IONA®

Artix™

Tutorial

Version 2.0.2, April 2004

Making Software Work Together™

Updated: 30-Apr-2004

M 3 1 9 5

Contents

Preface v

What is Covered in this Book v
Who Should Read this Book v
Organization of this Book v
Related Documentation v
Online Help vi
Suggested Path for Further Reading vi
Additional Resources for Help vii
Typographical Conventions vii
Keying Conventions viii

Chapter 1 Introduction 1

Introduction to the Tutorial 2

Chapter 2 The WSDL File 3

What Are Web Services? 4
What is WSDL? 6
A Complete WSDL File 14

Chapter 3 Coding the Web Service 17

The wsdltocpp Utility 18
The wsdltojava Utility 21
Generating Code 24
Generating the Client Application Code 26
Generating the Server Application Code 29
Completing the Coding 32
Building the Client Application 37
Running the Application 38
The C++ Application 39
The Java Application 40
Interoperability Between the C++ and Java Applications 41

Chapter 4 Using the Artix™ Designer 43

The Artix Designer 44
The Artix Workspace 45
Writing the WSDL File 50
Create the WSDL File 51
Define the Types 54
Define the Messages 64
Define the portType 69
Define the Binding 73
Define the Service 77
Developing an Application 81
Generating Starting Point Code 85
Defining Deployment Profiles 86
Defining Deployment Bundles 87
Generating the C++ and Java Code 91
Completing the Code 94
The C++ Client Code 95
The C++ Server Code 97
The Java Client Code 99
The Java Server Code 100
Compiling the Applications 101
Running the Application 103

Chapter 5 Faults and Exceptions 105
Raising Exceptions 106
Handling Runtime Exceptions 107
Working with WSDL Faults 109
Developing An Application 113

Chapter 6 Mortgage Calculator 121
The Mortgage Calculator Web Service 122
The WSDL File 124
The Application Code 127
Building and Running the Applications 131

Preface

What is Covered in this Book

The *Artix Basic Tutorial* provides a basic understanding of the concepts and terms used in the IONA Artix Standard product.

Who Should Read this Book

This manual is aimed at first-time Artix users. It is assumed that the reader is familiar with C++ and/or Java 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 book 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*
- *Designing Artix Solutions*
- *Deploying and Managing Artix Solutions*
- *Artix Installation Guide*
- *Artix Tutorial*
- *Developing Artix Applications with C++*

- *Developing Artix Applications with Java*
- *Artix Security Guide*

The latest updates to the Artix documentation can be found at http://iona.com/support/docs/artix/2.0/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*

   The Getting Started book describes the basic concepts behind Artix. It also provides a detailed walk through for developing a Web Service.
3. *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.
4. *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.
5. *Developing Artix Applications with C++*
   Developing Artix Applications in Java

   The development guide discusses the technical aspects of programming applications using the Artix API.
6. *Deploying and Managing Artix Solutions*

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

7. *Artix Security Guide*

   An introduction to the security features in Artix.

Additional Resources for Help

The IONA Knowledge Base contains helpful articles, written by IONA experts, about Artix and other products. You can access the knowledge base at the following location: (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 2.0.

In This Chapter

This chapter discusses the following topics:

Introduction to the Tutorial page 2

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 and wsdltojava utilities, command line applications that generate starting point 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 code for your application.

Chapter 5, "Faults and Exceptions" discusses Web service faults and their representation as C++ and Java 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 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++ or Java 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 programmer's guide *Developing Artix Applications*.

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?	page 4
What is WSDL?	page 6
A Complete WSDL File	page 14

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 (HTTP). 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 (ART). 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.

Using Artix

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, if you are programming in C++ 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 Designer, a tool through which you can write and edit WSDL files, convert CORBA Interface Definition Language (IDL) files, data files, and COBOL copybooks into WSDL, and generate code.

This tutorial introduces concepts in all four of these categories. The product documentation covers 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, which collectively define a Web service:

- Import Section
- Logical section
- Physical section

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 description of the Web service that is independent of any programming language, marshalling schema or protocol. This section of the WSDL file describes the data types and operations offered by the Web service. It is composed of three subsections:

- Types subsection
- Message subsection
- PortType subsection

Types subsection

The types subsection includes the definitions for specific 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 misled 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>
```

Message subsection

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 misled 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>
```

PortType subsection

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 interoperability 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:

- Binding subsection
- Service subsection

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, can be 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 supports 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 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, through its nested port elements, with the binding TutorialPortType_SOAPBinding. Each service element is associated with only one 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 namespace that applies to elements defined 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 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 and wsdltojava utilities to generate C++ and Java code from a WSDL file.

In This Chapter

This chapter discusses the following topics:

The wsdltocpp Utility	page 18
The wsdltojava Utility	page 21
Generating Code	page 24
Completing the Coding	page 32
Running the Application	page 38

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.
-plugin
   Whether to generate bus plugin code that registers an instance
    of the implementation object with the bus. If specified, the
    utility generates a file containing implementation code for
    the plugin. If omitted, the utility generates the
    registration code as part of the server mainline code.
-all
   Whether to generate code for all port types described in the
    WSDL file. If omitted, the utility will generate code only for
    the interface associated with the service and port specified
    through the -e and -t options.
-?
-flags
   Usage information.
```

There are other options in addition to those described. These additional options are not, however, commonly used and will not be discussed in this tutorial. Refer to the Artix product documentation for a complete discussion of the command line options.

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. If the WSDL file contains multiple service/port definitions and you do not use the -e and/or -t options, code will be generated for the first service and port defined in the WSDL file.

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.

The wsdltojava Utility

Generating Java Code

The wsdltojava utility is a command line tool that you will use to generate Java code from a WSDL file.

wsdltojava Utility Command Line Options

You control the output of the wsdltojava 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:

```
wsdltojava [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.
-p package
   The package name for the generated code. If omitted, the
   utility creates a package name from the value of the
   targetNamespace attribute in the WSDL file. If you do not want
   a package, enter -p "".
-impl
   Whether to generate starting point code for the Java class
   into which you will code the Web service implementation.
```

```
-server
   Whether to only generate code for the server stub classes and
    a server mainline.
-client
   Whether to generate client proxy classes only.
-sample
   Whether to generate starting point code for the client
    mainline application.
-all
   Whether to generate code for all port types described in the
    WSDL file. If omitted, the utility will generate code only for
    the interface associated with the service and port specified
    through the -e and -t options.
-?
-flags
   Usage information.
```

There are other options in addition to those described. These additional options are not, however, commonly used and will not be discussed in this tutorial. Refer to the Artix product documentation for a complete discussion of the command line options.

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. If the WSDL file contains multiple service/port definitions and you do not use the `-e` and/or `-t` options, code will be generated for the first service and port defined in the WSDL file.

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 Package

Generated Java code should be included within a package. You use the -p option to provide the package name.

Generating the Implementation Class

The wsdltojava 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. When you supply the `-server` option, the wsdltojava utility generates both the server stubs and mainline code.

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

Generating Code

Installing Artix

You must first install the Artix 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, or higher. If necessary, Artix will install the JRE during the installation process. Using the JRE restricts your development options to C++, although you will be able to run the Java versions of the product demos. If you want to develop Artix applications in Java, you must have a full JDK 1.4.1 (or higher) installation.

If you already have the v1.4.1 JDK installed, you will be able to run Artix against your existing installation. If you are using an existing 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\2.0\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/or edit this file.

The file `<installationDirectory>\artix\2.0\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.

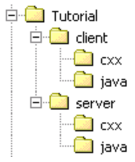
1. If you have an earlier version of the JRE or JDK installated on your computer, for example, v1.3.x, you may have already set a JAVA_HOME environment variable. You must remove this environment variable for Artix to work properly with the JRE included as part of the Artix installation. There is no need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

You must also be certain that your C++ compiler and libraries are available to your applications. With Windows, you may need to run the vcvars32.bat file to properly set your environment.[2]

Creating the Directory Structure

After installing Artix, a number of demo applications will be available in the `<installationDirectory>\artix\2.0\demos` directory. After you complete this tutorial you should review each of these demos.

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`, and within the `client` and `server` subdirectories, create the subdirectories `cxx` and `java`.



Copying the WSDL File

Return to the previous chapter and copy the contents of the HelloWorldTutorial.wsdl file into a text document.[3] Save the file as `HelloWorldTutorial.wsdl` into the `Tutorial\client` and the `Tutorial\server` directories. To view the contents with formatting, open the file in your Internet Explorer browser.

2. You may find it convenient to place a call to the vcvars32.bat file into the artix_env script file. In a text editor, open the artix_env script file and place the following entry at the beginning of the file:
   call "C:\Program Files\Microsoft Visual Studio\VC98\bin\vcvars32.bat";
   This syntax assumes that your Visual Studio installation is in the default location. The quotation marks and semicolon are required.
3. In the PDF version of this tutorial, the file content 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.

Generating the Client Application Code

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

Generating the C++ client application

1. Open a command window to the `<installationDirectory>\artix\2.0\bin` directory and run the `artix_env.bat` file.
2. Move to the `<installationDirectory>\artix\2.0\demos\Tutorial\client` directory by issuing the command:

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

3. Generate the C++ client application with the command:

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

The following files are generated within the `client\cxx` subdirectory:

- `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.
- `TutorialSampleClient.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 required if you have defined an anyType in your WSDL file. You do not need to review the contents of these files.
- `makefile`: A makefile that you can use to build the client application.

Generating the Java client application

1. Open a command window to the `<installationDirectory>\artix\2.0\bin` directory and run the `artix_env.bat` file.
2. Move to the `<installationDirectory>\artix\2.0\demos\Tutorial\client\java` directory by issuing the command:

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

3. Generate the Java client application with the command:

```
wsdltojava -p ArtixDemo -client -sample
   ../HelloWorldTutorial.wsdl
```

   The utility creates the subdirectory `ArtixDemo`, into which the following files are generated:

   - `Tutorial.java`: An interface that defines the method signatures for the Web service.
   - `TutorialTypeFactory.java`: Definition of a class that creates and manages anyTypes defined in your WSDL file.

- `TutorialDemo.java`: Starting point code for a client mainline application. For this simple example, the generated code in this file represents a fully functional application. With a more involved application, you would use this code as a template for writing a more complex client application.

Generating the Server Application Code

Generate the C++server application

To generate code for the server application:

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

```
cd ..\server
```

2. Generate the C++ server application with the command:

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

   The following files are generated into the `server\cxx` subdirectory:

   - `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.
   - `TutorialServerSample.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, perhaps by using the servant management classes. Refer to the product documentation (and/or the product demos in the servant_management subdirectory) for additional information on this topic.
   - `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, for example by overriding the activated method inherited from the implementation class' superclass. Refer to the product documentation for additional information on this topic.

- `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.

3. Move to the `server\java` directory by issuing the command:

```
cd ..\java
```

Generating the Java server application

To generate code for the server application:

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

```
cd ..\server
```

2. Generate the Java server application with the command:

```
wsdltojava -p ArtixDemo  -server -impl
   ../HelloWorldTutorial.wsdl
```

The utility creates the subdirectory `ArtixDemo`, into which the following files are generated:

- `Tutorial.java`: An interface that defines the method signatures for the Web service.
- `TutorialTypeFactory.java`: Definition of a class that creates and manages anyTypes defined in your WSDL file.

- `TutorialImpl.java`: 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.
- `TutorialServer.java`: Starting point code for a server mainline application. For this simple example, the generated code in this file represents a fully functional application. With a more involved application, you might extend the generated code.

Completing the Coding

Overview

The files describing the C++ or Java implementation classes contain 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.

The C++ Implementation Class

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 setvalue(const IT_Bus::String & value);
        const IT_Bus::String & getvalue() const;

      private:
        IT_Bus::String m_val;

    };
    . . .

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

        void setvalue(const IT_Bus::String & value);
        const IT_Bus::String & getvalue() const;

      private:
        IT_Bus::String m_val;

    };
    . . .
};
```

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

```
OutPart.setvalue("Hello " + InPart.getvalue());
```

The Java Implementation Class

In a text editor, open the file `TutorialImpl.java`. The code you add to the sayHi method body is:

```
return "Hello " + inPart;
```

Building the C++ 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\2.0\bin` directory and run the `artix_env.bat` file.
2. Move to the `<installationDirectory>\artix\2.0\demos\Tutorial\server\cxx` directory by issuing the command:

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

3. Build the server application with the command:

```
nmake all
```

This creates the server executable file server.exe.

