Creating Commercial Components (J2EE 1.3)

Enterprise Java Technology
Based Components

•

Technical White Paper

View Contents

Date: June 5, 2000
Revised: September 16, 2002

Authors: Faiz Arni, InferData Corporation
Andrew Pharoah, ComponentSource

www.componentsource.com

Email:

publishers@componentsource.com

farni@inferdata.com

US Headquarters	European Headquarters	US Headquarters
ComponentSource 3391 Town Point Drive, Suite 350, Kennesaw, GA 30144-7083 USA	**ComponentSource** 30 Greyfriars Road, Reading, Berkshire RG1 1PE United Kingdom	**InferData** 8200 N. MoPac Expy Ste. 250 Austin, TX 78759 USA
Tel: (770) 250 6100 **Fax:** (770) 250 6199 **International:** +1 (770) 250 6100	**Tel:** 0118 958 1111 **Fax:** 0118 958 1111 **International**: +44 118 958 1111	**Tel:** (888) 211-3421 **Fax:** (512) 306-8781 **International**: +1 (512) 306-8225

Contents

Introduction

This white paper has been constructed to help component authors develop and enhance off-the-shelf software components for server-side deployment and successful delivery on the 'open market'. Additionally, applying the principles of commercial-grade component development can greatly assist internal reuse, allowing components to be leveraged in multiple projects. Developers then have the option of furthering their development investment and releasing components into the open market, where we see a growing, pent-up demand. Information covered in the document is based on our knowledge and expertise of those component authors who successfully have established themselves in the component marketplace. The content is aimed at developers who wish to create components based on Sun Microsystems® Java 2, Enterprise Edition (J2EE) platform specification. In the following chapter we discuss the business benefits of using components and identify the functionality suitable for server-side component development in Java technology. Following this we detail the J2EE architecture, the EJB architecture, the Java Servlet and the Java ServerPages (JSP) technologies, the Java Messaging System (JMS) API and the environment in which these components and technologies can be used.

Commercial Overview

Software components are playing a major role in the eBusiness platform. Worldwide revenue in this market will increase at a compound annual growth rate (CAGR) of 40%, from $516 million in 1999 to $2.7 billion in 2004. (Source: IDC, The Software Construction Components Market, 6/2000).

Traditionally, server applications were built using proprietary transaction processing monitor (TP Monitor) systems. This made it difficult to write portable, enterprise-class software. With the introduction of Enterprise JavaBeans Components, server-side, enterprise software applications may now be created as a collection of software components or enterprise beans. These EJB based applications may now be deployed on any EJB-compliant application servers such as Sun-Netscape Alliance iPlanet , IBM® WebSphere , BEA Weblogic® or IONA iPortal Application Server . Increasingly enterprise application developers are employing component-based software development techniques, which enable them to reduce their time to market and improve their software quality. Software authors who are experts in a specific horizontal or vertical market sector are now "componentizing" their applications to meet the increasing demand for sophisticated business components. As such this represents a huge opportunity for you to unlock hidden revenues from years of research and development.

Why is buying a software component a good idea?

Everybody, software developers included, admit that they do something, (write a program or subroutine), better second time around. This is the essence of a "component", built and continuously improved by an expert or organization and encapsulating business logic or technical functionality. By buying a component a developer can add functionality to their application without sacrificing quality. Indeed quality should improve, as the component will have gone through several development iterations and include feedback from 1,000's of users.

What type of components will people buy?

Initially software components were used to provide technical functionality, such as SMTP for email or enhanced user interfaces. Developers are now requesting sophisticated components that solve real business issues from

component authors, such as *Credit Card Authenticating* components for E-Business applications. To find out what is in demand visit our Component Request Center: www.componentsource.com/business or look at the Case Studies of Authors who have already entered the 'open market' for components.

What is helping make this happen now?

The open Enterprise JavaBeans specification from Sun Microsystems is a component model for building server-side, enterprise class applications. EJB allows the component authors to focus on creating portable and reusable components rather than spending time on building complex proprietary "application framework environments" that "lock-in" users. The EJB specification requires the application servers to provide a host of services that the EJB-based components may depend upon. Since the services are specified using Java technology *interfaces*, the bean implementation is not tied to any application server vendor's implementation of those services. The EJB specification also enables the application server vendors to provide a robust, scalable, secure and transactional environment to host the EJBs. EJBs are implemented using the Java Programming Language as specified by Sun Microsystems. The Java programming language is a rich, portable, and secure language that supports automatic garbage collection, reflection and dynamic method dispatch.

To find out about the J2EE architecture and how to design, implement and deploy J2EE components - read the remainder of this white paper.

Component Overview

Identifying A Component Candidate

How do I identify a component candidate? - Understanding how a component works and how functionality differs from applications is important when identifying a suitable component candidate. In this section we investigate existing applications for potential functionality, consider component reusability and finally discuss the importance of business knowledge and how this applies to the components you write.

a) Analyze Application Functionality

Developers should look at the functions encapsulated in their own applications and others to assess the commercial viability of componentizing particular functions. Each component advertises one or more business interfaces. The users (or clients) of the component interact with it only through these interfaces. The clients are completely decoupled from the implementation of the component. The component implementation may be changed or upgraded without affecting the clients. One of the main characteristics of a component is that the business logic is separate from the data that a component manipulates. However, this does not apply to single parameter data that is passed to methods of the interface. For example, in EJB, a special kind of beans, known as *Entity* beans, are used to model persistent data. However the underlying data may come from any JDBC compliant data elements - the client has only an object view of the data!

b) Component Reusability

An important factor worth considering is a products commercial viability. Market demand determines whether a component is commercially viable or should be used only within your own organization. Typical examples include components that are directly linked to hardware such as monitoring components for alarm systems. Unless the components can be sold separately from the hardware the ability to sell the product online is greatly reduced.

Components that can be integrated without any consultation will succeed in what's known as the 'Open Market'. This market allows components to be distributed without any consultation or tailoring service. All information regarding the product is supplied in online documentation such as demonstrations, evaluations, help files and sample code.

For more information on the open market browse to:
http://www.componentsource.com/services/cbdiopen_market.asp

c) Expert Functionality

Expertise and knowledge are the two areas you should focus on when writing a software component. If you are developing a component from scratch then consider the components already on the market and assess whether you could offer a different or superior solution. Where possible write components that are related to your core business area. It's likely that these functions will be more valuable than peripheral functionality designed to provide a basic solution. For example, if your core business provides insurance underwriting services then concentrate on these core functions first as opposed to peripheral components such as a basic user interface components for data presentation in a grid or as a chart of graph.

Component Architectures

Where are components installed? - Components, unlike applications are deployed in either a client or sever environment. However, this can depend on the component containing a graphical user interface (GUI) A GUI or visual component would be of little use in an environment where servers run without screens. Apart from this, component functionality can run on client or server machines. However, this may be dependent on the usage of the component and other aspects defined below.

a) Client-Side Components

Client-side components can be implemented in a variety of ways depending on the functionality required. Their overall characteristic is that all logic is encapsulated and run on the client as opposed to a server that may serve many clients. Another factor unique to client-side components is licensing. Depending on complexity, client-side components may be restricted with user run-time licenses. Due to the nature of a client component it is possible that unique licenses are required per client machine. Client-side components can be implemented in the form of Presentation, Technical and Business components. Examples of each are detailed in the topic 'Component Types'.

b) Server-Side Components

Server-side components are relatively new to the commercial component market. Benefits enable the developer to provide solutions that run on a per server basis. These components serve many clients simultaneously without significant performance loss. Server-side components can also be upgraded efficiently removing the complexities of updating potentially thousands of desktop machines. Component logic is often run on powerful servers as opposed to a desktop machine. This makes the server-side component an excellent candidate for systems that require efficient throughput and performance.

c) Exceptions

Where possible you should design components in either a client or server architecture. However, there are a few components that are exceptions to the two definitions above. Typical examples include components that have a user interface that run in an client environment and are tightly coupled to components that run in a server environment e.g. stock/trading systems. These architectures exist for security reasons only i.e. the server component will only communicate with a specific client component and the client component will only communicate to a specific server component.

Component Types

What types of component are there? - Two main types of component exist - visual and non-visual components. Both types can encapsulate either technical or business knowledge. The differences between the two are dependent on functionality. For example if the component provides only a benefit to the developer e.g. a TCP/IP communication library then the component is categorized as technical. Business components provide a benefit to the developer and end-user by encapsulating business knowledge. Typical examples include address formatting and credit card validation components. Both visual and non-visual components have their benefits and in the following topics we will look at different examples of both in a client and server based environment.

a) Visual Components

Visual components present a pre-designed presentation interface to the user. Examples include data grids and charting components. These components are traditionally implemented as

JavaBeans™ Components, which is a Java technology-based component model. Visual components appear graphically in a component toolbox in Integrated Development Environments (IDE) such as IBM's VisualAge for Java . This allows the developer to select and physically draw the component onto a form and then manipulate properties via a design interface known as a property sheet. Another characteristic is that component functionality is run on the desktop machine as opposed to a powerful server. Because of this visual components are relatively lightweight on processing power. Visual components also provide developer licensing built in. This prevents users from copying the component into a development environment and using it at design-time. Developer licenses come as a file that must be on the users system in order for the component to work correctly.

