CHAPTER 1

Anatomy of an Enterprise Application

by Mark Johnson

J2EE™ BluePrints is a set of guidelines and best practices to help application architects and developers make most effective use of the Java™ 2 Platform, Enterprise Edition (J2EE™) in enterprise applications. This book describes the J2EE BluePrints application programming model and provides guidelines and best practices for using J2EE technology.

While J2EE BluePrints can't deliver a completed design for your application, it does provide support to architects and developers in a variety of forms:

Architectural recommendations describe effective ways of organizing a system globally to best meet the application's design goals.

Design issues explore the often competing forces affecting design decisions, and how J2EE technology may be used to resolve those forces in context. Design discussions focus on explaining issues, to help architects make informed choices about how to use the technology.

Implementation guidelines are specific advice on either opportunities to use J2EE technology effectively, or warnings about common pitfalls. Implementation guidelines address questions that arise while a system is being constructed. They usually include explanations of the intent of specific technical features of the J2EE platform.

Design patterns are formal descriptions of reusable solutions to recurring problems. J2EE BluePrints includes a catalog of design patterns that have proven useful in existing J2EE application designs.

This chapter offers a high-level introduction to the J2EE application programming model, using the Java™ Pet Store sample application architecture as an example. Since this chapter is an overview, it contains no source code; instead, it serves as a conceptual introduction and a roadmap to the rest of the book. Subsequent chapters drill into the details of each of the topics outlined in this chapter, and include discussions of common application design issues, diagrams, and code samples from the Java Pet Store application.

The code samples in this book come from release 1.3 of the Java Pet Store. The Java Pet Store continues to evolve, as new technologies appear and new usage patterns are identified. As a result, the discussions of the Java Pet Store in this book may not correspond exactly to the current source code. The current version of the Java Pet Store, as well as online supplementary book chapters, are available at the J2EE BluePrints website, `http://java.sun.com/j2ee/blueprints`.

The first section of this chapter explains the design goals of enterprise applications. Following that is a description of the application's architecture, explained in terms of tier decomposition, functional responsibilities, and design patterns. Finally, the chapter addresses a series of high-level design topics, again using the Java Pet Store as an example.

1.1 Enterprise application design goals

Enterprise applications, by their nature, share several common goals. Briefly, these goals include:

- *Extensibility and maintainability*. It should be easy to add new functionality to the system with little or no invasive surgery.
- *Division of work along skill lines.* When workers with different skill sets can work independently, they can focus on what they know best, improving productivity and overall quality.
- *Scalability, portability, availability*. Many enterprise applications must scale to support thousands of users. A portable application reduces the risk of vendor lock-in, and the risk of system obsolescence because a vendor’s products don’t meet your current needs. High availability ensures uninterrupted access to enterprise data and business functions.
- *Code reuse*. Code developed for an application should be reusable in multiple places in the application, and ideally in other applications, as well.

- *Interoperability*. An enterprise application should not assume that it controls or represents everything in an enterprise. It should be able to interoperate with other systems, using their services or providing services to them. In particular, an enterprise system should operate cleanly and efficiently with existing information assets, including legacy systems.
- *Focus on implementing business logic*. Programmers should spend as much time as possible developing code that addresses business issues, and minimize time spent dealing with system mechanisms.
- *Separation of code with differing rates of change*. Placing quickly- and slowly-changing code in separate modules improves maintainability, because the contracts between the modules change less frequently.
- *Ease of migration from a Web-centric to a component-based design*. It should be easy to migrate a pure-play Web application to a scalable, transactional component model when desired.

The Java Pet Store sample application uses J2EE technology to demonstrate how to build an application that meets these goals. It is a multi-tier enterprise application that currently supports a browser-based, business-to-consumer (B2C) shopping interface, an XML-based administrative application, and an XML message-based Web service layer for business-to-business (B2B) integration. The application is implemented in several tiers, each tier being responsible for a high-level design concern such as information persistence, application functionality, or user interaction.

Dividing an enterprise application into tiers supports the application's design goals with layers that separate differing design and implementation concerns. The next section explains the roles of the tiers in a J2EE application, and describes the technologies available in each tier.

1.2 J2EE Application Tiers

This section describes the tiers common to all J2EE applications, and explains how the Java Pet Store uses these tiers to meet its design goals.

1.2.1 Multi-tiered Architecture

J2EE applications are, by definition, multi-tiered. Each tier brings specific benefits to a design. A tiered architecture provides natural access points for integration with

existing and future systems, and for deployment of new systems and interfaces as needs arise. In fact, the administrative interface and the Web services layer were both created and integrated with the existing Pet Store code base after the initial Java Pet Store shopping application had been completed.

J2EE applications divide their functionality across several tiers, or functional layers, each of which serves a specific purpose. Figure 1.1 shows the tiers defined in the J2EE environment. This diagram is not specific to the Java Pet Store; the divisions shown apply to all J2EE applications. Each tier of a J2EE application

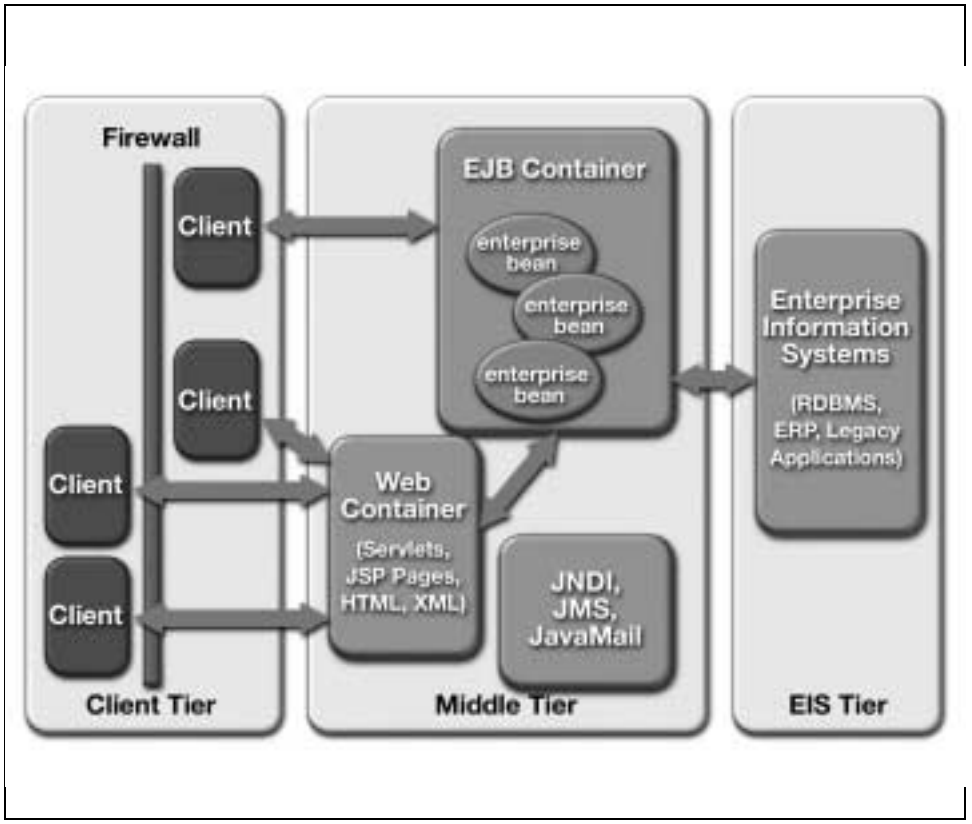

Figure 1.1 Each J2EE technology tier has a specific function

has specific responsibilities:

- The *Client tier* allows users to interact with the J2EE application. User inter-

action and data capture occur in the client tier. Client tier programs translate user actions and input into server requests, and format server responses into some form (usually visual) that a human can use.

- The *Web tier* makes application functionality available on the World Wide Web. It accesses data and business functionality from other tiers, manages screen flow, and often encapsulates some user interaction control. The Web tier typically interacts with client-tier programs using HTTP, and may access other tiers with a variety of protocols. In Web service scenarios, other applications take the place of client programs, accessing a J2EE application through its Web tier.
- The *Enterprise JavaBeans™ ("EJB™") tier* provides portable, scalable, available, and high-performance access to enterprise data and business rules. It offers object persistence and access to business logic implemented as enterprise bean components. A distributed object server (the "EJB server") manages efficient access to instances of these components. Other tiers access server-side enterprise bean instances using the Java™ Remote Method Invocation ("RMI") API. Enterprise beans provide a simplified programming model for distributed objects, since the environment manages the implementation details of multithreading, persistence, security, transactions, and concurrent data access. The enterprise beans or the EJB server may both access other tiers using a variety of methods.
- The *EIS tier* (Enterprise Information Systems tier) integrates a J2EE application with other enterprise information systems. It provides data storage and other information services to the J2EE application. Databases, enterprise resource planning systems, mainframe transaction processors, legacy systems, and enterprise integration technologies usually reside in the EIS tier. Other tiers access EIS-tier databases with a driver supporting the Java DataBase Connectivity™ ("JDBC™") API, and access non-database EIS resources with J2EE™ Connector Extensions ("Java Connectors"), CORBA, or vendors' proprietary protocols.