Building the Java Server Application

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

1. Move to the `<installationDirectory>\artix\2.0\demos\Tutorial\server\java` directory by issuing the command:

```
cd ..\java
```

2. Build the server application with the command:

```
javac ArtixDemo/*.java
```

This creates the server file TutorialServer.class.

The C++ Client Application

For this demo, you only need to work with the `TutorialClientSample.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;
         ArtixDemo::OutParameter   OutPart;
         client.sayHi ( InPart,  OutPart);
         */
    }
    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;
ArtixDemo::OutParameter   OutPart;

InPart.setvalue("Artix User");

client.sayHi ( InPart,  OutPart);

cout << "sayHi returned: " + OutPart.getvalue() << endl;
```

The Java Client Application

For this example, the generated code is a complete running application. To run the demo, there is no need to modify the generated code.

Building the Client Application

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

The C++ Application

1. Open a command window to the `<installationDirectory>\artix\2.0\bin` directory and run the `artix_env.bat` file.
2. Move to the `<installationDirectory>\artix\2.0\demos\Tutorial\client\cxx` directory by issuing the command:

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

3. Build the client application with the command:

```
nmake all
```

   This creates the client executable file client.exe.

The Java Application

1. Move to the `<installationDirectory>\artix\2.0\demos\Tutorial\cxx\java` directory by issuing the command:

```
cd ..\java
```

2. Build the client application with the command:

```
javac ArtixDemo/*.java
```

   This creates the client file TutorialDemo.class.

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\2.0\bin` directory and run the `artix_env.bat` file.
2. Alter the CLASSPATH to include the current directory. Issue the command:

```
set CLASSPATH=.;%CLASSPATH%
```

The C++ Application

You can start the C++ server application and then run the C++ client application.

Start the Server Application

To start the server application:

1. Move to the `<installationDirectory>\artix\2.0\demos\Tutorial\server\cxx` directory by issuing the command:

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

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\2.0\demos\Tutorial\client\cxx` directory by issuing the command:

```
cd ..\..\client\cxx
```

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.

The Java Application

You can start the Java server application and then run the Java client application.

Start the Server Application

To start the server application:

1. Move to the `<installationDirectory>\artix\2.0\demos\Tutorial\server\java` directory.
2. Start the server application with the command:

```
start java ArtixDemo.TutorialServer
```

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\2.0\demos\Tutorial\client\java` directory by issuing the command:

```
cd ..\..\client\java
```

2. Run the client application with the command:

```
java ArtixDemo.TutorialDemo sayHi
```

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.

Interoperability Between the C++ and Java Applications

To demonstrate that the C++ and Java Web service applications and clients are interoperable, you can run the Java client application against the C++ server application, and vice versa.

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	page 44
The Artix Workspace	page 45
Writing the WSDL File	page 50
Developing an Application	page 81
Generating Starting Point Code	page 85
Completing the Code	page 94
Compiling the Applications	page 101
Running the Application	page 103

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.

- Select **Start** | **Programs** | **IONA Artix 2.0** | **IONA Artix 2.0** | **Designer**.
- Or, open a command window to the directory `<`*`installationDirectory`*`>\artix\2.0\bin` and run the batch file `start_designer.bat`.

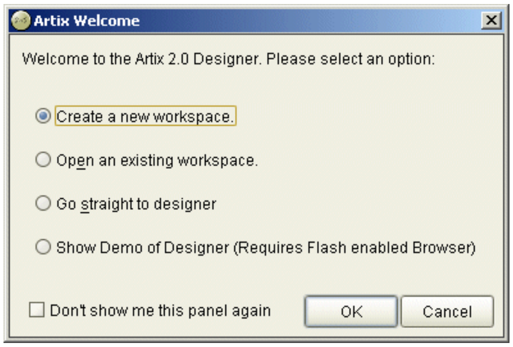
Selecting the menu entry simply runs the `start_designer.bat` file.

The Artix Workspace

Artix Workspaces

The Artix Designer maintains all of the files comprising an application within a larger entity called a workspace. The Artix workspace is analogous to a project grouping used by many Interactive Development Environment products.

When you start the designer, you are presented with the option of creating a new workspace, returning to an existing workspace, or starting the designer with no workspace open. You may also run a short video that illustrates some of the concepts of working with the designer.


Welcome to the Artix 2.0 Designer. Please select an option:

[✓ Create a new workspace.]

Open an existing workspace.

Go straight to designer

Show Demo of Designer (Requires Flash enabled Browser)

Figure 1: *Artix Welcome*

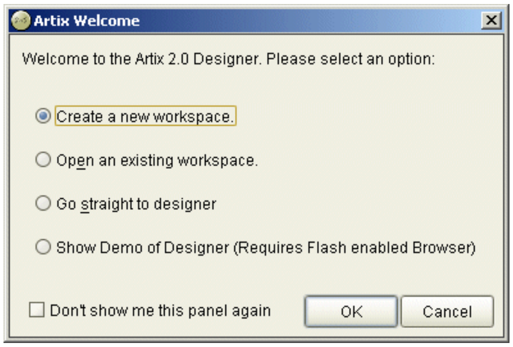
The name you assign to a workspace becomes the name of a directory under which the application files are stored. Artix will not let you assign two workspaces the same path and name. Consequently, when creating a new workspace you should not create the workspace directory manually; let the Artix Designer 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, and the designer 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 workspace and write an original WSDL file. You will then use the designer to generate starting point C++ and Java code.

Creating a New Workspace

1. After starting the Artix Designer, you are presented with the Welcome window. Select **Create a new workspace** and click the **OK** button.



Artix Welcome

Welcome to the Artix 2.0 Designer. Please select an option:

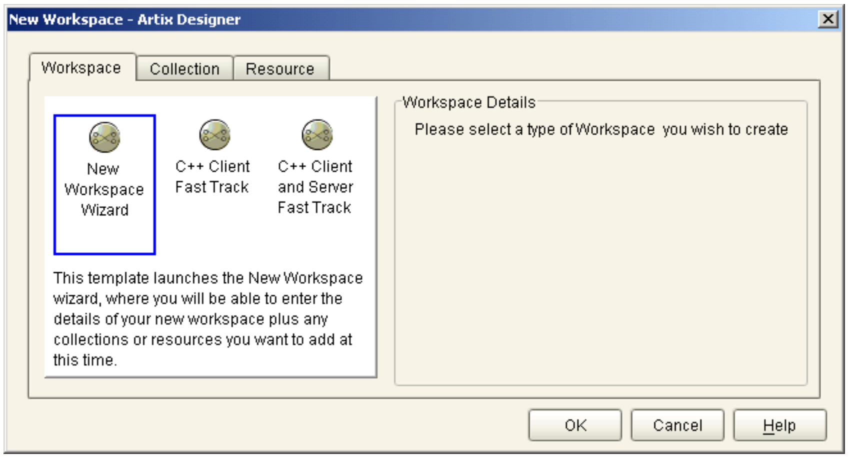
[✓ Create a new workspace.]
Open an existing workspace.
Go straight to designer
Show Demo of Designer (Requires Flash enabled Browser)

☐ Don't show me this panel again

Figure 2: *Artix Welcome*

If this window is not currently visible, select **File | New | Workspace**.

2. In the New Workspace window, select **New Workspace Wizard** and click the **OK** button.



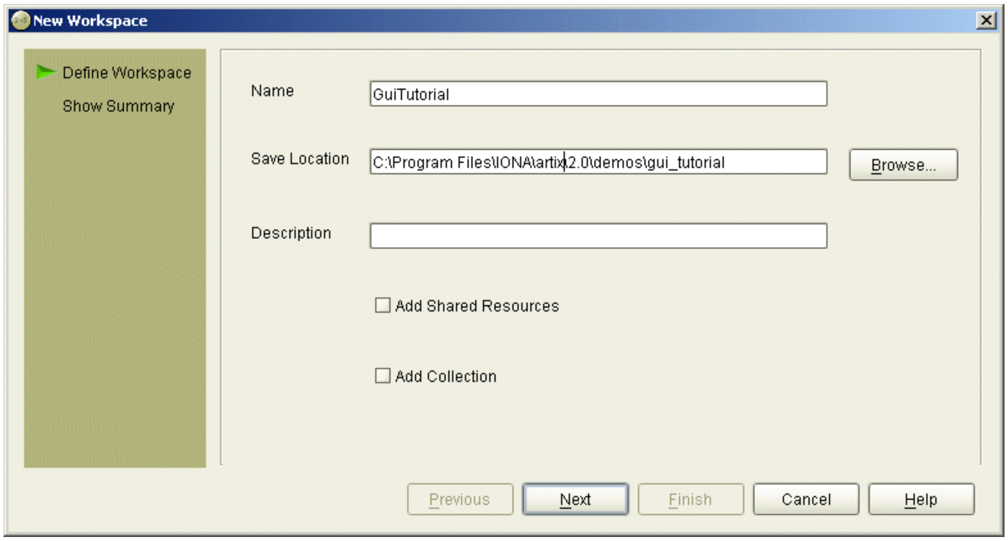
This template launches the New Workspace wizard, where you will be able to enter the details of your new workspace plus any collections or resources you want to add at this time.

Please select a type of Workspace you wish to create

Figure 3: *New Workspace*

3. In the Define Workspace panel, name your project **`GuiTutorial`**, and enter, or browse to, the directory that will contain your project.

   The Add Shared Resources checkbox is used 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 **Next** button.



Name: GuiTutorial

Save Location: C:\Program Files\IONA\artix\2.0\demos\gui_tutorial

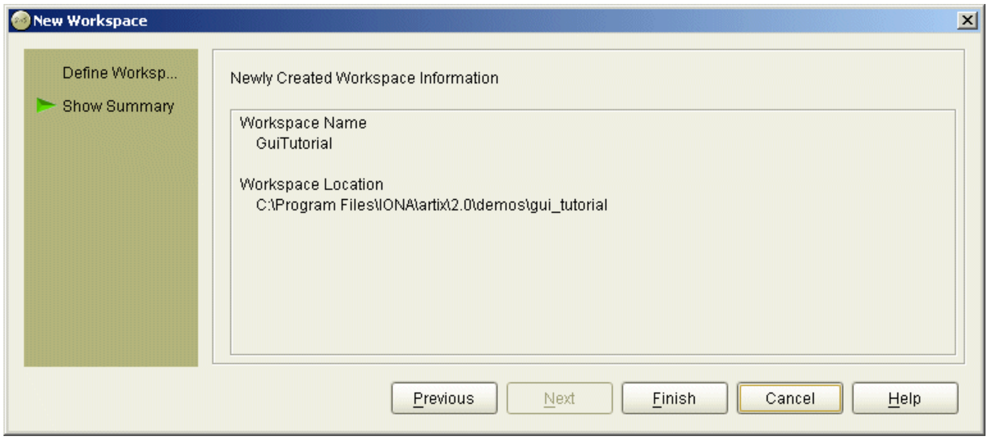
Description:

☐ Add Shared Resources

☐ Add Collection

Figure 4: *Naming the Workspace*

4. The Summary panel is shown. Click Finish to create the Workspace.



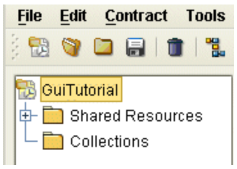
Newly Created Workspace Information

Workspace Name
GuiTutorial

Workspace Location
C:\Program Files\IONA\artix\2.0\demos\gui_tutorial

Figure 5: *Creating the Workspace*

The designer creates the workspace, and displays these in the left-hand pane as a list of shared resources (WSDL files) and collections (processes). You must save the project to actually create the project directories on your drive.



Writing the WSDL File

In this section

This section will lead you through the following tasks:

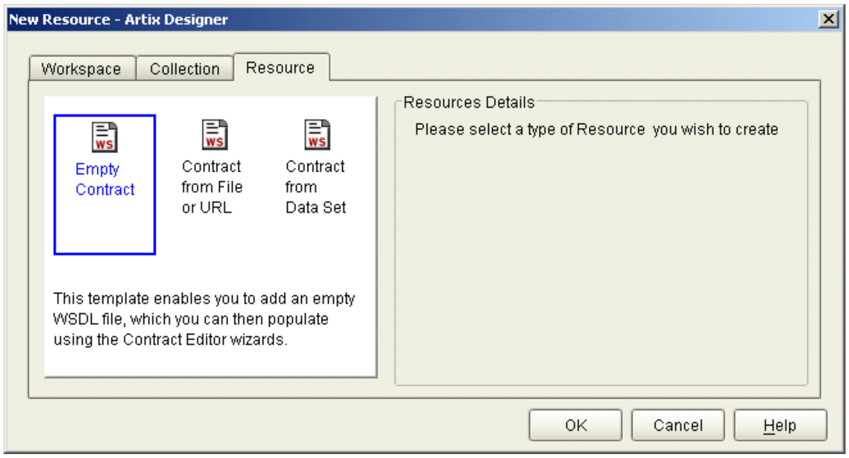
Create the WSDL File	page 51
Define the Types	page 54
Define the Messages	page 64
Define the portType	page 69
Define the Binding	page 73
Define the Service	page 77

Create the WSDL File

You will write the WSDL file as an entry under the Resources icon. You will then create collections corresponding to your client and server applications. Finally, you will associate your resource (WSDL file) with each of the collections and generate code.

1. Right click on the Shared Resources folder and select **New Resource** (or select **File | New Resource**).

   The New Resource wizard appears.



Please select a type of Resource you wish to create

This template enables you to add an empty WSDL file, which you can then populate using the Contract Editor wizards.

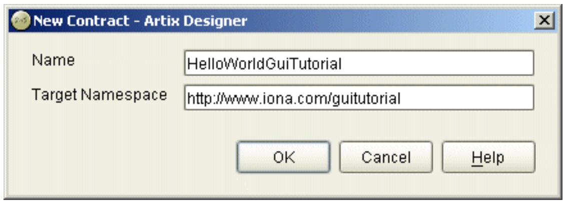
Figure 6: *New Resource*

2. Select Empty Contract and click OK.

   Note: The Contract from File or URL icon lets you to create a resource from an existing WSDL or IDL file, and the Contract from Data Set icon let you create a resource from a description of an existing data record format.

3. In the New Contract dialog, enter **`HelloWorldGuiTutorial`** into the Name text box, and enter **`http://www.iona.com/guitutorial`** into the TargetNamespace text box.

These entries become the values of the `name`, `targetNamespace`, and `xmlns:tns` attributes in the opening `<definitions>` element in the WSDL file.



Name: HelloWorldGuiTutorial

Target Namespace: http://www.iona.com/guitutorial

Figure 7: *Naming the Contract*

4. Click the OK button.

   The designer displays the various elements of the contract in the right-hand pane.

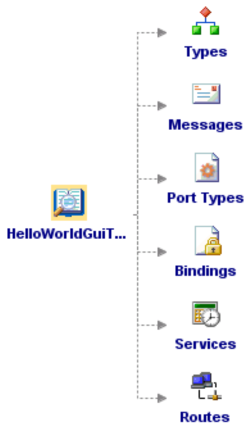
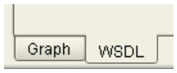
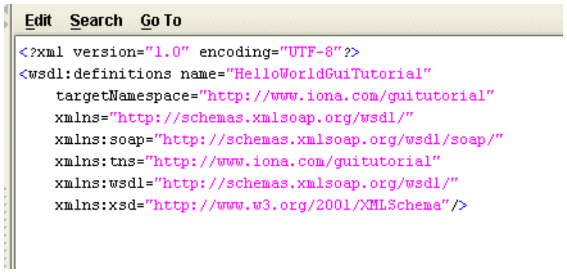


Figure 8: *New contract elements*

5. To view the contents of the WSDL file click on the WSDL tab.



The WSDL will look as follows:



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

Figure 9: *WSDL contents*

Currently the WSDL file only includes the opening `<definitions>` element.

Define the Types

The <types> section of the WSDL file contains your data type definitions. For this simple application you have several choices.

- You could choose not to define unique types for the application and use a basic type instead, 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.

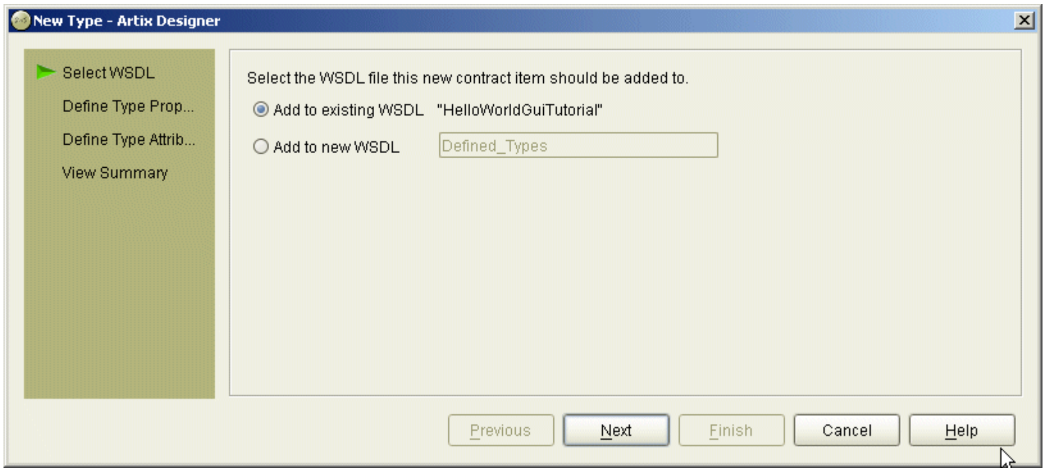
Defining the Simple Types

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. In the left-hand pane of the designer, highlight the HelloWorldGuiTutorial icon under Shared Resources and select **Contract | New | Type**.

   (Alternatively right-click on the HelloWorldGuiTutorial icon and select **New | Type** from the popup menu, or click on the Types icon in the right-hand pane and select NewType.)

2. In the Select WSDL screen, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.


New Type - Artix Designer

Select the WSDL file this new contract item should be added to.

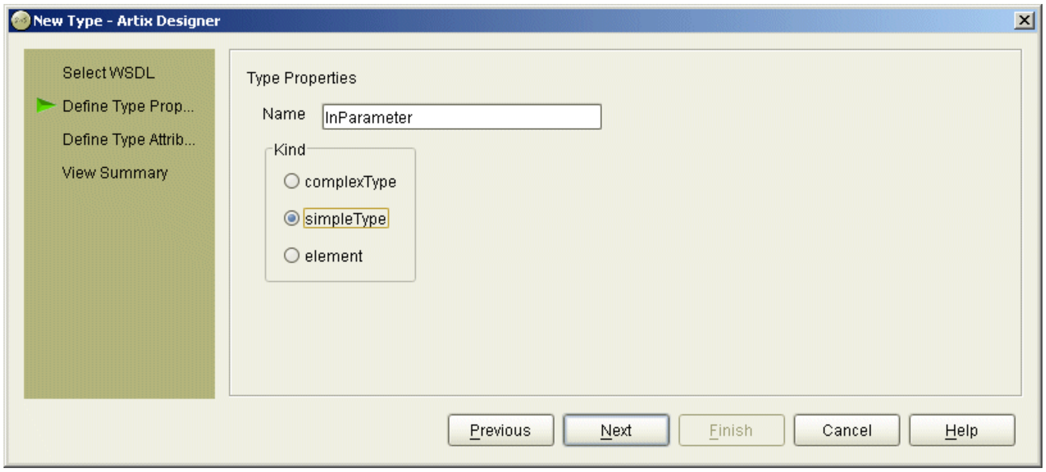
(●) Add to existing WSDL "HelloWorldGuiTutorial"

( ) Add to new WSDL Defined_Types

Figure 10: *Adding Type to existing WSDL*

Note: 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 selected content. This is the approach you would follow if you want to create WSDL file fragments for reuse.

3. In the Define Type Properties screen, enter **`InParameter`** into the Name text box and select the **simpleType** radio button. Click the Next button.



Type Properties

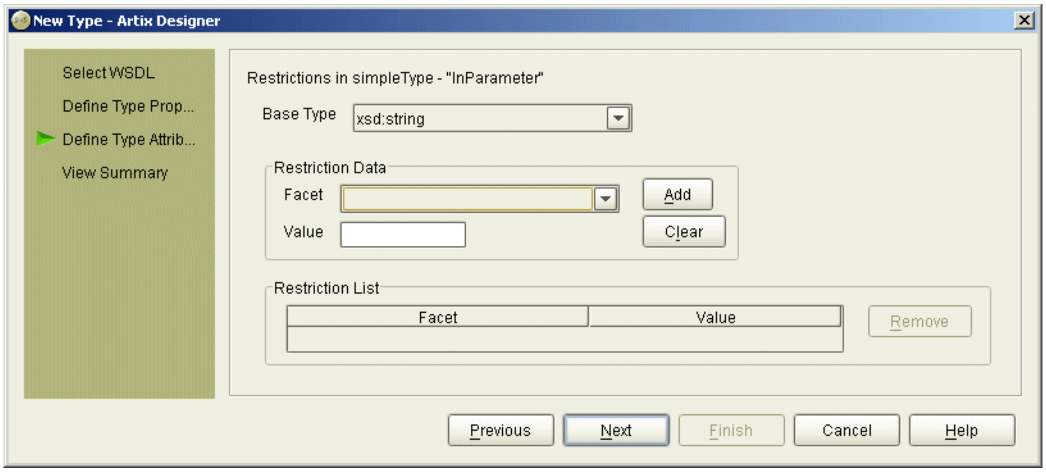
Name InParameter

Kind
complexType [✓ simpleType] element

Figure 11: *Specifying Type*

4. In the Define Type Attributes screen, 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 button.



Restrictions in simpleType - "InParameter"

Base Type: xsd:string

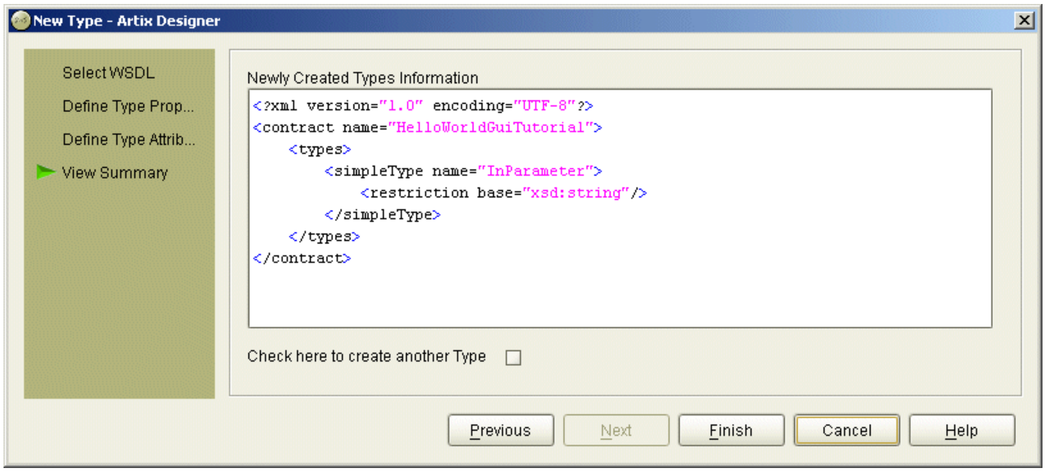
Facet:

Value:

Facet	Value

Figure 12: *Defining Type Attributes*

5. In the View Summary screen, you can review the content that will be added to the WSDL file. Click on the Finish button



```
<?xml version="1.0" encoding="UTF-8"?>
<contract name="HelloWorldGuiTutorial">
    <types>
        <simpleType name="InParameter">
            <restriction base="xsd:string"/>
        </simpleType>
    </types>
</contract>
```

Figure 13: *New Type Summary*

6. Repeat steps 1 to 5 to create 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.
7. Click on the Save icon or select **File | Save**.
8. If you double-click on the Types icon in the right-hand pane you will see the two new parameters:

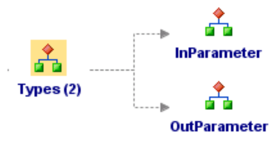
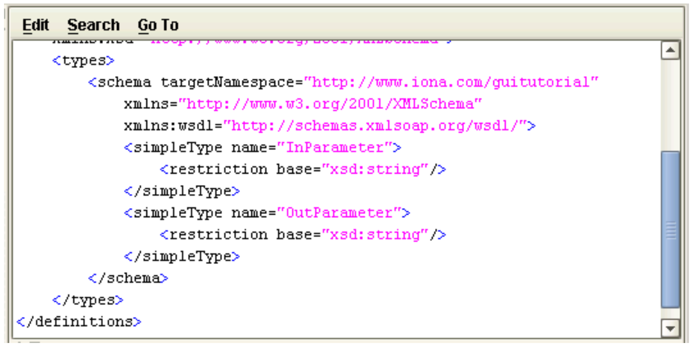


Figure 14: *New Types*

9. Select the WSDL tab to review the current contents of the WSDL file. Note that the <types> section has been added to the file



