Basic Tutorial

Version 1.3, December 2003

Making Software Work Together™

Updated: 13-Jan-2004

Contents

Preface

What is Covered in this Book

The *Artix Basic Tutorial* provides a basic understand the concepts and terms used in the IONA Artix Encompass product.

Who Should Read this Book

This manual is geared for first time Artix users. It is assumed that the reader is familiar with C++ coding.

Organization of this Book

This book will guide you through the development of several Artix applications. Initially you will use command line tools and the Artix Designer to build and deploy HelloWorld applications. At the end of the document you will be given an opportunity to build and deploy a more involved application. See Chapter 1 for a more complete detailing of the chapter contents.

Related Documentation

The document set for Artix includes the following:

- *Getting Started with Artix Encompass*
- *Getting Started with Artix Relay*
- *Designing Artix Solutions*
- *Deploying and Managing Artix Solutions*
- *Artix Installation Guide*

- *Artix Tutorial*
- *Developing Artix Applications in C++*
- *Developing Artix Applications in Java*
- *Artix Security Guide*

The latest updates to the Artix documentation can be found at http://iona.com/support/docs/artix/1.3/index.xml.

Online Help

The Artix Designer includes comprehensive online help, providing:

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

There are two ways to access the online help: via the Help menu in the Artix Designer, or by clicking the Help button on any interface dialog.

Suggested Path for Further Reading

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

1. *Artix Basic Tutorial*
2. *Getting Started with Artix C++*

   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.
3. *Developing Artix Applications in C++*

   The development guide discusses the technical aspects of programming applications using the Artix API.
4. *Artix Tutorial*

   The Tutorial guides you through developing Artix applications using all of the supported transports. This document contains advanced content that is not supported by the Artix Encompass Standard product.
5. *Deploying and Managing Artix Solutions*

The deployment guide describes deploying Artix enabled systems. It provides detailed examples for a number of typical use cases.

6. *Designing Artix Solutions with Artix Designer*

   The Artix Designer book describes how to use the Artix GUI to describe your services in an Artix contract.

7. *Designing Artix Solutions from the Command Line*

   This book provides detailed information about the WSDL extentions used in Artix contracts and explains the mappings between data types and Artix bindings.

Additional Resources for Help

The IONA Knowledge Base contains helpful articles, written by IONA experts, about Orbix and other products. You can access the knowledge base at the following location: (http://www.iona.com/support/knowledge_base/index.xml)

The IONA Update Center (http://www.iona.com/support/updates/index.xml) 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 book 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 book uses 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

Introduction

This tutorial describes the basics of creating a Web service using Artix™ Encompass Standard, v1.3.

In This Chapter

This chapter discusses the following topics:

Introduction to the Tutorial

What is Covered

This tutorial discusses the basics of building a Web service application using Artix Encompass Standard edition. The examples use SOAP encoding over the HTTP transport.

Chapter 2, "The WSDL File" discusses the content of a WSDL file and how this file provides all of the information needed to develop and access a Web service.

Chapter 3, "Coding the Web Service" discusses how you can use Artix to create a Web service from an existing WSDL file. This chapter details usage of the wsdltocpp utility, a command line application that generates the C++ code for your application.

Chapter 4, "Using the Artix™ Designer" instructs you on how to use the Artix Designer to write a WSDL file. You will also use the designer to generate starting point C++ code for your application.

Chapter 5, "Faults and Exceptions" discusses Web service faults and their representation as C++ exceptions. You will use the Artix designer to add fault handling to the application developed in Chapter 4.

Chapter 6, "Mortgage Calculator" gives you an opportunity to build a Web service that is significantly more involved than the HelloWorld examples developed in the earlier chapters.

With the exception of the work in Chapter 6, minimal C++ coding is required and the tutorial commentary provides all of the code you need to implement the examples. Consequently, even if you are not an experienced C++ programmer, you will gain a considerable understanding of Artix by working through this tutorial.

You may use this tutorial as the first piece of Artix product documentation that you review. When you complete this tutorial you will be ready to study the more detailed product documentation, specifically the programmers guide Developing Artix Applications in C++ and the other, more advanced, Artix tutorials.

CHAPTER 2

The WSDL File

A Web service is defined as an application that:
Is a service defined and described by an XML document.
Is a service that can be discovered using XML documents.

This chapter discusses how the Web Services Definition Language (WSDL) file can be used to satisfy these requirements.

In This Chapter

This chapter discusses the following topics:

What Are Web Services?

Web Service Concepts

The information services community generally regards Web services as application-to-application interactions that utilize XML data representations and the hypertext transfer protocol The advantages of Web services lie in the fact that the data encoding scheme and transport semantics are based on standardized, non-proprietary specifications. Additionally, string-based message content is human readable, can be created and manipulated by any programming development tool, and provides a high level of security and data integrity.

What is Artix™?

Artix extends the concept of Web services to include multiple data encoding schemas and transport protocols. Additionally, Artix provides transparent conversions between different data encoding schemas and/or transport protocols. As a consequence, you are now free to develop applications that integrate different middleware technologies without the burden of writing wrapper or adapter components.

Artix is built on IONA's Adaptive Runtime Technology. Application functionality is extended through configuration rather than coding. Customer applications are built on a collection of plugin libraries. If your application does not require the functionality provided by a plugin, you can exclude this library from your development and production environment. Additionally, the capabilities of the Artix product line can be easily extended through the introduction of new plugins. Existing applications, which obviously do not require the services of the new plugin, are unaffected.

The Artix runtime environment also includes a number of enterprise services, e.g., management or security, which support your production requirements.

There are three approaches to using Artix. First, you can write applications using the Artix Application Programming Interface (API). In this situation, you are writing new applications using Artix as your development tool. This is the approach introduced in this tutorial.

Second, you can integrate two existing applications, built on different middleware technologies, into a single application. In this situation, developers work with their current development tools and Artix functions as

a broker between the two dissimilar data encoding schemas and transport protocols. This approach requires the extended functionality of the Artix Encompass Advanced or Enterprise or Artix Relay Advanced or Enterprise products, and is not covered in this tutorial.

Finally, you can use Artix as a replacement for other middleware transport protocols. Your application code remains unchanged; the Artix libraries replace the middleware libraries within your executable. This approach is not covered in this tutorial.

Becoming Proficient with Artix

To become an effective Artix developer you need to understand four central concepts, only one of which is related to writing code. First, you need to understand the syntax for WSDL files and the Artix extensions to the WSDL specification.

Second, you need to understand the relationship between Artix WSDL extensions, Adaptive Runtime Technology plugins, and setting configuration entries.

Third, you need an understanding of the Artix API, and the IONA and Artix foundation class libraries, which you can use in your application.

Fourth, you must gain proficiency with the Artix Service Designer, a tool through which you can write and edit WSDL files, convert CORBA Interface Definition Language (IDL) files and data files into WSDL, and generate code.

This tutorial introduces concepts in all four of these categories. Subsequent tutorials and the product documentation cover each of these concepts in greater detail.

What is WSDL?

Web Services Description Language

A WSDL file is an XML document that is used to describe a Web service. In this respect, it is similar to a CORBA IDL file, an abstract C++ class, or a Java interface definition. Information within the WSDL file describes the operations offered by the Web service and the location of the Web service. Since the WSDL file is an XML document, it may be validated against an XML Schema document to insure its accuracy.

WSDL files include three sets of elements, and child elements, which collectively define a Web service. While a complete Web service definition requires content from each of these sections, a specific WSDL file does not need to include all three sections.

The Import Section

The Import section integrates content from other WSDL files into the current WSDL file. Like a CORBA IDL file, you can build a complex WSDL file by simply importing other WSDL files. Inclusion of import elements is optional, but its use greatly facilitates writing and maintenance of large, or complex, WSDL files.

The Logical Section

The Logical section includes a programming language, marshalling schema, and protocol neutral description of the Web service. This section of the WSDL file describes the data types and operations offered by the Web service. It is composed of three subsections.

The types subsection includes the definitions for complex data types used within an application. This subsection is an XML schema document that defines the format of these types. In creating a WSDL file, you may either include the XML schema as part of the file or import an existing schema file. By using file imports, you can maintain your type definitions in a separate file that is used by multiple applications. The types subsection describes how the data will be represented within your application's code. Each Web service development tool will map these data definitions into programming language specific data types and classes.

The following WSDL file fragment illustrates the contents of the types subsection. There are two data types defined: InParameter and OutParameter. Both types represent string values. Do not be mislead by the names: InParameter and OutParameter. The types can be used to represent any string value.

```
<types>
 <schema targetNamespace="http://www.iona.com/tutorial"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <simpleType name="InParameter">
   <restriction base="xsd:string"/>
  </simpleType>
  <simpleType name="OutParameter">
   <restriction base="xsd:string"/>
  </simpleType>
 </schema>
</types>
```

The message subsection describes how the data will be combined to form Web service requests and responses. For example, your messages might specify that a Web service request will require two pieces of data, termed parts, while the corresponding response includes only a single part.

The following WSDL file fragment illustrates the contents of the message subsection. There are two message definitions. Each message contains a single part. Do not be mislead by the names of the messages or parts. Regardless of the assigned name, the messages could be used to represent either a Web service request or response.

```
<message name="RequestMessage">
 <part name="InPart" type="tns:InParameter"/>
</message>

<message name="ResponseMessage">
 <part name="OutPart" type="tns:OutParameter"/>
</message>
```

The portType subsection describes how messages will be combined to define the operations available from the Web service. For example, a request/response type operation will specify one input message and one output message. A portType may include one, or more, operation

definitions. Each Web service development tool will map the portType to a class, each operation to a method in the class definition, and each message to either the input or output parameters of a method.

The following WSDL file fragment illustrates the contents of the portType subsection. In Artix, the portType name will become the name of the class that implements the Web service. This class will contain the method sayHi, whose signature includes an in parameter corresponding to the input element and an out parameter corresponding to the output element.

```
<portType name="TutorialPortType">
 <operation name="sayHi">
  <input message="tns:RequestMessage" name="sayHiRequest"/>
  <output message="tns:ResponseMessage" name="sayHiResponse"/>
 </operation>
</portType>
```

The syntax and format of the logical section is standardized through specifications issued by the World Wide Web Consortium (W3C). All Web service development tools must support these specifications to insure inter-operability between Web services developed with different tools.

The Physical Section

The Physical section includes the data marshalling schema and transport specific content, and describes the interaction of a Web service application with the runtime environment. The information in this section is specific to your current application. It is composed of two subsections.

The binding subsection describes how the data will be encoded during transmission, while the service subsection provides information specific to the transport protocol.

For standard SOAP encoded Web services, there are two formats to the binding subsection: rpc/encoded and document/literal. The syntax and contents of both formats are described in W3C specifications. For this reason, the contents of the binding element, and its child elements, is relatively sparse, as each Web service product implements the same specification and the interpretation of the marshalling schema and format can be coded into the product.

Artix Encompass Advanced and Enterprise, and Artix Relay Advanced and Enterprise, define alternative marshalling schemas and formats for the binding subsection. In WSDL files using these extensions, the contents of

the binding subsection will be more complex. Artix provides command line and graphical utilities that generate these more complex bindings, so you will not need to hand edit your WSDL files.