Client and Server Based Examples - *The Java Abstract Windowing Toolkit (AWT) and the Java Swing component set for building graphical user interfaces are examples of JavaBeans component implementations for client development. Server-side visual components such as Java applets reside on the web server and are automatically downloaded along with the web page that includes them. The downloaded applets execute within a "sandbox" on the client, thus protecting the client's environment from any malicious applet*.

b) Non-Visual Components

Non-visual components do not provide a pre-designed presentation interface to the user. These components are implemented as Enterprise JavaBeans Components or non-visual JavaBeans Components. For the EJB-based components, the functionality of the components is exposed only through the business interfaces, known as *remote* interfaces. For JavaBeans Components, the functionality is exposed primarily through properties of the beans. A non-visual JavaBean Component may still be manipulated and configured in a graphical development environment. The JavaBean Components specification also specifies a helper class, known as a *BeanInfo* class, that may be implemented to expose the functionality of a non-visual JavaBean Component. Using *BeanInfo*, the bean developer may expose the properties, methods and events of a component. Non-visual components are adaptable and can be run in either client or server environments. This allows the functionality to be plugged into any n-tier architecture providing the application developer with a universal solution. Non-visual components, typically designed to run in a server environment, allow many clients to access functionality simultaneously without loss in performance. Typical examples include online housekeeping functions that require the dedicated processing power of a server.

Client and Server Based Example - *ShoppingCart EJB, which is one the components of BEA WebLogic Commerce Server™ , is an example of a 'non-visual' component that executes in a EJB server environment. This component encapsulates the functionality of an online shopping cart used by e-businesses. In the past, most online stores had their own proprietary implementations of the shopping carts. The ShoppingCart EJB enables them use a well-tested, well-designed component in a "plug-and-play" fashion - they simply need to integrate and configure the EJB in their applications. The DataAccess beans are an example of components that execute both in a client and a server environment. These components hide all the details of accessing data from commercial database systems. The users of these components build their applications based on the component interface without coupling their code to the actual database specific code. The feature enables the clients to access any database system without having to modify or recompile their code*.

Implementation Language Issues

Enterprise JavaBeans Components may only be implemented using the Java programming language. The EJB standard specifies all the services provided to server-side side components in terms of Java technology interfaces and classes. Since the EJBs depend upon these services, they must be implemented using Java technology.

The Java 2 Platform Enterprise Edition™ (J2EE™) from Sun Microsystems specifies the support for using *RMI/IIOP* as the protocol of communication between clients and enterprise beans. Using *RMI/IIOP* enables the clients to use programming languages such as C++, C, Cobol and Java technology. *RMI* refers to Java's *Remote Method Invocation*. IIOP refers to the *Internet*

Inter-Orb Protocol. IIOP, specified by the Object Management Group (OMG), is the communication protocol for CORBA™ (Common Object Request Broker Architecture) based systems. CORBA enables the interoperability between systems written in different programming languages. In CORBA, you can have a client implemented using C, C++ or COBOL that may invoke remote methods on a server object implemented in, say, Java. This interoperability is possible because of IIOP.

a) Java Technology

Java technology has emerged as a very popular language for building E-business applications. Java technology comes with a rich set of pre-built classes or components. One of the strong features of Java technology is that it is portable. You can write your application once and run it on any machine that supports a Java Virtual Machine™ (JVM™). Java technology enables the development of concurrent, multi-threaded programs. Classes may be loaded dynamically, even from across the Internet, in the run-time system. The language itself is very secure, in that, you cannot 'forge pointers' to arbitrary areas in memory. The memory management is done automatically, thus freeing the developer from the mundane and often error-prone tasks of explicit memory management. With the introduction of EJB, Java Servlet Technology and JavaServer Pages™ (JSP™), Java technology has become an important language for building server applications.

b) Borland JBuilder

Borland JBuilder provides rapid development capabilities for Pure Java Enterprise JavaBeans that are consistent with its well-established support for conventional Java Beans. It includes visual two-way EJB designers for adding and editing properties, methods and events; wizards for creating entity and session beans, including home and remote interfaces; and an EJB Test Client wizard for testing your applications.

JBuilder 4 introduced the concept of an EJB group, which is a logical grouping of one or more beans that will be deployed to a jar file. This feature allows the user to create several EJB jars in a single project; an EJB group wizard simplifies the creation of EJB groups. For debugging, JBuilder 4 lets you launch a vendor's EJB container, and allows you to set breakpoints and debug your EJBs. Through the debugger, you can even step into server-side code from the client-side. The Entity Bean Modeler lets you create entity beans that map to existing tables, enabling object-to-relational database mapping of data sources, tables, and fields to entity beans. The modeler will create all the necessary Java code.

JBuilder also simplifies deployment of your EJBs. A deployment descriptor gets created when you create your EJB group, or you can import existing deployment descriptors into your EJB group. You can use the visual Deployment Descriptor Editor to edit information contained in the deployment descriptors, such as runtime environment properties for the EJB. The deployment wizard lets you the deployment tool of your target appserver. JBuilder provides tight integration with Borland Application Server 4.1and WebLogic 5.1, and allows the flexibility of adding other target application servers.

Java 2, Enterprise Edition

The J2EE platform is designed to provide server-side and client-side support for developing enterprise, multi-tier applications. Such applications are typically configured as a client tier to provide the user interface, one or more middle-tier modules that provide client services and business logic for an application, and backend enterprise information systems providing data management.

Developing applications using the J2EE platform frees the application developer to focus on the business application logic, while the platform is responsible for the following services.

- **Transaction Management** Handles database queries and updates, distributed transactions, commit, rollback, and crash recovery

Security and Authentication - Restricts access to business objects and data, based on client identities, roles, and models

- **Database Access** Manages caching and database connections through the JDBC interface, which provides vendor independent API to access most databases
- **Data Synchronization** Maintains the consistency of in-memory objects with their persistent couterparts
- **Messaging Services** Accesses Message Oriented Middleware (MOM) (like IBM s MQ series)
- **Naming and Directory Services** Provide access to servers like the Lightweight Directory Access Protocol (LDAP)
- **Resource Management** Reuses database connections through database connection pools
- **Load Balancing** Directs client requests across the servers
- **Concurrency Control** Regulates execution of EJB objects, which ensure that a business object is executed by only one thread at a time

Central to the J2EE platform is the notion of containers. Containers are standardized runtime environments that provide specific component services. Components can expect these services to be available on any J2EE platform from any vendor. For example, all EJB containers provide automated support for transaction and life cycle management servces. Containers also provide standardized access to enterprise information systems.

In addition, containers provide a mechanism for selecting application behaviors at assembly or deployment time through the use of deployment descriptors.

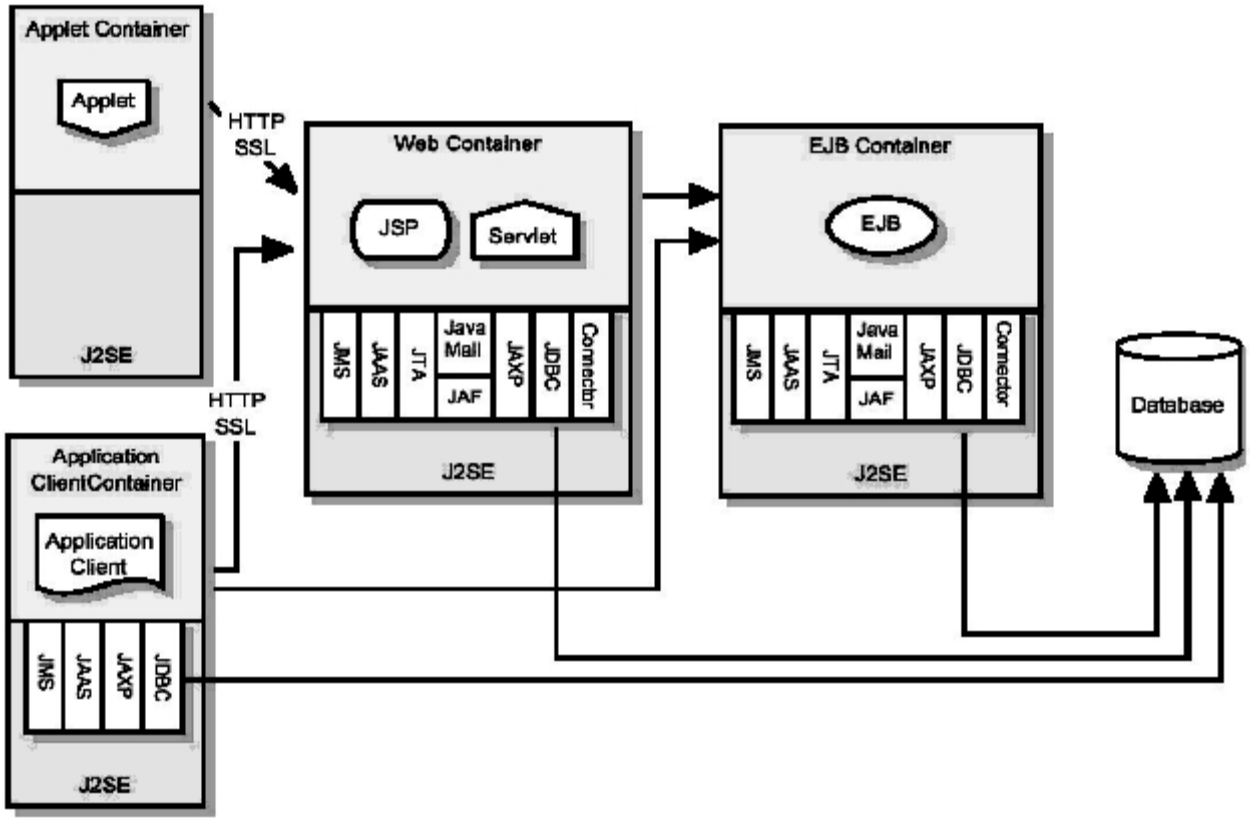

Figure 1

Figure 1 illustrates the J2EE architecture in a multi-tiered environment. J2EE supports four separate types of containers, corresponding to the four types of components:

1. Application client containers
2. Applet containers
3. Web containers
4. Enterprise bean containers

All types of containers must provide Java 2 Platform, Standard Edition (J2SE) V 1.3 compatible runtime environment.