```
    <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>
        </schema>
    </types>
</definitions>
```

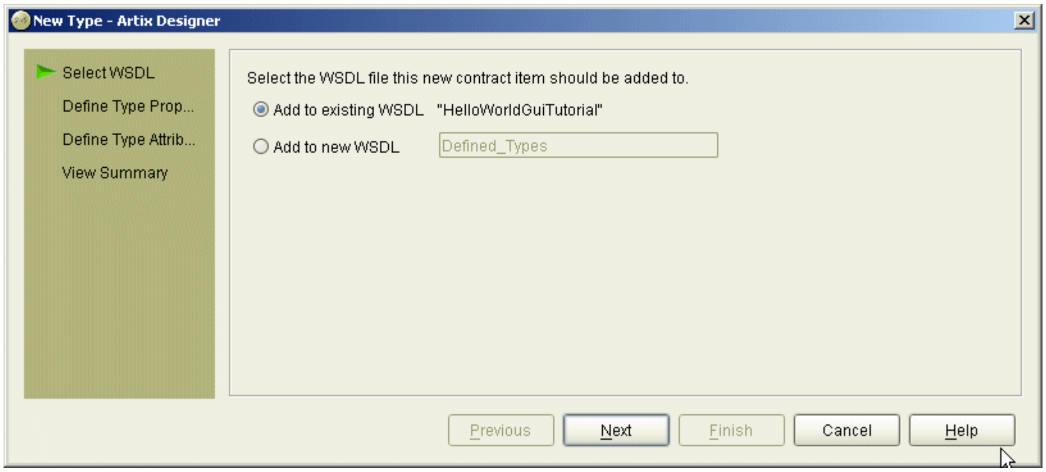
Figure 15: *WSDL with Types added*

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 Define Type Properties screen.

In the Define Type Attributes screen 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.

1. In the left-hand pane of the designer, highlight the HelloWorldGuiTutorial icon under Shared Resources and select **Contract | New | Type**.
2. In the Select WSDL screen, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.



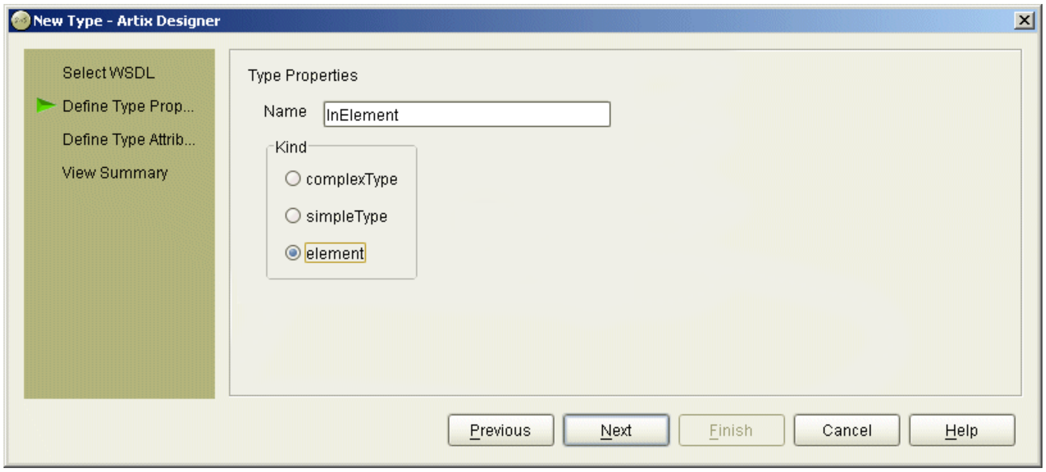
Select the WSDL file this new contract item should be added to.

[✓ Add to existing WSDL] "HelloWorldGuiTutorial"

Add to new WSDL: Defined_Types

Figure 16: *Adding type to existing WSDL*

3. In the Define Type Properties screen, enter **`InElement`** into the Name text box and select the **element** radio button. Click the Next button.



Type Properties

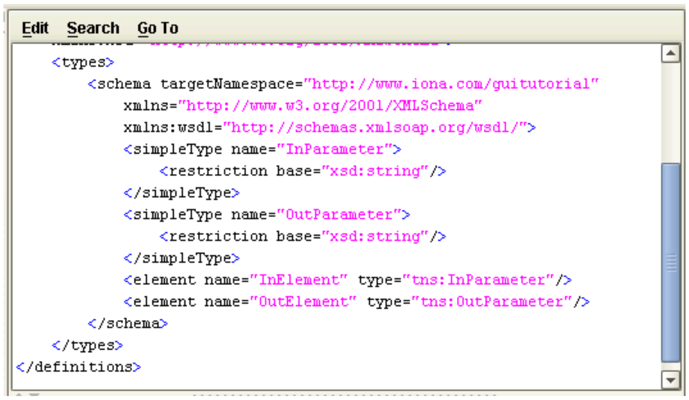
Name: InElement

Kind: ( ) complexType ( ) simpleType [✓ element]

Figure 17: *Define Type Properties*

4. In the Define Type Attributes screen select type **tns:InParameter**.
5. Click Next, then in the Summary screen click Finish.
6. Repeat steps 1 to 5 to create a second type named **`OutParameter`**. The type should be **tns:OutParameter**.
7. Click on the Save icon or select **File | Save**.

8. Highlight the HelloWorldGuiTutorial icon and review the contents of the WSDL file.



```
    <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>
```

Figure 18: *New Types in WSDL*

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.

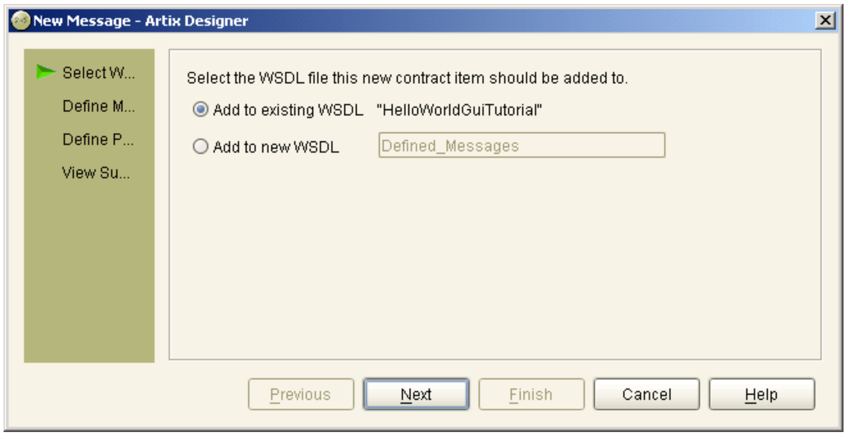
The following WSDL file fragment summarizes the content of the `HelloWorldGuiTutorial.wsdl` file.

```
<?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 **Contract | New | Message** (or right-click on the HelloWorldGuiTutorial icon and select **New | Message** from the popup menu).
2. In the Select WSDL panel, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.



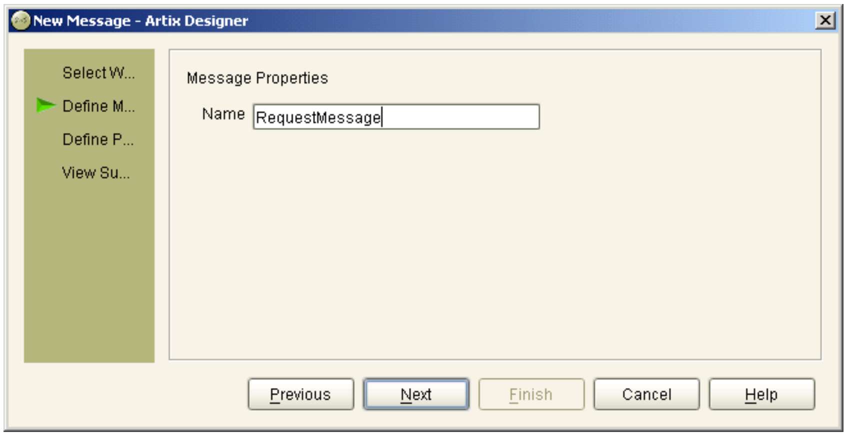
Select the WSDL file this new contract item should be added to.

(●) Add to existing WSDL "HelloWorldGuiTutorial"

( ) Add to new WSDL Defined_Messages

Figure 19: *Adding Message to existing WSDL*

3. In the Define Message Properties panel, enter **`RequestMessage`** into the Name text box. Click the Next button.



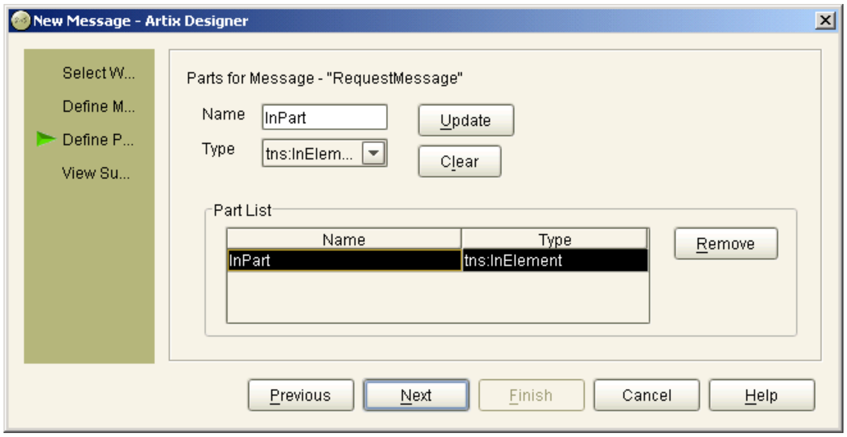
Message Properties

Name RequestMessage

Figure 20: *Naming Message*

4. In the Define Parts panel, enter **`InPart`** into the Name text box and select **tns:InElement** from the Type drop down list.

Note: Be careful — Do not select **tns:InParameter** from the Type drop down list



Parts for Message - "RequestMessage"

Name: InPart

Type: tns:InElem...

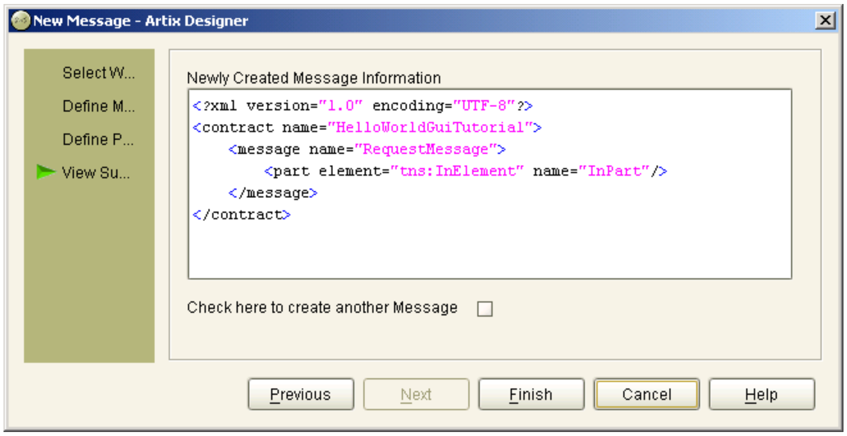
Part List

Name	Type
InPart	tns:InElement

Figure 21: *Adding parts*

5. Now click the **Add** 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 button.

6. In the View Summary panel, you can review the content that will be added to the WSDL file.



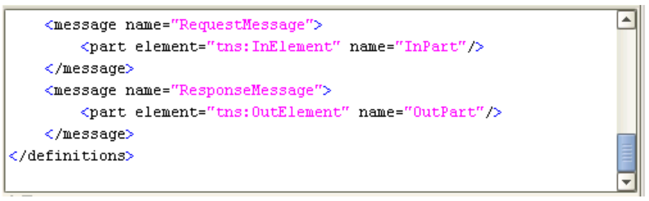
Newly Created Message Information

```
<?xml version="1.0" encoding="UTF-8"?>
<contract name="HelloWorldGuiTutorial">
    <message name="RequestMessage">
        <part element="tns:InElement" name="InPart"/>
    </message>
</contract>
```

Check here to create another Message ☐

Figure 22: *Message Summary*

7. Check the **Check here to create another message** checkbox and click the Next button.
8. Repeat steps 2 to 6 to create a second message – **`ResponseMessage`** – with a part named **`OutPart`** of type **tns:OutElement**. In the View Summary panel click the Finish button.
9. Click on the Save icon or select **File | Save**.
10. Highlight the HelloWorldGuiTutorial icon and click on the WSDL tab to review the contents of the WSDL file.



```
    <message name="RequestMessage">
        <part element="tns:InElement" name="InPart"/>
    </message>
    <message name="ResponseMessage">
        <part element="tns:OutElement" name="OutPart"/>
    </message>
</definitions>
```

Figure 23: *New messages definition in WSDL*

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

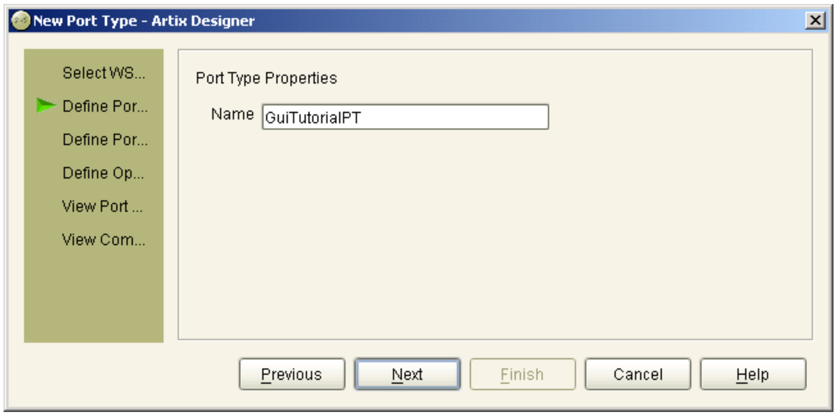
A portType contains operations, which are composed of one or more messages.