Note that these tiers are divided primarily along functional, not necessarily physical, lines. A tier boundary may or may not involve a network interface. In fact, co-locating two or more tiers on the same machine, or even within the same JVM[1], is a very common optimization strategy.

Applications are not required to use every tier. A given application design may include all or only some of the tiers. For example, a traditional two-tier enter-

prise application might consist of little more than a rich client written in the Java programming language, accessing an EIS-tier database using JDBC. The Java Pet Store has components in all four tiers.

Some J2EE technologies, such as servlets or enterprise beans, clearly belong in a specific tier; others, such as the Java[SM] Message Service ("JMS") API or the JAX family of XML APIs, are applicable across tiers. For example, Enterprise JavaBeans[TM] are available only in the EJB tier, but all tiers can make use of the XML APIs. The fact that each tier is optional, combined with a number of inter-tier communication mechanisms, makes technology selection extremely flexible.

With an understanding of the purpose of each tier, it's now possible to describe how the tiers in the Java Pet Store work together. The next few sections describe the J2EE tiers, and explain the role they play in the Java Pet Store.

1.2.2 Tier Structure

The next few sections of the book describe the tiers of the J2EE programming model. Before describing the tiers in detail, an explanation of the term *tier* is in order. The term "tier", as used in this book, refers to a defined collection of technologies that provides services to clients of the tier, and that possibly uses resources in other tiers to do so.

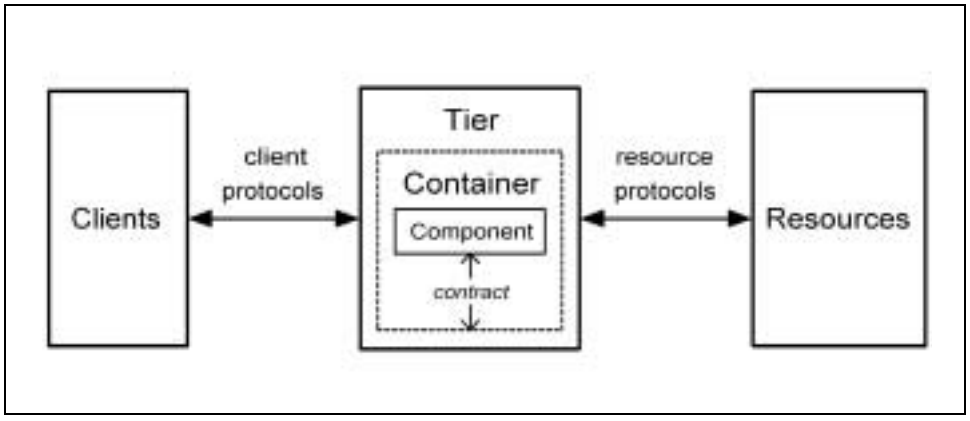

Figure 1.2 General structure of a tier in a J2EE application

1. The terms "Java[TM] virtual machine" and "JVM[TM]" mean a virtual machine for the Java platform.

Figure 1.2 is a diagram of an abstract tier in a J2EE application. Each tier contains specific technologies, uses resources (shown on the right in Figure 1.2) which reside in certain other tiers, and/or provides services to its clients (on the left), which reside still other tiers. The term "client" refers here to anything that may use the services the tier provides, and is not necessarily a "client program"; likewise, the term "resource" means anything the tier might use to do its job.

A program, class, or collection of related classes performing application functions within the tier is called a *component*. The component receives system services from and is often managed by its *container.* The component/container *contract* is an API that specifies what services a component can expect from its container, and what rules the component must follow. Some of the elements in Figure 1.2 do not exist for every tier; for example, the Client tier has only human beings (or perhaps devices) as its clients, and there is no "EIS tier container" *per se*.

In J2EE applications, the container takes responsibility for system-level services (such as threading, resource management, transactions, security, persistence, and so on). This arrangement leaves the component developer with the simplified task of developing business functionality. It also allows the implementation details of the system services to be reconfigured without changing the component code, making components useful in a wide range of contexts.

As an example of the container/component concept, outside of J2EE technology, consider a program running within a traditional operating system. A program is a component that performs business functions. An operating system is a container, offering system services to and managing instances of the program (processes). The component/container contract is the system call API. The container handles system details such as memory management and disk drive control, freeing the application developer to focus on business functionality.

Clients access the services within a tier by way of *client protocols*. A tier may use *resource protocols* to access *resources* in other tiers. A protocol is always specific to the technology being accessed; for example, Web technologies are always accessed using HTTP or HTTPS (which is HTTP with encrypted transport). Note again that the term *client* may mean either a program that interacts with a person (as in a "desktop client"), or any program that receives services from another program.

For example, describing the Web tier in terms of Figure 1.2, the clients of the Web tier are web browsers and Web service peers, with the client protocol being either HTTP or HTTPS. Resources used by the Web tier might include the EJB tier, the EIS tier, or the Web tier of another application, with resource protocols

being IIOP, JDBC, and HTTP, respectively. The Web tier container is a Web container supporting servlet and JSP page components, as defined by the J2EE specification.

J2EE defines this model for technologies in all tiers. The following sections describe each J2EE tier in terms of the purpose of the tier, the container and component technologies that operate within the tier, the types of clients and client protocols the tier can serve, and the resources and resource protocols the tier may use.

1.2.3 The Client Tier

The Client tier, as described above, is responsible for interacting with human users. Client tier programs prompt for input, interpret user actions, perform server requests, and present results. Clients differ by the tier or tiers they access, and the technologies used to implement them. This section briefly explains Client tier concepts and describes their implementations in the Java Pet Store.

Client types by tier

Figure 1.3 shows client types organized by which tier that type of client accesses, and indicates how each type of client accesses its tier. As mentioned previously, the

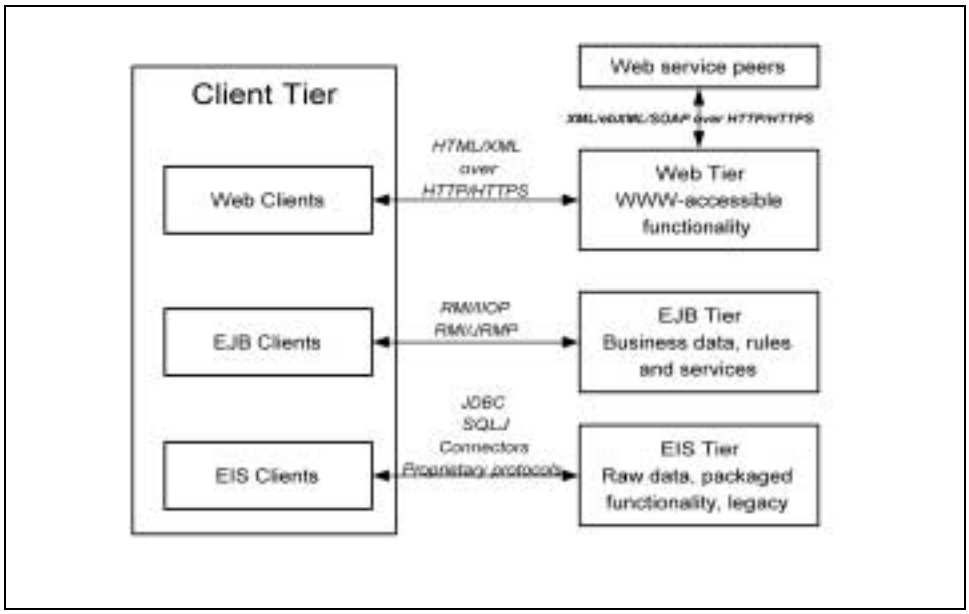

Figure 1.3 Application tiers, communication, and responsibilities

client tier uses resources in other tiers (the Web tier, the EJB tier, and the EIS tier, on the right in Figure 1.3). The users of the client tier are usually devices used by human beings, or Web service peers (shown separately in Figure 1.3).

The types of clients in the client tier of the J2EE application programming model are:

- *Web clients*, which access HTML or XML content on the Web tier using HTTP or HTTPS. Web clients can also receive content of arbitrary type.
- *EJB clients*, which access enterprise beans in an EJB server in the EJB tier, using RMI over either IIOP or JRMP.
- *EIS clients*, which access databases, packaged functionality such as ERP (enterprise resource planning) or scheduling systems, and legacy information systems. EIS clients use JDBC for databases, and either Java Connectors or vendor-proprietary protocols for other EIS-tier resources.
- *Web service peers*, which are external systems that access data and functionality on the Web tier. J2EE Web services are primarily peer-to-peer, rather than client-server, access to Web tier application functionality. (Web services also incorporate service discovery, runtime binding, and other functionality. In general, they are not limited to the Web tier.) Though possibly lacking a user interface, Web service peers can be considered Web clients because, like other Web clients, they request Web tier services and data using XML, usually as ebXML or SOAP messages, over HTTP.
- Not shown in Figure 1.3 are *multi-tier clients*, which access data and functionality from more than one tier.

