Patricia Seybold Group

Strategic Technologies, Best Practices, Business Solutions

Enterprise JavaBeans™ Technology

Server Component Model for the Java™ Platform

By Anne Thomas
Revised December 1998

Prepared for Sun Microsystems, Inc.

85 Devonshire Street, 5th Floor, Boston, MA 02109 • Phone 617.742.5200 • Fax 617.742.1028 • www.psgroup.com

Table of Contents

Illustrations and Table

Enterprise JavaBeans™ Technology

Server Component Model for the Java™ Platform

By Anne Thomas, Patricia Seybold Group
Revised December 1998

Prepared for Sun Microsystems, Inc.

Introduction

Enterprise JavaBeans Technology

Enterprise JavaBeans™ (EJB) technology defines a model for the development and deployment of reusable Java™ server components. *Components* are pre-developed pieces of application code that can be assembled into working application systems. Java technology currently has a component model called JavaBeans™, which supports reusable development components. The EJB architecture logically extends the JavaBeans component model to support server components.

Server Components

Server components are application components that run in an application server. EJB technology is part of Sun's Enterprise Java platform, a robust Java technology environment that can support the rigorous demands of large-scale, distributed, mission-critical application systems. EJB technology supports application development based on a multitier, distributed object architecture in which most of an application's logic is moved from the client to the server. The application logic is partitioned into one or more business objects that are deployed in an application server.

Java Application Servers

A Java application server provides an optimized execution environment for server-side Java application components. By combining traditional OLTP technologies with new distributed object technologies, a Java application server delivers a high-performance, highly scalable, robust execution environment specifically suited to support Internet-enabled application systems.

WORA

The Enterprise JavaBeans architecture defines a standard model for Java application servers to support "Write Once, Run Anywhere™" (WORA) portability. WORA is one of the primary tenets of Java technology. The Java virtual machine (JVM) allows a Java application to run on any operating system. But server components require additional services that are not supplied directly by the JVM. These services are supplied either by an application server or by a distributed object infrastructure, such as CORBA or DCOM. Traditionally, each application server supplied a set of proprietary programming interfaces to access these services, and server components

have not been not portable from one application server to another. For example, a server component designed to run in BEA Tuxedo could not execute in IBM TXSeries without significant modification. The EJB server component model defines a set of standard vendor-independent interfaces for all Java application servers.

Component Portability

Enterprise JavaBeans technology takes the WORA concept to a new level. Not only can these components run on any platform, but they are also completely portable across any vendor's EJB-compliant application server. The EJB environment automatically maps the component to the underlying vendor-specific infrastructure services.

Scenario

Stock Fund Account

For example, Money Makers, a large brokerage house, is using Enterprise JavaBeans technology to implement an application system to manage stock fund accounts. Money Makers wants to provide clients with a self-service electronic trading system. To effectively support both broker and client users, the application is implemented using a thin-client, distributed object architecture. The architecture of the application is depicted in Illustration 1.

Multiple Client Support

The stock fund application can support a variety of client devices, including desktop workstations, Web browsers, telephones, kiosks, smartcards, or other Internet-enabled appliances.

Communication Protocols

Client applications can communicate with the stock fund application using a variety of protocols. Java technology clients invoke the application using the native Java Remote Method Invocation (RMI) interface. RMI requests currently are transferred using the Java Remote Method Protocol (JRMP). In the future, RMI will be extended to support the industry-standard Internet InterORB Protocol (IIOP). Native language clients can invoke the application using CORBA IDL running over IIOP or a COM/CORBA internetworking service running over IIOP. The RMI client proxy could also be rendered as an ActiveX control to provide easy integration with any Windows application. Browsers can invoke the application through a servlet running on the HTTP server. The browser communicates with the servlet using HTTP, and the servlet communicates with the application using RMI.

Server Execution Environment

Money Makers has millions of clients and expects that the transaction volume on this system will be very high. Therefore, the company selected an EJB-compliant server from Big Guns System Software deployed on a fault-tolerant cluster of high-speed multiprocessors. The Big Guns system is based on an enterprise-class transaction processing (TP) monitor noted for its efficient use of resources. Big Guns supports transparent distribution and replication of application components.

Purchased Application

Rather than building the entire application from scratch, Money Makers purchased from Portfolio Incorporated, an application software vendor, a stock fund account

management system that is implemented as a set of Enterprise JavaBeans components. Portfolio developed and tested the application using an EJB environment provided by a database vendor.

Customization and Enhancement

The Portfolio application provides the functionality to manage client accounts and organize portfolios. However, it does not use up-to-the-minute stock prices to calculate the current account value, nor does it support online trading.

Money Makers customized the stock component to make it retrieve the most recent stock price from a live data feed. The company used an EJB-compliant stock feed component from Garage Enterprises, a startup company that operates on a fairly tight budget. The component was developed and tested using an EJB-compliant application server from the public domain.

Money Makers elected to implement the online trading function by encapsulating the existing trading system and integrating it with the Portfolio system using CORBA.

Enterprise Java™ Platform

Enabling Portability

The portability characteristics of EJB components are made possible by the Enterprise Java platform. The Enterprise Java platform consists of several standard Java application programming interfaces (APIs) that provide access to a core set of enterprise-class infrastructure services. (See the accompanying table for a description of the Enterprise Java APIs.) The term “enterprise” implies highly scalable, highly available, highly reliable, highly secure, transactional, distributed applications. Enterprise applications require access to a variety of infrastructure services, such as distributed communication services, naming and directory services, transaction services, messaging services, data access and persistence services, and resource-sharing services. These infrastructure services are frequently implemented on different platforms using different products and technologies, making it difficult to build portable enterprise-class application systems. The Enterprise Java APIs provide a common interface to the underlying infrastructure services, regardless of the actual implementation.

Existing Infrastructure Integration

Sun could have defined a new set of infrastructure services especially to support the Enterprise Java platform. It’s been fairly standard practice in the past for standards organizations to define new services to support new environments. For example, X/Open defined the Distributed Transaction Processing (DTP) model and the XA standard, OSF defined the Distributed Computing Environment (DCE) services, and OMG defined the CORBA Object Services (COS, also known as CORBAservices). Microsoft has defined a set of integrated NT services to support DCOM. Meanwhile, every TP Monitor and application server vendor generally provides its own proprietary set of infrastructure services. Many of these infrastructure services perform similar but non-interoperable functions.

