



**Commercial Component Creation (EJB™ 2.0)  
Enterprise JavaBeans™ Components Technology  
Based Components**

**Technical White Paper**

**Date:** April 5, 2001

**Authors:**

**Faiz Arni, InferData Corporation**

**Andrew Pharoah, ComponentSource**

**[www.inferdata.com](http://www.inferdata.com)**

**[www.componentsource.com](http://www.componentsource.com)**

**Email:** [farni@inferdata.com](mailto:farni@inferdata.com)

**Email:** [authors@componentsource.com](mailto:authors@componentsource.com)

**InferData US Headquarters**

8200 North MoPac Expy.  
Ste. 200  
Austin, TX 78759  
USA

**Toll Free:** (888)211-3421  
**Tel:** (512) 306-8225  
**Fax:** (512) 306-8781

**Other Locations**

821 Bourbon Court  
Mountain View, CA 94041  
USA

**Toll Free:** (888)211-3421  
**Tel:** (512) 656-8360  
**Fax:** (512) 306-8781

# Contents

---

## **Introduction**

### **Commercial Overview**

---

## **Component Overview**

### **Identifying A Component Candidate**

[Analyze Application Functionality](#)

[Component Reusability](#)

[Expert Functionality](#)

### **Component Architectures**

[Client-Side Components](#)

[Server-Side Components](#)

[Exceptions](#)

### **Component Types**

[Visual Components](#)

[Non-Visual Components](#)

### **Implementation Language Issues**

[Java™ Technology](#)

---

## **Enterprise JavaBeans™ Components (EJB™)**

### **Introduction**

#### **Why use EJB?**

[Transaction Processing Monitors](#)

[CORBA](#)

[Component Transaction Monitors](#)

### **Architecture of Enterprise JavaBeans**

#### **Components**

[Basic EJB Concepts](#)

[Types of EJB](#)

[Creating Enterprise JavaBeans Components](#)

[Remote Interface](#)

[Home Interface](#)

[Enterprise Bean Implementation](#)

[Deployment Description](#)

[Error Handling](#)

[Threading](#)

## **Design Considerations**

[Identify Component Scope](#)

[Choose Architecture](#)

[Prototype Interface](#)

### **Session Beans**

[Stateless Session Beans](#)

[Stateful Session Beans](#)

### **Entity Beans**

[Container Managed Persistence](#)

[Bean Managed Persistence](#)

### **Roles in EJB**

[Bean Developer](#)

[Application Assembler](#)

[Application Deployer](#)

[Server Provider](#)

[Container Provider](#)

[Administrator](#)

### **EJB Services**

[Transaction Services](#)

[Security Service](#)

[Naming Service](#)

[Persistence Service](#)

[Resource Management Service](#)

---

## **Documenting Commercial EJB Components**

### **Documentation Benefits**

[Reduction In Pre/Post Sales Support](#)

[The Confidence Factor](#)

### **Typical Documentation**

[Online Documentation](#)

[Demonstrations](#)

[Evaluations](#)

[Sample Code](#)

[Readme Files](#)

[Pre-requisites](#)

[Dependencies](#)

[Compatibility](#)

### **Component Testing**

### **Component Licensing**

[The Common Licensing Problem](#)

[C-LIC - The Common Licensing Solution](#)

---

## **Conclusion**

---

# Introduction

This white paper has been constructed to help component authors develop and enhance professional software components for server applications and for delivery on the 'open market'. 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® Enterprise JavaBeans™ (EJB™ 2.0) 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 EJB architecture and the environment in which these components can be used.

---

## Commercial Overview

The market for Software Components is expected to grow to around \$4.4 billion by 2002, \$1.0 billion from products and \$3.4 billion from related services. (Source: PricewaterhouseCoopers)

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](http://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 EJB architecture and how to design, implement and deploy EJB 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/Build/OpenMarket.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 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™ and webGAIN VisualCafé™. 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/IOP* as the protocol of communication between clients and enterprise beans. Using *RMI/IOP* enables the clients to use programming languages such as C++, C, Cobol and Java technology. *RMI* refers to Java's *Remote Method Invocation*. *IOP* refers to the *Internet Inter-Orb Protocol*. *IOP*, 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 *IOP*.

### **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.

---

## **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.

## **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 amounts of 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 vendor exposes the services in a proprietary way, the enterprise application becomes non-portable.

## **CORBA™**

At this point it might be appropriate to present the other technology that attempts to address the shortcomings of the TP Monitors. And that technology is CORBA and is specified by the OMG and is a industry standard. CORBA provides a rich 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 implementations of ORBS (Object Request Brokers).

## 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 container-specific deployment tool generates the appropriate proxy 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 focus primarily on the design and implementation of 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.