Some communication technologies, notably JMS, allow for synchronous or asynchronous communication between programs within or between tiers.

Client tier technologies

J2EE supports many types of client technologies. The most common Web client is the *Web browser*, which interacts with the Web tier using HTML (and increasingly, XML) over HTTP.

J2EE defines two types of client-tier components: *applets* and *application clients*. An *applet* is a graphical component running within an *applet container* hosted by a Web browser or other program. Applets are suitable only as Web clients.

Application clients are full-blown Java platform applications, usually running on the desktop with a standalone GUI. Application clients may access one or more application tiers. The *application client container* is the Java™ 2 Runtime Environment, Standard Edition ("JRE"). Since application clients have access to all inter-tier connectivity technologies, they can be used as clients to any tier.

Rich clients are standalone GUI applications, written in Java or another language to invoke application functions in the Web tier. Rich clients are often GUI front-ends to Web services. Such clients request application services with XML over HTTP, using standard protocols such as ebXML or SOAP. Rich clients written in the Java language often use the Java™ Foundation Classes/Swing ("JFC/Swing") API for presentation and user interaction. Standard Java extension APIs for XML, such as JAX-RPC, simplify rich client implementation. Since rich clients do not necessarily use Java technology, and are not defined as "components" by the J2EE specification, there is no "rich client container".

Installation and upgrades for standalone application clients and rich clients can be managed transparently with Java WebStart™ technology.

Finally, clients not based on Java technology may interoperate with J2EE applications if they have access to the protocol for the corresponding tier (sometimes through an appropriate protocol bridge). Such clients include *browser plugins* (such as MacroMedia Flash), *browser helper apps* (such as streaming media players), and *non-Java technology clients* (such as a scriptable spreadsheet application).

Java Pet Store clients

As of this writing, the Java Pet Store has four clients: three Web clients and one EIS client. Figure 1.4 shows the four Java Pet Store clients interacting with the Java Pet Store Web and EIS tiers.

When users place orders for pets using through their Web browsers, they are

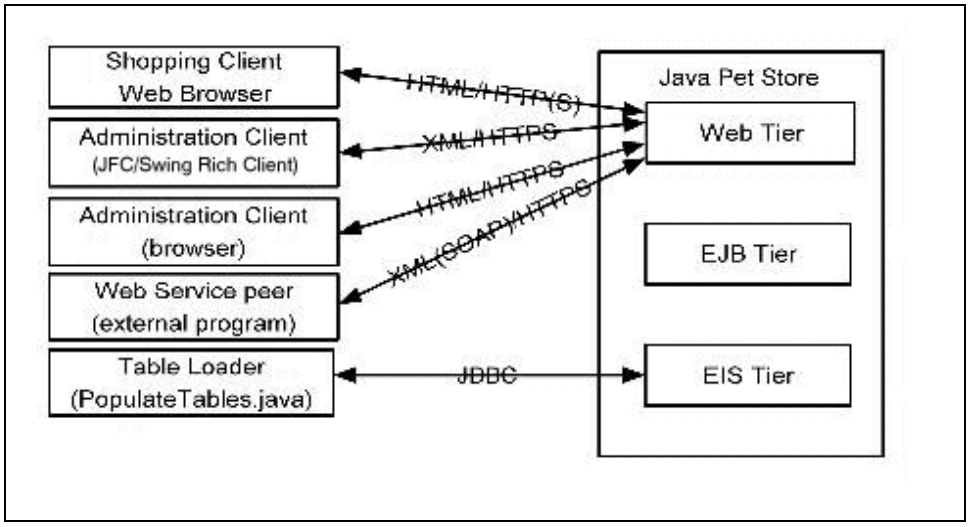

Figure 1.4 Tier interaction of Java Pet Store clients

using the Shopping client. The Shopping client is a Web browser interacting with the Java Pet Store Web tier. The browser displays HTML generated by Web-tier Java™ Servlet technology ("servlets") and pages created with JavaServer Pages™ (JSP™) technology, as well as static HTML pages. Communication is over HTTP for "anonymous" users, and over HTTPS for authenticated (or "signed on") users.

The Administration application allows a Pet Store administrator to approve or deny customer orders and view statistics using a Java WebStart-enabled Web service rich client. A collection of JSP pages provides an equivalent, traditional Web browser interface to the administration application.

The B2B Web service layer of the Java Pet Store performs application functions when requested by Web service peers. These peers execute service requests on the Web service layer using SOAP messages.

Finally, the EIS-tier application client `PopulateTables.java` creates and populates database tables when the Java Pet Store is first installed. PopulateTables has a command-line interface. It opens a JDBC connection directly to the database in the EIS tier and executes SQL statements through the connection.

1.2.4 The Web Tier

In a J2EE application, the Web tier provides application functionality over the HTTP protocol. Application logic may be implemented directly by the Web tier, as in a traditional "Web application". Or, it may act as a simple protocol translator, directly transforming HTTP requests to calls on some other tier, and rendering the results of those requests as HTTP responses. But most solutions fall somewhere between these two extremes. Common responsibilities for the Web tier include:

- *Data presentation and input collection.* The Web tier formats and delivers application data and input forms as HTML. It also collects the results of HTTP POST, GET, and PUT operations, passing them on to other software layers for processing.
- *Dynamic content generation.* As part of data presentation, Web tier components can generate content on-the-fly, personalized for a particular context. For example, a servlet can generate a graphic showing the current values of stocks in a portfolio, thereby creating content specific to a particular user and time.
- *Screen flow*. Browser-based applications typically operate by displaying a succession of pages. The Web tier is often responsible for selecting the next screen to be displayed, based on user inputs and other state.
- *User and application state management.* Most nontrivial applications require a concept of state, which is a collection of data values associated with a particular user, application, transaction or task in progress. The contents Java Pet Store shopping cart is a typical example of such state. The Web tier provides facilities for managing state.
- *Protocol translation.* The Web tier accesses other information sources and make them available through HTTP. For example, the Web tier could allow applets to access enterprise beans by translating between HTTP and the Java RMI API.

Web servers accept requests from and produce responses to Web clients using either HTTP or HTTPS. HTTP can deliver any kind of content, from simple text or HTML files to high-volume streaming media.

Figure 1.5 shows the Web tier in communication with other tiers. Web clients

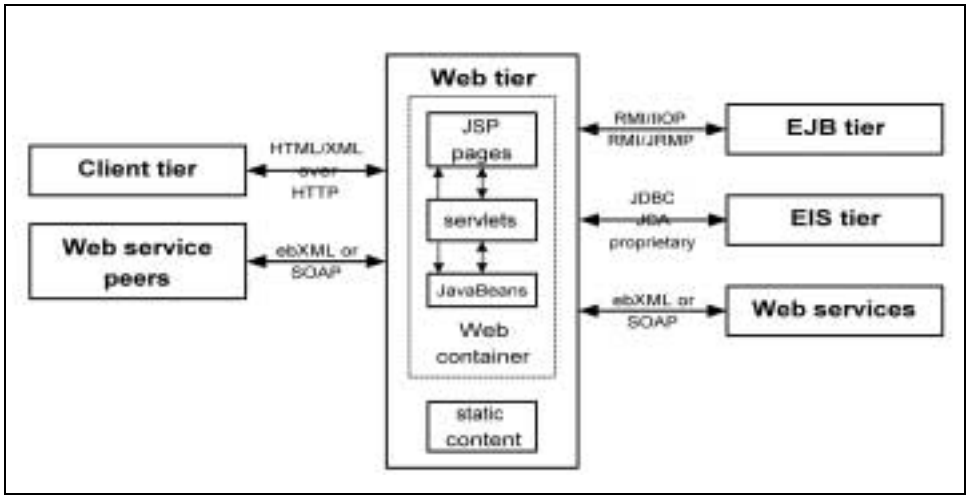

Figure 1.5 The Web tier, Web tier technologies, and communication with other tiers

and Web service peers interact with the Web tier using the HTTP protocol. The Web tier can implement its services using the EJB tier (by way of RMI/IIOP or RMI/JRMP), the EIS tier (using JDBC, Java Connectors, CORBA, or proprietary protocols), or by invoking Web services of other systems (using ebXML or SOAP). Web tier applications are composed of JSP pages, servlets, components based on the JavaBeans™ component architecture ("beans"), and static content. The next section describes technologies that can be used in the Web tier to create Web applications.

