CHAPTER 6

Enterprise Application Integration

ENTERPRISE information systems—the collection of relational and legacy database systems, enterprise resource planning (ERP) systems, and mainframe transaction processing systems—provide the critical information infrastructure for an enterprise's business processes. These varied systems hold the information that an enterprise needs to carry out its daily operations. It is essential that new applications developed for an enterprise be able to integrate with these enterprise information systems (EIS).

EIS integration has always been of great importance, and this has given rise to enterprise application integration, or EAI. EAI enables an enterprise to integrate its existing applications and systems, plus it enables the addition of new technologies and applications. Enterprises must leverage their existing systems and resources even as they adopt new technologies. Considering the cost already invested in these existing systems, no business can afford to discard them. Plus, since these systems often contain valuable data needed by the enterprise, the enterprise is not likely to disrupt them. Yet, at the same time, enterprises continually grow and require new applications. To keep their businesses growing and to remain cost effective, enterprises must integrate their existing systems with these new applications, and not replace existing systems with new applications written from scratch. The emergence of Web-based architectures and Web services adds impetus for enterprises to integrate their EISs and expose them to the Web.

The emergence of the Web and Web services is not the only factor driving the need for integration. More and more, enterprises are either merging or acquiring

other enterprises. Such mergers and acquisitions usually entail merging and combining two divergent information technology (IT) systems. Not only are the IT systems different, but, as a further challenge, they each may have standardized on using different integration technologies within their respective environments. Emerging Web services standards are another factor driving Web services-based EAI. These standards are making it possible to integrate heterogeneous systems easily and cost effectively.

In today's environment, a typical enterprise has a multitude of existing applications running on diverse platforms and operating systems. Although these applications may very well rely on the same or similar data, they keep that data in different formats. Thus, the integration problem encompasses both data and system platforms.

These are but a few examples of the complexities that enterprise application integration must address. Not only must EAI handle integrating applications, it must also address integrating data and technologies so that enterprises can easily share business processes and data.

Using several scenarios, this chapter illustrates the key integration considerations. It describes the J2EE 1.4 platform technologies that help with integration, and presents some integration design approaches and guidelines.

6.1 Integration Requirements and Scenarios

Before delving in the details of the technologies, it is helpful to illustrate the major concerns for integrating enterprise information systems and to get a sense of the extent of the EAI problem. We discuss three types of integration scenarios: data integration, application integration, and business process integration. Often, an enterprise's integration needs span these different types.

6.1.1 Typical Integration Scenarios

Data integration involves integrating existing data living in different enterprise systems, and it often occurs when an enterprise relies on multiple types of database systems. For example, some database systems may be relational, others may be hierarchical or based on objects, and still others may be file based or even legacy stores. As an illustration, a newly-developed Web-based order management system might have to integrate with an existing customer order database. Data integration involves not only integrating different data systems, but it also entails integrating different informational or data models.

Application integration involves integrating new applications with existing or legacy applications. Since an enterprise's business relies on the continuity of its existing applications, it is important to integrate the new applications with minimal disruptions. Integrating with home-grown legacy applications presents a bigger challenge, since these systems have no vendor-provided or off-the-shelf adapter layers. Not only do you have to write the adapter layers yourself, these home-grown applications may be more idiosyncratic, with an architecture that is more opaque and difficult to understand.

Business process integration involves integrating an enterprise's existing systems to support a set of business processes. A *business process* is a series of (often asynchronous) steps that together complete a business activity. For example, the Adventure Builder enterprise has a business process for fulfilling purchase orders submitted to its order processing center. This order fulfillment business process includes such steps as validating a customer's credit card, communicating with various suppliers to fill different parts of an order, and notifying the customer of the order status at various stages of processing. This integration is often accomplished by exchanging documents, which more and more are XML documents, among business processes according to defined business rules. The different processes transform the documents by applying their individual business rules, and then they route the documents to other processes. (These business processes and rules are often referred to as orchestration. Within an orchestration, an individual business process implements a choreography to accomplish a unit of useful work.) Good examples are those business processes that do such things as handle purchase orders and invoices, or incorporate supplier catalogs. Each such business process runs a workflow that interacts with other entities, either internal or external.

6.1.2 Example Integration Scenarios

Examining some typical scenarios helps to bring these integration requirements into better perspective, and we use the scenarios present in our Adventure Builder enterprise example for illustration. One common integration scenario is that of a travel agency business, such as Adventure Builder, that wants to make its inventory available online to expand its customer reach. The agency already has existing applications and databases in place for its business—its catalog, inventory, order processing, and customer relationship management (CRM) systems, for example—and these need to be enhanced to accommodate the e-store. Adventure Builder purchased and customized the CRM package, but its order processing system is home

grown. As much as possible, the travel agency wants to reduce software duplication and keep its infrastructure costs to a minimum. To that end, it wants to use these same systems to handle the e-store business, especially since it expects its existing customers to also buy products through the online store. Adventure Builder also wants to leverage its customer service department and have these same specialists service both the store front as well as the online customers. However, it does expect to include additional databases relevant for the Web site only.

This scenario illustrates an application and data integration problem. The enterprise's existing databases store information needed by the e-store, which may need to update these databases. The databases have existing security settings, plus protocols for transactions. The vendor for the CRM system has provided a J2EE connector that can be used to plug the CRM system into the enterprise's J2EE application server. However, the order processing system, since it is home grown, has no such support.

Figure 6.1 illustrates the architecture of this application.

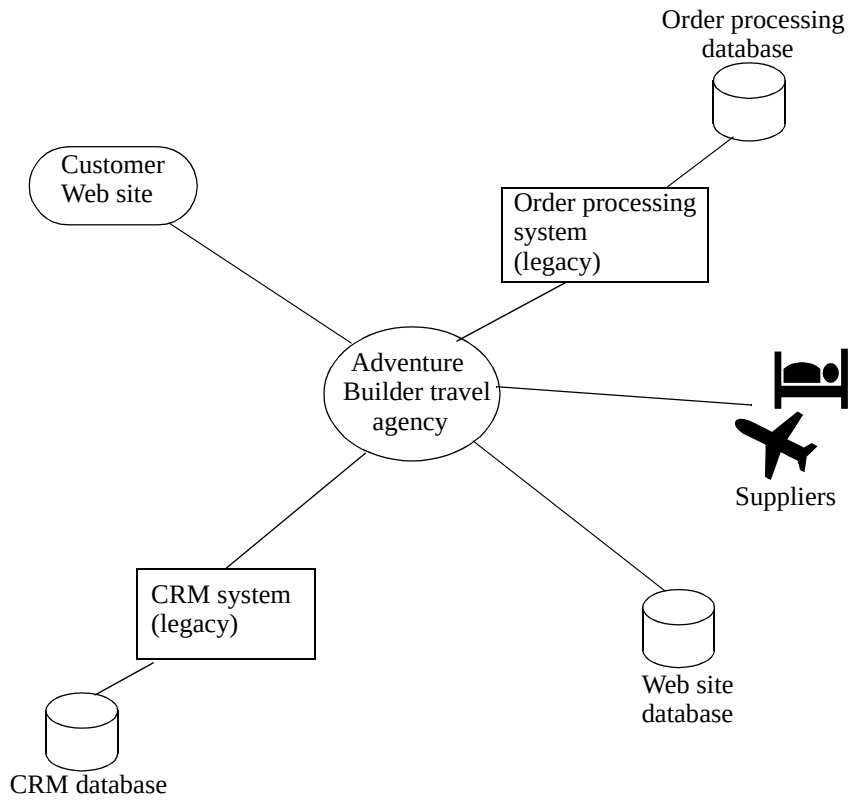

Figure 6.1 An Application Scenario

Although depicting an estore, this example scenario touches upon many of the integration requirements that pertain to all enterprises. Because an e-store is open to anyone on the Internet, it may potentially have a large number of users, making scalability and performance important issues. Security is an important consideration, since Adventure Builder's Web site handles customer data that it must keep private. The enterprise has the further challenge of ensuring that its legacy systems are not stretched beyond their capabilities, and that heterogeneous platforms may host the EIS systems.

A different form of this scenario may have an enterprise that relies on an order fulfillment center to process orders placed through the e-store Web site. A separate department owns the order fulfillment center, and it uses its own set of databases separate from the e-store's Web site. To keep the two data models

decoupled, orders flowing between the e-store and the order fulfillment center are kept in XML format. Communication is also asynchronous, allowing clients to continue their own work while an order is processed. In essence, the order fulfillment center runs a business process whenever it receives an order. The business process, following a set of automated rules, interacts with several systems in a workflow to complete or fulfill the order with no human intervention. Part of the workflow includes sending confirming e-mails to customers and keeping records for administrative reports.

