CHAPTER 8

Application Architecture

BY now you should have a good understanding of the J2EE platform and its technologies. You should also have a clear idea of how best to apply these technologies in your Web service application. Using an example application, this chapter pulls it all together to show recommended approaches to designing an entire Web service application.

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. We explain the particular path that we took to architect and design the Adventure Builder application, including the significant challenge of constructing the application modules so that they work together smoothly. We explain the motivational factors and issues that we had to consider, and show not only the decisions we reached but how we came to these decisions. By illustrating our design process, we hope to make it easier for you to determine how best to architect and design your own Web service applications.

Note: The same application described here is currently in an early access state of development. As such, the description of the application is limited.

8.1 Overview of Adventure Builder

Let's start by reviewing what was said in Chapter 1. Adventure Builder enterprise, which is an *existing* enterprise that sells adventure packages for vacationers, is considering going online. The enterprise plans to create a Web site to expand its customer reach and integrate its functionality with its numerous partners.

Recall that the Adventure Builder enterprise provides customers with a catalog of adventure packages, accommodations, and transportation options. Customers select from these options to build a vacation. For example, a customer might choose an Adventure on Mt Kilimanjaro. In addition to selecting accommodations, mode of transport, and preferred dates, the customer selects from adventure activities, such as mountain climbing to reach the summit, mountain biking to a hidden waterfall, and hiking and exploring the mountain.

The enterprise, prior to considering the move to a Web presence, had already implemented an extensive computer system for its business operations. These business operations include functions to:

- Interact with interested customers
- Receive and process customer orders
- Manage customer relations, including tracking customer preferences and updating customers on the status of an order
- Verify and obtain approval for payment
- Interact with business partners—airlines, hotels, car rental agencies, adventure or activity providers—to provide a complete package to customers

Now that we have reviewed the Adventure Builder enterprise's requirements and functional needs—how it looks on the outside to customers and other users—let's examine the application's internal setup. That is, let's look at how the application was architected to meet these user requirements. The Adventure Builder application consists of four application modules: a Web site customer application, and three applications for fulfilling an order, all built on the J2EE 1.4 platform technologies. Adventure Builder is designed to receive, handle, and fulfill customer purchase orders—in short, it is an order fulfillment system. Currently, the application is in an "early access" stage, which means that not all functionality has been implemented. In place are some key applications of the order processing center module, in particular an order tracking application to track orders currently

being processed and an order receiver application to handle the submission of orders. There is also a browser client and module to let customers browse the catalog and place orders. Figure 8.1 illustrates these various application modules and their relationships to each other and to outside partners and services.