The following WSDL file fragment illustrates the contents of the binding and service subsections. In the second and third lines, you can see that the TutorialPortType_SOAPBinding describes the marshalling of data for the TutorialPortType. Each binding element is associated with only one portType, although a WSDL file may contain multiple binding elements associated with the same portType. Note that the binding specifies the rpc/encoded format. Later in this tutorial you will use the Artix Designer graphical users interface to write a WSDL file using the document/literal format.

```
<binding
   name="TutorialPortType_SOAPBinding"
   type="tns:TutorialPortType">
 <soap:binding
    style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="sayHi">
  <soap:operation soapAction="" style="rpc"/>
  <input name="sayHiRequest">
   <soap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://soapinterop.org/" use="encoded"/>
  </input>
  <output name="sayHiResponse">
   <soap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://soapinterop.org/" use="encoded"/>
  </output>
 </operation>
</binding>

<service name="HelloWorldService">
 <port
     binding="tns:TutorialPortType_SOAPBinding"
     name="HelloWorldPort">
  <soap:address location="http://localhost:9000"/>
 </port>
</service>
```

The service subsection is associated with the binding TutorialPortType_SOAPBinding. Each service element is associated with only one binding, although a WSDL file may include multiple service elements associated with the same binding.

Although the W3C provides specifications for some binding and service definitions (for example, the Simple Object Access Protocol [SOAP] binding), it is permissible for Web service development tools to define alternative binding and service representations. To support multiple data encoding schemas and transport protocols, Artix extends the W3C specifications. These Artix extensions are provided as components of the Artix Encompass Advanced and Enterprise and Artix Relay Advanced and Enterprise products. The Artix Encompass Standard product is limited to SOAP encoding over the HTTP transport.

Namespace Definitions

Every element in a WSDL file must belong to a namespace. Namespace declarations are scoped. A declaration may exist globally over the entire WSDL document or locally within an element and its enclosed child elements.

Namespaces are identified through namespace prefixes, which are generally defined within the opening root element of the WSDL file. Prefixes defined within the root element have global scope and are available throughout the entire WSDL file. However namespaces may be defined, or redefined, within an element. In this case, the scope of the prefix applies only to the element and its child elements.

If an element name is not qualified with a namespace prefix, the element belongs to the default namespace. When writing a WSDL file, you can redefine the default namespace within an element. By redefining the default namespace locally, you can reduce the effort needed to define the child elements contained within this element.

Later in this tutorial you will use the Artix Designer to write a WSDL file. The designer completely manages the namespace and prefix declarations. You will not need to edit these entries.

The following WSDL file fragment illustrates the contents of the opening root <definitions> element. This element contains the global namespace prefixes that are themselves prefixed with xmlns, which corresponds to the default namespace.

```
<definitions
  name="HelloWorldTutorial"
  targetNamespace="http://www.iona.com/tutorial"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.iona.com/tutorial"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Let's review what each entry represents.

The name attribute is an arbitrary, user-defined name assigned to this WSDL file.

The targetNamespace attribute is an arbitrary, user-defined identifier for the default namespace within this WSDL file. Although the value of this attribute appears to be an Internet URL, it need not actually represent a Web page. The URL format is used to insure uniqueness; you can use any unique content for this value. Note that the value of the targetNamespace attribute and the value of the xmlns:tns namespace prefix are identical. Elements within the WSDL file prefixed with tns are also assigned to the target namespace.

The attribute xmlns defines the default namespace and corresponds to the schema that defines the structure of a WSDL document. This entry is a valid URL and you can use it to retrieve a copy of the XML schema file that describes the WSDL file contents. Elements within your WSDL file that do not include a namespace prefix become members of this namespace. These elements must be defined in the XML schema file available at http://schemas.xmlsoap.org/wsdl/.

The xmlns:soap attribute defines the namespace that must be used when adding elements that describe a SOAP binding. Again this entry is a valid URL and you can use it to retrieve a copy of the XML schema file that describes the SOAP binding. You can see how this namespace prefix is used in the WSDL file fragment described in "The Physical Section".

The xmlns:tns attribute is the namespace prefix for elements defined in this WSDL file document. Its value is assigned by the user and is identical to the value of the targetNamespace attribute. You use the tns prefix to refer to the original default namespace within elements that have a locally defined target namespace.

The attribute xmlns:xsd defines the namespace for the basic XML types. This too is a valid URL that you can use to obtain further information about the XML basic types.

Finally, the attribute xmlns:wsdl also defines the default namespace; it is the same value as the xmlns attribute. You use this prefix within an element where you have redefined the default namespace. For example, in the types element,

```
<types>
 <schema targetNamespace="http://www.iona.com/tutorial"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <simpleType name="InParameter">
   <restriction base="xsd:string"/>
  </simpleType>
  <simpleType name="OutParameter">
   <restriction base="xsd:string"/>
  </simpleType>
 </schema>
</types>
```

the default namespace is redefined to be the value associated with the xmlns:xsd prefix — http://www.w3.org/2001/XMLSchema. This is because this element contains many child elements that are described within the this namespace and by redefining the default namespace it is no longer necessary to include the xsd prefix with each element. However, since there may also be a need to use an element that is defined within the http://schemas.xmlsoap.org/wsdl/ namespace, you now need a corresponding prefix, which is defined locally within the opening <schema> element.

A Complete WSDL File

The HelloWorld Web Service

The following WSDL file describes a simple HelloWorld Web service. In the earlier sections of this chapter, you reviewed the contents of this file.

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

<definitions
  name="HelloWorldTutorial"
  targetNamespace="http://www.iona.com/tutorial"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.iona.com/tutorial"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<types>
 <schema targetNamespace="http://www.iona.com/tutorial"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <simpleType name="InParameter">
   <restriction base="xsd:string"/>
  </simpleType>
  <simpleType name="OutParameter">
   <restriction base="xsd:string"/>
  </simpleType>
 </schema>
</types>

<message name="RequestMessage">
 <part name="InPart" type="tns:InParameter"/>
</message>

<message name="ResponseMessage">
 <part name="OutPart" type="tns:OutParameter"/>
</message>
```

```
<portType name="TutorialPortType">
 <operation name="sayHi">
  <input message="tns:RequestMessage" name="sayHiRequest"/>
  <output message="tns:ResponseMessage" name="sayHiResponse"/>
 </operation>
</portType>

<binding
   name="TutorialPortType_SOAPBinding"
   type="tns:TutorialPortType">
 <soap:binding
    style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="sayHi">
  <soap:operation soapAction="" style="rpc"/>
  <input name="sayHiRequest">
   <soap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://soapinterop.org/" use="encoded"/>
  </input>
  <output name="sayHiResponse">
   <soap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://soapinterop.org/" use="encoded"/>
  </output>
 </operation>
</binding>

<service name="HelloWorldService">
 <port
     binding="tns:TutorialPortType_SOAPBinding"
     name="HelloWorldPort">
  <soap:address location="http://localhost:9000"/>
 </port>
</service>

</definitions>
```

You will use this file in the following chapter to create a Web service application.

CHAPTER 3

Coding the Web Service

Once you have a WSDL file, you can generate code and develop a Web service application. The discussion in this chapter illustrates how to use the Artix™ wsdltocpp utility to generate C++ code from a WSDL file.

In This Chapter

This chapter discusses the following topics:

The wsdltocpp Utility

Generating C++ Code

Once you have a WSDL file, whether you write it yourself or obtain it from another source, you will want to write an application. With Artix Encompass Standard, you can write a client application against an existing Web service, you can write a server application that implements the Web service, or you can write both the client and server applications. The wsdltocpp utility is a command line tool that you will use to generate C++ code from a WSDL file.

wsdltocpp Utility Command Line Options

You control the output of the wsdltocpp command line utility through command line options. By specifying the appropriate options, you can generate exactly the code you need. The syntax used to invoke the wsdltocpp utility is:

```
wsdltocpp [options] {WSDL-URL}
```

Where {WSDL-URL} is the path, or Web location, of the WSDL file, and options may be:

```
-e Web-service-name
   The value of the name attribute in the <service> element.  If
   the WSDL file includes multiple <service> elements, and the -e
   option is not specified, the value defaults to the name of the
   first <service> element in the WSDL file.
-t port
   The value of the name attribute in the <port> element.  If a
   <service> element contains multiple <port> elements, and the
   -t option is not specified, the value defaults to the name of
   the first <port> element.  If neither the -e nor -t option is
   specified, the first <port> element within the first
   <service> element in the WSDL file is used for code
   generation.
-d output-directory
   The directory into which to generate the code.
   Defaults to the directory in which the wsdltocpp utility runs.
-n namespace
   The C++ namespace for the generated code.
   Defaults to the global namespace.
```

```
-impl
   Whether to generate starting point code for the C++ class into
    which you will code the Web service implementation.
-m { NMAKE | UNIX }
   Whether to generate a makefile for the selected platform.
   Choose either NMAKE for Windows or UNIX for Unix.
-server
   Whether to generate server stub classes only.
-client
   Whether to generate client proxy classes only.
-sample
   Whether to generate starting point code for client and/or
    server mainline applications.  Works in conjunction with the
    -server and -client options.
-?
-flags
   Usage information.
```

There are other options in addition to those described. These additional options are not, however, available within the Artix Encompass Standard product and will not be discussed in this tutorial.

Specifying the Service/Port

If your WSDL file includes multiple <service> elements, or multiple <port> elements within a single <service> element, you need to specify what <service> and/or <port> should be referenced during the code generation process. You use the -e and -t command line options to specify these values. For the simple HelloWorldTutorial.wsdl file developed in the previous chapter, there is only a single <service> element containing a single <port> element. Consequently, you will not need to use these options when generating code from this WSDL file.

Specifying the C++ Namespace

Generated C++ code should be included within a C++ namespace. You use the -n option to provide the name of the namespace. Use of this option is not required, but it is good programming style to generate code within a namespace.

Generating the Implementation Class

The wsdltocpp utility will generate starting point code for the Web service implementation class when you supply the -impl option. There is no way to specify names for the generated files; the names of the generated files are derived from either the portType name or the name of the WSDL file.

Therefore, if you use the -impl option multiple times, the starting point code will be regenerated, wiping out any code you may have added to an earlier version of the generated files.

Application Specific Code Generation

You can control whether code is generated only for client applications, only for server applications, or for both types of applications. Use the -client option to restrict code generation to client related files; use the -server option to restrict code generation to server related files.

The -sample option indicates whether starting point client and/or server mainline code should be generated. You must be careful not to overwrite the client mainline code once you have begun coding your application. If you need to rerun the wsdltocpp utility, be certain not to use the -sample option.

Generating the Makefile

If desired, the wsdltocpp utility will create a makefile. The makefile will be complete for the type of application you are creating. That is, if you are only building a client application, the makefile will not include any references to files and classes specific to server applications.

Generating Code

Installing Artix

You must first install the Artix Encompass Standard product. Follow the instructions in the installation guide. For Windows NT4, service pack 6a, Windows 2000 Professional, service pack 3, or Windows XP Professional, you will need Microsoft Visual Studio or Microsoft Visual C++, v6.0, service pack 3 or higher, or Microsoft Visual C++, v7.0.

Artix requires the Java Runtime Environment v1.4.1. If necessary, Artix will install the JRE during the installation process. If you already have the v1.4.1 JRE or JDK installed, you will be able to run Artix against your existing installations. If you are using an existing JRE or JDK installation, you must set the environment variable JAVA_HOME to the installation directory. If you are using the JRE installed by Artix, you do not need to set the JAVA_HOME environment variable.[1]

Configuring Artix

During the installation process, Artix creates two configuration files. The file <installationDirectory>\artix\1.3\etc\domains\artix.cfg is the main configuration file. During start-up, every Artix process reads this file. For the purposes of this tutorial you do not need to be concerned with the contents of this file. As you develop more involved applications, you will extend and edit this file.