In another variation to this scenario, the order fulfillment center might interact with external business partners to complete its workflow. For example, the order fulfillment center might rely on a separate credit card billing service enterprise to process its customer billing. Some of its products may be supplied directly by another vendor. The order fulfillment process initiates multiple business processes at each of the external businesses with which it interacts.

A human resources application, designed for internal use only, may have a similar scenario to an e-store, since it concerns a new application that uses existing enterprise assets. However, the application's internal use limits the scalability, performance, and security concerns. The principal concerns with internal applications such as this are fast delivery time (since IT budgets are small), platform heterogeneity, and the capacity to grow the application and support multiple types of clients as the enterprise expands. The application may also want single sign-on for users across various security domains, but at the same, permit access only to employees with the proper access privileges as expressed by company-wide rules.

Internal applications may also provide some limited mobile functionality; that is, they need to be accessible from mobile devices such as PDAs and cell-phones. For example, a travel expense tracking application may want to allow employees who are on the road to keep track of their expense records. This also applies to enterprises whose departments are geographically scattered, since it may be more efficient for employees to use the Internet for internal communications. Web service interfaces are particularly useful in these situations, although there are the additional concerns of distributed and multiple security domains.

Often, systems developed for internal use rely on home-grown legacy systems. While systems bought from third party vendors may be integrated with standard connectors supplied by their vendors, this is not true for home-grown systems. These home-grown, or one-off, applications must still be integrated with other applications in the enterprise and accessed from a Web browser.

6.2 J2EE Integration Technologies

Now that we have examined some common integration scenarios, let's look at the technologies that are available on the J2EE 1.4 platform to help with these issues. The J2EE 1.4 platform provides a set of EIS integration technologies that address the EIS integration problem. These include relational database integration technologies (such as JDBC, Enterprise JavaBean architecture container-managed persistence, and Java Data Objects), messaging technologies (such as Java Message Service and message-driven beans), EIS access technologies (particularly the Java Connector architecture), and Web services and XML technologies for manipulating documents.

Let's briefly examine some of the available integration technologies. The section "Integration Design Approaches" on page 259 maps these technologies to different integration problems, illustrating when and how to use them most effectively.

6.2.1 Relational Database Integration Technologies

Relational database management systems (RDBMS) are the most prevalent form of enterprise data store. The J2EE platform provides three technologies for integrating data in RDBMS:

- JDBC—Developers can use the JDBC APIs to access relational data in a tabular form.
- Enterprise JavaBeans container-managed persistence (CMP)—Developers use container-managed persistence to do object-relational (O/R) mapping. By mapping database table data to Java objects, developers can deal with an object view of the data rather than a tabular, relational view. CMP also encapsulates the objects into higher-level managed components with transactional and security features.
- Java Data Objects (JDO)—An O/R mapping technology that generates Java classes as opposed to components. Note that JDO is optional in the J2EE 1.4 platform. Since it is optional, your application server may not support JDO, or it may support JDO in a non-standard manner.

6.2.1.1 JDBC

The JDBC API defines a standard Java API for integration with relational database systems. A Java application uses the JDBC API for obtaining a database connection, retrieving database records, executing database queries and stored procedures, and performing other database functions.

Many application component providers use the JDBC 2.0 or 3.0 API for accessing relational databases to manage persistent data for their applications.

The JDBC API has two parts: a client API for direct use by developers to access relational databases and a standard, system-level contract between J2EE servers and JDBC drivers for supporting connection pooling and transactions. Developers do not use the contract between J2EE servers and JDBC drivers directly. Rather, J2EE server vendors use this contract to provide pooling and transaction services to J2EE components automatically. Note that, according to the JDBC 3.0 specification, the JDBC system-level contracts can be the same as the Connector architecture system contracts. Conceptually, JDBC drivers are pluggable resource adapters.

An application component provider uses the JDBC client-level API for such operations as obtaining a database connection, retrieving database records, executing queries and stored procedures, and performing other database functions.

6.2.1.2 EJB Container-Managed Persistence

Container-managed persistence (CMP), which is a resource manager–independent data access API for entity beans, has been expanded and enhanced in the J2EE 1.4 platform. CMP technology enables applications to be easily integrated with various databases or resource managers, plus it enhances portability.

CMP shields the developer from the details of the underlying data store. The developer does not need to know how to persist or retrieve data to or from a particular data store, since the EJB container handles these tasks. Instead, the developer need only indicate what data or state needs to be stored persistently.

In addition, a developer uses the same API—that is, the EJB CMP methods—regardless of the underlying type of resource manager. The same entity bean can thus be used with any type of resource manager or database schema. The technology makes it possible to develop enterprise beans that can be customized at deployment to work with existing data. That is, the same bean implementation can be deployed to work with many different customer data schemes. The mapping done at deployment may vary for each customer set up, but the bean itself is the same. Since the EJB container generates suitable data access code for each situa-

tion, the bean developer does not have to know or care about the underlying resource manager-specific code. Furthermore, since it has complete control over managing persistence, the container can optimize database access for better performance.

The J2EE 1.4 platform includes the most up-to-date EJB specification and CMP technology. Rather than declaring persistent variables in a bean's implementation, developers include get and set accessor methods for persistent variables, similar to JavaBeans properties. These accessor methods are also `abstract` methods, so the bean does not provide an implementation for them. Instead, the EJB container provides the implementations for these methods.

The CMP architecture also includes container-managed relationships, which allows multiple entity beans to have relationships among themselves. Container-managed relationships are handled in much the same way as container-managed persistence. The bean implementation merely provides get and set accessor methods for these fields, and the container provides the method implementations. Similarly, the developer specifies the relationships in the deployment descriptor.

6.2.1.3 Java Data Objects

Java Data Objects (JDO) is an API that provides a standard, interface-based Java model abstraction of persistence. Application developers can use the JDO API to directly store Java domain model instances into a persistent store (a database). You may consider JDO as one alternative to using JDBC or enterprise beans with container-managed persistence.

There are some benefits to using JDO. Since JDO keeps applications independent or insulated from the underlying database, application developers can focus on their domain object model and not have to be concerned with the persistence details, such as the field-by-field storage of objects. JDO also ensures that the application uses the optimal data access patterns for best performance.

6.2.2 Messaging Technologies

Messaging systems allow unrelated applications to communicate asynchronously and reliably. They are often based on an architecture in which the individual applications communicate on a peer-to-peer basis. Not only do the communicating parties not have to be closely tied to each other, they can also remain relatively anonymous.

The J2EE platform provides the Java Message Service (JMS) API, which is a standard Java API defined for enterprise messaging systems. Along with JMS, the

J2EE platform also provides message-driven beans. Message-driven beans are EJB components that process asynchronous messages delivered via JMS or some other messaging system.

Although messaging systems provide many of the same EAI advantages as Web services, Web services go a step further. Principally, Web services support multiple vendors and the ability to go through firewalls using Internet standards. Web services also support a flexible XML format. Since Web services standards are still evolving, it is best to use an integration architecture such as that proposed in “Integration Design Approaches” on page 259.

6.2.2.1 Overview of Messaging Technologies

Prior to the advent of Web services, developers often chose messaging systems (called MOM for Message Oriented Middleware) to create an integration architecture. With a messaging system, two systems can communicate with each other by sending messages. Such messages, which are delivered asynchronously, typically consist of two parts: one part—the message body—contains the business data and the other part—the message header—contains routing information. Since messages are sent asynchronously, the sender does not have to wait for the message to be delivered to the receiver.