### 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 (or delegates) 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 some sort of a *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 *EJBO* object 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 *EJBO* object and c) for obtaining the return value from the call to the *EJBO* object and sending that value, in a serialized fashion, possibly over the network, to the caller. The caller is invariably a *Client-Proxy*. The *EJBO* object, 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 *EJBO* object 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 <i>EJBO</i> object and marshals the return value back to the ClientProxy.
<i>EJBO</i> object	EJB Compiler	Server	Intercepts all method invocations on the enterprise bean and provides the EJB services. Delegates call to the enterprise bean.
Enterprise Bean	Bean Developer	Server	Implements the real business logic of the application for the methods advertised in the remote interface.

## Types of Enterprise JavaBeans Components

Enterprise JavaBeans are divided into three 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.

## Creating Enterprise JavaBeans Components

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

1. Remote interface (*Entity and Session Beans only*)
2. Home interface (*Entity and Session Beans only*)
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 may be running a process. Therefore any error that occurs should be handled by the client and interpreted in such a way that any error message displayed is generated by the client and is in context with the process that has failed. 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 the dependencies your component references. Because of this, you must not raise these errors to your client application.

## 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. Bv

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 implication will allow authors to publish highly focused discrete components and provide customers 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 scaleable 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

### **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 session 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, say `createAccount`, to create a new `Account` bean on the server.

### 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

### 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.

### **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 to ensure data synchronization. BMP beans are relatively difficult to develop and harder to maintain, yet they are extremely flexible.

## **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 callback 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

### **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 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.

### **Application Assembler**

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

### **Application Deployer**

An Application Deployer is typically an IT manager who deploys, after modifying the deployment description file, 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.

### **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™.

### **Container Provider**

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

### **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. The most important of which are the transaction and security services. The rest of the services are listed below.

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

### **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<sup>1</sup>. 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.

Attribute	Description
<b>Not Supported</b>	When a caller invokes a method, the caller's transaction, if any, is suspended and is resumed after the method call.
<b>Required</b>	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.
<b>Supports</b>	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.
<b>RequiresNew</b>	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.
<b>Mandatory</b>	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 <code>javax.transaction.TransactionRequired</code> exception.
<b>Never</b>	If the caller is in a transactional context, the container throws a <code>java.rmi.RemoteException</code> . If the caller is not in a transactional context, the container does not start a new transaction.

## 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

---

<sup>1</sup> The bean-managed transaction attribute is specified at a bean scope (not method). All the methods of such beans use the bean-managed transaction attribute.

at run time, thus enabling the bean developer to implement object level security.

### **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.

### **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.

### **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.

---

## **Documenting Commercial EJB 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: <http://www.componentsource.com/java>

### **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 under "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 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 into 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 it's 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) Pre-requisites

Pre-requisites provides the customer with details on required software, product size, required memory, service packs where appropriate and publicly available drivers such as the 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. If this rigorous testing process is not followed then the likelihood of damaging a customers system is high. 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.

**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.

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 economical and effective solution.

## Component Licensing

*How do I license my component?* - Unlike application licensing, few licensing utilities exist for component licensing. Those that currently exist often operate a 'two-phase unlock' process which requires manual intervention to generate the retail unlock key. However, this form of licensing does not suit 'mass market' adoption. The availability of the C-LIC (Common Licensing) component enables authors to provide a 'Try-Before-You-Buy' licensing solution.

### a) The Common Licensing Problem

Components sold on the open market are typically 'Black Box' architecture. This means that all functionality is encapsulated and cannot be adapted by the developer except through the public interface. Because of this, providing an evaluation that allows the developer to 'road test' a component is important when securing a sale. Nowadays, customers expect one download that runs in evaluation mode for a set number of days. Once this evaluation has expired, functionality is disabled until a license key is purchased and entered, unlocking the component into a full retail version. Often the best form of protection is to use a reminder/nag screen that launches each time the calling application runs the component. This prevents users without a license from releasing an application into a commercial environment.

## **b) C-LIC - The Common Licensing Solution**

C-LIC is the ComponentSource license technology used to adapt a full retail product into an evaluation. However, please note the current version does not support copy protection. C-LIC can be integrated into a majority of languages that support the creation of software components. Its method of working is similar to that used in application software. C-LIC was made available to enable component authors to create a fully or part functioning evaluation protected by a 'nag' screen reminding users that the component is unlicensed. The nag screen allows customers to browse to the relevant product page and purchase the license key used to unlock the product into the full retail component. The license key is provided by ComponentSource and is generated by our own proprietary encryption.

C-LIC can also protect different levels of functionality. For example if your standard version has 10 functions and your professional version 20 functions then the purchase of a standard license will unlock 10 functions only - the other 10 functions will remain in evaluation mode. Please Note: C-LIC V1.1 does not provide "copy protection".

---

## **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 would like to find more information on building components visit:

<http://www.componentsource.com/build/>

<http://www.componentsource.com/business/>

If you have any feedback on this white paper or questions about creating commercial software components email us on: [authors@componentsource.com](mailto:authors@componentsource.com)

**ComponentSource**