The file <installationDirectory>\artix\1.3\bin\artix_env.bat is used to set the Artix development and runtime environments. You must run this file in every command window before building or running an Artix application.

Creating the Directory Structure

After installing Artix, a number of demo applications will be available in the <installationDirectory>\artix\1.3\demos directory. After you complete this tutorial you should review each of these demos.

1. If you have an earlier version of the JRE or JDK installed on your computer, for example, v1.3.x, you may have already set a JAVA_HOME environment variable. You must remove this environment variable for Artix to work properly with the JRE included as part of the Artix installation. There is no need to remove the existing JRE/JDK. You must not remove the JAVA_HOME environment variable if it points to your JRE/JDK v1.4.1 installation.

For this tutorial, you will add another subdirectory under the demos directory. Under the demos directory, create the subdirectory Tutorial. Under the Tutorial directory, create the subdirectories client and server.

Copying the WSDL File

Return to the previous chapter and use the Adobe Reader Select Text tool to copy the contents of the HelloWorldTutorial.wsdl file into a notepad or wordpad document. Since the file spans two pages, you will need to copy and paste the contents from the first page and then copy and paste the contents from the second page. Save the file as HelloWorldTutorial.wsdl into both the Tutorial\Client and the Tutorial\server directories. To view the contents with formatting, open the file in your Internet Explorer browser.

Generating the Client Application Code

You will separately generate the client and server application code in their respective directories.

1. Open a command window to the <installationDirectory>\artix\1.3\bin directory and run the artix_env.bat file.
2. Move to the <installationDirectory>\artix\1.3\demos\Tutorial\client directory by issuing the command:

```
cd ..\demos\Tutorial\client
```

3. Generate the client application with the command:

```
wsdltocpp -n ArtixDemo -client -sample -m NMAKE
   HelloWorldTutorial.wsdl
```

The following files are generated:

- Tutorial.h: A header file that defines the method signatures for the Web service.
- TutorialClient.h, TutorialClient.cxx: Header and implementation files that define the client proxy class. This client proxy class will implement the virtual sayHi method. You do not need to edit the code in these files, but you should review the contents of the header file. Note that there are multiple constructors defined. In

this tutorial your code will use the first constructor, which does not require any input from your code. As you develop more complex applications you will learn the value of the alternative constructors. The alternative constructors will not be discussed in this tutorial.

- SampleClient.cxx: The starting point code for your client application. In this tutorial, you will need to flush out the coding in this file.
- HelloWorldTutorial_wsdlTypes.h, HelloWorldTutorial_wsdlTypes.cxx: Header and implementation files that include the definitions for the classes that represent the data types defined in the WSDL file <types> section. *You must review the contents of the header file, from which you will learn the APIs needed to work with the generated type definitions.*
- HelloWorldTutorial_wsdlTypesFactory.h, HelloWorldTutorial_wsdlTypesFactory.cxx: Header and implementation files for factory classes that create instances of your application specific data types. You do not need to review the contents of these files.
- makefile: A makefile that you can use the build the client application.

Generating the Server Application Code

To generate code for the server application:

1. Move to the Tutorial\server directory by issuing the command:

```
cd ..\server
```

2. Generate the server application with the command:

```
wsdltocpp -n ArtixDemo -server -sample -m NMAKE -impl
   HelloWorldTutorial.wsdl
```

The following files are generated:

- Tutorial.h: A header file that defines the method signatures for the Web service.

- TutorialServer.h, TutorialServer.cxx: Header and implementation files that define the server stub class. You do not need to review the contents of these files.
- SampleServer.cxx: The starting point code for your server mainline application. In this tutorial, you will not need to add code to this file. In more complex applications, you may need to extend the generated code.
- TutorialImpl.h, TutorialImpl.cxx: Header and implementation files that contain starting point code for your Web service implementation class. For this tutorial you will need to add coding to the method bodies corresponding to the Web service operations. In more complex applications, you may need to edit the header file as well as add code to the implementation file.
- HelloWorldTutorial_wsdlTypes.h, HelloWorldTutorial_wsdlTypes.cxx: Header and implementation files that include the definitions for the classes that represent the data types defined in the WSDL file <types> section. *You must review the contents of the header file, from which you will learn the APIs needed to work with the generated type definitions.*
- HelloWorldTutorial_wsdlTypesFactory.h, HelloWorldTutorial_wsdlTypesFactory.cxx: Header and implementation files for factory classes that create instances of your application specific data types. You do not need to review the contents of these files.
- makefile: A makefile that you can use the build the server application.

Completing the Coding

The Implementation Class

The TutorialImpl.cxx file contains compilable code, but there is no processing logic in the method bodies. In this demo application, you only need to add processing logic to the implementation class' sayHi method.

In a text editor, open the TutorialImpl.cxx file and note the signature for the sayHi method.

```
void
TutorialImpl::sayHi(
    const ArtixDemo::InParameter InPart,
    ArtixDemo::OutParameter & OutPart
) IT_THROW_DECL((IT_Bus::Exception))
{
}
```

The method includes two parameters, the first representing the part within the input message and the second representing the part within the output message. The return type is void.

All C++ methods in Artix have void return types and output is always represented by out parameters. At first this may seem to be an inconvenience. But when you consider the facts that input and output messages could include multiple parts and that WSDL has no concept of a return value, only input and output messages, this approach makes sense. In parameters correspond to the parts of the input message and out parameters correspond to the parts of the output message. Since an output message does not assign greater importance to one of its possible multiple parts, it would be impossible for the code generating logic to select which part should correspond to a return value.

You must now add processing logic to the sayHi method. The desired processing is straight forward — just return a message that includes the input. For example, Hello Artix User, where Artix User corresponds to the value of InPart.

This seems simple. InParameter and OutParameter both correspond to an xsd:string, so it would seem that you could simply concatenate "Hello " with InPart and assign the new string to OutPart. But, not so fast.

Look into the HelloWorldTutorial_wsldTypes.h file and find the definitions for InParameter and OutParameter. They are not strings, but classes that encapsulate a member variable that is a string. To get and set the value of this member variable you must use the accessor methods.

```
namespace ArtixDemo
{
    . . .

    class InParameter : public IT_Bus::AnySimpleType
    {
      public:
        InParameter();
        InParameter(const InParameter & value);
        . . .

        void set_value(const IT_Bus::String & value);
        const IT_Bus::String & get_value() const;

      private:
        IT_Bus::String m_val;

    };
    . . .

    class OutParameter : public IT_Bus::AnySimpleType
    {
      public:
        OutParameter();
        OutParameter(const OutParameter & value);
        . . .

        void set_value(const IT_Bus::String & value);
        const IT_Bus::String & get_value() const;

      private:
        IT_Bus::String m_val;

    };
    . . .
};
```

Consequently, the code you add to the sayHi method body becomes:

```
OutPart.set_value("Hello " + InPart.get_value());
```

Building the Web Service Server Application

Now that you have completed coding the implementation object, you can build the application.

1. Open a command window to the <installationDirectory>\artix\1.3\bin directory and run the artix_env.bat file.
2. Move to the <installationDirectory>\artix\1.3\demos\Tutorial\server directory by issuing the command:

```
cd ..\demos\Tutorial\server
```

3. Build the server application with the command:

```
nmake all
```

This creates the server executable file server.exe.

The Client Application

For this demo, you only need to work with the SampleClient.cxx file. Although this file is compilable, it does not actually invoke the Web service operations. Open the file and examine the generated code.

```
#include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_cal/iostream.h>

#include "TutorialClient.h"

IT_USING_NAMESPACE_STD
using namespace ArtixDemo
using namespace IT_Bus;

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

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

        // Sample invocation calls are shown in
        // commented lines below.
        /*
         ArtixDemo::InParameter   InPart_0;
         ArtixDemo::OutParameter   OutPart_1;
         client.sayHi ( InPart_0,  OutPart_1);
         */
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.Message()
             << endl;
        return -1;
    }
    return 0;
}
```

Note that the code generation process flushed out a simple invocation of the sayHi method, but the code is commented out and there is no value assigned to the in parameter and no output statement to display the value returned in the out parameter.

You need to remove the comment delimiters and edit the code as follows:

```
ArtixDemo::InParameter    InPart_0;
ArtixDemo::OutParameter   OutPart_1;

InPart_0.set_value("Artix User");

client.sayHi ( InPart_0,  OutPart_1);

cout << "sayHi returned: " + OutPart_1.get_value() << endl;
```

Building the Web Service Client Application

Now that you have completed coding the client mainline, you can build the application.

1. Open a command window to the <installationDirectory>\artix\1.3\bin directory and run the artix_env.bat file.
2. Move to the <installationDirectory>\artix\1.3\demos\Tutorial\client directory by issuing the command:

```
cd ..\demos\Tutorial\client
```

3. Build the client application with the command:

```
nmake all
```

This creates the client executable file client.exe.

Running the Application

Set the Runtime Environment

Before you can run the application, you must set the environment.

1. Open a command window to the <installationDirectory>\artix\1.3\bin directory and run the artix_env.bat file.

Start the Server Application

To start the server application:

1. Move to the <installationDirectory>\artix\1.3\demos\Tutorial\server directory by issuing the command:

```
cd ..\demos\Tutorial\server
```

2. Start the server application with the command:

```
start server
```

A new command window opens and the server application starts.

Run the Client Application

To run the client application:

1. Move to the <installationDirectory>\artix\1.3\demos\Tutorial\client directory by issuing the command:

```
cd ..\client
```

2. Run the client application with the command:

```
client
```

The client application invokes on the Web service and displays the return.

Stop the Server Application

Issue Ctrl-C in the command window running the server application.

CHAPTER 4

Using the Artix™ Designer

In the previous chapters, you used a pre-written WSDL file to build your Web service application. In this chapter, you will use the Artix Designer to write an equivalent WSDL file. You will also use the designer to generate the starting point code.

In This Chapter

This chapter discusses the following topics:

The Artix Designer

The Designer

The Artix Designer is a graphical user interface based application through which you will write and/or edit WSDL files. Although there are other XML editing tools that you could use to write a WSDL file, the Artix Designer has an understanding of the Artix WSDL extensions and is a much easier way to write the WSDL files used in an Artix application. For example, the designer will automatically add the required namespace declarations and prefix definitions when you build Artix applications that involve other data marshalling schemas, transport protocols, or routing.

The designer is also integrated with the Artix command line tools, for example, the wsdltocpp utility, so that you can also use the designer to generate starting point code. Integration with other command line utilities allows the designer to import IDL files and convert their contents into WSDL, generate starting point code for Java Web service applications, or convert WSDL files into IDL.

Starting the Artix Designer

In Windows you have two ways to start the Artix Designer.

1. You can select the Start > Programs > IONA Artix 1.3 > IONA Artix 1.3 > Designer menu entry.
2. You can open a command window to the directory <installationDirectory>\artix\1.3\bin and run the batch file start_designer.bat.

Selecting the menu entry simply runs the start_designer.bat file.

The Artix Project

Artix Projects

Similar to other interactive development environment applications, the Artix Designer maintains all of the files comprising an application within a larger entity called a project.

When you start the designer, you have the option of creating a new project, returning to an existing project, or simply starting the designer.

The name you assign to a project becomes the name of a directory under which the project files are stored. Artix will not let you give two projects the same path and name. Consequently, when creating a new project you should not create the project directory manually; let the Artix Designer create the directory.

