CHAPTER 8

Application Architecture and Design

PREVIOUS chapters in this book described different design considerations and motivations for Web services. They also described the various J2EE technologies used for implementing Web services and showed how developers might apply these technologies in an application. Where possible, the chapters offered guidelines for good design and highlighted the advantages and disadvantages among the technologies.

In this chapter, we illustrate how to apply these guidelines to the design and implementation of a real Web service application, the adventure builder enterprise. When architecting and designing Web service applications, you are faced with the significant challenge of constructing the various application modules so that they work together smoothly. We explain the motivational factors and issues that need to be considered, and make these issues concrete by showing how we came to the decisions we eventually made as we architected the adventure builder application. Through this examination, we hope to make it easier for you to determine how best to architect and design your own Web service applications.

8.1 Overview of Adventure Builder

We begin with an examination of the adventure builder application from a high-level, business perspective. Once we have outlined the application's functions, we shift to examining the architecture and design of the application itself.

Recall that the adventure builder enterprise provides customers with a catalog of adventure packages, accommodations, and transportation options. From a browser, customers select from these options to build a vacation, such as an Adventure on Mt Kilimanjaro. Building a vacation includes selecting accommodations, mode of transport, and adventure activities, such as mountaineering, mountain biking to a hidden waterfall, and hiking the mountain. After assembling the vacation package, the customer clicks the submit order option. The customer Web site builds a purchase order and sends it to the order processing center (OPC). The order processing center, which is responsible for fulfilling the order, interacts with its internal departments and the external partners and suppliers to complete the order.

In essence, the adventure builder enterprise consists of a front-end customer Web site, which provides a face to its customers, and a back-end order processing center, which handles the order fulfillment processing. See Figure 8.1.

As you can see from this diagram, there are four types of participants in adventure builder's business process:

1. **Customers**—Customers of the adventure builder Web site shop and place orders for vacation packages. They expect to have certain services available to them, such as the ability to track orders and receive e-mails about order status. Customers are expected to be browser based.
2. **Customer Web site**—The customer Web site provides the Web pages that let customers shop and place orders. The Web site communicates with the order processing center to submit and track purchase orders.
3. **Order processing center**—The order processing center (OPC) is at the heart of the adventure builder enterprise, responsible for coordinating all activities necessary to fulfil a customer order. It interacts with:
   - The customer Web site to handle order-related matters
   - Customers, sending them e-mail notifications regarding orders
   - A credit card service to collect payment
   - Suppliers to fulfill the items in an order
   - Internal adventure builder departments to support and manage the order fulfillment process
4. **External partners**—External partners, such as airlines, hotels, and activity providers, supply the services or components of a vacation. Other partners, such as a bank or credit card company, collect payments for the enterprise.

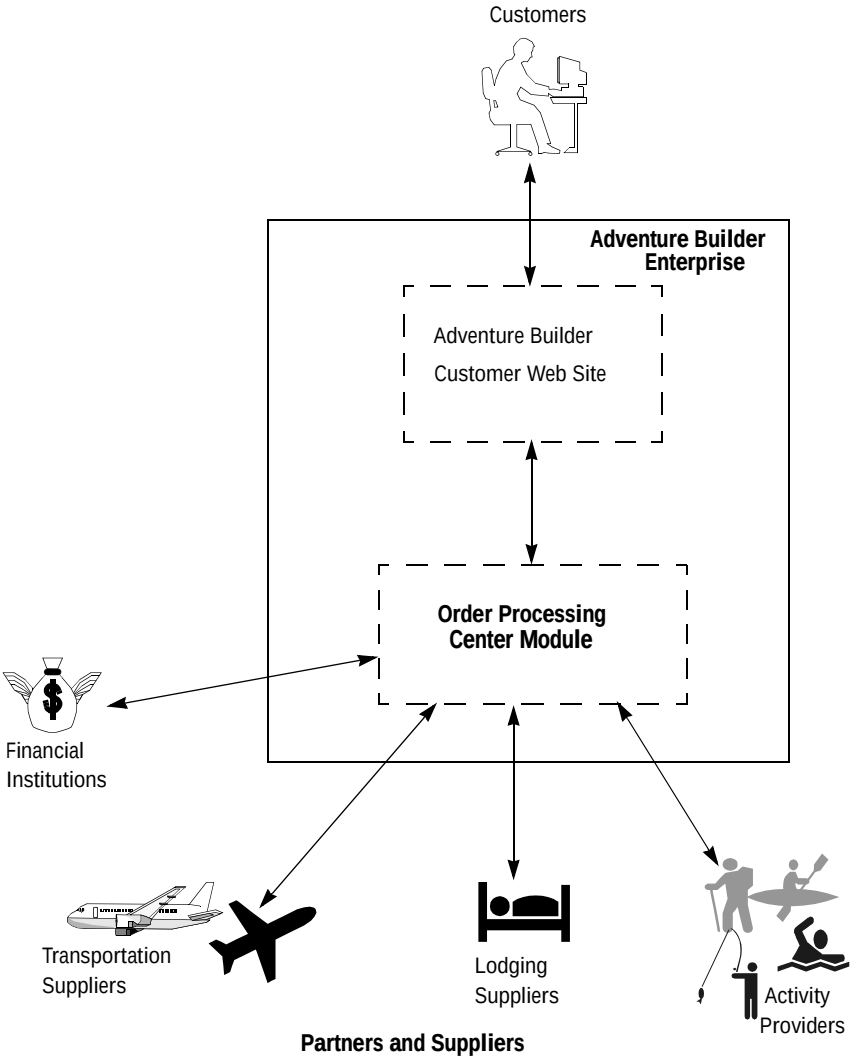

Figure 8.1 Adventure Builder Enterprise and Its Environment

The business problem for adventure builder is to ensure that these participants interact successfully so that it can sell and fulfill adventure packages. To solve this problem, the enterprise must architect, design, and build appropriate J2EE applications that provide the needed business functionality and tie the application modules together. Since the order processing center is the core module of the

application, let's look at it in more detail, starting with its responsibilities for coordinating and communicating with other business units to fulfill orders. (Since this book is about Web services, we do not cover the design of the customer Web site in detail. Other books, particularly *Designing Enterprise Applications with the J2EE Platform, Second Edition*, address this area. See "References and Resources" on page xx.)

The order processing center module needs to perform the following functions:

- Receive customer orders from the customer Web site and process these orders
- Coordinate activities according to the business workflow rules
- Track an order's progress through the steps of the order fulfillment process
- Manage customer relations, including tracking customer preferences and updating customers on the status of an order. This includes sending formatted e-mails to customers about order status.
- Manage financial information, including verifying and obtaining approval for payment
- Interact with business partners—airlines, hotels, and adventure or activity providers—to fulfill a customer's adventure package.
- Provide and maintain a catalog of adventures and allow customers to place orders. The order processing center catalog manager needs to interact with external suppliers and also keep its customer offerings up to date.

8.1.1 Order Processing Center Sub-Modules and Interactions

First, let's examine the order processing center by decomposing it into its logical sub-modules. Figure 8.2 shows the main sub-modules of the order processing center and their relationships to the other participants.

- Order Receiver—Accepts purchase orders from the customer Web site. This sub-module starts the order fulfillment processing in the back end of the enterprise. Each order is identified by a unique ID.
- Workflow Manager—Enforces the process flow rules within the adventure builder application and tracks the state of each order during its processing. The workflow manager interacts with the internal departments and coordinates all the participants in the business process.

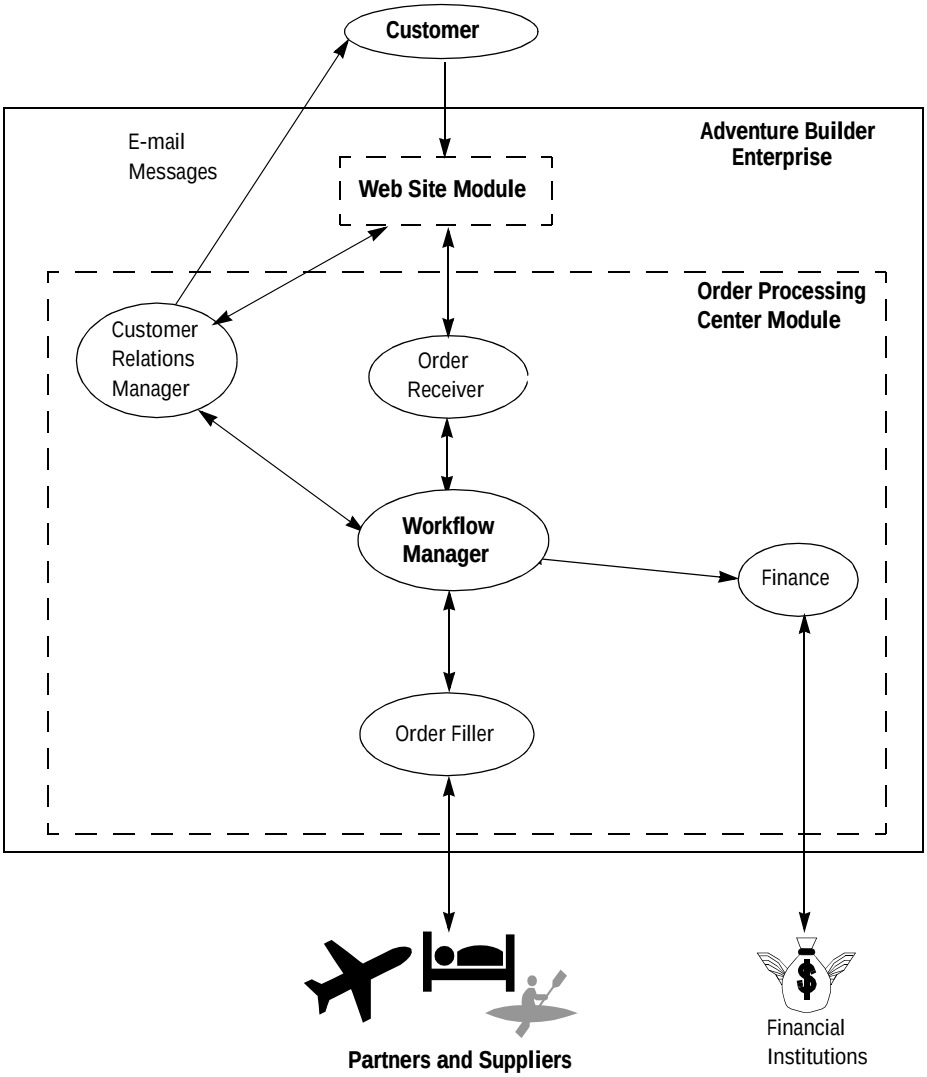

Figure 8.2 Order Processing Center Sub-Modules

- Finance—Interacts with external banks and credit card services that collect payments, and manages financial information.
- Customer Relations Manager (CRM)—Provides order tracking information to the customer Web site application. This sub-module also sends formatted e-mail notices about order status directly to clients.