Web tier components

Web tier components are software components that extend a Web server. The first Web servers delivered only static content, usually stored in the server machine's file-system. The server performed a simple mapping from a requested URL to the path of the file to be delivered.

The desire for dynamic content lead to the Common Gateway Interface (CGI), in which a URL maps to a program that produces content, instead of mapping to a content file itself. CGI was the first dynamic content generation technology, and remains a popular option. Yet CGI's weaknesses have motivated the creation of

several server-side extension mechanisms. J2EE offers two types of components for extending the functionality of the Web tier: *servlets* and *JSP pages (JavaServer Pages)*. (See Figure 1.5.)

Servlets are Java classes managed by a *servlet container* hosted by a Web server. Servlets are mapped to URLs by the servlet container. HTTP requests to a servlet's URL are passed to an instance of the servlet, which generates response content that is returned to the client as the HTTP response. Servlets receive input variables from the HTTP request, can maintain state across invocations, and have control of the content type the server returns. Servlets provide a portable, safe, efficient, and highly general mechanism for extending a Web server.

Servlets are powerful, but they require programming. *JSP pages* provide a "structural" way to create dynamic textual content, as an alternative to the "programmatic" method offered by servlets. A JSP page is an HTML page with special markup produces dynamic content.

For example, the Java Pet Store has a page that shows a summary of the data about a particular type of pet: an image of the pet, the name and description, price, and quantity in inventory. All of these pages have an identical structure, differing only by the data on the page. A servlet to produce this page would be mostly print statements, printing all of the HTML tags, with just a few statements printing data about the pet. In a JSP based solution, the page is written as an HTML page, and special markup indicates where in the page the dynamic data (from the database) belong.

Although a JSP page's source code looks like HTML, it's actually implemented by a servlet. When a client accesses a JSP page, the Web server translates the JSP page into a Java language servlet, compiles and loads the servlet, and then passes the request to the servlet. (If the servlet is already loaded, the server skips the translation, compilation, and loading.) So, a JSP page is simply an easy way to specify a servlet that produces mostly structured data.

Servlets and JSP pages can encapsulate state in the Web tier with *JavaBeans components* inside the servlet container. Servlets access these beans directly, while JSP pages access them by way of custom tags. The Java Pet Store uses Web tier beans to encapsulate application state such as the contents of the shopping cart, orders, and product information. While these components are "classic" JavaBeans components, the beans' implementations may access other tiers, including the EJB tier.

Java Pet Store Web tier

The Web tier of the Java Pet Store serves content to three of the four Java Pet Store clients, using a combination of servlets, JSP pages, and Web tier JavaBeans components. This section provides a quick overview of how the Web tier works in the Java Pet Store.

All requests from browsers acting as Shopping clients are routed through a single servlet, `MainServlet.java`, which acts as a Front Controller for the shopping client. This servlet acts as a sort of switchboard, dispatching incoming requests to Web tier handler objects that implement application functions. Handler classes may perform functions as simple as serving a requested JSP page, or as complex as creating a new user and sending verification e-mail. Based on the request and the handler result, MainServlet also selects the next page to display as a response.

A small collection of JSP pages provide a Web browser interface for the Administration client. In addition, administrators can update orders from within a spreadsheet, using a macro that communicates with two Web tier components. The macro requests a JSP page that produces an XML-formatted list of orders. After the administrator edits the spreadsheet, the macro sends an XML document of updates to a servlet that updates orders in the database.

The Web services layer provides Web service peers with access to application functionality by way of XML (SOAP) over HTTPS.

The Java Pet Store Web tier manages application state used for data presentation, such as screen flow information, locale, and error messages. Business state, such as the content of the shopping cart, is managed by stateless session beans in the EJB tier.

1.2.5 The EJB tier

A J2EE application's EJB tier provides application functionality by way of a distributed object server called an EJB server. *Enterprise beans* are software components, each composed of a small collection of related classes, running inside an *EJB container*, which is hosted by the EJB server. Programs in other tiers can find, create, request services of, modify, and destroy enterprise bean instances running within the EJB server. For example, the contents of the Java Pet Store shopping cart are managed by an enterprise bean in the EJB tier.

The EJB server and container handle complex, system-level mechanisms, freeing component and application developers to focus on business processes and data. The EJB server and container handle distributed object protocols, object per-

sistence, resource pooling, and thread, process and state management; they transparently provide high availability and scalability; and they enable deploy-time configuration of transactional behavior, persistence, and security.

The division of labor between enterprise beans (which implement application functionality) and the EJB server and container (which provide system services) also separates the roles of component, server, and container developers. Component and application developers can focus on writing business code, while application server and EJB container developers specialize in writing efficient servers. Each kind of developer works primarily in his or her own sphere of expertise.

Figure 1.6 shows the communication between the EJB tier and programs in other tiers. Application clients in the client tier and Web servers and services in the Web tier communicate with enterprise beans in the EJB tier through the Java RMI API over either IIOP or JRMP. JMS clients in any tier can interact with EJB tier message-driven beans (MDBs, a feature of the EJB 2.0 specification). Any program that uses enterprise beans is called an EJB client, whether or not that program is in the client tier. Enterprise beans can access databases in the EIS tier using a JDBC driver, and can access other enterprise information systems with Java Connectors, CORBA, or proprietary protocols.

There are two mechanisms for storing components with persistent state (entity beans). With Container-Managed Persistence (CMP), the container automatically

manages the persistent state of the beans it contains. The EJB container accesses

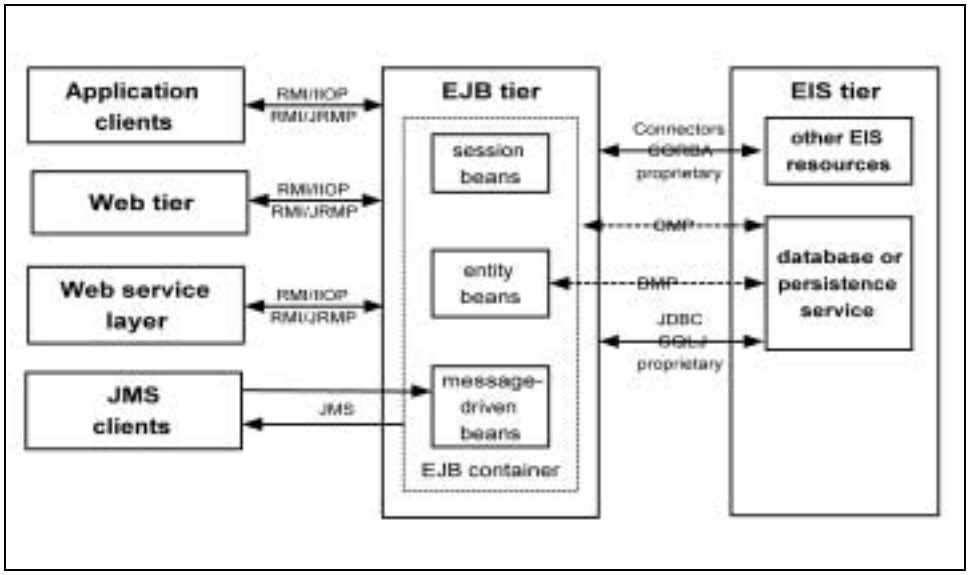

Figure 1.6 EJB tier interaction with other tiers

the EIS tier using a protocol selected by the container provider. Bean-Managed Persistence (BMP) allows a bean developer to manage a bean's state manually, using any EIS tier protocol. BMP and CMP are not protocols, but they appear in Figure 1.6 to point out the special relationship between the EJB and EIS tiers in the case of persistent components.

The EJB tier offers a large list of benefits, including:

- *Automated resource pooling.* The EJB server automatically pools resources such as database connections and component instances, addressing scalability bottlenecks.
- *Automated instance lifecycle management.* Component instances are created, destroyed, and swapped in and out of memory by their container in a structured, efficient way.
- *Concurrency.* The EJB container automatically manages the difficult problem of multiple clients accessing the same distributed data or functionality.

- *Declarative functionality.* Security, transactional behavior, and persistence are implemented by the server, and configured when the application is deployed.
- *Scalability.* Enterprise beans are designed to work correctly in a clustered environment.
- *High availability and fault tolerance.* EJB servers can provide failover and high availability.
- *Distributed object model.* Developers can create and use distributed objects, letting the development framework and runtime handle such details as marshalling and remote procedure calls.

EJBs offer many other benefits; see the chapter on the EJB tier for more details. The EJB tier realizes many of the enterprise design goals listed at the beginning of this chapter.

Types of Enterprise Beans

There are three types of enterprise beans: *session beans*, which correspond to a client/server interaction or conversation; *entity beans*, which implement functionality associated with persistent state; and *message-driven beans* (MDB), which provide asynchronous execution of application functionality in response to JMS messages.