J2EE Technologies

The J2EE architecture is based on the following Java technologies:

a) Component Technologies

- Enterprise JavaBeans (EJB) for building business logic and objects
    - Entity, Session, and Message-Driven beans
- Servlets for building the controller logic
- JSP for building the presentation logic

b) Service Technologies

- JDBC for accessing databases
- JTA/JTS to manage transactions
- JNDI to interface with enterprise naming and directory services
- Connectors to integrate with proprietary enterprise information systems
- JAXP for parsing XML documents in a portable fashion
- JAAS for plugging in authentication and authorization modules

c) Communication Technologies

- Java IDL to interoperate with CORBA-based systems
- RMI-IIOP to interoperate with CORBA-based systems
- JMS to integrate with enterprise messaging systems such as IBM MQ series
- JavaMail to generate and send email programmatically

Designing Systems for J2EE

A well-designed J2EE application invariably follows the Model-View-Controller (MVC) design

pattern long favored by graphical user interface developers. MVC pattern is utilized heavily in J2EE architecture as it is a good way to divide functionality among objects to minimize the degree of coupling.

- Model represents application data and the business rules that govern access to this data
- View renders the contents of a model
- Controller defines application behavior

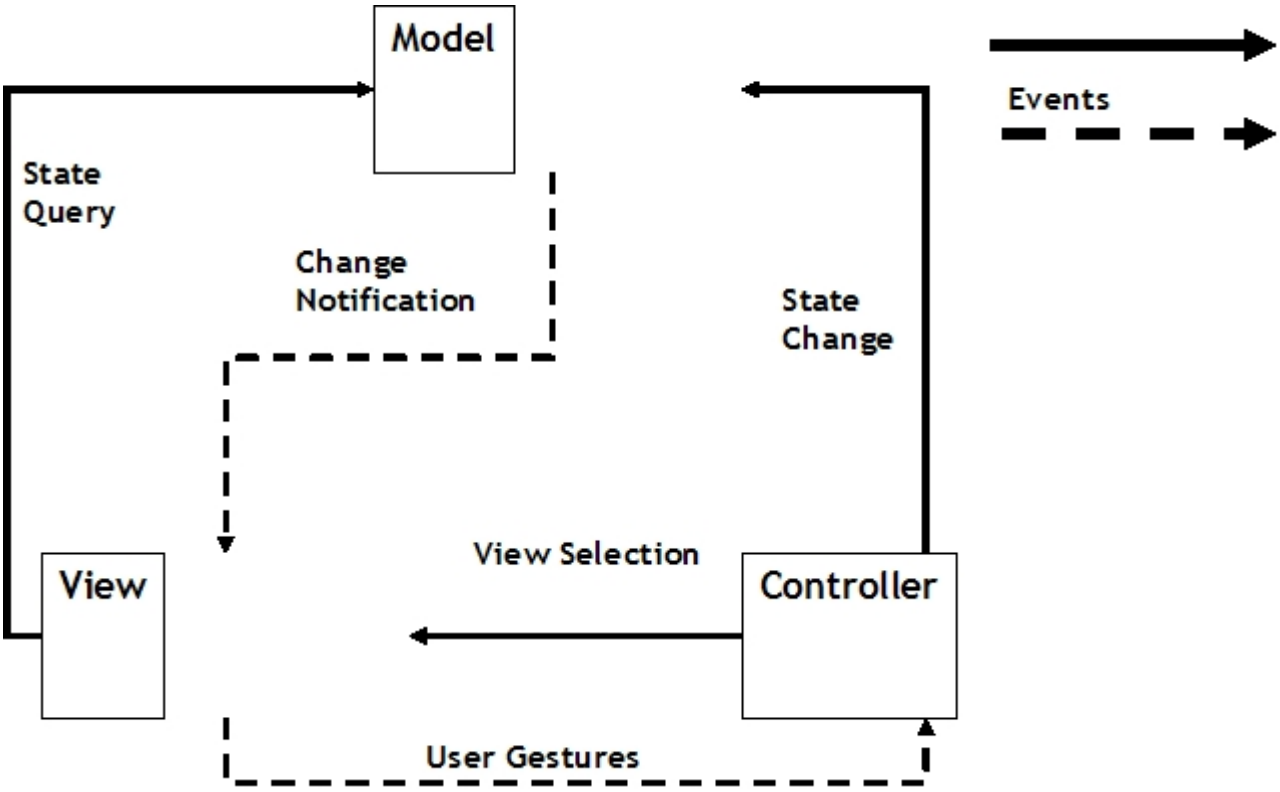

Figure 2

Figure 2 illustrates the basic flow in a MVC architecture. The MVC pattern was originally developed for the traditional graphical user interface based applications. Yet it is straightforward to map these concepts into the domain of multi-tier J2EE applications: user gestures are HTTP requests, *controller* is implemented by Java servlets and session beans (servlet does the flow control and session beans embody business logic). The *model* is implemented using entity beans, and the *view* is implemented using JSP pages.

One of the advantages of MVC pattern is that one model can have multiple views and changing views does not change the model code.

Packaging J2EE Applications

The process of assembling components into modules and modules into enterprise applications is called packaging. The process of installing and customizing an application in an operational environment is called deployment. The J2EE platform provides facilities to make the packaging and deployment process simple. It uses JAR files as the standard package for modules and applications, and XML-based deployment descriptors for customizing components and applications. Creation of a J2EE application is a two-step process. First, create the J2EE modules, for example, EJB module. Second, package these modules together to create the J2EE application. Also note that all J2EE modules are independently deployable units.

A J2EE Application consists of:

- one or more J2EE modules, e.g. EJB module packaged as EJB jar file, or a web module packed as a WAR (Web Archive) file
- one J2EE application deployment descriptor - application.xml
    - application.xml lists all the J2EE modules

And each module consists of:

- one or more J2EE components
    - e.g. multiple Enterprise JavaBeans
- one component level deployment descriptor
    - e.g. ejb-jar.xml

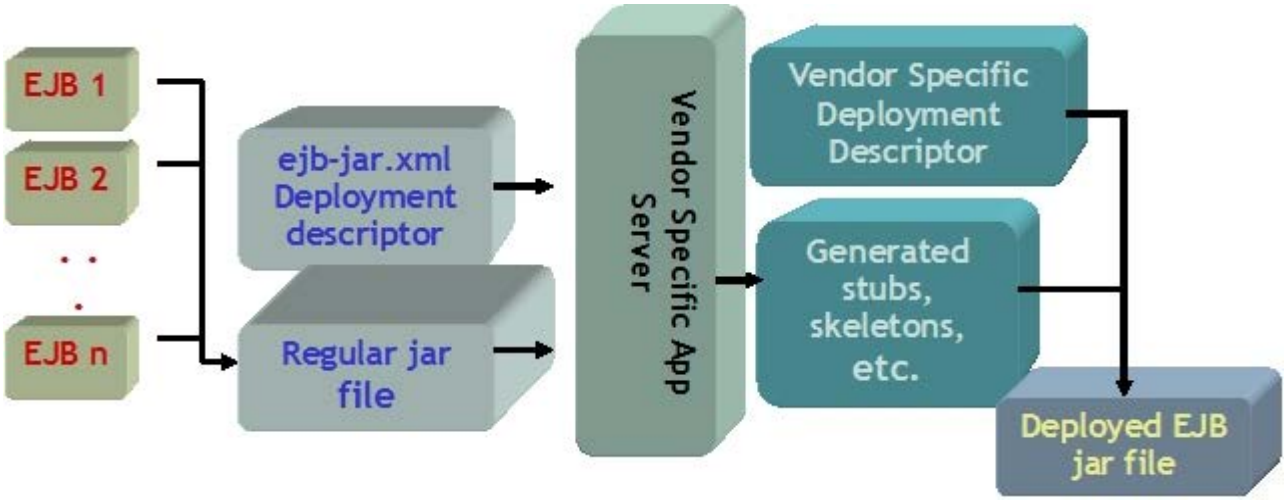

Figure 3

An EJB module is the smallest deployable and usable unit of enterprise beans, and is packaged and deployed as an EJB JAR file as shown in figure 3. It contains:

- Java class files for the enterprise beans and their remote and home interfaces and for an entity bean, possibly its primary key class.
- Other Java classes that the implementation of the EJB uses.
- An EJB deployment descriptor that provides both the structural and application assembly information for the enterprise beans.

You can package one enterprise bean in each module, all enterprise beans of an J2EE application in one module, or divide EJB modules according to their functionality or origin. For example, all off-the-shelf enterprise beans in one module and all in-house enterprise beans in one module, etc.

