CHAPTER 3

J2EE Overview

Over the years, the Java technology platform has grown out of its original applet client/server origins into a robust server-side development platform. Initial platform packages introduced built-in threading support and provided abstractions to I/O and networking protocols; newer versions of the Java Software Development Kit (SDK) continued to enhance these abstractions and introduce newer framework offerings.

The momentum of producing technology frameworks supporting enterprise server-based development has continued, and has been formalized into the J2EE platform offering. The motivation of this offering is to provide developers with a set of technologies that support the delivery of robust enterprise-scale software systems. IT professionals are presented with an ever-changing business and technology landscape. Technology professionals must balance the demands for new automation requirements against the existence of existing line of business applications; simply using the technology du jour perpetuates the problem of integrating existing legacy systems. The goal of the J2EE platform is to offer a consistent and reliable way in which these demands can be met with applications that possess the following characteristics:

- **High Availability**—Support and exist in a 24/7 global business environment.
- **Secure**—Ensure user privacy and confidence in business function and transactions.
- **Reliable and Scalable**—Support high volumes of business transactions accurately and in a timely manner.

This chapter offers an overview of the J2EE architecture, a brief discussion of the specification's component design and the solutions they provide, and describes which J2EE technologies this book will focus on.

First, some background.

All J2EE technologies are built upon the Java 2 Standard Edition (J2SE). It includes basic platform classes, such as the Collections framework, along with more specific packages such as JDBC and other technologies that support client/server-oriented applications that users interact with through a GUI interface (e.g., drag-and-drop and assistive technologies). Note that platform technologies are not limited to framework implementations. They also include development and runtime support tools such as the Java Platform Debugger Architecture (JPDA).

Technologies specific to developing robust, scalable, multitiered server-based enterprise applications are provided within the J2EE platform offering. While still supporting client/server-based architectures, J2EE platform technologies provide support for distributed computing, message-oriented middleware, and dynamic Web page development. This chapter and most of this book will deal specifically with some of these technologies. In particular, WebSphere 5.0 (the focus of this book) implements the J2EE 1.3 platform specification. A list of the technologies from J2EE 1.3 (along with the supported levels) is shown in Table 3.1.

Table 3.1 J2EE technologies.

Supported Technology	Level required by J2EE 1.3
Java IDL (Interface Definition Language) API	(Provided by J2SE 1.3)
JDBC Core API	2.0 (Provided by J2SE 1.3)
RMI-IIOP API	(Provided by J2SE 1.3)
JNDI API	(Provided by J2SE 1.3)
JDBC Extensions	2.0
EJB (Enterprise Java Beans)	2.0
Servlet API	2.3
JSP (JavaServer Pages)	1.3
JMS (Java Message Service)	1.0
JTA (Java Transaction API)	1.0
JavaMail	1.3
Java Activation Framework (JAF)	1.0
JAXP (Java API for XML Parsing)	1.1
Java 2 Connector Architecture (J2C)	1.0
JAAS (Java Authentication and Authorization Service)	1.0

Table 3.2 J2EE 1.4 technologies implemented by WebSphere 5.0.

J2EE 1.4 Technology
JAX-RPC (Java API for XML-based RPC)
SAAJ (SOAP with Attachments API for Java)
JMX (Java Management Extensions)

In addition to the required technologies for J2EE 1.3, WebSphere Application Server 5.0 implements a number of J2EE-compatible technologies in advance of support of J2EE 1.4. In particular, WebSphere also supports technologies which will be required in J2EE 1.4 (Table 3.2).

3.0.1 J2EE Component Design

One of the most appealing features of object technology is its ability to combine function and data into a single element, also referred to as an object. Arguably, a single object implementation could be classified as a component, but components offer more functionality than providing access to data and performing functions against this data. Flexibility is achieved with designs that can consist of multiple classes related through composition and inheritance. The word component implies that they are a part of something whole, indicating that components require some kind of reference problem space where they can be applied. The J2EE specification provides this frame of reference for components that can be used, extended and combined by developers to deliver robust enterprise applications.