A session bean implements a conversation between a client and the server side of a J2EE application. Session beans are commonly used to implement business processes or workflow. The client uses the session bean's home interface to create a server-side session bean, and then has a conversation with the bean by calling its methods through the bean's remote interface. The bean may maintain *session state* (data specific to the session) across multiple method calls. Beans that maintain session state are called *stateful* beans; otherwise, they are called *stateless* beans. Every stateful session bean instance serves exactly one client, since it keeps track of state associated with that client. A stateless session bean can serve multiple clients, since it maintains no session state; therefore, stateless session beans are more scalable than stateful beans. The Java Pet store implements its shopping cart as a stateful session bean, and its catalog as a stateless session bean.

Entity beans are persistent business objects. They typically represent business entities, such as Customer or Account; or business processes, such as LoanApplication or PolicyEnrollment. Entity bean persistence is managed either by the container (called Container-Managed Persistence, or CMP), or by the bean itself

(called Bean-Managed Persistence, or BMP). The container can also manage transactions and security for entity beans. The EJB container ensures that entity bean state survives any crash or restart of the container. (You don't want your Customers or Accounts to be lost because a server was restarted.) Entity beans are especially useful for managing multiple concurrent access to persistent business data. The Java Pet Store includes several entity beans, including Account, Order, and Inventory.

A message-driven bean (MDB) allows clients to asynchronously invoke server-side business logic. MDBs have no home or remote interfaces; instead, they are generic JMS message consumers, to which clients may send asynchronous messages. MDBs are stateless components, so they maintain no information about the client between calls. The EJB container provides MDBs with transactions, security, concurrency, and other services, just as it does for other enterprise bean types. The EJB container improves server performance by pooling MDB instances, and also handles message acknowledgement, when appropriate. Message-driven beans work with queues and both durable and nondurable topic subscriptions.

Enterprise Bean Programming Model

An *enterprise bean class* is a Java class that implements application functionality within an EJB server's *EJB container*; see Figure 1.6. Clients from any tier can remotely find, create, use, and destroy server-side entity and session bean instances using a collection of auxiliary classes and interfaces associated with the bean. Some of these auxiliary classes and interfaces are written by developers, and some are generated by the deployment tools. Each enterprise bean class, together with its auxiliary classes and interfaces, form a software component called an *enterprise bean* that can be deployed as a unit within any EJB server.

A client manipulates an enterprise bean (except for *message-driven beans*, see below) through the bean's *home* and *component* interfaces. An enterprise bean *home interface* has methods for creating, finding, and destroying server-side enterprise bean instances, and may also have *home business methods* that modify or delete multiple bean instances in a single operation. An enterprise bean *component interface* contains the business methods for a single instance of the bean, plus methods that the server calls to manage the bean's lifecycle (creation, destruction, state management, and so on).

A Bean Provider writes the enterprise bean class itself, plus the bean's home and component interfaces. The client-side implementations of the home and component interfaces (the home class and component class) are generated by deploy-

ment tools, and handle the communication between the client and the EJB server. Clients can access an enterprise bean only through the bean's home and component interfaces, and never directly. Since all method calls from a client to an enterprise bean are indirect (through the generated home or component class), the EJB container can control how and when calls to the enterprise bean class occur. The indirection allows the EJB container to provide functionality such as lifecycle management, security, and transactions between enterprise bean class method calls.

Message-driven beans, new in EJB 2.0, are written by a Bean Provider as enterprise bean classes, but have no home or remote interfaces. Instead, MDBs are created and managed by the EJB container, and receive messages from the JMS Destination (Topic or Queue).

As of the EJB 2.0 specification, an enterprise bean can access another enterprise bean through a *local interface* (obtained through a *local home*). Local interfaces provide a way for enterprise beans in the same JVM instance to access each other efficiently. A local interface works very much like a remote interface, except that local interfaces visible only within a single JVM instance; therefore, only other enterprise beans[2], and not external clients, can use local interfaces. Since enterprise beans that access each other through a local interface are inside the same container and JVM instance, a call from one bean to the other can be a direct method call, instead of an expensive marshalled Java RMI call. Calls through local interfaces are still managed by the container, but avoid the cost of unnecessary remote communication overhead.

Java Pet Store EJB tier

The Java Pet store uses the EJB tier to encapsulate virtually all application functionality. Several kinds of clients access Java Pet Store business logic and data through the EJB tier, which automatically provides transaction control and concurrency management. The Java Pet Store contains all three kinds of enterprise beans. Java Pet Store session beans include:

- *Customer* [stateless]: serves as a facade for the Order and Account entity beans.
- *Mailer* [stateless]: sends e-mail confirming an order.
- *Catalog* [stateless]: provides catalog browsing.

2. Some J2EE product providers also allow Web-tier components to access local interfaces.

- *ShoppingCart* [stateful]: contains items in Customer's virtual shopping cart.
- *ShoppingClientController* [stateful]: controller for shopping clients.
- *AdminClientController* [stateless]: controller for admin clients.

Java Pet Store entity beans include:

- *Account*: a customer's account information.
- *Order*: a merchandise order placed with the shopping client.
- *Inventory*: what's available in the warehouse.
- *ProfileMgr*: manages user customization information.

As mentioned above, enterprise beans in the Java Pet Store handle concurrent access to shared data (updating the Inventory, for example). Enterprise beans are also more scalable than a Web tier-only solution, and support multiple client types with a single source base (both the Web and spreadsheet clients for the Administration application use the same enterprise beans).

Note that the entity beans are all coarse-grained objects. There are no Address or PhoneNumber entity beans; such classes are modelled as "value objects" in the Java Pet Store. Value objects are unnecessary when using version 2.0 enterprise beans, since local interfaces make fine-grained entity beans practical. Because of the latency of remote communication, remote interfaces should be used only for coarse-grained objects. Local interfaces are usable for either coarse- or fine-grained objects.

1.2.6 The EIS tier

The EIS (Enterprise Information Systems) tier contains data and services implemented by non-J2EE resources. Databases, legacy systems, ERP (Enterprise Resource Planning) and EAI (Enterprise Application Integration) systems, process schedulers, and other existing or purchased packages reside in the EIS tier. Integration in the EIS tier allows system designers to choose the EIS resources that best fit their needs, and still interoperate seamlessly with J2EE. J2EE applications can

access legacy systems in the EIS tier, making it easier to plan phased projects, and maintaining the value of existing IT investments.

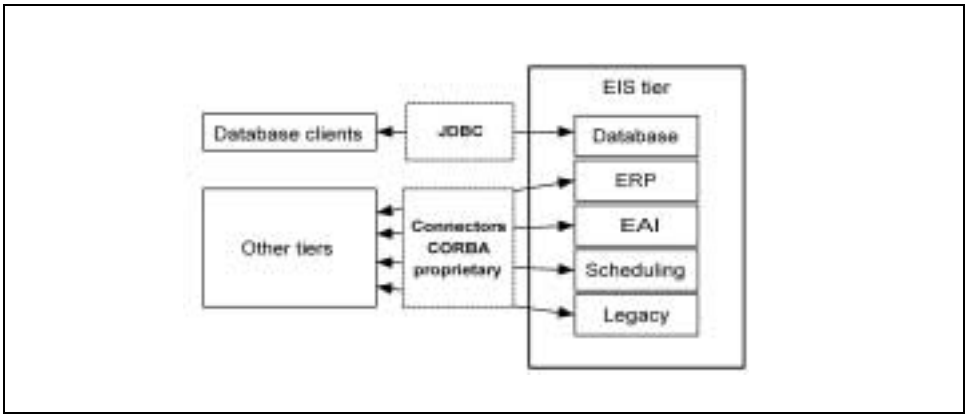

Figure 1.7 EIS tier clients and interface technologies

Figure 1.7 shows the relationship between the EIS tier and the clients it serves. Clients of the EIS tier may reside in any tier, including the EIS tier itself. EIS tier databases (or other object persistence systems) are accessible by way of JDBC. Clients can access non-database resources with Java Connectors, CORBA, or protocols proprietary to the resource vendor.

EIS tier technologies

There are no J2EE "components" or "containers" as such in the EIS tier. But there are J2EE technologies that make EIS tier resources available to J2EE applications. EIS tier databases are most often accessed with JDBC, a connection-based API for communicating with generic SQL databases.

Java Connectors are the recommended way to interface third-party EIS systems with J2EE applications. The Java Connector Architecture defines connection management, transaction, and security contracts between J2EE and a resource adapter for a third-party system. EIS system vendors can use the Java Connector Architecture to create Connectors for their products, and sell their products into the J2EE market. Organizations may create connectors for their legacy systems, making those systems easy to integrate into new or existing J2EE applications.

J2EE applications may also access EIS resources with proprietary APIs and/or CORBA, at the cost of compromising many of the benefits of J2EE (portability, transactionality, stability, etc.).

