Web Service Patterns: Java Edition

PAUL B. MONDAY

Web Service Patterns: Java Edition
Copyright (©)2003 by Paul B. Monday

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN (pbk): 1-59059-084-8

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Technical Reviewer: Kunal Mittal

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Assistant Publisher: Grace Wong

Project Managers: Sofia Marchant and Beth Christmas

Copy Editor: Kim Wimpsett

Compositor and Proofreader: Kinetic Publishing Services

Indexer: Valerie Robbins

Artist: Kinetic Publishing Services

Cover Designer: Kurt Krames

Production Manager: Kari Brooks

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 4

Exploring the Architecture Adapter Pattern

THE PREVIOUS CHAPTER EXPLAINED the Web Service architecture in terms of the Service-Oriented Architecture pattern. Web Services implement the service-oriented architecture using Simple Object Access Protocol (SOAP) as a communication mechanism between services and Universal Description, Data, and Discovery (UDDI) as a directory implementation. Web Services Description Language (WSDL) describes the interface to a Web Service. Web Services do not support inheritance or polymorphism, and they do not delve into the service implementation techniques. On the other hand, Web Services build on such a small set of primitive types and concepts that you can use virtually any semi-modern programming language to build service implementations.

In the beginning days of Web Services, Java programmers coded socket listeners that received SOAP messages, parsed them, and called the proper code in the Java language. There are significant challenges to writing the code that converts between the Web Service implementation of a service-oriented architecture and the Java platform. Thankfully, tools, such as some of the tools included with Apache Axis, automate the creation of this code that converts data from the Web Service architecture to the Java architecture. At design time, it is easy to wrap up the responsibilities of this conversion from Web Services to Java in a simple design pattern: the Architecture Adapter pattern.

In this chapter, you will look at the architecture adapter as a generic pattern. You then dissect the architecture adapter in terms of service deployment in Apache Axis and service consumption from Java.

Facilitating Communication Between Architectures

Before digging too deeply into the architecture adapter, it is worth taking a few moments to discuss exactly what is the true nature of architecture. Once you have an understanding of some of the highlights of architecture, you can explore the issues surrounding Web Services and the Java platform.

What Is Architecture?

There are many definitions of architecture, and there are many interpretations of those definitions. The architecture of a system, in general, discusses its structure through a set of the following:

- Architecture components that describe the core blocks of a system
- Connectors that describe the mechanisms and expectations on communication paths between components
- Task flows that illustrate how an application uses the components and connectors to fulfill a requirement

The architecture of a system becomes very complex very quickly, especially for an enterprise-class application. Architecture documents usually consist of 100 or more pages.

For this discussion, you need to focus on the content of the architecture with respect to the components and connectors. Typically, at the root of architecture are an architectural style and a variety of architecture patterns. Style is somewhat difficult to describe. When you think of an architectural style, you can compare it to physical building architecture. In the physical world, an architectural style dictates the dominant feel of a building. Typically, neighborhoods all contain the same architectural style. Unique architectural styles, such as Western Indian architecture, have elements that make it unique. For the Western Indian architectural style, one thinks of Islamic-style domes, ornate columns, and marble building blocks. The Taj Mahal in India is a perfect example.

Compare the Western Indian style with Frank Lloyd Wright's prairie architectural style, recognized as one of the first original American architectural styles. The Prairie style contains dominant horizontal lines; includes large, sweeping roofs; and weaves common Japanese architectural styles. The style attempts to connect the structure itself with the dominant Midwest prairie landscape.

Now, imagine placing the Taj Mahal next to a home built in the prairie style. The two architectures are fundamentally different, and only a master landscaper could make the transition from one structure to the other structure appear seamless and be functional. The transition contains elements of both architectural styles, yet blends its own techniques to help the transition between the two dominant styles. This transition landscaping is an architecture adapter.

Common elements of software architecture styles include the following:

The dominant communication style: Two common communication styles are *message-based communication* and *call-return communication*. The former is similar to today's message services, such as the Java Message Service (JMS), that provide loose coupling and that lend themselves well to asynchronous communications. Java's method call mechanism is an example of the call-return communication mechanism. In call-return communication, the thread of control originates with and returns to the method caller.

The dominant structuring technique for the functional implementation: Component and object-oriented styles are common in architecture. A component architectural style indicates a tendency toward the loose coupling of components and a high degree of cohesion within the components. This style is valuable when you want to create boundaries between components and allow programmers to easily restructure applications, acquire new units of functionality, and replace pieces of functionality. Object-oriented styles do not stress loose coupling as much. Often, if you want to take an object and use it in another application, you bring many other classes and dependencies with you. In a component style, the act of reusing functionality in an entirely different program is trivial.