There are two common messaging styles: point-to-point and publish and subscribe. A point-to-point messaging style is used when messages are sent to only one receiver. The recipient receives messages sent to it through a queue specifically set up for the receiver. A message sender sends messages to this queue, and the recipient retrieves (and removes) its messages from the queue. Publish and subscribe, on the other hand, is intended to be used when there can be multiple recipients of a message. Rather than a queue, this style uses a topic. Messages are sent—or, more correctly, published—to the topic, and all receivers interested in these messages subscribe to the topic. Any message published to a topic can be received by any receiver that has subscribed to the topic. See Figure 6.2

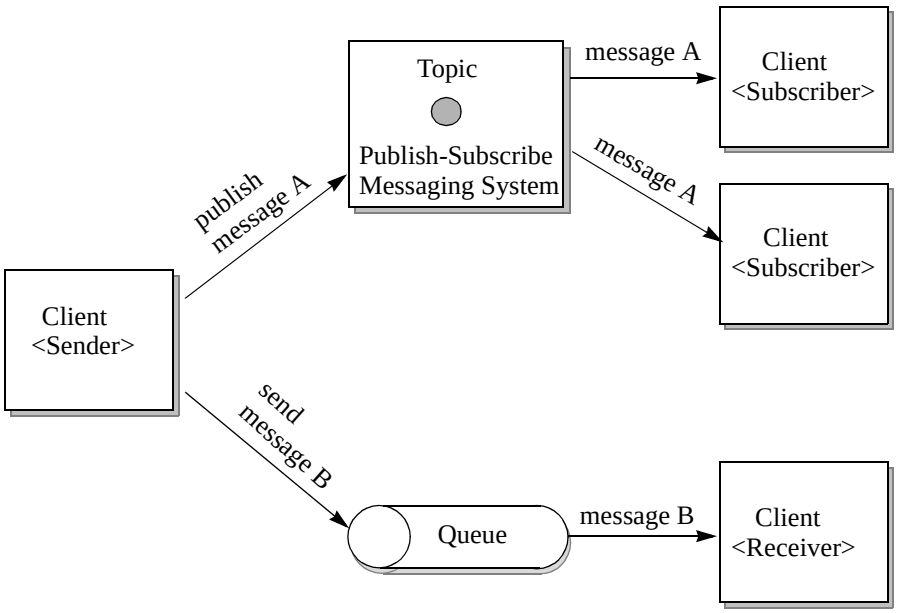

Figure 6.2 Messaging System Queues and Topics

In addition, a typical MOM system has a message router that is responsible for accepting messages from the sender and then ensuring message delivery according to the agreed-upon quality of service to the receiver. The message router uses the message header information to determine where and how to route the message contents.

When used for integration, an enterprise very likely requires that all participating EIS systems communicate with each other by sending messages via the messaging system. As a result, enterprises typically standardize on one vendor for their MOM system, and they use that vendor's adapters to accommodate their various EISs. In return, enterprises using messaging systems gain the benefits of asynchronous messaging calls: messages are queued and delivered when the target system is available without constraining the sending system.

Furthermore, messaging technology is considered a fairly mature technology. Most MOM systems provide a number of quality of service features, such as reliable once and only once delivery, load balancing, scalability, fault tolerance, and transactional support.

However, the proprietary nature of MOM systems results in some significant disadvantages. Since they use proprietary protocols over the network, it is usually more difficult to mix and match MOM products from different vendors. Many MOM systems also force developers to use their proprietary APIs to send and receive messages. As a result, application developers must customize their applications for different MOM systems.

6.2.2.2 Java Message Service

The Java Message Service (JMS) API, a standard Java API defined for enterprise messaging systems, can be used across different types of messaging systems. A Java application uses the JMS API to connect to an enterprise messaging system. Once connected, the application uses the facilities of the underlying enterprise messaging system (through the API) to create messages and to communicate asynchronously with one or more peer applications.

For the J2EE 1.4 platform, JMS includes some enhancements. In particular, the addition of common interfaces enables you to use the JMS API so that it is not specific to either a point-to-point or publish-subscribe domain. A JMS provider may also use the J2EE Connector architecture to integrate more closely with the application server. (The section "EIS Access Technologies" on page 256 discusses the J2EE Connector architecture.)

In many ways, JMS is to messaging systems what JDBC is to database systems. Just as JDBC provides a standard interface to many database systems, JMS provides a standard API for MOM systems. In fact, JMS changed the proprietary nature of MOM systems by providing a standard Java API that interfaces with any MOM system. The developer now writes to this standard API rather than to individual, proprietary APIs. The J2EE platform further simplified—and made more portable—the integration of a MOM system with a J2EE application server.

Similar to its support for JDBC, the J2EE platform has added support to JMS for a connection-oriented operational style: developers can look up a factory and and a connection in the same way. Like JDBC, JMS supports transaction and can continue a JTA transaction started by either a Web or EJB component.

Since messaging systems such as JMS decouple the sender and receiver, these communicating parties can run on different hardware and software platforms. The asynchronicity of these systems means that communicating applications need not be currently running to receive messages. This protects the communicating applications from system failures and other types of partial outages, conditions that are not uncommon in network situations.

JMS and other messaging systems also bring a dynamic quality to EIS systems. Components can be added or removed to the network without affecting other systems. Systems do not need to have their throughputs perfectly match since they can interact with the messaging system at their own pace. For example, if one application sends messages more rapidly than the receiving application can retrieve these messages, the messaging system keeps these messages in the queue indefinitely. As a result, good overall throughput is achieved since each part of the system works at its own optimum capacity.

6.2.2.3 Message-Driven Beans

Message-driven beans, which are EJB components that receive incoming enterprise messages from a messaging provider, contain the logic for processing these messages. The business logic of a message-driven bean—which may include initiating a workflow, performing a computation, or sending a message—may be driven by the contents of the message itself or merely the receipt of the message.

Message-driven beans are particularly useful in situations where messages need to be automatically delivered and asynchronous messaging is desired. They enable applications to be integrated in a loosely-coupled, but still reliable, fashion. Message-driven beans are also useful when the delivery of a message should be the event initiating a workflow process, or when a specific message must trigger a subsequent action.

When an application produces and sends a message to a particular message destination, the EJB container activates the correct type of message-driven bean (from a pool of available message-driven beans). The activated bean instance consumes the message from the message destination. Since they are stateless, any instance of a matching type of message-driven bean can handle the message.

Implementing a message-driven bean is straightforward. The bean developer extends the `javax.ejb.MessageDrivenBean` interface, which includes the bean's lifecycle methods. The developer also extends a message listener interface for the bean—the message listener interface corresponds to the enterprise's messaging system. For enterprises using JMS, the developer extends the `javax.jms.MessageListener` interface.

The methods of the message listener interface are of most interest to the developer, since the developer embeds the business logic to handle particular messages within these methods. For example, when a message-driven bean consumes JMS messages, the developer codes the message-handling business logic within the `onMessage` method. When a message appropriate for this bean arrives at a

message destination, the EJB container invokes the message-driven bean's `onMessage` method.

With message-driven beans, you can make the invocation of the bean part of a transaction. That is, you can ensure that the message delivery from the message destination to the message-driven bean is part of any subsequent transactional work initiated by the bean's logic. If problems occur with the subsequent logic causing the transaction to roll back, the message delivery also rolls back—and the message is available in the destination for another message-driven bean instance to pick up.

6.2.3 EIS Access Technologies

The J2EE 1.4 platform includes the J2EE Connector architecture, a technology designed specifically for accessing enterprise information systems (EISs). The Connector architecture simplifies integrating diverse EISs into the platform, since each EIS can use the same, single resource adapter to integrate with all compliant J2EE servers.

The J2EE Connector architecture provides a standard architecture for integrating J2EE applications with existing EISs and applications, and particularly for data integration with non-relational databases. The Connector architecture enables adapters for external EISs to be plugged into the J2EE application server. Enterprise applications can use these adapters to support and manage secure, transactional, and scalable integration with EISs. The EIS vendor knows that its adapter will work with all J2EE-compliant application servers, and the compliant J2EE server can connect to multiple EISs. See Figure 6.3.

The earlier version of the Connector architecture focused on synchronous integration with EISs, while the current version under development as part of J2EE 1.4 extends this core functionality to support asynchronous integration. That is, it supports both outbound and inbound message-driven integration that is protocol independent.

