Artix™

Getting Started with Artix

Version 1.1, July 2003

Making Software Work Together™

Updated: 08-Aug-2003

M 3 1 0 2

Contents

List of Figures

Preface

Overview

The *Artix Getting Started Guide* is designed for use by anyone who needs to understand the concepts and terms used in the IONA Artix product, as well as anyone who needs to install Artix or maintain installed Artix systems.

Audience

This manual is geared for first time Artix users. It is assumed that the reader is familiar with the middleware systems discussed in this manual.

Organization of this guide

This guide is divided as follows:

- "Artix Concepts" provides general information about Artix and how it is used.
- "Using Artix Designer to Develop an Integrated System" presents a walk through of how to solve an integration problem with the Artix Designer.
- "Using Artix Command Line Tools to Develop an Integrated System" presents a walk through of the same integration scenario using the Artix command line tools.
- "Building the Widget Web Server" shows how to use Artix to build a C++ Web service from an Artix contract.

Online help

Artix includes online help, providing detailed information about various aspects of the Artix Graphical User Interface.

The **Help** menu provides access to this online help.

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*

The latest updates to the Artix documentation can be found at http://iona.com/docs/artix.

Reading path

If you are new to Artix, it's recommended that you read the documentation in the following order:

1. *Getting Started With Artix*
2. *Artix Tutorial*
3. *Artix User's Guide*
4. *Artix C++ Programming Guide*

Additional resources

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

http://www.iona.com/support/kb/

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

http://www.iona.com/support/update/

Getting help

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 doc-feedback@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, Windows NT, Windows 95, or Windows 98 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 Concepts

Artix uses middleware transports, message transformation, Web Service Definition Language (WSDL) service contracts, and message routing to solve integration problems.

In this chapter

This chapter discusses the following topics:

Introduction to Artix

Overview

Artix is a new approach to application integration, one that exploits the middleware technologies and products already present within an enterprise. Artix provides a rapid integration approach that increases operational efficiencies and makes it easier for an enterprise to adopt or extend a Service Oriented Architecture (SOA).

Artix is a middleware "bus." An Artix Bus (also known more generally as a Service Bus or an Enterprise Service Bus) is the collection of service providers and service consumers that are connected by Artix. An Artix Service Access Point (SAP) is where a service provider or service consumer is attached to the Bus.

An Artix SAP can support several different usage patterns, including middleware switch, router, and bridge:

- A switch provides a transformation from one transport to another. This can be thought of as a memory-to-memory copy of the payload data, without change to other portions of the message. For example, a switch would connect a system sending SOAP data over HTTP to a system expecting SOAP data over MQSeries.
- A bridge transforms one payload format, or marshalling system, to another without altering the content of the message. For example, a bridge would connect a CORBA system to a system expecting SOAP data over MQSeries.
- A router receives a message and decides where to send it based on a set of rules.

A single Artix SAP or application can provide any, or all of these usage patterns simultaneously.

By performing transformations between message formats and letting you choose the underlying transport, Artix facilitates the integration of applications. For example, you can using Artix to connect a CORBA application to an MQSeries application, where the CORBA application uses the IIOP transport, while the MQSeries application uses the MQSeries transport.

The CORBA and MQSeries applications can remain unchanged while Artix transforms the messages “on the wire” to the required payload format. Neither application is aware of any change, and each continues to operate as it always has. The difference is that the two applications are now integrated; they “understand” each other’s messages, and they each use a transport that makes the most sense for the enterprise overall.

Supported transports/protocols

A transport is an on-the-wire format for messages. A protocol is a transport that is defined by an open specification. Thus MQ and Tuxedo are transports, while HTTP and IIOP are protocols. In Artix documentation, both protocols and transports are referred to as transports. Artix supports the following message transports:

- HTTP
- Tuxedo
- MQSeries
- TIBCO/Rendezvous™
- IIOP
- IIOP Tunnel

An in-depth discussion of the differences between IIOP and IIOP Tunnel is beyond the scope of this chapter. Suffice it to say that IIOP Tunnel lets you use IONA’s Application Server Platform infrastructure (if present in your enterprise) and exploit its qualities of service in support of non-CORBA payloads. Usage of all these transports is described fully in the *Artix Users Guide*.

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.

Benefits of Artix

The Artix approach differs from the approach used by Enterprise Application Integration (EAI) products. The EAI approach typically uses a “canonical” format in an EAI hub. All messages are transformed from a source application’s native format to this canonical format, and then transformed again to the format of the target application. Each application requires two adapters that translate to and from the canonical format.

However, requiring two translations for every message incurs high overhead. Many enterprises prefer high-performance solutions that directly transform a small set of message types over a more general solution with lower performance.

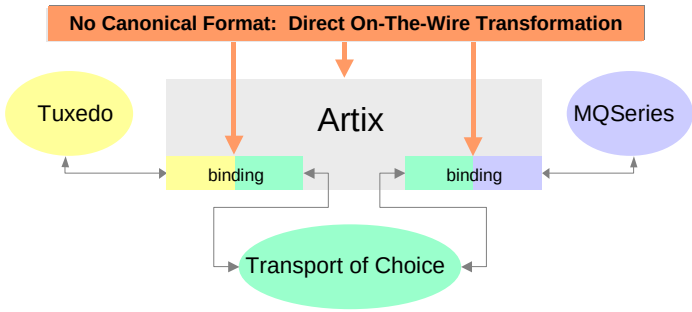

Figure 1: *Artix High-Performance Architecture*

Because Artix connects applications at the middleware transport level, Artix connections resemble the way network switches connect telephones. Like network switching, Artix hides the details of the connection and provides very high performance.

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. The Logical contract 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)
<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.

Artix Deployment Modes

Overview

Artix Service Access Points (SAPs) can be deployed in two modes:

- Standalone
- Embedded

Standalone mode

In standalone mode, Artix runs in a separate process (as an `itartix_service` executable), and is invoked as a service. Building a standalone Artix SAP is the approach least invasive to existing applications, but has lower performance than embedded mode. Standalone mode requires that routing rules be specified as part of the SAP's Artix contract.

Embedded mode

In embedded mode, an application is modified to invoke Artix functions directly and locally, as opposed to invoking a standalone Artix service. This approach is the most invasive to the application, but also provides the highest performance.

Embedded mode requires linking the application with Artix-generated stubs and skeletons to connect client and server (respectively) to the Service Bus. Embedded mode requires compiling the type-safe interfaces generated from an Artix contract and linking to the relevant libraries.

Thus, an Artix end-point is an Artix runtime deployment that implements one or more contracts, where at least one contract was used to generate a stub and skeleton. An end-point can thus be compiled into a new or existing application. In addition one or more transports and their marshalling can be deployed via the contracts.

In contrast, an Artix service is an instance of an Artix runtime deployed with one or more contracts where no language-specific stubs or skeletons are generated. The service runs in standalone mode and acts as a daemon that has no compile time dependencies. Contracts are deployed to the service, which are in turn dynamically read to configure the service. The service can serve more than one contract. Each contract defines a logical component with at least two transports and their respective bindings. Routing has to be defined between the two or more transports, with their respective bindings, for the contract to be functional.

Message Routing

Overview

Artix routing allows a message to be redirected from one WSDL port to another. Routing is supported at an end-point and is required in a standalone service deployment.

Artix supports the following types of routing:

- Port-based (including routing based on port properties or characteristics)
- Operation-based
- Content-based

Port-based and Operation-based routing rules can be fully expressed in a WSDL contract, while content-based routing is supported only at the application level.

Port-based routing acts on the port or transport-level identifier, and is the most efficient form of routing (port-based routing can also make a routing decision based on port properties, for example, on the message header or message identifier). Thus Artix can route messages based on the origin of a message or service request, or based on the message header or identifier.

Operation-based routing lets you choose route based on the operations contained within a port definition.

Content-based routing is the least efficient, because the source message must be parsed in order to make the routing decision.

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 the 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.

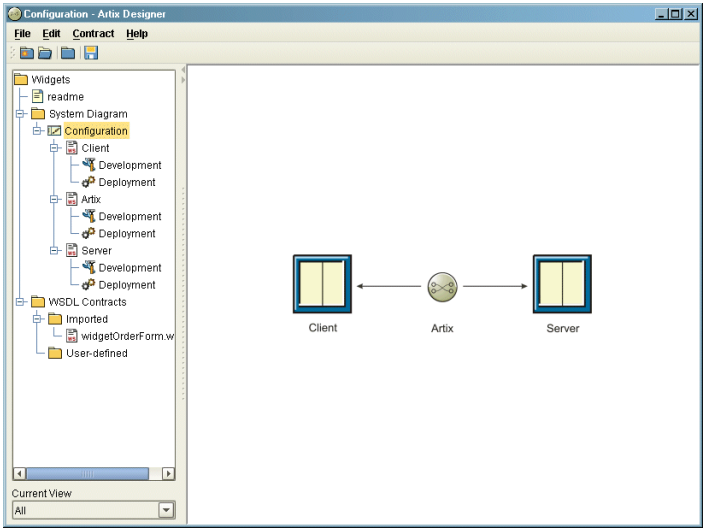

Figure 2: *Client-Server System Diagram*

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. Figure 3 shows the project tree for the system shown in

Figure 2 on page 11.