- Order Filler—Exchanges messages with the various external suppliers to fulfill purchase orders. Messages include supplier purchase orders and invoices.

Keeping the picture of the various sub-modules in mind, let's map out the processing flow that occurs when a customer places an order. In particular, it is important to trace the messages and documents exchanged among participants.

- A purchase order flows from the customer Web site to the order processing center.
- The finance department and the credit card services exchange credit card payment requests and verifications.
- The order processing center order filler and the external suppliers exchange supplier purchase orders, invoices, and other documents.
- The workflow manager and other departments or sub-modules within the order processing center exchange internal messages.

Figure 8.3 shows the order fulfillment workflow. When it receives an order for an adventure package from a customer, the order processing center persists a purchase order. Before proceeding, it verifies that the customer has the available funds or credit for the purchase. If not, the order processing center cancels the order, notifies the customer, and updates the order status. Otherwise, it proceeds to fulfill the order, which entails breaking the entire adventure package order into sub-orders (such as a sub-order for a hotel room, another sub-order for airline reservations, and so forth). It sends the sub-orders to the appropriate suppliers, who fulfill these portions of the order and return invoices. The order processing center then joins these invoices for sub-orders so that the entire order is invoiced. The order processing center also updates the status of the order and notifies the customer.

8.2 Order Processing Center Architecture and Design

We start the examination of the order processing center architecture and design with the problem statement. We then look at the key design choices that were made to solve the problem.

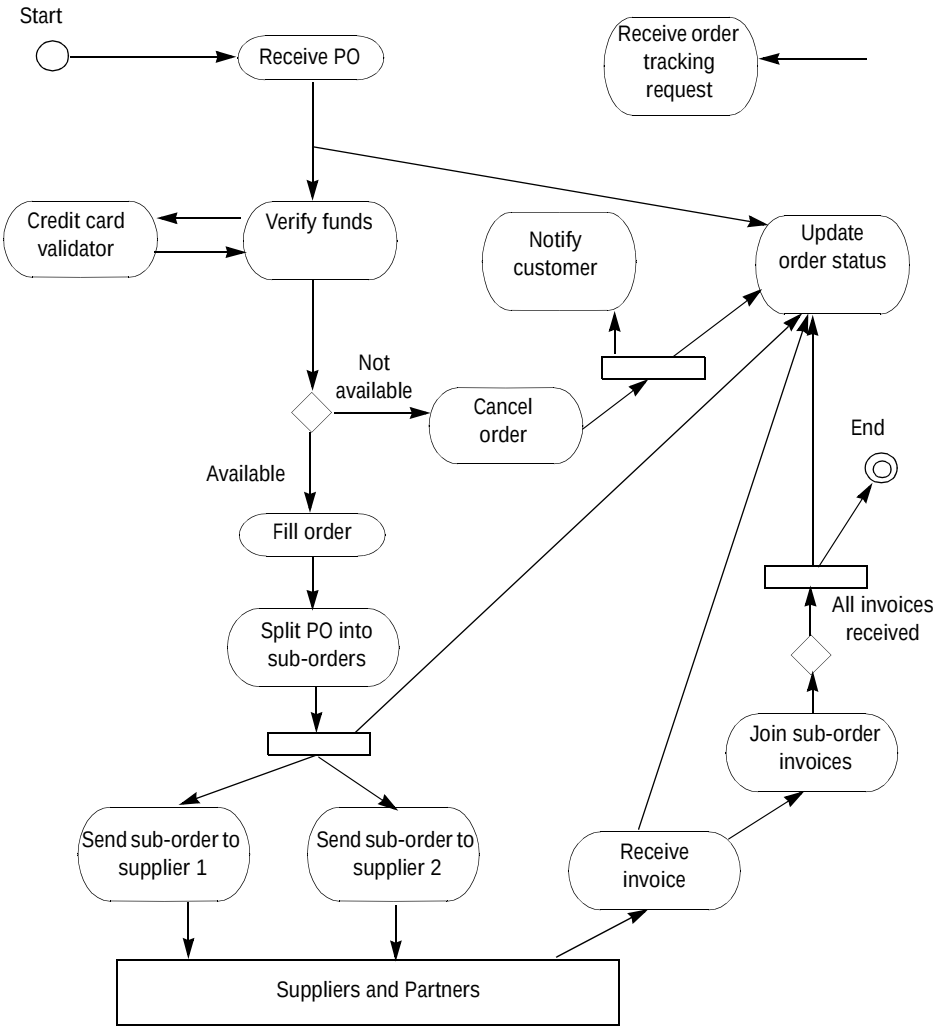

Figure 8.3 Order Fulfillment Workflow

8.2.1 Web Service Interaction and Message Exchange

The key architectural problem that the adventure builder enterprise must solve is the communication and interaction among different entities or applications, both internal and external entities. (Refer to Figure 8.2 on page 343, which shows the lines of communication between the application entities.) These entities and applications must communicate with each other, often exchanging messages such of documents

of various types across numerous boundaries, and work together in a coordinated fashion to solve particular business problems.

Let's take a closer look at the message exchanges that occur during the order fulfillment process, starting with the communication between the customer and the customer Web site, and then from that Web site to the order processing center. Such communication is fairly typical of most Web services-based applications.

- Clients or customers communicate with the customer Web site via a Web browser.
- Once a customer places an order, the customer Web site communicates a purchase order to the order processing center.
- The customer Web site also follows up with the order processing center to ascertain the current status of an order.
- The order processing center sends credit card processing details for verification to the credit card service.
- The order processing center sends purchase orders to suppliers and receives invoices from these suppliers.

Within the order processing center module, different application entities handle such functions as receiving, tracking, and processing orders. These entities need to interact with each other to coordinate the processing of an order. Other application entities handle requests for order status.

Since you are building an application that enables different entities to communicate and interact to accomplish a set of tasks, you need to make design choices that foster this interaction. These choices center around communication technologies and message formats.

8.2.1.1 Communication Technologies

You must decide on the best communication technology for your application. Recall from "Communication" on page 23 that, when developing on the J2EE platform, you can use such communication technologies as JMS, RMI/IIOP, Web services, and so forth. Each technology is appropriate for different circumstances. For each interaction among the order processing center entities, we chose a particular communication technology.

For adventure builder, we chose to use Web services as the technology for communication between the order processing center and entities external to the order processing center, such as for the exchange of purchase orders and invoices with external partners and suppliers. We made this choice because we are primarily interested in achieving the greatest degree of interoperability among all modules and among all types of clients.

It is important for adventure builder to be able to integrate its functionality with that of its numerous partners, since they actually provide the services adventure builder offers to customers. Web services with its emphasis on interoperability gives these partners the greatest degree of flexibility in the technologies that they use. Web services also provide a good interface for separating and decoupling software systems. This is important for adventure builder since it is building on existing systems and modules developed and owned by different departments within the enterprise.

- For communication between the order processing center and entities external to the order processing center, we chose to use Web services.

Within the order processing center module, the communication among sub-modules primarily uses JMS. Since the order processing center controls the entire environment within the module, communication can be more tightly coupled. Most of the communication is between the order processing center workflow manager and the various department sub-modules, and all sub-modules are within this environment. Given the control over the environment and that most communication is asynchronous, using JMS is appropriate. Also, since there is additional overhead to using Web services, an application should use it only when it requires the benefits of Web services. For the communication among the entities within the order processing center, the overhead of Web services outweigh its benefits.

- We chose to use JMS for communications among entities within the order processing center.

Generally, Web services are a good choice for communication external to the enterprise, but they can also be applied to internal situations. The order processing center and customer Web sites are within the same enterprise, albeit different departments, and they use Web services rather than JMS. The order processing center module makes its services available as a Web service to the customer Web site—that is, the Web site uses the order processing center's Web service to fulfill

an order. There are advantages to this implementation. For one, the Web site for the order processing center module may be hosted outside the firewall in a demilitarized zone (DMZ), even though the order processing center module itself is always inside the firewall. The Web site may conveniently use the HTTP port available on the firewall to communicate with the order processing center. Also, Web services allow both client and server to be on different hardware and software platforms. This makes Web services a natural choice, since the adventure builder Web site may be hosted on a different hardware platform (and software platform) from the order processing center module. Developers may want this platform flexibility since the Web site is on the Internet and may need to scale to handle very large loads. Furthermore, the Web site must be responsive to customers, whereas the order processing center module, since it works asynchronously, is concerned with achieving high throughput.

8.2.1.2 Message Format

The adventure builder business problem involves the exchange of many different types of messages, whose payloads contain documents such as purchase orders and invoices from different suppliers. Some of these message payloads adhere to an internal format and others follow standard formats. Since the system must handle multiple message types, we use XML as the message payload format.

While we can pass information either as XML documents or as Java objects, the choice of communication technology has implications for how information is passed through the system. Primarily, we use XML documents for communication on the edges on the enterprise, especially for messages exchanged with trading partners.

But since we also use JMS internally within the order processing center, we had to choose to either pass XML documents between sub-modules or to convert documents to Java objects. Since we are modelling a document-oriented system, and since the sub-modules represent different departments, we decided to pass XML documents among them. By passing information in this format, internally each department has control over binding to the particular Java objects they need within their module. Had we bound from XML documents to Java objects at the edges of the enterprise, then this sub-module independence would not be the case. Passing XML documents also plays well for our particular architecture, which has a centralized workflow coordinator or manager that keeps the individual sub-modules and departments from directly communicating. In short, this keeps our department sub-modules more loosely coupled.

- We chose XML to be the message format for communication with internal as well as external entities.