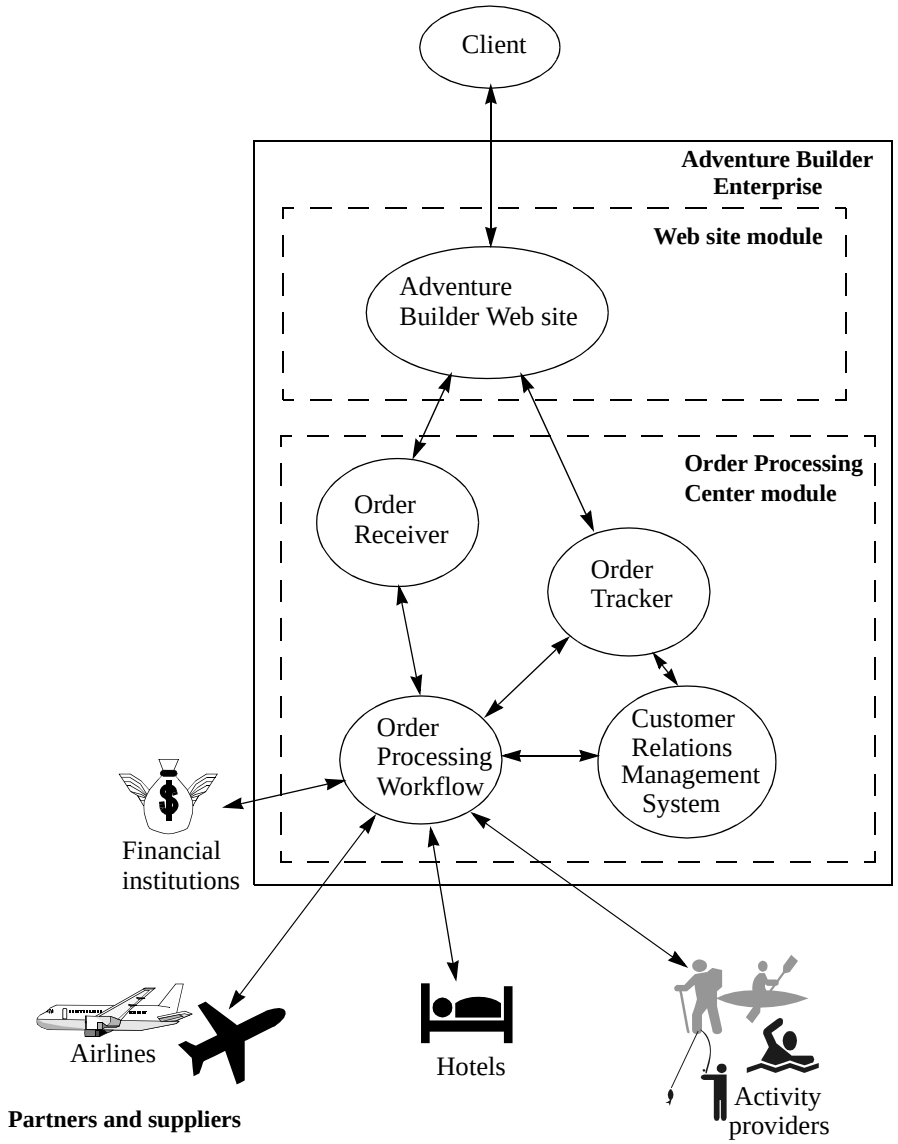

Figure 8.1 Adventure Builder Enterprise Modules

The diagram shows how any type of client—browser client, stand-alone client, hand-held client, or a client based on any platform—accesses the Adventure Builder enterprise through the Adventure Builder Web site. The Web site application module interacts with Adventure Builder's Order Processing Center (OPC) application module. The OPC module in turn encompasses modules and applications to receive orders, track orders, process orders, and maintain customer relations. For example, the order processing workflow application interacts with financial institutions and credit card companies for payment, plus it interacts with business partners to set up activity packages, reserve airline flights, hotels, and so forth.

8.2 Adventure Builder Architecture

Let's start with a closer look at the key Web service interactions in Adventure Builder, particularly at how the various applications within the enterprise work together to fulfill an order. After we identify these interactions, we'll examine the underlying design issues.

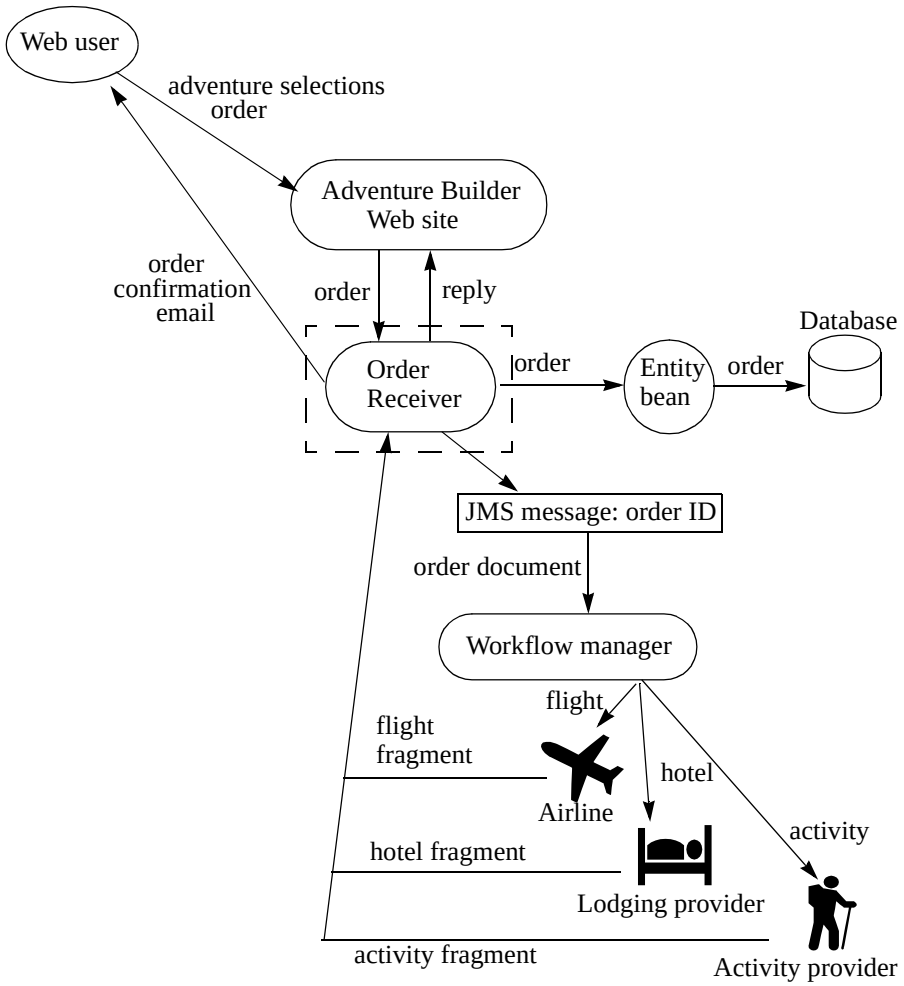

Figure 8.2 Adventure Builder Work Flow

Distilling the functionality down to a simple level, the Adventure Builder 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 by making a Web service call to the order receiver module and passing the order information as a message. The order receiver module receives the order message, which it persists in the database using an entity bean with container-managed per-

sistence to represent the order. Before doing so, it generates an order identifier, and that identifier becomes the primary key for the entity bean.