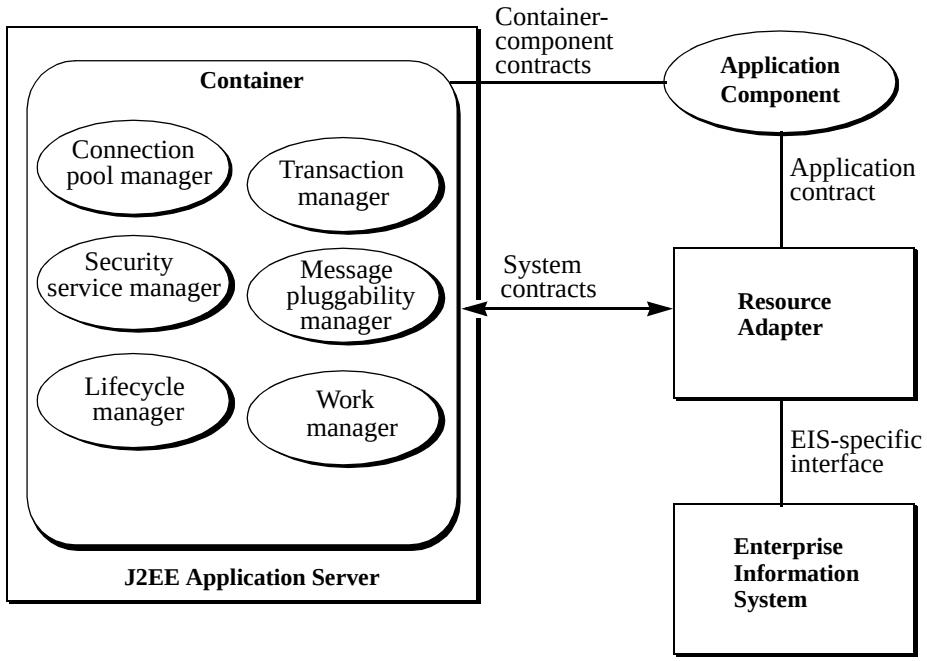

Figure 6.3 Connector Architecture System and Contracts

A *resource adapter* is a J2EE component that implements the J2EE Connector architecture for a specific EIS. Communication between a J2EE application and an EIS occurs through this resource adapter. In a sense, think of a resource adapter as analogous to a JDBC driver, since it provides a standard API for access between the J2EE server and the external resource.

The J2EE Connector architecture, through its contracts, establishes a set of programming design guidelines for EIS integration. The J2EE application server and an EIS resource adapter collaborate, through a set of system-level contracts, to keep security, transaction, and connection mechanisms transparent to the application components. Application components and resource adapters rely on the application-level contract for their communication.

The application-level contract defines the API used by a client to access a resource adapter for an EIS. This API may be the Common Client Interface (CCI), which is a generic API for accessing multiple heterogeneous EISs, or it may a resource adapter-specific API.

The initial release of the Connector architecture, which is part of the J2EE 1.3 platform, established three system-level contracts, as follows:

- Connection management contract—Supports connection pooling to an underlying EIS, an important requirement for scalable applications.
- Transaction management contract—Supports local and global transactions, and enables management of (global) transactions across multiple EISs.
- Security management contract—Enables secure interchanges between an EIS and a J2EE application server and protects EIS-managed resources.

The most recent release (version 1.5) of the Connector architecture expanded the capabilities of resource adapters. This release expanded the transaction support for a resource adapter. Previously, a transaction had to start from an enterprise bean on the J2EE application server, and it remained in force during operations on the EIS. Now, transactions can start on the EIS and be imported to the application server. The J2EE Connector architecture specifies how to propagate the transaction context from the EIS to the application server. The Connector architecture also specifies the container's role in completing a transaction and how it should handle data recovery after a crash.

The current version also specifies additional system-level contracts to the initial three contracts just noted. These new contracts are:

- Messaging "pluggability" contract—Extends the capabilities of message-driven beans so that they can handle messages from any message provider rather than being limited to handling only JMS messages. By following the APIs for message handling and delivery specified in this contract, an EIS or message provider can plug its own custom or proprietary message provider into a J2EE container. JAXM is a good example of this type of message provider.
- Work management contract—Enables the J2EE application server to manage threads for resource adapter, ensuring that resource adapters use threads properly. When a resource adapter improperly uses threads, such as when it creates too many threads or fails to release threads, it can cause problems for the entire application server environment. Poor thread handling by a resource adapter can inhibit server shutdown and impact performance. To alleviate this problem, the work management contract enables the application server to pool and reuse threads, similar to pooling and reusing connections.

In addition, the work management contract gives resource adapters more flexibility for using threads. The resource adapter can specify the execution context for a thread. The contract allows a requesting thread to block—stop its own execution—until a work thread completes. Or, a requesting thread can block while it waits to get a work thread; when the application server provides a work thread, both the requesting and work threads execute in parallel. Yet another option, a resource adapter can submit the work for the thread to a queue and have it execute at some later point; the adapter continues its own execution without waiting further. Thus, a resource adapter and a thread may execute in conjunction with each other or independently, using a listener mechanism if need be to notify the resource adapter that the work thread has completed.

- Lifecycle management contract—Enables an application server to manage the lifecycle of a resource adapter. With this contract, the application server has a mechanism to bootstrap a resource adapter instance at deployment or at server startup, and to notify the adapter instance when it is undeployed or when the server is shutting down.

6.2.4 Web Service and XML Technologies

The J2EE 1.4 platform provides a rich set of Web service APIs and XML document-manipulating technologies. The Web service APIs—JAX-RPC, JAXR, SAAJ, JAXB, and JAXP—provide a standard Java API for integrating applications and other systems. A Java application can use these APIs to obtain and use a Web service. These APIs are particularly useful when an application or system exposes a Web service layer explicitly for integration purposes.

The XML, XSLT, and XML schema APIs define standard Java APIs for integration data models. They permit you to define an interface data model, manipulate disparate documents, and perform transformations between data types. These XML APIs give you the ability to structure a data model for passing data among different systems.

Chapter 2 discusses these technologies in greater depth.

6.3 Integration Design Approaches

The J2EE 1.4 platform provides several technologies for integration, and you can combine them to fit your integration requirements. These technologies, together with the other platform capabilities, give you a rich platform for designing an integration solution. IT architects face the challenge of combining the technologies in

the most effective and flexible manner possible to create an integration architecture that can adapt to changing business needs and strategies.

Often, enterprises attempt integration as a "one-off" solution—that is, they treat each integration problem as a "one-off" instance with its own special solution. With this type of approach, an enterprise integrates each system in an individual, custom manner—not the most efficient or effective integration approach. It is far better to have an integration architecture that grows with the needs of the enterprise. The J2EE platform, with its extensive support for Web services, makes such an architecture possible. A J2EE application server acts as the universal connector between the EIS systems and new EAI applications. In a sense, the J2EE application server is the integration hub. The data model in the application server becomes the canonical agreed-upon data model to which all others—EISs and new EAI applications—adapt.

A good integration architecture consists of a set of "integration layers" each of which provide certain quality of services. By integration layer we mean the interface or endpoint at which distinct systems intersect. Integration architects must decide how to implement these various integration layers in the most suitable manner. Generally, integration architects first consider the requirements of each situation. For example, the architects of the Adventure Builder enterprise chose Web services as the integration layer for its supply chain. As a result, all interactions with its suppliers must happen through the Web service interfaces.

Integration layers serve two purposes: they expose the existing EIS systems and they provide a fixed standard against which new applications are programmed. That is, the integration layer defines the interface between the EIS systems and new applications. While both the new application and the existing EIS may change, the integration layer should remain the same. An integration layer needs to be the stable point in your system.

In some situations, the integration approach is fairly obvious. For example, when a client requires a tightly-coupled interaction with a relational database, clearly the JDBC API is the design strategy to use. Similarly, another well-known design strategy recommends using the J2EE Connector architecture for data integration with non-relational databases. These two design strategies are particularly appropriate when the client requires an API for connecting, querying, and updating a database with transactional semantics.

Other situations require you to combine technologies to meet the integration requirements. You might use a Connector architecture layer on top of a non-relational database, but then, based on the client requirements, you might add a Web service layer on top of the connector. Still other situations might necessitate using

a Web service-based message provider that provides such services as store-and-forward capabilities, message delivery guarantees, and so forth.

It is helpful to view this as three separate tasks (each requiring a design and implementation phase):

- Decide on the integration layer, including where it should be located and what form it should take. With the J2EE platform, various integration layers are possible, including data access objects (DAO) and connectors, enterprise beans and JMS, and Web services. Remember that you want the integration layer to be such that other applications or systems can evolve easily.
- Decide how to adapt each EIS to the integration layer
- Decide how to write new applications against these integration layers

Figure 6.4 shows the integration design approaches possible with the J2EE technologies. In the next sections we take a closer look at these different EAI design approaches, and then provide some guidelines for implementing these approaches.

.