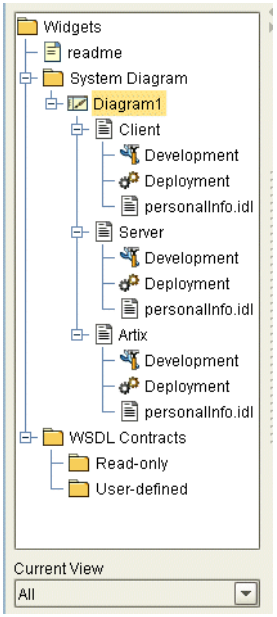

Figure 3: *Artix Project Tree*

The drop down list at the bottom of the tree 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 select 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 4, 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.

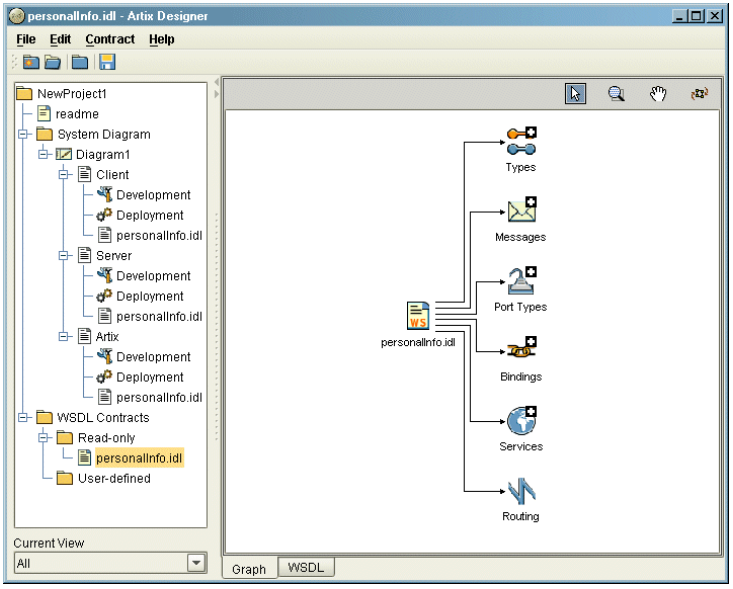

Figure 4: *Artix Contract Editor*

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 5 shows the screen for editing a `complexType`.

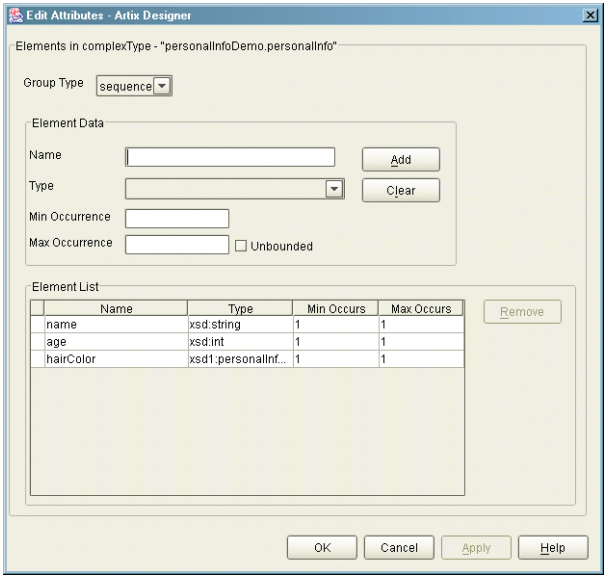

Figure 5: *Editing a complexType*

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 6 shows the message editor's main dialog.

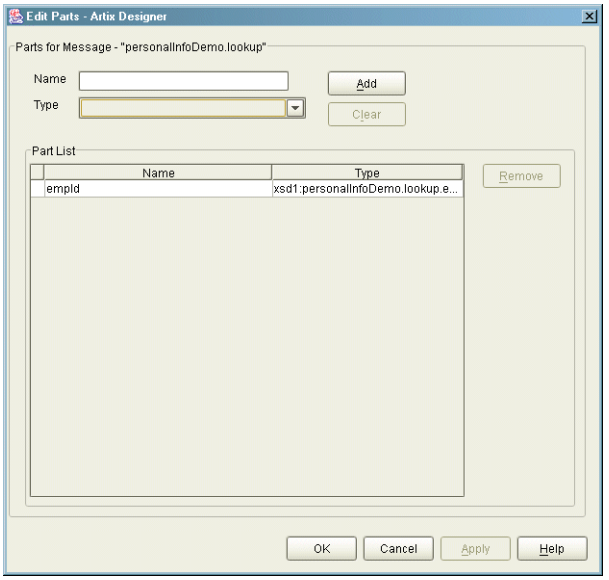

Figure 6: *Adding Parts to a Message*

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 7 shows the interface editor.

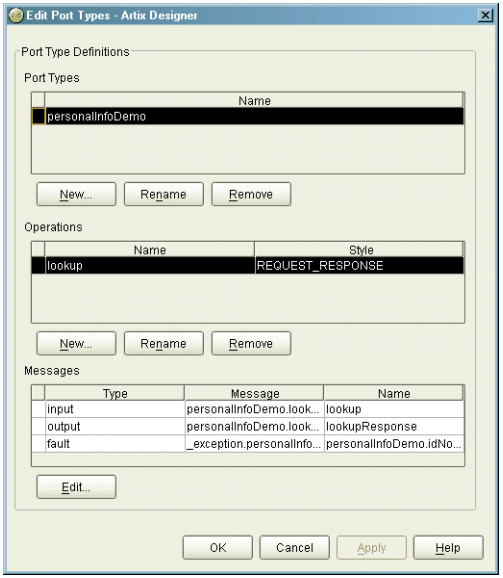

Figure 7: *Editing a PortType*

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 8 shows the operation editor.

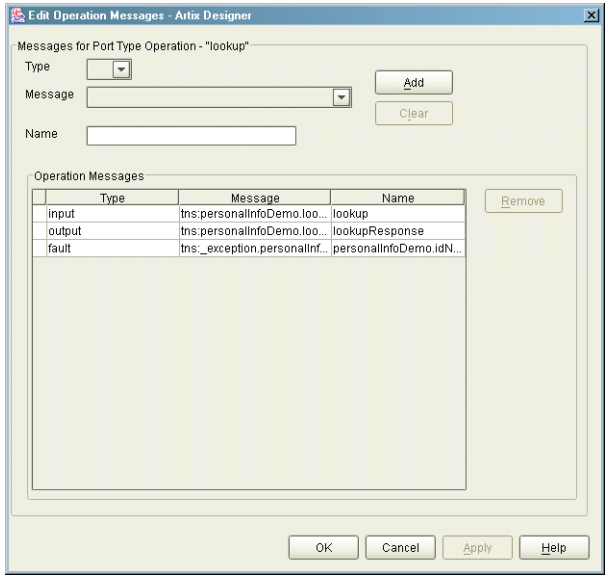

Figure 8: *Editing an Operation*

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 9, 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.

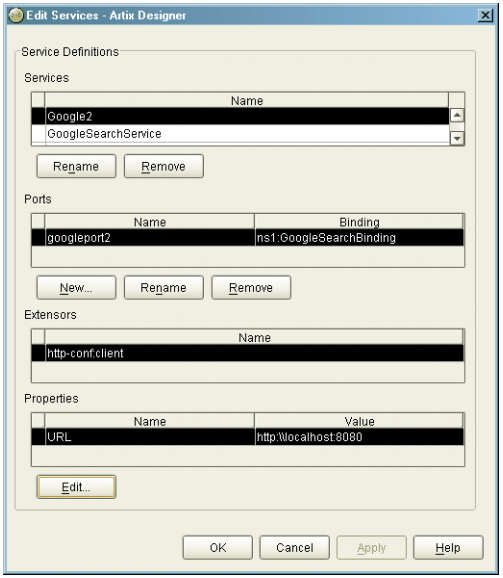

Figure 9: *Artix Service Editor*

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 10 shows the properties for an Artix HTTP port.

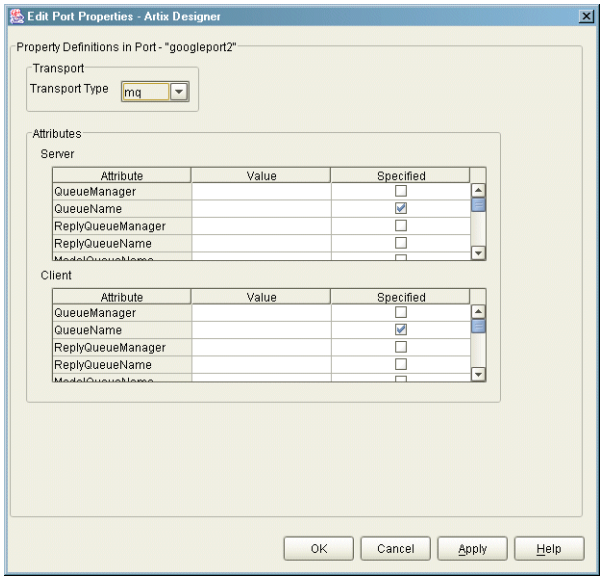

Figure 10: *Editing the Properties of an HTTP Port*

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 11, 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.