- 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 **Contract | New | Port Type** (or right-click on the HelloWorldGuiTutorial icon and select **New | Port Type** from the popup menu).
2. In the Select WSDL panel, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.

3. In the Define Port Type Properties screen, enter **`GuiTutorialPT`** into the Name text box. Click the Next button.

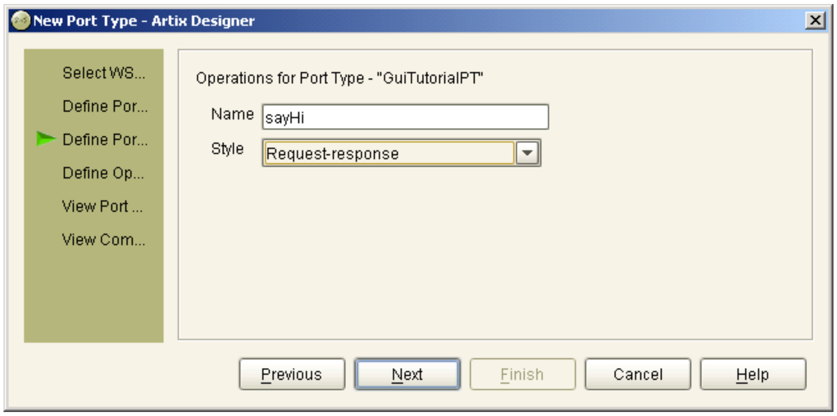


Port Type Properties

Name GuiTutorialPT

Figure 24: *Port Type Properties*

4. In the Define Port Type Operations screen, enter **`sayHi`** into the Name text box. Select **Request-response** from the Style drop down list and click the Next button.



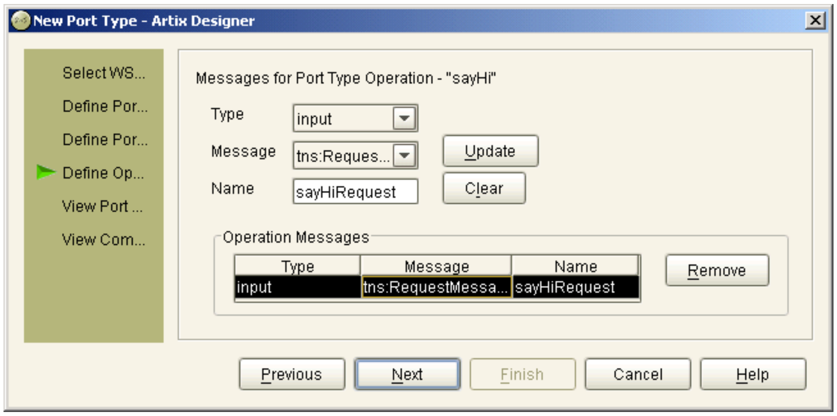
Operations for Port Type - "GuiTutorialPT"

Name sayHi

Style Request-response

Figure 25: *Port Type Operations*

5. In the Define Operation Messages screen, 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 button, which transfers the input message to the Operation Messages control.



Messages for Port Type Operation - "sayHi"

Type: input
Message: tns:Reques...
Name: sayHiRequest

Operation Messages

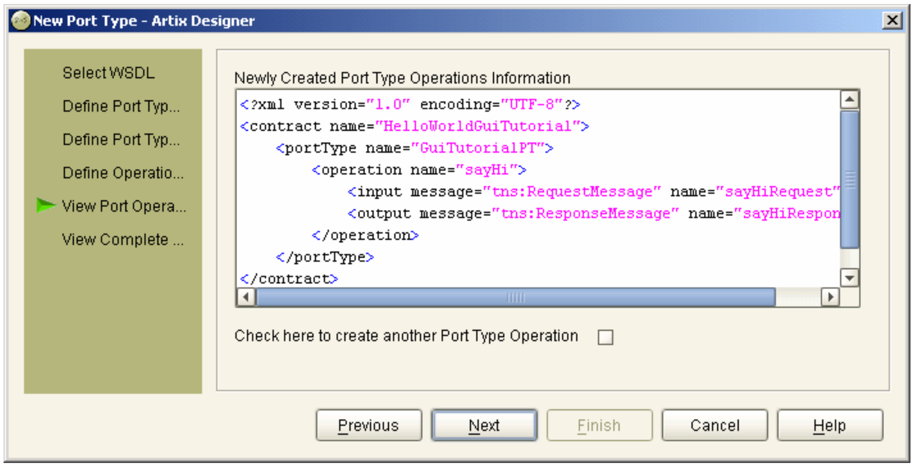
Type	Message	Name
input	tns:RequestMessa...	sayHiRequest

Figure 26: *Define Operation Messages*

7. Now, click on the Type drop down list. Note that input no longer appears in the listing; an operation can have only one input message. Select **output** from the Type list then 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 button to transfer the ouptut message to the Operation Messages control.

   If you click on the Type drop down list you will see 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 button and review the content in the View Port Operations Summary screen. Since this example only requires one portType, click on the Next and then Finish buttons.



Newly Created Port Type Operations Information

```
<?xml version="1.0" encoding="UTF-8"?>
<contract name="HelloWorldGuiTutorial">
    <portType name="GuiTutorialPT">
        <operation name="sayHi">
            <input message="tns:RequestMessage" name="sayHiRequest"
            <output message="tns:ResponseMessage" name="sayHiRespon
        </operation>
    </portType>
</contract>
```

Check here to create another Port Type Operation ☐

Figure 27: *Define Operation Messages*

10. Click on the Save icon or select **File | Save**.

11. 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.

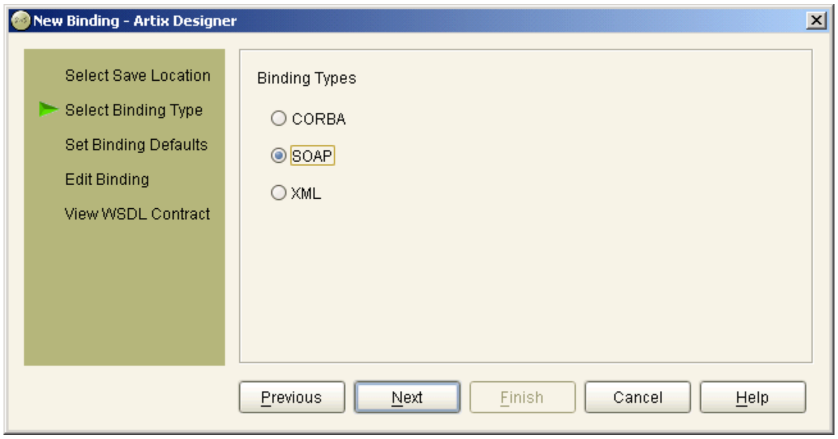
```
<portType name="GuiTutorialPT">
 <operation name="sayHi">
  <input message="tns:RequestMessage" name="sayHiRequest"/>
  <output message="tns:ResponseMessage" name="sayHiResponse"/>
 </operation>
</portType>
```

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 **Contract | New | Binding** (or right-click on the HelloWorldGuiTutorial icon and select **New | Binding** from the popup menu).
2. In the Select WSDL screen, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.
3. In the 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 button.

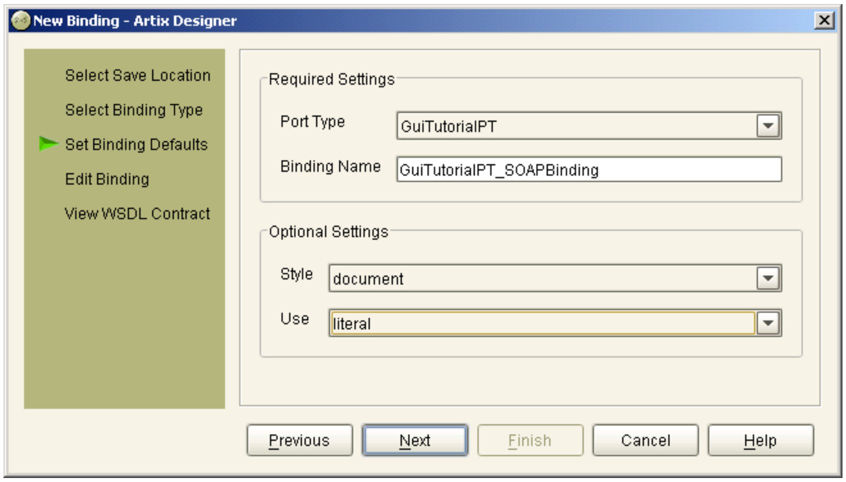


Binding Types

CORBA [✓ SOAP] XML

Figure 28: *Binding Type*

4. In the Select Port Type screen, select the **GuiTutorialPT** entry from the Port Type drop down list.



Required Settings

Port Type: GuiTutorialPT

Binding Name: GuiTutorialPT_SOAPBinding

Optional Settings

Style: document

Use: literal

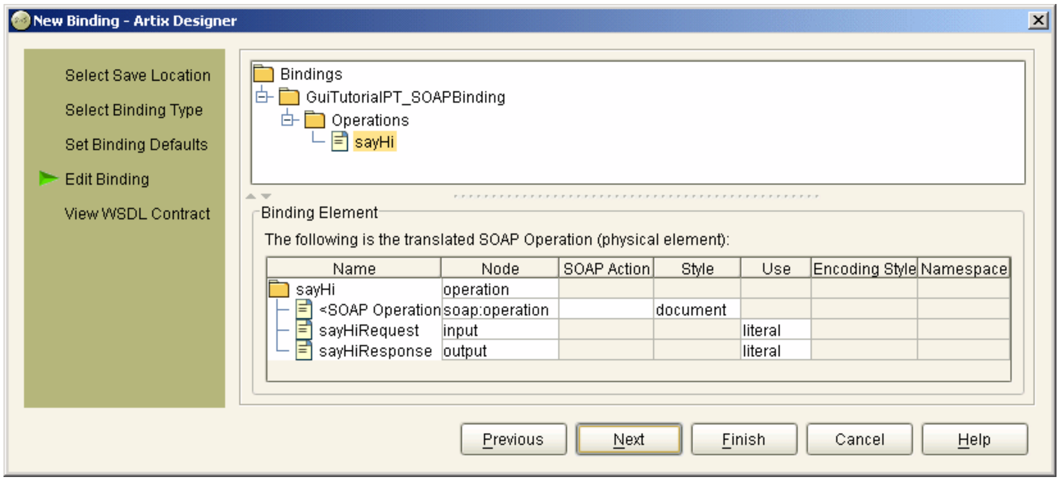
Figure 29: *Port Type*

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 Optional Settings 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 button.

6. In the Edit Binding screen, highlight the sayHi icon representing your operation and review the binding details. Click the Next button.



The following is the translated SOAP Operation (physical element):

Name	Node	SOAP Action	Style	Use	Encoding Style	Namespace
sayHi	operation					
<SOAP Operation	soap:operation		document			
sayHiRequest	input			literal		
sayHiResponse	output			literal		

Figure 30: *Port Type*

7. In the View WSDL Contract screen, you can review the new content that will be added to the WSDL file. Finally, click the Finish button.
8. Click on the Save icon or select **File | Save**.
9. 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.