No inherent problems exist with either type of architecture. The strengths and weaknesses balance out and often reflect preferences and experiences of the senior architecture staff. Once an architect promotes a particular architectural style, it permeates the entire application implementation.

Using Web Services and Java

Web Services and the Java platform have dramatically different architectural styles. The Web Service architecture, based on the service-oriented architecture, has its roots in the component-based architectural style, and it facilitates all of the dominant communication architectural styles: message-based and call-return. The Java platform is a classic embodiment of the object-oriented architecture style with a call-return communication style.

Java interacts with Web Services in two scenarios:

- Java serves as the platform for writing a service implementation that turns into a Web Service at deployment time.
- A Java program needs to use one or more Web Services.

The first scenario deals with how to represent programmatic function written with object-oriented techniques into an architecture that minimizes dependencies and does not stress or allow rich class hierarchies. The second scenario deals with representing loosely coupled and relatively flat structures in a rich, object-oriented environment. The communication mechanisms also differ between the two architectures. Either of the two scenarios requires a conversion from Java's call-return communication style to the Web Service dominant message-based communication style.

The challenges in combining the two architectural styles is not as difficult as putting a Prairie style home next to the Taj Mahal, but it is a significant challenge rife with minute details that take several iterations to get correct. As you architect and design your system, it is important that you isolate the component responsible for making the conversion between the architectures. The root motivation of the architecture adapter is to isolate this conversion process out of the primary business components and logic.

Understanding the Structure of an Architecture Adapter

A convenient representation for an architecture adapter is a single component with a set of requirements on it and the boundary interfaces that the requester expects to see. This leaves the details of the mediation between the two architectures to a lower level of design and implementation (see Figure 4-1).

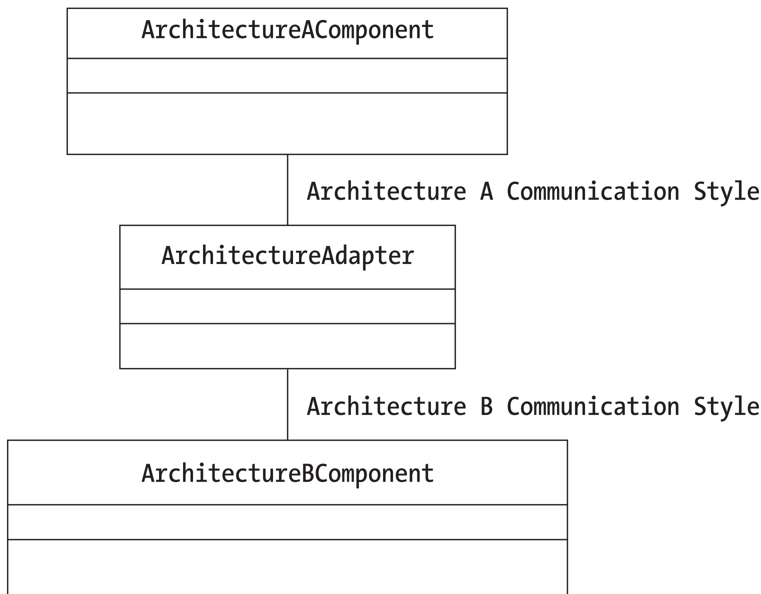

Figure 4-1. Structure of an architecture adapter solution

The low-level design and implementation of an architecture adapter is a bit more involved. Designs will be radically different based on the architectures involved, how reusable and generic the architecture adapter will be, and the facilities readily available for doing the necessary conversions between the architectural styles. The closer the match between the source and target architectures, the thinner and easier it is to write the architecture adapter. Conversely, the further apart that the source and target architectures are, the more complex and difficult it is to write the adapters.

Understanding Components of an Architecture Adapter

Three components make up the structure of a complete solution. A single component, the `ArchitectureAdapter`, holds the responsibilities of mediating between the two architectures involved. The three components are as follows:

ArchitectureAComponent: The A component implements functionality in a particular architectural style. For example, it may implement its functionality using the Java platform and, therefore, use an object-oriented, call-return architectural style.

ArchitectureBComponent: The B component implements complementary functionality in a different architectural style than the A component. Most likely, the complementary functionality did not occur in a preplanned fashion. Instead, the A and B component are often purchased functionality in different software upgrade cycles. For example, a company purchased the A component as it installed an Enterprise Resource Planning (ERP) system and purchased the B component when it decided it needed a Customer Relationship Management (CRM) solution to facilitate its growing customer base.