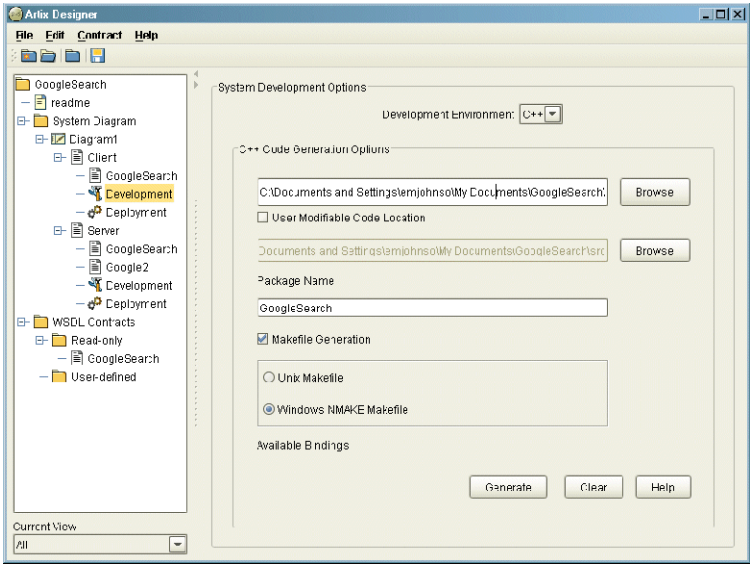

Figure 11: *Development Tool*

If the service's contract contains a CORBA binding, the development tool will can also used to 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 12, 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.

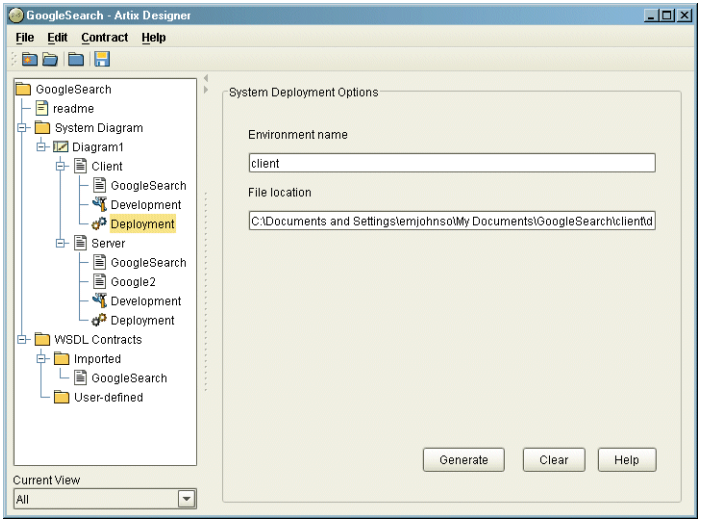

Figure 12: *Deployment Tool*

CHAPTER 2

Using Artix Designer to Develop an Integrated System

The Artix Designer simplifies the work of creating integrated software applications that use multiple transports and payload formats.

In this chapter

This chapter discusses the following topics:

The Integration Project

The problem scenario

Your company's inventory control and just-in-time ordering system is implemented using CORBA. When the manufacturing floor needs more parts, the system generates a purchase order and e-mails it to the vendor. When the vendor fulfills the order, they e-mail a bill to your company's billing department.

In order to cut labor costs, one of your company's largest vendors has just updated their ordering system to use a Web service front end, and has provided a description of this Web service front end in a WSDL file. The vendor still fulfills orders placed by e-mail but now charges a 10% premium for any order that is not processed via the new Web service.

Your company has determined that it will cost too much to continue e-mailing orders to this vendor, that there is no other vendor whose offerings are competitive, and that it is far too expensive to develop an entirely new inventory control and ordering system. Your company decides to modify the existing ordering system to use the vendor's Web service front end.

As the CORBA expert, you are given the task of integrating the two systems. You are the only person assigned to the task and given two weeks to complete it.

How Artix simplifies solving the problem

Artix simplifies the solution to this problem by providing the following:

- Automated generation of the IDL that describes the CORBA components of the project, from the WSDL provided by the vendor
- Automated generation of the binding information needed to map CORBA constructs to Web services constructs
- A routing editor that simplifies the creation of the rule directing messages to the proper interfaces
- Automated generation of the required configuration information
- The ability to implement the solution using a familiar programming model
- A light-weight runtime service that provides high-speed translation between the components of the integrated system

Using Artix Designer

Overview

Artix Designer lets you define and build many different types of integration solutions. In this case, the problem is one of integrating with an existing Web service, so the first step is obtaining a description of that service. A full description 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.

An operating Web service is defined in a WSDL document, and a CORBA application's interfaces are described in IDL. Artix can import IDL and WSDL directly, and convert them into Artix contracts (which are themselves WSDL files that may include IONA's extensions). Even if a service description is less formal than an existing IDL or 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.

Starting the integration project

You contact the vendor's IT department in order to obtain a description of the Web service interface. The IT department might provide the Internet address of the WSDL file that defines this service, or their e-mail reply might include the file itself. In any case, the required WSDL document is shown in Example 2.

Example 2: *Vendor WSDL document*

```
<?xml version="1.0" encoding="UTF-8"?>
```

Example 2: *Vendor WSDL document*

```
<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/">
      <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>
```

Example 2: *Vendor WSDL document*

```
      <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>
  <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 how to interact with the vendor's ordering system by way of XML documents. 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 the Artix Designer

Overview

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

Windows

On a Windows system you can start the Artix Designer form the **Start** menu. Select **Programs|IONA|Artix|Artix Designer**. You can also start the 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\6.0\bin
```

UNIX

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

1. Run `artix_env` to source the Artix environment.
2. Run `start_designer` to start the GUI.

Once the GUI is running

1. Select **Go straight to designer** on the welcome screen shown in Figure 13.

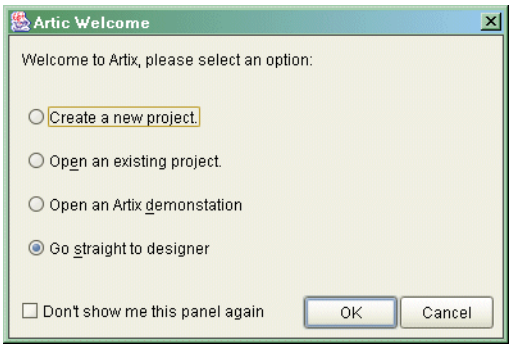

Figure 13: *Welcome Screen*

2. You will see a screen like Figure 14.

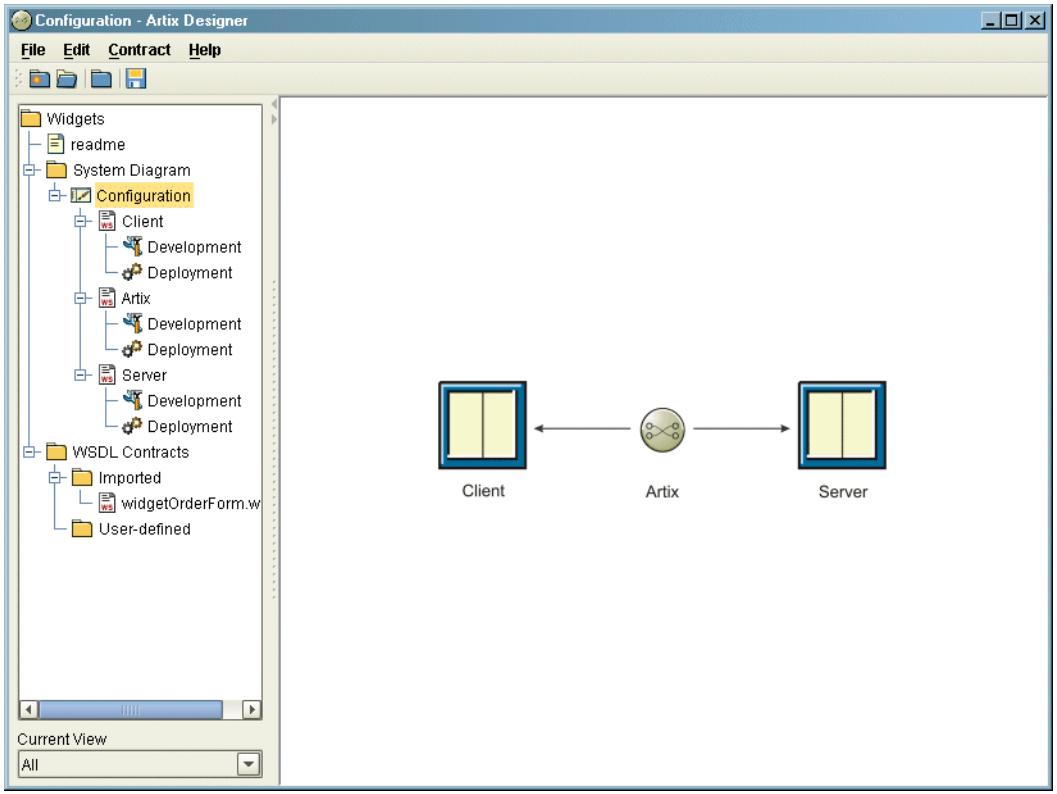

Figure 14: *Artix Designer*

Creating an Artix Project

Overview

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

Procedure

To create a new Artix project complete the following steps:

1. Select **New|Project** from the designer's **File** menu.
2. You will see a screen like Figure 15.

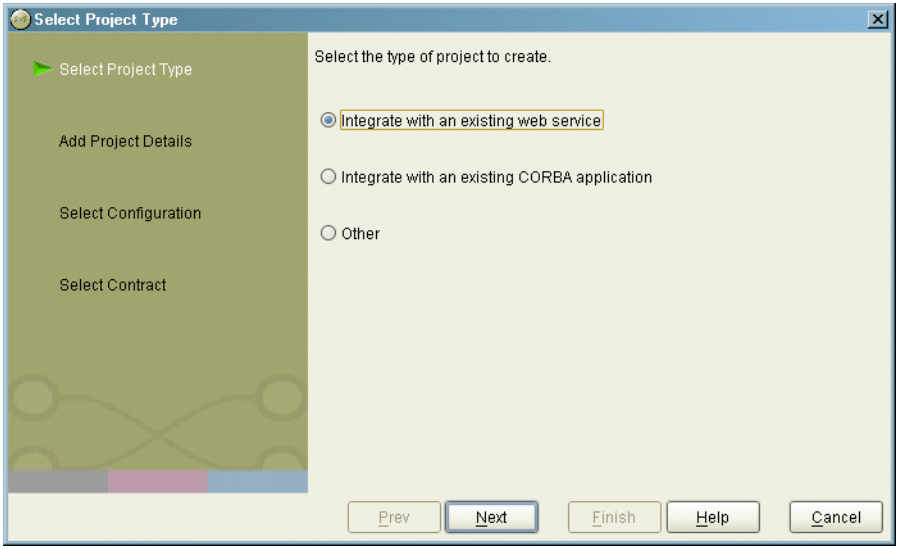

Figure 15: *Select Project Type*

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

5. You will see a screen like Figure 16.

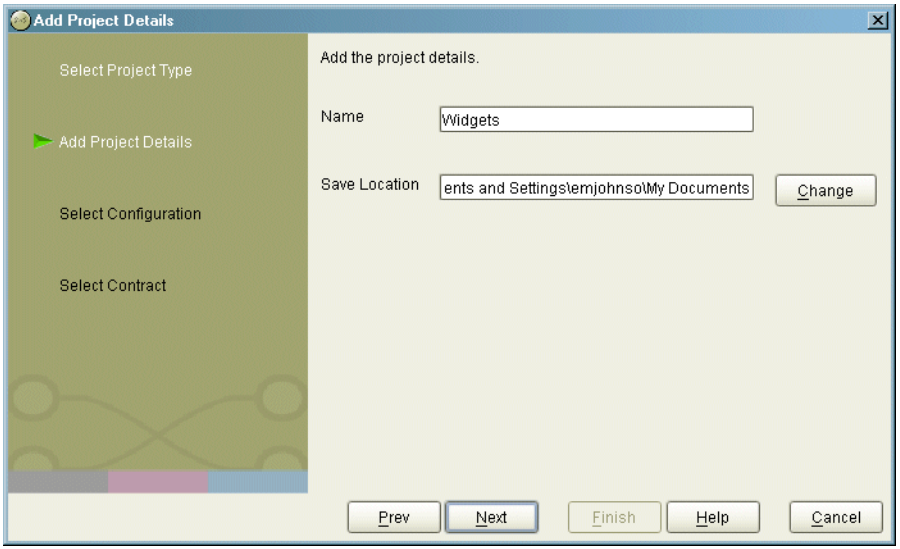

Figure 16: *New project details*

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 17 appears:.

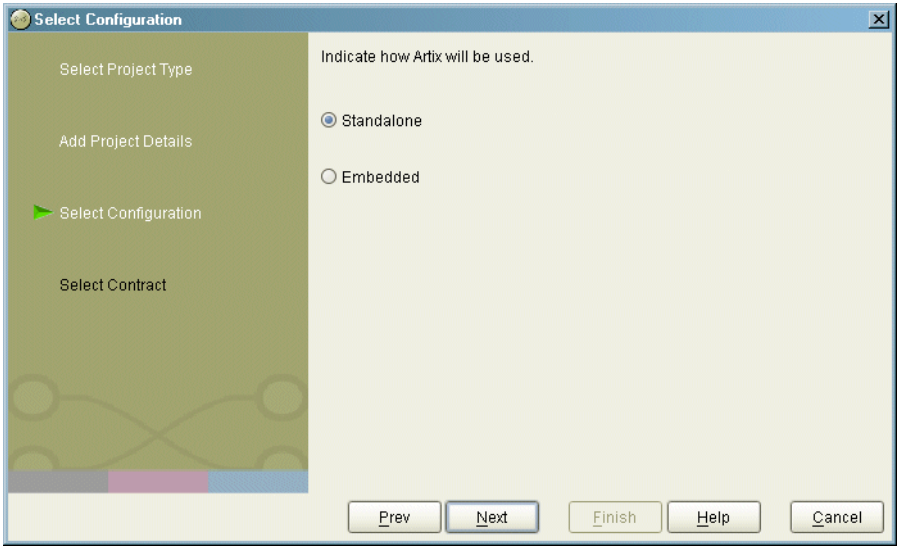

Figure 17: *System Configuration*

11. Select **Standalone**.
12. Click **Next**.

13. You will see a screen like Figure 18.

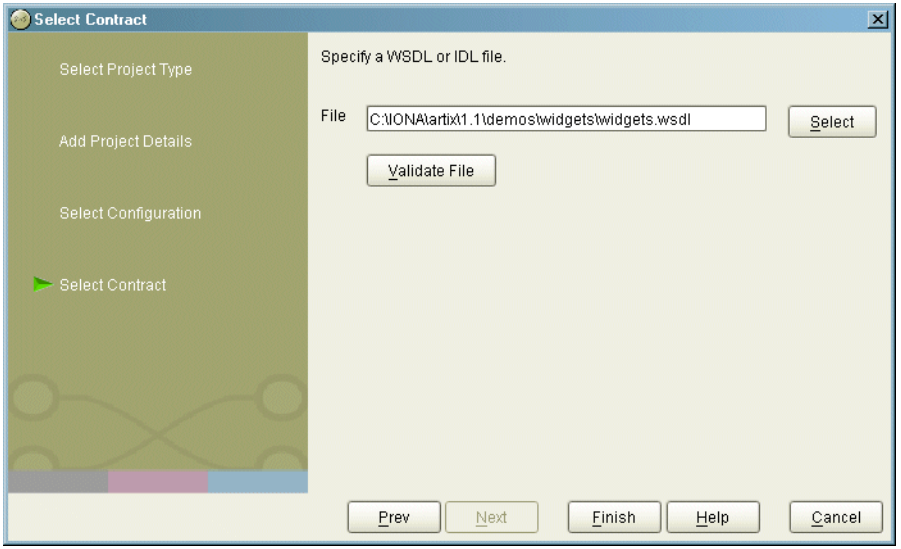

Figure 18: *WSDL File Selection*

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 19.

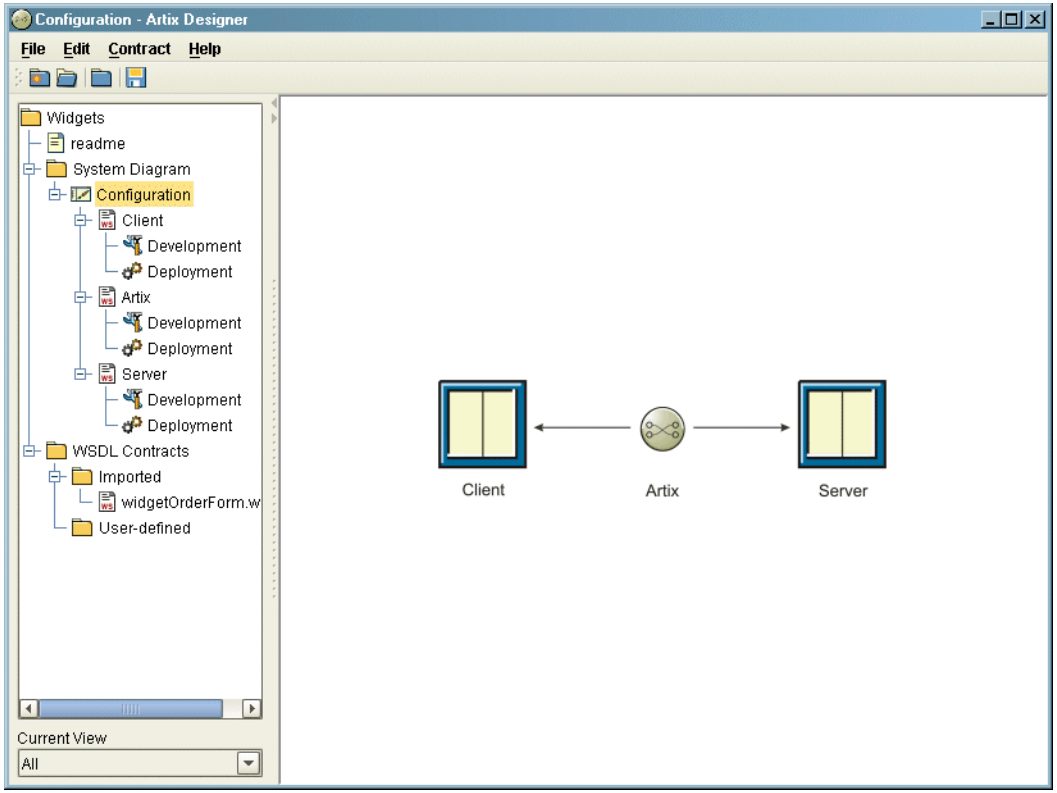

Figure 19: *Widget Service Starting Point*

Describing the Server

Overview

The WSDL file that was imported when you created the new project fully describes the server process for your project. This is the web service you CORBA system will need to send information to when placing an order for widgets.