J2EE components defined for the platforms exploit the OO nature of Java by applying design patterns that provide both white and black box extensibility and configuration options. The platform components use inheritance and composition throughout their design, providing a way for custom configuration by developers. Also, defining components in an abstract way can allow systems built using those components to work regardless of how each vendor implements each concrete component implementation.

Studying these design techniques employed in the platform implementations can help make your own designs more elegant. These object design techniques are nothing new and have been applied throughout the years in other OO languages. Two design themes that take different approaches in supporting component configuration are discussed in the following sections.

3.0.2 Configurable Implementations

A specific design technique often used in the J2EE platform is the notion of describing completely abstract designs (through the use of interfaces) that allow the entire implementation to be configurable. This means that developers are aware of, and have visibility to, a set of interface types without regard to how they are implemented; implementation is the vendor's responsibility. This allows developers to choose the best available solutions. Figure 3.1 shows the dual relationship interfaces create between developers and vendors.

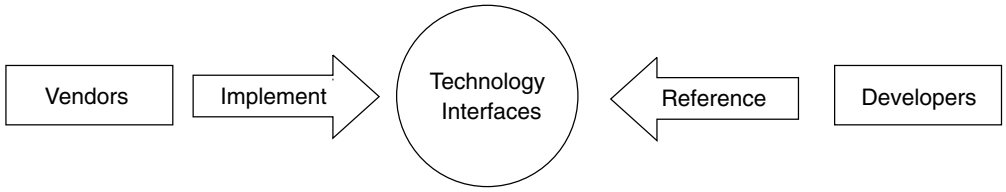

Figure 3.1 Relationship between developers and vendors.

3.0.3 Configurable Algorithms

Not all technology implementations are exclusively interfaces. Most have a combination of generalized class definitions that interact with interface types. Consider the servlet package; it provides a `javax.servlet.GenericServlet` implementation that is defined abstractly along with providing a servlet interface type. While this may seem redundant, designers of the servlet API have provided a way for developers to take advantage of an abstract configurable algorithm, and have provided an abstract configurable implementation that can serve as the basis of a concrete implementation. (For more on this dual nature, see the discussion comparing abstract classes and interfaces.)

ABSTRACT CLASSES OR INTERFACES?

Inheritance is a feature of the OO paradigm that captures the imagination of developers when they first encounter this technology. The ability to define and classify hierarchies of data structures and create state and behavior that is extended for specific functionalities provides an excellent way to deliver solutions that can be extended in a white box manner.

White box-based designs utilize inheritance by implementing a base class that is extended by developers, and the appropriate elements are overridden with the desired functionality. Java provides language constructs that help communicate what can and cannot be overridden at construction time. Methods and class definitions can be defined as abstract, requiring developers to supply concrete implementations. Access modifiers such as final and private can be utilized to prohibit methods from being overridden. Combining these elements effectively can yield what is referred to as a configurable algorithm. The base class implements generalized methods that perform a set of algorithmic steps. Within this scope of base methods calls, abstract methods appear that are overridden, fulfilling a given method's implementation.

Using inheritance exclusively can result in deep hierarchies that may lead to coupled implementations, usually the result of an abstract design that requires a large number of abstract methods to be implemented. Java interfaces provide an alternative abstract mechanism that allows for a more independent implementation without regard to an existing hierarchy.

Inheritance is useful for designs that have algorithms or behavior that can be generalized and utilized by extending classes. Designs that require most or all of its implementation to be defined by extending classes can be

communicated using interface definitions. Implementers are given complete freedom in how the interface methods carry out their operations. However, interfaces enforce a more rigid contract, and changing an interface design can make a larger impact on existing implementations. Therefore, an effective way to evolve a design, in lieu of booking a lot of initial design time, is to initially utilize inheritances and let an abstract design evolve. Once the required signatures have been discovered, and it turns out that a configurable implementation is necessary, interface(s) can be produced.