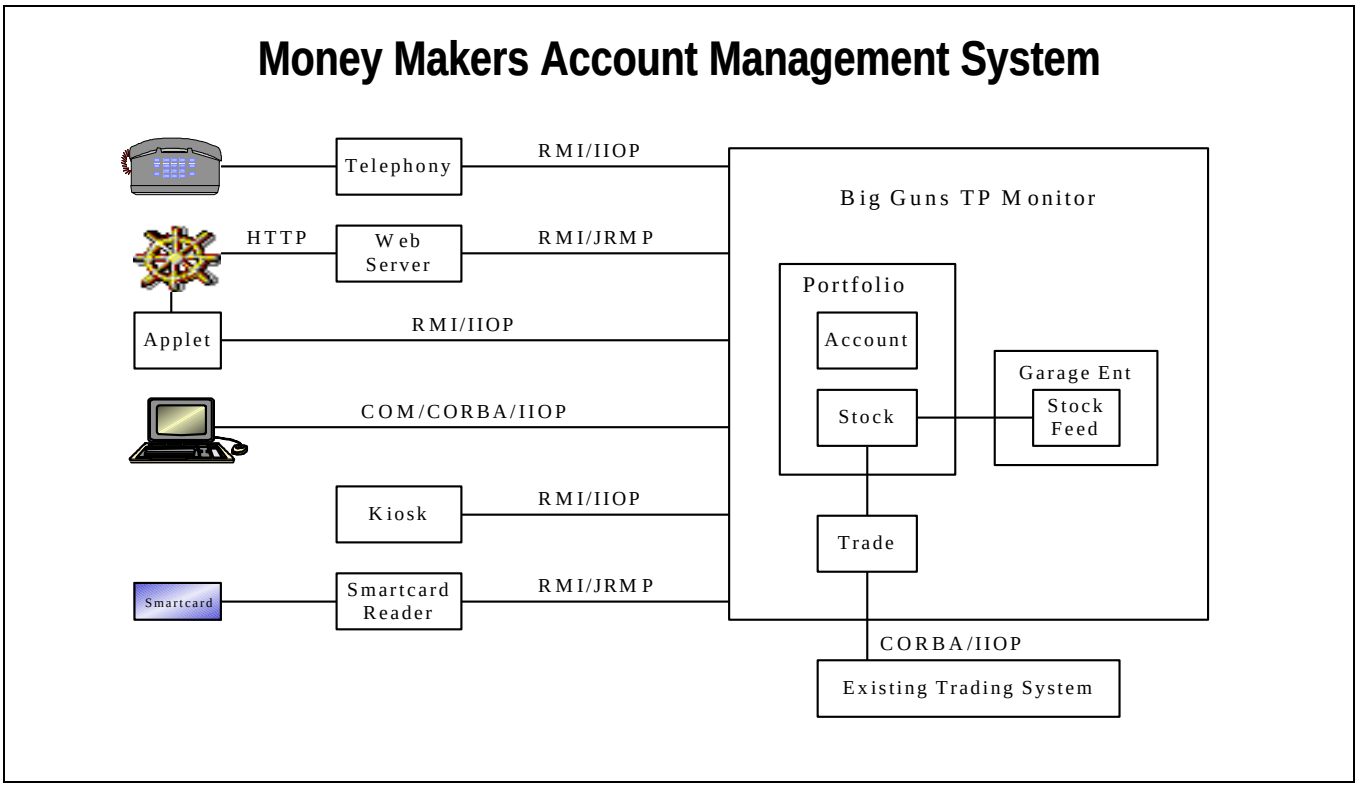

Illustration 1. The Money Makers account management system combines applications from two vendors and integrates with an existing application system. The application supports a variety of client devices through a variety of protocols.

Rather than re-inventing the wheel, Sun took a different route. The Enterprise Java platform defines a set of standard Java APIs that provide access to existing infrastructure services. Using these Java technology APIs, a developer can implement an application system that makes use of whatever enterprise services happen to exist on the platform in use. For example, an application that uses IBM TXSeries, DCE services, and Oracle on one system could be moved to a different system and automatically use BEA Systems M3, LDAP, and Sybase instead. No porting effort would be required.

Platform- and Vendor-Neutral

The Enterprise Java APIs are completely platform- and vendor-neutral. The APIs are designed to layer on multivendor heterogeneous infrastructure services. Each API provides a common programming interface to a generic type of infrastructure service. If each vendor that supplies this type of service implements this API, then an application could access any service provider through the common interface.

ODBC Metaphor

The Enterprise Java APIs take Microsoft's ODBC metaphor—one common interface to all relational databases—and apply it to all infrastructure services. For example, the Java Naming and Directory Interface (JNDI) provides a standard interface to naming and directory services. The JNDI API could be used to access any vendor's directory

service, such as Microsoft Active Directory, Novell NDS, Sun NIS+, or any LDAP directory.

Enterprise Java APIs

API	Description
EJB	The Enterprise JavaBeans API defines a server component model that provides portability across application servers and implements automatic services on behalf of the application components.
JNDI	The Java Naming and Directory Interface API provides access to naming and directory services, such as DNS, NDS, NIS+, LDAP, and COS Naming.
RMI	The Remote Method Invocation API creates remote interfaces for distributed computing on the Java platform.
Java IDL	The Java Interface Definition Language API creates remote interfaces to support CORBA communication in the Java platform. Java IDL includes an IDL compiler and a lightweight, replaceable ORB that supports IIOP.
Servlets and JSP	The Java Servlets and Java Server Pages APIs support dynamic HTML generation and session management for browser clients.
JMS	The Java Messaging Service API supports asynchronous communications through various messaging systems, such as reliable queuing and publish-and-subscribe services.
JTA	The Java Transaction API provides a transaction demarcation API.
JTS	The Java Transaction Service API defines a distributed transaction management service based on CORBA Object Transaction Service.
JDBC™	The JDBC Database Access API provides uniform access to relational databases, such as DB2, Informix, Oracle, SQL Server, and Sybase.

Table. The Enterprise Java APIs.

Industry Acceptance

In order to be successful, this approach requires widespread buy-in from the vendor community. The APIs will be useless if the infrastructure vendors don't implement support for the APIs in their existing infrastructure products. Sun was quite diligent to ensure that the Enterprise Java platform conforms to the most prevalent de jure and de facto standards. Sun also recruited expertise from the most elite players in each area of infrastructure services to ensure that the Enterprise Java APIs are suitable for high-volume, mission-critical application development. Industry leaders in transaction management (IBM, Compaq/Tandem, BEA Systems, etc.), persistence management (Oracle, Sybase, Informix, etc.), and directory services (HP, IBM, Novell, etc.) participated in the development of the Enterprise Java APIs. According

to early indications, the industry has embraced Enterprise Java technology with as much gusto as it embraces Java technology itself. A staggering number of vendors have announced their intent to support the Enterprise Java APIs, and many vendors have already delivered compatible products.