Procedure

To describe the server in your Artix project complete the following steps:

1. Select the **widgetOrderForm** contract from the **Imported** folder of the project tree.
2. Drag the contract to the **Server** icon under the **System Diagram** folder on the project tree.
3. A copy of the contract will appear under the **Server**.

Describing the CORBA Client

Overview

To describe the CORBA client you need to modify the WSDL document that describes the server so that it includes the information needed to represent a CORBA object capable of implementing the same logical interface as the Web service. The needed information consists of a CORBA binding for the Web service's `portType`, a CORBA type map which maps the logical data described in the contract to concrete CORBA data types, and a CORBA port that defines the IOR used to by the CORBA client to invoke on the server. In this case however, the server is going to be an Artix instance mimicing a CORBA server and passing the request onto the Web service.

In this section

This section discusses the following topics:

Adding the CORBA Binding and Type Mapping

Overview

Artix Designer provides a tool to automatically generate a CORBA binding and the associated type map from a logical interface defined in an imported Artix contract. The Designer generates a new contract fragment, that imports the original contract, to hold the CORBA information.

Procedure

To add the CORBA binding and type map information to your CORBA client complete the following steps:

1. Select the **widgetOrderForm** contract from the **Imported** folder of the project tree.
2. Drag the contract to the **Client** icon under the **System Diagram** and drop it on the icon.
3. The contract will appear under the **Client**.
4. Select the **widgetOrderForm** contract from under the **Client** icon.
5. Select **Contract|New|Binding** from the menu at the top of the Designer.

6. You will see a screen like Figure 20.

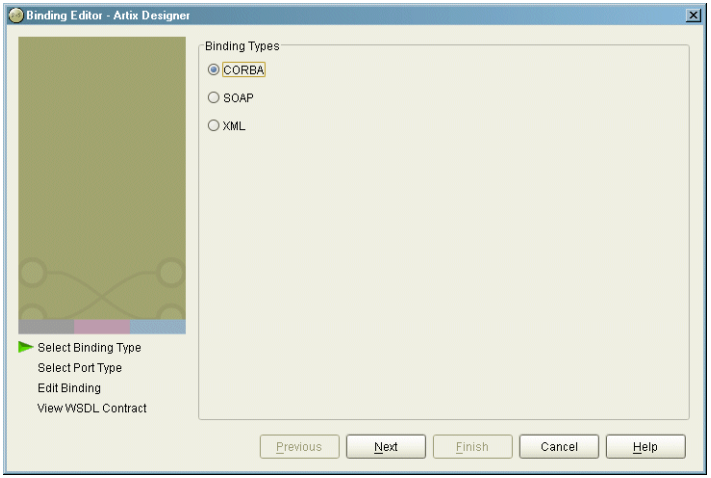

Figure 20: *Binding Selection Dialog*

7. Select **CORBA**.
8. Click **Next** to select the interface to bind.

9. You will see a screen like Figure 21.

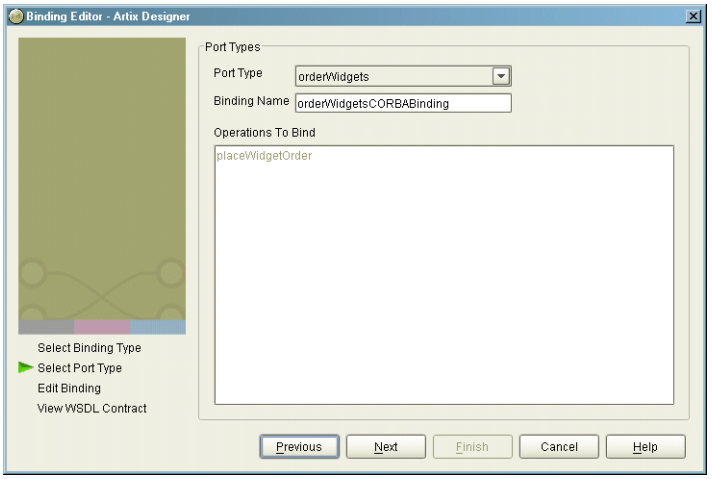

Figure 21: *Interface Selection Screen*

10. From the **PortType** pull-down list select **orderWidgets**.
11. Enter **`orderWidgetsCORBABinding`** for the Binding Name.
12. Click **Next** to review the binding and type map information

13. You will see a screen similar to Figure 22.

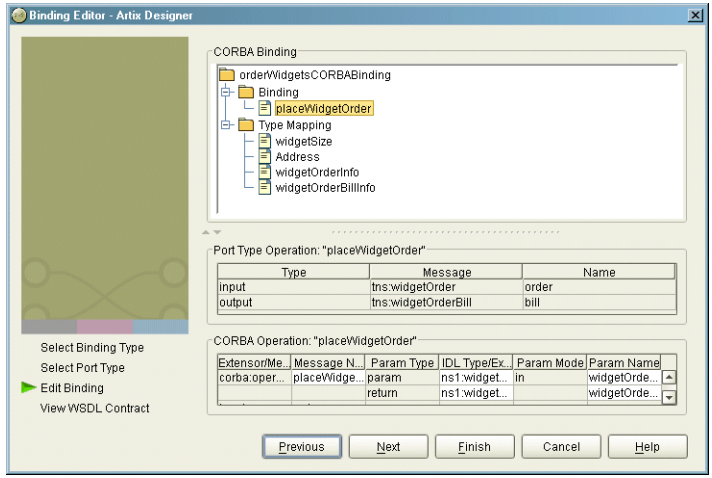

Figure 22: *Binding review*

14. Click on the elements on the **CORBA Binding** tree to review how they are mapped to a CORBA binding.
15. Click **Finish** to add the CORBA binding to your contract.
16. A new contract, `orderWidgetsCORBABinding`, will be added under the **Client** node of the project tree.

Adding the CORBA Port

Overview

Because CORBA is a unique protocol in that it specifies both a payload format and a transport, you cannot create a CORBA port in an Artix contract until it has a valid CORBA binding. After creating the CORBA binding and type map, you can now add a CORBA port to your client.

In WSDL ports are described within service elements. You can either define the new CORBA port inside the service describing the HTTP port. However, because in this example the HTTP port and the CORBA port are part of separate applications and are hosted by different organizations, it make sense to describe the CORBA port in a separate service.

Procedure

To add a new service containing a CORBA port to your client complete the following steps:

1. Select the **Client** node on the project tree.
2. Select **Contract|New|Service** from the menu.
3. Enter **`orderWidgetsCORBAService`** in the **Name** field.
4. Click **Next** to define the port.
5. Enter **`orderWidgetsCORBAPort`** in the **Name** field.
6. Select **`orderWidgetsCORBABinding`** from the **Binding** pull-down list.
7. Click **Next** to enter the port attributes.

8. You will see a screen similar to Figure 23.

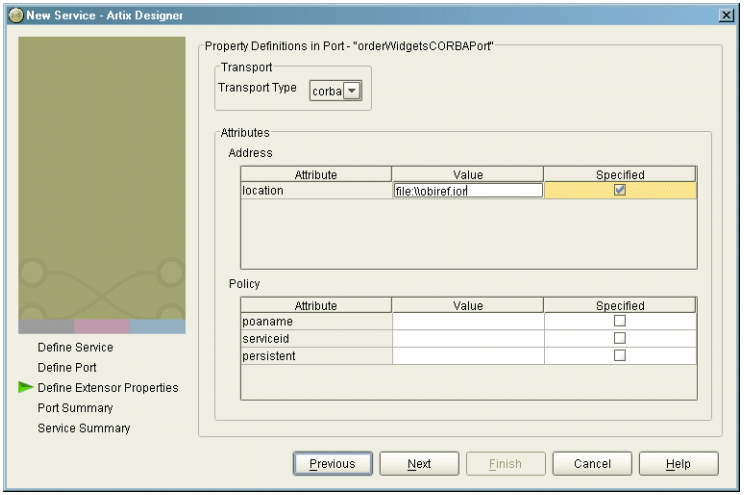

Figure 23: *Port Attributes*

9. Select **corba** from the **Transport Type** pull-down list.
10. Enter **`file:\\objref.ior`** in the location field.
11. Click **Next** to review the port settings.
12. Click **Next** to review the service settings.
13. Click **Finish** the add the new service.
14. A new contract, **orderWidgetsCORBAService**, will be added under the **Client** node of the project tree.