Java Pet Store EIS tier

The Java Pet Store EIS tier has a single client in the client tier, `PopulateTables.java`, which creates and loads sample data into the database at installation time. Yet both the Web tier and the EJB tier are also clients of the EIS tier.

The Web tier accesses the EIS tier directly through JDBC technology when displaying lists of catalog items to the user. As a performance optimization, the Web component that shows items opens a JDBC connection to the EIS tier database and reads the item data directly. Since catalog data change slowly, and data access in this case is read-only and nontransactional, there's little need for what the EJB tier provides. So, the catalog item list Web component simply ignores the EJB tier, and accesses the EIS tier directly in this case.

The EJB tier can perform Bean-Managed Persistence for its entity beans using JDBC technology to store and load bean instance data. At the time the Java Pet Store was first written, CMP was not portable, and so the Java Pet Store designers choose BMP over CMP. Future versions of the Java Pet Store may demonstrate portable use of CMP with the EJB 2.0 specification.

1.3 Application Architecture

Any enterprise application of realistic size requires a consistent architectural vision. Poorly-factored enterprise applications, with no clear requirements, and maintainable only by the developers that created them, eventually collapse under their own weight.

J2EE BluePrints recommends Model-View-Controller (MVC) as the base architectural pattern for interactive J2EE applications. This section gives a brief introduction to MVC and describes how the Java Pet Store is organized around an MVC framework.

J2EE also recommends applying a set of design patterns to solve common application design problems. This section quickly outlines some common J2EE design patterns.

1.3.1 MVC Basics

The Model-View-Controller (MVC) architecture organizes an interactive application design by separating data presentation, data representation, and application behavior (see Figure 1.8). The *Model* represents the structure of the data in the application, as well as application-specific operations on those data. A *View* (of which there may be many) presents data in some form to a user, in the context of some business function. A *Controller* translates user actions (mouse motions, keystrokes, words spoken, etc.) and user input into business method calls on the Model, and selects the appropriate View based on user preferences and Model state. Essentially, a Model abstracts application state and functionality, a View abstracts application presentation, and a Controller abstracts application behavior in response to user input.

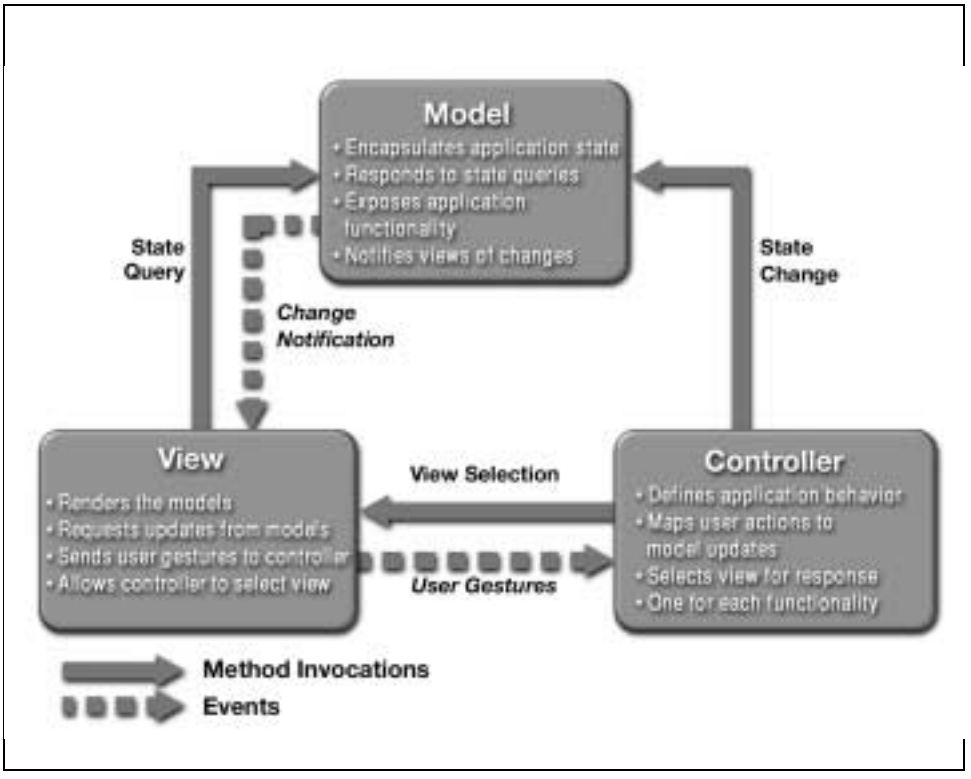

Figure 1.8 The Model-View-Controller architectural pattern

MVC provides many benefits to a design. Separating Model from View (that is, separating data representation from presentation) makes it easy to add multiple data presentations for the same data, and facilitates adding new types of data presentation as technology develops. Model and View components can vary independently (except in their interface), enhancing maintainability, extensibility, and testability. Separating Controller from View (application behavior from presentation) permits run-time selection of appropriate Views based on workflow, user preferences, or Model state. Separating Controller from Model (application behavior from data representation) allows configurable mapping of user actions on the Controller to application functions on the Model.

Non-interactive applications can modify the MVC concept, sending input to a controller (that interprets input instead of user gestures), and providing result sets as "views". MVC is not necessarily the best architectural pattern for every system.

1.3.2 Java Pet Store MVC design

The Java Pet Store is a multi-tier MVC application. Figure 1.9 shows a high-level block diagram of the Java Pet Store's shopping subsystem in terms of MVC architecture and tier decomposition. The Java Pet store Views are JSP pages, composed with templates and displayed in an HTML browser. The Controller maps user input from the browser to request events, and forwards those events to the Shopping Client Controller in the EJB tier. The Shopping Client Controller performs business methods on the Model Enterprise Beans, and then forwards any resulting updates to a cache of JavaBeans components that mirror Model data in the Web tier. The View JSP pages may later use these Web-tier Model beans to produce up-to-date dynamic content. The next few sections describe the View, Controller, and Model in the Java Pet Store design. Each section refers to the corresponding shaded area in Figure 1.9. For more detailed descriptions, design discussions, code examples, and a description of additional sample application functionality, see *Appendix D, The Java Pet Store Sample Application*, starting on page NN.

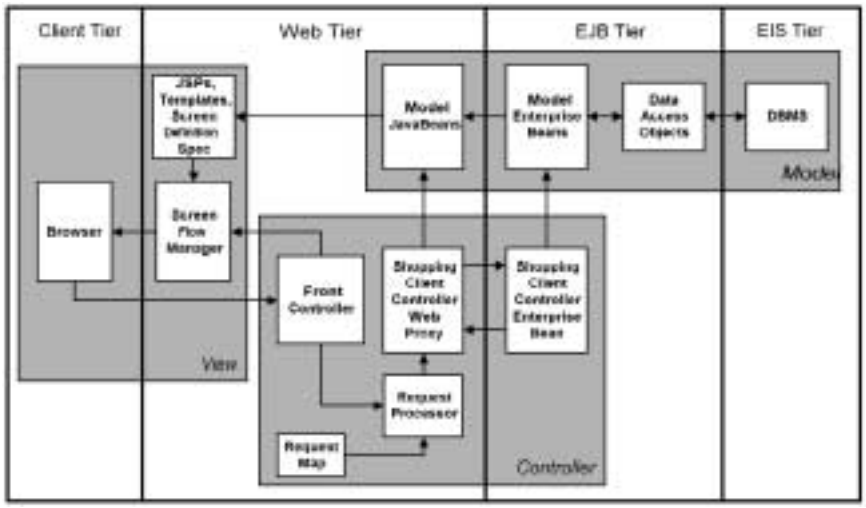

Figure 1.9 The MVC and tier partitioning of the Java Pet Store

1.3.2.1 Java Pet Store Views

Java Pet Store shopping client consists entirely of JSP pages, assembled by the Web tier and displayed by a client-tier Web browser. The View selection and display mechanism is declarative, extensible, and internationalizable. This section describes how Java Pet Store Views are implemented.

The Web-tier Controller interprets each HTTP POST or GET request from the browser as a request for an application service, such as viewing a catalog page or adding an item to the shopping cart. Based on the result of the service request and the user's current state, the Web-tier screen flow manager chooses and assembles the next screen (implemented as a JSP page), and serves the resulting HTML to the client.

Selection of the next view is properly the role of the Controller. So the view selection mechanism is Controller functionality, while templating, JSP page assembly, and content delivery are all View functionality. If the JSP page being served is a template, the screen flow manager assembles several pages, one for each section of the template, into a single HTML response. The JSP pages display current Model data by referring to Web-tier JavaBeans components that mirror data values in EJB tier Model objects. While Java Pet Store JSP pages always produce HTML, other content types (such as XML, PDF, Flash, etc.) could also be supported if desired.