8.2.2 Communication Architecture

Figure 8.4 shows the architecture of the communication infrastructure for adventure builder.

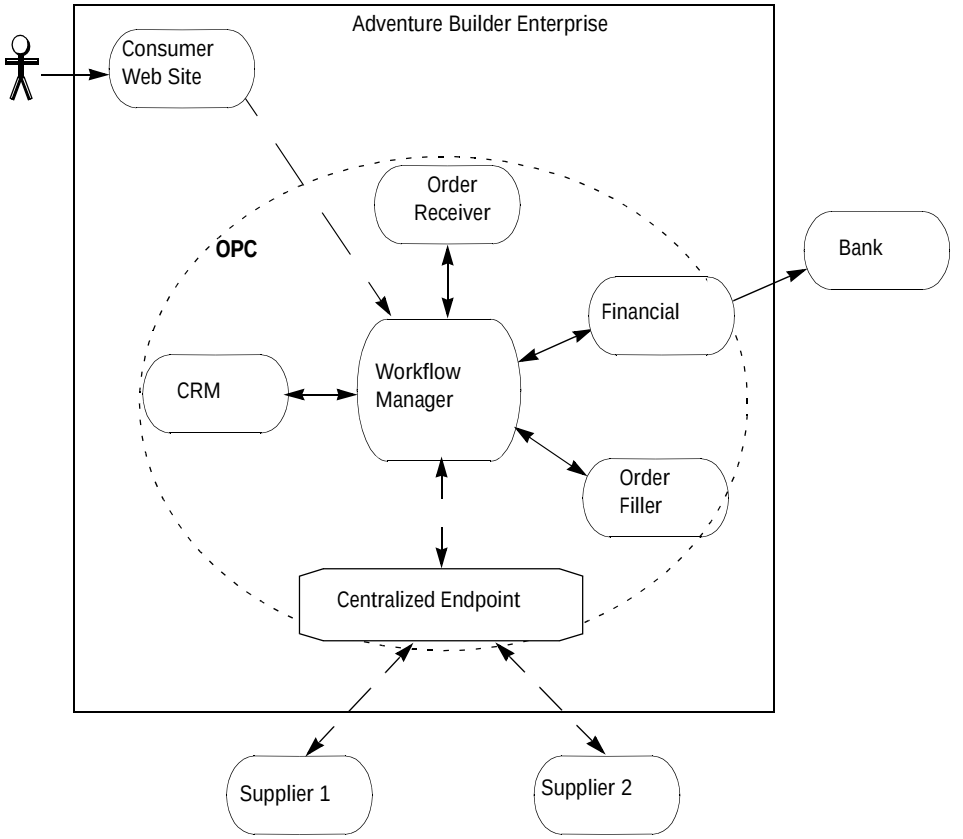

Figure 8.4 Adventure Builder Communication Structure

Note the following about the architecture:

- The order processing center uses a workflow manager, which contains all process rules and flow logic, to coordinate processing of the orders. The workflow manager also keeps track of the current state of each order while it is processed.
- Each participant knows its business functionality and how to execute its own step in the workflow; it does not need to know other steps in the process.
- Generally, each order processing center participant (such as the order filler) has a JMS resource and receives XML documents. The JMS resource is the integration point between the manager and each work module.
- A centralized endpoint manages the interactions between the suppliers and the order processing center. Rather than having a separate Web service endpoint for each message, all interactions between the order processing center and suppliers are grouped together and handled by a common endpoint and interaction layer.
- Other parts of the order processing center may act directly as clients to Web services without using the centralized endpoint. For example, the finance module uses the credit card service.

The order processing center internally uses a hub-and-spoke model where the workflow manager coordinates the participants in the order fulfillment process. Each participant receives an XML document from the workflow manager, processes the XML document by applying its portion of the business logic, and returns the result to the workflow manager. The workflow manager determines the next step to execute in the workflow and dispatches the appropriate request, by sending an XML document, to the proper participant. Each individual participant executes their part of the business functionality and has no need to know the overall process of the workflow.

The Web service endpoints, used on the edges of the order processing center for interactions between the order processing center and its external entities, have different considerations. You should consider each endpoint design individually, based on its particular interactions. Sometimes, to make communication more manageable, it's possible to group together sets of interactions into a simplified structure. See the next section for more details about endpoint considerations and choices. See "Web Service Communication Patterns" on page 358 for a discussion

of more complex Web service interactions, as well as how to provide structure to Web service communications.

8.3 Endpoint Design Issues

The adventure builder developers considered a set of issues when designing a Web service. These issues encompassed such areas as the development approach for the Web service interface, the type of endpoint to use, the granularity of the service, the types of parameters to pass to the interface, the structure of the service's interaction and business processing layers, delegating to the business logic, the types of clients accessing the service endpoint, and deployment issues.

Earlier chapters addressed many of these issues. Here we try to show how such decisions are made in a real-world, comprehensive application. Note that this chapter bases its discussion using three of adventure builder's Web service interfaces:

1. The Web service interface between the order processing center and the customer Web site to receive customer orders
2. The order tracking Web service, which handles client requests for order status
3. The Web service interface between the order processing center and the external partners or suppliers

8.3.1 Web Service Interface Development Approach

When you implement a Web service, one of the first decisions you must make is the approach to use for developing the service interfaces. We followed the guidelines suggested in Chapter 3, particularly those in "Designing the Interface" on page 66. Let's look at the Web service interactions between the customer Web site and the order processing center, and also between the order processing center and the suppliers. Different issues pertain to the design of these service interfaces.

Because the same organization develops and controls the customer Web site and the order processing center, it is easy to propagate to the Web site module changes to the Web service interface that the order processing center provides to the Web site. Since we have control over all parties that have access to this Web service interface, the issue of stability of the interface is less important. Because it is easier, we opted to use the Java-to-WSDL approach to design and implement the Web service interfaces between the Web site and the order processing center.

The Web service interaction between the order processing center and the suppliers is a business-to-business interaction. Any order processing center changes to the interface affect many different suppliers. A stable interface also enables the participants to evolve in different ways. It is of paramount importance with such interactions that the interface between the communicating entities be stable. Since legal issues (see "Handling XML Documents in a Web Service" on page 105 for information about legal issues) may be important with business-to-business interactions, exchanged business documents should have an agreed-upon format and content, and for this it is best to use documents with existing standard schemas. For these reasons we used the WSDL-to-Java approach for the Web service interface between the order processing center and its suppliers.

- Although somewhat more complex, the WSDL-to-Java approach satisfies the overriding need for a stable Web service interface.
- To ensure that the WSDL-to-Java approach did not break interoperability, use various WSDL and schema editing tools to ensure that the WSDL and other artifacts comply with WS-I standards.

8.3.2 Endpoint Type Considerations

After you settle on the service interface development approach, you need to choose the type of endpoint for the interface. Recall from "Choice of the Interface Endpoint Type" on page 67 that you may implement a Web service endpoint either as a JAX-RPC service endpoint on a Web tier or as an EJB service endpoint on an EJB tier. The choice of endpoint type is primarily based on whether or not the business logic uses enterprise beans. If the business logic does use enterprise beans, it is often convenient to use an EJB service endpoint. If the application is primarily Web tier based, then it is best to use a JAX-RPC service endpoint.

The order processing center module exposes all of the Web services in the adventure builder application. Since the order processing center is implemented with a set of enterprise beans, it makes sense to implement these Web services as EJB service endpoints. Using a JAX-RPC service endpoint would introduce a Web layer that acts merely as a proxy, directing requests from the Web tier to the EJB tier and adding no value.

Using EJB service endpoints has some additional minor advantages as well. You can declare an EJB method to be transactional, resulting in the method body executing within a transaction. We found this useful since the order processing center requires transactional access to the database. An EJB service endpoint also

allows method-based security controls, which is useful if you want certain methods to be publicly accessible but other methods to be restricted to authenticated users. You need to do additional work to get the equivalent capabilities in a JAX-RPC service endpoint.

8.3.3 Granularity of Service

It is important to design the service interface with the proper level of granularity. Because we want our application to perform well, we use coarse-grained interfaces for the Web services. When adding a Web service interface to existing applications, you want to identify the Web service interfaces that can be exposed. To do so, it is useful to look at the session bean-based application façades. Generally, it is not a good idea to convert each such application façade session bean into a Web service interface. You should design Web services to be more coarse grained than individual session beans. A good way to achieve a coarse-grained interface is to design the Web service around the documents it handles, since documents are naturally coarse grained. The adventure builder application applies these principles by exposing Web services that primarily exchange well-defined documents—purchase orders and invoices. These Web services may call multiple session bean methods to implement the services' business logic.

8.3.4 Passing Parameters as Documents or Java Objects

As already noted, a client invokes a service's functionality by calling the appropriate method on the service interface and passing the expected parameters. Developers must decide on the type of parameters passed to the Web service interfaces, and the parameter types determine the style of the interface. Parameters may be passed as either JAX-RPC value types or XML documents, which are represented as `SOAPElement` objects in the service interface. See "Parameter Types for Web Service Operations" on page 72 and "Exchanging XML Documents" on page 107 for detailed discussions on these choices. Note that since on the wire everything is in XML, the JAX-RPC value types are automatically mapped to their equivalent XML representations.

Passing XML documents raises different issues than passing Java objects. XML documents involve a certain amount of complexity, since the requestor must build the document containing the request and the receiver must validate and parse the document before processing the request. However, XML documents are better for addressing the business-to-business transaction considerations.

We used object parameters for the service interfaces between the customer Web site and the order processing center applications. Since both are contained within one enterprise, these services do not encompass business-to-business functionality. Moreover, the parameters map to specific document types that can be mapped easily to their equivalent JAX-RPC value types. Using these JAX-RPC value types eliminates the complexity of creating and manipulating XML documents.

Code Example 8.1 is an example of a Java interface for the order tracking service. It has one method, `getOrderDetails`, which expects a `String` argument and returns a Java `Object`, `OrderDetails`. A client submits an order identifier to retrieve information about an order, which is much simpler than having the client submit an XML document containing the same information.

```
public interface OrderTrackingIntf extends Remote {
    public OrderDetails getOrderDetails(String orderId)
                throws OrderNotFoundException, RemoteException;
}
```

Code Example 8.1 Interface Using Java Parameters

Choosing the return value type is not as straightforward. The `getOrderDetails` method returns specific order details, such as user name and shipping address, rather than the complete purchase order. It is possible to return this information as either a Java `Object` or in an XML document. We chose to return the order details in a `OrderDetails` Java `Object` since the details returned to the client—user identifier, addresses, and so forth—map to a specific schema. Hence, the service interface can create and use the equivalent JAX-RPC value types. The order fulfillment Web service interface, `PurchaseOrderIntf`, is designed similarly. (See Code Example 8.1.)