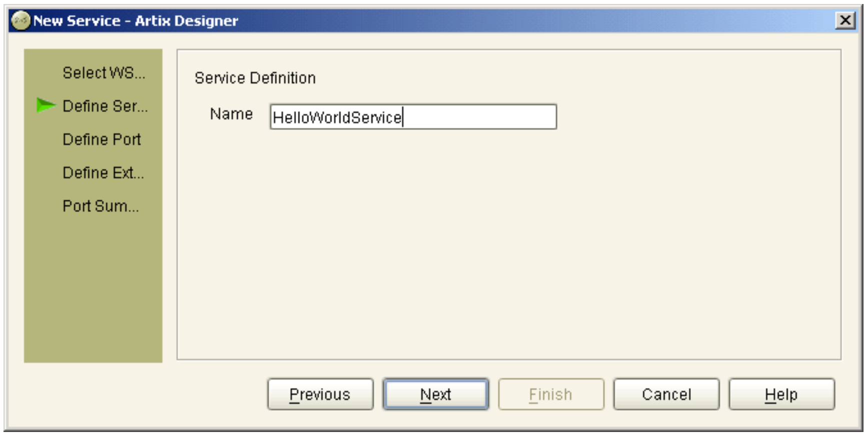
```
<binding name="GuiTutorialPT_SOAPBinding"
         type="tns:GuiTutorialPT">
 <soap:binding style="document"
   transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="sayHi">
  <soap:operation soapAction="" style="document"/>
  <input name="sayHiRequest">
   <soap:body use="literal"/>
  </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 **Contract | New | Service** (or right-click on the HelloWorldGuiTutorial icon and select **New | Service** from the popup menu).
2. In the Select WSDL window, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.
3. In the Define Service window, enter **`HelloWorldService`** into the Name text box. Click the Next button.

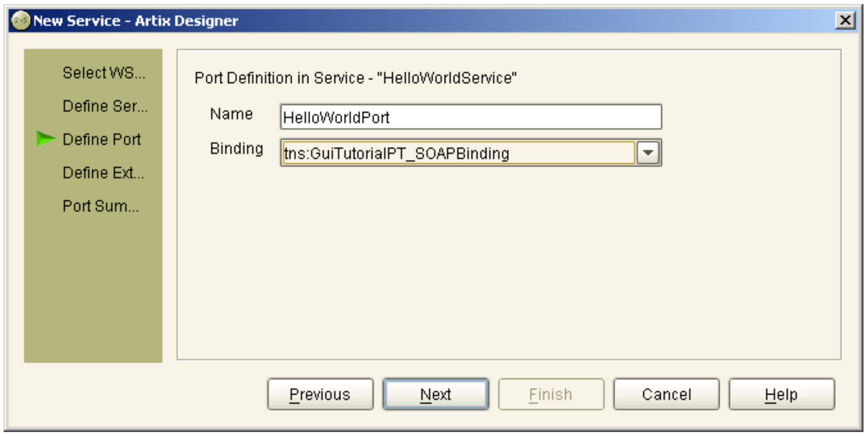


Service Definition

Name: HelloWorldService

Figure 31: *New Service*

4. In the 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 button.



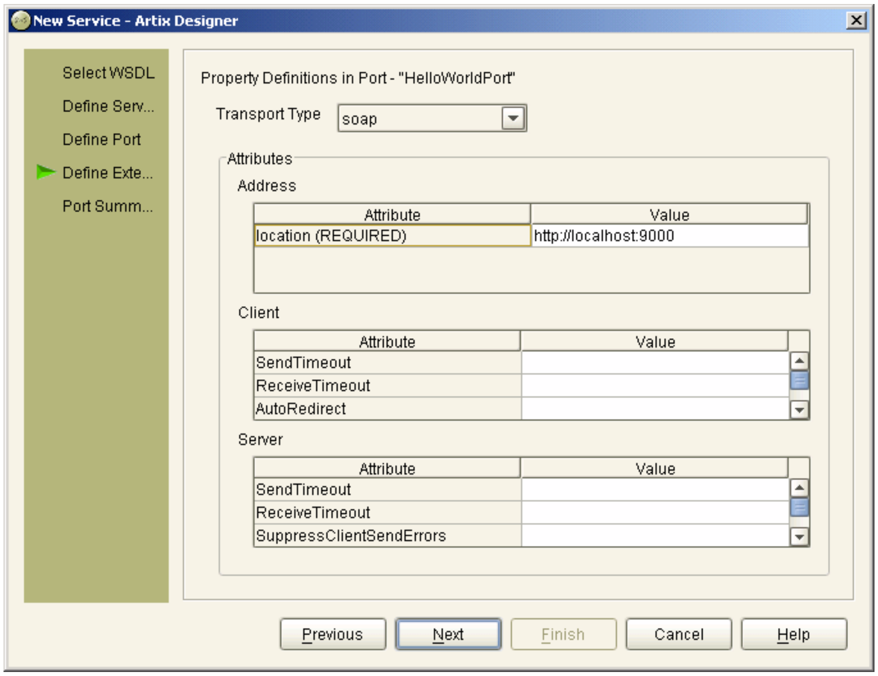
Port Definition in Service - "HelloWorldService"

Name: HelloWorldPort

Binding: tns:GuiTutorialPT_SOAPBinding

Figure 32: *New Service*

5. In the 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.



Property Definitions in Port - "HelloWorldPort"

Transport Type: soap

Attributes

Address

Attribute	Value
location (REQUIRED)	http://localhost:9000

Client

Attribute	Value
SendTimeout	
ReceiveTimeout	
AutoRedirect	

Server

Attribute	Value
SendTimeout	
ReceiveTimeout	
SuppressClientSendErrors	

Figure 33: *New Service*

6. Click the Next button. In the Port Summary window, you can review the new content that will be added to the WSDL file. Finally, click the Finish button.
7. Click on the Save icon or select **File | Save**.

8. 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. The designer added this prefix definition to the WSDL file during the service definition effort.

Developing an Application

Project Summary

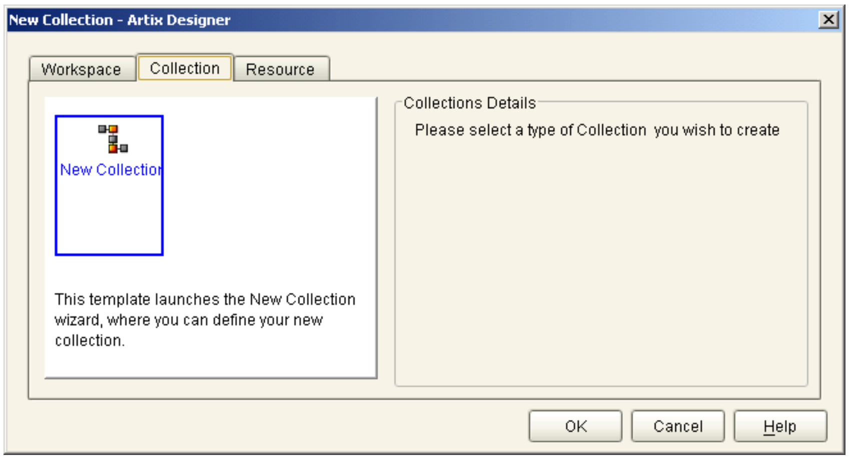
You have now completed writing the WSDL file. Currently the file is located under the Shared Resources icon within the Artix Designer and in the `GuiTutorial\Resources` directory on your drive. Other than the fact that this WSDL file uses element types and document/literal encoding, this WSDL file is 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 creating Collections, which are the resources that are used by a Web service component.

The Client Application

When you create a new collection, Artix places a corresponding icon under the Collections icon in the designer's explorer panel.

1. To create a collection, select **File | New | Collection**. (Alternatively, you can highlight the Collections icon, right click, and select **New Collection** from the popup menu).
2. In the New Collection dialog, select the **New Collection** icon and click the OK button.



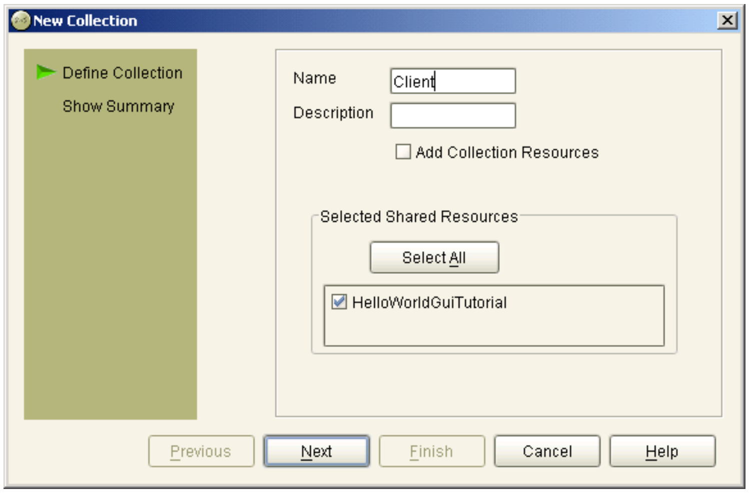
This template launches the New Collection wizard, where you can define your new collection.

Please select a type of Collection you wish to create

Figure 34: *New Collection Wizard*

3. In the Define Collection screen, enter **`Client`** into the Name text box. Note that the designer automatically adds the HelloWorldGuiTutorial resource to the collection. If you had a workspace with multiple entries

under the Shared Resources icon, you might not want to include each resource in every collection. Simply uncheck the entry and it will be excluded.



Name: Client

Description:

☐ Add Collection Resources

Selected Shared Resources

☑ HelloWorldGuiTutorial

Figure 35:

4. Click on the Next and Finish buttons.
5. Click on the Save icon or select **File | Save**.

The designer's explorer panel now includes an icon representing the Client application. Note the nested icon and italic font representing the included resource file. This format indicates that the collection's resource is a link to the resource file listed under the Shared Resources icon and not a resource uniquely associated with this application.

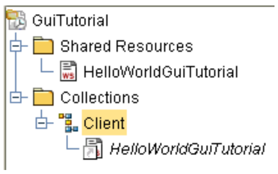


Figure 36: *New Client item*

In a more complex application, your collection might include both shared resources and resources specific to the application. In this situation, you could create a new resource within the collection rather than as a shared resource.

The Server Application

Following the same procedure, create a second collection named **`Server`** that also includes the **HelloWorldGuiTutorial** resource.

Generating Starting Point Code

Approach

The Artix Designer is capable of generating starting point code for multiple platforms and operating systems and for multiple implemenation languages. Depending on your objectives, you may want to generate only client code, only server code, or only some of the files produced by the wsdltocpp or wsdltojava utilities. You provide this information to the designer by defining a Deployment Profile and Deployment Bundle.

- The Deployment Profile is a collection of platform, operating system, and implementation language specifications.
- The Deployment Bundle is a collection of application related specifications.

Defining Deployment Profiles

A profile may be used with multiple deployment bundles in generating code for the same platform, operating system, and implementation language.

The C++ Deployment Profile

1. Highlight the icon representing your workspace (the GuiTutorial icon) and select **File | New | Deployment Profile** (or right click on the GuiTutorial icon and select **New | Deployment Profile** from the popup menu).
2. In the Profile Details screen, enter **`cxx_profile`** for the Name entry and select **Windows** from the Operating System drop down list.
3. In the Artix Location screen, confirm that the paths to your Artix Installation Directory and `artix_env` script files are correct. Select the **C++** radio button and click on the Next and Finish buttons.
4. Click on the Save icon or select **File | Save**.

This profile will be used to generate C++ code for the Windows operating system.

The Java Deployment Profile

1. Highlight the icon representing your workspace (the GuiTutorial icon) and select **File | New | Deployment Profile** (or right click on the GuiTutorial icon and select **New | Deployment Profile** from the popup menu).
2. In the Profile Details screen, enter **`java_profile`** for the Name entry and select **Windows** screen the Operating System drop down list.
3. In the Artix Location window, confirm that the paths to your Artix Installation Directory and `artix_env` script files are correct. Select the **Java** radio button and click on the Next and Finish buttons.
4. Click on the Save icon or select **File | Save**.

This profile will be used to generate Java code for the Windows operating system.

Defining Deployment Bundles

Since you have separate collections for the client and server applications, and you want to generate both C++ and Java starting point code, you will need to define four deployment bundles:

- C++ client
- C++ server
- Java client
- Java server

The Client Deployment Bundles

Create two deployment bundles, one containing the specifications for your C++ client application and the second containing the specifications for your Java client application.

The C++ Client Bundle

1. Highlight the icon representing the Client collection.
2. Right click and select **New | Deployment Bundle** from the popup menu. (Alternatively, select **File | New | Deployment Bundle**).
3. In the Bundle Details screen, enter **`cxx_client`** into the Name text box. In the Location text box, and enter the path to directory into which you want to generate the starting point code. For this application, store the files in the suggested location:
   `C:\IONA\artix\2.0\demos\GuiTutorial\Client\cxx_client`.

   Now, select the **cxx_profile** entry from the Deployment Profile drop down list.

   Finally, select the **Client** radio button and click the Next button.
4. In the Code Generation screen, confirm the target directory for the code generation process. Select **HelloWorldService** and **HelloWorldPort** from the service and port drop down lists. Since your WSDL file only contains one service/port, these lists only contain one item.

   Finally, enter **`GUI`** as the value for the Namespace within the Code Generation Options grouping. This entry becomes the C++ namespace within the generated code.
5. Click the Next button twice, and then the Finish button.
6. Click on the Save icon or select **File | Save**.

The Java Client Bundle

1. Highlight the icon representing the Client collection.
2. Right click and select **New | Deployment Bundle** from the popup menu. (Alternatively, select **File | New | Deployment Bundle**).
3. In the Bundle Details screen, enter **`java_client`** into the Name text box. In the Location text box, and enter the path to directory into which you want to save the starting point code. For this application, store the files in the suggested location:

   `C:\IONA\artix\2.0\demos\GuiTutorial\Client\java_client`.

   Then, select the **java_profile** entry from the Deployment Profile drop down list.

   Finally, select the **Client** radio button and click the Next button.
4. In the Code Generation screen, confirm the target directory for the code generation process. Select **HelloWorldService** and **HelloWorldPort** from the service and port drop down lists. Since your WSDL file only contains one service/port, these lists only contain one item.

   Finally, click the **Override Namespace as packaging name** checkbox enter **`com.iona`** as the value for the Java Package within the Code Generation Options grouping. This entry becomes the Java package hierarchy within the generated code.
5. Click the Next button twice, and then the Finish button.
6. Click on the Save icon or select **File | Save**.

The Server Deployment Bundles

Create two deployment bundles, one containing the specifications for your C++ server application and the second containing the specifications for your Java server application.

The C++ Server Bundle

1. Highlight the icon representing the Server collection.
2. Right click and select **New | Deployment Bundle** from the popup menu. (Alternatively, select **File | New | Deployment Bundle**).

3. In the Bundle Details screen, enter **cxx_server** into the Name text box. In the Location text box, and enter the path to directory into which you want to save the starting point code. For this application, store the files in the suggested location:

   `C:\IONA\artix\2.0\demos\GuiTutorial\Server\cxx_server.`

   Then, select the **cxx_profile** entry from the Deployment Profile drop down list.

   Finally, select the **Server** radio button and click the Next button.
4. In the Code Generation screen, confirm the target directory for the code generation process. Select **HelloWorldService** and **HelloWorldPort** from the service and port drop down lists. Since your WSDL file only contains one service/port, these lists only contain one item.

   Finally, enter **GUI** as the value for the Namespace within the Code Generation Options grouping. This entry becomes the C++ namespace within the generated code.
5. Click the Next button twice, and then the Finish button.
6. Click on the Save icon or select **File | Save**.

The Java Server Bundle

1. Highlight the icon representing the Server collection.
2. Right click and select **New | Deployment Bundle** from the popup menu. (Alternatively, select **File | New | Deployment Bundle**).
3. In the Bundle Details screen, enter **java_server** into the Name text box. In the Location text box, and enter the path to directory into which you want to save the starting point code. For this application, store the files in the suggested location:

   `C:\IONA\artix\2.0\demos\GuiTutorial\Server\java_server.`

   Then, select the **java_profile** entry from the Deployment Profile drop down list.

   Finally ,select the **Server** radio button and click the Next button.
4. In the Code Generation screen, confirm the target directory for the code generation process. Select **HelloWorldService** and **HelloWorldPort** from the service and port drop down lists. Since your WSDL file only contains one service/port, these lists only contain one item.

Finally, click the **Override Namespace as package name** checkbox, and enter `com.iona` as the value for the Java Package within the Code Generation Options grouping. This entry becomes the Java package hierarchy within the generated code.

5. Click the Next button three times, and then the Finish button. The skipped windows are used when specifying more advanced code generation processes.
6. Click on the Save icon or select **File | Save**.

Generating the C++ and Java Code

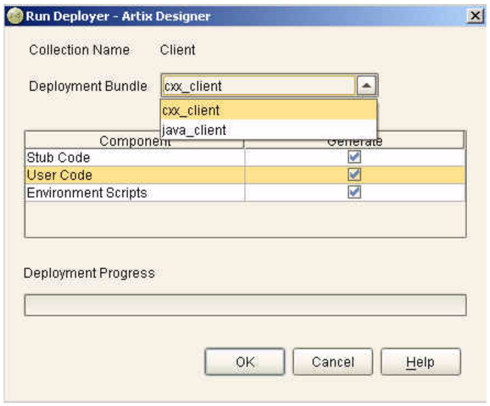
Now that you have the deployment profile and associated deployment bundles, generating the code is simple.

The Client Applications

You will first generate the code for the C++ client application and then repeat the process for the Java client application.

The C++ Client

1. Highlight the Client icon and select **Tools | Run Deployer**. (Alternatively, right click on the Client icon and select **Run Deployer** from the popup menu).
2. In the Run Deployer screen, select **cxx_client** from the Deployment Bundle drop down list. Note that the list only displays bundles that were defined for the client application.
3. Note that the deployer is preconfigured to generate Stub Code and Environment Scripts for the client application. Since you want to generate all of the client starting point code, check the box under the Generate heading for the User Code component.



Collection Name: Client

Deployment Bundle: cxx_client (cxx_client, java_client)

Component	Generate
Stub Code	✓
User Code	✓
Environment Scripts	✓

Deployment Progress

4. Click the OK button. The code generation process runs to completion. Click the Close button to dismiss the window.

The Java Client

1. Highlight the Client icon and select **Tools | Run Deployer**. (Alternatively, right click on the Client icon and select **Run Deployer** from the popup menu).
2. In the Run Deployer screen, select **java_client** from the Deployment Bundle drop down list. Note that the list only displays bundles that were defined for the client application.
3. Note that the deployer is preconfigured to generate Stub Code and Environment Scripts for the client application. Since you want to generate all of the client starting point code, check the box under the Generate heading for the User Code component.
4. Click the OK button. The code generation process runs to completion. Click the Close button to dismiss the window.

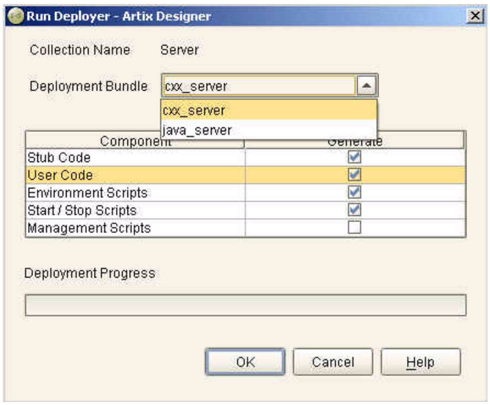
The Server Applications

You will now generate the code for the C++ server application and then repeat the process for the Java server application.

The C++ Server

1. Highlight the Server icon and select **Tools | Run Deployer**. (Alternatively, right click on the Server icon and select **Run Deployer** from the popup menu).
2. In the Run Deployer screen, select **cxx_server** from the Deployment Bundle drop down list.

3. Note that the deployer is preconfigured to generate Stub Code and Environment Scripts for the client application. Since you want to generate all of the server starting point code, check the box under the Generate heading for the User Code component.



Collection Name Server

Deployment Bundle cxx_server

cxx_server
java_server

Component	Generate
Stub Code	✓
User Code	✓
Environment Scripts	✓
Start / Stop Scripts	✓
Management Scripts	☐

Deployment Progress

4. Click the OK button. The code generation process runs to completion. Click the Close button to dismiss the window.

The Java Server

1. Highlight the Server icon and select **Tools | Run Deployer**. (Alternatively, right click on the Server icon and select **Run Deployer** from the popup menu).
2. In the Run Deployer screen, select **java_server** from the Deployment Bundle drop down list.
3. Note that the deployer is preconfigured to generate Stub Code and Environment Scripts for the client application. Since you want to generate all of the server starting point code, check the box under the Generate heading for the User Code component.
4. Click the OK button. The code generation process runs to completion. Click the Close button to dismiss the window.

Close the Artix Designer

You have now finished with the Artix Designer. Close the application by selecting **File | Exit**.

Completing the Code

The Generated Code

Through the code generation process you created four applications: the C++ client, the Java client, the C++ server, and the Java server. All these applications will compile and run. However, since there is no business logic in the sayHi method body, and since the C++ client code does not actually make a request against the Web service, running the applications will not produce output. You must now complete the coding in the files representing the C++ and Java implementation objects and in the C++ client mainline file. The Java client mainline file is actually a complete, albeit very basic, application.

In this section

This section covers the following tasks:

The C++ Client Code	page 95
The C++ Server Code	page 97
The Java Client Code	page 99
The Java Server Code	page 100

The C++ Client Code

The code generation process creates the following files.

GuiTutorialPT.h

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 InParameter & InPart,
          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.

GuiTutorialPTClient.h/.cxx

These files represent the client proxy class. Your client mainline code must instantiate an instance of this class to invoke on the Web service. The proxy class includes multiple constructors, a destructor, and a method for each of the Web service's operations.

In this simple application your client code will use the no argument constructor. Alternative constructors allow you to change the WSDL file, service name, or port name initialization values. One constructor allows initialization from an Artix reference. Review the product documentation to learn how and when to use these alternate constructors.

HelloWorldGuiTutorial_ wsdlTypes.h/.cxx

These files are common to both the client and server applications and include the definitions and implementations for the classes that represent your application specific types. *You must review the contents of these files to understand how to use the APIs of these classes.*

HelloWorldGuiTutorial_ wsdlTypesFactory.h/.cxx

These files are common to both the client and server applications and include definitions and implementations for the factory methods required if your application specific types includes the anyType.

For this application, you do not need to be concerned with the contents of these files.

GuiTutorialPTClientSample

For the client application, you only need to work with the `GuiTutorialPTClientSample.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.

