JavaBeans vs Enterprise JavaBeans

JavaBeans has been at the center of many new paradigms and technologies that have emerged since its inception. Among emerging technologies, Enterprise JavaBeans has generated tremendous interest in the business computing community. However, a common misconception is that an Enterprise JavaBean is an extension of a "plain vanilla" JavaBean with enterprise functionality.

While both JavaBeans and Enterprise JavaBeans are software component models, their purpose is different. A JavaBean is a general-purpose component model, whereas EJB, as the name suggests, is a component model that is enterprise specific. Even though these models have entirely different architectures, they adhere to certain underlying principles that generally govern a software component model. We'll use these principles and the basic characteristics of software components to compare JavaBeans and Enterprise JavaBeans.

Goals
The underlying theme of both JavaBeans and Enterprise JavaBeans is, as Sun puts it, "Write once, run anywhere" (WORA). Accordingly, the primary objective of both models is to ensure portability, reusability and interoperability of Java software components.

JB
JavaBeans takes a low-level approach to developing reusable software components that can be used for building different types of Java applications (applets, stand-alone apps, etc.) in any area. Whether you're developing a simple applet or a complicated application, JavaBeans can be integrated into your system with ease. Already a large number of vendors offer JavaBeans in a variety of fields. Some JavaBeans may be common to multiple fields. A chart bean, for instance, can be used in scientific, engineering and business computing applications.

EJB
Enterprise JavaBeans takes a high-level approach to building distributed systems. It frees the application developer to concentrate on programming only the business logic while removing the need to write all the "plumbing" code that's required in any enterprise application. For example, the enterprise developer no longer needs to write code that handles transactional behavior, security, connection pooling, networking or threading. The architecture delegates this task to the server vendor.

What Are Beans?
JB
A JavaBean - a bean - is a reusable software component that can be visually manipulated in a builder tool. Builder tools help you assemble applications by visually connecting beans. This doesn't mean you can't build applications in the conventional way. If you prefer, you can hand-code applications by using beans. When you build applications by visual connection, no coding is involved (though in semivisual tools you may have to write some code).

Builder tools also help you customize beans visually. Once customized, you can save beans as serialized prototypes (.ser files). When beans are part of an application, they run just like any other object. They are instantiated differently, though, as saved beans have to be resurrected from their serialized prototypes.

EJB
An Enterprise JavaBean - an EJB - is a reusable server-side software component. Enterprise JavaBeans facilitate the development of distributed Java applications, providing an object-oriented transactional

environment for building distributed, multitier enterprise components. An EJB is a remote object, which needs the services of an EJB container in which to execute.

Architecture

JB

The JavaBean specs define a component model to build, customize, assemble and deploy general-purpose Java software components. In the JavaBean model, the structure and behavior of a bean is described by three basic features: properties, methods and events.

Properties are a bean's named attributes that can be edited to customize a bean. *Methods* describe a bean's behavior. *Events* serve two purposes:

1. ***Bean connection:*** When a bean is running in a builder tool, events enable visual connection.
2. ***Notification:*** When a bean is running in an application, events notify occurrences and pass data from source to target.

When a bean is inserted in a visual builder tool, its exposed properties, methods and events are discovered through the twofold process called *introspection,* which involves:

1. Discovery from the explicit information, which is provided by the bean provider through a bean-specific class called BeanInfo.
2. Automatic discovery of a bean's features by using the reflection API. To facilitate this, methods in the bean have to adhere to certain naming conventions as stated in the JavaBeans specs.

The JavaBeans 1.0 specification dealt mainly with the design-time behavior of JavaBeans. It was later augmented by a set of three specifications code-named "Glasgow" to address some runtime issues. The "Extensible Runtime Containment and Services Protocol" spec, which is part of Glasgow, describes the relationship between a bean and its container at runtime. As shown in Figure 1, a bean that conforms to this spec can be part of a nested containment structure and can utilize arbitrary services from its container.

EJB

The Enterprise JavaBeans spec defines a server component model and specifies how to create server-side, scalable, transactional, multiuser and secure enterprise-level components. Most important, EJBs can be deployed on top of existing transaction processing systems including traditional transaction processing monitors, Web, database and application servers.

As shown in Figure 2, a typical EJB architecture consists of:

- An EJB server
- EJB containers that run on these servers
- Home objects, remote EJB objects and Enterprise beans that run in these containers
- EJB clients
- Other auxiliary systems like the Java Naming and Directory Interface (JNDI), the Java Transaction Service (JTS) and Security services

Unlike JavaBeans that use introspection, the EJB container uses the EJBMetaData class to query an EJB for its metadata at any given time.