The order processing center service interface with suppliers exchanges XML documents rather than Java objects. Since suppliers are external to the enterprise, the service interface must be stable. Legal issues require exchanging business documents that conform to well-defined schemas. There are times when ensuring a stable interface is worth more than the simplicity of passing Java objects and the complexity of handling XML in the application code. The service interface may also need to support multiple document types in its methods.

Code Example 8.2 illustrates a JAX-RPC interface generated from the WSDL description where the order processing center acts as a client of a supplier and an XML document passes between them. To fulfill part of the customer's order, the order processing center module sends a purchase order to the supplier's Web service. For example, to fulfill a customer's travel request, the order processing center sends an order contained in an XML document to the airline supplier's Web service.

```
public interface AirlineReservationIntf extends Remote {
    public String submitDocument(SOAPElement reservationRequest)
                                            throws RemoteException;
}
```

Code Example 8.2 Interface Using XML Documents as Parameters

The service has a single method, `submitDocument`, which takes a reservation request document and returns a status. The service receives the request, does some preprocessing—such as validation of the document and security checks—then stores the request in a database for later processing.

Keep in mind that when a Web service method passes different types of XML documents, the WSDL description should use the generic type `anyType` to indicate the type of the method's parameters. Code Example 8.2 shows how `anyType` is mapped to `SOAPElement` in the generated JAX-RPC interface. Because the WSDL does not have the information to describe these documents, a separate schema holds the complete description of the documents. You need to publish the schemas for all documents that are exchanged. Publishing entails making the schemas available to clients at some known URL or registry.

On the Java platform, it is best to send an XML document as a `javax.xml.transform.Source` object. (See the discussion in "Java Objects as Parameters" on page 73 and "XML Documents as Parameters" on page 76.) However, we chose to send the XML document as a `SOAPElement` object because the WS-I Basic Profile does not yet support `Source` objects. (For more information, see "Interoperability" on page 86.)

8.3.5 Layering the Service

Normally, it's best when designing a service to separate the service's interaction layer from the business logic processing layer. This is especially true when an

incoming request uses a different data model than the data model used by the business logic. Recall that a service may divide its processing into layers if it handles requests that require significant preprocessing or when there is extensive mapping between the internal data model and the incoming data.

The order fulfillment and order tracking Web service interfaces, since they use parameters that are objects, have no need for document validation or transformation. For both, there is minimal mapping from one data model to another, since the passed data mirrors for the most part the internal data model. Thus, both services do very minimal preprocessing of requests. As a result, we chose to merge the interaction and processing layers for these services.

Consider the order tracking Web service, which receives an order identifier as input and returns some order details. Since little preprocessing is required, the interaction and processing layers are merged into one layer. Although merged into one layer, the order tracker Web service interface does *not* expose the internal data model. It is important to keep the Web service interface from being tightly coupled to the internal data model, and not exposing the internal data model through the service interface accomplishes this. By avoiding tight coupling, we can change the internal data model without affecting the Web service interface and clients.

Instead, we created an `OrderDetails` Java object to hold the data in the service interface, thus removing the need to change the order tracker interface if the internal data model changes. The client's view of the `OrderDetails` object through the order tracker interface remains the same regardless of changes to the data model. See Figure 8.5.

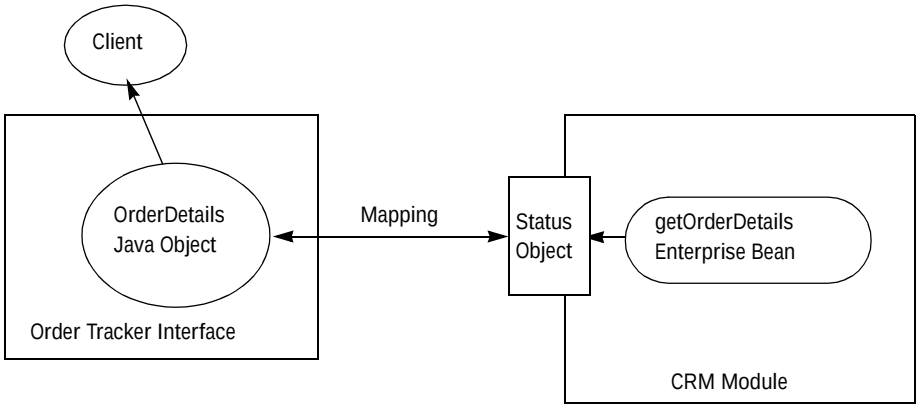

Figure 8.5 Separating CRM and Order Tracker Data Models

The Web service interfaces between the order processing center and the suppliers must validate and transform incoming XML documents to the internal data model before processing the document contents. We chose to do the validation and transformation in the interaction layer of the service before delegating the request to the processing layer.

8.3.6 Delegating to Business Logic

After receiving and preprocessing (if required) a request, the endpoint must next map the request to the appropriate business logic and delegate it for processing. Recall from "Delegating Web Service Requests to Processing Layer" on page 92 that much of this decision hinges on whether the request is processed synchronously or asynchronously.

The order tracking Web service interface uses a synchronous interaction for handling requests—processing is fast and a client waits for the response. Thus, the service interface maps a request to the appropriate business logic and immediately returns the results. adventure builder's Web services for processing the order workflow are asynchronous in nature since their business processing may be time consuming to complete and the client does not wait for the response. For these other Web service interactions—client submitting purchase order to the order processing center, order processing center placing orders with the suppliers, suppliers invoicing the order processing center—the service interfaces pre-process the requests and then delegate the request to the business logic. The interfaces use JMS messages to send the requests to a JMS queue associated with the business logic.

8.3.7 Client Considerations

For a client, the principal architectural consideration is choosing the best JAX-RPC service interface API to use to access the Web service. Clients can use a stub approach, dynamic proxies, or dynamic invocation interfaces (DII). When choosing a particular API, it is important to consider client requirements such as coupling, portability, the ability to dynamically locate and call services at runtime, and ease of development. Refer to Chapter 5 for more details.

Adventure builder's Web service clients use stubs, which is the simplest of the three modes. The stub mode requires the WSDL document along with the JAX-RPC mapping file to generate a set of proxies at development. A client at runtime uses these generated proxies to access a service. The stub mode of proxy genera-

tion provides the client with a tightly coupled view of the Web service, and it is a good choice when the service endpoint interface does not change.

With stubs, adventure builder uses the `javax.xml.rpc.Service` interface `getPort` method to access the Web service. Using this method removes port and protocol binding dependencies on the generated service implementation classes. As a result, it improves the portability of the client code since the client does not have to hard code a generated stub class. See “Locating and Accessing a Service” on page 219.

We also implemented exception handling from the client’s perspective. To do this, we examined the interface generated from the WSDL file to see what possible exceptions the service might throw and then decided how to handle individual exceptions. See “Handling Exceptions” on page 230. The customer Web site, when it catches exceptions thrown by the order tracking service, maps the exceptions to client-specific exceptions. The client application detects these exceptions and redirects the user to an appropriate error page.

8.3.8 Publishing Web Service Details

A Web service endpoint is described by its WSDL file, and clients of the service need access to the WSDL to obtain basic information about the service. One way to disseminate this information to clients is to publish the WSDL files in a registry, if the service is open to the general public. Or, you can arrange a common location for the WSDL file that is known only to selected clients. You also must decide how to expose other details pertinent to the service, such as the business document schemas.

We decided to make the adventure builder Web services available in a well-known location, which clients can obtain from the deployment environment. These Web service interfaces are meant for specific business interactions between specific entities rather than for consumption by the general public. For this reason, we decided against publishing the services in a registry. Similarly, we decided to place the business document schemas at a well-known location, from which intended clients use these schemas.

8.4 Web Service Communication Patterns

Up to now we have concentrated on Web service interactions between two entities. Let’s take a look at the bigger picture and see how Web service interactions operate within a larger business collaboration. Often Web service interactions are part of

larger business collaborations that may involve multiple synchronous and asynchronous Web service interactions. Although Web service interactions within a single business collaboration may be related, current Web service technologies treat these interactions as stateless. As a result, a developer needs a way to associate messages that are part of a single business collaboration. This issue is made more difficult because collaboration sometimes requires that messages be split into multiple sub-messages, where each sub-message is processed by a different entity or partner. Not only does the developer need to relate these sub-messages to the original message, but the developer may also need to join together replies to the sub-messages.

As part of good Web service endpoint design, you need to consider the interactions in which the service is involved. Client-to-service endpoint communication is one such interaction. You may have other, more complex interaction patterns even involving multiple participants. For example, a particular request may take several Web service interactions to complete. This often happens when there are multiple steps in a workflow. Since different services process parts of a request, you need to correlate these interactions and identify individual requests that together make up a larger job.

The following sections present some strategies for handling these Web service communication problems. You can use these strategies separately or in combination, depending on an individual scenario.

8.4.1 Correlating Messages

Sometimes a business collaboration requires multiple Web service calls. In these cases, the developer needs a way to indicate that the messages passed in these Web service calls are related. A developer should use a correlation identifier to keep together messages for related Web service calls.

A correlation identifier is useful for Web services with interactions that are more complex than can be handled by a simple reply. When a Web service must invoke multiple processing steps to handle the request, the service often needs some means to identify the request at a later time. Correlation identifiers provide just such an identity for a request.

For example, a correlation identifier might be useful in these scenarios:

- A client submits an order and later wants to track the order status.
- A request is broken up into multiple subtasks that are asynchronously processed. The request manager aggregates subtasks as they complete, finally marking the entire job complete. Often, many jobs run at the same time, requir-

ing the manager to correlate each job's multiple subtasks. Adventure builder's order processing module illustrates this scenario.

- A Web service interaction may require a higher level of service in terms of handling errors and recovery. If an error occurs, the service may have to track the state of the request.
- A client needs to make several request and reply interactions as part of a conversation, and it needs to identify those calls belonging to the conversation.
- A client may make duplicate submissions of a message, and the service needs to eliminate these duplications.