```
InParameter    InPart;
OutParameter   OutPart;

InPart.setvalue("Artix User");

client.sayHi ( InPart,  OutPart);

cout << "sayHi returned: " + OutPart.getvalue() << endl;
```

The C++ Server Code

The code generation process creates the following files.

GuiTutorialPT.h	The header file that is common to both the client and server applications.
GuiTutorialPTServer.h/.cxx	These files represent the server stub class. Your code does not directly use this class. Rather, the implementation class is a subclass of the GuiTutorialPTServer class.
HelloWorldGuiTutorial_ wsdlTypes.h/.cxx	These files are common to both the client and server applications and include the definitions and implementations for the classes that represent your application specific types. *You must review the contents of these files to understand how to use the APIs of these classes.*
HelloWorldGuiTutorial_ wsdlTypesFactory.h/.cxx	These files are common to both the client and server applications and include definitions and implementations for the factory methods required if your application specific types includes the anyType. For this application, you do not need to be concerned with the contents of these files.
GuiTutorialPTServerSample.cxx	This file represents the server mainline application. For this application you do not need to edit the contents of this file. The server mainline instantiates an instance of the implementation class and registers it with the Artix runtime. The process then enters an event loop to process incoming requests.
GuiTutorialPTImpl.h/.cxx	These files represent your Web service's implementation class. The `GuiTutorialPTImpl.cxx` file contains compilable code, but there is no processing logic in the method bodies. For the server application, you only need to add processing logic to the implementation class' sayHi method. In a text editor, open the `GuiTutorialPTImpl.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.setvalue("Hello " + InPart.getvalue());
```

The Java Client Code

The code generation process creates the following files.

GuiTutorialPT.java	This file represents the interface definition common to both the client and server applications. This interface defines the operation offerred by the Web service. `public String sayHi(String inPart) throws RemoteException`
GuiTutorialPTTypes.java and GuiTutorialPTTypesFactory.java	Definition of the classes that create and manage anyTypes defined in your WSDL file.
GuiTutorialPTDemo.java	This file represents the client mainline application. For this simple example, the generated code in this file represents a fully functional application. With a more involved application, you would use this code as a template for writing a more complex client application.

The Java Server Code

The code generation process creates the following files.