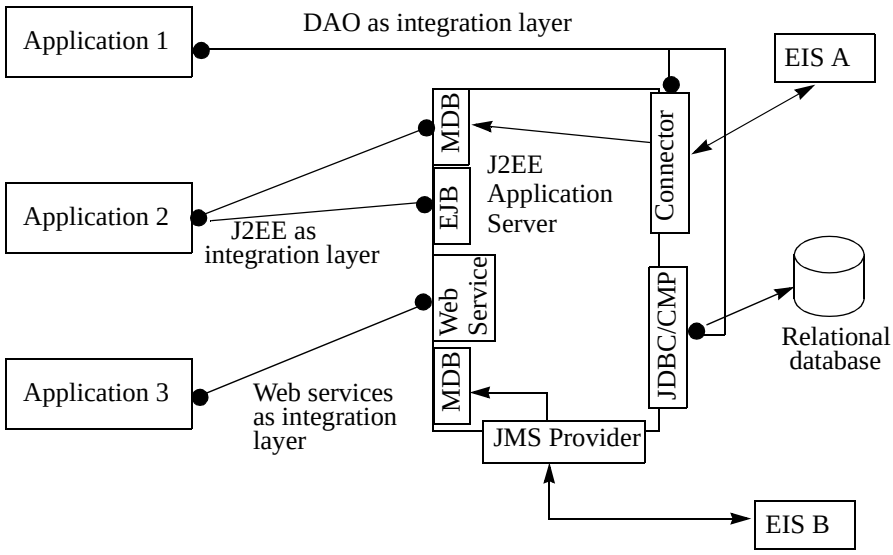

Figure 6.4 EIS Integration Design Approaches

Let's see how you might apply this strategy to the Adventure Builder enterprise. The Adventure Builder enterprise decides to use two layers for integration:

1. Web services as an integration layer for its supply chain.
2. Web services as an integration layer for communicating among different departments. For example, the Adventure Builder Web site uses a Web service to send an order to the order processing center. The order processing center integrates EISs within its department using a J2EE application server as the integration layer. Hence, the order processing center fulfills an order using JMS and EJB technologies for integrating its various EIS systems, customer relations management, billing systems, and so forth.

6.3.1 Web Services Approach

One approach for EAI is to use a Web service as the integration layer. With this approach, an enterprise's EIS systems expose their functionality by implementing Web services. They make their Web service interfaces available to other applications by providing WSDL descriptions of them. In addition, the integration layer may also include XML schemas for the documents used in any parameters and return values. Essentially, the WSDL description of the service interface and document schemas becomes the integration layer, or the point of stability.

This approach lets developers leverage the advantages of Web services. They can write new enterprise applications with any technology that supports Web services, and the applications may be run on a variety of hardware and software platforms. However, this approach falls short when the new applications have certain additional requirements, such as transactional semantics or stringent security requirements. We discuss how to handle these issues in subsequent sections.

Adventure Builder is a good example of the Web services approach. As noted, the Adventure Builder enterprise uses Web services for integrating its supply chain. (Chapter 11 discuss the exact structure of the application, but here we highlight details that pertain to integration.) The Adventure Builder architects decide, in consultation with the suppliers, on the schemas for the documents that they intend to exchange. (See "Designing Domain-Specific XML Schemas" on page 182.) They define the documents shown in Code Example 6.1 for their purchase order and invoice.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<Invoice>
```

```
We will fill in the real invoice (or purchase order) later...
</Invoice>
```

Code Example 6.1 Supplier Invoice

Similarly, the architects standardize on the WSDL to use for invoices when fulfilling an order. Code Example 6.2 shows the WSDL for the Web service endpoint to which the suppliers send their invoices.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
We will fill in the WSDL file for receiving invoices later
</>
```

Code Example 6.2 WSDL for Receiving the Invoices

It is important to keep the WSDL and the XML documents stable, since suppliers (such as airlines, hotels, and activity providers) use them for their interaction with the Adventure Builder enterprise.

With this approach, you create Web service interfaces for those EIS systems that participate in Web service interactions. A simple—albeit naive—way to do this is to modify the EIS to directly export a Web service interface. However, such a modification may require the EIS to maintain an internal SOAP and HTTP stack, and this may prove too intrusive and disruptive for some EISs.

Using a connector to plug the EIS system into the J2EE application server is a better approach, since it takes advantage of the capabilities of the J2EE server. (The connector by its nature does not intrude on the EIS.) Then, from the J2EE application server, use JAX-RPC to expose a SOAP layer. For example, the Adventure Builder enterprise wants to use Web services to integrate its existing CRM system, which provides services to manage customer relations, to process orders. The department that owns the CRM module not only wants to maintain control of the software, it wants to use a generic interface that can handle user requests from multiple sources (Web site, telephone, OEM channels, and so forth). Web services are the best way to create such a generic interface, for these reasons:

- Web services establish clear, defined boundaries—Since Web services can provide an interface with clear, defined boundaries of responsibilities, the

CRM department has the responsibility to only maintain the endpoint and publish a WSDL describing the endpoint.

- Web services provide controlled access—Outside requests to the CRM must come in through the service interface, and the CRM department can apply its access control parameters to limit access, plus it can log each access.
- Web services support multiple platforms—Because it does not control the hardware and software platforms other departments use, the CRM department can accommodate any platform by using Web services.
- Web services are best suited for applications with a limited need for transactions and security—The main purpose of the CRM system is to allow status queries on existing orders. As such, it has little need for transactions. It also has limited need for security, since all access to the module happens within the corporate firewall.

Developers have a choice of endpoint types for implementing the Web service representing the EIS: either an EJB service endpoint or a Web-based JAX-RPC service endpoint. (Chapter 3 discusses the relative merits of these endpoints.) Solely from an integration point of view, either type of endpoint works well since Web and EJB components can both directly use JDBC and J2EE connectors. For example, the Adventure Builder enterprise receives invoices using a broker implemented as an EJB service endpoint. (See Chapter 11.)

Returning to the CRM problem, the Adventure Builder architects devised the following solution: they developed a small J2EE application that uses a J2EE connector to connect to the CRM system. They then implemented a JAX-RPC endpoint to expose the required Web service functionality.

Today, since many EIS vendors are providing built-in Web services support, developers can leverage this support directly to avoid writing a JAX-RPC or SOAP interface and using connectors. If you choose to follow this path, ensure that the EIS vendor provides a WS-I basic profile-compatible Web service. If the vendor does not, consider writing an adapter layer yourself to handle any differences.

6.3.2 Using Enterprise Beans and JMS

You can use enterprise beans and JMS layers, separately or combined, to develop an integration interface. With this approach, you use a J2EE application server to provide an enterprise bean layer for synchronous interactions. For asynchronous

interactions, you place a message bus in the enterprise and require that applications use it.

The Adventure Builder enterprise uses this strategy for integrating applications within one department. For example, a single department owns the order processing module. Within that department, different groups handle various aspects of order processing, such as credit card payments, supply chain interactions, customer relations, and so on. By using a message bus, interactions among these departmental groups is handled in a loosely-coupled, asynchronous manner. When it needs to provide synchronous access, a group may use a remote enterprise bean interface.

This approach also benefits by requiring that all messages be exchanged in XML format. For example, the Adventure Builder enterprise can use the same invoice document listed in Code Example 6.1 when sending a JMS message within its order processing center. The message-driven bean that receives the JMS message can apply XML validation and translation logic just like any Web service endpoint.

6.3.3 DAO and Connector Approach

The EIS itself can be the integration layer, which is the case when you use a connector or a data access objects (DAO). When using this approach, the new enterprise application is programmed directly against an interface provided by the EIS. For new applications that are also J2EE applications, you use a connector to access the EIS. You can either buy an off-the-shelf connector or write your own.

If you want to provide a simple isolation layer, you should consider writing a DAO class to hide the internal details of the EIS. A DAO class can also expose an API that is easier to use. However, in general, with this approach the new application is tightly coupled with the EIS since the application directly uses the data model and functionality of the EIS.

Using a DAO class or a connector reduces the complexity of the integration layer. Since there is minimal layering, it also increases performance. However, it is an approach that works best when the integration problem is small in scope. Since it does not put into place a true integration architecture, this approach may limit the ability to scale the integration layer as the enterprise grows. Given these advantages and disadvantages, consider using DAO classes or connectors as the basic building blocks for other strategies. The Adventure Builder application follows this style, using DAO classes and connectors as building blocks for the enterprise beans with JMS and Web services approaches.