Notice that there is no need for an interaction layer between the Web site and the order receiver modules. It's possible to omit an interaction layer because the same department owns both the order receiver and the Web site. As the sole owner of both modules, the department has complete control over changes to the modules and can easily ensure that changes to these modules stay synchronized. Although an interaction layer can be necessary and helpful, using such a layer when it is not necessary is not advantageous. An interaction layer can reduce overall throughput. It can also decrease the responsiveness of the Web site, since the work of an order receiver interaction layer to handle order placement calls from the Web site would be synchronous.

After persisting the order in the database, the order receiver module asynchronously starts the processing of the order by sending a JMS message. Since order processing is asynchronous, the order receiver module does not wait for processing to complete but instead immediately returns a reply to the Web service call.

A workflow manager message-driven bean receives the JMS message. This message-driven bean is responsible for directing the order fulfillment workflow and ensuring that the order is processed successfully. The workflow manager takes the order, splits it into multiple smaller orders, and then sends these smaller pieces to the various suppliers for fulfillment. For example, the activity providers fulfill the activity request, while the lodging provider fulfills the hotel request. The interaction with the various suppliers is through Web services using a document-oriented interface. (See "Designing Domain-Specific XML Schemas" on page 181.) When it is notified that the suppliers have completed fulfilling all parts of the order, the order receiver module executes the equivalent of a global transaction to commit the order—instead of actually initiating a transaction, the order receiver collects the completed order fragments from the individual suppliers and joins them together. When it determines that the entire order is fulfilled, the order receiver sends an order confirmation email to the user. The system also periodically checks to ensure that orders are not left waiting for responses from suppliers.

As you can see, the key architectural problem we must solve is the need for different entities (or applications) to communicate and work together to solve particular business problems. Communication occurs across numerous boundaries, and several entities must coordinate their work efforts. Clients communicate with the Adventure Builder Web site. The Web site module must send customer orders and requests for follow-up status to the OPC module. Within the OPC module, different application entities handle receiving, tracking, and processing orders.

Other application entities handle requests for order status. There must be cooperation and communication between these different entities.

We start by addressing the high-level design issues for the order tracking module within the OPC, focusing on a single interaction between two applications. These design considerations include:

- Choosing a communication technology
- Deciding on a development approach
- Selecting an endpoint style
- Choosing a client approach

Then, in "Order Tracker Design Details" on page 324, we describe where and why we made these choices in Adventure Builder.

8.2.1 Choose Communication Technologies

You must decide on the best communication technology for your application. Recall from "Communication" on page 24 that, when developing on the J2EE platform, you can use such technologies as JMS, XML, RMI/IIOP, Web services, and so forth.

For Adventure Builder, we have chosen to use Web services since we are primarily interested in achieving the greatest degree of interoperability—that is, interoperability among all modules and among all types of clients. We use Web services because we want the Adventure Builder application to be available to as many clients as possible, regardless of the platform or technologies that the client is using. 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. By having a Web service interface between these different modules, they can be kept separate and independent of each other.

It is also important for Adventure Builder to be able to integrate its functionality with that of its numerous partners. This is essential since it is the partners that 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. Since Web services also allow for the most reusability, different applications can reuse the Web services order processing capabilities.

The OPC module makes its services available to the Adventure Builder Web site as a Web service. That is, the Web site uses the OPC's Web service to fulfill an order. There are advantages to this style of implementation. For one, the Web site for the OPC module may be hosted outside the firewall, even though the OPC module itself is always inside the firewall. The Web site may conveniently use the HTTP port available on the firewall. 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 OPC 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 OPC module, since it works asynchronously, is more concerned with achieving high throughput.

Another high-level choice we need to make is the form in which to pass information among modules. 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.

8.2.2 Select a Development Approach

The first question involves deciding on the approach to use for developing these different Web service interfaces. We followed the guidelines suggested in Chapter 3, particularly those in "Designing the Interface" on page 71. Our choice was as follows: either define the Java interface and derive the WSDL description from that interface (Java-to-WSDL approach) or define the WSDL and then derive the Java interface from the WSDL (WSDL-to-Java approach). We chose to use the Java-to-WSDL approach for all our Web service interactions because:

- First and foremost, this approach is simple and straightforward.
- Using the Java-to-WSDL approach, we do not have to know the details of WSDL and the WS-I interoperability specifications.
- The application server that we used to build this application—the Sun ONE Application Server 8.0, Platform Edition—provides tools that create WS-I-compliant WSDL files from Java interfaces.
- Since tools produced the standard WSDL document, we did not have to directly manipulate the WSDL document. As a result, we could be confident that the tool would handle WSDL conformance to the WS-I specifications, thus ensur-

ing that our application and Web service integrates with heterogeneous platforms.