ArchitectureAdapter: The architecture adapter mediates between the two architecture styles inherent in the `ArchitectureAComponent` and the `ArchitectureBComponent`. To mediate properly, the adapter must offer a natural interface to both components, regardless of the complexities of mediating the service interaction. The implementation must convert one component's architecture entirely to the other component's architecture and maintain the behavior and expectations of both clients. To do this, data styles must be mapped properly, differences in the behavior of communications must be mapped properly, and even such complexities as converting a rich object hierarchy to a flat component interface must be achieved gracefully.

Frequently, the architecture adapter splits into two halves, one that communicates directly with the A component and one that communicates directly with

the B component. The two halves then implement their own architectural style to communicate with each other, as shown in Figure 4-2.

In Figure 4-2, `ClientAdapterA` and `ClientAdapterB` make up the entire functionality of the `ArchitectureAdapter`. By splitting the `ArchitectureAdapter` into halves, it becomes easier to add a third component type into the mix. Without the intermediate architecture style, you must build two new architecture adapters—one to communicate with each of the existing components. With this modified design, you can build a single adapter to communicate into the intermediate architectural style. Welcome to one of the primary motivations for the Web Service architecture: a mechanism to provide mediation between different architectural styles. Web Services implement the Architecture Adapter pattern in such a way that it is simple to mine differing architectures for functionality.

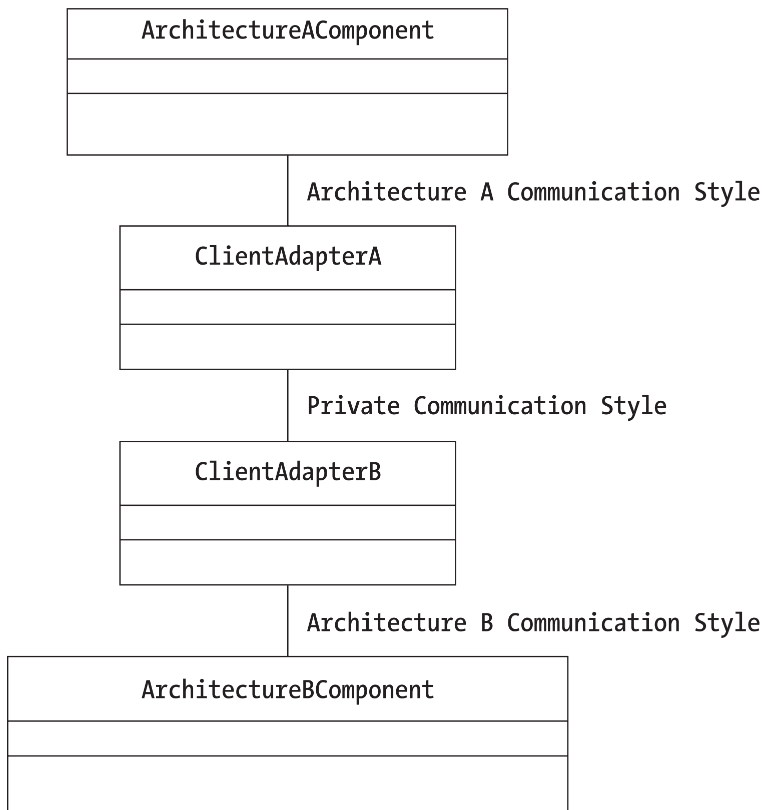

Figure 4-2. Lower-level design of an architecture adapter

Understanding Collaborations Between Architecture Adapter Components

The interesting part of the collaborations between components is in the call style and the data structures passed between components. Instead of using the first structuring from Figure 4-1, I use the second structuring from Figure 4-2 to show a sequence that more closely resembles a Web Service scenario. Figure 4-3 shows the sequence of operations for component A to make a call against component B.