The correlation identifier must be unique in the participating systems. In addition, you must decide whether the client or the service generates the identifier, and where to generate it. For example, the calling client may generate an identifier, including it with calls to a service. Or, the service may generate the identifier and assign it to the incoming call, and perhaps return the identifier with the response.

The server can also embed useful information into an identifier that it generates, making processing easier. For example, a client often needs confirmation indicating that the server has received the client's message. An identifier generated at the server side can serve as a confirmation identifier. Server-side identifier generation also allows better control by providers, since they can employ their own policies.

The customer Web site collects information from a Web user and puts that information together into an order. The Web site then places the order into adventure builder's system via a Web service call to the order receiver module, passing the order information as a message. The order information includes the client-generated correlation identifier. The order receiver module receives the order message and uses the correlation identifier to eliminate duplicate order submissions.

A correlation identifier is a form of context information that needs to be communicated from the client to the server. For strategies on how to pass the correlation identifier from client to server, see "Passing Context Information on Web Service Calls" on page 366.

❒ Use correlation identifiers for associating groups of related messages.

8.4.2 Splitting and Joining Messages

Often, a service receives a request, which it fulfills as a series of subtasks run in parallel. When all tasks complete, the service gathers the results and checks if the criteria for completion is met. A service can split a message into subtasks, often including a correlation identifier with the individual pieces so that they can be rejoined successfully when they have all completed. Such identifiers may be needed since often subtasks complete out of sequence. A service can also independently use split and join operations.

For example, the order processing center receives a single purchase order from a customer. Figure 8.3 on page 345 shows the workflow for handling a message. The order processing center splits that order into a set of sub-orders: an order for an airline reservation, an order for a hotel room, and orders for all activities. It then sends each sub-order to the suppliers, and they process their tasks, invoicing the order processing center when completed. The order processing center aggregates these supplier invoices to fulfill the order.

How do you implement this split and join pattern? For guidelines on splitting an incoming document, see "Fragmenting Incoming XML Documents" on page 153. The order processing center splits a purchase order document into multiple XML documents by extracting the relevant information for each supplier, then sending that relevant information as a sub-order to the suppliers. It includes a correlation identifier with the sub-order to correlate all submessages generated from a single message.

For the join operation, the order processing center waits for all suppliers to fulfill the sub-orders, then it does some further processing to complete the order. Notification from suppliers may arrive in an arbitrary, nonsequential, order. The order processing center implements the join operation using message-driven beans. When it processes a sub-order, a supplier sends an invoice message to the order processing center through a Web service call. On receiving the invoice message, the order processing center workflow manager checks whether the join condition is met and updates its state accordingly. When all invoices are received, the workflow manager changes the order status to "COMPLETED" and directs the CRM system to notify the user.

Figure 8.6 shows split and join operations on an adventure builder customer purchase order. The customer orders an adventure package, plus hotel and airline reservations. All this information is contained with the purchase order. The order processing center logic splits the purchase order into sub-orders, and sends each sub-order to the appropriate supplier for processing. For example, the order pro-

cessing center sends the activity sub-order to an activity supplier and the airline sub-order to the airline reservation processing. Each supplier, when it completes a sub-order, submits an invoice for its work. The order processing center receives the separate invoices at varying times. It checks to see if the join condition is met and updates the order status accordingly.

- ❒ When splitting messages, it is good to use correlation identifiers to later associate the responses during the join.

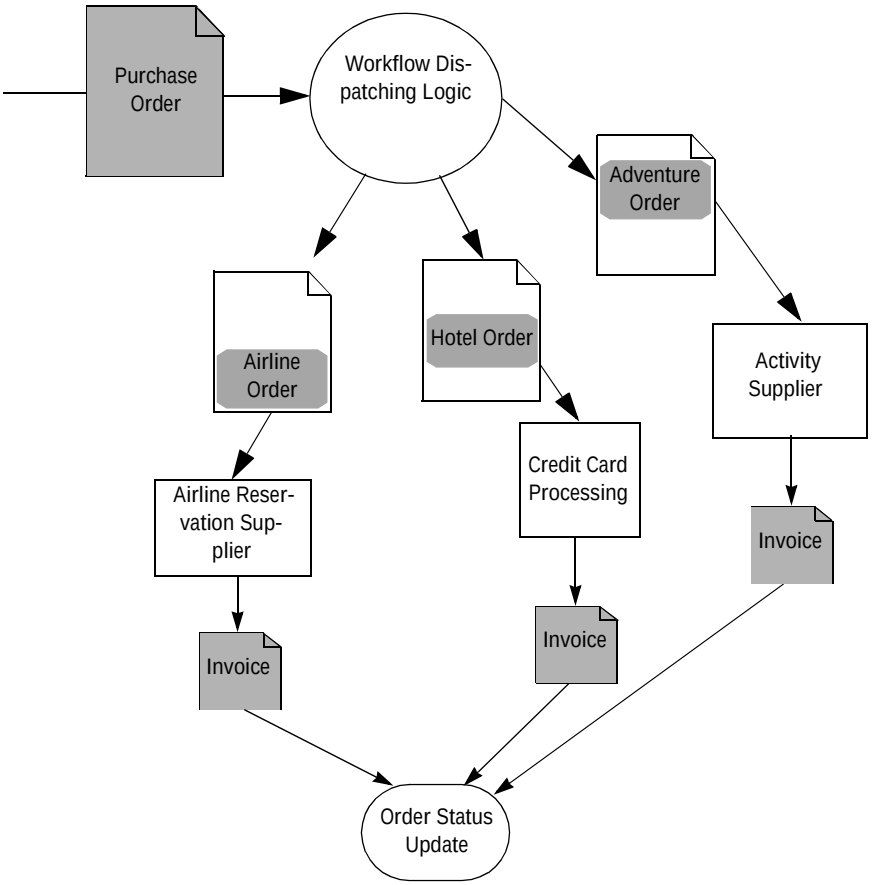

Figure 8.6 Split and Join Operations

8.4.3 Refactoring Synchronous to Asynchronous Interactions

Sometimes your application requires an asynchronous Web service interaction. Web services that use asynchronous interactions tend to be more responsive and scale better. However, Web services provide only a synchronous mode of operation, especially when using the HTTP protocol. In your application you can convert this synchronous interaction to an asynchronous request and reply. You may be able to refactor some of the synchronous interactions of a Web service application to asynchronous interactions.

Figure 8.7 illustrates a synchronous interaction. Notice how the client must suspend its processing until the service completes processing of the request and returns a response.

On the other hand, with the asynchronous interaction shown in Figure 8.8, the client continues its processing without waiting for the service. In both the synchronous and asynchronous communication figures, the vertical lines represent the passage of time, from top to bottom. The vertical rectangular boxes indicate when the entity (client or service) is busy processing the request or waiting for the other entity to complete processing. In Figure 8.8, the half-arrow indicates asynchronous communication and the dashed vertical line indicates that the entity is free to work on other things while a request is being processed.

When a component makes a synchronous call to another component, the calling component must wait for the receiving component to finish its processing. If the calling component instead makes the call asynchronously, the caller can proceed with its own work without waiting for the receiving component to do its job.

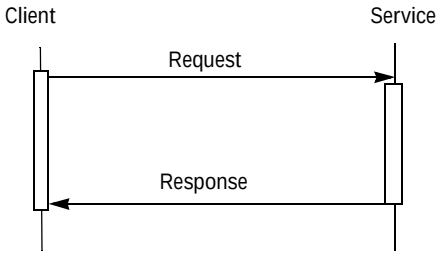

Figure 8.7 Synchronous Communication

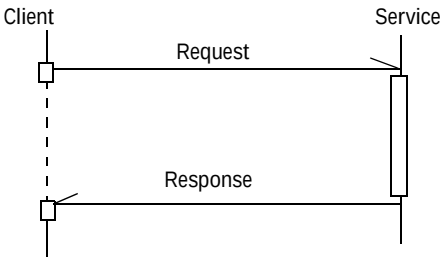

Figure 8.8 Asynchronous Communication

Let's look at what needs to be done to convert a synchronous endpoint to an asynchronous endpoint.

- Change a synchronous request/reply interaction into a request that returns `void` as the immediate reply.
- Change the client code to send the message request and immediately return.
- Have the client establish a reply address for receiving the response and include this reply address in the request. By including a reply address, the service knows where to send the response message.
- A correlation identifier needs to be established between the request message and the service response message. Both the client and the service use this identifier to associate the request and the reply.
- Refactor the server endpoint to accept the request, send it to a JMS queue for processing later, and immediately return an acknowledgement to the client. The service retains the correlation identifier.
- Have the service processing layer use a message-driven bean to process the received request. When a response is ready, the service sends the response to the reply address. In addition to the results, the response message contains the same correlation identifier so that the service requestor can identify the request to which this response relates.
- Have the client accept the response at a Web service that it establishes at the reply address.

