  system diagram 10
  type editor 13

B
binding 7, 27, 50

C
contract 7
contract editor
  binding editor 16
  interface editor 14
  message editor 14
  service editor 16
  type editor 13

I
interface editor
  operation editor 15

M
message 26, 49

O
operation 7

P
payload format 3, 8
portType 7, 26, 49

S
service 27, 50
Service Access Point 6, 7

T
types 26, 49

W
Web Services Definition Language 7
WSDL 24, 47