6.3.4 Combining Approaches into an Integration Architecture

In many cases, architects combine these various integration layers into a single integration architecture. The end result—the mix of integration layer types—is in large measure driven by the requirements of each enterprise's situation. Architects need to consider the realities of the current state of the technologies and weigh that against future promises.

For example, Web services, as they exist today, have some shortcomings: they do not deliver the heavy-duty process integration, data transformation, scalability, and security capabilities required by many EAI scenarios. Similarly, services for transactional integrity and reliable messaging are not yet in place. Since security and transactional context propagation are critical business requirements, these are important factors to consider.

Data binding is another issue to consider. Web services are a good solution when data binding requirements are straightforward, such as mapping simple data types to Java data types. However, when it is necessary to manipulate complex relational or binary from an EIS, you may want to consider other solutions, such as using the J2EE Connector architecture, which provides a metadata facility to dynamically discover the data format in the EIS.

Table 6.1 compares these approaches for different integration problems, and indicates the approaches best suited for these integration problems. The Adventure Builder enterprise uses Web services for partner and inter-department interactions, and the enterprise beans with JMS (EJB/JMS) approach for intra-department interactions.

Table 6.1 Comparing Different Integration Approaches

	DAO-Connector Approach	EJB/JMS Approach	Web Services Approach
Coupling with EIS	Tight coupling. Uses EIS data model directly	Can add a layer of abstraction in the EJB/JMS layer	No hardware/software platform coupling. Can add multiple layers of abstractions and translations.
Supporting asynchronous operations	J2EE 1.4 adds asynchronous capabilities to connectors	MDB provides an easy to use abstraction for receiving asynchronous events from EISs	Currently no asynchronous support

Table 6.1 Comparing Different Integration Approaches

	DAO-Connector Approach	EJB/JMS Approach	Web Services Approach
Transactional Support	Available	Declarative, automatic context propagation	Currently not available
Performance	Highest	Overheads because of remote calls, and requirements of running a server	Significant overheads because of remote calls, requirements of running a server, and of XML processing and validation
Heterogeneous platform support	Requires that the client is programmed in Java	Requires a J2EE application server (available on a broad range of hardware/software platforms)	Supported on a variety of hardware/software platforms
Security features	Can directly integrate with the EISs security model	Provides app-server security mechanisms	Limited. SSL and username/password authentication support
Supporting asynchronous operations	J2EE 1.4 adds asynchronous capabilities to connectors	MDB provides an easy to use abstraction for receiving asynchronous events from EISs	Currently no asynchronous support

6.4 Data Integration Guidelines

Recall that enterprise data may be kept in various types of data stores, including relational databases, non-relational databases, directory servers, and so forth. Data mapping is one approach for integrating data from these different types of data stores, while data transformation and data filtering are other approaches.

6.4.1 Data Mapping

Since each such data source may use different data types and layouts, it is often difficult to map data from one source to another. Thus, a key issue to consider when integrating data sources is to decide on a mapping layer. Generally, you have these options for relational data:

- ❒ Create a formal object-oriented data model.
- ❒ Create a generic data-holder layer.

The formal object-oriented data model relies on object-relational mapping technologies to map data from relational data sources to an object-oriented format. You may use such mapping technologies as Enterprise JavaBeans container-managed persistence, Java Data Objects, or even the data access object pattern. There are a number of advantages to this option. For one, you can reap the traditional advantages of an object-oriented approach, notably reusability, since you establish a mapping layer that can be reused by other applications. When you use the EJB container-managed persistence technology, you also can rely on the EJB container's security features to control access to the data. You also leverage the performance benefits of the EJB container-managed persistence engine, which uses data caching to improve performance. Finally, by using these technologies, you can take advantage of the mapping tools that come with them. (See "Mapping Schemas to the Application Data Model" on page 194 for more information.)

The other option for relational data is to create a generic layer to hold the data. For this approach, you use JDBC APIs to handle data from relational data sources. The JDBC `RowSet` technology, in particular, makes it easy to access relational data in an efficient manner. The `RowSet` technology, through the `WebRowSet` feature, gives you an XML view of the data source. Its `CachedRowSet` capabilities lets you access data in a disconnected fashion, while the `FilteredRowSet` functions give you the ability to manage data in a disconnected manner, without requiring heavyweight DBMS support. By using a generic layer to hold data, you have a simpler design, since there is no real layering, and you avoid the conceptual weight of a formal data model. This approach may have better performance, particularly when data is accessed in a tabular manner.

- ❒ To access data stored in non-relational data sources, it is best to use connectors.

Connectors let you plug in non-relational data sources to the J2EE environment. For non-relational systems that do not have ready-made connectors available, such as home-grown systems, you need to write your own connector. See "Integrating Home-Grown Legacy Systems" on page 276.

6.4.2 Data Transformation

Data transformation—the ability to convert data to an acceptable application format—is a common requirement for newer applications that need to access legacy data. Such transformation functionality is necessary because most legacy systems were not designed to handle the requirements of these subsequent applications.

To illustrate, a legacy system may store dates using an eight digit integer format; for example, storing the date September 23, 2003 as `20030923`. Another application that accesses this legacy system needs the same date formatted as either `MM/DD/YYYY` or `09/23/2003`. The application needs to access the eight digit date from the legacy system and transform it to a usable format.

Another example of data transformation might involve customer data. Customer data spans a range of information, and might include identity and address information to credit and past ordering information. Different systems may be interested in different parts of this customer data, and hence each system may have a different notion of a customer.

Even schemas, including industry-standard schemas such as Electronic Data Interchange For Administration, Commerce, and Transport (EDIFact), Universal Business Language (UBL), and RosettaNet, must be transformed to other schemas. Often enterprises need to use these industry-standard formats for external communications while at the same time they use proprietary formats for internal communications.

One way you might solve the data transformation problem is to require that all systems use the same standard data format. Unfortunately, this solution is unrealistic and impractical, as illustrated by the Y2K problem of converting the representation of a calendar year from two digits to four digits. Although going from two to four digits should be a minor change, the cost to fix this problem was enormous. System architects must live with the reality that data transformations are here to stay, since different systems will inevitably have different representations of the same information.

A good strategy for data transformation is to use what is known as the canonical data model. An enterprise might set up one canonical data model for the entire enterprise or separate models for different aspects of its business. Essentially, a canonical data model is a data model independent of any application. Rather than transforming data from one application's format directly to another application's format, you transform the data from the various communicating applications to this common canonical model. You write new applications to use this common format and adapt legacy systems to the same format.

XML provides a good means to represent this canonical data model, for a number of reasons:

- XML, through a schema language, can rigorously represent types. By using XML for your canonical model, you can write various schemas that unambiguously define the data model. Before XML, you had to use a text document to describe the canonical data model types. It was easy for such a document to get out of sync with the actual types used by the model. In addition, a text document could not enforce use of the proper types.
- XML schemas are enforceable. You can validate an XML document to ensure that it conforms to the schema of the canonical data model.
- XML is easier to convert to an alternate form, even when using declarative means such as style sheets.
- XML is both platform and programming-language neutral. Hence, you can have a variety of systems use the same canonical data model without requiring a specific programming language or platform.

Code Example 6.3 shows an XML document representing contact information. This could be the canonical model of contact information used by the Adventure Builder enterprise. Since this document is published internally, all new applications requiring this data type can make use of it.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<ContactInformation
       xmlns="http://simple.example.com/CInfoXmlDoc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
           "http://simple.example.com/CInfoXmlDoc
           file:./CInfoXmlDoc.xsd">
    <Name>John doe</Name>
    <Address>
        <Street>1234, Heavenly Drive</Street>
        <City>Hell</City>
        <State>Timboktu</State>
        <Country>NoWhere</Country>
    </Address>
    <HomePhone>123-456-7890</HomePhone>
```

```
      <EMail>johndoe@nowhere.com</EMail>
</ContactInformation>
```

Code Example 6.3 XML Document With Contact Information

It is usually a good idea to provide the schema for the canonical data model. By having the schema available, you can validate the translated documents against it and newer applications can use the schema to define their own models. Code Example 6.4 shows the XSD schema file for this contact information.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://simple.example.com/CInfoXmlDoc"
    xmlns=" http://simple.example.com/CInfoXmlDoc"
    elementFormDefault="qualified">
    <xsd:element name="ContactInformation">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="Name" type="xsd:string" />
                <xsd:element name="Address">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="Street"
                                      type="xsd:string" />
                            <xsd:element name="City"
                                      type="xsd:string" />
                            <xsd:element name="State"
                                      type="xsd:string" />
                            <xsd:element name="Country"
                                      type="xsd:string" />
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
                <xsd:element name="HomePhone" type="xsd:string" />
                <xsd:element name="EMail" type="xsd:string" />
            </xsd:sequence>
        </xsd:complexType>
```

```
    </xsd:element>
</xsd:schema>
```