Component-Based Computing

Multitier Application Architecture

Scalability

The Enterprise Java platform has been designed to provide an environment that is suitable for the development and deployment of fully portable Java technology-based enterprise-class application systems. The true measure of an enterprise application system often comes down to system scalability and total throughput. How many users can concurrently use the system? How many objects can be instantiated in a given time frame? How many transactions can be processed per second?

Next-Generation Client/Server

Enterprise application systems support high scalability by using a multitier, distributed application architecture. A *multitier application* is an application that has been partitioned into multiple application components. Multitier applications provide a number of significant advantages over traditional client/server architectures, including improvements in scalability, performance, reliability, manageability, reusability, and flexibility.

Application Partitioning

In a traditional client/server application, the client application contains presentation logic (window and control manipulation), business logic (algorithms and business rules), and data manipulation logic (database connections and SQL queries)—a "fat client." The server is generally a relational database management system (which is actually not a part of the application.) In a multitier architecture, the client application contains only presentation logic—a "thin client." The business logic and data access logic are partitioned into separate components and deployed on one or more servers.

Increased Scalability and Performance

Moving the business and data manipulation logic to a server allows an application to take advantage of the power of multithreaded and multiprocessing systems. Server components can pool and share scarce resources, such as processes, threads, database connections, and network sessions. As system demands increase, highly active components can be replicated and distributed across multiple systems. Although modern client/server systems can easily support hundreds of concurrent users, their scalability has limits. Multitier systems can be built with essentially no scalability limits. If the design is efficient, more or bigger servers can be added to the environment to boost performance and to support additional users. Multitier systems can scale to support hundreds of thousands or millions of concurrent users.

Increased Reliability

A multitier environment can also support many levels of redundancy. Through replication and distribution, a multitier architecture eliminates any bottlenecks or single points of failure. The multitier approach supports high reliability and consistent system availability to support critical business operations.

Increased Manageability

A thin-client application is easier to manage than traditional client/server applications. Very little code is actually deployed on the client systems. Most of the application logic is deployed, managed, and maintained on the servers. Fixes, upgrades, new versions, and extensions can all be administered through a centralized management environment.

Increased Flexibility

The multitier application architecture supports extremely flexible application systems. The majority of the application logic is implemented in small modular components. The actual business logic in the components is encapsulated behind an abstract, well-defined interface. The code within an individual component can be modified without requiring a change to the interface. Therefore, a component can be changed without impacting the other components within the application. Multitier applications can easily adapt to reflect changing business requirements.

Reusability and Integration

By the nature of its interface, a server component is a reusable software building block. Each component performs a specific set of functions that are published and accessible to any other application through the interface. A particular business function can be implemented once and then reused in another application that requires the function. If an organization maintains a comprehensive library of components, application development becomes a matter of assembling the proper components into a configuration that performs the required application functions.

Multi-Client Support

Any number of client environments can access the same server component through its interface. A single multitier application system can support a variety of client devices, including traditional desktop workstations, Web clients, or more esoteric clients, such as information appliances, smartcards, or personal data assistants.

Multitier Impetus

Although server components and multitier concepts have been around for nearly a decade, relatively few organizations have put them to use. Until recently, most organizations did not feel the scalability pressures that required a multitier architecture. But the impetus of Web-based computing is driving a growing interest in the multitier approach. Web-based business applications require a thin-client application architecture to support massive scalability and to support browser-based clients and rapid applet downloads.

More Difficult Development

Unfortunately, building multitier applications isn't quite as easy as building client/server. Multitier applications have to interact with a variety of middleware services. In order to attain the scalability, performance, and reliability characteristics of multitier computing, the applications must support multithreading, resource sharing, replication, and load balancing.

Application Servers

An application server automates some of the more complex features of multitier computing. An application server manages and recycles scarce system resources, such as processes, threads, memory, database connections, and network sessions on behalf of the applications. Some of the more sophisticated application servers offer load-balancing services that can distribute application processing across multiple systems. An application server also provides access to infrastructure services, such as naming, directory, transactions, persistence, and security. Until recently, though, every application server used a proprietary set of interfaces. Each enterprise application had to be implemented with a specific runtime environment in mind, and applications were not portable across application servers—not even Java applications.

Enterprise JavaBeans Specification

The Enterprise JavaBeans specification defines a standard model for a Java application server that supports complete portability. Any vendor can use the model to implement support for Enterprise JavaBeans components. Systems, such as TP monitors, CORBA runtime systems, COM runtime systems, database systems, Web server systems, or other server-based runtime systems can be adapted to support portable Enterprise JavaBeans components.

Overview of Components

Components

A *component* is a reusable software building block: a pre-built piece of encapsulated application code that can be combined with other components and with handwritten code to rapidly produce a custom application.

Containers

Components execute within a construct called a *container*. A container provides an application context for one or more components and provides management and control services for the components. In practical terms, a container provides an operating system process or thread in which to execute the component. Client components normally execute within some type of visual container, such as a form, a compound document, or a Web page. Server components are non-visual and execute within a container that is provided by an application server, such as a TP monitor, a Web server, or a database system.

Component Model

A *component model* defines the basic architecture of a component, specifying the structure of its interfaces and the mechanisms by which it interacts with its container and with other components. The component model provides guidelines to create and implement components that can work together to form a larger application. Application builders can combine components from different developers or different vendors to construct an application.

Granularity

Components come in a variety of shapes and sizes. A component can be very small, such as a simple GUI widget (e.g., a button), or it can implement a complex application service, such as an account management function.

Standard Interface

In order to qualify as a component, the application code must provide a standard interface that enables other parts of the application to invoke its functions and to access and manipulate the data within the component. The structure of the interface is defined by the component model.

Customization without Source Code

An application developer should be able to make full use of the component without requiring access to its source code. Components can be customized to suit the specific requirements of an application through a set of external property values. For example, the button component has a property that specifies the name that should appear on the button. The account management component has a property that specifies the location of the account database. Properties can be used to support powerful customization services. For example, the account management component might allow a user to add a special approval process for withdrawals over a certain dollar amount. One property would be used to indicate that special approval functions are enabled, a second property would identify the conditions that require special approvals, and a third property would indicate the name of the approval process component that should be called when the condition exists.

Component Marketplace

One of the promises of component technology is a world in which customized business solutions can be assembled from a set of off-the-shelf business objects. Software vendors could produce numerous specialized business components, and organizations could select the appropriate components to match their business needs. Thus far, there is a fairly rich supply of off-the-shelf, third-party, client-side development components. For the moment, the market for server-side components is still very young. As more and more organizations adopt the server component architecture, the market is likely to mature rapidly. Application software companies are already beginning to implement applications using server components. Some e-commerce vendors are beginning to supply individual application functions, such as a shopping cart and a credit validation service, as customizable components.