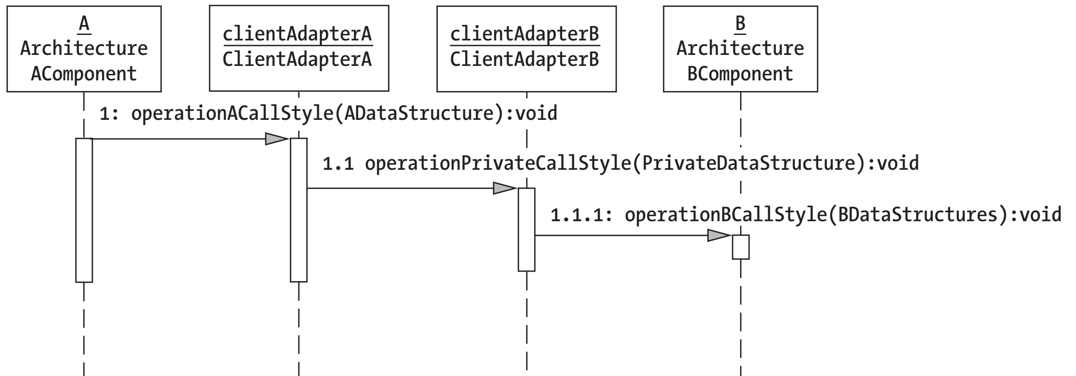

Figure 4-3. Sequence of operation calls between architectures

The call between component instance `A` and `clientAdapterA` takes place in the operational call style inherent in the A architecture and with data structures native to the A architecture. The first client adapter converts the call to a neutral third architectural style and makes a call in this third style with the appropriate data structures to the second client adapter. This second client adapter converts the operation to `B`'s architectural style and makes the call to the `B` component with data structures and a call style native to the B architecture.

Preparing to Implement Architecture Adapters

The Architecture Adapter pattern isolates complexity to the adapter and away from client and target service code. When implementing the Architecture Adapter pattern, you should follow a few general rules. Specific scenarios, such as Web Services to Java, drive more implementation details based on the architectural styles involved:

Establish a common pattern for traversals between architectures and build reusable adapters whenever possible: If you traverse between architectures once in a program, you will probably do it repeatedly. There are likely common serialization and de-serialization techniques for moving data into and out of the architectures. Encapsulate these techniques. Reusable adapters will also include a generic dispatcher or mapping table to select the proper target service and invocation method.

Architectures that use different programming paradigms are more difficult to adapt than similar programming paradigms: Architecture adapters between object-oriented languages will usually be much easier to write than an architecture adapter between an object-oriented language and a procedural language. Although the architectural details may appear simple, leave enough time to address design and implementation complexities for the adapters.

Plan for performance concerns: Serialization and de-serialization of data takes time. Be aware that the adapter route can never perform as well as code that is entirely native to an implementation. For example, calling a C function from Java to calculate the first five prime numbers would be wasteful because of the adapter overhead. Calling a C function to locate the $10{,}679^{th}$ prime number would likely pay off in terms of performance.

Architecture adapters should decrease the complexity within a component for accessing functionality in a different architecture: Using an architecture adapter in a component should allow client programmers to program in a single language with method calls that appear as if the programmer is simply calling another class or procedure. Isolating this complexity should make the primary code line easier to read and maintain.

Use prebuilt architecture adapters and platforms, such as Web Services, whenever possible: If you plan, there are few reasons to build your own architecture adapters. The Java Native Interface (JNI) and Web Services are two excellent technologies for allowing Java to utilize other architectures. These layers can be difficult to get right, so try to reuse whenever possible.

Understanding Architecture Adapters in Web Services

To understand how architecture adapters facilitate better and easier programmer practices, you will look at a slightly more complex Web Service scenario than the previous chapter. You will also fully leverage the facilities in Apache Axis

so that you do not have to manually build the architecture adapters. In fact, from here on out, Apache Axis tools allow you to be entirely SOAP ignorant. Instead, you access Web Services from Java method calls using architecture adapters generated from the WSDL interface of a service, as illustrated in Figure 4-4, a sequence diagram based on Figure 4-3.

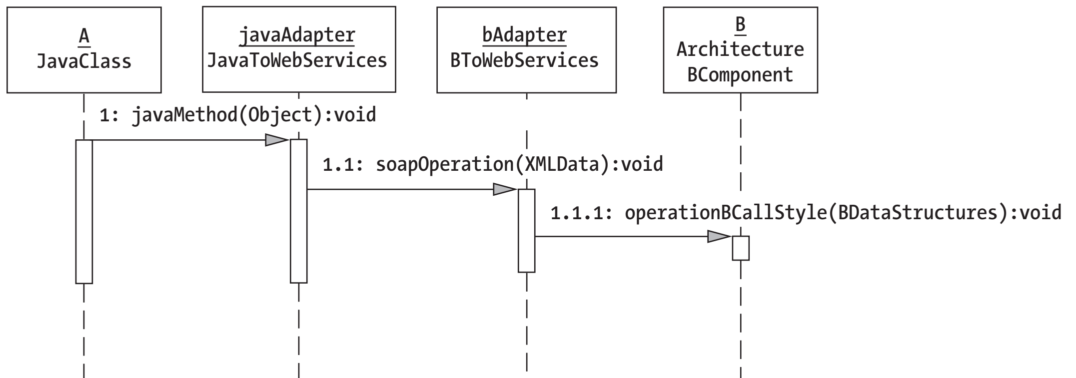

Figure 4-4. Java to Web Service sequence diagram

In Figure 4-4, I took liberties to place the expected Java method call, to the conversion to SOAP, and, finally, to a third, unknown architecture—the mysterious architecture B.