8.2.3 Endpoint Considerations

A key architectural decision for the service endpoint entails the tier—EJB or Web tier—on which the endpoint resides. Choosing the best interface style for the endpoint is another important architectural decision. Recall from Chapter 3 that Web services may be implemented either on a Web tier or on an EJB tier. That is, the endpoint representing the Web service is either a service endpoint implemented on a Web tier or an EJB endpoint implemented on an EJB tier.

We examined the functional requirements for the processing layer of the order fulfillment Web service interfaces—that is, the Web service interface between the Web site and OPC modules and the interface between the OPC module and the various suppliers—and determined that these Web services required:

- Transactional access to a database
- Protection from multi-threaded access
- Restricted access to specific methods according to types of clients

We decided that the Enterprise JavaBean technology works best to meet these requirements when implementing the business logic, so we chose to use EJB service endpoints for all these Web service interfaces. (See “Choice of the Interface Endpoint Type” on page 76 for more information.)

Although we chose an EJB service endpoint, we considered using a JAX-RPC service endpoint for the order tracking Web service interface that is between the Web and OPC modules. Since this interface did not require transactional access or multi-threaded access protection, a JAX-RPC service endpoint would have worked. The interface could have been designed to use JAX-RPC service endpoint that issued JDBC calls to retrieve order status. However, since our design was based on an existing EJB-centric processing layer that was already dealing with workflow and order status requests, using a JAX-RPC service endpoint would have unnecessarily introduced a Web layer for this service. Such a Web layer would have merely acted as a proxy, directing requests from the Web tier to the EJB tier. We wanted to avoid adding unnecessary complexity and extraneous

layers to the design. For these reasons—to avoid both writing an additional JDBC interface to retrieve order status and introducing a new Web layer to act just as a proxy to the EJB layer—we used an EJB service endpoint that uses the existing enterprise bean interface for getting an order status.

8.2.4 Types of Parameters

A client invokes a service's functionality, and exchanges parameters with the service, in either an object-oriented or document-oriented fashion. Recall from "Designing a Service's Interaction Layer" on page 71 that an object-oriented approach implies invoking a method call on a service, passing data to the method as parameters to the call, while a document-oriented approach implies passing messages as XML documents. Typically, messages describe the function to perform plus any associated data that the service might need to correctly carry out the operation.

Keep in mind that the two interface styles have different implications regarding synchronicity of the communication. Object-oriented interactions tend to be more request-response in nature, implying a more synchronous form of communication, while document-oriented interactions usually are more asynchronous.

Here we examine some considerations applicable to Adventure Builder's Web service interfaces, starting with the order tracking Web service interface. (See Code Example 8.1.)

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

Code Example 8.1 Order Tracking Web Service Interface

Since the order tracking Web service has one method that retrieves data about an order, it was a straightforward matter to use the order identifier for its `getOrderDetails` method argument. A client needs to submit an order ID to get information about an order. It would have been clearly overkill to have the client submit an XML document containing this same information. For the order identifier argument, we chose a suitable Java parameter, a `String` object.

The `getOrderDetails` method returns specific order details (such as user name and shipping address) rather than the complete purchase order. Choosing the return value type is not as straightforward. It's possible to return this set of detail

information as either a Java `Object` or in an XML document. We chose to return the order details in a `OrderDetails` Java `Object` for the following reasons:

- Details returned to the client—user identifier, addresses, and so forth—are generally stable in format. Since the details do not change much, the WSDL for this service remains stable.
- The same organization within the Adventure Builder enterprise is responsible for both the Web site and the OPC modules. Ownership by the same group makes it easier to implement matching changes to both modules.

Given these factors, we chose the simplicity of returning the details in Java `Objects` rather than assembling and disassembling XML documents.

Now let's consider the order fulfillment Web service, which is between the Web site module and the order processing center module. See Code Example 8.2.

```
public interface PurchaseOrderIntf extends Remote {
    public String submitPurchaseOrder(WSPurchaseOrder poObject)
                                          throws RemoteException;
}
```

Code Example 8.2 Order Fulfillment Web Service Interface

It was also straightforward to use a Java `Object`, `poObject`, for the `submitPurchaseOrder` method, since this is the simplest way to submit the order details. As with the order tracking interface, the purchase order data is fairly stable and the same organization controls both modules.