Server Components

Shared Servers

In order to achieve the most benefit from the multitier architecture, server components should be implemented as shared servers. But building a shared server is harder than building a single-user application function. Highly scalable shared servers need to support concurrent users, and they need to efficiently share scarce system resources, such as threads, processes, memory, database connections, and network connections. For business operations, shared servers must participate in transactions. In many cases, a shared server needs to enforce security policies.

Plug-and-Play Assembly

A component builder doesn't especially want to implement multithreading, concurrency control, resource-pooling, security, and transaction management in every component. If these services were implemented in each component, achieving true plug-and-play application assembly would be very difficult. A component model

standardizes and automates the use of these services, thereby enabling easy application development.

Container

An application server provides a *container* to manage the execution of a component. When a client invokes a server component, the container automatically allocates a process thread and initiates the component. The container manages all resources on behalf of the component and manages all interactions between the component and the external systems.

Example Application Servers

There are many different types of application servers in common use today, and each provides a container for some type of server-based request. For example:

- A TP monitor contains transactions and manages shared resources on behalf of a transaction. Multiple transactions can work together and rely on the TP monitor to coordinate the extended transaction.
- A database management system (DBMS) contains database requests. Multiple database clients can submit requests to the database concurrently and rely on the DBMS to coordinate locks and transactions.
- A Web server contains Web page requests. Multiple Web clients can submit concurrent page requests to the Web server. The Web server serves up HTML pages or invokes server extensions or servlets in response to requests.

Portability across Containers

The operations and behaviors of a container are defined by its component model. Unfortunately, each container implements its own set of services with its own service interfaces. As a result, components developed for one type of environment are usually not portable to any other type of environment. The Enterprise JavaBeans component model, however, is designed to deliver a portability layer for these container systems.

Enterprise JavaBeans Component Model

Overview of Enterprise JavaBeans Technology

JavaBeans Development Components

The Enterprise JavaBeans component model logically extends the JavaBeans component model. The JavaBeans component model defines a standard mechanism to develop portable, reusable Java technology *development* components, such as widgets or controls. JavaBeans technology can be used in any visual Java technology integrated development environment (IDE), such as IBM Visual Age, Inprise JBuilder, Sybase PowerJ, and Symantec Visual Café. Java developers use a visual Java IDE to build Java classes, Java applets, Java applications, or Java technology components. A JavaBeans component (a bean) is a specialized Java class that can be added to an application development project and then manipulated by the Java IDE. A bean provides special hooks that allow a visual Java development tool to examine and customize the contents and behavior of the bean without requiring access to the

source code. Multiple beans can be combined and interrelated to build Java applets or applications or to create new, more comprehensive, or specialized JavaBeans components.

Enterprise JavaBeans Component Model

The Enterprise JavaBeans component model logically extends the JavaBeans component model to support *server components*. Server components are reusable, prepackaged pieces of application functionality that are designed to run in an application server. They can be combined with other components to create customized application systems. Server components are similar to development components, but they are generally larger grained and more complete than development components. Enterprise JavaBeans components (enterprise beans) cannot be manipulated by a visual Java IDE in the same way that JavaBeans components can. Instead, they can be assembled and customized at deployment time using tools provided by an EJB-compliant Java application server.

Simplifying Development

The Enterprise JavaBeans architecture provides an integrated application framework that dramatically simplifies the process of developing enterprise-class application systems. An EJB server automatically manages a number of tricky middleware services on behalf of the application components. EJB component-builders can concentrate on writing business logic rather than complex middleware. The results are that applications get developed more quickly and the code is of better quality.

Implicit Services

The EJB model supports a number of implicit services, including lifecycle, state management, security, transactions, and persistence.

- Lifecycle. Individual enterprise beans do not need to explicitly manage process allocation, thread management, object activation, or object destruction. The EJB container automatically manages the object lifecycle on behalf of the enterprise bean.
- State Management. Individual enterprise beans do not need to explicitly save or restore conversational object state between method calls. The EJB container automatically manages object state on behalf of the enterprise bean.
- Security. Individual enterprise beans do not need to explicitly authenticate users or check authorization levels. The EJB container automatically performs all security checking on behalf of the enterprise bean.
- Transactions. Individual enterprise beans do not need to explicitly specify transaction demarcation code to participate in distributed transactions. The EJB container can automatically manage the start, enrollment, commitment, and rollback of transactions on behalf of the enterprise bean.
- Persistence. Individual enterprise beans do not need to explicitly retrieve or store persistent object data from a database. The EJB container can automatically manage persistent data on behalf of the enterprise bean.

Portability Layer

The Enterprise JavaBeans model defines the interrelationship between an enterprise bean component and an enterprise bean container. Enterprise JavaBeans components do not require the use of any specific container system. A vendor can adapt any application server to support Enterprise JavaBeans technology by adding support for the services defined in the specification. The services define a contract between an enterprise bean and the container, effectively implementing a portability layer. Any enterprise bean can run in any application server that supports the Enterprise JavaBeans contracts.

Execution Services

An Enterprise JavaBeans-compliant application server, called an EJB server, must provide a standard set of services to support enterprise bean components. Enterprise JavaBeans components are transactional; therefore, an EJB server must provide access to a distributed transaction management service. The EJB server must also provide a container for the enterprise beans, which is called an EJB container. The EJB container implements the management and control services for one or more classes of Enterprise JavaBean objects. The EJB container also provides lifecycle management, implicit transaction control, persistence management, transparent distribution services, and security services on behalf of the enterprise bean. In most circumstances, a single vendor would provide both an EJB server and an associated EJB container, although the specification allows the separation of these services. For example, a third-party vendor may provide an add-on container that implements persistence through object/relational mapping.

Potential Enterprise JavaBeans Systems

The exact natures of process management, thread-pooling, concurrency control, and resource management are not defined within the scope of the Enterprise JavaBeans specification. Individual vendors can differentiate their products based on the simplicity or sophistication of the services. A software vendor might elect to develop a new application server specifically to support Enterprise JavaBeans components. It is more likely, however, that vendors will simply adapt their existing systems. A number of application servers are currently available, and any of these systems could be extended to support a container for Enterprise JavaBeans components. An impressive number of vendors are extending a wide variety of products, including:

- TP monitors, such as IBM TXSeries and IBM CICS/390
- Component transaction servers, such as Sybase Jaguar CTS
- CORBA systems, such as BEA Systems M3, IBM WebSphere Advanced Edition, and Inprise VisiBroker/ITS
- Relational database systems, such as IBM DB2, Informix, Oracle, and Sybase
- Object database systems, such as GemStone GemStone/J
- Object/relational caching systems, such as Persistence PowerTier and Secant Extreme

- Web application servers, such as BEA WebLogic, Bluestone Sapphire, IBM WebSphere, Netscape Application Server, Oracle Application Server, Progress Apptivity, SilverStream Application Server, and Sun NetDynamics.