To illustrate the facilities of Apache Axis for building architecture adapters, you use a set of classes that make up the beginnings of a customer database. Figure 4-5 shows the diagram for the primary classes involved in making up a collection of customers. As you can see, a single class, `CustomerCollectionImpl`, contains the customer object query methods. Customer data comes from a `CustomerImpl` class containing basic information, such as their address, and more advanced information, such their primary credit card and Internet address information, which is separated into the `CustomerInformationImpl`.

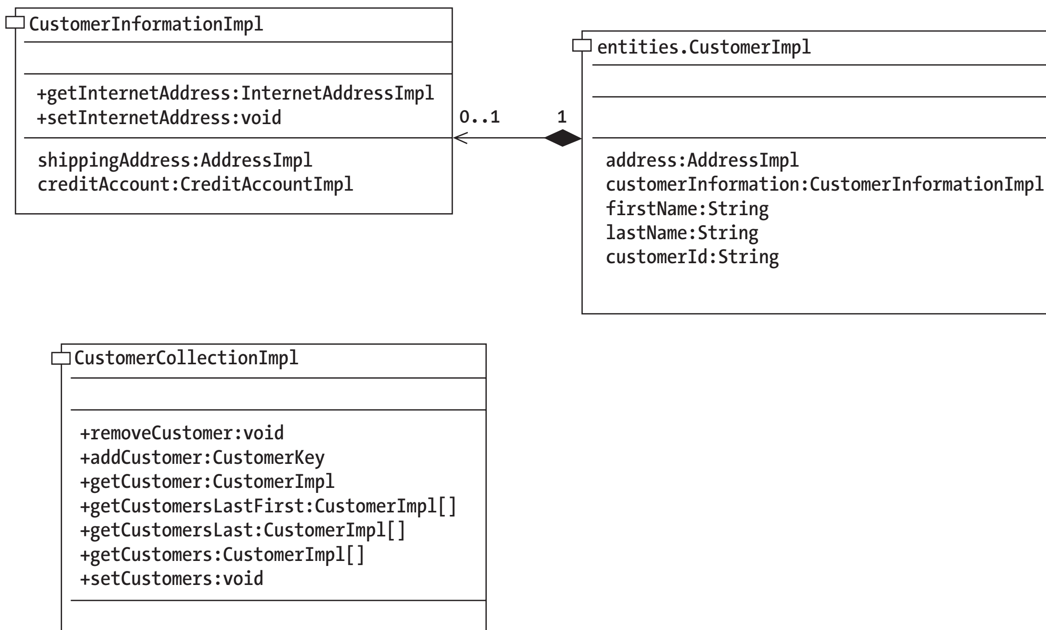

Figure 4-5. Customer collection class diagram

Several important design principles within these classes make the transition to Web Services easier:

- The customer data classes adhere to the JavaBeans contracts.
- There is no operation overloading.
- Methods return arrays where multiple values are possible.

The reasons to apply these principles are to create predictability and avoid exploitation of the object-oriented paradigm. In turn, these things are important to help ease the transition from the rich object-oriented paradigm of Java to the flatter component model of Web Services. Although it is possible to build an architecture adapter that makes the transition between architectures possible, the more you exploit one architecture, the more difficult the adapter will be to write. As tools become more advanced, the level of support for conversion between architectures will grow and the complexity of the classes can increase.

As far as the code goes, the most interesting part of the code is the implementation of the `CustomerCollectionImpl` class. In Figure 4-5, you will notice that

there is no explicit containment of `CustomerImpl` classes from the `CustomerCollectionImpl` class. This technique is a reflection of using Java Data Objects (JDO) for the persistence mechanism underneath your classes. The `CustomerCollectionImpl` class uses a series of queries against a `CustomerImpl` extent rather than explicitly loading the collection and sorting through the objects each time a client requests a single customer.

In terms of architecture adapter responsibilities, Apache Axis uses an inbound/outbound division of labor. One adapter set takes care of Web Services to Java conversions (or to another language), and another adapter takes care of the Java to Web Service conversion. The former takes SOAP messages and converts them to Java method invocations on proper object instances. The latter takes Java method invocations, converts them to SOAP messages, and routes them appropriately.