The OPC interactions with its suppliers differs from the order tracking Web service interactions. The order tracker service in an example of an object-oriented interface—information is passed between client and service as Java objects. In contrast, the OPC interactions with its suppliers model a supplier business that exchanges documents with its customers. As such, we use the document-centric interface style. In one such interaction, the OPC acts as a client of a supplier. To fulfill part of the customer's order, the OPC module sends a purchase order to the supplier's Web service. For example, to fulfill the customer's flight request, the OPC may send an order to the airline supplier. The OPC modules sends its request

as an XML document to the supplier Web service. Our examination of this service interaction highlights the design issues for document-centric Web services.

When services are designed to be used with many partners, which is true for the OPC to supplier Web service, it is important to have a very stable integration point. Stability of the service interface is of primary importance because the service's clients may be other companies or different departments within the same enterprise. Unlike the customer Web site to OPC interactions, these different companies and departments do not have control over both the client and service sides of the interaction. Thus, they depend on the interface remaining constant.

Let's look at the airline supplier interface in detail. (See Code Example 8.3.)

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

Code Example 8.3 Airline Reservation Interface

The service has a single method, `reserveAirlineSeat`, which takes a reservation request document and returns a reservation response document. 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. As it completes the order, it builds a response document to return to the client.

When designing an interface that must remain stable, good design dictates using a document-centric interface and then using the documents being exchanged as the primary integration point. This means that we use a separate schema for the reservation request—this schema becomes the stable integration point. We must also represent the return document as a schema.

Other factors also influenced our choice to send XML documents as the information exchange medium.

- The interaction between the OPC module and suppliers crosses enterprises, which is typical of business-to-business exchanges. For legal reasons, interactions that span enterprises require well-defined documents.
- By passing information in XML documents, applications can apply XML security technologies (such as encryption or digital signatures) to entire documents (or to parts of the documents) at the XML level.
- With XML documents, providers can define and expose the XML schema describing the document format. Having a schema available makes it easy to validate incoming documents. A schema enables a client that uses the service to apply mapping technologies (such as using JAXB to map document elements to Java objects) from within the schema itself.

Keep in mind that with a document-oriented interface, the WSDL is neither the primary integration point nor is it fully descriptive, since its description of the reservation request and response documents are `String` parameter and return values. (An object-oriented interface, in contrast, would maintain descriptions of every field in the purchase order.) Because the WSDL only keeps a single string to describe parameters, and it cannot provide details about what these documents contain, a separate schema holds the complete descriptions 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 it could entail negotiating some location with individual clients.

❒ On the Java platform, it is most efficient to send an XML document as a `javax.xml.transform.Source` object. (See the discussion in “Java Objects as Parameters” on page 83 and “Receiving XML Documents as Parameters” on page 107.) However, we chose to send the XML document as a `String` object because the WS-I interoperability requirements do not yet support `Source` objects. (For more information, see “Interoperability” on page 87.)

8.2.5 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 requirements such as coupling, portability, the ability to dynamically locate and call services at runtime, and ease of development.

Since the service endpoint interface that models the OPC module's interaction with the Web site is not expected to change, clients can use the stub approach. The stub approach may be the simplest of the three approaches, especially if the service's WSDL document is available at development. The stub approach 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 style of proxy generation provides the client with a tightly-coupled view of the Web service, and it is a good choice when the interface for the service endpoint does not change.

The stub approach is easiest for a client developer, since all necessary classes, including the service endpoint interface against which the client program is developed, are generated for a client. However, the stub approach is not portable across JAX-RPC implementations. Despite its portability deficiencies, we opted to use the stub approach for the communication between the Adventure Builder consumer application and the OPC module because of its other benefits.

The third approach, DII, is more suited to locating services dynamically at runtime in a loosely-coupled manner. Since more happens dynamically, the DII approach does come with an added cost of performance. However, it is often more difficult to develop programs with the DII approach, and this is its primary drawback.

8.3 Order Tracker Design Details

Let's examine some of the design details for Adventure Builder, beginning with the order tracker module. Recall that the Web services portion of Adventure Builder performs two key interactions with its customers: customers can submit orders and they can track order details.

The OPC module provides the functionality for receiving and handling order submissions, and later for tracking these orders. A customer who buys an adventure package essentially initiates a purchase order for the package. Adventure Builder sends a purchase order to the OPC module via a Web service, and the OPC module's order receiver gets the purchase order properly registered and identified (by assigning it an order identifier).

We extended Adventure Builder's OPC module to allow customers to track their orders. To have this functionality be available to as broad a range of customers as possible, we chose to use Web services. Thus, any type of client, even other applications, can access this functionality through the Web service. The OPC module maintains an order tracker within its customer relations manager (CRM) module. After submitting an order, customers can track its status, via a Web service interface, using the order identifier assigned at the time of submission.

Order tracker is a simple model and we tried to keep its implementation as simple as possible. Since order tracker needs only the order identifier, we pass the order identifier as a parameter when making a method call on the order tracker service interface. Once it locates an order, order tracker returns the order details as a Java object. Because the model is so simple, we implemented the order tracker with the easier-to-use Java object-oriented interaction style.