Versatility

The Enterprise JavaBeans model enables a much higher level of integration and interoperability than ever existed before. Enterprise JavaBeans applications can be developed using any Enterprise JavaBeans technology-compliant environment, and users can deploy the applications in any other environment. As requirements for higher performance, increased scalability, or tighter security arise, users can move the applications to an environment with more comprehensive and sophisticated services.

Architectural Details

The Big Picture

Enterprise JavaBeans Server

The EJB server provides an environment that supports the execution of applications developed using Enterprise JavaBeans technology. It manages and coordinates the allocation of resources to the applications.

EJB Container

Illustration 2 shows a representation of an EJB container. The EJB server must provide one or more EJB containers, which provide homes for the enterprise beans. An EJB container manages the enterprise beans contained within it. For each enterprise bean, the container is responsible for registering the object, providing a remote interface for the object, creating and destroying object instances, checking security for the object, managing the active state for the object, and coordinating distributed transactions. Optionally, the container can also manage all persistent data within the object.

Installing Enterprise Beans in an EJB Container

Any number of EJB classes can be installed in a single EJB container. A particular class of enterprise bean is assigned to one and only one EJB container, but a container may not necessarily represent a physical location. The physical manifestation of an EJB container is not defined in the Enterprise JavaBeans specification. An EJB container could be implemented as a physical entity, such as a multithreaded process within an EJB server. It also could be implemented as a logical entity that can be replicated and distributed across any number of systems and processes.

Transient and Persistent Objects

Enterprise JavaBeans technology supports both transient and persistent objects. A transient object is called a *session* bean, and a persistent object is called an *entity* bean.

SESSION BEANS. A session bean is created by a client and in most cases exists only for the duration of a single client/server session. A session bean performs operations on behalf of the client, such as accessing a database or performing calculations. Session beans can be transactional, but (normally) they are not recoverable following a system crash. Session beans can be stateless, or they can maintain conversational state across methods and transactions. The container manages the conversational state

of a session bean if it needs to be evicted from memory. A session bean must manage its own persistent data.

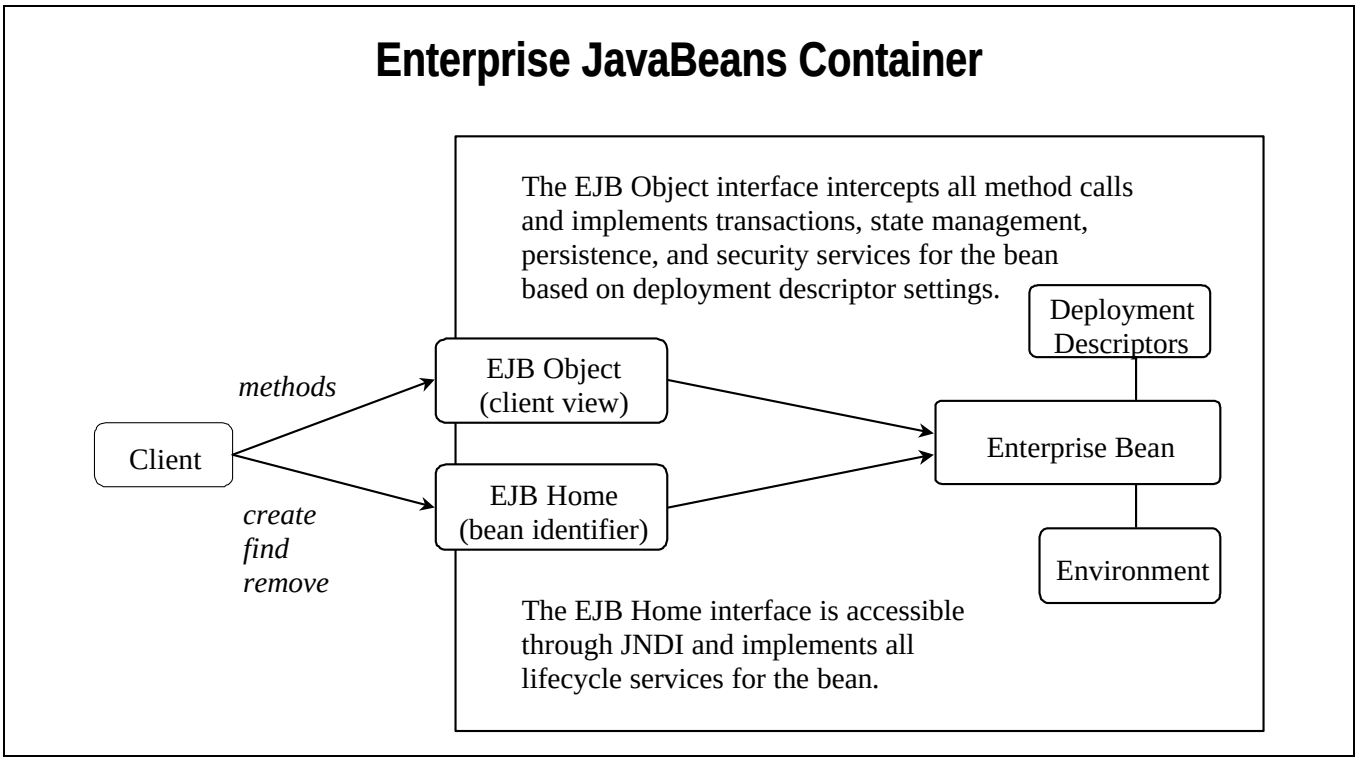

Illustration 2. Enterprise beans are deployed in an EJB container within an EJB server. The EJB container acts as a liaison between the client and the enterprise bean. At deployment time, the container automatically generates an EJB Home interface to represent the enterprise bean class and an EJB Object interface for each enterprise bean instance. The EJB Home interface identifies the enterprise bean class and is used to create, find, and remove enterprise bean instances. The EJB Object interface provides access to the business methods within the bean. All client requests directed at the EJB Home or EJB Object interfaces are intercepted by the EJB container to insert lifecycle, transaction, state, security, and persistence rules on all operations.

ENTITY BEANS. An entity bean is an object representation of persistent data that are maintained in a permanent data store, such as a database. A primary key identifies each instance of an entity bean. Entity beans can be created either by inserting data directly into the database or by creating an object (using an object factory Create method). Entity beans are transactional, and they are recoverable following a system crash.