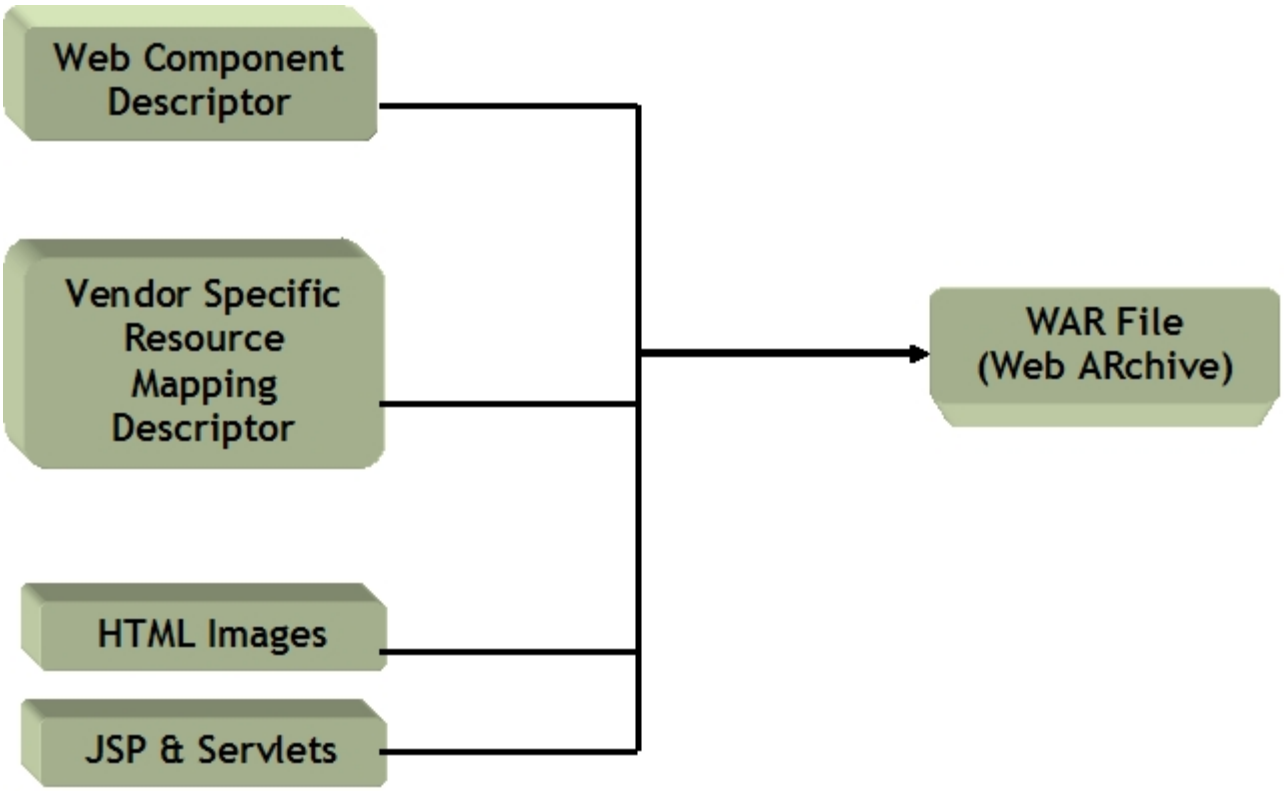

Figure 4

A Web module is the smallest deployable and usable unit of Web resources. A Web module is packaged and deployed as a Web ARchive (WAR) file, a JAR file with a .war extension as illustrated in figure 4. It contains:

- Java class files for the servlets and the classes that they depend on, optionally packaged as a library JAR file.
- JSP pages and their helper Java classes.
- Static documents (for example, HTML, images, sound files, and so on.)
- Applets.
- A Web deployment descriptor.

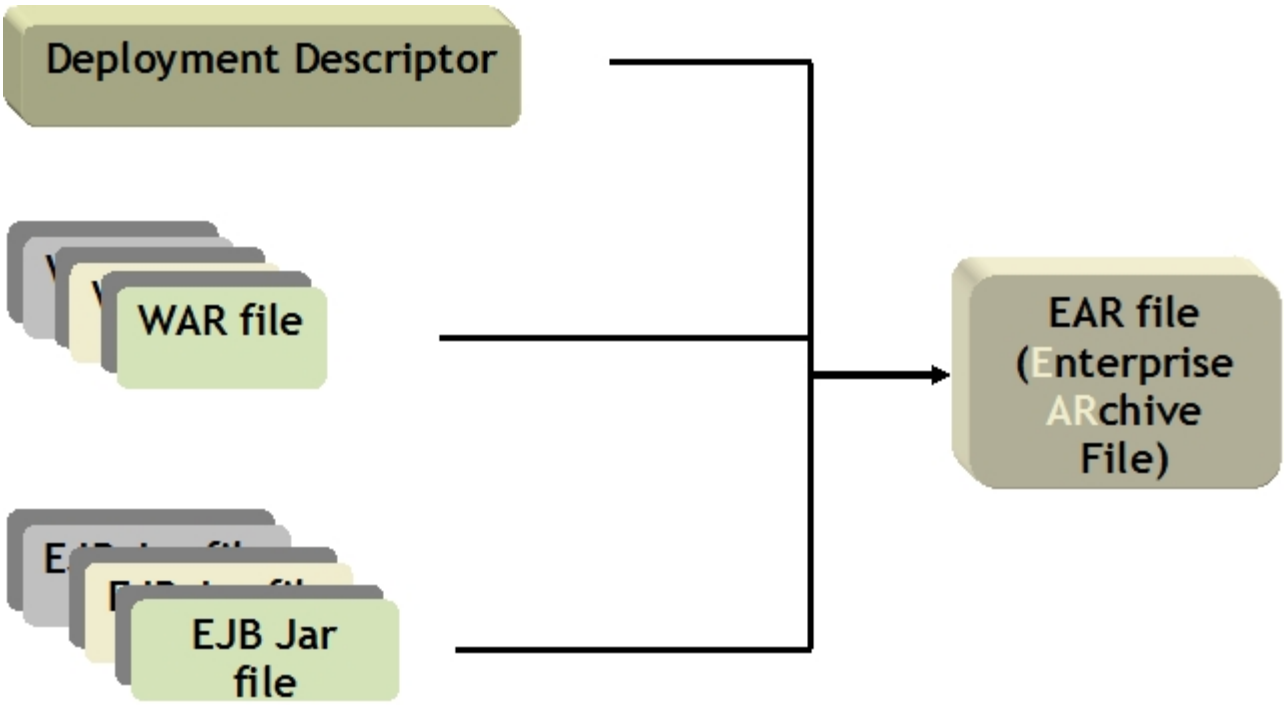

Figure 5

Figure 5 illustrates the packaging of a J2EE application into an EAR file.

Java Servlets

Java Servlets are web components of a J2EE application that extend the functionality of a Web server. In J2EE applications, servlets are hosted by the Web container. Servlets (HttpServlets) receive an HTTP request from a client and dynamically generate a response either in HTML or XML and send this response back to the client. Servlets may also access enterprise resources such as databases or EJB in the EJB server. Servlets maintain session information on behalf of the clients accessing them and may also interact with other servlets. As mentioned above, servlets play the role of a *controller* in a J2EE application.

JavaServer Pages

JSP (JavaServer Pages) is a Java-based, server-side scripting technology that enables the dynamic creation of web content. JSP pages are text files, usually with .jsp suffix. The Web container hosts JSP files. JSP files contain HTML along with embedded Java code. When a request for a JSP page is received, the HTML portion is passed straight through and the Java code portions are executed. The generated dynamic content (e.g. HTML) is spliced into the page before sent back to the browser. A JSP page is compiled into a servlet before execution.

JSP enjoys all the advantages of Java technology: an object-oriented language with strong typing, encapsulation, exception handling, automatic memory management, multi-threading, as well as security and platform independence. JSP also has full access to the underlying Java platform, database access, directory services, distributed computing, and so forth. JSP development emphasizes component-centric application design, and enables separation of presentation logic from business logic. Java developers are responsible for developing business logic components and exposing the components functionalities through Java Beans and custom tags. Web page designers are responsible for using the Java beans and custom tags to construct the presentation and layout portions of a web document.

Java Messaging Service (JMS)

JMS is an API for accessing enterprise-messaging systems such as IBM MQSeries etc, in a portable and standard way. The messaging application code is decoupled from the message-oriented middleware. This allows the application developer to switch the messaging system without having to change (or even re-compile!) the application. A JMS message contains well-

defined information that describes specific business actions. The JMS supports both point-to-point and publish-subscribe models of messaging systems.

Java API for XML Parsing (JAXP)

JAXP provides a common interface for creating and using the standard SAX, DOM, XSLT APIs in Java that is independent of vendor s implementation for these APIs. This enables a J2EE developer to write portable code for handling XML documents. The API includes facilities for SAX parsing, DOM parsing and transformations using XSLT.

Java Authentication and Authorization Service (JAAS)

Java Authentication and Authorization Service (JAAS) provides a pluggable architecture that provides a set of classes to authenticate user to perform certain operations. Permissions are granted in a way that is similar to the Java sandbox security model. A program needs to be modified to use JAAS. By default, Java programs do not use JAAS. JAAS is required in J2EE 1.3 and will be part of JDK 1.4.

Enterprise JavaBeans Components

Introduction

Enterprise JavaBeans Components is a specification of the server-side component model for building and deploying enterprise-class applications. The enterprise application developer may build his/her application as a set of interconnected enterprise beans and deploy such an application in an EJB-compliant application server. The specification also mandates that EJB-compliant application servers provide a set of enterprise services through well-defined Java interfaces. The standard also specifies certain interfaces that all developer-written components should implement in order for them to be deployed in an EJB-compliant application server. In other words, the EJB server promises a set of services and, in return, expects the components (enterprise beans) to implement certain interfaces so the server may manage these components. The EJB standard enables the enterprise developer to focus on the actual business logic of the application, encoded in the beans, and the EJB server is responsible for all the enterprise services such as *Concurrency*, *Persistence*, *Transaction Management*, *Security Management, Naming Services, Object Distribution* and *Resource Management*. EJB based applications are secure, robust, scalable, portable and transactional. To understand the problem space EJB addresses, let us consider the motivations for using this technology.