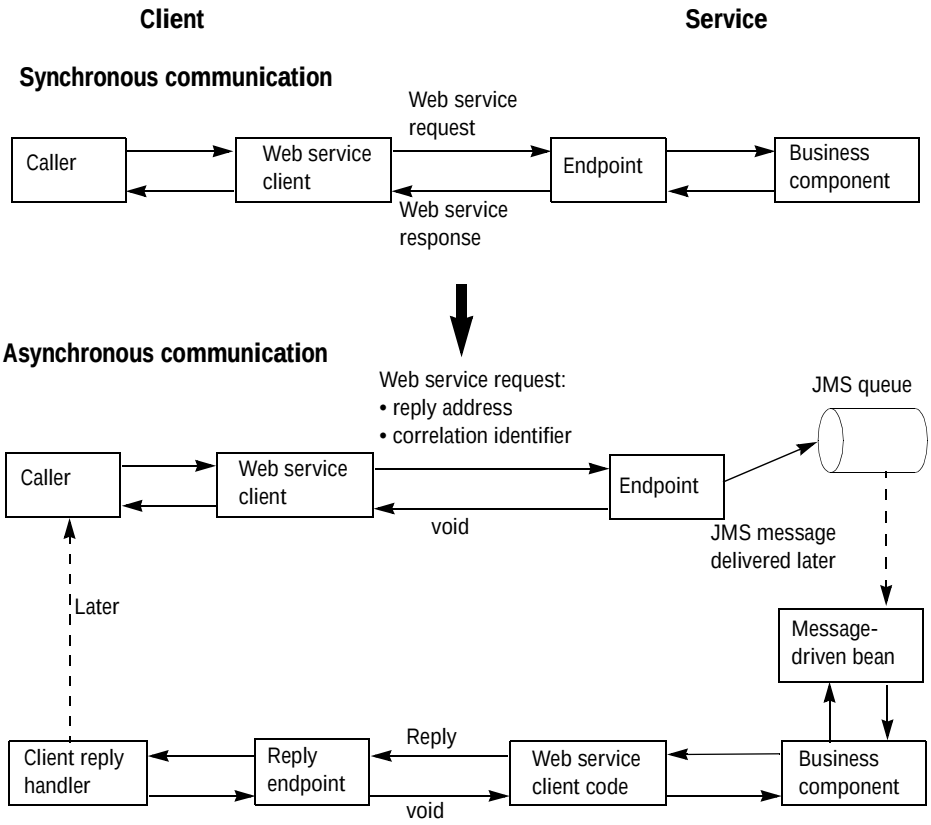

Figure 8.9 Converting from Synchronous to Asynchronous Communication

The JMS queue stores the messages until the message-driven beans are ready to process them, thereby controlling the rate of delivery. Thus, the component can process the messages without being overwhelmed, which helps the service to scale better. Figure 8.9 illustrates the process of converting an endpoint from synchronous to asynchronous communication.

Optionally, the client can further increase its responsiveness by employing some asynchronicity when making the Web service call itself. For example, a stand-alone client that uses a Swing GUI might spawn a thread to make the Web service call, allowing the client to continue its user interactions while the Web service handles the call. Or, a J2EE client component might use a JMS queue to make Web service calls—that is, rather than making the JAX-RPC call directly,

the client places the request into a JMS queue and a message-driven bean makes the Web service call to deliver the request.

8.5 Managing Complex Web Service Interactions

An application may have multiple interactions with the same Web service, it may access different Web services, or it may expose many Web service endpoints. There may be different types of interactions. An application may be both a service and a client. A service endpoint may receive documents of multiple types and hence needs to handle them accordingly. All this makes for complexity in managing Web service interactions.

Message-level metadata and a canonical data model are common helper strategies that support complex interactions, and they are good building blocks for adding functionality to a Web service message exchange. By consolidating Web service endpoints, you can make an application's Web service interactions more manageable.

8.5.1 Passing Context Information on Web Service Calls

Web services may need to exchange context information when handling messages. This context data augments the message contents by providing some extra information about the contents and may include information on how to handle the message itself. For example, a purchase order consists of the message contents—items to purchase, customer and financial information—and you also may need to pass some extra information about the message, such as its processing priority. The Web service may also need additional context information indicating how to process the message, such as processing instructions or hints. This context information is often passed along with the message. The following are some use cases where such additional data may be needed.

- Security information needs to be embedded within the document. For example, you may want to include some message-level security or add a digital signature to an accounts payment document as part of a credit card transaction.
- A correlation identifier needs to be included with the message to indicate that the message is associated with a logical group of messages exchanged in a workflow.

- Transactional or reliability information may be included along with the message contents to indicate additional quality of service activities that should be applied before processing the message contents.
- A message needs to be processed on a priority basis.

Several strategies exist for including context information with a message exchange, as follows:

1. Context information can be passed as an extra parameter in the Web service method. You can define the Web service interface to accept not only the XML document message, but to also include a second parameter that encapsulates metadata.
2. Context information also can be passed as part of the document itself. You can embed the extra metadata within the XML document. When you parse the document, you extract the needed information.
3. Context information also can be passed in the SOAP message header. You can embed the metadata in the SOAP message header and write a handler to extract it and pass it to the endpoint.

The first strategy, including context information as part of the service interface by adding an extra field for the input parameters or return values, effectively makes the context information part of the WSDL. For example, Code Example 8.3 shows how the boolean parameter `priorityProcessing` is added to the `submitPurchaseOrder` method. The value of the parameter indicates whether the order should be processed before other orders. Besides complicating the service interface, you must remember to update the interface—and regenerate the WSDL—if you add to or change the context information. For these reasons, the strategy of including context information as part of the service interface results in code that is harder to maintain.

```
public interface MyWebService extends Remote {
// priorityProcessing is the data indicating that the purchase
// order needs to be processed on a priority basis
```

```
public String submitPurchaseOrder(PurchaseOrder poObject,
          boolean priorityProcessing) throws RemoteException;
}
```

Code Example 8.3 Context information in a Service Interface

The second strategy embeds the information directly in the message contents. (See Code Example 8.4.) You create additional elements within the XML document itself, making the context information part of the data model—either part of the request parameter message or the reply return data. Add the necessary elements to the message schema, then add the data elements to the XML document before sending the message. The receiving endpoint parses the document and extracts the information into its application code. There are disadvantages to this strategy. You must revise the document schema when you change the context information. To retrieve the context information, you must parse the document, which can be costly especially if the document passes through intermediaries. There is also no logical separation between the schema for the document request and the schema for the context information. For these reasons, embedding the context information in the message contents may also result in code that is harder to maintain.

```
<PurchaseOrder>
<Id>123456789 </Id>
...
<POContextInfo> <!-- This is where the context data begins -->
    <PriorityProcessing>True</PriorityProcessing>
    <!-- Other context data elements -->
</POContextInfo>
...
</PurchaseOrder>
```

Code Example 8.4 Context Information in a Document

The third strategy is to embed context information as a new subelement in a message's SOAP header. For this strategy, clients need to embed this information in a SOAP header and the service needs to extract it. The service can use a JAX-RPC handler to intercept the SOAP message and then extract the header while still

keeping it associated with the document. The handler uses this context information to take appropriate actions before the business logic is invoked in the endpoint. This strategy is more elegant because it does not require you to modify the XML schema for the data model exported by the service. Instead, the handler unobtrusively determines the context information from the envelope's header, leaving the message body untouched and unaffected. See Code Example 8.5, which shows how to add context infromation to a SOAP header.

- ❒ This strategy is preferred to the others since it keeps the context information separate from the document contents and the service interface. This strategy also lets you encapsulate the context information processing in a handler, which keeps the handling logic for context information removed from the business logic.

There are some performance implications of this strategy, since each Web service endpoint invocation runs this handler to manipulate the SOAP message. The context information may be needed in only a few cases, but the handler runs with every invocation. This strategy may also be more difficult to implement.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="SoapEnvelopeURI"
       SOAP-ENV:encodingStyle="SoapEncodingURI">
    <SOAP-ENV:Header>
       <ci:PriorityProcessing>True</ci:PriorityProcessing>
    </SOAP-ENV:Header>
<SOAP-ENV:Body>
        <opc:submitPurchaseOrder xmlns:opc="ServiceURI">
           <opc:PurchaseOrder>
              <!-- contents of PurchaseOrder document -->
           </opc:PurchaseOrder>
        </opc:submitPurchaseOrder>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Code Example 8.5 Context Information in SOAP Headers

Sometimes, the handler needs to pass the context information (or the results of processing the context information) to the endpoint implementation. The way to do this is to use the JAX-RPC `MessageContext` interface, since this interface is

available to both JAX-RPC and EJB service endpoints. If the handler sets the context information in the `MessageContext` interface, the service endpoint can access it. To illustrate, you first set up a handler class similar to the one shown in Code Example 8.6.

```
public class MyMessageHandler extends
          javax.xml.rpc.handler.GenericHandler {
   public boolean handleRequest(MessageContext mc) {
       SOAPMessage msg = ((SOAPMessageContext)mc).getMessage() ;
       SOAPPart sp = msg.getSOAPPart();
       SOAPEnvelope se = sp.getEnvelope();
       SOAPHeader header = se.getHeader();
       SOAPBody body = se.getBody();
       if (header == null) {
          // raise error
       }
       for (Iterator iter = header.getChildElements();
                    iter.hasNext();) {
          SOAPElement element = (SOAPElement) iter.next();
          if (element.getElementName().getLocalName()
                 .equals("PriorityProcessing")) {
             mc.setProperty("PriorityProcessing",
                 element.getValue());
          }
       }
    ...
    return true;
}
```

Code Example 8.6 Passing Context Information from Handler to Endpoint

Then, you can get access to `MessageContext` in the endpoint that receives the request. For an EJB service endpoint, `MessageContext` is available with the bean's `SessionContext`. Code Example 8.7 shows the enterprise bean code for the endpoint.

```
public class EndpointBean implements SessionBean {
   private SessionContext sc;
```

```
    public void businessMethod() {
        MessageContext msgc= sc.getMessageContext();
        String s = (String)msgc.getProperty("PriorityProcessing");
        Boolean priority = new Boolean(s);
        ...
    }
    public void setSessionContext(SessionContext sc) {
        this.sc = sc;
    }
    ...
}
```

Code Example 8.7 Endpoint Receiving Context Information from a Handler

8.5.2 Handling Multiple Document Types

A Web service, over time, invariably will need to expand its ability to accept documents of different types. Developers may add new functionality to the same service interface, and this functionality mandates new document types. Although you can accommodate these changes by adding new methods to the interface to accept alternate forms or types of documents, this approach may result in an explosion in the number of methods.

A better strategy for handling multiple document types is to design the service interface with this possibility in mind. You can do so by writing a generic method to represent a document-centric interaction. For example, instead of having multiple methods for each document type such as `submitSupplierInvoice`, `submitSupplier2Invoice, and submitBillingInfo`, the endpoint interface has a single method called `submitDocument` that receives all document types. To accept any type of document, you need to use `xsd:anyType` as the parameter type for the `submitDocument` method. The service interface does not have to change to accommodate new document types, just the service implementation changes. See Code Example 3.19 on page 110 for an example.

Since your interface changed to a single method, the implementation needs to change appropriately. The `submitDocument` method needs to map these documents to the processing logic that handles them. You can apply a Command pattern to this strategy to achieve flexibility in handling these documents. With a Command pattern, you place the commands to manage all schemas of the various document types in one place, identifying each command by schema type. To add a new

schema, you simply add a new command that handles the new schema.