Developing the CORBA Interface

Overview

Artix generates IDL describing the logical interfaces that are bound to a CORBA binding. Once Artix has generated the IDL, you are responsible for developing the application code to support the interface in your CORBA application. The application code can be written using either the CORBA model, as shown in this example, or using Artix generated stub and skeleton code which is linked with the existing CORBA application.

Procedure

To develop a simple CORBA client to implement the new interface complete the following steps:

In the Artix Designer

1. Select the **Development** icon under the **Client** node on the project tree.
2. Select **IDL** from the **Development Environment** pull-down list.
3. Enter **`widgets.idl`** in the **IDL Location** field.
4. Click **OK** to generate the IDL.

In your development environment

5. Use the CORBA IDL compiler to generate the stub code from `widgets.idl`.

   If you have IONA's Application Server Platform v6.0 or later installed on your system use the following command:

   ```
   idl -base widgets.idl
   ```

6. Copy the client mainline code from Appendix B into a file called `client.cxx`.
7. Build the simple CORBA client.

Describing the Artix Service

Overview

The actual integration of your client and server are done by a standalone instance of the Artix service. The service's behavior is completely described by an Artix contract. This contract needs to contain descriptions of all of the services which will be integrated by this instance of the Artix service and the routing rules describing how each of the services are integrated. The Designer provides straight forward tools for describing the service integration rules.

Procedure

To describe your Artix service complete the following steps:

Adding the interface and service descriptions to the Artix service

1. Select the **widgetOrderForm** from under the **Client** node and drag it to the **Artix** node of the project tree.

   This adds the logical interfaces and the server's SOAP over HTTP service to the **Artix** service.
2. Select **widgetOrderFormCORBABinding** from under the **Client** node and drag it to the Artix node of the project tree.

   This adds the CORBA binding information for the client to the Artix service.
3. Select **widgetOrderFormCORBAService** from under the **Client** node and drag it to the **Artix** node of the project tree.

   This adds the client's CORBA service and port information to the Artix service.

Adding the routing information to the Artix service

4. Select the **Artix** node on the project tree.
5. Select **Contract|New|Route** from the menu at the top of the Designer.

6. You will see a screen like Figure 24.

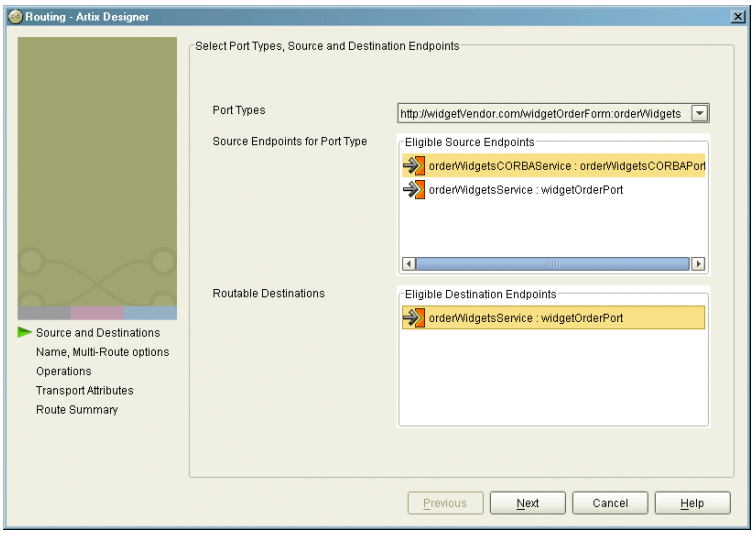

Figure 24: *Select Route Source and Destinations*

7. Select **http://widgetVendor.com/widgetOrderForm:orderWidgets** from the **PortTypes** pull-down list.
8. Select **orderWidgetsCORBAService:orderWidgetsCORBAPort** in the **Source Endpoints for PortType** field.
9. Select **orderWidgetsService:widgetOrderPort** in the **Routable Destinations** field.
10. Select **Next** to name the route.
11. Enter **`widgetRoute`** in the **Route Name** field.
12. Click **Next** to select the operations to route between.

13. You will see a screen like Figure 25.

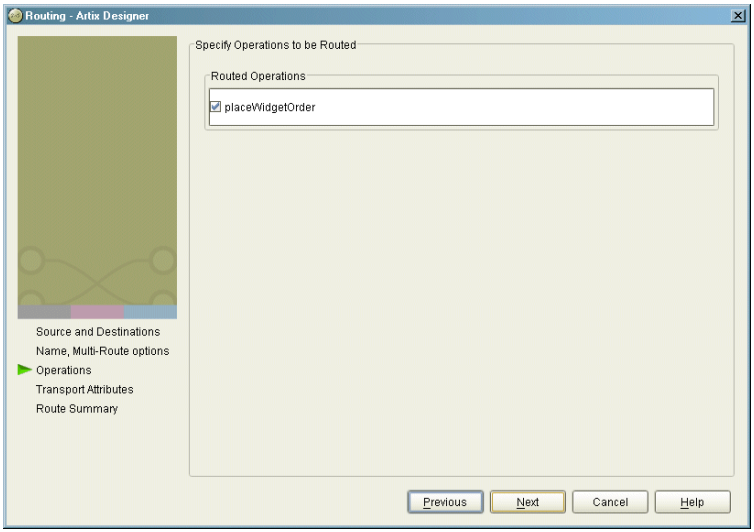

Figure 25: *Select Routing Operations*

14. Select **placeWidgetOrder** in the **Routed Operations** field.
15. Click **Next** to select the port attributes to use in routing.

16. You will see a screen like Figure 26.

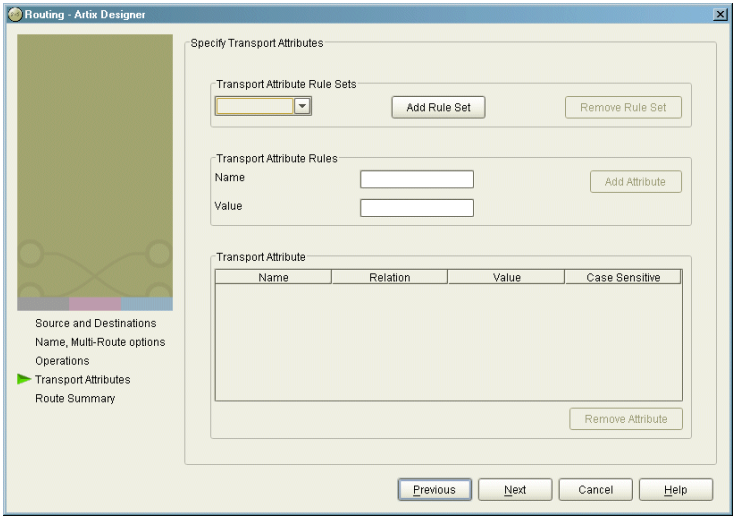

Figure 26: *Select Routing Port Attributes*

17. For this example port attributes are not used for routing, so click **Next**.

18. You will see a screen like Figure 27 which summarizes the route you added to the contract.

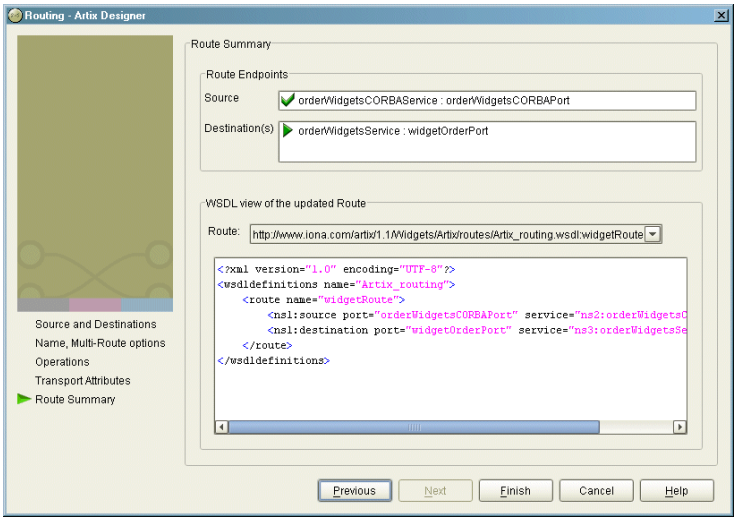

Figure 27: *Widget Route Summary*

19. Select **Finish** to create the route.
20. A new contract called **Artix_Routing** will be added to the **Artix** node of the project tree.

Deploying the Artix Service

Overview

The Artix standalone service requires some configuration information and the assembled Artix contracts to run properly. The Designer packages the configuration, the composite Artix contract, and start and stop scripts for the service into a deployment bundle for you. This bundle simply needs to be unpacked and the service is ready to integrate your systems.

Procedure

To deploy your Artix standalone service complete the following steps:

1. Select the Deployment Icon under the Artix node in the project tree.
2. Enter **`widgets`** for the **Environment Name**.
3. Click **Generate** to generate the configuration file.
4. A file containing the configuration for your Artix service, the contracts describing its behavior, and start and stop scripts is placed in your project directory.

Running the Integrated System

Overview

Once all of the components are generated, your system is ready to be tested. You will need to start the Artix service before starting the CORBA client because the Artix service needs to generate the IOR for the CORBA client.

Note: The directions for build the Web service for this example are shown in Appendix A.