Why use Enterprise JavaBeans Components?

To understand the need for EJB, it is useful to understand the relative merits of TP Monitors and CORBA systems. Here we present the strengths and weakness of both these architectures.

a) Transaction Processing Monitors

Traditionally enterprise-class systems were implemented using systems generally known as Transaction Processing Monitors or TP Monitors. Large scale enterprise applications such as banking, insurance and airline reservation systems are built using TP monitors. Some of the popular TP Monitors are IBM's CICS® and BEA Tuxedo®. TP Monitors were a natural choice for enterprise applications because they handled all the database transactions efficiently and in a manner where the enterprise developer did not have to explicitly write code to manage transactions.

TP Monitors are designed to handle large workloads and manage concurrent access to enterprise application resources. TP Monitors also handle the security management, database access and the network connectivity for the enterprise applications. In other words, TP Monitors provide these services so that an application programmer may focus on implementing the business logic of the application.

In a way, you may think of the TP monitors as an Operating System for business applications.

When you use a normal Operating System, you expect a host for services such as virtual memory management, file system management etc. from the system, you do not code for those services in your normal applications. Similarly, enterprise applications may expect to find services pertaining to Transaction, Security, Concurrency, Resource Management etc. from the TP Monitors.

Given that TP Monitors do so much for an enterprise programmer, why not use them? Why worry about EJB? For all their strengths, typical TP monitors suffer from two major drawbacks. First, most TP monitors do not have a component model. The services are offered, typically, as functions which leads to monolithic applications as opposed to component based applications. It is very hard to replace one service implementation with another. For example, for an e-commerce application, you might want to have the flexibility to replace the credit card processing object with a superior implementation. Lack of an object model prevents you from doing that easily. It also makes it hard for you to implement 'objects' that reside on the server but are dedicated to specific clients and execute programs, on the server, on behalf of their 'owner' clients. A typical example of this kind of application is a Shopping cart or a Mobile Smart Agent.

The second, and more serious, problem with typical TP Monitors is the lack of portability of enterprise applications implemented using them. Enterprise applications implemented on TP Monitors are usually tied to a proprietary API and model. It is usually a large effort to port an enterprise application from TP Monitor system to some other vendor's TP Monitor. The problem arises because there is no standard for TP Monitors. Each TP Monitor may implement all the necessary services that an application might require and use, but since each vendor exposes the services in a proprietary way, the enterprise application becomes less-portable.

b) CORBA™

At this point it might be appropriate to present the other technology that attempts to address the shortcomings of the TP Monitors. CORBA is specified by the OMG and is an industry standard. CORBA provides a component model for the server-side programming and also specifies a host of services called CORBAServices such as OTS (Object Transaction Services) and SecurityServices. Server applications written in CORBA are for the most part portable across multiple ORBS (Object Request Brokers).

c) Component Transaction Monitors

Enterprise JavaBeans Components essentially combine the strengths of traditional TP Monitors and CORBA. In other words, using EJB, you get all the benefits of TP Monitors and the portability and component model of CORBA. EJB servers belong to a class of systems commonly known as ***Component Transaction Monitors*** or **CTM**. Using an EJB-compliant server, a developer may build enterprise-class applications rapidly, focusing purely on the business and application logic. All the infrastructure services are now the responsibility of the server and are provided automatically to the application. The developer can configure these services declaratively - the configuration is specified using XML. The enterprise application is implemented as a set of EJB components, with well defined business interfaces, that are deployed on an EJB server. The developer is no longer tied to any one implementation of the application server and may simply deploy her application on any EJB-compliant application server, such as *iPlanet*, *WebLogic, WebSphere* or *iPortal*, without even recompiling the application! The EJB server generates the appropriate objects to provide the enterprise services and ties them with the developer-implemented components during the application deployment.

In summary, you would use the EJB component architecture if you want to build portable, component-based, scalable, secure, transactional and robust enterprise applications rapidly. You would also use EJBs if you want to implement only the business logic and want the application server to handle all the system services.

Architecture of Enterprise JavaBeans Components

Enterprise JavaBeans Components are components that are deployed on EJB-compliant application servers. EJB components are contained within special objects known as containers.

The container is responsible for providing the system-level enterprise services to the enterprise beans.

a) Basic EJB Concepts

The EJB components reside on the server. Their functionality is exposed to their clients and callers through well-defined interfaces. The callers may be in the same process space as the server objects or they could be in another process and even another machine. The callers communicate with the server object through the use of a *Client-Proxy* object. This *Client-Proxy* object has the same interface as the EJB component, but for each method invocation by the caller on the *Client-Proxy*, the proxy simply sends a corresponding request to the real bean implementing the service on the server. This involves serializing the caller's request and sending it, perhaps using network connections, to the server. On the server, another object, called a *Skeleton Object*, intercepts this request and re-creates the original method call from the serialized input it receives and eventually invokes the relevant method on the enterprise bean. The *Skeleton* object then obtains the return value from the enterprise bean, upon the completion of the bean method invocation, and sends it back to the *Client-Proxy* in a serialized form. Finally the *Client-Proxy* de-serializes the return value and constructs the values of the appropriate types and returns them to the original caller. The presence of the *Client-Proxy* and the *Skeleton* is completely transparent to the caller - the caller thinks it is communicating with the enterprise bean. On the server, placed between the Skeleton object and the enterprise bean, is another object called the *Server-Proxy* or the EJBObject. Every call from the *Skeleton* to the enterprise bean is routed through the EJBObject, which checks for concurrent access, transactional contexts and security contexts before allowing the methods on the enterprise bean to be invoked. You may think of this object as a form of *guard* object for the bean.

In summary, the *Client-Proxy* object resides on the client and is the server object's proxy on the client. The *Client-Proxy* is responsible for a) locating the target object (the server object for which this *Client-Proxy* serves as a proxy), b) sending the request to the server, usually in a serialized form, over the network and c) for retrieving the return values back from the server and constructing the return value for the caller. The *Skeleton* object and the EJBObject reside on the server along with the enterprise bean. The *Skeleton* is responsible for a) receiving the request from the client, possibly from across the network, b) de-serializing and constructing a call to the corresponding EJBObject and c) for obtaining the return value from the call to the EJBObject and sending that value, in a serialized fashion, possibly over the network, to the caller. The caller is invariably a *Client-Proxy*. The EJBObject, after it receives the call from the *Skeleton* object, ensures that it is safe and legal to carry out that operation. If the operation is legal, subject to security, transaction and concurrency policies, the EJBObject calls the corresponding method on the enterprise bean and returns any return value it gets back to the *Skeleton* object. And finally, the enterprise bean simply executes the business method that it had been programmed to do so by the bean developer.

As you may notice, the bean developer simply provides the bean implementation that concerns itself only with business logic. All the objects that provide the enterprise services are generated by the system. The following table summarizes the various objects and their roles.

Object	Created by	Resides on	Service
ClientProxy	EJB Compiler	Client	Marshals client requests to server and unmarshals return values
Skeleton	EJB Compiler	Server	Unmarshals ClientProxy's request and invokes the corresponding method on the EJBObject and marshals the return value back to the ClientProxy.
EJBObject	EJB Compiler	Server	Intercepts all method invocations on the enterprise bean and provides the EJB services. Delegates call to the enterprise bean.
Enterprise	Bean		Implements the real business logic of the

Bean	Developer	Server	application for the methods advertised in the remote interface.

b) Types of Enterprise JavaBeans Components

Enterprise JavaBeans are divided into two broad categories:

- Session Beans
- Entity Beans
- Message-Driven Beans

Session beans are used to model process or task-oriented aspects of your application. It is useful to think of a session bean as a client's representative on the server. Session beans provide a session-oriented view of the server to the client. Entity beans are used to model data, typically persistent data, of an enterprise application. Enterprise beans correspond to rows in relational database tables or to objects in object-oriented database systems. Both Session beans and Entity beans are synchronous i.e. the caller, upon a method invocation on these types of beans, blocks until the method execution completes. Message-driven beans, on the other hand, enable the development of asynchronous applications. A Message-driven bean is basically a message consumer, registered as a Message Listener on a Message Queue. Whenever messages are delivered on the message queue, the corresponding Message-driven beans are notified by the EJB container. We shall explore all three kind of beans in greater detail later in this document.

c) Creating Enterprise JavaBeans Components

To define an EJB, the bean developer has to define the:

1. Remote interface
2. Home interface
3. Bean implementation
4. Deployment description information

Remote Interface

This interface exposes the business functionality of the enterprise bean in a declarative way. That is, the interface lists what functionality this bean would provide and not how it would provide it. From a client programmer's perspective, the interface lists all the business methods of the enterprise bean that calling applications may invoke when using this bean. Components for the open market are defined as 'Black Box'. This means that all functionality is encapsulated and therefore no implementation code is available to the user except the Remote interface. The EJB compiler generates the Server-Proxy (EJBObject) and the Client-Proxy classes that implement this interface. Message-driven beans do not have a remote interface.

Home Interface

This interface exposes the life cycle methods of the enterprise bean. This interface contains methods to create, remove or find beans on the server. The clients of the EJB use this interface to manage the life-cycle of the enterprise beans on the server. You may also think of this as the factory interface for the EJB components. The EJB Compiler generates a pair of classes, one for the client-side and the other for the server-side, that implement this interface. The server-side implementation of the home object has the code to instantiate and initialize the enterprise bean, associate it with an EJBObject and finally return to the client a corresponding instance, in a serialized manner, of the Client-Proxy class. The client-side implementation of this interface acts as a client-proxy for the server-side implementation of this class. When an enterprise bean is deployed, the EJB server instantiates the server-side home object and registers this object in a Naming Context (Name Server) under a name specified in the deployment descriptor.