Creating a Web Services to Java Service Implementation

Apache Axis is, essentially, an implementation of the Architecture Adapter pattern taken to an extreme. Using tools, you can generate an architecture adapter that allows you to call methods in Java and convert them to a Web Service call using SOAP; this is the A architecture adapter from the pattern structure. This SOAP message is received by an Apache Axis implementation running within a Web application server that parses the SOAP message and makes the appropriate method call to the target service. The receiving end is the B architecture adapter from the pattern structure.

These processes form the core of the architecture adapter between SOAP and the target service, or Web Services and the target service, depending on your perspective. Rather than creating a custom adapter for each target service, Apache Axis uses a single generic adapter that leverages plug-in message processing chains and message dispatchers. The entire design is online at the Apache Axis homepage. You are going to view the design from the perspective of a user of Apache Axis.

You deployed a small Web Service in the previous chapter, but it did not have the complexities of the customer collection that you will deploy in this chapter. Recall that the Web Service Deployment Descriptor (WSDD) tells Axis the details about a particular service so that Axis can route messages to the service implementation. This process sets up a generic architecture adapter with specific information about your service. The generic architecture adapter in Axis requires you to give the following:

- The service name (call the service `CustomerCollection`)
- The provider (use the Axis built-in Java RPC provider)
- The class name of the target service implementation (`com.serviceroundry.books.webservices.entities.CustomerCollectionImpl`)
- The methods to turn into service targets (expose all methods on the `CustomerCollectionImpl` class by using *)
- Complex data types necessary for a user to access the exposed service

The only significant difference from the previous chapter is the addition of tags to identify the complex data types. Listing 4-1 illustrates a `beanMapping` tag to identify the `AddressImpl` JavaBean as a complex data type. Recall from Figure 4-5 that the `CustomerImpl` class contains a reference to a customer's address. Axis requires the bean mapping tag to have additional data for the Web Service architecture that does not exist in the architecture supported by Java. You must add information about the namespace that the address resides in—which is a similar concept to packages—but not enough to allow the engine to use the package name as is. You also must tell the deployment tool the language that the complex data type uses—in this case, Java. Finally, you could give the Axis engine information about special serializers (Java classes adhering to an Axis class interface) that you write to help move a bean or class from one architecture to the other. In this case, you use the basic JavaBean patterns, and there is not special handling for your classes, so a serializer is unnecessary. In fact, you will not use serializers throughout this entire book.

Listing 4-1. WSDD File for `CustomerCollectionImpl`

```
<service name="CustomerCollection" provider="java:RPC">
// . . .
<beanMapping qname="myNS:Address"
xmlns:myNS="urn:CustomerCollection"
languageSpecificType=
"java:com.servicefoundry.books.webservices.entities.AddressImpl"/>
// . . .
</service>
```

Using the Apache Axis administration tool (as shown in the previous chapter), submit the file to Apache Axis for proper configuring of the Axis server-side engine, otherwise known as an architecture adapter.

Consuming Web Services with Apache Axis

Without architecture adapters, consuming Web Services would be a difficult and tedious job. Most likely, you would end up writing the architecture adapters yourself as you learned the patterns that your language uses to build SOAP messages and submit them to the Web Service. Fortunately, you do not have to build your own. WSDL's Extensible Markup Language (XML)-based representation and strict definition allows tool vendors to write language-specific tools that build the architecture adapter. The tools convert WSDL into interface and code to turn the language-specific request into a SOAP request to a specific Web Service.

Apache Axis comes with WSDL2Java, a tool that converts a WSDL to a Java interface and architecture adapter implementation; a programmer's job in consuming Web Services just got a lot easier. Instead of constructing SOAP messages, covered in the previous chapter, a consumer manipulates Java classes and objects directly, which then interact with the Web Service environment. Listing 4-2 shows how to create a customer object from a Java program that accesses the Web Service deployed in the previous section. Access occurs through an architecture adapter built using WSDL2Java.

Listing 4-2. Customer Creation Through the Client-Side Architecture Adapter

```
CustomerCollectionImplService service = new
CustomerCollectionImplServiceLocator();
CustomerCollectionImpl port = service.getCustomerCollection();

Address ai = new Address();
ai.setAddressLine1("Web Service Line 1");
ai.setAddressLine2("Web Service Line 2");
ai.setCity("Highlands Ranch");
ai.setState("CO");
ai.setZipCode("80129");
Customer ci = new Customer();
ci.setAddress(ai);
ci.setFirstName("Paul (Web)");
ci.setLastName("Monday");
port.addCustomer(ci);
```

Out of the code, the only slightly abnormal calls are the first two lines of code. These lines retrieve the customer collection service offered previously in this chapter. Listing 4-2 is far easier than constructing a SOAP document and submitting it to the customer collection Web Service, and it is much more natural for a Java programmer to manage. In this case, I put the cart before the

horse; I decided to illustrate how you use the architecture adapter before showing you how to build the architecture adapter.