Screens are defined in an XML file deployed with the application. Screen definitions, which name and describe the contents of each page in the application, are loaded by the screen flow manager on initialization. Screens can be defined or changed by simply changing the contents of the XML screen definitions file and redeploying, with no recompilation necessary. Only the names of the screens are compiled into the application.

The screen definitions file also defines screen flow; that is, selection of the next screen to display. In most cases, the screen flow can be defined declaratively in the screen definitions file. In cases where next screen to display must be determined programmatically, the screen definition may defer the selection of the next screen to a programmer-defined "flow handler" class that selects and returns the identifier of the next screen.

Java Pet Store views are incrementally internationalizable. The screen flow manager loads and serves pages appropriate to the current locale. The entire look and feel of the application, or just selected parts of it, may be customized to fit the language and culture of the client being served. For example, an Arabic version of the Pet Store would have text running from right to left, and might have a navigation bar along the right, rather than the left, side of the screen.

1.3.2.2 Java Pet Store Controller

The Controller for the Java Pet Store centers around a Front Controller servlet that encapsulates and dispatches service requests to the EJB tier. This section describes how the Controller executes calls on the Model in response to user requests, and selects and updates Views.

The JSP pages direct every browser POST or GET operation in the application to the URL of the Front Controller servlet. The operation being requested (create account, create order, etc.) arrives encoded in one of the data items in the request. The Front Controller directs the incoming request to the Request Processor, which translates the HTTP request into an application event corresponding to the requested operation. For example, an OrderEvent is a request to create a new order, and contains the data necessary to create the order.

The Request Processor sends the application event to the Shopping Client Controller, a Web-tier proxy for the Shopping Client Controller (stateful session) enterprise bean. The Shopping Client Controller proxy sends the application event to the Shopping Client Controller enterprise bean, and receives as a result a list of enterprise beans that may have been updated. The Shopping Client Controller then uses this update list to notify the affected Model beans, which refresh their con-

tents by pulling data from the enterprise beans they mirror. As noted above, the JSP pages that form the Views may later use the Model beans to present up-to-date Model data from the Web tier, without incurring a round-trip call to the EJB server.

Once the Request Processor has completed its task of dispatching the request, the Front Controller directs the Screen Flow Processor to select the next View and serve it to the client.

1.3.2.3 Java Pet Store Model

The Model in the Java Pet Store is implemented as a collection of enterprise bean classes. These enterprise beans provide pure business logic, with no reference to how (or if) the information will be displayed. This section describes how the Java Pet Store model works.

The core of the Model is the collection of enterprise beans that provide the business logic for the Java Pet Store. When the Shopping Client Controller enterprise bean receives a request event from the Web-tier Shopping Client Controller, it dispatches the event to an EJB tier "handler" class that fulfills the business request. After the handler executes, the Shopping Client Controller enterprise bean returns to the Web tier a list of names of enterprise beans that may have been updated. The Web tier Shopping Client Controller notifies the Web-tier beans of the update (as explained above), and the beans refresh their cached data by pulling new values from the enterprise beans they mirror.

For example, when the Shopping Client Controller enterprise bean receives an CartEvent (which manipulates items in the shopping cart) from the Web tier, it passes the event to a CartHandler, which actually changes the contents of the shopping cart. The Shopping Client Controller notifies the Web tier that the Cart object has changed, so the Model JavaBean in the Web tier representing the shopping cart refreshes its internal state from the ShoppingCart enterprise bean.

Model entity beans persist to the back-end database in the EIS tier using BMP (Bean-Managed Persistence). BMP in the Java Pet Store is vendor-independent, because it is written in terms of Data Access Objects, which abstract away the dependencies a resource may have on vendors or versions.

1.3.3 J2EE Design Patterns

A design pattern is description of a proven, reusable solution to a recurring design problem, emphasizing the context and forces surrounding the problem, and the con-

sequences and impact of the solution. A design pattern is the product of the collective experience of a group or community, not just a good idea dreamed up by a programmer. A design pattern is a formal way of describing a solution that is known to work in a particular situation, because the solution has worked repeatedly in the past. In short, design patterns allow you to learn from others' successes, instead of from your own failures.

J2EE BluePrints defines a set of design patterns useful for developing enterprise applications, based on experience with the Java Pet Store, communications with product vendors and J2EE application developers, and discussions with J2EE technology specification leads and expert group members. They express effective techniques for using J2EE technology to solve common enterprise application design challenges. Many of the design patterns described here appear in other contexts, and have been adapted here to the context of enterprise applications and J2EE technology. All of the patterns are used in the Java Pet Store, most of them in multiple places.

Following is a brief description of each of the J2EE BluePrints design patterns.

Model-View-Controller Architecture: Increase reusability by partially decoupling data presentation, data representation, and application operations. Enable multiple simultaneous data views.

Data Access Object: Decouple business logic from data access logic and adapt the resource being accessed, so that the type of resource can change easily and independently.

Fast-Lane Reader: Accelerate read-only data access by not using enterprise beans.

Front Controller: Centralize view management (navigation, templating, security, *etc.*) for a Web application in a single object that handles incoming client requests.

Page-by-Page Iterator: Efficiently access a large, remote list by retrieving its elements as sublists of value objects.

Session Facade: Provide a unified, workflow-oriented interface to a set of enterprise beans.

Value Object: Efficiently transfer remote, fine-grained data by sending a coarse-grained view of the data.

1.4 J2EE Design Topics

This final introductory section briefly describes common enterprise application design concerns. Each section below quickly outlines a particular design area addressed by design discussions in the book. J2EE BluePrints design discussions present contextual considerations that arise when designing applications, and provide design guidelines in terms of that context. Extended design discussions also explain in detail why J2EE technology works the way it does. Any technology can be used more effectively when the intent underlying its design is understood, and J2EE technology is no exception.

Each topic below contains pointers to in-depth discussion of the topic elsewhere in the book; either as an entire chapter, or as relevant parts of other chapters.

1.4.1 Packaging and Deployment

Packaging refers to assembling archives of class, resource, and configuration files into reusable components and/or applications. *Deployment* is integrating and installing packaged applications and components into a runtime environment.

A J2EE component or application is a collection of class, resource, and configuration files packaged into an archive file. These archives are modified ZIP files with standard filename extensions like ".jar" (Java ARchive) for application clients and applets, ".war" (Web ARchive) for Web applications, and ".ear" (Enterprise Application aRchive) for enterprise beans and other EJB tier technologies. The archive can be *deployed* as an single unit to extend the functionality of a client or server.

Each archive contains a *deployment descriptor*, which is an XML file that defines a component or an application's interface to its environment. A deployment descriptor defines the relationship between the component or application being deployed, the container inside which the component runs, and other information resources and services (such as server addresses) the component needs in order to operate in the environment. The deployment descriptor controls such functions as how components manage their persistent state, security restrictions, access to network resources like naming services and directories, and component transactional behavior. Once the application is packaged (that is, assembled into an archive along with a valid deployment descriptor), the archive can be transferred to the server for loading and execution. The server reads the archive file's contents, integrates the new component with existing application(s), and makes the new functionality available as a service.

Clients have some packaging and deployment controls, as well. For example, an applet and its supporting files can be packaged into a JAR file and delivered directly to a browser in a single request.

1.4.2 Transactions

The J2EE platform provides both programmatic and declarative control of the transactional behavior of J2EE components. A transaction enforces the semantic consistency of a system by grouping one or more operations into a single logical unit that is "ACID": that is, atomic (all operations occur once, or none of them do), consistent (a committed transaction leaves data in a usable, meaningful state), isolated (operations in one transaction are unaffected by uncommitted operations in another), and durable (committed updates are permanent). Transactions simplify application development by relieving programmers of many of the details of isolation and error handling.

The J2EE 1.3 specification requires that J2EE products support transactions for JDBC connections, JMS sessions, and Connectors (transactional resource adapters). Developers must be able to mix and match JDBC database accesses, transactional JMS message deliveries, and transactional Connector invocations within the same transaction context.

Transactions for enterprise beans can be controlled declaratively by setting transactional attributes in the beans' deployment descriptor. The attributes indicate the transactional behavior of each of an enterprise bean's methods. For example, a bean method whose transactional attribute is `Required` either uses an the existing transactional context if one exists, or creates an new transaction, invokes the method, and then tries to commit the transaction. Declarative transaction demarcation makes enterprise beans more flexible, since their transactional behavior can be configured at deploy time, instead of written into the source code of the component itself.

Transactions in JSP pages, servlets, and session beans may be controlled programmatically using the Java Transaction API (JTA).

J2EE applications can execute transactions distributed across multiple databases and EIS-tier resources using 2-phase commits.

The Transactions chapter of this book explains transaction control in detail, provides guidelines for using transactions in the Web and EJB tiers, describes how to configure transaction behavior at deployment time, and outlines the restrictions on what may and may not be done with transactions in various components.