Code Example 6.4 XSD Schema for Contact Information

In addition to the XML form, the canonical model representation may be needed in the Java object form. Often, the Java object form is needed, for example, when a substantial amount of business logic is written in the application server. To use the canonical form for new code, you convert the canonical data types (from an XML schema to Java or vice-versa) in the data access object (DAO) layer. That is, you use the XML document or Java objects as the integration point. (See “Web Services Approach” on page 262.) For example, Code Example 6.5 shows the equivalent Java classes for the canonical model of contact information.

```
public class ContactInformation {
    public string getName();
    public Address getAddress();
    public string getHomePhone();
    public string getEMail();
    // class implementation ...
}
public class Address {
    public string getStreet();
    public string getCity();
    public string getState();
    public string getCountry();
    // class implementation ...
}
```

Code Example 6.5 Java Object Equivalents for Canonical Contact Information

After establishing a canonical data model, you must devise a strategy to convert any alternate data representations to this model. Because it plugs its enterprise systems—billing, order processing, and CRM—in via an application server, the Adventure Builder enterprise exposes the canonical data model only through the external interfaces exposed by the application server. (See “Canonical Data

Model" on page 53 for a more complete description of the Adventure Builder integration architecture.) That is, only those components with external interfaces—Web service applications, remote enterprise beans, and so forth—expose the canonical data model. Although the external world sees only the canonical data model, the Adventure Builder enterprise needs to transform its internal data representations (used by its various EISs) to this same canonical model. There are two ways to do this:

1. Use XSL style sheets to transform these alternate data representations, either when data comes in or when data goes out. XSL style sheets work for XML-based interfaces.
2. Use a transition facade approach and do programmatic object mapping in the DAO layer. That is, you set up a DAO to connect an EIS to the application server. Write the DAO so that it exposes only the canonical data model and have it map any incoming data to the appropriate internal data model.

To understand the XSL style sheet approach, let's consider how the Adventure Builder enterprise receives invoices containing contact information. In Adventure Builder's case, various suppliers submit invoices and each supplier may have a different representation (that is, a different format) of the contact information. Furthermore, Adventure Builder's various EISs may each have a different representation of the same contact information. Code Example 6.6 shows an example of a typical supplier's contact information.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<ContactInformation
       xmlns="http://simple.example.com/CInfoXmlDoc"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
           "http://simple.example.com/CInfoXmlDoc
           file:./CInfoXmlDoc.xsd">
   <FirstName>John</FirstName>
   <LastName>Doe</Last Name>
   <Address>
       <Line1>1234, Heavenly Drive</Line1>
       <Line2></Line2>
       <City>Hell</City>
       <State>Timboktu</State>
```

```
        <Country>NoWhere</Country>
    </Address>
    <Phone>123-456-7890</Phone>
    <EMail>johndoe@nowhere.com</EMail>
</ContactInformation>
```

Code Example 6.6 Example of Supplier Contact Information

Compare this listing of contact information with that of Adventure Builder's canonical model, shown in Code Example 6.3. Notice how this supplier splits the contact information format for the customer name into two elements: `<FirstName>` and `<LastName>`, whereas Adventure Builder used one element, `<Name>`. Similarly, the supplier splits the street address, which Adventure Builder calls `<Street>`, into two elements, `<Line1>` and `<Line2>`, and, for the telephone number, labels the element `<Phone>` instead of `<HomePhone>`.

Given these differences, Adventure Builder can convert an invoice from this supplier to its canonical data model by applying in the interaction layer of the Web service the style sheet shown in Code Example 6.7.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
Need to fill in more details..
    <xsl:output method="xml" indent="yes" encoding="UTF-8"/>
    <xsl:strip-space elements="*" />

    <xsl:template match="/ContactInformation">
        <HomePhone><xsl:value-of select="//OrderId" /></HomePhone>
    </xsl:template>
</xsl:stylesheet>
```

Code Example 6.7 Style Sheet to Convert Supplier Invoice to Canonical Model

The Web service message broker in Adventure Builder applies the style sheet when an invoice is sent to a supplier. See "Reuse and Pool Parsers and Style Sheets" on page 235 for more information about pooling style sheets. Chapter 11 shows how Adventure Builder uses a Web service message broker. Essentially, a

Web service message broker is useful when you want to centralize multiple document-oriented Web service interactions in your applications. The same broker can also handle the work that is done in the interaction layer for both incoming and outgoing Web service calls.

Transition facades, which can apply to any object representation of data, are a more general solution for transformations. You can use transition facades to hide extra information or to provide simple mappings. To use a facade, you write a Java class within which you do manual data transformation and mappings.

You should also consider using one of the many tools available for data transformations. These tools simplify the data transformation task and make it easier to maintain.

- ❒ When data is in an XML document, it is easier to write XSL style sheets that do the required transformations.
- ❒ When you access data using EJB container-managed persistence, you can either directly modify the container-managed persistent classes or write facades to do the required transformations.

6.4.3 Data Filtering

When you do not have access to an application's code, such as for off-the-shelf packaged applications or for applications that cannot be modified because they are critical to a working business system, you might consider using filtering, also called an adapter pattern. That is, you construct a filter that sits in front of the application and does all necessary data translations.

Generally, data filtering goes hand-in-hand with data transformations. The canonical data model, because it must support all use cases within an enterprise, is often a good candidate for filtering. Since many applications do not need access to all data fields, you can filter data and simplify application development and improve performance.

There are two types of filtering, and each has its own use cases:

1. Filtering that hides information but saves it for later use
2. Filtering that outputs only needed information

For example, in the Adventure Builder enterprise, a workflow manager receives the invoices sent by the different suppliers. Since it needs only an id field

to identify the workflow associated with the invoice, the workflow manager can filter the document to retrieve only this information. However, since it may need to pass the entire invoice—all the data fields in the invoice—to the next step in the workflow, the workflow manager must preserve the entire document. The workflow manager can accomplish this using flexible mapping. (See "Flexible Mapping" on page 199.)

On the other hand, you may want to apply data filtering before sending information. For example, a credit card processing component of a workflow may need to send credit information. The component should send only information required for privacy protection, and should use data filtering to remove information that need not be passed to another application.

Filtering can be applied at the database level, too. Using filtering, you can obtain a simplified view of the data tailored to a particular application. In addition, using EJB container-managed persistence for data transformation makes it easier to filter data. The container-managed persistence mapping tools let you select a subset of the database schema, and this is analogous to filtering. If you are using the JDBC `RowSet` approach, you need only select the columns for the data that you care about.

You can also write facades that appropriately filter the data. In this case, the client code accesses data only through these facades.

6.5 Application Integration Guidelines

Now that we've examined integration design approaches and techniques, let's look at some guidelines for integrating enterprise applications with EIS systems.

We examine guidelines for integrating legacy systems, particularly home-grown systems, and guidelines for using command beans.Although not as common, we also mention guidelines for achieving integration by through using metadata, registries, and even versioning and evolution.

6.5.1 Integrating Home-Grown Legacy Systems

Most enterprises have home-grown EISs, also referred to as one-off systems, and these systems need to be integrated. As we have already seen, connectors are the best solution for integrating EISs in a J2EE environment. However, because these are home-grown systems, there are no off-the-shelf or vendor-provided connectors

available for them, as there are for well-known EISs. As a result, you may have to write your own connector for a home-grown EIS.

- ❒ The recommended way to write a connector is to use a connector builder tool. A tool such as this not only generates a connector, but it also creates a SOAP layer.

There are times when you may want to integrate an EIS that has no programming interface. Often, this happens with legacy mainframe applications. In these cases, you can resort to "screen scraping" to create a programming interface. With screen scraping, you code your application so that it act as an end user entering data into the mainframe application, and this serves as a programming interface. You then write a connector that uses this programming interface to accomplish its integration actions.

When resorting to screen scraping, be sure to keep in mind the limitations of the legacy system. Although you can integrate a new application to a legacy system, the original system has limitations that may make it unable to handle the new functionality. For example, suppose you used screen scraping to integrate a front-end application with a mainframe application designed to be interactive. The front-end application, since it has no human constraints, can suddenly pump in a high volume of requests to the mainframe application, which may not have been designed to handle such a load. It is hard to detect a problem such as this until runtime, when the system is suddenly overwhelmed with a high volume of requests. By this point in the development cycle, it may be quite expensive to fix the application.