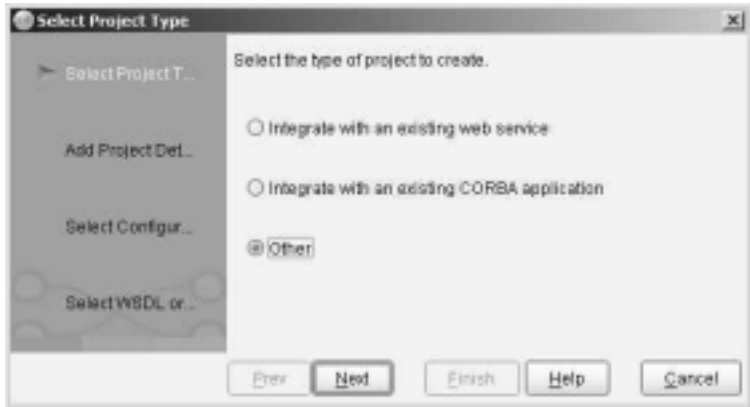
It is not necessary to create all of the application files using the designer. For example, one approach is to import an existing WSDL file into the designer and then edit the file, if necessary. Alternatively, you can import a CORBA IDL file into the designer, which then transforms the contents of the IDL file into the equivalent WSDL file.

In this tutorial you will use the designer to create a new project and write an original WSDL file. You will then use the designer to generate starting point C++ code.

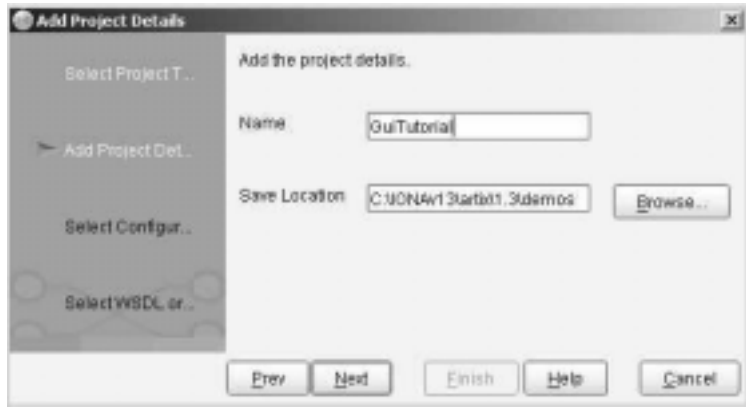
Creating a New Project

After starting the Artix Designer, you are presented with the Welcome window. Select the Create a New Project radio button and click on the OK command button.

In the Select Project Type window, select the Other radio button and click the Next command button.



In the Add Project Details window, name your project GuiTutorial, enter, or browse, to the directory that will contain your project, and click the Next command button.



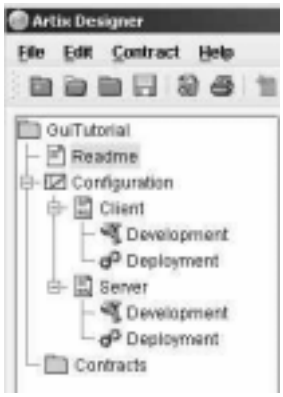
In the Select Configuration window, select the Embedded radio button and click the Next command button.

An Embedded application means that the Web service endpoints, the client and/or server applications, will be Artix applications, written against the Artix API. A Standalone application means that the Web service endpoints *may* be applications written with other middleware technologies, for example, CORBA, WebSphere™ MQ, TIBCO Rendezvous™, or Tuxedo. In this situation, the Artix process serves as a switch, bridge, or router, transforming requests from one marshalling scheme and transport to a different marshalling scheme and/or transport.

The Artix Encompass Standard product will only create Embedded applications. The Artix Encompass Advanced and Enterprise or the Artix Relay Advanced and Enterprise products support development of Standalone applications.

You can use the Select WSDL or IDL window to import an existing WSDL or IDL file. For example, you could import the WSDL file used in the previous chapters. In this tutorial, you will be using the Artix designer to write a new WSDL file, so simply click the Finish command button.

The designer creates the project, which it displays as a hierarchy of directories. You must save the project to actually create the project directories on your drive; the Contracts directory will not be created until you either create, or import, a WSDL file.



The Client and Server icons represent WSDL files. If you highlight one of these icons and then select the WSDL tab at the bottom of the window,

you can view the contents of the corresponding WSDL file. Since you have not yet supplied any content for the WSDL file, these files only contain the opening root element <definitions>, which includes common namespace prefix definitions.

Writing the WSDL File

Approaches

Now that you have an Artix project, you have several approaches to writing a WSDL file. If you are only interested in writing a Web service server application, you could add all of the WSDL file content to the Server.wsdl file represented by the Server icon.

Alternatively, if you are only interested in writing a client application using an existing WSDL file, you could import that file into the project.

It is more likely, however, that you will be interested in writing both client and server applications against the same WSDL file. The Artix Designer has been optimized for this situation in that it allows you to write one WSDL file, or several WSDL file fragments, and reuse them in multiple applications.

In this tutorial, you will write one WSDL file for both the client and server applications. Rather than adding the contents to the WSDL files represented by the Client and Server icons, you will write a third WSDL file and include it by reference into the Client.wsdl and Server.wsdl files. Your client application will use the file Client.wsdl while your server application will use the file Server.wsdl, both of these files will include all of the content of the WSDL file you will write. The only differences between Client.wsdl and Server.wsdl will be the values assigned to the name, targetNamespace, and xmlns:tns attributes in the opening <definitions> element.

Creating the WSDL File

You will write the WSDL file as an entry under the Contracts icon. You will then use the designer to import the completed file into the files Client.wsdl and Server.wsdl.

Highlight the Contracts icon and either select the File > New Contract menu entry, or right click on the Contracts icon and select New Contract from the popup menu. The New Contract — Artix Designer window opens.

- Enter HelloWorldGuiTutorial into the Name text box.
- Enter http://www.iona.com/guitutorial into the TargetNamespace text box.
- Click the OK command button.

These entries become the values of the name, targetNamespace, and xmlns:tns attributes in the opening <definitions> element. You can view the contents of the WSDL file by highlighting the HelloWorldGuiTutorial icon and clicking on the WSDL tab. Currently the WSDL file only includes the opening <definitions> element.

Define Types

The <types> section of the WSDL file contains your data type definitions. For this simple application you have several choices. You could not bother to define unique types for the application and use a basic type, for example, xsd:string, as message parts. Alternatively, you could define simple types, that is new types that are derived from an existing xsd type. This was the approach used in the earlier example. Finally, you could define element types, which are wrappers around other defined types. This approach is especially useful if your types are complex or highly structured, for example, a structure or array. Additionally, using elements as message parts allows selection of the document/literal encoding format for your binding.

Defining the Simple Types

In this example, you will employ the third approach. You will first need to define simple types derived from xsd:string and then define element types that contain these simple types.

1. Highlight the HelloWorldGuiTutorial icon and select the Contract > New > Type menu entry, or right click on the HelloWorldGuiTutorial icon and select New > Type from the popup menu.
2. In the New Type — Artix Designer — Select WSDL window, select the Add to existing WSDL "HelloWorldGuiTutorial" radio button and click the Next command button.
3. Throughout the entire WSDL file creation process you will add your new content to the current WSDL file. Selecting the Add to new WSDL radio button will create another WSDL file that will include select content. This is the approach you would follow if you want to create WSDL file fragments for reuse.
4. In the New Type — Artix Designer — Define Type Properties window, enter InParameter into the Name text box and select the simpleType radio button. Click the Next command button.

5. In the New Type — Artix Designer — Define Type Attributes window, select xsd:string from the Base Type drop down list. The remaining controls are used to further restrict the simple type, for example, limiting the length of the string to a specific number of characters or to one of a restricted number of entries. For this example, you do not need these additional restrictions. Simply click on the Next command button.
6. In the New Type — Artix Designer — View Summary window, you can review the content that will be added to the WSDL file. Click on the Finish command button.
7. Repeat the process, creating a second simple type named OutParameter. Note that the Base Type drop down list now includes tns:InParameter as a valid type. Be careful, as newly defined types are added to the top of the list; you will need to scroll down the list to find the xsd:string entry.

Highlight the HelloWorldGuiTutorial icon and select the WSDL tab. Review the current contents of the WSDL file. Note that the <types> section has be added to the file.

Defining the Element Types

You now want to define element types that wrap each of your simple types. This process is identical to defining a simple type except that you must select the element radio button in the New Type — Artix Designer — Define Type Properties window.

In the New Type — Artix Designer — Define Type Attributes window you are only presented with a Type drop down list. Since an element type is simply a wrapper around another type, there are no additional options.

Create two element types:

- InElement of type tns:InParameter.
- OutElement of type tns:OutParameter.

Again, highlight the HelloWorldGuiTutorial icon and review the contents of the WSDL file. Note that the <types> section now includes four definitions:

- Simple type InParameter, of type xsd:string.
- Simple type OutParameter, of type xsd:string.
- Element type InElement, of type tns:InParameter.

- Element type OutElement, of type tns:OutParameter.

The following WSDL file fragment summarizes the content of the HelloWorldGuiTutorial.wsdl file.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorldGuiTutorial"
             targetNamespace="http://www.iona.com/guitutorial"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:tns="http://www.iona.com/guitutorial"
             xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <types>
  <schema targetNamespace="http://www.iona.com/guitutorial"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <simpleType name="InParameter">
    <restriction base="xsd:string"/>
   </simpleType>
   <simpleType name="OutParameter">
    <restriction base="xsd:string"/>
   </simpleType>
   <element name="InElement" type="tns:InParameter"/>
   <element name="OutElement" type="tns:OutParameter"/>
  </schema>
 </types>

</definitions>
```

Define the Messages

Now that you have defined the required types, you can begin to define the messages. Your types will be used as the message parts.

1. Highlight the HelloWorldGuiTutorial icon and select the Contract > New > Message menu entry, or right click on the HelloWorldGuiTutorial icon and select New > Message from the popup menu.
2. In the New Message — Artix Designer — Select WSDL window, select the Add to existing WSDL "HelloWorldGuiTutorial" radio button and click the Next command button.

3. In the New Message — Artix Designer — Define Message Properties window, enter RequestMessage into the Name text box. Click the Next command button.
4. In the New Message — Artix Designer — Define Parts window, enter InPart into the Name text box and select tns:InElement from the Type drop down list. *Be careful — Do not select tns:InParameter from the Type drop down list.*

   Now click the Add command button; your part is added to the Part List control. If your message requires multiple parts (which is not the situation in this example), you would simply define another part and add it to the Part List.

   Finally, click the Next command button.
5. In the New Message — Artix Designer — View Summary window, you can review the content that will be added to the WSDL file.
6. Check the Check here to create another message checkbox and click the Next command button.
7. Create a second message – ResponseMessage – with a part named OutPart of type tns:OutElement. In the New Message — Artix Designer — View Summary window click the Finish command button.

Highlight the HelloWorldGuiTutorial icon, click on the WSDL tab, and review the contents of the WSDL file. The following WSDL file fragment shows the new content of the HelloWorldGuiTutorial.wsdl file.