Configurable implementations utilizing interfaces are the underpinnings of providing vendor-independent J2EE technology designs.

Interfaces and effective abstractions are the means by which J2EE components achieve vendor neutrality. The J2EE specifications simply define the APIs, types, life cycles, and interactions of objects within the technology frameworks. Vendors can then apply their efforts toward the agreed-upon contracts and specifications. Developers write to these specifications. You may ask: "Won't that create a dependency on these contracts, and if they change,won't my code be affected?" The short answer is yes, you are dependent upon versions of these contracts, but engaging them in a consistent way and knowing that they are community supported should help minimize this concern.

In addition to describing WebSphere Application Servers as a J2EE implementation product, this book will provide patterns and approaches for neutralizing this dependency.

3.0.4 Who Defines These Specifications?

Another key advantage of Java and the J2EE standard is the way in which component solutions are identified and defined. Early on, Sun promoted the openness of the Java language, initially by giving it away.

Advancement of Java technology and the formulation of the J2EE specification have been carried out by the JCP (Java Community Process). Community is the operative word; any interested individual or organization can participate. For individuals participation is free; organizations pay nominal dues. Delegates from the membership propose, review, and accept technology specification proposals. While not an open source initiative, but under a community license that still allows Sun to be steward of the language, the JCP encourages community participation.

Ideas are proposed through the creation of a Java Specification Request (JSR). Members evaluate and vote on the JSR for merit. Once accepted the JSR becomes an official technology component and goes through the design and development process by a committee made up JCP members—usually a cross section of well-known vendor members.

The advantage of community participation is the proliferation of new frameworks/components that are derived and designed from a wide point of view, arguably larger than proprietary-based technology that may be more influenced by market pressures. These market pressures still exist in the JCP environment, but the checks and balances of the membership can make them have less influence over the manner in which the problem is solved.

Of course, there is a down side to this approach. Whereas a company can be very nimble in getting a solution out the door by using a custom-built design, standardized solutions must receive approval and validation from the community which can take time.

3.1 Why J2EE?

Reuse is an adjective that can beckon the attention of developers, managers, and bean counters. It promises savings in the form of shorter development efforts and higher quality. Unfortunately, reuse has been oversimplified, and overhyped, resulting in a minimized impact. For instance, some reuse (of the base class libraries) occurs just through using Java as a programming environment. If you add J2EE components, more reuse occurs. Of course, this is not the business domain type of reusability that would allow us to snap together applications as in the proverbial IC chip analogy made by Brad Cox. Nevertheless, Java's OO nature and the standards of J2EE are a progression toward achieving high degrees of reuse.

J2EE-based technologies provide what can be classified as horizontal technology reuse (Figure 3.2). Contracts, primarily in the form of Java interfaces, allow developers to use vendor-supplied technology solutions with a high degree of transparency. Imagine if the JDBC specification did not exist and developers had to write directly to vendor-supplied APIs; of course, then-