We chose to use an EJB endpoint for Adventure Builder. Often, developers implement Web service endpoints on the EJB tier to leverage the services of the EJB container. In the case of Adventure Builder, the OPC module was already implemented with an EJB component. When we extended Adventure Builder to Web services, we did not want to redo all the code, and certainly did not want to rewrite the OPC module. In addition, we also need to persist data to a database and continue to use the persistence and transaction services of the EJB container. For these reasons, it made sense to use an EJB endpoint when exposing the OPC functionality as a 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 the interaction layer has a different object model from the business logic processing layer. However, in our case, the order tracker's interaction layer object model is virtually identical to the CRM business logic object model. For even more simplicity, we have merged the interaction and processing layers into one layer.

Although merged into one layer, the order tracker Web service interface does not expose the internal CRM data model. It is important to keep the Web service interface from being tightly coupled to the CRM module, and we accomplish this by *not* exposing the internal data model through the service interface. This lets us change the internal data model of the CRM without affecting the Web service interface and clients.

Instead, we created an `OrderDetails` Java object to hold the data for the order tracker interface. The CRM module uses a different data model, a `status` object, and the `OrderDetails` and `status` objects are kept separate from each other. A separate piece of code maps these two objects. If the CRM data model changes,

we need only change the mapping code. Thus, the client's view of the `OrderDetails` object through the order tracker interface remains the same regardless of changes to the CRM `status` object. See Figure 8.3.

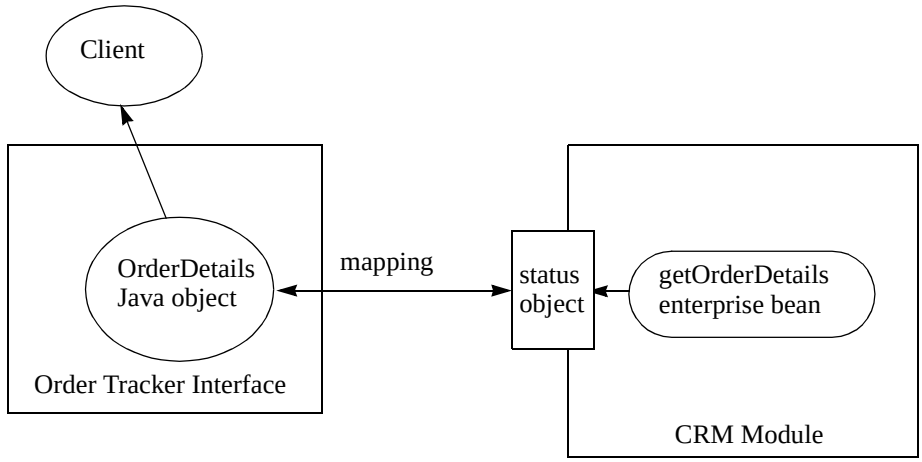

Figure 8.3 Separating CRM and Order Tracker Data Models

- Do not directly expose internal data of an application to the Web service interface.

The drawback of such an approach is the extra effort required to implement it. With the availability of numerous tools for creating Web services, it is quite easy and tempting to do the opposite: to publish a Web service that just expose the internal data model. That is a reasonable choice for a number of cases, especially when the XML schema based external data model is not the point of stability in the system.

8.3.1 Order Tracker Endpoint Design

For endpoint design, you can choose to start with a WSDL description and generate Java interfaces, or you can start with Java interfaces and generate a WSDL file. Generally, choose the mode that is most closely aligned to your experience and expertise. If you are a knowledgeable Java programmer, it makes sense to leverage your experience and start with Java interfaces rather than having to become adept with a new language, such as WSDL.

For Adventure Builder's Web services interactions, we chose to use the Java-to-WSDL approach—we developed the service by first coding the application using Java classes and then using tools to generate the WSDL document from these Java classes. We choose the Java-to-WSDL object-oriented approach because it is simpler and easier to use than the document-oriented approach. It also lets us leverage our Java expertise, since with this approach, we define an endpoint interface using the Java programming language and then let the tools generate the WSDL document. Developers need to be proficient in the Java programming language, but they do not have to also be experts in WSDL concepts and terminology. It is expected that developers are good Java coders, but many developers are less knowledgeable about WSDL details. The Java-to-WSDL approach has another advantage: it enables us to stay compliant with the WS-I basic profile.

Next, we had to decide whether the endpoint would use an object-oriented approach—and pass information as Java objects—or a document-oriented approach and pass information as XML documents.

We also had to decide whether to model the endpoint as an EJB service endpoint or a JAX-RPC service endpoint. We chose an EJB service endpoint because the application is an existing enterprise bean-centric application. The existing application resides on the EJB tier and has no current Web tier components, so it is natural to use an EJB service endpoint. Had we decided to use a JAX-RPC service endpoint, we would have had to implement a Web tier component to handle only the endpoint functionality. Since no other Web tier components are needed, it is not worth the extra overhead of using a JAX-RPC service endpoint.