Container-Managed Persistence

Entity beans can manage their own persistence, or they can delegate persistence services to their container. If the bean delegates persistence to the container, then the container automatically performs all data retrieval and storage operations on behalf of the bean.

Entity Objects Are Optional

In Release 1.0 of the Enterprise JavaBeans specification, support for session beans is required, but support for entity beans and container-managed persistence is optional. Mandatory support for these features will be required in a future version of the specification.

Standard Contracts

An enterprise bean can be deployed in any EJB server, even though different servers implement their services in different ways. The EJB model ensures portability across different EJB servers using a set of standard contracts between the EJB container and the enterprise bean. Each enterprise bean is required to implement a specific set of interfaces that allows the EJB container to manage and control the object. The EJB container is required to invoke these interfaces at particular stages of execution.

Wrapping and Interception

An EJB container manages the enterprise beans that are deployed within it. Client applications do not directly interact with an enterprise bean. Instead, the client application interacts with the enterprise bean through two wrapper interfaces that are generated by the container: the EJB Home interface and the EJB Object interface. As the client invokes operations using the wrapper interfaces, the container intercepts each method call and inserts the management services.

EJB Home

The EJB Home interface provides access to the bean's lifecycle services. Clients can use the Home interface to create or destroy bean instances. For entity beans, the Home interface also provides one or more finder methods that allow a client to find an existing bean instance and retrieve it from its persistent data store.

Naming and Registration

For each class installed in a container, the container automatically registers the EJB Home interface in a directory using the Java Naming and Directory Interface (JNDI) API. Using JNDI, any client can locate the EJB Home interface to create a new bean instance or to find an existing entity bean instance. When a client creates or finds a bean, the container returns an EJB Object interface.

EJB Object

The EJB Object interface provides access to the business methods within the enterprise bean. An EJB Object represents a client view of the enterprise bean. The EJB Object exposes all of the application-related interfaces for the object, but not the interfaces that allow the EJB container to manage and control the object. The EJB Object wrapper allows the EJB container to intercept all operations made on the enterprise bean. Each time a client invokes a method on the EJB Object, the request goes through the EJB container before being delegated to the enterprise bean. The EJB container implements state management, transaction control, security, and persistence services transparently to both the client and the enterprise bean.

Declarative Attributes

The rules associated with the enterprise bean governing lifecycle, transactions, security, and persistence are defined in an associated Deployment Descriptor object. These rules are defined declaratively at deployment time rather than programmatically at development time. At runtime, the EJB container automatically

performs the services according to the values specified in the deployment descriptor object associated with the enterprise bean.

Context Object

For each active enterprise bean instance, the EJB container generates an instance context object to maintain information about the management rules and the current state of the instance. A session bean uses a SessionContext object, and an entity bean uses an EntityContext object. The context object is used by both the enterprise bean and the EJB container to coordinate transactions, security, persistence, and other system services. Also associated with each enterprise bean is a properties table called the Environment object. The Environment object contains the customized property values set during the application assembly process or the enterprise bean deployment process.

Distribution Services

Remote Method Invocation

Enterprise JavaBeans technology uses the Java Remote Method Invocation API to provide access to enterprise beans. An enterprise bean developer must define an RMI Remote interface for each enterprise bean. The container generates the EJB Object interface from the Remote interface definitions.

Location Transparency

RMI is a high-level programming interface that makes the location of the server transparent to the client. The RMI compiler generates a stub object for each remote interface. The stub object is installed on the client system (or can be downloaded at runtime) and provides a local proxy object for the client. The stub implements all the remote interfaces and transparently delegates all method calls across the network to the remote object.

Protocols

The Enterprise JavaBeans specification asserts no requirements for a specific distributed object protocol. RMI can support multiple communications protocols. The Java Remote Method Protocol is the RMI native protocol. It supports all functions within RMI. The next release of RMI will add support for communications using the CORBA standard communications protocol, Internet InterORB Protocol. IIOP supports almost all functions within RMI. Enterprise beans that rely only on the RMI/IIOP subset of RMI are portable across both protocols. Third-party implementations of RMI support other protocols, such as Secure Sockets Layer (SSL).

Native Language Integration

By using IIOP, enterprise beans can interoperate with native language clients and servers. IIOP allows easy integration between CORBA systems and EJB systems. Enterprise beans can access CORBA servers, and CORBA clients can access enterprise beans. Using a COM/CORBA Internetworking service, ActiveX clients can also access enterprise beans and enterprise beans can access COM servers. Potentially, there could also be a DCOM implementation of Enterprise JavaBeans technology.

State Management

Resource Optimization

Individual EJB server systems can implement different policies to manage scarce resources, such as memory and threads. Some EJB servers might maintain all active objects in memory, while others might evict every object after every method call. Some EJB servers might use a least-recently-used (LRU) policy to evict objects when resources get tight. Regardless of the policies used, each EJB server must provide state management for objects.

Stateful Session Beans

A session bean represents work being performed by an individual client. In some cases, that work can be performed entirely within a single method call. In other cases, the work may span multiple method requests. If the work spans multiple methods, the object must maintain the user's object state between method calls. For example, in an e-commerce application, the CheckCreditAuthorization method performs an entire unit of work and requires no maintained state. The ShoppingCart object, on the other hand, must keep track of all the items selected while the customer is browsing until the customer is ready to buy the items. The state management options for a session bean are defined in the StateManagementType attribute in the Deployment Descriptor object. All entity beans are inherently stateful.

Automatic State Management

If an object is stateless, the container automatically resets the state within an object instance after each method call. If the object is stateful, the container automatically maintains the object conversational state until the session object is destroyed, even if the object is temporarily evicted from memory. The Enterprise JavaBeans architecture provides a simple programming model to allow developers to perform specific functions whenever objects are loaded or evicted from memory. The Enterprise JavaBeans model isolates these functions in the ejbLoad, ejbStore, ejbActivate, and ejbPassivate methods in each enterprise bean class.

Stateless vs. Stateful Servers

Stateless servers use fewer resources and can be more easily recycled than stateful servers. Since each instance of a stateless object class is identical, the instances can be pooled and reused repeatedly. Therefore, many transaction processing experts claim that stateless servers scale better and are more appropriate for high-volume transaction systems. But if application requirements dictate an extended, multi-method conversation, it may be more efficient to use stateful servers.

Persistence Management

Persistent Objects

An entity bean represents persistent data. An entity object generally exists for an extended period of time, and it can be used by many clients.

Persistence Programming Model

Enterprise JavaBeans technology provides a simple programming model for managing object persistence. Persistence functions must be performed whenever objects are created or destroyed or whenever objects are loaded or evicted from memory. The Enterprise JavaBeans model isolates these functions in the ejbCreate, ejbPostCreate,

ejbRemove, ejbLoad, ejbStore, ejbActivate, and ejbPassivate methods in each enterprise bean class.