Procedure

To test your Artix project complete the following steps:

1. Go to the widget project directory you created.
2. Unpack the widgets deployment bundle.
3. Run `artix_env`.
4. Start the Artix standalone service with the following command:

```
start_artix_service
```

5. Go to the server directory.

   If you built the server using the command line tools, the server will be located at `IT_PRODUCT_DIR\artix\1.1\demos\widgets`.

   If you built the server using Artix Designer, the server will be located in the `server/src` folder of your project directory.
6. Start the server with the following command:

```
start server
```

7. Go back to the widgets project directory.
8. Start the client with the following command:

```
client
```

9. Answer the questions to complete the widget order form.
10. 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\1.1\demos\widgets\corba>client
initializing ORB
narrowing CORBA::Object to orderWidgets

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
Widget Order demo complete.
```

CHAPTER 3

Using Artix Command Line Tools to Develop an Integrated System

Artix command line tools simplify the work of creating integrated software applications that use multiple transports and payload formats.

In this chapter

This chapter discusses the following topics:

The Integration Project

The problem scenario

Your company's inventory control and just-in-time ordering system is implemented using CORBA. When the manufacturing floor needs more parts, the system generates a purchase order and e-mails it to the vendor. When the vendor fulfills the order, they e-mail a bill to your company's billing department.

In order to cut labor costs, one of your company's largest vendors has just updated their ordering system to use a Web service front end, and has provided a description of this Web service front end in a WSDL file. The vendor still fulfills orders placed by e-mail but now charge a 10% premium for any order that is not processed via the new Web service.

Your company has determined that it will cost too much to continue e-mailing orders to this vendor, that there is no other vendor whose offerings are competitive, and that it is far too expensive to develop an entirely new inventory control and ordering system. Your company decides to modify the existing ordering system to use the vendor's Web service front end.

As the CORBA expert, you are given the task of integrating the two systems. You are the only person assigned to the task and given two weeks to complete it.

How Artix helps

Artix simplifies the solution to this problem by providing the following:

- Automated generation of the IDL that describes the CORBA components of the project, from the WSDL provided by the vendor
- Automated generation of the binding information needed to map CORBA constructs to Web services constructs
- The ability to implement the solution using a familiar programming model
- A light-weight runtime service that provides high-speed translation between the components of the integrated system

Using Artix

Overview

Artix lets you define and build many different types of integration solutions. In this case, the problem is one of integrating with an existing Web service, so the first step is obtaining a description of that service. A full description 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.

An operating Web service is defined in a WSDL document, and a CORBA application's interfaces are described in IDL. Artix can import IDL and WSDL directly, and convert them into Artix contracts (which are themselves WSDL files that may include IONA's extensions). Even if a service description is less formal than an existing IDL or 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.

Starting the integration project

You contact the vendor's IT department in order to obtain a description of the Web service interface. The IT department might provide the Internet address of WSDL file that defines this service, or their e-mail reply might include the file itself. In any case, the required WSDL document is shown in Example 4.

Example 4: *Vendor WSDL document*

```
<?xml version="1.0" encoding="UTF-8"?>
```

Example 4: *Vendor WSDL document*

```
<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/">
      <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>
```

Example 4: *Vendor WSDL document*

```
      <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>
  <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 how to interact with the vendor's ordering system by way of XML documents. 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.

Adding the CORBA Information

Overview

Artix provides the command line tool `wsdltocorba` to generate the appropriate CORBA binding in your Artix contract. `wsdltocorba` also generates the IDL needed to develop the CORBA components of your system.

Procedure

To generate the appropriate CORBA bindings and IDL file complete the following steps:

1. Go to `IT_PRODUCT_DIR\artix\bin`.
2. Run the `artix_env` script to set up the Artix environment.
3. Go to `IT_PRODUCT_DIR\artix\1.1\demos\widgets`.
4. Run `wsdltocorba` using the following command:

```
wsdltocorba -corba -idl -i orderWidgets
            -b orderWidgetsCORBABinding widgets.wsdl
```

5. The following files will be generated:

`widgets-corba.wsdl` A modified version of the original contract that includes the information needed to describe the CORBA system.

`widgets.idl` The IDL file describing the interface for the CORBA system.

6. Edit `widgets-corba.wsdl` to include a CORBA port by adding the portion of the code below in bold.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
    <types>
    ...
    </types>
    <message name="widgetOrder">
        <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
    </message>
    <message name="widgetOrderBill">
        <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
    </message>
```

```
    <portType name="orderWidgets">
    ...
    </portType>
    <binding name="orderWidgetsBinding" type="tns:orderWidgets">
    ...
    </binding>
    <binding name="orderWidgetsCORBABinding" type="tns:orderWidgets">
    ...
    </binding>
    <service name="orderWidgetsService">
        <port binding="tns:orderWidgetsBinding" name="widgetOrderPort">
            <soap:address location="http://localhost:8080"/>
        </port>
    </service>
    <service name="orderWidgetCORBAService">
      <port binding="tns:orderWidgetsCORBABinding" name="widgetCORBAPort">
        <corba:address location="file://objref.ior" />
      </port>
    </service>
    <corba:typeMapping targetNamespace="http://www.iona.com/corba/typemap/orderWidgets.idl">
    ...
    </corba:typeMapping>
</definitions>
```

Adding the Routing Information

Overview

The details of how Artix decides where to forward messages is defined using IONA extensions to WSDL. These are defined within the namespace `http://schemas.iona.com/routing` and the namespace is typically given the short name `routing`. For all integrations using the Artix standalone service, you need to specify at least one source and one destination.

Procedure

To add the routing information to your Artix contract complete the following:

1. Add the following to the namespace declarations at the beginning of `widgets-corba.wsdl`.

```
xmlns:routing="http://schemas.iona.com/routing"
```

2. Add the highlighted code to the end of `widgets-corba.wsdl`.

```
<definitions ...>
...
  <corba:typeMapping targetNamespace="http://www.iona.com/corba/typemap/orderWidgets.idl">
  ...
  </corba:typeMapping>
  <routing:route name="widgetRoute">
    <routing:source service="tns:orderWidgetCORBAService" port="tns:widgetCORBAPort" />
    <routing:destination service="tns:orderWidgetsService" port="tns:widgetOrderPort" />
  </routing:route>
</definitions>
```

Developing the CORBA Interface

Overview

Artix can generate the IDL describing the interface when it creates the CORBA binding and type map information in your Artix contract. However, you are responsible for developing the application code to support the interface in your CORBA application. The application code can be written using either the CORBA model, as shown in this example, or using Artix generated stub and skeleton code which is linked with the existing CORBA application.

Procedure

To develop a simple CORBA client to implement the new interface complete the following steps:

1. Use the CORBA IDL compiler to generate the stub code from `widgets.idl`.

   If you have IONA's Application Server Platform v6.0 or later installed on your system use the following command:

   ```
   idl -base widgets.idl
   ```

2. Copy the client mainline code from Appendix B into a file called `client.cxx`.
3. Build the simple CORBA client.

Configuring the Artix Switch

Overview

The Artix standalone service provides an easy and fast mechanism for connecting two services that speak different languages. It reads the contract, parses it, generates the ports needed for each service, intercepts the messages, and performs the required translations. All it requires is the Artix contract describing the services and their integration that you generated in the previous steps. In addition the standalone service needs to be configured to load the correct plugins and load the correct Artix contract.

To fully configure an instance of the Artix standalone service, you need to create two configuration scopes. One for the service itself and one for the process that stops the service. The most important values used in configuring the standalone service are `orb_plugins` and `plugins:routing:wsdl_url`. `orb_plugins` lists the plugins the service loads when it starts up. For this example you need to load the plugins for CORBA, HTTP, SOAP, and routing. `plugins:routing:wsdl_url` tells the service where to find the Artix contract that defines its behavior. The path specified is relative to the starting directory of the service.

Procedure

To properly configure the Artix standalone service for your project complete the following steps:

1. Locate the file the following file:

Windows

```
%IT_PRODUCT_DIR%\artix\1.1\etc\domains\artix.cfg
```

UNIX

```
$IT_PRODUCT_DIR/artix/1.1/etc/domains/artix.cfg
```

2. Open the file in a text editor.

3. Add the configuration scopes shown Example 5 in to the very end of the file.

Example 5: *Widget Artix Configuration Scope*

```
widget_artix_service
{
  orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
                 "iiop", "soap", "http", "ws_orb", "routing"];

  event_log:filters = ["*=ERROR+FATAL"];

  plugins:routing:wsdl_url="widgets-corba.wsdl";

  plugins:artix_service:shlib_name = "it_artix_service_svr";

  plugins:artix_service:iiop:port= "8900";

  plugins:artix_service:iiop:host= "localhost";

  plugins:artix_service:direct_persistence="true";

  policies:iiop:server_address_mode_policy:publish_hostname=
  "true";
};

widget_artix_service_admin
{
  orb_plugins = ["iiop_profile", "giop", "iiop"];

  initial_references:IT_ArtixServiceAdmin:reference=
    "corbaloc:iiop:1.2@localhost:8900/IT_ArtixServiceAdmin";
};
```

4. Save the file.

Running the Integrated System

Overview

Once all of the components are generated, your system is ready to be tested. You will need to start the Artix service before starting the CORBA client because the Artix service needs to generate the IOR for the CORBA client.

Note: The directions for build the Web service for this example are shown in Appendix A.

Procedure

To test your Artix project complete the following steps:

1. Go to the Artix `bin` directory.

UNIX