Message-driven beans do not have a home interface.

Enterprise Bean Implementation

This class contains the real implementation of the EJB component. The EJB implementation class has to implement:

1. All the methods specified in the remote interface.
2. Methods corresponding to methods specified in the home interface.
3. Callback methods specified by the EJB specification so that the EJB container may manage and interact with the enterprise bean by invoking these methods. All session beans must implement the interface, `javax.ejb.SessionBean`. And all entity beans must implement the interface, `javax.ejb.EntityBean`.

Deployment Descriptors

Every EJB needs a deployment description document to aid the application server in deploying the EJB and in generating the proxy classes for the home and remote interfaces. The deployment description is specified using XML. Sun Microsystems has defined a Document Type Definition (DTD) for EJB deployment descriptors. In the deployment descriptor, the bean developer associates the Remote Interface, the Home Interface and the Bean Implementation. The name, under which the server-side home object is registered on a Name Server, is also specified here. The deployment descriptor is also used to declaratively specify the transaction and security attributes of various methods of the remote and home interfaces. The deployment descriptor file is used to define Principals denoting users, groups of users or devices. The deployment descriptor is also used to specify the Method Permissions. Principals are associated with beans methods defining an implicit access control policy. The deployment descriptor is also used to specify environmental variables, resource management information, enterprise bean type information, database connection pools and database mapping information for Entity beans. If the Entity bean contains any dependent data (i.e. data from other tables), then such relationships also need to be specified in the deployment descriptor. The bean developer must provide reasonable default values for various attributes in the Deployment Descriptor file - the deployer may override some of these during deployment of the bean. For Message-driven beans, the message queue, on which the bean is registered as a message listener, is also specified in the deployment descriptor file.

Error Handling

Handling errors in a component is not the same as handling application errors. Firstly, you need to consider that any error not handled in a EJB will be sent back to the client that called the method. For that reason, you must ensure that the information the client receives is meaningful. A client interface should be totally unaware that a component it is using may be running other processes. Therefore any error that occurs should be processed by the component and only passed back to the client in a form that can be interpreted by that client. All errors produced by a component must be included in the specification of the component - this is what the client depends on. Below are the main techniques for handling errors in EJB.

Handling Errors Internally - Handling errors within an EJB is no different to handling errors in a standard application. If a method unexpectedly generates an error then unless an error handling routine is included, the calling application will crash. To avoid this situation, intercept the error, assess its severity and take corrective action, either by resuming to a specific line of code or by *throwing* an appropriate exception to the calling function.

Passing Errors Back to the Client - To return an error back to the calling client you must throw a *RemoteException*. Throwing a

relevant RemoteException will let a client know about the cause of the error. All EJB clients must be prepared to deal with the possibility of receiving a RemoteException.

Raising Errors from Error Handlers - The majority of methods and properties you write will contain error handler routines. Where an error handler receives an unexpected error then returning a generic 'unexpected error' exception will not help the client find a solution. A good practice is to return the methods name that failed and the parameters that were passed to it. This information can then be passed back to the component author for investigation.

Handling Errors from Another Component - If your EJB references a third party EJB then you must handle all errors (known or unknown) that the secondary component may generate. Developers using your component may have no knowledge of these dependencies. Because of this, you must not raise these errors to your client application, unless they are specified in the interface of your component.

Threading

The EJB architecture assumes responsibility for managing concurrency. Do not try to explicitly manage threads or thread synchronization as this may interfere with the EJB server's thread management. Also, the EJB server is free to use multiple JVMs and your explicit thread management may not work correctly.

Design Considerations

How do I develop a software component? - Before writing a component you should analyze the functionality and architecture first. In this section we discuss components functional boundaries, assess where a component will physically run and how to implement an extensible interface. Considering these elements will prevent the inclusion of unnecessary functions and provide a focused solution for developers.

a) Identify Component Scope

It is important when designing a component to identify the functionality that should be included and the functionality that is best incorporated into another component. A component should allow a developer to integrate a precise solution as opposed to one that provides features over and above a basic requirement. For example, designing a business component that provides addressing services could include various functions such as address duplication, post coding and address formatting. In this example the three functions are mutually exclusive and should be implemented separately.

However, if the component was an address duplication component that incorporated extended functionality e.g. off-line batch duplication then this functionality should be included. It is possible to create one component that can be sold at three different levels. By using the ComponentSource licensing technology (C-LIC), it is possible to block extended functionality. This allows authors to publish one component but sell a separate standard, professional and enterprise edition.

Defining component scope will help ensure a component does not become monolithic and mimic an application without an interface. Unbundling functionality into separate components will prevent the component from becoming over complex and difficult to maintain. The advent of online purchasing and the removal of packaging and shipping costs has meant there no longer is a need to bundle disparate functionality into one component or to market several components in one suite. Removal of this traditional cost allows authors to publish highly focussed discrete components and provide customers with a wider choice.

b) Choose Architecture

Choosing architecture will depend on the functionality the component will provide. As discussed earlier in the chapter 'Component Overview' client components are often visual in some respect

such as grids, charting and toolbar components. However, non-visual components may fall into this category if the functionality is 'lightweight' and does not severely impact the processor. Typical examples include file encryption and communication components. If the component functionality can be used in a multi-user environment then consider developing a scalable server based component. This should be implemented preferably as an Enterprise JavaBean Component for scalability and transaction handling.

Installing components in a server environment is less time consuming than having to install a component on several client machines. The improved performance and upgradeability benefit that server components offer is reflected in the price and provides component authors with an opportunity to generate revenues based on a server architecture. Server based components will provide the backbone to future Application Service Providers (ASP) and consequently developing server components now, will position you for the future growth in this market.

c) Prototype Interface

Prototyping a component interface can be a useful exercise and will help determine the complexity of integrating the component into an application. Component integration should be a relatively quick process. If the interface has hundreds of public properties, methods and events then it's probably too complex and will confuse users and generate support issues. A technique, which can help prevent this problem, is to write the help file before implementation. This will help you detail a functional specification and pinpoint any areas that could be consolidated or improved upon.

Session Beans

Session beans can be thought of as extensions of client's application on the server. Session beans represent transient activities. Session beans should be used to model the functional part of an enterprise application. Session beans are not sharable by multiple clients, i.e. each client has a reference to its own, private session bean on the server. Session beans are divided in to two categories:

- Stateless Session Beans
- Stateful Session Beans

a) Stateless Session Beans

Stateless Session beans do not maintain any state (at least none that the client can depend on) between multiple remote method invocations by a client on the same Client-Proxy of the Session bean. The effect of using these beans is the same as invoking a remote function on a server. Use this kind of a bean for executing some process or program on the server where the duration of the method execution constitutes a complete unit of work or transaction from the client's perspective. Example of these kinds of beans are those that perform some atomic action on behalf of a client, e.g. an `AccountManager` bean that has a business method, `createAccount`, to create a new `Account` bean on the server.

b) Stateful Session Beans

Stateful Session beans maintain state across multiple remote method invocations by a client on the same Client-Proxy of the session bean. The client, through a Client-Proxy, has a reference to session bean on the server. Every method invocation by the client on the Client-Proxy is routed to the same session bean instance on the server. The session starts when the client obtains the Client-Proxy referencing a server-side session object and ends when the client explicitly *removes* the Client-Proxy. Use this kind of a session bean where you want to carry out a long conversation with the server and want the server object to save the conversational state. An example of this kind of a bean is an online shopping cart, where a client might use the remote methods to add, remove or modify the products in the shopping cart.

Entity Beans

Entity beans represent the data part of an enterprise application. Entity beans are usually mapped either to records of a relational database system or to objects of an object oriented database. Entity beans are transactional, in that changes to their state typically occur within the context of a transaction. An instance of an Entity bean, unlike an instance of a session bean, may be

referenced by multiple clients. Concurrent access is allowed, subject to the transactional state of the bean and the transaction isolation levels in force at the time of the concurrent method invocations on the bean. The container is responsible for ensuring that the data in the bean instance and the corresponding data on the database are synchronized. All entity beans require the presence of an object, called the *Primary Key* object, that represents the primary key fields of the table that this entity bean represents. The home interface of an Entity bean is required to contain the standard finder method, ***findByPrimaryKey***. This method locates an Entity bean given a primary key object. Bean developers may also specify other finder methods, known as custom finders, which locate Entity bean(s) based on some other criteria. Entity beans are divided into two categories:

- Container Managed Persistence
- Bean Managed Persistence

a) Container Managed Persistence

For Container Managed Persistence (CMP) Entity beans, the container is responsible for reading the data from the database to populate a bean (this may entail a SQL query execution by the container) and for writing the contents of the bean to the database (this may entail a SQL insert or update to be executed by the container). The bean developer may build a CMP entity bean without having to write a single line of database code in the bean implementation. However, the instance variables of the CMP entity bean that map to specific columns on a table have to be specified in the deployment descriptor, along with the table and database names. If the CMP bean contains any fields that map to records from other tables (also known as dependent data), then such mapping also have to be explicitly specified in the deployment descriptor. This mapping information is crucial for the container to generate the necessary database code to manage persistence. If the home interface contains some custom finder methods, then the query execution code for such finders has to be specified in the deployment descriptor. These queries are specified using the query language **EJBQL**. CMP beans are easier to develop because the developer does not have to deal with database programming. However, CMP beans are not viable when you want to map entity beans onto complex joins or proprietary storage devices.

b) Bean Managed Persistence

For Bean Managed Persistence (BMP) Entity beans, the bean developer is responsible for writing the database code for reading, writing, modifying and deleting the records from the database in order to map the bean to some persistent data. Unlike CMP, the mapping of the bean to persistent data is coded into the bean's implementation. The database access methods are implemented as callbacks and the container is still responsible for invoking these methods on the bean at appropriate times during the life-time of the bean. BMP beans are more complex to develop and maintain, yet they are extremely flexible.

c) Message-Driven Beans (MDB)

Message-driven beans enable the development of asynchronous applications. A MDB bean is registered as a message recipient or message consumer on a message queue. The name of the queue is specified in the deployment descriptor. When message senders send messages on this message queue, the EJB container notifies the MDB by invoking the call-back method, **onMessage**. This method is passed a *Message* parameter encapsulating the message. The bean developer must implement the necessary business logic in this method. The message senders can be any kind of applications (i.e. they do not have to be EJB clients). A MDB bean is also anonymous in that no client (or bean) may hold a reference to a MDB bean only the EJB container may access the bean directly. A MDB bean has neither a *remote* interface nor a *home* interface.