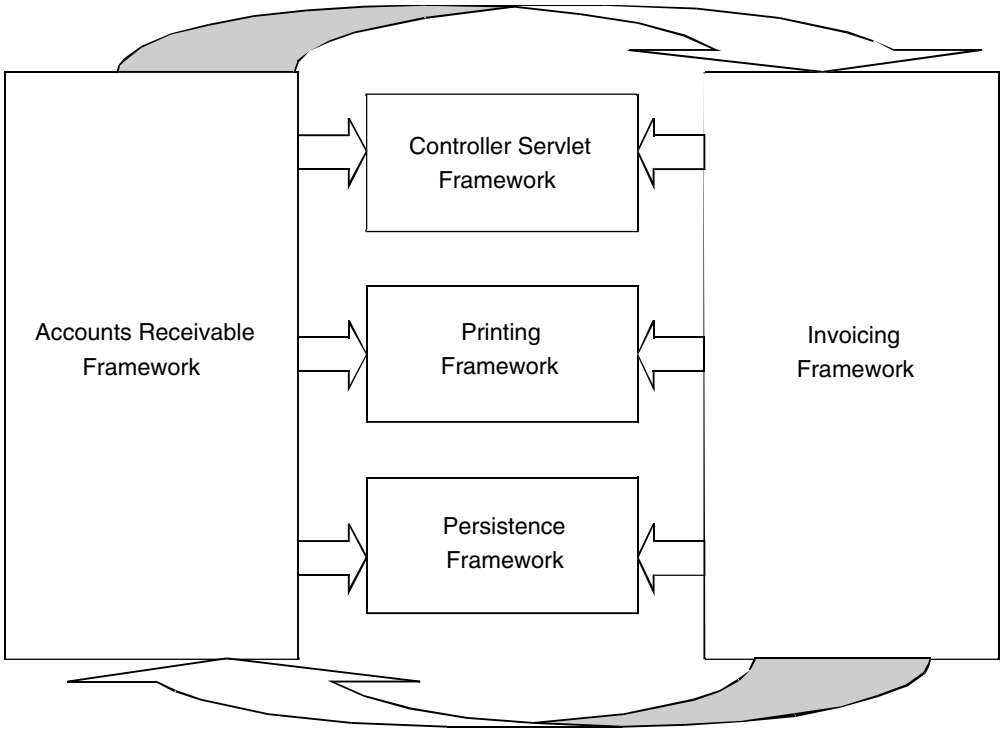

Figure 3.2 Horizontal technologies.

good OO developers would build designs that would decouple and wrapper vendor-specific APIs with a neutralized access API. Even though SQL is also a standard, each new target SQL-based data source would require a modification to this neutral API.

Fortunately, the JDBC specification allows the burden of access APIs to be moved to vendors. Developers simply acknowledge the specified contracts and generalized implementations and use them in applications to execute SQL against any vendor honoring the JDBC specification, which most, if not all, do.

Other horizontal technologies are vendor neutralized in a similar approach, allowing developers to concentrate on application-specific logic by using standards-compliant solutions. This frees developers from worrying about having to produce or refactor a horizontal implementation. Instead, the best vendor-supplied solutions can be engaged, resulting in shorter delivery times of applications that are robust and scalable.

3.2 J2EE Architecture

Planes, trains, and automobiles are all assembled using well-accepted blueprints and parts supplied by countless vendors. One way that this is carried out is through industry-accepted blueprints that define specifications for construction and how they are to be used. Under this same premise, the J2EE specification defines these interfaces, their life cycles, and interactions they must carry out. The specification also describes roles that can be held by resources involved in the development and deployment of server-based applications.

The J2EE specification introduces an architectural concept of a container. Containers are defined to house J2EE components within a layer boundary. Containers manage component relationships within tiers and resolve dependencies of components between these tiers. Figure 3.3 illustrates the J2EE containers and their associated component dependencies.

To understand where these components exist within the topology of an application, consider that a given application can be partitioned as follows:

- **Client Container**—User interface implementation resident on a client workstation.
- **Web Container**—Server-based user-interface implementation accessed via HTTP.
- **EJB Container**—Captures and defines enterprise business data and function; provides a mechanism for distribution of business objects and for transactional support of complex business interactions.
- **Information Systems Back End**—A database, messaging system, or EIS that provides data and functions to the system.

Applications may utilize all or, at a minimum, the client and Web tiers; within each tier J2EE technologies will be engaged to perform application functions. Some will occupy an obvious tier, as is the case with the JSP/Servlet technologies. Obviously, these belong in the Web tier. Other technologies play a supporting role and may appear in any or all tiers. For instance, it's easy to see the requirement of interprocess messaging (JMS) appearing in all of the client, Web, and EJB tiers.