```
<message name="RequestMessage">
 <part element="tns:InElement" name="InPart"/>
</message>

<message name="ResponseMessage">
 <part element="tns:OutElement" name="OutPart"/>
</message>
```

Define the portType

A portType contains operations, which are comprised of one or more messages. A oneway operation will include only an input message; the client application will not receive a response from the Web service. A request:response operation includes an input message, an output message, and zero, or more, fault messages. Defining and coding fault messages will be discussed in the following chapter.

For this example, you will define a portType that includes one request:response operation –sayHi – that uses RequestMessage as its input and ResponseMessage as its output. There is nothing significant about the names assigned to the messages or parts; name assignments are to assist the developer, Artix doesn't care what names are used. An identical application could be created by naming the messages One and Two and the parts X and Y.

1. Highlight the HelloWorldGuiTutorial icon and select the Contract > New > Port Type menu entry, or right click on the HelloWorldGuiTutorial icon and select New > Port Type from the popup menu.
2. In the New Port Type — Artix Designer — Select WSDL window, select the Add to existing WSDL "HelloWorldGuiTutorial" radio button and click the Next command button.
3. In the New Port Type — Artix Designer — Define Port Type Properties window, enter GuiTutorialPT into the Name text box. Click the Next command button.
4. In the New Port Type — Artix Designer — Define Port Type Operations window, enter sayHi into the Name text box. Select Request-response from the Style drop down list and click the Next command button.
5. In the New Port Type — Artix Designer — Define Operation Messages window, select input from the Type drop down list and tns:RequestMessage from the Message drop down list. The Name sayHiRequest appears in the Name text box. If desired, you can change this entry to something more meaningful to your application. In this example, leave the suggested content.
6. Click the Add command button, which transfers the input message to the Operation Messages control.
7. Now, select output from the Type drop down list. Note that input no longer appears in the listing; an operation can have only one input message. Select tns:ResponseMessage from the Message drop down list. The Name sayHiResponse appears in the Name text box; leave this suggested content.
8. Click the Add command button to transfer the ouptut message to the Operation Messages control.

You can click on the Type drop down list and note that the output entry no longer appears in the listing; an operation can have only one output message. While you will not be adding fault messages to this operation, multiple fault message may be added to an operation. Consequently, you can repeat this process to add one or more fault messages to the operation.

9. Finally, click on the Next command button and review the content in the New Port Type — Artix Designer — View Port Operations Summary window. Since this example only requires one portType, click on the Next and then Finish command buttons.

Highlight the HelloWorldGuiTutorial icon, click on the WSDL tab, and review the contents of the WSDL file. The following WSDL file fragment shows the new content of the HelloWorldGuiTutorial.wsdl file.

```
<portType name="GuiTutorialPT">
 <operation name="sayHi">
  <input message="tns:RequestMessage" name="sayHiRequest"/>
  <output message="tns:ResponseMessage" name="sayHiResponse"/>
 </operation>
</portType>
```

Define the Binding

A binding describes how the messages will be marshalled. Each binding is associated with a single portType, although the same portType may be associated with multiple bindings.

In the previous example, the binding used the rpc/encoded style. In this example, you will specify the document/literal style, which is required when message parts are element types.

1. Highlight the HelloWorldGuiTutorial icon and select the Contract > New > Binding menu entry, or right click on the HelloWorldGuiTutorial icon and select New > Binding from the popup menu.
2. In the New Binding — Artix Designer — Select WSDL window, select the Add to existing WSDL "HelloWorldGuiTutorial" radio button and click the Next command button.
3. In the New Binding — Artix Designer — Select Binding Type window, select the SOAP radio button. The other binding choices are not available in the Artix Encompass Standard product. Click the Next command button.

4. In the New Binding — Artix Designer — Select Port Type window, select the GuiTutorialPT entry from the Port Type drop down list.

   Actually, since your WSDL file only contains one portType definition, this is the only entry in the list. But if there were multiple portTypes defined, you would need to select the desired portType from the list.

   Note that a suggested Binding Name is already entered into the Name text box. You can change this entry; the only requirement is that each binding in the WSDL file, if you create multiple bindings, have a unique Binding Name.

5. In the SOAP Setting control group there are two drop down list controls. From the Style list, select document, and from the Use list, select literal. If you select an invalid combination, for example rpc/encoded or document/encoded, you will not be able to move to the next window. Click the Next command button.

6. In the New Binding — Artix Designer — Edit Binding window, highlight the sayHi icon representing your operation and review the binding details. Click the Next command button.

7. In the New Binding — Artix Designer — View WSDL Contract window, you can review the new content that will be added to the WSDL file. Finally, click the Finish command button.

Highlight the HelloWorldGuiTutorial icon, click on the WSDL tab, and review the contents of the WSDL file. The following WSDL file fragment shows the new content of the HelloWorldGuiTutorial.wsdl file.

```
<binding name="GuiTutorialPT_SOAPBinding"
         type="tns:GuiTutorialPT">
 <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="sayHi">
  <soap:operation soapAction="" style="document"/>
  <input name="sayHiRequest">
   <soap:body use="literal"/>
  </input>
  <output name="sayHiResponse">
   <soap:body use="literal"/>
  </output>
 </operation>
</binding>
```

Define the Service

A service provides transport specific information. Each service element may include one, or more, port elements. The port elements must be uniquely identified through the value of the name attribute. Each port element is associated with a single binding element, although the same binding element may be associated with one, or more, port elements. In addition, a WSDL file may contain multiple service elements.

In this example, the WSDL file contains one service element, which contains a single port element.

1. Highlight the HelloWorldGuiTutorial icon and select the Contract > New > Service menu entry, or right click on the HelloWorldGuiTutorial icon and select New > Service from the popup menu.
2. In the New Service — Artix Designer — Select WSDL window, select the Add to existing WSDL "HelloWorldGuiTutorial" radio button and click the Next command button.
3. In the New Service — Artix Designer — Define Service window, enter HelloWorldService into the Name text box. Click the Next command button.
4. In the New Service — Artix Designer — Define Port window, enter HelloWorldPort into the Name text box and select GuiTutorialPT_SOAPBinding from the Binding drop down list. Actually, since your WSDL file only contains one binding definition, this is the only entry in the list. But if there were multiple bindings defined, you would need to select the desired binding from the list. Click the Next command button.
5. In the New Service — Artix Designer — Define Extensor Properties window, select SOAP from the Transport Type drop down list and enter http://localhost:9000 as the value for the location attribute. This is the only required entry, and you may specify any port number you choose. Refer to the Artix documentation for a discussion of the other extensor properties.
6. Click the Next command button. In the New Service — Artix Designer — Port Summary window, you can review the new content that will be added to the WSDL file. Finally, click the Finish command button.

Highlight the HelloWorldGuiTutorial icon, click on the WSDL tab, and review the contents of the WSDL file. The following WSDL file fragment shows the new content of the HelloWorldGuiTutorial.wsdl file.

```
<service name="HelloWorldService">
 <port binding="tns:GuiTutorialPortType_SOAPBinding"
       name="newPort">
  <soap:address location="http://localhost:9000"/>
  <http-conf:client/>
  <http-conf:server/>
 </port>
</service>
```

The elements with the namespace prefix http-conf are unique to Artix and represent the unspecified extensor properties. Note the inclusion of the http-conf namespace prefix definition in the opening definitions element.

Developing an Application

Project Summary

You have now completed writing the WSDL file. Currently the file is located under the Contracts icon within the Artix Designer and in the GuiTutorial\Contracts directory on your drive. Other than the fact that this WSDL file uses element types and document/literal encoding, this WSDL file is functionally equivalent to the file used in the previous example. You could repeat the earlier example using this WSDL file instead of the file provided in this document.

In this example, you want to use the same WSDL file for both the client and server applications. With the Artix Designer you accomplish this goal by dragging and dropping the HelloWorldGuiTutorial icon onto the Client and Server icons.

The Client Application

Highlight the HelloWorldGuiTutorial icon, hold down the left mouse button, and drag and drop the icon onto the Client icon.

Highlight the Client icon and click on the WSDL tab. Note that the Client.wsdl file now includes an import element, which identifies the HelloWorldGuiTutorial.wsdl file.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Client"
    targetNamespace="http://www.iona.com/
      artix/1.3.0/GuiTutorial/Client/Client"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:ns1="http://www.iona.com/guitutorial"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/artix/
      1.3.0/GuiTutorial/Client/Client"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <import
      location="../../Contracts/HelloWorldGuiTutorial.wsdl"
      namespace="http://www.iona.com/guitutorial"/>
</definitions>
```

Your client application will be generated from the Client.wsdl file, which incorporates the content of the HelloWorldGuiTutorial.wsdl file.

The Server Application

Highlight the HelloWorldGuiTutorial icon, hold down the left mouse button, and drag and drop the icon onto the Server icon.

Highlight the Server icon and click on the WSDL tab. Note that the Server.wsdl file now includes an import element, which identifies the HelloWorldGuiTutorial.wsdl file.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="Server"
    targetNamespace="http://www.iona.com/
      artix/1.3.0/GuiTutorial/Server/Server"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:ns1="http://www.iona.com/guitutorial"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/artix/
      1.3.0/GuiTutorial/Server/Server"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <import
      location="../../Contracts/HelloWorldGuiTutorial.wsdl"
      namespace="http://www.iona.com/guitutorial"/>
</definitions>
```

Your server application will be generated from the Server.wsdl file, which incorporates the content of the HelloWorldGuiTutorial.wsdl file.

Generating Code from Composite WSDL Files

The WSDL files Client.wsdl and Server.wsdl are composite WSDL files, that is, the description of the Web service is contained in multiple WSDL files that are imported into these files. When you generate the application code, you will use the composite WSDL file as the description of the Web service. Artix will transparently work down through the included files to find the required information.

For each WSDL file listed in the composite file, and for the composite WSDL file itself, Artix will generate four files:

- <WSDLfileName>_wsdlTypes.h
- <WSDLfileName>_wsdlTypes.cxx
- <WSDLfileName>_wsdlTypesFactory.h
- <WSDLfileName>_wsdlTypesFactory.cxx

Generating Code

The Client Application

Highlight the development icon under the Client icon. The System Development Options window appears.

For this example, select C++ from the Development Environment drop down list. The suggested entry in the Code Location text box indicates that the code will be generated into a directory under the GuiTutorial\Client directory.

Since you are generating code for a client application, you deselect the Generate Implementation Code check box. The suggested C++ namespace is derived from the project name; you are free to change this entry to a more meaningful value.

The Select Service and Select Port drop down lists are used to select the service and port against which code will be generated. Since there is only one service and port in your WSDL file, the lists contain only one entry. These drop down lists correspond to the -e and -t command line options for the wsdltocpp utility.

Select the appropriate radio button within the Generate Makefile group.

Click the Advanced Options command button. The WSDL to C++ Conversions Options — Artix Designer window opens.

Since you are building the client application, deselect the Generate Server Code checkbox. The Generate Sample Code checkbox is equivalent to the -sample option for the wsdltocpp utility and the Generate Client Code checkbox is equivalent to the -client option for the wsdltocpp utility.

Click the OK command button, then click the OK command button on the System Development Options window. A message box confirms code generation success.

The Server Application

Highlight the development icon under the Server icon. The System Development Options window appears.

For this example, select C++ from the Development Environment drop down list. The suggested entry in the Code Location text box indicates that the code will be generated into a directory under the GuiTutorial\Server directory.

Since you are generating code for a server application, you select the Generate Implementation Code check box. The suggested C++ namespace is derived from the project name; you may change this to a more meaningful value. Since you are building the client and server applications separately, you do not need to use the same C++ namespace value in both applications.

The Select Service and Select Port drop down lists are used to select the service and port against which code will be generated. Since there is only one service and port in your WSDL file, the lists contain only one entry. These drop down lists correspond to the -e and -t command line options for the wsdltocpp utility.

Select the appropriate radio button within the Generate Makefile group.

Click the Advanced Options command button. The WSDL to C++ Conversions Options — Artix Designer window opens.

Since you are building the server application, make certain to deselect the Generate Client Code checkbox. The Generate Sample Code checkbox is equivalent to the -sample option for the wsdltocpp utility and the Generate Server Code checkbox is equivalent to the -server option for the wsdltocpp utility.

Click the OK command button, then click the OK command button on the System Development Options window. A message box confirms code generation success.

Completing the Code

Close the Artix Designer

You have now finished with the Artix Designer. Close the application by selecting the File > Exit menu item. Alternatively, you can click on the close box in the upper right corner of the window, or click on the icon in the upper left corner of the window and select Close from the drop down menu.

The GuiTutorial.h File

This header file is common to both the client and server applications. It contains the signatures for each of the Web service operations. Open this file in a text editor and review the signature for the sayHi method.