As noted previously, we also chose to merge the interaction and processing layers of the service endpoint. Since no preprocessing occurs, the order tracker service requires very little from an interaction layer. It is simpler to merge the implementation of the two layers when one of the layers has so little work to do, as in this case.

On examination, we realized that the order tracker service had no need for SOAP handlers, so none were implemented. As for handling exceptions, the service endpoint ideally throws a service-specific exception if an order is not found. Currently, however, the service returns a `null` string for this error.

The order tracker service endpoint consists of an interface, `OrderTrackingIntf`, implemented by an enterprise bean. The interface uses the `OrderDetails` object to return to the client details about an order. See Figure 8.4.

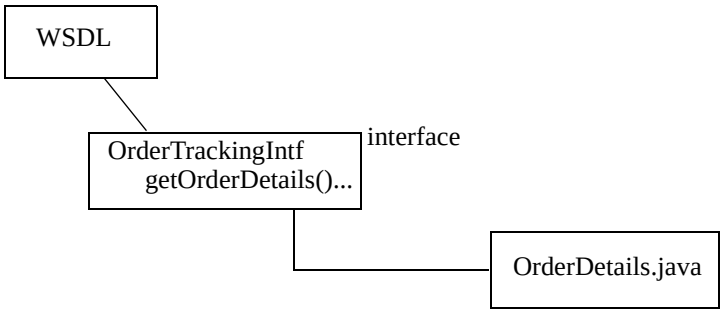

Figure 8.4 Order Tracker Service Endpoint Design

Furthermore, the Adventure Builder application is wholly contained within the context of one enterprise. As such, it does not need to concern itself with some of the legalities that come into play in the business-to-business processing arena. When such legalities must be addressed, they are often best handled by the document-oriented approach.

As we design the endpoint, we want to keep certain points in mind to avoid areas of brittleness. We want to be sure that the interface, which is expressed in WSDL, is stable. Stability is important because clients are dependent on the service and changes to the WSDL impact clients of the service. For stability, it is important to keep separate the exposed service endpoint interface and the data model of the existing application. We do not want to expose the system's data model directly in the WSDL, since doing so would necessitate changing the WSDL whenever there were changes to the data model—again, impacting clients of the service.

With these factors in mind, we tried to follow these guidelines:

- When designing a Web service on top of an existing application, we considered creating an extra set of classes to be part of the interaction layer.
- During an early prototyping phase and to facilitate development, we felt it was permissible to expose in the Adventure Builder service endpoint interface some parts of existing application classes as return objects or parameters.
- Later in the release cycle, we created a repository with the XML schemas for the common types used in our enterprise application. For example, we created

ContactInfo and Address schemas, and these two schemas are used throughout the application.

Figure 8.5 shows a UML diagram for the order tracking endpoint interface. The interaction layer consists of the WSDL description, the endpoint interface, a service implementation bean, and an `OrderDetails` object, which contains `ContactInfo` objects. The `OrderDetails` object is returned to clients of the service. The `ContactInfo` objects are part of the existing system's data model.

The processing layer represents the functionality of the existing application. It is here that the application keeps track of orders while they are being fulfilled. This tracking information is kept by `ProcessManager` and the `PurchaseOrderLocal` enterprise beans.