Entity Object Persistence

An entity object can manage its own persistence, or it can delegate its persistence to its container. The persistence options for an entity bean are defined in the ContainerManagedFields attribute in the deployment descriptor object.

Bean-Managed Persistence

If the entity object manages its own persistence, then the enterprise bean developer must implement persistence operations (e.g., JDBC or embedded SQL calls) directly in the enterprise bean class methods.

Container-Managed Persistence

If the entity object delegates persistence services, the EJB container transparently and implicitly manages the persistent state. The enterprise bean developer does not need to code any database access functions within the enterprise bean class methods. The first release of the Enterprise JavaBeans specification does not define how the EJB container must manage object persistence. A vendor may implement a basic persistence service in the EJB container that simply serializes the enterprise bean's state and stores it in some persistent storage. Alternatively, a vendor may implement a more sophisticated persistence service that, for example, transparently maps the object's persistent fields to columns in an underlying relational database. A vendor may also implement persistence using an embedded object database.

Session Object Persistence

Session objects, by definition, are not persistent, although they may contain information that needs to be persisted. As with bean-managed entity objects, session objects can implement persistence operations directly in the methods in the enterprise bean. Session objects also often maintain a cache of database information that must be synchronized with the database when transactions are started, committed, or aborted. An enterprise bean developer can implement transaction synchronization methods directly in the enterprise bean class using the optional SessionSynchronization interface. The afterBegin, beforeCompletion, and afterCompletion notifications signal transaction demarcation points, allowing the object to read or write data to the database as needed.

Transaction Management

Distributed Transactions

Although Enterprise JavaBeans technology can certainly be used to implement nontransactional systems, the model was designed to support distributed transactions. Enterprise JavaBeans technology requires the use of a distributed transaction management system that supports two-phase commit protocols for flat transactions.

JTS

The Enterprise JavaBeans specification suggests but does not require transactions based on the Java Transaction Service (JTS) API. JTS is the Java technology binding of the CORBA Object Transaction Service (OTS). JTS supports distributed transactions that can span multiple databases on multiple systems coordinated by

multiple transaction managers. By using JTS, an Enterprise JavaBeans Server ensures that its transactions can span multiple Enterprise JavaBeans servers.

JTA

Enterprise JavaBeans applications communicate with a transaction service using the Java Transaction API (JTA). JTA provides a programming interface to start transactions, join existing transactions, commit transactions, and roll-back transactions.

Simplicity

Although transaction demarcation in a centralized application is fairly straightforward, it gets a lot trickier when an application consists of a variable number of autonomous application components that call back and forth to each other. Enterprise JavaBeans technology dramatically simplifies application development by automating the use of distributed transactions. All transaction functions can be performed implicitly by the EJB container and the EJB server. Individual enterprise beans need not make any transaction demarcation statements within their code. Since no transaction code is required within the application logic, the enterprise beans are simpler to write and are portable across different transaction managers.

Declarative Transaction Rules

The transaction semantics for an enterprise bean are defined declaratively rather than programmatically. At runtime, the EJB container automatically implements transaction services according to the TransactionAttribute attribute specified in the deployment descriptor object.

Transaction Attributes

The Enterprise JavaBeans model supports six different transaction rules:

- **TX_BEAN_MANAGED**. The TX_BEAN_MANAGED setting indicates that the enterprise bean manually manages its own transaction control. EJB supports manual transaction demarcation using the Java Transaction API.

- TX_NOT_SUPPORTED. The TX_NOT_SUPPORTED setting indicates that the enterprise bean cannot execute within the context of a transaction. If a client (i.e., whatever called the method—either a remote client or another enterprise bean) has a transaction when it calls the enterprise bean, the container suspends the transaction for the duration of the method call.

- TX_SUPPORTS. The TX_SUPPORTS setting indicates that the enterprise bean can run with or without a transaction context. If a client has a transaction when it calls the enterprise bean, the method will join the client's transaction context. If the client does not have a transaction, the method will run without a transaction.

- TX_REQUIRED. The TX_REQUIRED setting indicates that the enterprise bean must execute within the context of a transaction. If a client has a transaction when it calls the enterprise bean, the method will join the client's transaction context. If the client does not have a transaction, the container automatically starts a new transaction for the method.

- TX_REQUIRES_NEW. The TX_REQUIRES_NEW setting indicates that the enterprise bean must execute within the context of a new transaction. The container always starts a new transaction for the method. If the client has a transaction when it calls the enterprise bean, the container suspends the client's transaction for the duration of the method call.
- TX_MANDATORY. The TX_MANDATORY setting indicates that the enterprise bean must always execute within the context of the client's transaction. If the client does not have a transaction when it calls the enterprise bean, the container throws the TransactionRequired exception and the request fails.

Security

Java Security

The Enterprise JavaBeans model utilizes the Java security services supported in Java Development Kit (JDK™) 1.1.x. Java platform security supports authentication and authorization services to restrict access to secure objects and methods.

Enterprise JavaBeans Security

Enterprise JavaBeans technology automates the use of Java platform security so that enterprise beans do not need to explicitly code Java security routines. The security rules for each enterprise bean are defined declaratively in a set of AccessControlEntry objects within the deployment descriptor object. An AccessControlEntry object associates a method with a list of users that have rights to invoke the method. The EJB container uses the AccessControlEntry to automatically perform all security checking on behalf of the enterprise bean.

Enterprise JavaBeans Deployment

Packaging

Enterprise JavaBeans components can be packaged as individual enterprise beans, as a collection of enterprise beans, or as a complete application system. Enterprise JavaBeans components are distributed in a Java Archive File called an ejb-jar file. The ejb-jar file contains a manifest file outlining the contents of the file, plus the enterprise bean class files, the Deployment Descriptor objects, and, optionally, the Environment Properties objects.

Deployment Descriptors

The Deployment Descriptor objects are used to establish the runtime service settings for an enterprise bean. These settings tell the EJB container how to manage and control the enterprise bean. The settings can be set at application assembly or application deployment time.

The DeploymentDescriptor object specifies how to create and maintain an Enterprise Bean object. This object defines, among other things, the enterprise bean class name, the JNDI namespace that represents the container, the Home interface name, the Remote interface name, and the Environment Properties object name. The DeploymentDescriptor object contains an array of ControlDescriptor objects, which specify the transaction semantics that should be applied to the enterprise bean, and an

array of AccessControlEntry objects, which specify the security rules that should be applied to the enterprise bean.

Session beans and entity beans have slightly different requirements; therefore, there are two different types of deployment descriptors.