```
virtual void
  sayHi(
    const GuiTutorial::InParameter & InPart,
          GuiTutorial::OutParameter & OutPart
     ) IT_THROW_DECL((IT_Bus::Exception)) = 0;
```

Note that although the message parts were defined as the element types InElement and OutElement, the method signature uses C++ classes derived from the simple types InParameter and OutParameter.

Your coding for this version of the demo application is identical to the coding you implemented in the earlier tutorial example.

The Implementation Class

The TutorialImpl.cxx file contains compilable code, but there is no processing logic in the method bodies. In the server application, you only need to add processing logic to the implementation class' sayHi method.

In a text editor, open the TutorialImpl.cxx file and note the signature for the sayHi method.

The method includes two parameters, the first representing the part within the input message and the second representing the part within the output message. The return type is void.

The code you add to the sayHi method body is:

```
OutPart.set_value("Hello " + InPart.get_value());
```

Building the Web Service Server Application

Now that you have completed coding the implementation object, you can build the application.

1. Open a command window to the <installationDirectory>\artix\1.3\bin directory and run the artix_env.bat file.
2. Move to the <installationDirectory>\artix\1.3\demos\GuiTutorial\Server\src\cpp directory by issuing the command:

```
cd ..\demos\GuiTutorial\Server\src\cpp
```

3. Build the server application with the command:

```
nmake all
```

This creates the server executable file server.exe.

The Client Application

For the client application, you only need to work with the SampleClient.cxx file.

Note that the code generation process flushed out a simple invocation of the sayHi method, but the code is commented out and there is no value assigned to the in parameter and no output statement to display the value returned in the out parameter.

You need to remove the comment delimiters and edit the code as follows:

```
GuiTutorial::InParameter    InPart_0;
GuiTutorial::OutParameter   OutPart_1;

InPart_0.set_value("Artix User");

client.sayHi ( InPart_0,  OutPart_1);

cout << "sayHi returned: " + OutPart_1.get_value() << endl;
```

Other than the C++ namespace, this coding is identical to the coding in the previous example.

Building the Web Service Client Application

Now that you have completed coding the client mainline, you can build the application.

1. Open a command window to the <installationDirectory>\artix\1.3\bin directory and run the artix_env.bat file.
2. Move to the <installationDirectory>\artix\1.3\demos\GuiTutorial\Client\src\cpp directory by issuing the command:

```
cd ..\demos\GuiTutorial\Client\src\cpp
```

3. Build the client application with the command:

```
nmake all
```

This creates the client executable file client.exe.

Running the Application

Set the Runtime Environment

Before you can run the application, you must set the environment.

1. Open a command window to the <installationDirectory>\artix\1.3\bin directory and run the artix_env.bat file.

Start the Server Application

To start the server application:

1. Move to the <installationDirectory>\artix\1.3\demos\GuiTutorial\Server\src\cpp directory by issuing the command:

```
cd ..\demos\GuiTutorial\Server\src\cpp
```

2. Start the server application with the command:

```
start server
```

A new command window opens and the server application starts.

Run the Client Application

To run the client application:

1. Move to the <installationDirectory>\artix\1.3\demos\GuiTutorial\Client\src\cpp directory by issuing the command:

```
cd ..\..\..\Client\src\cpp
```

2. Run the client application with the command:

```
client
```

The client application invokes on the Web service and displays the return.

Stop the Server Application

Issue Ctrl-C in the command window running the server application.

CHAPTER 5

Faults and Exceptions

This chapter focuses on how to declare faults in WSDL files and how to handle the corresponding C++ exceptions in Artix™ client and server applications.

In This Chapter

This chapter discusses the following topics:

Raising Exceptions

Types of Artix Exceptions

C++ exceptions may originate from three different sources.

- The Artix runtime libraries may throw a C++ exception.
- The Artix runtime services, for example, the locator service, may throw a C++ exception.
- The business logic within the Web service itself may throw a C++ exception.

In each case, the exception is returned to the client application.

The WSDL file provides no information about exceptions originating from the Artix runtime libraries as these exceptions are not directly related to your Web service contract. These exceptions are returned as subclasses of the Artix class IT_Bus::Exception. Consequently, your client code must use try{} and catch (IT_Bus::Exception){} blocks to gracefully handle possible exceptions.

Many of the Artix runtime services are described in WSDL files, and a service's operations may include fault messages. If your application uses these services, your application must also include the client-side classes generated from this WSDL file. In this case, you can use the runtime service's WSDL file, and the contents of the generated code, to understand how the WSDL faults map to C++ classes. Your application code will use these classes to handle the service's exceptions.

When you write the WSDL file that describes your Web service, you may include zero or more fault messages in each request:response operation. When you run the wsdltocpp utility, these fault messages become C++ classes that your application code will use to handle your application's exceptions.

Handling exceptions raised by either an Artix runtime service or your application's business logic is similar. You enclose your application code within a try{} block and use one, or more, catch{} blocks to handle the possible exceptions.

Handling Runtime Exceptions

Types of Runtime Exceptions

Artix includes an extensive collection of runtime exceptions, which primarily represent errors that may arise during marshalling and transport.

- IT_Bus::ConnectException
- IT_Bus::DeserializationException
- IT_Bus::IOException
- IT_Bus::NoDataException
- IT_Bus::SecurityException
- IT_Bus::SerializationException
- IT_Bus::ServiceException
- IT_Bus::TransportException
- IT_Bus::UserFaultException

These exceptions are defined in corresponding header files, which are located in the <installationDirectory>\artix\1.3\include\it_bus directory.

Since IT_Bus::Exception is the superclass for all of these exception classes, you may use the IT_Bus::Exception class' API to extract details about what caused the exception.

The IT_Bus::Exception Class API

The IT_Bus::Exception class is actually just a typedef of the IT_Bus::FWException class. The header file that includes this typedef entry is <installationDirectory>\artix\1.3\include\it_bus\types.h.

Your application code will never create an instance of a runtime exception. Consequently, the only API methods you need are used to obtain a description, and optionally a message code, describing the processing error.

The IT_Bus::Exception::Message() method returns an informative description of the error that caused the runtime exception.

The IT_Bus::Exception::Error() method returns an exception code.

Handling IT_Bus::Exception

You have already seen an example of handling the runtime exceptions. The code generated for your client application includes a try{} block around all of the application logic and a catch(IT_Bus::Exception){} block.

```
int
main(int argc, char* argv[])
{
    . . .

    try
    {
        . . .
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.Message()
             << endl;
        return -1;
    }
    return 0;
}
```

This is generally all that is needed, although your code could catch each of the runtime exceptions separately.

Working with WSDL Faults

Defining WSDL Faults

A WSDL fault is simply a message, which may contain zero or one part. A message corresponding to a WSDL fault is referenced by the <fault> child element under the <operation> element. A request:response operation may include zero, or more, child fault elements. If appropriate to the Web service, the same fault message may be associated with multiple operations.

The wsdltocpp utility creates a C++ class corresponding to the message; a message part becomes an instance variable and accessor methods are provided to manipulate the value of this variable. If a fault message needs to contain multiple parts, you need to define a complex type, which then becomes the type of the message part.

You will need to study the generated code to understand how to create and manipulate the exception class.

As with messages representing a request or response, fault messages may contain either encoded or literal element parts. The following fragment illustrates the WSDL file definition of a fault message.

```
<types>
 <schema targetNamespace="http://www.iona.com/guitutorial"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  . . .

  <complexType name="FaultDetails">
   <sequence>
    <element name="FaultMsg" type="xsd:string"/>
    <element name="FaultID"   type="xsd:int"/>
   </sequence>
  </complexType>
  <element name="DoIKnowYou" type="tns:FaultDetails"/>

 </schema>
</types>

<message name="UnknownUser">
 <part element="tns:DoIKnowYou" name="theFault"/>
</message>
```

Note that the message, UnknownUser, only contains one part, theFault, which is an instance of the element DoIKnowYou. DoIKnowYou is a wrapper around the complex type FaultDetails, which contains two pieces of information, a string message and a numeric code.

The operation that uses this fault must include a fault child element within the operation element, as illustrated in the following fragment.

```
<portType name="GuiTutorialPT">
 <operation name="sayHi">
  <input message="tns:RequestMessage" name="sayHiRequest"/>
  <output message="tns:ResponseMessage" name="sayHiResponse"/>
  <fault message="tns:UnknownUser" name="sayHiFault"/>
 </operation>
</portType>
```

If you are using the Artix Designer to create your WSDL file, you do not need to worry about how to include the fault message in the binding; the designer will handle this task.

When you run the wsdltocpp utility, two C++ classes are generated. The class FaultDetails corresponds to the complex type. This class includes variables corresponding to the FaultMsg and FaultID elements and accessor methods to manipulate these values. The class UnknownUserException corresponds to the UnknownUser message. This class includes a variable of type FautlDetails and accessor methods to manipulate this value.

Throwing the Exception

While both class definitions include a copy constructor, neither class includes a constructor that allows you to set the instance variables. Consequently, you must handle initialization in your code.

To throw the exception from your Web service's code, you must first instantiate and initialize an instance of the FaultDetails class and then use this instance to initialize an instance of the UnknownUserException class. Finally, your code throws the UnknownUserException instance.

```
FaultDetails faultData;
faultData.setFaultMsg("User unknown to me");
faultData.setFaultID(200);

UnknownUserException ex;
ex.settheFault(faultData);

throw ex;
```

Handling the Exception

The exception UnknownUserException is derived from the Artix class IT_Bus::UserFaultException, which is derived from IT_Bus::Exception.

Consequently, you must include code to catch this exception before your code that handles IT_Bus::Exception. Since the catch block receives a reference to the UnknownUserException object, your code needs to use the accessor method to obtain the FaultDetails object and then extract the FaultMsg and FaultID.