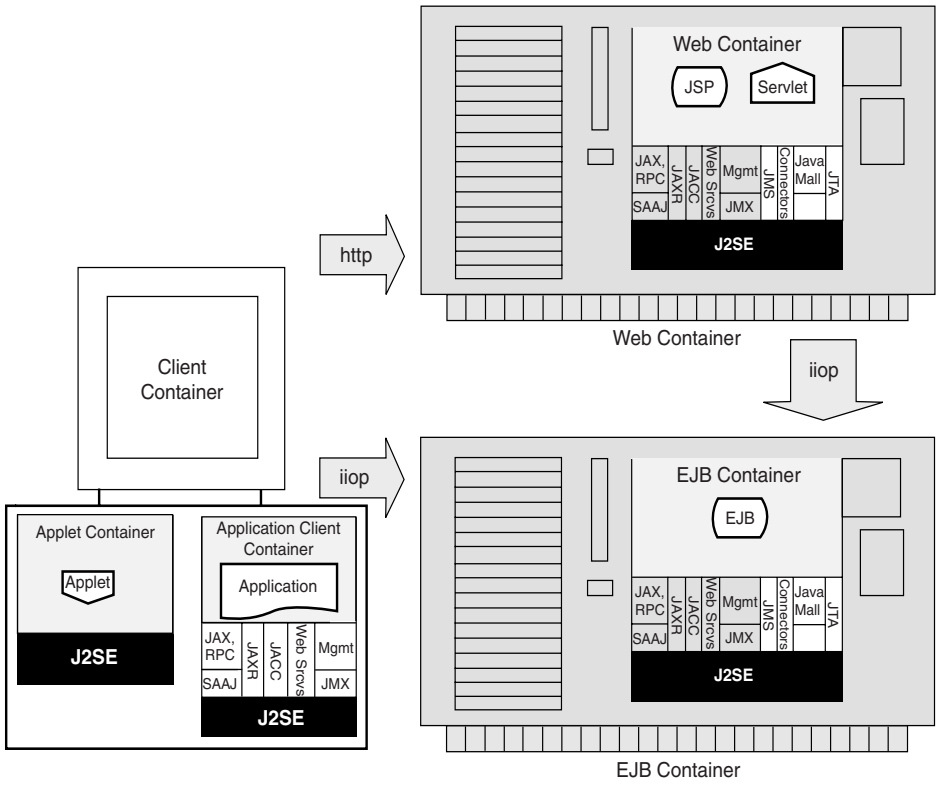

Figure 3.3 Container diagram.

Notice the presence of J2SE in every container definition diagrammed in Figure 3.3. This reflects the foundation for all containers. Other technologies shown may or may not appear within a container definition because they are determined by application requirements. The following sections describe components defined within container boundaries.

3.2.1 JDBC

Potentially the catalyst technology for Java, JDBC allows developers to interact with vendor-enforced JDBC data sources using generic interfaces. Statement execution, connection resolution, and result set processing can be carried out using the specification interfaces. Although in most cases the data source is relational-based, the specification interfaces does not require this.[1] This allows developers to execute SQL in a vendor neutral fashion.

1. The JDBC design is slanted toward row-based result sets, therefore the majority of JDBC support comes from relational database vendors.

3.2.2 Servlet/JSP

Servlet technology is the mechanism used to create dynamic Web pages, in the same spirit that early Common Gateway Interface (CGI) technology was used to provide a personalized interaction with a Web site. Servlet technology allows browser resident clients to interact with application logic residing on the middle tier using request and response mechanisms of the HTTP protocol.

JSP technology is built upon servlet technology. Its purpose is to help blend HTML-based page definition and dynamic-based Java expressions into a single HTML-like document resource.

3.2.3 EJB

EJBs support the ability to create distributed components that can exist across Java application process boundaries and server topologies. More than simply providing access to distributed objects, the specification supports transactions with two-phase commit support, security, and data source access.