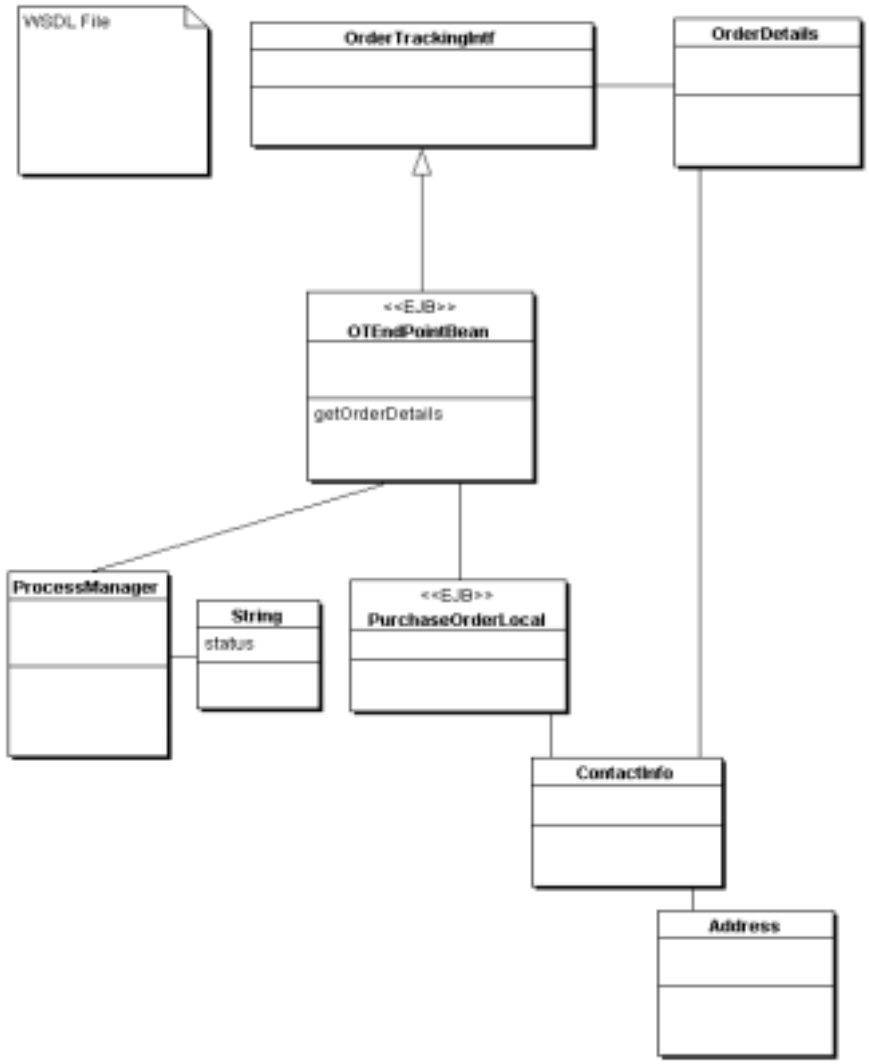

Figure 8.5 Order Tracking Endpoint UML Diagram

8.3.1.1 Order Tracker Client-Side Design

The client side of the order tracker module consists principally of a Web site to which we have added a page from which a customer can track an order. When designing the client, we first decided how customers would invoke the service. We

had to choose an invocation style of either stubs, proxies, or dynamic invocation (DII).

Since the decision on the invocation style requires some knowledge of the contents of the Web service's WSDL file, it is first necessary to locate the WSDL file. Generally, it is best to either generate stubs or have dynamic proxies for the client to use to invoke the service, rather that use the DII approach, which may be more complex and slower. We decided to use stubs for our client, since the stubs approach is easiest to use. Later, when we enhance the client, we intend to change to the dynamic proxy approach, since it provides for greater portability.

After we selected the stubs invocation style, we built the client according to this style. As much as possible, we used JAX-RPC runtime tools (such as those available with the J2EE 1.4 SDK) to generate the required interfaces and artifact classes (stubs and ties). 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. The order tracker client has the classes and interfaces shown in Figure 8.6.

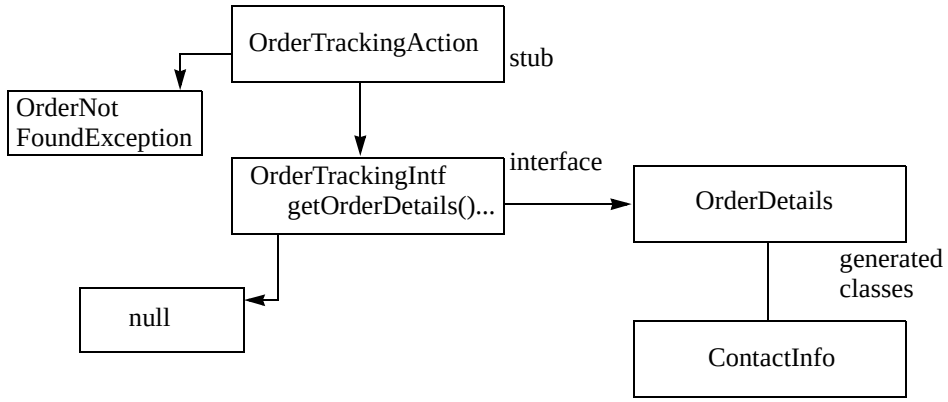

Figure 8.6 Order Tracker Client Classes and Interfaces

The client of the order tracker interface receives details about a found order in the `OrderDetails` Java object. The client extracts the data from the object and populates a JSP with the order details. The Web application displays the order information from the JSP to the end user.

Currently, the service throws standard JAX-RPC exceptions if a system error occurs, although at a later point it will throw service-specific exceptions. If an order is not found, then the service endpoint returns `null`. The client application, when it receives a null, creates the `OrderNotFoundException` application exception, which it returns to the Web application. The Web application displays an appropriate message to the end user.

- Although not recommended, the service endpoint currently returns `null` if an order is not found. Ideally, the service endpoint should throw a service-specific exception, such as `OrderNotFoundException`, to indicate this error condition. It is better to throw a service-specific exception because such an exception can be described in the WSDL document, which increases the service's interoperability.