GuiTutorialPT.java	The interface definition common to both the client and server applications.
GuiTutorialPTTypes.java and GuiTutorialPTTypesFactory.java	Definition of the classes that create and manage anyTypes defined in your WSDL file.
GuiTutorialPTServer.java	Starting point code for a server mainline application. For this simple example, the generated code in this file represents a fully functional application. With a more involved application, you might extend the generated code.
GuiTutorialPTImpl.java	Starting point code for your Web service's implementation class. For this tutorial you will need to add coding to the method bodies corresponding to the Web service operations.

In a text editor, open the file `GuiTutorialPTImpl.java`. The code you add to the sayHi method body is:

```
return "Hello " + inPart;
```

Compiling the Applications

Compiling the C++ Applications

To build the C++ client and server applications:

1. Open a command window to the `<installationDirectory>\artix\2.0\bin` directory and run the `artix_env.bat` file.
2. Move to the `<installationDirectory>\artix\2.0\demos\GuiTutorial\Client\cxx_client\src\cpp` directory by issuing the command:

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

3. Build the client application with the command:

```
nmake all
```

4. Move to the `<installationDirectory>\artix\2.0\demos\GuiTutorial\Server\cxx_server\src\cpp` directory by issuing the command:

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

5. Build the server application with the command:

```
nmake all
```

Do not close the command window; you will use the same window to compile the Java applications.

Compiling the Java Applications

To build the Java client and server applications:

1. In the open command window, place the current directory onto the CLASSPATH with the command:

```
set CLASSPATH=.;%CLASSPATH%
```

2. Move to the `<installationDirectory>\artix\2.0\demos\GuiTutorial\Client\java_client\src\java` directory.
3. Build the client application with the command:

```
javac com/iona/*.java
```

4. Move to the `<installationDirectory>\artix\2.0\demos\GuiTutorial\Server\java_server\src\java` directory by issuing the command:

```
cd ..\..\..\..\Server\java_server\src\java
```

5. Build the server application with the command:

```
javac com/iona/*.java
```

Close the command window.

Running the Application

The C++ Applications

To run the C++ client against the C++ server:

1. Open a command window to the `<installationDirectory>\artix\2.0\bin` directory and run the `artix_env.bat` file.
2. Move to the `<installationDirectory>\artix\2.0\demos\GuiTutorial\Server\cxx_server\src\cpp` directory by issuing the command:

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

3. Start the server process with the command:

```
start server
```

   The server process starts in a new command window.

4. Move to the `<installationDirectory>\artix\2.0\demos\GuiTutorial\Client\cxx_client\src\cpp` directory by issuing the command:

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

5. Run the client process with the command:

```
client
```

   Observe the message in the client process window.

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

Do not close the command window; you will use the same window to run the Java applications.

The Java Applications

To run the Java client against the Java server:

1. In the open command window, place the current directory onto the CLASSPATH with the command:

```
set CLASSPATH=.;%CLASSPATH%
```

2. Move to the `<installationDirectory>\artix\2.0\demos\GuiTutorial\Server\java_server\src\java` directory.
3. Start the server process with the command:

```
start java com.iona.GuiTutorialPTServer
```

   The server process starts in a new command window.
4. Move to the `<installationDirectory>\artix\2.0\demos\GuiTutorial\Client\java_client\src\java` directory by issuing the command:

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

5. Run the client process with the command:

```
java com.iona.GuiTutorialPTDemo sayHi <a_Name>
```

   Observe the message in the client process window.
6. Stop the server process by issuing the command Ctrl-C in its command window.

Interoperability

You may also run the C++ client against the Java server or the Java client against the C++ server. Use the steps in the previous sections as a guide.

CHAPTER 5

Faults and Exceptions

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

In this chapter

This chapter discusses the following topics:

Raising Exceptions	page 106
Handling Runtime Exceptions	page 107
Working with WSDL Faults	page 109
Developing An Application	page 113

Raising Exceptions

Types of Artix Exceptions

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++ or Java 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. In C++ applications, 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. In Java applications, these exceptions are returned as java.lang.Exception.

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++ and Java 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 code generation utilities, these fault messages become C++ or Java 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\2.0\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\2.0\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 C++ client application includes a try{} 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;
}
```

The Java client application's main method includes a throws Exception clause. To handle the runtime exceptions, you could place a try{} block around the remote method invocation followed by a corresponding catch{} block.

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 when using the document literal paradigm, 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.

When you run the wsdltojava utility, two Java 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 FaultDetailsException corresponds to the UnknownUser message. This class includes instance variables corresponding to the FaultMsg and FaultID elements and accessor methods to manipulate these values. Your code uses the FaultDetailsException directly; there is no need to use the FaultDetails class.

Throwing the Exception

While both C++ class definitions include a copy constructor, neither class includes a constructor that allows you to set the instance variables. Consequently, 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;
```

In the Java FaultDetailsException class, there is a constructor that allows you to set values for the FaultMsg and FaultID. Consequently, your code can instantiate, initialize, and throw the exception in a single line of code.

```
throw new FaultDetailsException("User unknown to me", 200);
```

Handling the Exception

In C++, 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;
}
```

In Java, the client mainline code generated by the wsdltojava utility includes a catch{} block to process the FaultDetailsException. Do not be concerned with the exception stack trace; the generated code prints this information.

```
catch ( com.iona.FaultDetailsException ex )
  {
    System.out.println
     ("Exception: com.iona.FaultDetailsException has Occurred.");
    ex.printStackTrace();
  }
```

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 Shared Resources icon and click on the WSDL tab. The WSDL file contents are displayed in the panel.
2. Select **Contract | Edit | Services** (or right click on the HelloWorldGuiTutorial icon and select **Edit | Services** from the popup menu).
3. In the Edit Services screen, highlight the HelloWorldService icon in the top panel and click on the Delete button. Confirm your decision by clicking the Yes button. Then click on the Apply and OK buttons. View the WSDL file contents and confirm that the `<service>...</service>` section has been removed.
4. Select **Contract | Edit | Bindings** (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 the Delete button. Confirm you decision by clicking the Yes button. The click on the Apply and OK 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 under the Collections icon and view the WSDL file contents, you will observe the edited content. These icons actually represent links to the WSDL file in the Shared Resources directory, so edits are applied to the file associated with the Client and Server collections.

Create the data types

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

1. Highlight the HelloWorldGuiTutorial icon and select **Contract | New | Type**, (or right click on the HelloWorldGuiTutorial icon and select **New | Type** from the popup menu).
2. In the Select WSDL screen, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.
3. In the Define Type Properties window, enter **`FaultDetails`** into the Name text box and select the **complexType** radio button. Click the Next button.
4. In the Define Type Attributes screen, 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 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 button. Your sequence now includes two members. Click the Next and Finish 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 Define Type Properties window.
7. In the 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.

Define the fault message

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 **Contract | New | Message** menu entry, (or right click on the HelloWorldGuiTutorial icon and select **New | Message** from the popup menu).
2. In the Select WSDL screen, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.
3. In the Define Message Properties screen, enter **`UnknownUser`** into the Name text box. Click the Next 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 button; your part is added to the Part List control. Finally, click the Next button.
5. In the View Summary screen, you can review the content that will be added to the WSDL file. Click the Finish button.

Edit the portType definition

Finally you need to edit the portType definition.

1. Highlight the HelloWorldGuiTutorial icon and select **Contract | Edit | Port Types** (or right click and select Edit > Port Types from the popup menu).
2. In the Edit Port Types screen, highlight the **sayHi** operation in the top panel and then click the **Edit** button below the Operation Messages grouping control.
3. In the Edit Operation Messages screen select **fault** from the Type drop down list. From the Message drop down list select the **tns:UnknownUser** entry. Click on the Add button, which adds the fault message to the Operation Messages listing. Finally, click on the Apply, OK, and OK buttons.
4. Review the contents of the WSDL file and confirm that the sayHi operation now includes a fault element.

Recreate the SOAP binding

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

1. Highlight the HelloWorldGuiTutorial icon and select **Contract | New | Binding**, (or right click on the HelloWorldGuiTutorial icon and select **New | Binding** from the popup menu).
2. In the Select WSDL window, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.
3. In the Select Binding Type window, select the **SOAP** radio button. Click the Next button.
4. In the 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 Optional Settings 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 button.
6. In the Edit Binding window, highlight the **sayHi** icon representing your operation and review the binding details. Click the Next button.
7. In the View WSDL Contract window, you can review the new content that will be added to the WSDL file.
8. Finally, click the Finish 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 **Contract | New | Service**, (or right click on the HelloWorldGuiTutorial icon and select **New | Service** from the popup menu).
10. In the Select WSDL screen, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click the Next button.
11. In the Define Service window, enter **`HelloWorldService`** into the Name text box. Click the Next button.
12. In the Define Port window, enter **`HelloWorldPort`** into the Name text box and select **GuiTutorialPT_SOAPBinding** from the Binding drop down list. Click the Next button.

13. In the Define Extensor Properties screen, 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 screen you choose.
14. Click the Next button. In the Port Summary window, you can review the new content that will be added to the WSDL file. Finally, click the Finish button.

Generating the Application Code

You will use the Artix Designer to generate starting point code for the application. Since the Deployment Profiles and Deployment Bundles have already been created, you only need to regenerate the starting point code and then reimplement and recompile the applications.

Repeat the steps described in "Generating the C++ and Java Code" on page 91.

Completing the Code

In the C++ 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.getvalue() != "Artix User")
{
  FaultDetails faultData;
  faultData.setFaultMsg("User unknown to me");
  faultData.setFaultID(200);

  UnknownUserException ex;
  ex.settheFault(faultData);

  throw ex;
}
OutPart.setvalue("Hello " + InPart.getvalue());
```

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

```
GuiTutorial::InParameter    InPart;
GuiTutorial::OutParameter   OutPart;

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

In the Java application, the client mainline produced by the wsdltojava utility already includes a catch{} block to handle the FaultDetailsException.

```
try {
    if ("sayHi".equals(args[0]))
    {
        . . .
    }
catch ( com.iona.FaultDetailsException ex )
    {
        System.out.println
          ("Exception: com.iona.FaultDetailsException
            has Occurred.");
        ex.printStackTrace();
    }
```

You need to add code to the GuiTutorialPTImpl.java file, providing an implementation for the sayHi method that throws the FaultDetailsException.

```
public String sayHi(String inPart) throws FaultDetailsException
{
    String _return = null;
    if (inPart.equals("Artix User"))
    {
        _return = "Hello " + inPart;
    }
    else
    {
        FaultDetailsException fd = new FaultDetailsException
            ("User unknown to me", 200);
        throw fd;
    }
    return _return;
}
```

Building the Application

Now that you have completed coding, you can build the application. Repeat the steps described in “Compiling the Applications” on page 101.

Running the Application

Run the applications as described in “Running the Application” on page 103.

When running the C++ client, Artix User is supplied to the Web service when you do not provide a name on the command line. If you provide a name on the command line, that name is passed to the Web service. When running the Java client supply "Artix User" or another name as the required parameter (quotation marks around Artix User are important).

Note that the server simply throws the exception, which the client applications catch and display if the name is not Artix User.

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	page 122
The WSDL File	page 124
The Application Code	page 127

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.

You can use the WSDL file described in the next section and the wsdltocpp, or wsdltojava, utility (discussed in Chapter 3, "Coding the Web Service") to generate the starting point code.

Finally, you can use 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 in the PDF form of this document the content spans three pages, 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 and wsdltojava utilities, or the Artix Designer, generate all of the files you need to write this application. This chapter provides an example solution for the C++ application.

The C++ Application

- `CalculatorClientSample.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.
- `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;
    Input.getPrincipal().setvalue
      (strtod(argv[1], (char**)NULL));
    Input.getPercentageRate().setvalue
      (strtod(argv[2], (char**)NULL));
    Input.getYears().setvalue(atoi(argv[3]));

    MortgageCalculator::BankAnswer   Output;
    client.CalculateMonthlyPayment ( Input,  Output);

    cout << "Principal: $" << Output.getPrincipal().getvalue()
           << endl;
    cout << "Term: " << Output.getYears().getvalue()
            << " years" << endl;
    cout << "Rate: " << Output.getPercentageRate().getvalue()
           << "%" << endl;
    cout << "Payment: $" << Output.getPayment().getvalue()
           << endl;
  }
  catch (PaymentExceedsLimitException& ex)
  {
    cout << endl << "PaymentExceedsLimitException Raised"
           << endl;
    cout << "\tMonthly payment too large: $"
           << ex.getCalculatedPayment().getvalue() << 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().getvalue();
  IT_Bus::Double rate = Input.getPercentageRate().getvalue();
  IT_Bus::Int term = Input.getYears().getvalue();

  // 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.setvalue(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 and Running the Applications

The C++ Application

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

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.