6.5.2 Using Command Beans

Application integration, or integrating a new application with existing EISs, varies according to the legacy system. Consider using command beans for application integration, or components called Java Interaction Objects (JIO). A Java Interaction Object is a wrapper for a resource adapter API. (A resource adapter is a connector providing a client interface or connector interaction API to an EIS.) JIOs form the basis of SOAP services for a resource adapter. (Note that a JIO encompasses the same concept as a DAO.)

To a client, a JIO exposes a simpler API than the J2EE Connector interaction API, thus providing a convenient way for a client application to invoke an EIS API method. Typically, you develop a JIO for a specific business function or set of

functions, then client applications use these JIOs or the JIOs act as service endpoints. Another common use is to layer JIOs on top of connectors, thus isolating applications from the details of the connector interactions. Often, each resource adapter interaction has a corresponding JIO. The JIO methods pass input records to the adapter as parameters and return output records to the client application as return values. At the same time, they automatically manage connections, transactions, and authentication. Since a JIO implementation follows the J2EE Connector architecture Common Client Interface (CCI) application programming model, it is thus portable across all environments (managed and nonmanaged).

6.5.3 Metadata

Metadata is information that describes the characteristics of a data set. For an EIS system, metadata describes the content, quality, and condition (among other characteristics) of the EIS's data.

EISs provide a range of access to this metadata. Some EISs provide programmatic access to their metadata, while others provide the metadata information in text form only, as a form of documentation. Sometimes, the metadata information is implicitly available through database system support tools. For example, some EISs let you use JDBC to query the database to learn its table names and schemas. Similarly, an EIS may give you access to its various quality of service parameters.

Metadata is often relevant to integration. Metadata allows you to use tools to discover properties of enterprise systems, and from this discovery to create appropriate, easy-to-use facades to the systems. By having access to metadata, tools can generate more meaningful classes.

Web services are designed to support metadata. Web services rely on WSDL files, which essentially provide metadata describing the services, the operations offered by the services, parameters for these operations, and so forth. Since these WSDL files are XML documents, they are accessible to tools and other programs. (See "Web Services Definition Language" on page 41.)

Using an integration layer helps make metadata more explicit. When metadata is implicitly associated with EIS data, tools may have a difficult time discovering the metadata. For example, a database table column may represent distances from a certain point, and these distances are implicitly measured in miles. A tool that accesses the data from this table column may not be able to determine if the distance is measured in miles or kilometers. By creating an integration layer you can make this information explicit. You can name the methods that access the data in

such a way as to indicate the associated metadata. For example, rather than have a `getDistance` method, you can call the method `getDistanceInMiles`. For Web services, the standard document formats that describe the service are designed to make the metadata more explicit.

Another way to make metadata explicit and accessible, especially for medium to large enterprises, is to store the metadata in a central registry. You can store this information in a traditional LDAP directory. (LDAP, which stands for Lightweight Directory Access Protocol, is an Internet protocol that programs can use to look up information from a server.) You can store Web services-related metadata in a UDDI registry, which is explained in the next section.

You can make your EIS's metadata available to others in several ways. You can enable metadata support in your enterprise systems. The WSDL file that accompanies a Web service already defines metadata for the service. You may also provide specific methods that retrieve metadata related to quality of services, parameter constraints, and so forth. Publishing a schema for a document that a service accepts is another way to provide metadata.

6.5.4 Using Registries for Integration

You may want to consider using registries to integrate an application with an existing EIS, especially when you want a loose binding between the application and the system with which it is integrated. However, using a registry comes with the additional overhead of running and maintaining a registry server and the added programming complexity to have your application use the registry. The registry server may also be a single point of failure for your system.

Generally, it is not worthwhile to use registries for a small enterprise whose applications integrate with just a few EISs. Registries make more sense for medium and large integration architectures, and they work best when more metadata is put in the registry. You want to maximize the use of the registry among as many applications as possible. When deciding what information to store in the registry, try to store data that is useful to more than one application.

Storing metadata in the registry also helps if you need to analyze the impact of any changes to the application.

Also, consider implementing directory-based data sharing, since this enables applications to operate collectively. Such data sharing reduces management costs and improves an enterprise's overall responsiveness to business change. If you use this approach, be sure to set up authorization policies to control access to the data.

It is also important when using a directory service to locate authoritative or reliable sources of the data. A major reason for the failure of a directory service is outdated or suspect data. Users quickly learn that they cannot rely on the service. Even applications can be programmed to recognize and bypass unreliable services. Keep in mind, however, that a single application or database is rarely authoritative for all required data, and this may cause a data integration problem of its own. There are times, too, when individual people are the only authoritative source of data, in which case appropriate checks and approval rules are needed when capturing the data to ensure its quality.

6.5.5 Versioning and Evolution

Applications integrated with EISs change over time, as do the EISs. As these systems and applications evolve over their lifetimes, new versions of them emerge. Integration of the applications and EISs must be able to handle this evolution and versioning process.

Many enterprise applications change because of changes to the underlying business requirements. EISs change for similar reasons, too, but they may also change because of hardware and software upgrades and bug fixes. New functionality may be added to applications and existing functionality retired. Sometimes policies change and, as a result, applications must access EISs in a different manner.

Generally, it is easier to evolve the various components—the enterprise applications and EISs—than it is to evolve the integration layer, since it is the point of stability. Let's first look at some strategies to evolve EISs, and then discuss how to evolve the different types of integration layers.

When evolving changes to an EIS are internal and the external, visible functionality is left the same, you should strive to keep unchanged the original external interface specified in the integration layer. To do this, change the internal implementation to adapt to the original external interface.

It is different when the integration layer evolves. One way to handle this is to use a transformation layer that accepts messages in the older style and transforms them to the current format. Often, it is easier to evolve integration layers that are Web services than other types of integration layers.

There are recommended strategies for evolving EISs and integration layers. When you control both ends of the integration point (the Web services endpoint and the client), it is often easier to upgrade them both at the same time. When just the internal implementation of the Web service changes—and these changes have

no affect on the external interface—then you merely plug in the new implementation.

Internal implementation changes to the Web service that cause changes to the external interface can easily be handled if these changes enhance the service but do not modify the original service contract. In these cases, you can provide two sets of WSDL files, one describing the original service and the other describing the new, enhanced service.

However, more complex scenarios exist, and you cannot expect clients to immediately migrate to a new service implementation. Some clients may never migrate to the new service. For more complex scenarios, a good strategy is to publish a separate Web service endpoint that provides the new service version. Clients can migrate to the new service when convenient for them. You can keep the old service interface but re-implement the existing endpoint to use the new implementation of the service. You may even be able to publish the new endpoint under the same URL.

6.5.6 Writing Applications for Integration

One area in which you have quite a bit of flexibility involves how you code your new EAI application. Since the application is new, you have the maximum amount of choice for implementing how the application will access the various resources that it requires. The J2EE platform provides some facilities to help with this task.

The Java Business Integration (JBI) Systems Programming Interface (SPI), based on JSR 208, extends the J2EE platform with a set of business integration interfaces and components. It sets up a standard way for defining an enterprise's business processing. JBI defines a *business protocol* as the interaction among a set of business processes, whereas an *abstract business process* is the metadata that describes how a business process choreographs its role within the interaction. A *business process* may be any actor that participates in the interaction, and could be as simple as a data transformation table or a business rule.

6.6 Conclusion

This chapter examined some of the key issues for enterprise application integration. It began by illustrating typical integration problems encountered by enterprises, and then described the various integration technologies available with the J2EE 1.4 platform. These J2EE technologies are not limited to Web services only, though Web

services represent one valid integration approach most useful in certain situations. In addition to Web services, the J2EE platform has technologies for integrating with relational databases and other types of data stores, messaging technologies, and EIS access technologies.

The chapter described the major J2EE application integration approaches—including using Web services, enterprise beans and JMS, connectors, and data access objects—and detailed the situations when it was most appropriate to use these different approaches. It also showed how the various approaches could be combined.

The chapter concluded with a set of guidelines for achieving data, application, and business process integration. It also presented some guidelines for integrating home-grown systems, along with how to leverage the capabilities of registries for integration purposes.

Chapter 7 discusses the approaches for implementing secure Web services.