1.4.3 Security

Application security is the subset of application functionality concerned with controlling access to information and resources. System security is responsible for:

- verifying the identity of users or other entities accessing the system,
- ensuring that system services are available only to authorized agents,
- protecting the integrity and confidentiality of information assets,
- preventing tampering or other abuse of system messages, and
- providing auditing paths to enforcing accountability.

J2EE provides services that support all of these security goals, except for auditing, which is optional.

Security is an essentially complex topic. This short introductory section can only touch on some of the major themes.

Authentication

Authentication is the process of verifying the identity of a user, system, or other entity, usually for the purpose of access to system resources. Authentication ensures that communicating parties are who they claim to be. For example, a user may need to prove his or her identity to a system by providing an authenticator, such as a password, known only to the creator of the identity. Or, a user may need to verify the identity of the system he or she is accessing, to ensure that information is being sent to an authentic receiver. A system may need to authenticate itself with another system before being allowed to request the latter system for a service. *Mutual authentication* is when a two or more users or systems authenticate each other bidirectionally.

The authentication requirements of J2EE components are defined in the deployment descriptor for the component, or may be controlled programmatically. Component containers enforce the authentication constraints on the components they host.

J2EE-compliant Web and EJB containers, as well as application clients, must support the Java Authentication and Authorization Service (JAAS 1.0) APIs. JAAS is a comprehensive set of APIs for programmatically controlling authentication and authorization, regardless of what underlying security technology (Kerberos, NIS+, etc.) is used.

Authorization

Authorization is the process of enforcing constraints on access to system resources. Authorization constraints are based on *permissions* defined for access to the resources, and *capabilities* assigned to the users or entities trying to access the resources.

The Java 2 Platform, Standard Edition (J2SE™) security model defines permissions for many system-level functions, such as creating files, opening sockets, and so on. These mechanisms are available for authorization control for components in the Client tier.

In addition to standard J2SE platform permissions, the J2EE platform defines authorizations for access to Web and EJB tier resources based on the concepts of *principals* and *security roles*. A *principal* is anything that can be authenticated by an authentication protocol. The principal consists of a name and authentication data, and represents the identity of some agent (user, system, or component) that is trying to access a resource. Principals can be mapped into *roles*, which are logical groupings of users, and then permissions for access to resources are granted to those roles. A principal must be properly authenticated before it can be considered to be "in" a role. Once in a role, a principal has access to all resources defined by that role.

Authorization for access to system resources, such as a group of URLs in the Web tier, or the ability to execute a method on an enterprise bean in the EJB tier, can be controlled declaratively or programmatically. Declarative security lets a deployer configure the security of a component in the deployment descriptor for the component, freeing developers from the difficulties of managing security manually. Declarative security also makes components more flexible, since their security behavior can be changed without having to modify application code; and is more portable, because the components don't have to be written in terms of any particular security technology. The container manages all of the security, and application components focus on delivering business functionality. Programmatic security provides more control, at the potential cost of portability and flexibility.

As with authentication, authorization policies are enforced by the container, for both declarative and programmatic authorization control. Authorization comes after authentication; that is, identity is always determined first, and then the rules about which identities have access to which resources are applied.

Integrity and Confidentiality

Integrity of data in a system refers to ensuring that data can not be modified without detection. *Confidentiality* addresses the question of data privacy, ensuring that data are accessible only to authorized parties. In an application, data integrity and confidentiality are controlled by authentication and authorization, as described above.

Distributed systems rely on messaging to move data to where it is needed. Integrity mechanisms such as digital signatures and message digests prevent messages from being intercepted and modified or used inappropriately, or from being sent by some party and later repudiated. Confidentiality mechanisms such as key exchange and encryption protect sensitive data in messages from unauthorized access.

Auditing

Auditing is the process of logging security-related events for the purpose of analyzing security risks or breaches, and for assigning accountability when problems occur. Such logs are of use for deciding who is responsible for a security breach, for explaining how a breach occurred, and for analyzing the extent of the damage (in terms of compromised confidentiality or unauthorized use) caused by the breach.

The J2EE specification does not specify any particular requirements or APIs for auditing in J2EE systems. As such, auditing functionality will vary from one J2EE Product Provider to another.

1.4.4 Portability

J2EE application *portability* allows an appropriately configured application or component to be deployed in environments from multiple J2EE Product Providers without changing the source code. J2EE offers several levels of portability, each of which corresponds to a particular specification:

- *Source-code portability* allows a single source base to produce the same results, regardless of hardware platform, operating system, runtime system, or compiler. The Java language is highly source-code portable because its specification clearly spells out such details as byte order, memory management behavior, and the size and format of primitive types.
- *CPU portability* means that a compiled program will run identically on different CPUs. The Java Virtual Machine provides CPU portability by specifying a virtual instruction set to which Java (or other) language source code compiles,

isolating the compiled code's behavior from the underlying CPU.

- *OS portability* allows a developer to write a program that accesses system-level resources in a uniform way, regardless of the underlying operating system. The Java Platform provides OS portability by defining a "virtual operating system" that gives developers a unified model of system services.
- *J2EE application portability* allows J2EE application developers to write client/server components and applications that deploy and execute identically regardless of the underlying server implementation or vendor. The J2EE Platform Specification defines how the various J2EE platform technologies (servlet containers, EJB containers, transactions, security, and so on) must behave, so J2EE application developers can depend on consistent behavior across vendor implementations.

Application portability is one of the major value propositions of the J2EE platform, because it eliminates vendor lock-in, returning control of information assets, both data *and* business rules (encoded in software), to the owners of those assets. No longer are mission-critical data and business processes held hostage to representation in vendor-proprietary formats and technologies.

The promise of freedom of technology choice can only be realized if an application is truly portable between J2EE implementations. And sometimes, it may be acceptable to trade away some portability for performance gains or features that go beyond what the J2EE specification requires.

1.4.5 Internationalization

As business goes global, information systems need to be able to present data and interact with users in their own language. *Internationalization* is designing an application to be deployable in multiple human languages. A *locale* is a set of properties that indicate to an application how data are to be displayed and stored. *Localization* is the process of customizing an internationalized application to a specific locale.

Internationalization is not limited to user interface; it has effects in all tiers. Some internationalization issues include:

- Character encodings: different languages have different ways of encoding characters, and some languages have tens of thousands of characters, requiring multibyte character representations. Processing and storing multibyte strings

has implications for all tiers.

- View layout: the way a page or interface is laid out or displayed may vary by locale; for example, Arabic and Hebrew read right-to-left, and page or interface layout may change to adapt to this difference.
- User interface resource management: internationalized applications typically store user interface resources such as label text separate from the source code, and substitute the text corresponding to the current locale at runtime. Managing these resources effectively and cleanly is an important design goal.
- Data value format: data types such as time, date, currency, and so on vary from culture to culture. The Java Platform provides ways to format data appropriately for a particular locale.

1.4.6 XML-based Web Services

A *Web service* is, simply put, application functionality made available on the World Wide Web. A Web service consists of a network-accessible service, plus a formal description of how to connect to and use the service. The language for formal description of a Web service is an application of XML. A Web service description defines a contract for how another system can access the service for data, or in order to get something done. Development tools, or even autonomous software agents, can automatically discover and bind existing Web services into new applications, based on the description of the service.

For example, imagine the e-commerce site of a company that ships orders of physical goods to customers. The customer may want to request an estimate of the total shipping cost for goods ordered. If shipping companies offer a shipping rate quotes Web service, the merchandise ordering application can contact the shipper, request a quote (by invoking the service as its description requires), and present the resulting estimate to the customer.

Web services often encode requests for services as XML, often in the form of SOAP (Simple Object Access Protocol) messages. A SOAP message is a form of remote procedure call, with XML as the marshalling format, and usually using HTTP as the transport protocol. The standardized syntax of XML provides an open, uniform, and portable way to exchange information.

J2EE is the most mature platform available for creating and deploying XML-based Web services today. Enterprise beans can easily be exposed to the World Wide Web as XML Web Services. Servlets and JSPs in the Web tier provide

enabling technologies for presenting Web service functionality as XML (or in other formats), either directly or by delegating to enterprise beans. Full support for existing interoperability standards such as CORBA, and the flexibility of Java Connectors make J2EE an excellent choice as for integrating legacy applications with the Web, either as Web applications or as Web services. The J2EE specification requires that J2EE implementations support the Java API for XML Parsing (JAXP 1.0), which provides uniform access to any XML parser conforming to the W3C Recommendations for DOM 2 or SAX 2.

1.4.7 Messaging

Enterprise application architecture is rapidly evolving toward a model of loosely-coupled, autonomous network services, collaborating via reliable, secure, transactional message passing. While a great deal of attention is currently being paid to Web services, such RPC-style communication has its limitations. The Java Messaging Service (JMS) provides the necessary infrastructure for robust, reliable, synchronous or asynchronous messaging in enterprise applications.