EJB technology is utilized to help support scalable application architecture by making enterprise business logic and data available and accessible to Web container function. EJBs' ability to support transactions across server boundaries in a distributed fashion is key to supporting large-scale, transaction-based applications.

3.2.4 Connector

EJB technology provides a distributed transaction-based environment for external resources. In many, but not all, cases these sources are relational based. The connector specification provides a mechanism for EJB technology to interact with other, non-relational resources in an implementation-independent manner.

3.2.5 JMS

JMS provides vendor-neutral point-to-point and publish/subscribe messaging solutions. The JMS service provider will provide an implementation based upon the JMS APIs. JMS is the primary mechanism in J2EE for allowing asynchronous communication between components. It can be used to provide asynchronous update of components running in networked client containers, or it can be used to allow asynchronous communication with back-end EISs.

3.2.6 Java Mail

This technology is a framework that implements an interface to an e-mail system. The framework is provided with J2EE in binary form. Also included is a set of APIs that support POP3 and SMTP mail protocols. While we will cover the other core J2EE APIs in this book, we will not cover Java Mail in any depth because, in truth, this API is rarely used.

3.2.7 JTA

Transaction support is abstracted using Java Transaction API (JTA). This API provides a generic API that allows applications, applications servers, and resource managers to participate in defining and executing heterogeneous transaction boundaries.

3.2.8 JAX-RPC

Java API for XML-based RPC (JAX-RPC) allows Java developers to create client and end-point Simple Object Access Protocol (SOAP)-based Web service functions. Developers can utilize Java-based classes to define Web services and clients that exercise Web-services, effectively shielding the developer from the complexity of interacting with the SOAP protocol. As with SAAJ and JMX, JAX-RPC is a required part of the J2EE 1.4 platform.

3.2.9 SAAJ

This technology (SOAP with Attachments API for Java) provides a Java API that allows the formatting of XML messages in conformance with the SOAP specifications. This should not be confused with JAX-RPC, which also supports SOAP, but provides Web services support for message composition and support for SAAJ, which allows the attachment of MIME-encoded binary documents to SOAP messages. SAAJ is a required part of the JAX-RPC API, so we will discuss it only within the context of JAX-RPC.

3.2.10 JMX

Java Management Extension (JMX) allows a generalized way that distributed and Web-based applications can provide monitoring and instrumentation services, independent of the vendor application server. We won't discuss programming to this API in WebSphere, but we will discuss how it is used in WebSphere administration.

3.3 J2EE Platform Roles

Besides defining a standard blueprint for vendor neutral enterprise computing components, the J2EE specification identifies roles that participate in producing, creating, and supporting information systems built upon the J2EE platform. These roles as defined in the J2EE 1.3 specification are described in the sections which follow.

3.3.1 J2EE Product Provider

The role is responsible for providing the J2EE containers that house the specification components. In addition, they are required to provide deployment and management tools used to manage J2EE applications deployed to the product. IBM plays the role of a product provider with its WebSphere application server product.

3.3.2 Application Component Provider

This role identifies the provider of components such as enterprise bean developers, and HTML document designers, and programmers that create components used to produce J2EE applications. This book exists primarily to educate developers who will fill this role.

3.3.3 Application Assembler

The act of using J2EE components to construct an application is the role defined by the specification—an application developer. Assembly implies that components are created and defined within an Enterprise Archive (EAR) file for deployment to containers. We will discuss integrating J2EE components and packaging them as EAR files for deployment.

3.3.4 Deployer

The deployer is responsible for deploying enterprise Java components into an operating environment that has a J2EE server supplied by a product provider. Deployment is typically made up of three steps: (1) installation, which involves moving the application (.ear) to the server environment; (2) configuration of any external dependencies required by the resource; (3) Execution of the installed application.

While our primary focus is application development, and not deployment, we will discuss areas where the two roles meet.