```
catch(UnknownUserException& ex)
{
  FaultDetails& fd = ex.gettheFault();
  cout << "Error Message: " << fd.getFaultMsg() << endl;
  cout << "Error ID: " << fd.getFaultID() << endl;
  return -1;
}
```

Developing An Application

The GuiTutorial Application

The application developed in the preceding chapter can be easily modified to demonstrate fault usage. Since you must define new types representing the fault details, a new message, and modify the sayHi operation details, the changes will also impact the binding definition. Consequently, it is easiest to delete the existing binding and service elements from the WSDL file and recreate these entries once the other modifications are complete.

Modifying the WSDL File

Start the Artix Designer and return to the GuiTutorial project.

1. Highlight the HelloWorldGuiTutorial icon under the Contracts icon and click on the WSDL tab. The WSDL file contents are displayed in the panel.
2. Select the Contract > Edit > Services menu entry or right click on the HelloWorldGuiTutorial icon and select Edit > Services from the popup menu.
3. In the Edit Services — Artix Designer window, highlight the HelloWorldService icon in the top panel and click on the Delete command button. Confirm your decision by clicking the Yes command button. Then click on the Apply and OK command buttons. View the WSDL file contents and confirm that the <service>...</service> section has been removed.
4. Select the Contract > Edit > Bindings menu entry or right click on the HelloWorldGuiTutorial icon under the Contracts icon. and select Edit > Bindings from the popup menu.
5. Select the Edit Bindings window, highlight the GuiTutorialPT_SOAPBinding icon in the top panel and click on the Delete command button. Confirm you decision by clicking the Yes command button. The click on the Apply and OK command buttons. View the WSDL file contents and confirm that the <binding>...</binding> section has been removed.

If you highlight the HelloWorldGuiTutorial icon under either the Client or Server icons and view the WSDL file contents, you will observe the edited content. These icons actually represent links to the WSDL file in the Contracts directory, so edits are applied to the file imported into the Client.wsdl and Server.wsdl files.

You now create the data types that represent the exception details.

1. Highlight the HelloWorldGuiTutorial icon and select the Contract > New > Type menu entry, or right click on the HelloWorldGuiTutorial icon and select New > Type from the popup menu.
2. In the New Type — Artix Designer — Select WSDL window, select the Add to existing WSDL "HelloWorldGuiTutorial" radio button and click the Next command button.
3. In the New Type — Artix Designer — Define Type Properties window, enter FaultDetails into the Name text box and select the complexType radio button. Click the Next command button.
4. In the New Type — Artix Designer — Define Type Attributes window, select sequence from the Group Type drop down list. From the Type drop down list, select the xsd:string entry and enter FaultMsg into the Name text box. Click the Add command button, which transfers this element to the Element List panel.
5. To add the second member of the FaultDetails sequence, select xsd:int from the Type drop down list and then enter FaultID into the Name text box. Click the Add command button. Your sequence now includes two members. Click the Next and Finish command buttons to complete the type definition entry.
6. You now want to define an element type that wraps your complex types. This process is identical to defining the complex type except that you must select the element radio button in the New Type — Artix Designer — Define Type Properties window.
7. In the New Type — Artix Designer — Define Type Attributes window, you are only presented with a Type drop down list. Since an element type is simply a wrapper around another type, there are no additional options.
8. Create one element type:
   - DoIKnowYou of type tns:FaultDetails.

You must now define the fault message. There is nothing that links this message to an operation fault except how you use the message when defining an operation.

1. Highlight the HelloWorldGuiTutorial icon and select the Contract > New > Message menu entry, or right click on the HelloWorldGuiTutorial icon and select New > Message from the popup menu.
2. In the New Message — Artix Designer — Select WSDL window, select the Add to existing WSDL "HelloWorldGuiTutorial" radio button and click the Next command button.
3. In the New Message — Artix Designer — Define Message Properties window, enter UnknownUser into the Name text box. Click the Next command button.
4. In the New Message — Artix Designer — Define Parts window, enter theFault into the Name text box and select tns:DoIKnowYou from the Type drop down list. Now click the Add command button; your part is added to the Part List control. Finally, click the Next command button.
5. In the New Message — Artix Designer — View Summary window, you can review the content that will be added to the WSDL file. Click the Finish command button.

Finally you need to edit the portType definition.

1. Highlight the HelloWorldGuiTutorial icon and select the Contract > Edit > Port Types menu entry or right click and select Edit > Port Types from the popup menu.
2. In the Edit Port Types — Artix Designer window, highlight the sayHi operation in the top panel and then click the Edit command button below the Operation Messages grouping control.
3. In the Edit Operation Messages — Artix Designer window select fault from the Type drop down list. From the Message drop down list select the tns:UnknownUser entry. Click on the Add command button, which adds the fault message to the Operation Messages listing. Finally, click on the Apply, OK, and OK command buttons.

Review the contents of the WSDL file and confirm that the sayHi operation now includes a fault element.

Now you need to recreate the SOAP binding and service and port definitions.

1. Highlight the HelloWorldGuiTutorial icon and select the Contract > New > Binding menu entry, or right click on the HelloWorldGuiTutorial icon and select New > Binding from the popup menu.
2. In the New Binding — Artix Designer — Select WSDL window, select the Add to existing WSDL "HelloWorldGuiTutorial" radio button and click the Next command button.
3. In the New Binding — Artix Designer — Select Binding Type window, select the SOAP radio button. Click the Next command button.
4. In the New Binding — Artix Designer — Select Port Type window, select the GuiTutorialPT entry from the Port Type drop down list. Note that a suggested Binding Name is already entered into the Name text box. You can change this entry; the only requirement is that each binding in the WSDL file, if you create multiple bindings, have a unique Binding Name.
5. In the SOAP Setting control group there are two drop down list controls. From the Style list, select document, and from the Use list, select literal. Click the Next command button.
6. In the New Binding — Artix Designer — Edit Binding window, highlight the sayHi icon representing your operation and review the binding details. Click the Next command button.
7. In the New Binding — Artix Designer — View WSDL Contract window, you can review the new content that will be added to the WSDL file.
8. Finally, click the Finish command button. Highlight the HelloWorldGuiTutorial icon, click on the WSDL tab, and review the contents of the WSDL file.
9. Once again, highlight the HelloWorldGuiTutorial icon and select the Contract > New > Service menu entry, or right click on the HelloWorldGuiTutorial icon and select New > Service from the popup menu.
10. In the New Service — Artix Designer — Select WSDL window, select the Add to existing WSDL "HelloWorldGuiTutorial" radio button and click the Next command button.

11. In the New Service — Artix Designer — Define Service window, enter HelloWorldService into the Name text box. Click the Next command button.
12. In the New Service — Artix Designer — Define Port window, enter HelloWorldPort into the Name text box and select GuiTutorialPT_SOAPBinding from the Binding drop down list. Click the Next command button.
13. In the New Service — Artix Designer — Define Extensor Properties window, select SOAP from the Transport Type drop down list and enter http://localhost:9000 as the value for the location attribute. This is the only required entry, and you may specify any port number you choose.
14. Click the Next command button. In the New Service — Artix Designer — Port Summary window, you can review the new content that will be added to the WSDL file. Finally, click the Finish command button.

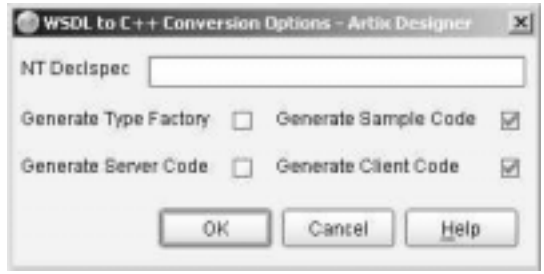
Generating the Application Code

You will use the Artix Designer to generate starting point code for the application.

1. Highlight the Development icon under the Client icon. The System Development Options window appears.
2. For this example, select C++ from the Development Environment drop down list. The suggest entry in the Code Location text box indicates that the code will be generated into a directory under the GuiTutorial\Client directory.

   Since you are generating code for a client application, you deselect the Generate Implementation Code check box.
3. The Select Service and Select Port drop down lists are used to select the service and port against which code will be generated. Since there is only one service and port in your WSDL file, the lists contain only one entry. These drop down lists correspond to the -e and -t command line options for the wsdltocpp utility.
4. Select the appropriate radio button within the Generate Makefile group.

5. Click the Advanced Options command button. The WSDL to C++ Conversions Options — Artix Designer window opens.



   Since you are building the client application, deselect the Generate Server Code checkbox. The Generate Sample Code checkbox is equivalent to the -sample option for the wsdltocpp utility and the Generate Client Code checkbox is equivalent to the -client option for the wsdltocpp utility.

6. Click the OK command button, then click the OK command button on the System Development Options window. A message box confirms code generation success.

7. Now highlight the development icon under the Server icon. The System Development Options window appears.
8. For this example, select C++ from the Development Environment drop down list. The suggest entry in the Code Location text box indicates that the code will be generated into a directory under the GuiTutorial\Server directory.

   As you are generating code for a server application, you select the Generate Implementation Code check box. The suggested C++ namespace is derived from the project name; you may change this to a more meaningful value. Since you are building the client and server

applications separately, you do not need to use the same C++ namespace value in both applications.

9. The Select Service and Select Port drop down lists are used to select the service and port against which code will be generated. Since there is only one service and port in your WSDL file, the lists contain only one entry. These drop down lists correspond to the -e and -t command line options for the wsdltocpp utility.
10. Select the appropriate radio button within the Generate Makefile group.
11. Click the Advanced Options command button. The WSDL to C++ Conversions Options — Artix Designer window opens.

Since you are building the server application, make certain to deselect the Generate Client Code checkbox. The Generate Sample Code checkbox is equivalent to the -sample option for the wsdltocpp utility and the Generate Server Code checkbox is equivalent to the -server option for the wsdltocpp utility.

12. Click the OK command button, then click the OK command button on the System Development Options window. A message box confirms code generation success.

Note: Although the types FaultDetails and UnknownUser are defined in the HelloWorldGuiTutorial.wsdl file, Artix v1.3 places the code corresponding to the element UnknownUser into the files Client_wsdlTypes.h/.cxx. This class definition should be in the files HelloWorldGuiTutorial_wsdlTypes.h/.cxx.

Completing the Code

In the implementation class you need to complete the sayHi method. You will modify the previous coding so that the UnknownUserException is thrown unless the value of InPart is Artix User.

```
if (InPart.get_value() != "Artix User")
{
  FaultDetails faultData;
  faultData.setFaultMsg("User unknown to me");
  faultData.setFaultID(200);

  UnknownUserException ex;
  ex.settheFault(faultData);

  throw ex;
}
OutPart.set_value("Hello " + InPart.get_value());
```

In the client application SampleClient.cxx file, remove the comment delimiters and replace with the following code.

```
GuiTutorial::InParameter    InPart_0;
GuiTutorial::OutParameter   OutPart_1;

// Set user name to command line parameter
InPart_0.set_value("Artix User");
if (argc > 1)
{
  InPart_0.set_value(argv[1]);
}
// Alternative code to set user name
/*
argc > 1 ? InPart_0.set_value(argv[1]) : \
             InPart_0.set_value("Artix User");
*/
client.sayHi ( InPart_0,  OutPart_1);
cout << "sayHi returned: " + OutPart_1.get_value() << endl;
```

Also add a new catch{} block before the existing catch{} block.