Constructing the architecture adapter for a particular Web Service with Apache Axis is as simple as obtaining a WSDL representation of the Web Service and running a tool against the WSDL. It is important to realize that although your service implementation is in Java, the WSDL does not expose any syntax or features that are unique to Java; instead, the WSDL is a pure Web Service construct.

The complete WSDL for the `CustomerCollectionImpl` is too large to show in this chapter, but Figure 4-6 is a graphical depiction of the WSDL file contents at a high level of abstraction. In it, you should see that the customer collection interface, detailed in Figure 4-6, forms the Web Service itself with the customer data described as a series of data type definitions.

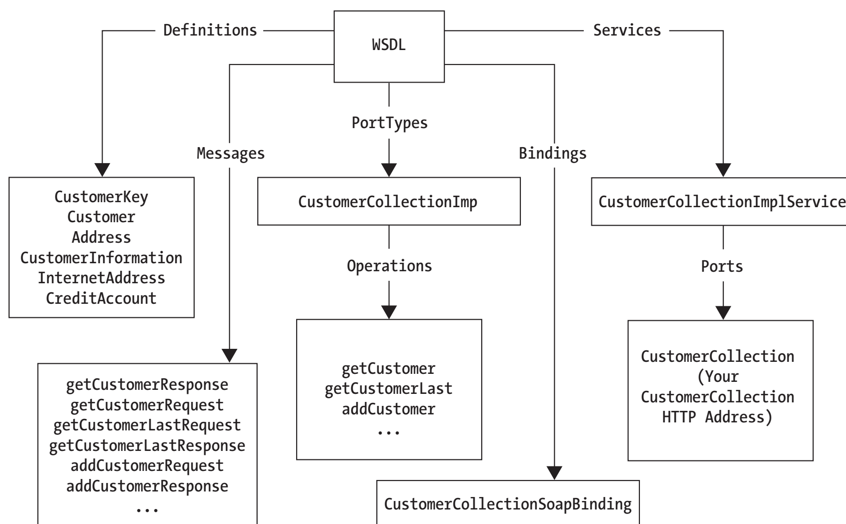

Figure 4-6. WSDL high-level depiction of a customer collection

The data type definitions from the WSDL turn into JavaBeans for use in conjunction with the Web Service. Your client may construct the JavaBeans to pass as parameters to the customer collection Web Service, or the JavaBeans are data returned from method calls on the Web Service.

The WSDL file necessary to generate the architecture adapter gets generated when you access the Web Service with the `?wsdl` parameter on the end of the Web Service's URL (`http://localhost:8080/axis/services/CustomerCollection?wsdl`). To have Axis generate the Java-side architecture adapter, run the Apache Axis WSDL2Java tool against the `CustomerCollection.wsdl` file. The output from the tool is a set of classes that you import into your application if you want to access the Web Service.

WSDL2Java creates several classes that act as the adapter between Java and the Web Service, as well as any classes that are required to interact with the Web Service, such as parameters and return types. The classes include the following:

- **CustomerCollectionImpl:** An interface representing the port types on the CustomerCollection Web Service.
- **CustomerCollectionImplService:** The service interface for a class factory responsible for creating the architecture adapter for the customer collection.
- **CustomerCollectionImplServiceLocator:** The implementation of the service locator that builds an implementation of the customer collection using the location of the service identified in the WSDL as a default or the location identified as a parameter to the creation method.
- **CustomerCollectionSoapBindingStub:** The `CustomerCollectionSoapBindingStub` is an architecture adapter that the service locator returns to the requesting client. This adapter generates SOAP messages based on the methods and data that are set using the methods in the interface `CustomerCollectionImpl`.
- **Address, Customer, CustomerInformation, CustomerKey, InternetAddress:** These classes are JavaBeans that represent customer data types in the WSDL file. The `CustomerCollectionImpl` class serializes these to SOAP contents when the architecture adapter moves information from the client to Web Services.

Early in the chapter, you saw code that creates a new customer. Operations on individual JavaBeans, such as `Address` and `Customer`, have no effect on the actual values that the Web Service represents; only operations on the retrieved `CustomerCollectionImpl` object instance affect server-side data.