3.3.5 System Administrator

This role is not new to the J2EE landscape. Administrators are responsible for configuring and monitoring the operating environments where J2EE servers exist, tasks are accomplished by using the appropriate tools from the J2EE product provider. We will not examine this role in our book. For more information on performing system administration with the WebSphere family of products, see [Francis] or the WebSphere Application Server InfoCenter.

3.3.6 Tool Provider

Tool providers furnish tools that help with the construction, deployment, and management of J2EE components. Tools can be targeted to all platform roles defined by the specification. This book describes WebSphere Application Developer, used to develop components, making IBM a tool provider.

Currently, the specification does not require or provide a mechanism by which these tools are standardized. However, the specification has referenced a possibility of this being so in the future.

3.4 J2EE Versions and Evolution

Java's momentum has moved it from a niche programming language into a mainstream language that is robust enough for a spectrum of applications from scientific to business. Java's object-based environment allows the fundamental language to remain relatively stable with extensions coming in platform technologies, such as those defined in J2EE. Developers and vendors fulfilling these technology specifications with best-of-breed implementations have arguably formed a so-called critical mass. There is no reason why this momentum should not continue; new versions of current technologies along with new technologies will continue to augment the J2EE technology platform. Already, a single J2EE version increment from 1.3 to 1.4 has introduced Web services technology and XML support.

What does this mean to the Java developer? One point of view is that change in designs and APIs leads to a maintenance nightmare. Another, more optimistic view, is that smaller development cycles will result in more stable and robust software. Developers can protect themselves from API creep through consistency, generalization, and the application of design patterns. This book will not only describe how these technologies are used, specifically with IBM WebSphere, but will provide patterns that can be used to implement these technologies in a malleable way.

3.5 A J2EE Perspective

It is not enough to download the J2EE platform components and start writing enterprise applications with just any tool. Choosing the right development environment and application server determines whether complexity will be shielded and managed, or be an ever-present struggle during the development process. Realizing the full potential of J2EE technologies requires more than just a tool—it requires a pattern-based approach that engages tool-produced artifacts.

Many choices are available to the developer who wants to use J2EE technology. Some are free, others are vendor-supported. Obviously if you are reading this you are interested in the IBM WebSphere suite of products. The remaining chapters capture and describe the complete cycle from development to deployment utilizing the WebSphere Studio Application Developer product and its integrated tooling support. Besides providing a complete tutorial on how to utilize these tools and the application server, we will describe design approaches and patterns that can help make your development and deployment process, as well as the resulting software, flexible to better meet changing business needs.

As we discussed in Chapter 1, we see the idea of layered application architecture as being critical to J2EE, and to understanding the architecture of the WebSphere product family. Figure 3.4 illustrates how the different technologies we've just covered fit into that layered architecture.

At the top of our architecture is the presentation layer. We'll discuss providing presentation layers based on HTML using Java servlets, JSP, and eXtensible Stylesheet Language Transformations (XSLT). XSLT, not a J2EE technology, is a mechanism for transforming XML documents into HTML (commonly used along with servlets and JSP). We will also consider Web services in this layer, even though they are considered to be a program-to-program communication mechanism. We'll also examine how to test servlet-based applications using the open-source HTTPUnit tool.

Next comes the controller/mediator layer, which captures the notion of application flow and adapts the domain model layer to the presentation layer. We'll examine several ways of implementing controller logic, including implementing it with servlets, using the Struts open-source application framework, and even using message-driven beans (which are EJBs called through JMS) as application controllers for asynchronous logic flows.