Figure 8.10 illustrates how you might apply this strategy. The Web service receives different types of documents from its clients. Each type of document maps to a schema. The service maintains a separate command for each document type represented by its schema. The command mapping happens in the interaction layer, separate from the processing layer and the business logic.

Using this strategy to handle multiple document types has an additional benefit: You can have the contents of the document, rather than the invoked method, determine the processing. To do so, you extend the logic of the command selection to include some of the document content. For example, if you wanted to apply business logic specific to a requestor, you might choose a command based on document type plus the identity of the requestor. For example, the adventure builder enterprise can apply additional business logic—such as verifying an invoice's authenticity—to invoices sent in by less trusted partners.

❒ Create an interface that has a single method to receive documents of different types. Use the Command pattern to map each document to its processing logic.

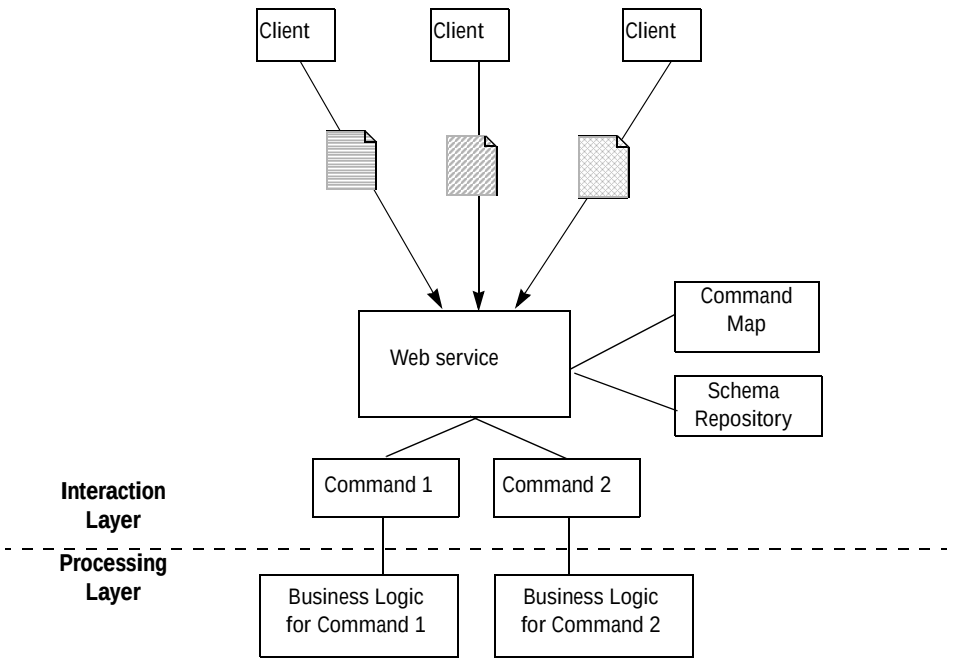

Figure 8.10 Using a Command Pattern to Handle Multiple Document Types

8.5.3 Consolidating Web Service Interactions

Consolidating Web service interactions is a good way to simplify the complexity of a Web service application. What starts as a straightforward Web service application with basic interactions between a client and a single service often grows into an application that exposes more endpoints and may itself act as a client to other services. The application communication becomes more complicated, and the business logic often becomes closely tied to a particular service. When the number of Web service interactions grows beyond a certain point, you may want to factor out common code for reuse.

For service interactions that rely on document passing, consider using a centralized manager that can broker the Web service interactions and consolidate the interaction layer of your Web services. This centralized manager can be responsible for both incoming and outgoing Web service interactions, and it can also consolidate all client access to a service. A centralized manager strategy applies only to interactions involving document passing, because such interactions can use an interface with a single method to handle all possible documents. Interfaces that pass objects as parameters do not have the flexibility to be this generic—a method passing or returning an object must declare the object type, which can only map to a single document type. The centralized manager needs to handle multiple document types by using the strategy described in "Handling Multiple Document Types" on page 371.

Take, for example, the order processing center interactions with its various suppliers. The order processing center is both a client of and a service to multiple suppliers. The order processing center and the suppliers engage in XML document exchange interactions, and often the interactions are asynchronous since they may span a fair amount of time. Without a centralized manager, your enterprise may have many point-to-point interactions among internal modules and external partners. This may result in the Web service code existing in many places throughout the enterprise. Each endpoint in these point-to-point interactions replicates the same work done by other endpoints, as do each of the clients. Keep in mind that as the number of services you expose grows, it becomes harder to manage this complexity.

Figure 8.11 illustrates how a centralized manager can simplify an enterprise's interactions with its external partners. The top part of the figure shows several entities of an enterprise each communicating with various external partners. The bottom part of the figure shows how much simpler it is to route the same interactions through a centralized manager.

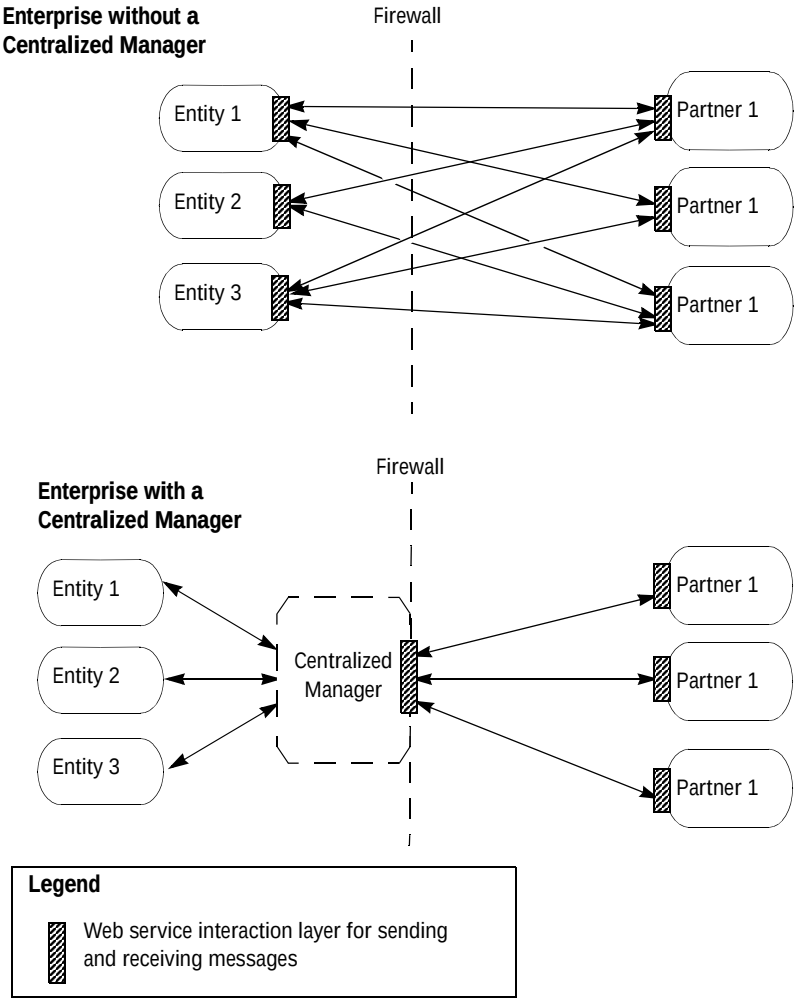

Figure 8.11 Enterprise Interactions with External Partners

To simplify things, the order processing center uses this centralized manager strategy to consolidate the Web service interaction layer code. The centralized manager handles all the Web service requests from the suppliers—it does any necessary preprocessing and interaction layer work, then dispatches requests to the appropriate processing layer business action component. To illustrate, we implemented this strategy between the order processing center Order Filler submodule and the supplier Web services. The centralized manager handles XML documents

sent in by suppliers. The centralized manager extracts information from these incoming requests, and this information lets it know how to handle the request. It may reformat the message content or transform the content to the internal canonical form used by the enterprise. The centralized manager routes the reformatted message to the correct target system. It can also validate incoming documents and centralize the management of document schemas. For outgoing Web service calls, the centralized manager acts as the client on behalf of the internal modules.

Not only does it centralize interaction layer and client functionality so that it is easier to manage incoming and outgoing Web service messages, use of this strategy also decouples components that access a service from those that focus on business logic. Your enterprise ends up with a clean Web service layer that is decoupled from the processing layer.

For handling incoming requests, you can establish a single interface and endpoint that all clients use. You can add context information to each request to enable an easy identification of the type of the request, such as purchase order or invoice. When this single endpoint acts as the interaction layer for several Web services, it can perform a number of functions, depending on what is available to it. If it has access to schemas, the endpoint can perform validation and error checking. If it has access to style sheets, the endpoint can transform incoming requests to match the canonical data model.

The centralized manager may also handle security tasks appropriate for the interaction layer and can translate incoming documents to an internal, enterprise-wide canonical data model. The key point is that the centralized manager needs access to information that enables it to perform these services.

A centralized manager may also use a message router to route requests to the appropriate business component recipients in the processing layer. A message router is responsible for mapping requests to recipients.

❒ As the number of Web service interactions grow within your enterprise, consider consolidating them in a centralized manager.

8.5.4 Canonical Data Model

Establishing a canonical data model is a good strategy for exchanging messages that rely on different data models. (See “Data Transformation” on page 275 for more details on using a canonical data model.) The adventure builder enterprise receives messages from suppliers through the centralized manager, which provides a natural place to hold a canonical data model. The canonical data model we use is repre-

sented in XML. The order processing center provides the canonical data model used for its interaction with the Web site.

8.6 Building More Robust Web Services

The J2EE 1.4 platform and its Web service technologies conform to the WS-I Basic Profile standards. Conforming to these standards means that the platform uses HTTP as the underlying transport for Web service calls. Unfortunately, from a business communication point of view, the HTTP protocol provides only a low-level degree of robustness. To illustrate, HTTP has no automatic retries to reestablish a connection that disconnects. If the receiving end fails, the entire HTTP call fails and there is no attempt to connect later. This low-level degree of robustness is not necessarily bad, since it means that HTTP can accommodate the participation of all kinds of systems and networks in a distributed Internet environment. HTTP is also a fairly simple protocol that any system can implement with a reasonable amount of effort.