- The SessionDescriptor object extends the DeploymentDescriptor object and adds attributes to indicate whether or not a session bean is stateless or stateful.
- The EntityDescriptor object extends the DeploymentDescriptor object and adds attributes to indicate which fields within the object should be persisted automatically by the container.

Environment Properties

An enterprise bean developer can provide an Environment Properties object to allow the application developer to customize the bean to suit the needs of the application. For example, a property might be used to specify the location of a database or to specify a default language.

Industry Support

Tremendous Buy-In

The industry has shown tremendous support for the Enterprise JavaBeans technology initiative. Most major vendors—including IBM, Oracle, Sybase, Netscape, and BEA Systems—have participated in the definition of the Enterprise JavaBeans specification. These vendors are in the process of implementing support for Enterprise JavaBeans technology in their products.

EJB Servers

The first EJB-compliant application servers began to appear in August 1998. Products that are shipping or in beta as of this writing include:

- BEA WebLogic Tengah
- Bluestone Sapphire/Web
- GemStone GemStone/J
- IBM WebSphere Advanced Edition
- Novera jBusiness
- Oracle8i
- Oracle Application Server
- OrchidSoft Vanda
- Persistence PowerTier
- Progress Apptivity
- Secant Extreme
- Valto Ejipt

Additional EJB-compliant application servers should be available in early 1999 from vendors, such as Forte, Fujitsu, Haht, Inprise, Informix, Netscape, Sun, Sybase, and Vision.

Application Vendors

Application software vendors have also expressed their support for Enterprise JavaBeans technology. The model increases the versatility of packaged application solutions. Applications implemented using Enterprise JavaBeans components can be deployed on a much broader array of systems. In addition, they support easier customization and integration with existing application systems. EJB-compliant applications and components are available or in development from the following companies:

- Athena Design Integer (a collaborative spreadsheet)
- Digital Harbor personal productivity application components
- EC-Cubed electronic commerce components
- IBI EDA components (to provide integration with host data and applications)
- IBM San Francisco Frameworks (General Ledger, Order Management, etc.)
- NovaSoft Novation (an Electronic Document Management System)
- Oracle Applications (ERP)
- Seven Mountains personal productivity application components
- TradeEx procurement components

EJB Competition

Microsoft Hold-Out

The primary hold-out against Enterprise JavaBeans technology is Microsoft. Although Microsoft Transaction Server (MTS) could be adapted to support Enterprise JavaBeans components, Microsoft is not likely to make the effort. The Java portability message would undermine Microsoft's message of tight integration based on the NT platform. Microsoft is pushing developers to build component-based application systems based on the COM component model. MTS provides a container system for COM server components, providing transactional and security services similar to those provided in Enterprise JavaBeans servers. COM+, the next generation of MTS, will provide a few additional capabilities, such as dynamic load-balancing and queued request-processing.

MTS Limitations

Although COM components provide many of the same benefits as Enterprise JavaBeans components, there are some significant differences.

- Platform Limitations. COM components rely on a COM runtime system and the DCOM communication protocol. Although COM and DCOM have recently been ported to Solaris and other Unix platforms, COM container systems are not available on these platforms. COM server components can realistically be deployed only on Windows 95 or NT. Enterprise JavaBeans components are platform independent.

- Vendor Limitations. Very few vendors provide container systems for COM and DCOM components. At the moment, COM containers are available from only two vendors: Microsoft and Sybase. The COM model does not extend far enough to support interoperability and transparent portability across these two environments. Any number of container systems can be adapted to support

Enterprise JavaBeans components, and many vendors have announced their intentions to support the model, including BEA Systems, Fujitsu, IBM, Oracle, Sybase, and Netscape. The Enterprise JavaBeans model ensures interoperability and portability across all of these environments.

- Client Limitations. COM and DCOM services are generally not available on non-Windows clients, such as network computers, information appliances, or smartcards. Enterprise JavaBeans components can support any Internet-enabled client device.

- Limited State and Persistence Management. MTS does not support a distinction between transient and persistent objects. All MTS components are the equivalent of stateless session objects. MTS automatically flushes all object state at the end of every transaction, and developers are responsible for managing all conversational state. MTS does not support automatic persistent objects. Developers are responsible for managing all persistent data. Enterprise JavaBeans technology supports stateless and stateful transient objects and persistent objects.

- More Restrictive Transaction Management. Although MTS supports declarative transaction management, COM components must still implement some transaction demarcation code within the application logic. Even so, MTS applications don't have as much flexibility to override the automatic MTS transaction services. Enterprise JavaBeans technology requires no transaction demarcation code within the application logic, but, if a developer chooses, the enterprise bean can completely control its transaction behavior.

Benefits and Conclusions

Component Portability

The Enterprise JavaBeans architecture provides a simple and elegant server component container model. The model ensures that Java platform server components can be developed once and deployed anywhere, in any vendor's container system. Even though the container systems implement their runtime services differently, the Enterprise JavaBeans interfaces ensure that an enterprise bean can rely on the underlying system to provide consistent lifecycle, persistence, transaction, distribution, and security services.

Architecture Independence

The Enterprise JavaBeans architecture is completely independent from any specific platform, protocol, or middleware infrastructure. Applications that are developed for one platform can be picked up, moved, and redeployed to another platform. EJB applications can scale from a small single-processor, Intel-based Novell environment to a large multiprocessor, UltraSPARC™ environment to a massive Sysplex IBM mainframe environment—all without modification.

Developer Productivity

The Enterprise JavaBeans architecture improves the productivity of application developers. The Enterprise JavaBeans environment automates the use of complex

infrastructure services, such as transactions, thread management, and security checking. Component developers and application builders do not need to implement complex service functions within the application programming logic.

Highly Customizable

Enterprise JavaBeans applications are highly customizable. The underlying component model supports customization without requiring access to source code. Application behaviors and runtime settings are defined through a set of attributes that can be changed at deployment time.

Wrap and Embrace

The Enterprise JavaBeans architecture is an extremely compatible evolutionary environment. The Enterprise Java services layer over existing infrastructure services. Organizations are not required to implement yet another incompatible set of middleware technologies. Enterprise JavaBeans technology enhances, enables, and simplifies popular systems, such as CORBA or DCOM.

Versatility and Scalability

The Enterprise JavaBeans model is based on an extremely versatile and powerful multitier, distributed object architecture that relies on industry-standard protocols. The model is appropriate for small-scale applications or large-scale business transactions. As application requirements grow, applications can migrate to progressively more powerful operating environments. The environment inherently supports Web-based applications and a variety of other Internet-enabled client devices. Additional client systems can be added at any time without modification of the core application systems. Enterprise JavaBeans technology provides an environment that is designed to grow with the industry to support new technologies as they emerge.