In the domain layer, we'll examine how to implement domain logic using Java Beans (or, more correctly, Plain Old Java Classes) and EJBs. We'll also show you how to test your domain logic using the open-source JUnit toolkit. To support the persistence of objects in the domain

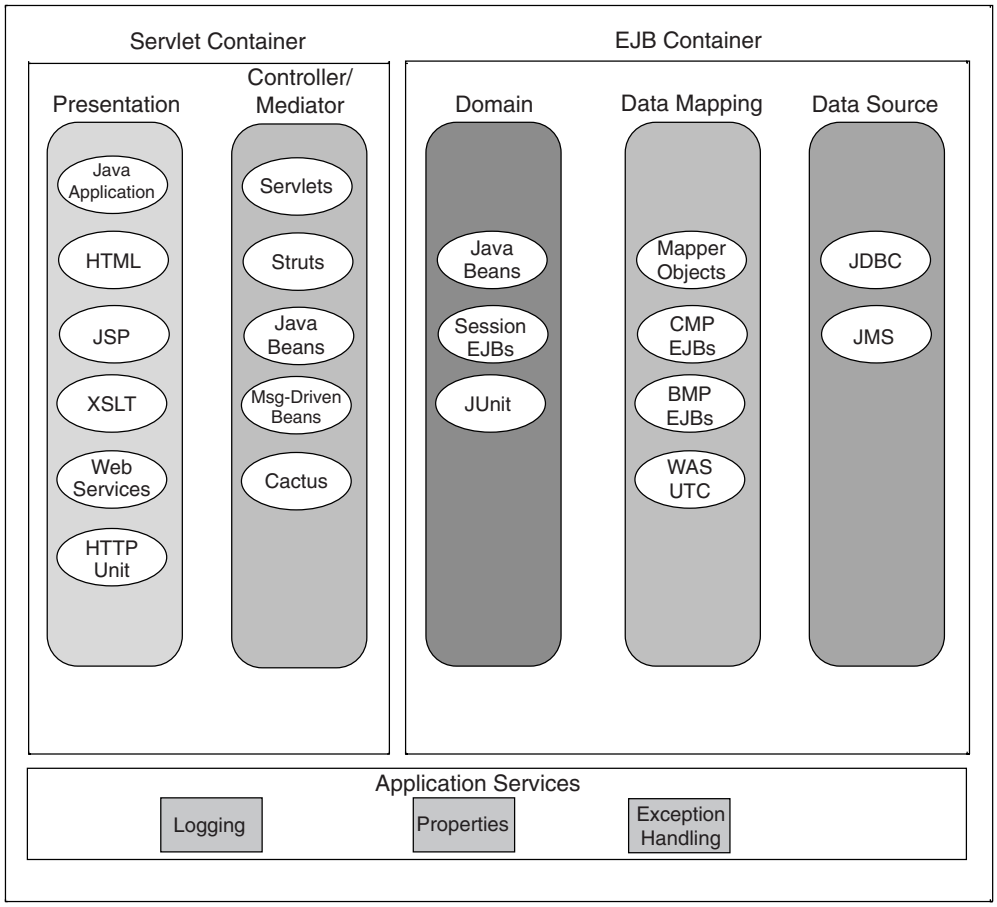

Figure 3.4 Layered J2EE application architecture.

layer, we'll examine the mapping layer in depth, discovering how to use mapper objects, bean-managed persistence (BMP) and container-managed persistence (CMP) entity EJBs.

In addition to showing you how to test with JUnit, we'll show you how to use the WebSphere Studio Universal Test Client. Finally, we'll examine the two most common sources of data for J2EE programs, JMS and JDBC.

While the book will proceed roughly in the order we've outlined, it won't cover the layers in strict order because not every system uses every technology we've described. Instead, you can rely on the application architecture graphic (Figure 3.4), which will appear at the beginning of every chapter starting in Chapter 5, to help you understand where the technologies fit into the overall J2EE architecture.

3.6 Summary

This chapter described Java's evolution from an initial way of delivering client/server type applications via the Web, to a robust OO platform that can support large-scale multiuser enterprise applications. With the addition of J2EE-based technologies, which are supported by the Java community at large, Java technology is a viable choice for developing applications of varying deployment topologies, platforms requirements, and business requirements.