```
$IT_PRODUCT_DIR/artix/1.1/bin
```

Windows

```
%IT_PRODUCT_DIR%\artix\1.1\bin
```

2. Run `artix_env`.
3. Go to the widgets demo directory.

UNIX

```
$IT_PRODUCT_DIR/artix/1.1/demos/widgets
```

Windows

```
%IT_PRODUCT_DIR%\artix\1.1\demos\widgets
```

4. Start the Artix standalone service with the following command:

```
itartix_service -ORBname widget_artix_service run -background
```

5. Go to the server directory.

   If you built the server using the command line tools, the server will be located at `IT_PRODUCT_DIR\artix\1.1\demos\widgets`.

If you built the server using Artix Designer, the server will be located in the `server/src` folder of your project directory.

6. Start the server with the following command:

```
start server
```

7. Go back to the widgets demo directory.
8. Start the client with the following command:

```
client
```

9. Answer the questions to complete the widget order form.
10. 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 6 shows the output from a sample run of the Artix project.

Example 6: *Sample Widget Order*

```
C:\IONA\artix\1.1\demos\widgets\corba>client
initializing ORB
narrowing CORBA::Object to orderWidgets

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
```

Example 6: *Sample 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
Widget Order demo complete.
```

Building the Widget Web Server

In addition to providing middleware integration, Artix provides the tools to create high-performance C++ Web services using standard C++ programming techniques.

Overview

Both the Artix Designer and the Artix command line tools can generate C++ server stub code and C++ client proxy code for the interfaces described in an Artix contract. The Artix generated code hides the complexity of the underlying transport implementation from the application developer and exposes the objects generated from the contract so that they are usable as if they were standard C++ objects. This means that the application developer can focus on implementing the application logic without worrying about how the application communicates with the outside world.

For a detail description of programming with Artix read the *Artix C++ Programmer's Guide*.

In this appendix

This appendix discusses the following topics:

Using Artix Designer

Overview

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

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

Procedure

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

1. Start Artix Designer.

Windows

```
start_designer
```

UNIX

```
artix_env
start_designer
```

2. Follow the directions for creating an Artix project shown in "Creating an Artix Project" on page 33.
3. Follow the directs for describing the widget server shown in "Describing the Server" on page 38.
4. Select the **Development** icon under the **Server** node in the project tree.

5. You will see a screen similar to Figure 28.

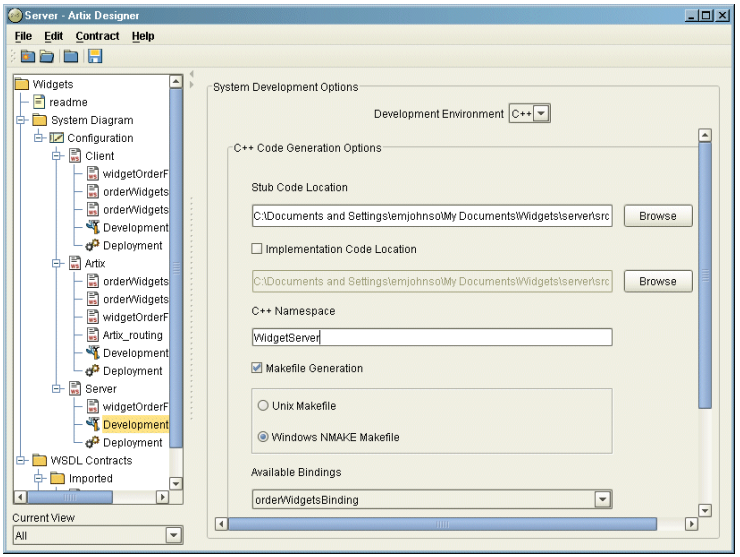

Figure 28: *Widget Server Development Screen*

6. Select **C++** from the **Development Environment** pull-down list.
7. Enter **`WidgetServer`** for the **Package Name**.
8. Select the appropriate type of makefile generation for your platform.
9. Select **orderWidgetsBinding** from the **Available Bindings** pull-down list.
10. Select **orderWidgetsService** from the **Available Services** pull-down list.
11. Click **Generate**.
12. The following files are generated in the `server/src` directory of your project folder:

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

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 Example 7 on page 78, to `orderWidgetsImpl.cxx` to add the application logic to the server.
14. Build the server.

UNIX

```
make server.exe
```

Windows

```
nmake server.exe
```

Using the Command Line Tools

Overview

Artix has a command line tool, `wsdltocpp`, that generates server stubs and client proxy code from Artix contracts. The benefit of this tool is that it can be included in makefiles to help automate the building of applications that incorporate Artix code and make migrating to newer versions of the product easier.

Procedure

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

1. Go to the Artix `bin` directory.

UNIX

```
$IT_PRODUCT_DIR/artix/1.1/bin
```

Windows

```
%IT_PRODUCT_DIR%\artix\1.1\bin
```

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

UNIX

```
$IT_PRODUCT_DIR/artix/1.1/demos/widgets
```

Windows

```
%IT_PRODUCT_DIR%\artix\1.1\demos\widgets
```

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

UNIX

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

Windows

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

5. The following files are generated in the `server/src` directory of your project folder:

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

   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 Example 7 on page 78, to `orderWidgetsImpl.cxx` to add the application logic to the server.
7. Build the server.

UNIX

```
make server.exe
```

Windows

```
nmake server.exe
```

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 7 shows the implementation code for the sample widget Web service.

Example 7: *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 7: *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);
}
```

The CORBA Client Code

The mainline for the Demo CORBA client is pure CORBA code.

Overview

The CORBA portion of the widgets example is intended to be a CORBA client. As such it does not require any CORBA services to be running. The Artix switch publishes the IOR to a file which the client reads. This can be modified to take advantage of a CORBA naming service, but that is beyond the scope of this demo.

Client source

The mainline used in this demo is shown in Example 8.

Example 8: *Widget CORBA client*

```
#include <it_cal/iostream.h>
#include <it_cal/fstream.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <omg/orb.hh>

#include "widgets.hh"

IT_USING_NAMESPACE_STD

const char* const objref_file = "../objref.ior";
```

Example 8: *Widget CORBA client*

```
long get_amount()
{
  long amount;

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

  cin >> amount;

  return(amount);
}

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;
```

Example 8: *Widget CORBA client*

```
  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 8: *Widget CORBA 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 = CORBA::string_dup(temp);

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

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

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

  cout << "Enter ZIP Code:" << flush;
  cin >> temp;
  address.zipCode = CORBA::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 8: *Widget CORBA 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;
}
```

Example 8: *Widget CORBA client*

```
int main(int argc, char** argv)
{
  // Initialize the ORB.
  CORBA::ORB_var orb;
  try
    {
      cout << "initializing ORB" << endl;
      orb = CORBA::ORB_init(argc, argv);
    }
  catch (CORBA::SystemException& se)
    {
      cerr << "ORB_init failed: " << se << endl;
      return 1;
    }
  if (CORBA::is_nil(orb))
    {
      cerr << "ORB_init returned nil object reference\n";
      return 1;
    }

  // Obtain stringified object reference from file.
  CORBA::String_var objref_string;
    {
  const char* filename = objref_file;
  IT_ifstream is(filename);
  if (!is.good())
  {
  cerr << "error opening " << filename << endl;
  return 1;
  }
  is >> objref_string;
  if (objref_string.in() == 0 || strlen(objref_string.in()) == 0)
  {
  cerr << "object reference string has zero length\n";
  return 1;
  }
  }
```

Example 8: *Widget CORBA client*

```
  // Destringify the object reference.
  CORBA::Object_var tobj;
  try
  {
tobj = orb->string_to_object(objref_string.in());
  }
  catch (CORBA::SystemException& se)
  {
cerr << "string_to_object failed: " << se << endl;
return 1;
  }

// Narrow the object reference.
orderWidgets_var proxy;
try
  {
    cout << "narrowing CORBA::Object to orderWidgets" << endl;
    proxy = orderWidgets::_narrow(tobj);
  }
catch (CORBA::SystemException& se)
  {
    cerr << "orderWidgets::_narrow failed: " << se << endl;
    return 1;
  }
if (CORBA::is_nil(proxy.in()))
  {
    cerr << "orderWidgets::_narrow returned a nil object
 reference\n";
    return 1;
  }

try
  {
    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();
```

Example 8: *Widget CORBA client*

```
    widgetOrderBillInfo *bill;

    cout << "Sending Widget Order" << endl;
    bill = proxy->placeWidgetOrder(order_form);

    print_bill(bill);

    CORBA::string_free(order_form.order_date);
  }
  catch (CORBA::SystemException& se)
  {
    cerr << "orderWidgets failed: " << se << endl;
    return 1;
  }

  try
  {
    orb->shutdown(IT_TRUE);
  }
  catch (CORBA::SystemException& se)
  {
    cerr << "CORBA::ORB::shutdown failed: " << se << endl;
    return 1;
  }

    cout << "Widget Order demo complete." << endl;
    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>`, `<opperation>`, `<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' property in of a contract.

Index