Leveraging Architecture Adapters in the Case Study

The P.T. Monday Coffee Company application uses architecture adapters only when interfacing with Web Services. Further, Axis generates the architecture adapters, and they have no impact on developers in terms of coding responsibilities. On the other hand, the build environment is definitely more complex with the extra generation steps. Direct usage of architecture adapters occurs in three places:

- When service behaviors require interactions with external Web Services
- When you expose service behaviors as Web Services
- When service behaviors must interact with the service directory to publish or locate a service

In the case of interacting with the service directory, you use an off-the-shelf package, UDDI4J, to interact with the service directory. The client-side class

library handles service interactions. It is likely that developers built much of the client-side access library with a tool similar to the Apache Axis WSDL2Java tool.

Generation of the architecture adapters that connect SOAP messages to the service behaviors occurs upon deployment of a service through the administration facilities of Apache Axis, as discussed in this chapter and the previous chapter. Generation of the architecture adapters from WSDL files, for your service behaviors to access outside Web Services, occurs using the WSDL4Java tool at build time for your application.

Identifying Important Classes and Files in the Case Study

Table 4-1 shows the primary code discussed in this chapter that you should browse in the downloaded source code.

Table 4-1. Sample Location

FILE	LOCATION	DESCRIPTION
CustomerCollectionImpl.java	src\com\servicefoundry\books\webservices\entities	The primary Java class that becomes a Web Service by using the Apache Axis deployment mechanisms. This class uses JDO as a persistence mechanism.
CustomerImpl.java	src\com\servicefoundry\books\webservices\entities	A class whose object instances represent customers in your application. In this chapter, the customer implementation is not a Web Service; however, the customer collection implementation does serialize instances of customers to your client application.
CustomerInformationImpl.java	src\com\servicefoundry\books\webservices\entities	Additional information about customers, such as credit card numbers. Like the customer, this class is not a Web Service, though the customer collection ends up returning instances of the customer information indirectly through the customer instances.
TestCustomerCollectionWebService	src\com\servicefoundry\books\webservices\tests	A client-side test program that uses the client-side architecture adapter to access the customer collection Web Service.

Using Ant Targets to Run the Case Study

Table 4-2 describes the targets to run for the ant environment to see the programs and chapter samples in operation. Before running any samples, be sure you read and perform all of the install steps in Appendix A.

Table 4-2. Ant Targets

TARGET	DESCRIPTION
`testcustomercollectionwebservice`	This runs the `TestCustomerCollectionWebService` program. You must have deployed the Web Services according to the instructions in Appendix A.

Summary

The architecture adapter is a powerful pattern that allows you to treat two architectures and the communication between them as a single component. You are able to place specific responsibilities and expectations on a construct, the architecture adapter, that allows designers to concentrate on how to best facilitate the mediation between two different architecture styles.

Web Services with Java use architecture adapters in two locations: from the client Java program to SOAP and from SOAP to the Java service implementation. Of course, you need to keep in mind that once the Java service implementation is a Web Service, there is no requirement that you use the service from Java. Architecture adapters for C#, COBOL, or any other language are possible with the communication facilitated through the third architectural style embodied in Web Services.

You saw the design of the architecture adapters built into Apache Axis for offering Web Services, as well as the architecture adapters that Apache Axis' WSDL2Java tool builds for Java clients of Web Services. The client-side and server-side adapters are similar in design and serve reverse needs, one adapting Java to Web Services, the other adapting Web Services to Java. You deployed and used services through the architecture adapters to show the simplicity that isolation of architecture conversions to a single pattern brings to the programming environments.

At this point, you should have an adequate understanding of the mechanisms you will use to expose application functionality to the outside world. You have not spent considerable time on the service directory that your partners use to locate your services. In the next chapter, you will learn about service directories and the patterns inherent in them. Chapter 5, "Exploring the Service Directory Pattern," wraps up the discussion of Web Service foundation patterns, so you can move on to designing specific entities and constructs for use with Web Services.

Related Patterns

The Architecture Adapter pattern relates to all of the example code in this chapter. This is simply because the client code is all written in pure Java using the client-side architecture adapters generated by Apache Axis. Pattern-wise, the Architecture Adapter pattern is most closely related to the following pattern:

- **Service-Oriented Architecture:** The service-oriented architecture uses architecture adapters to help with implementation transparency, one of the primary characteristics of the architecture.

Additional Reading

- Apache Axis Architecture Guide: `http://xml.apache.org/axis/`
- Gamma, Erich et. al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.