Roles in EJB

The EJB specification defines a number of roles in the application development and deployment process. They are:

- Bean Developers (Component Authors)

- Application Assemblers
- Application Deployers
- Server Providers
- Container Providers
- Administrators

a) Bean Developer (Component Author)

This role is played by the component developer who specifies the Remote Interface, Home Interface, Bean implementation class and the deployment description. The Bean developer compiles all the source files and packages them in a JAR (Archive) file. The Bean developer then runs a vendor supplied tool, commonly referred to as an EJB Compiler, to generate the client and server proxy objects for the remote and home interfaces. Finally another JAR file is created containing all the compiled class files for developer written source files and EJB Compiler generated source files. EJBs are packaged in such JAR files.

b) Application Assembler

This role is played by the application writers who re-use or buy EJB-based components and assemble them to create applications or other components.

c) Application Deployer

An Application Deployer is typically an IT manager who deploys, after modifying the deployment description files, the application on a EJB-compliant application server. The deployer is responsible for setting the proper Access Control Lists (ACL), database mappings for CMP Entity Beans and the name of the home object.

d) Server Provider

This role is played by vendors who implement the application servers based on the EJB specification. e.g. Sun-Netscape Alliance iPlanet™, IBM® WebSphere™, BEA™ Weblogic® or IONA iPortal Application Server .

e) Container Provider

This role is played by the writers of containers that hold he EJB components. The components themselves reside within a server. At this point, since the interface between the container and the server is not fully specified, this role would be played by the Server Provider.

f) Administrator

This role is played by the EJB server administrator who would be responsible for managing the database connections, Naming Servers and performance monitoring.

EJB Services

The EJB application servers provide a host of services for the enterprise beans listed below.

- Transaction Services
- Security Services
- Naming Service
- Persistence
- Resource Management

a) Transaction Service

Transactions in EJB can be divided into two broad categories: *Bean-Managed* and *Container-Managed*. The EJB architecture requires that the EJB container support the interface, **javax.transaction.UserTransaction**, defined in the Java Transactions API (JTA). Bean-Managed Transaction beans use the **UserTransaction** to explicitly demarcate transactions. Transaction services may also be used without having to code against the **UserTransaction** interface, i.e. they may be container-managed. For container-managed transactions, the developer may specify the transaction attributes, listed in the table below, in the deployment

descriptor. Container-managed transaction attributes may be specified at the method scope . The EJB system also passes the transaction context implicitly with method calls on enterprise beans. For example, when a caller is in a transactional context and invokes a method on a bean, the caller's transaction context is passed implicitly to the bean's method. The table below lists all the transaction attributes on methods that are supported for container-managed transactions. The table also describes how the transaction context propagation may be controlled.

The bean-managed transaction attribute is specified at a bean scope (not method). All methods of such beans use the bean-managed transaction attribute.

Attribute	Description
Not Supported	When a caller invokes a method, the caller's transaction, if any, is suspended and is resumed after the method call.
Required	If the caller is in a transactional context, the called bean becomes part of the caller's transactional context. If the caller is not in a transactional context, the container starts a new transaction for the duration of the method.
Supports	If the caller is in a transactional context, the called bean becomes part of the caller's transactional context. If the caller is not in a transactional context, the container does not start a new transaction.
RequiresNew	If the caller is in a transactional context, the caller's transaction is suspended and a new transaction is created for the duration of this method execution. If the caller is not in a transactional context, the container creates a new transaction is created for the duration of this method execution.
Mandatory	The caller must be in a transactional context before invoking a method on the bean. The called bean becomes part of the caller's transactional context. If the caller is not in a transactional context, the container throws a `javax.transaction.TransactionRequired` exception.
Never	If the caller is in a transactional context, the container throws a `java.rmi.RemoteException`. If the caller is not in a transactional context, the container does not start a new transaction.

b) Security Service

The EJB security uses the `javax.security.Principal` class for identifying callers. When a client identifies itself to the server, then, for every remote method that the client invokes on the server object, the client's security context is passed implicitly to the server object. The Server-Proxy uses this security context information to authorize the invocation of the method on the enterprise bean. The authorization policies are specified in the deployment descriptor files. Even for inter-bean communication, the security context is passed implicitly, like a hidden argument. The security context is also available to the bean at run time, thus enabling the bean developer to implement object level security.

c) Naming Service

EJB servers use a naming service to register their home objects under names specified in the deployment descriptor. These naming services may be accessed using the Java Naming and Directory Interface (JNDI) API. The JNDI API has methods to register, un-register and lookup objects by name. Using JNDI enables the Naming Service provider to use any implementation as long as the API is JNDI.

d) Persistence Services

EJB architecture provides persistence support in the following two ways:

1. For CMP Entity beans, the container reads the data from the database into the bean before the first time any method is executed on the bean in a transaction. The container automatically writes out the bean to the database at the end of successful completion of a transaction, say when a commit occurs.
2. For BMP, the container calls the appropriate callback methods (ejbLoad and ejbStore) on the enterprise bean. The bean developer would have written the code in these two methods to read the data from the database and update the data into the database respectively.

As you may notice, the bean developer is no longer responsible for synchronizing the bean with the underlying database, the container handles it automatically.

e) Resource Management Services

EJB architecture provides resource management services in a number of ways. Database connections are typically pooled, as are the threads. This increases the performance noticeably. EJB servers also keep unattached enterprise bean instances in a pool and reuse these instead of invoking the memory manager. EJB servers achieve scalability by using a limited number of bean instances to serve a large number of clients. They do it by using a technique known as activation/passivation. When the system needs a bean to service a client and if the system does not have any free beans in the pool, the server 'passivates' a bean by writing out its state on to a secondary storage and reusing that beans memory. If at a later time, the passivated bean receives a request from the client, the server 'activates' it using an available bean instance.

EJB 2.0 Features

In this section we will highlight some of the new features that make the EJB2.0 platform extremely powerful and flexible for building industrial strength, robust and maintainable enterprise applications.

a) Local Objects

In EJB 1.1, the remote interfaces caused a marshal/unmarshal overhead during method invocation regardless of whether components are located on the same JVM or not. The parameters were always passed by value. EJB 2.0 introduces two new interfaces for EJB objects that may be accessed only within the container (i.e. these beans cannot be accessed remotely). The interfaces are:

- `EJBLocalHome` - corresponding to `EJBHome`
- `EJBLocalObject` - corresponding to `EJBObject`

`EJBLocalHome` and `EJBLocalObject` support callers collocated in the same EJB container and do not incur any marshal/unmarshal overhead during method invocations the parameters are passed by reference.

b) Container Managed Persistence Model

In the new model, CMP fields are no longer defined in the bean class. Instead we define abstract accessor methods for these fields. The entity bean class is now an abstract class. The CMP fields are described in deployment descriptor (DD). And, based on DD and abstract accessor methods, the container generates a concrete bean class that extends the abstract bean class. EJB 2.0 Containers are required to support both EJB 1.1 and 2.0 CMP models.

The new model also allows us to represent relationships, called Container Managed Relationships (CMR), between entity beans to closely model the relationship between tables in the schema. Local interfaces allow fine-grained entity beans to be used as dependent objects. This solves both the efficiency and persistence issues. The query language, EJB QL, provides portable way to write finder methods.

EJB 2.0 lets the container manage the relationships between entity beans. The following relationships are supported:

- One-to-one
- One-to-many
- Many-to-many

CMR fields are defined in terms of the local interfaces of the related beans. An entity bean that does not have a local interface can have only unidirectional relationships from itself to other entity beans (The lack of a local interface prevents other entity beans from having a relationship *to* it). Relationships may be either bi-directional or unidirectional. Relationships are expressed using CMR fields in the deployment descriptor.

c) EJB Query Language (EJB QL)

EJB QL is used for defining query methods in a vendor independent way (before EJB 2.0, each vendor had its own way of specifying queries). EJB QL is based on the SQL-92 syntax and is tailored to work with the abstract persistence schema of CMP 2.0. It may be used in two scenarios:

- Write queries for finder methods.
- Writing queries for abstract `ejbSelectXXX()` methods.
    - `ejbselect` methods allow a developer to push data-centric processing to the database systems.

EJB QL queries are defined in terms of the abstract persistence schema of entity beans and not in terms of the underlying data store. At run time, query methods typically execute in the native language of the underlying database management system. A container that uses a relational database for persistence might translate EJB QL statements into standard SQL 92 and an object-database container might translate the same EJB QL statements into an object query language.