```
catch(UnknownUserException& ex)
{
  FaultDetails& fd = ex.gettheFault();
  cout << "Error Message: " << fd.getFaultMsg() << endl;
  cout << "Error ID: " << fd.getFaultID() << endl;
  return -1;
}
```

Building the Application

Now that you have completed coding, you can build the application.

1. Open a command window to the <installationDirectory>\artix\1.3\bin directory and run the artix_env.bat file.
2. Move to the <installationDirectory>\artix\1.3\demos\GuiTutorial\Client\src\cpp directory by issuing the command:

```
cd ..\demos\GuiTutorial\Client\src\cpp
```

3. Build the client application with the command:

```
nmake all
```

This creates the client executable file client.exe.

To build the server application:

1. Move to the <installationDirectory>\artix\1.3\demos\GuiTutorial\Server\src\cpp directory by issuing the command:

```
cd ..\..\..\Server\src\cpp
```

2. Build the server application with the command:

```
nmake all
```

This creates the server executable file server.exe.

Running the Application

To start the server application:

1. Start the server application with the command:

```
start server
```

A new command window opens and the server application starts.

To run the client application:

1. Move to the <installationDirectory>\artix\1.3\demos\GuiTutorial\Client\src\cpp directory by issuing the command:

```
cd ..\..\..\Client\src\cpp
```

2. Run the client application with the command,

```
client <someName>
```

   which submits <someName> to the Web service's sayHi method.

   Or with the command:

```
client
```

   which submits Artix User to the Web service's sayHi method.

The client application invokes on the Web service and displays either the anticipated greeting or the details of the exception.

Stop the Server Process

Stop the server process by issuing Ctrl-C in its command window.

CHAPTER 6

Mortgage Calculator

This chapter is your "final exam." With minimal guidance, you will use the Artix™ Designer to recreate a WSDL file that describes a mortgage calculator Web service. Once you write the WSDL file, you will generate the starting point code and build the service.

In This Chapter

This chapter discusses the following topics:

The Mortgage Calculator Web Service

The Service

Assume you work in the information services department of a banking institution. The bank has decided to deploy a mortgage calculator Web service so that mortgage brokers and potential customers can determine the amount of a monthly mortgage payment. You have been given the assignment of creating this Web service.

The service is relatively simple. It receives a request that includes the amount of the anticipated loan, the annual interest rate percentage, and the term of the loan in years. The service returns the same information and the monthly payment value.

In a real work situation, you would be given the responsibility of defining the data types, writing the WSDL file, and implementing the service. If you want, you can approach this tutorial in the same way. Here is all of the information you need to know:

- Input values:

  Dollar amount of loan, for example, $100,000.00.

  Annual interest percentage rate, for example, 8.75.

  Term of loan in years, for example, 25.

- Output values:

  Dollar amount of loan.

  Annual interest percentage rate.

  Term of loan in years.

  Monthly payment.

- Faults:

  The bank does not issue mortgages that exceed $2000.00 per month. If the monthly payment is greater than this amount, the service should raise the fault PaymentExceedsLimit, which includes the monthly payment value as a member.

- Monthly payment formula:

  Numerator = (($ amount of loan) * (monthly interest rate))

  Denominator = (1 - ((1 +(monthly interest)) ** (-(Term * 12))))

Monthly Payment = Numerator \ Denominator

Where:

Monthly interest rate is a decimal value equal to the annual interest percentage rate divided by 1200.

* represents multiplication.

** represents exponentiation.

/ represents division.

Alternatively, you can use the WSDL file described in the next section and the wsdltocpp utility (discussed in Chapter 3, “Coding the Web Service”) to generate the starting point code.

Finally, you can employ the WSDL file described in the next section as a guide and use the Artix Designer to recreate the WSDL file and generate the starting point code (as discussed in Chapter 4, “Using the Artix™ Designer”).

Which approach you take is your decision.

The WSDL File

The Web Service Description

The following WSDL file describes the Mortgage Calculator Web service. Although this Web service can be adequately described using simple data types and messages with multiple parts, this WSDL file consolidates all of the required input and output values into complex types and each method contains only one part.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
   name="MortgageCalculator"
   targetNamespace="http://www.bank.com/mortgagecalculator"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:http-conf=
    "http://schemas.iona.com/transports/http/configuration"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:tns="http://www.bank.com/mortgagecalculator"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <types>
  <schema
    targetNamespace="http://www.bank.com/mortgagecalculator"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <element name="CalculateFault" type="tns:MonthlyPayment"/>
   <element name="Request" type="tns:LoanRequest"/>
   <element name="Response" type="tns:BankAnswer"/>
   <simpleType name="LoanAmount">
    <restriction base="xsd:double"/>
   </simpleType>
   <simpleType name="Term">
    <restriction base="xsd:int"/>
   </simpleType>
   <simpleType name="AnnualInterestRate">
    <restriction base="xsd:double"/>
   </simpleType>
   <simpleType name="MonthlyPayment">
    <restriction base="xsd:double"/>
   </simpleType>
```

```
  <complexType name="BankAnswer">
   <sequence>
    <element name="Principal" type="tns:LoanAmount"/>
    <element name="Years" type="tns:Term"/>
    <element name="PercentageRate"
             type="tns:AnnualInterestRate"/>
    <element name="Payment" type="tns:MonthlyPayment"/>
   </sequence>
  </complexType>
  <complexType name="LoanRequest">
   <sequence>
    <element name="Principal" type="tns:LoanAmount"/>
    <element name="Years" type="tns:Term"/>
    <element name="PercentageRate"
             type="tns:AnnualInterestRate"/>
   </sequence>
  </complexType>
 </schema>
</types>

<message name="PaymentExceedsLimit">
 <part element="tns:CalculateFault"
       name="CalculatedPayment"/>
</message>
<message name="CustomerRequest">
 <part element="tns:Request" name="Input"/>
</message>
<message name="BankResponse">
 <part element="tns:Response" name="Output"/>
</message>

<portType name="Calculator">
 <operation name="CalculateMonthlyPayment">
  <input message="tns:CustomerRequest"
         name="CalculateMonthlyPaymentRequest"/>
  <output message="tns:BankResponse"
          name="CalculateMonthlyPaymentResponse"/>
  <fault message="tns:PaymentExceedsLimit"
         name="CalculateMonthlyPaymentFault"/>
 </operation>
</portType>
```

```
 <binding name="Calculator_SOAPBinding" type="tns:Calculator">
  <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="CalculateMonthlyPayment">
   <soap:operation soapAction="" style="document"/>
   <input name="CalculateMonthlyPaymentRequest">
    <soap:body use="literal"/>
   </input>
   <output name="CalculateMonthlyPaymentResponse">
    <soap:body use="literal"/>
   </output>
   <fault name="CalculateMonthlyPaymentFault">
    <soap:fault name="CalculateMonthlyPaymentFault"
                use="literal"/>
   </fault>
  </operation>
 </binding>

 <service name="MortgageService">
  <port binding="tns:Calculator_SOAPBinding"
        name="MortgagePort">
   <soap:address location="http://localhost:9000"/>
   <http-conf:client/>
   <http-conf:server/>
  </port>
 </service>

</definitions>
```

If you want to use this file directly, you will need to copy the content into a text file. Since the content spans three pages of this document, you will need to copy and paste from each page separately, recombining the extracts in the proper order in your text file. You can then proceed as described in Chapter 3.

If you want to use this file as a guide to writing your own WSDL file with the Artix Designer, you should start the designer, create a new project, and then create the WSDL file. Start with the simple type definitions, then the complex types followed by the element types. Next, define the messages, port type, binding, and service. You can then create the client and server applications, and generate starting point code, as described in Chapter 4.

The Application Code

Application Files

The wsdltocpp utility and the Artix Designer generate all of the files you need to write this application.

- SampleClient.cxx is your client application. You will need to add the code that makes the invocation and prints out the results. You will also want to allow the user to input loan amount, annual interest rate, and term as command line arguments to the client application. Assume that the input will not include the dollar or percent signs.
- <WSDLfileName>_wsdlTypes.h is the file that contains your type definitions. You must review the content of this file to understand the API for each of these types.

Note: If you use the provided WSDL file and the wsdltocpp utility, type definitions are generated into the file <WSDLfileName>_wsdlTypes.h. If you use the Artix Designer and the composite WSDL approach to write your own WSDL file, the definition of the exception class is inthe files Client_wsdlTypes.h and Server_wsdlTypes.h.

- CalculatorImpl.cxx is your implementation object. You will need to add the code that calculates the monthly payment from the input values. Remember that the processing logic in the CalculateMonthlyPayment method needs to throw the PaymentExceedsLimitException if the monthly payment is greater than $2000.00.

The Client Application Code

The following code fragment is a workable solution to this coding assignment.

```
#include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_cal/iostream.h>

#include "CalculatorClient.h"

IT_USING_NAMESPACE_STD
using namespace MortgageCalculator;
using namespace IT_Bus;
```

```
int
main(
    int argc,
    char* argv[]
)
{
 cout << "Calculator Client" << endl;
 try
  {
    IT_Bus::init(argc, argv);
    CalculatorClient client;
    MortgageCalculator::LoanRequest   Input_0;
    Input_0.getPrincipal().set_value
      (strtod(argv[1], (char**)NULL));
    Input_0.getPercentageRate().set_value
      (strtod(argv[2], (char**)NULL));
    Input_0.getYears().set_value(atoi(argv[3]));

    MortgageCalculator::BankAnswer   Output_1;
    client.CalculateMonthlyPayment ( Input_0,  Output_1);

    cout << "Principal: $" << Output_1.getPrincipal().get_value()
           << endl;
    cout << "Term: " << Output_1.getYears().get_value()
            << " years" << endl;
    cout << "Rate: " << Output_1.getPercentageRate().get_value()
           << "%" << endl;
    cout << "Payment: $" << Output_1.getPayment().get_value()
           << endl;
  }
  catch (PaymentExceedsLimitException& ex)
  {
    cout << endl << "PaymentExceedsLimitException Raised"
           << endl;
    cout << "\tMonthly payment too large: $"
           << ex.getCalculatedPayment().get_value() << endl;
  }
  catch(IT_Bus::Exception& e)
  {
    cout << endl << "Error : Unexpected error occured!"
           << endl << e.Message() << endl;
    return -1;
  }
  return 0;
}
```

The CalculatorImpl Code

The following code fragment is a workable solution to this coding assignment (only the code for the CalculatorImpl class is shown).

```
#include "CalculatorImpl.h"
#include <it_cal/cal.h>
// Add include for math.h
#include <math.h>

using namespace MortgageCalculator;

CalculatorImpl::CalculatorImpl(IT_Bus::Bus_ptr bus)
  : MortgageCalculator::CalculatorServer(bus) {}

CalculatorImpl::~CalculatorImpl() {}

void
CalculatorImpl::CalculateMonthlyPayment(
    const MortgageCalculator::LoanRequest & Input,
    MortgageCalculator::BankAnswer & Output
) IT_THROW_DECL((IT_Bus::Exception))
{
  IT_Bus::Double principal = Input.getPrincipal().get_value();
  IT_Bus::Double rate = Input.getPercentageRate().get_value();
  IT_Bus::Int term = Input.getYears().get_value();

  // Convert yearly interest rate to monthly interest rate
  rate/=1200;

  // Multiply principle by rate
  // The value in numerator will become the payment
  IT_Bus::Double numerator = principal*rate;

  // Convert years to months
  term*=12;

  // Calculate value of denominator
  IT_Bus::Double denominator = (1 - ((pow((1+rate), (-term)))));

  // Calculate monthly payment
  numerator/=denominator;
  // Numerator now holds the monthly payment
```

```
  // Declare a variable to return the payment
  MonthlyPayment payment;
  // Set payment into return object
  payment.set_value(numerator);

  if (numerator >= 2000)
  {
    // Create and throw exception
    PaymentExceedsLimitException ex;
    ex.setCalculatedPayment(payment);
    throw ex;
  }

  // Initialize return
  // Transfer values from input
  Output.setPrincipal(Input.getPrincipal());
  Output.setYears(Input.getYears());
  Output.setPercentageRate(Input.getPercentageRate());
  // Set monthly payment
  Output.setPayment(payment);
}
```

Note the include statement for the math.h file; this header file is not included by the Artix code generation process.

Building the Application

Compile the applications as described in either Chapter 3 or Chapter 4.

Running the Application

Open a command window and set the environment by running the artix_env.bat file.

Move to the directory containing your server application. Start the server with the command:

```
start server
```

Move to the directory containing your client application. Run the client, providing loan amount, annual interest rate, and term (in years) as command line arguments; do not enter $ or % signs.

```
client <loan amount> <annual interest rate> <term>
```

Run the client with several combinations of input and confirm that the exception is properly thrown and handled.