Despite using more reliable hardware and software systems and communication links, enterprises have no guarantee against failures. Failures may occur during a Web service request between a client and a Web service that may leave an application in an ambiguous state. For example, a problem may occur when either party (client or server) fails while in the midst of a JAX-RPC call. Other problems may occur due to lost messages, messages arriving out of order, messages mistakenly seen as duplicates when the same message arrives multiple times, or when the contents of messages are corrupted.

The WS-I Basic Profile does not specify a standard mechanism for handling Web service failures, and hence the J2EE platform standards do not yet include standard support for all such failures. Emerging standards specifications are beginning to enable robust Web service communications using standard interoperable mechanisms. In the meantime, you can achieve more robust Web service communication by adding this functionality to your own application code. It is worth the effort even though you may not be able to handle all types of catastrophic failures.

Writing your own code to make a service more robust may result in a reduction in interoperability. Loss of interoperability can be an unfortunate side effect when implementing solutions ahead of industry standards. In the following sections, we provide some strategies to add some robustness to your Web services. The solutions we propose here are limited to scenarios that involve a single request/reply message exchange between a J2EE client (a servlet or an enterprise

bean) and a J2EE Web service endpoint using JAX-RPC as the communication technology.

8.6.1 Use Idempotent Endpoints

Duplication of messages can be a problem in a distributed environment. Different situations can cause duplicate messages, such as sender retries because of communication failures, maliciousness, or bugs in the code. Duplicate messages may also be due to user error, such as when an impatient user presses a submit button more than once. A message receiver must guard against unintended side effects (such as processing an order twice) because of these duplicate messages. Creating idempotent endpoints is one strategy a service can use to handle duplicate messages.

Idempotent refers to a situation where repeated executions of the same event have the same effect as a single execution of the event. Making your endpoints idempotent avoids the problem of a service processing duplicate instances of the same message or request. Even if a client mistakenly—or even maliciously—submits a request such as a purchase order (via a JAX-RPC call) more than once, the effect is the same as if the request was submitted just once.

Service endpoints that only perform read operations are naturally idempotent, since these operations do not have any side effects. For example, the adventure builder application's CRM order tracking Web service is naturally idempotent since it only invokes read operations. For Web services that perform updates or otherwise change state, you need to explicitly build idempotency. One way to do this is to leverage the semantics of the endpoint logic. If you know the effect of a single execution of an operation, you can change the program logic to ensure that multiple executions of the operation have the same result as a single execution.

Let's look at how to design an idempotent endpoint for services that perform updates or change state. First, to detect duplicate requests, you need to assign a correlation identifier to client interactions with a service. (See "Correlating Messages" on page 359.) This correlation identifier can be passed as context information and intercepted and processed by a JAX-RPC handler. (See "Passing Context Information on Web Service Calls" on page 366.) When the request is received, the endpoint should check to see if it is a duplicate request. For example, the order processing center's `submitPurchaseOrder` method can be made idempotent since we know its operation depends on an order identifier key value. Before executing the business logic, the `submitPurchaseOrder` method stores the order in the database with the order identifier as its primary key. If the same purchase order is sent again, the second attempt to store the same order identifier in the database results

in a duplicate key exception, preventing the order from being processed a second time.

8.6.2 Use Client Retries with Idempotent Endpoints

Idempotent endpoints can help to set up a fault-tolerant Web service interaction. Because multiple service requests have the same effect as a single request, clients of a idempotent service can retry message requests until they are successful, without fear of causing duplicate actions. However, using idempotent endpoints for a fault-tolerant Web service adds to the application's complexity and may adversely affect performance. It is also not interoperable: Specifications for this area are still being designed, and it is doubtful that the standards that ultimately result will work with current custom-built schemes.

With a fault-tolerant design, a client needs to retry sending messages until successful. Typically the client retries only a fixed number of times and then fails. You also design the endpoint to be idempotent. You may also need an acknowledgment message, which can be part of the JAX-RPC reply message or a separate message (if using asynchronous processing).

Often before executing a retry, a client waits briefly so that transient conditions in the network or on the server may clear. When using synchronous Web service calls, the client—especially when it is a real person using the client—may find these waits unacceptable. In cases such as this, consider converting synchronous calls to asynchronous calls. See "Refactoring Synchronous to Asynchronous Interactions" on page 363.

While you design your logic, remember that a failure can occur at any point during a JAX-RPC interaction—when the client makes a call or the service receives it, while the service processes the call, when the service sends a reply or the client receives it, or when the client processes the reply. To recover from such failures, you may want the application code to log interaction state on the client as well as the endpoint. If a failure occurs, you can use the log to recover and finish the interaction, deleting the log when complete or otherwise marking it as finished.

How does a client effectively use a service with this type of robustness? You need to formulate a contract—a set of understood interaction rules—between the client and the service endpoint. These rules might specify that retries are allowed and how many, the overall protocol, and so forth. Generally, the client makes the call and can repeat the call if it is not notified of success within a certain time limit. The service is responsible for detecting duplicate calls.

8.6.3 Handling Asynchronous Interaction Error Conditions

Application design should include handling recoverable and irrecoverable exception conditions in a user-friendly manner. However, error handling is more complicated with distributed systems. Asynchronous interactions add further to application error handling complexity, principally because the interaction requestor is not available to receive an error report. In addition, human intervention may be needed to resolve the error condition.

Let's look at how the adventure builder application handles exception conditions. For its Web service interactions, the adventure builder application gives the interaction requestor a correlation identifier for the submitted request. The service endpoint implementation catches any exceptions that occur when the request is preprocessed. Exceptions might be an invalid XML document or XML parsing and translation errors. For asynchronous interactions, the service endpoint implementation uses JMS to send requests to the processing layer. In those cases, it may also encounter JMS exceptions. For these errors, the endpoint passes a service-specific exception to the requestor. For more information on designing this portion of the exception handling mechanism, see "Handling Exceptions" on page 80.

In adventure builder, exceptions may occur as follows:

1. **Case 1:** During any Web service interaction between the Web site and the order processing center or between the order processing center and the external suppliers. These Web service interaction exceptions are synchronous in nature and thus easier to handle than exceptions that arise in cases 2 and 3.
2. **Case 2:** Within the order processing center, while processing an order during any stage of the workflow management operation.
3. **Case 3**: During the order processing center interaction with partners, such as the suppliers and the credit card service.

Code Example 8.8, which illustrates Case 1, shows a portion of the code for the order processing center Web service interface that receives purchase orders from the Web module. The interface throws two kinds of service-specific exceptions: `InvalidPOException`, which it throws when the received purchase order is not in the expected format, and `ProcessingException`, which is throws when there is an error submitting the request to JMS. Note that the platform throws `RemoteException` when irrecoverable errors, such as network problems, occur.

```
public interface PurchaseOrderIntf extends Remote {
   public String submitPurchaseOrder(PurchaseOrder poObject)
           throws InvalidPOException, ProcessingException,
           RemoteException;
}
```

Code Example 8.8 Handling Exceptions in Web Service Calls

Exceptions that occur when processing an order either within the order processing center or during the interactions with partners (Cases 2 and 3) require a different handling approach. Since request processing is asynchronous, the client that placed the request is not waiting to receive any exceptions that you might throw. You also need to differentiate exceptions that require a client's intervention from those that may be temporary and possibly resolved on a retry. If an exception requires a client's intervention, you must inform the client in some way. For exceptions that may be temporary, you can designate a certain number of retries before giving up and informing the client.

For example, consider the order processing workflow of the order processing center. After the endpoint receives the purchase order from the client and successfully puts it in the workflow manager's queue, it returns a correlation identifier to the client. The client then goes about its own business. At this point, the workflow manager takes the order through the various workflow stages. Different exceptions can occur in each stage. These exceptions can be broadly categorized as those that require immediate client intervention and those that require an application retry.

Let's look first at exception conditions that require human intervention, such as notifying the customer or an administrator. Examples of such exceptions might be an incoming purchase order that needs to be persisted to the data store but the database table does not exist. Or, the credit card agency might not grant credit authorization for an order because of an invalid credit card number. When these exception conditions occur, it is impossible to continue purchase order processing without human intervention. The adventure builder application informs the customer via e-mail when such conditions happen.

Now let's look at handling exceptions that require the application to retry the operation. As an order progresses through the workflow stages, various exception conditions of a temporary nature may occur. There may be an error placing the order in a JMS queue, the database connection may be busy, a service may not be

available, and so forth. For example, the workflow manager might try to invoke the supplier or bank Web service while that latter service is down. Or, the order processing center database may be temporarily unavailable due to a back-up operation. Since temporary error conditions often resolve themselves in a short while, asking the customer to intervene in these situations does not make for the best customer experience.

A strategy for handling temporary error conditions involves keeping a status indicator to signal retrying a failed step at a later time. The adventure builder application workflow manager, as it moves a purchase order through various stages, tracks the status of the order at each stage. If an error occurs, the workflow manager flags the order's status to indicate an error. By using the timer bean mechanism available in the J2EE 1.4 platform, the workflow manager can periodically examine orders with an error status and attempt to move these orders to their next logical stage in the flow. The timer bean activates after a specified period of time, initiating a check of orders flagged with errors and prompting the workflow manager to retry the order's previously failed operation. The order status, in addition to keeping an error flag, tracks the number of retry attempts. The workflow manager seeks human intervention when the number of retry attempts exceeds the fixed number of allowable retries.

8.7 Conclusion

In this chapter we examined the adventure builder enterprise application, in particular its Web service implementations. We discussed the rationale for our different Web service application design choices and illustrated how we went about implementing these various endpoints and communication patterns. Drawing on examples from the application, we tried to bring to life many of the Web service architectural and design issues.

The chapter covered the details of Web service design and implementation, including establishing a service's interaction and processing layers, selecting the appropriate endpoint type, determining the proper granularity, passing parameters between client and service, delegating to business logic, publishing and deploying the service, and client considerations. It also discussed how to extend a Web service's quality of service. It considered strategies such as using context information with messages, relying on a canonical data model, including a correlation identifier, and using registries, among other strategies for adding robustness to a service.

In this book, we have tried to cover the Web service standards as they currently exist and how you can develop Web services on the J2EE platform that conform to these standards. We have also tried to show how you can use the J2EE platform technologies to achieve additional robustness and quality of service beyond the current standards for your Web services.