Documenting Commercial J2EE Components

Documentation Benefits

a) Reduction in Pre/Post Sales Support

Documentation for components sold in the open market is particular important as 'face to face' interaction does not take place between author and customer. Providing a comprehensive set of documentation will ensure that pre/post sales support is kept to a minimum. Providing pre sales documentation i.e. a thorough component specification prevents many of the refund situations common in traditional 'box product' channels.

Traditional channels sell product by providing marketing information but not the finer detail covered in help files and other technical documentation. Providing information such as help files and evaluations enables customers to make an 'informed' purchase decision. Documenting and publishing known issues such as Frequently Asked Questions (FAQ's) on a regular basis will also help reduce technical support after the sale.

b) The Confidence Factor

Components sold on the open market may be 'Black Box' i.e. the source code is hidden. Because of this, trust is extremely important between customer and author. Therefore, provision of detailed product information such as evaluations, help files and white papers is essential for building confidence in potential customers.

Typical Documentation

What documentation should I provide? - The following section provides a detailed insight into the different types of documentation that should be provided when selling components in a commercial market. For examples of presenting online documentation in a concise and professional style browse our top selling products at: **a) Online Documentation (HTML, HLP and PDF Files)**

HTML is probably the best format of documentation you can provide and can be used for displaying information in text and graphical format. Typical examples include product overviews with screen shots and/or related diagrams. Customer can view HTML instantly as opposed to other document formats that must be downloaded first. Writing a help file is relatively easy and can be achieved using help authoring tools. More information on these tools can be found on our Web site: Help Authoring Tools.

Portable Data Files (PDF) are documents that can be viewed on IBM compatible or MAC platforms. The PDF file enables the creation of technical documentation in a 'book' format. Therefore, converting a published manual into an electronic form is probably the most efficient way to achieve this. The drawback with PDF files is the requirement of a proprietary viewer that must be downloaded first. To write a PDF file you will need to download the Adobe® PDF Writer.

b) Demonstrations

Developing a product demonstration can prove a valuable asset in the documentation you provide customers. Exposing component functions will help users understand the benefits of the product as a component-based solution. Demonstrations are compiled applications assembled with the component. They are not like evaluations that allow developers to use the component in a development environment. More information on evaluations is covered in the following topic.

The objective of a demonstration is to educate users on the functionality incorporated inside the component. The interface should demonstrate the main functions in a format that is understandable for all customers. Because of this it's important to remove industry jargon and acronyms that may confuse users. For data bound components, providing the option of entering a Data Source could be of benefit. This allows users to connect to internal data sources in their own organization and apply meaningful data in context with the component.

Demonstrations often reference dependencies and therefore testing the demonstration on a clean machine is extremely important. Clean systems contain freshly installed operating systems removing the potential hazards of previously loaded software. If your demonstration references any dependencies then you must create an installation kit. Sometimes it's beneficial to include the demonstrations within the evaluation kit and thus remove the need to write and maintain two separate kits.

Finally, the quality of a demonstration is directly correlated to the perceived quality of the final retail product. Where possible, design your demonstration in-line with an accepted standard. This helps build a perception of quality and trust with customers - remember demonstrations can make or break a sale.

c) Evaluations

Component authors recognize evaluations will help secure a product sale. Once a customer is happy with a specification they often trial the component to check the component will actually provide the functionality they are looking for. Customers do not doubt component based development, but may have concerns with an 'independent' solution. Because of this, component evaluations are essential. Unlike applications, component evaluations add value and play a significant role in the pre sales process.

Writing an evaluation will require consideration of security. Producing a component that displays a reminder screen or setting time limits hidden in cryptic keys within the registry are just some of the techniques currently used. Setting a 5-10 day trial period for technical components and 10-30 days for complex business components is recommended. This gives the customer enough time to evaluate the product and make a decision whether to buy.

An ideal evaluation is the full retail restricted by a security feature detailed above. This prevents users having to download the evaluation and retail component separately. ComponentSource has made available a license protection facility called C-LIC primarily designed to protect evaluations that can be unlocked into full retail products. C-LIC displays a reminder screen requesting the user to enter a license key provided when the full retail is purchased.

d) Sample Code

Sample code is particularly useful when developers need to prototype and assess component functionality. A good technique is to provide the sample code used in the component demonstration. If possible, this should be provided in a basic, intermediate and advanced version. This will allow the developer to grasp how the demonstration was developed and the stages of advancement throughout its development cycle.

Sample code usually is the final step that customers evaluate before making a decision whether to buy. Therefore its important to maintain a good perception by commenting all code and explaining exactly what happens and why. The quality of sample code will directly correlate to the quality of your final product. Because of this professionally written sample code using correct naming conventions, coding structures and error handling is essential. If the sample code is well structured then it can be reused in actual projects. This makes the whole process of integration far less complex and useful for developer's who need to rapidly assemble a component-based solution.

e) Readme Files

In this topic we list the various information that a Readme file should contain. Most installation scripts provide users with an opportunity to view a Readme file for last minute changes or errata information once installation is complete. These files should be written in a universal file format i.e. a text (TXT) file or HTML file. This prevents users having to own proprietary applications such as Microsoft Word to view the file. The following list provides an insight into the various information supplied in component Readme files.

Products Changes - this section is extremely important and should note all the functional changes that have been made in comparison to previous versions and any changes to documentation, installation etc.

Bug Fixing - bugs resolved from previous versions should be fully documented. Include the component version that contained the bug and a description of what has changed. This is particularly important if the component's interface has been changed.

System Requirements - Although compatibility information is supplied in our own sales documentation its worth reiterating this information in your Readme file. This should include information such as operating system for deployment, safety levels, threading standards etc.

Definitions of Component Filenames - Listing the filenames of all components (including dependencies) is particularly useful if the user is attempting to identify a problem. Although help and dependency files include this information, Readme files are often browsed as well.

Detailed Installation Notes - This should include information on how to de-install and update previous versions. A troubleshooting section should also be included defining solutions to common installation problems.

Notes on Sample Projects - Document any assumptions, known issues etc. If possible, describe each of the projects and the functions they expose. In addition to this defining a project's complexity i.e. basic, intermediate or advanced can also be of help.

Distribution Information - Particularly useful when a user creates an installation kit. Your component may reference many other dependencies, therefore detailing this information will help the developer create a tailored installation kit and prevent

many of the 'missing dependency' issues when testing.

Known Issues - You must document all known issues. If possible, also explain why the problem arises. If you do not provide this information then it's likely that unnecessary technical support issues will arise. Documenting known issues will demonstrate that you care and are focused on providing a future solution.

f) Prerequisites

Prerequisites provides the customer with details on required software, product size, required memory, service packs where appropriate and publicly available drivers such as reference implementations of JNDI drivers. It is worth including the minimum and recommended size when defining memory and hard disk allocation.

Component Testing

How do I test a component? - Thorough testing is paramount to the success of a component being accepted in the open market. All evaluations and sample code should be tested in addition to the full retail product for functionality, installation and de-installation. An issue that should be approached with care is the dependencies referenced by your component. Most installation tools require the selection of the original component's project file. This allows the wizard to analyze all references selected at the time the component was compiled. Absence of dependent files referenced by other dependent files is probably the most common installation issue. This is why testing on a clean machine, on all operating systems and all development environments is imperative. Therefore, to create a clean machine you must:

Format Hard Disk - If you only reinstall the operating system then static files that do not require registration may have already been installed. Therefore, without formatting the disk there is no guarantee that the installation will work on all machines.

Install Operating System - Make a note of any service packs applied as this must be included in the component's documentation i.e. the Readme file

Install Development Environment - Again, document any service pack installations. Always select the standard installation otherwise certain files may be missing causing erroneous errors when you test. This may include the development language for design time testing and the application server for deployment testing.

Once the above steps are complete you can image the disk allowing you to re-clean your environment in minutes. Image applications take a snapshot of your clean system, with operating system and development environment installed. This prevents the long cycle of re-installing everything before testing can re-commence. A good practice is to allocate a hard disk per operating system per development environment. As several disks can be installed in one machine, imaging an environment provides an efficient solution.

Test installation - Although we test the product installation thoroughly we recommend you also test the product to your best ability. This will ensure the swift progress of the component through our QA system.

Conclusion

Build components and enter the component market now!

Many different companies of various sizes from around the world are creating new EJB based components.

- BEA Systems - eCommerce and eBusiness EJBs
- Diamelle - eCommerce EJBs
- Eon Technologies - Financial & Banking EJBs
- Xenosys Corporation - Internet Investment, Billing and Banking EJBs

Customer demand for components is currently outstripping supply - as a result an opportunity exists for experts to create components and enter the "open market" for components.

If you have any feedback on this white paper or questions about creating commercial software components email us on: publishers@componentsource.com

Revision History:

First Published: June 5, 2000

Revised: November 1, 2000 (Updated Information specific to EJB 2.0 specification)
Revised: January 5, 2001 (Updated Information specific to Borland JBuilder)
Revised April 5, 2001 (Updated information specific to changes in EJB 2.0 specification)
Revised September 16, 2002 (Expanded to cover J2EE)

Contributions:
John Cheesman, ComponentSource
Shobana Narasimhan, Borland

ComponentSource Acknowledges the technical assistance of InferData in the preparation of this white paper.

About InferData
InferData Corporation is a leading provider of training, mentoring, and consulting services in advanced object-oriented technologies, and enterprise application development. Founded in 1993, our goal is to help clients achieve a sure-footed, successful transition to Object-Oriented technology. We offer a wide array of courses for all skill levels in Object-Oriented methodology, J2EE, Web Services Java, C++ and CORBA. These courses are offered at client sites worldwide. For more information, please call (888) 211-3421 or visit www.inferdata.com