Some of the advantages of pursuing an EJB solution are:

- EJB gives developers architectural independence.
- EJB is WORA for server-side components.
- EJB establishes roles for application development.
- EJB takes care of transaction management.
- EJB provides distributed transaction support.

- EJB helps create portable and scalable solutions.
- EJB integrates seamlessly with CORBA.
- EJB provides for vendor-specific enhancements.

Application Development Roles

JB

Even though the JavaBeans specification doesn't mention application development roles explicitly, we can infer the following two roles:

1. ***Bean provider:*** develops, customizes and packages beans
2. ***Bean user:*** customizes, assembles and deploys beans

EJB

The EJB specification assigns specific roles for project participants charged with enterprise application development utilizing EJBs. For instance, business developers can focus on writing code that implements business logic. Deployers of EJB can take care of installation issues in a simple and portable fashion. The server vendor can take care of providing support for complex system services and make available an organized framework for an EJB to execute in, without assistance from EJB developers. The EJB specification defines six primary roles:

1. ***Enterprise Bean provider:*** develops the enterprise bean
2. ***Application assembler:*** connects the beans together and packages them
3. ***Deployer:*** deploys the packaged bean on the server
4. ***EJB server provider:*** provides a framework that can execute the EJB containers with transactional support
5. ***EJB container provider:*** provides tools to generate the container classes that encapsulate the bean at runtime
6. ***System administrator:*** responsible for configuration and administration of the environment on which the EJB server executes

APIs

Both the JavaBean and the EJB specs define APIs for bean development, execution and deployment.

JB

- ***java.beans:*** The APIs from the original JavaBeans specification are implemented in the java.beans package, which includes classes and interfaces needed for both bean providers and visual builder tools.
- ***java.beans.beancontext:*** The APIs from "The Extensible Runtime Containment and Services Protocol" spec are implemented in the java.beans.beancontext package, which includes classes and interfaces for implementing bean context services, bean contexts and bean context children. This package is available only in Java 2.

Both are core Java packages and are therefore available in different vendor implementations of Java.

EJB

- ***javax.ejb:*** The APIs defined in the Enterprise JavaBeans specs are included in the javax.ejb package, which is a Java standard extension. In addition to this, EJB relies on the APIs defined for the Java Transaction API (JTA), the Java Transaction Service (JTS) and the Java Naming and Directory Interface (JNDI).

Bean Characteristics

Structure and Behavior

JB

A bean, which is identified by a class, can encapsulate any functionality. While the bean class need not extend

any other class, it needs to implement the serializable or externalizable interface either directly or through inheritance. Even though a single class identifies a bean, its functionality can be spread over many classes through inheritance and delegation. An important restriction on the bean class is that it should have a no-argument constructor.

Since a bean's internal structure and implementation details aren't exposed, you can't extend a bean's functionality as is. You can, however, customize it to suit your application. To extend a bean's functionality, you need to create a new bean by extending the existing bean class. In other words, you need to use the bean as a class library.

As we alluded to in the Architecture section, a bean's interface to the outside world (which includes visual builder tools, other beans and applications) is through its properties, methods and events. This interface is at the bytecode level, which means there is no need to recompile the bean when it's assembled in an application.

For the properties, methods and events to be discovered through introspection, the related methods have to follow certain signatures and naming conventions specified in the JavaBeans specs. In addition, a bean provider can also furnish feature descriptors through a design-time-only class called BeanInfo, which is specific to a bean class.

EJB

An EJB typically encapsulates business logic that operates on data. An EJB's interface to the outside world is through its Home and Remote interfaces. While the Home interface defines a factory to create new beans and find existing beans, the Remote interface defines the business methods that the bean supports. Each packaged EJB is identified by its Home interface and its Home object, its Remote interface and its EJB object, the enterprise bean class implementation and its deployment descriptors.

In EJB there's no need for a BeanInfo class because the deployment descriptors in conjunction with the EJBMetaData class take care of the bean description.

As mentioned earlier, an EJB isn't represented by a single class, but by the enterprise bean implementation, its home interface and its remote interface.

A typical Home interface for a hypothetical ParentEJBean would look like this:

```
import javax.ejb.*;
import java.rmi.*;

public interface ParentHome extends EJBHome {
Parent create(char relationship, String name)
throws CreateException, RemoteException;
Parent findByPrimaryKey(ParentPK parentKey)
throws FinderException, RemoteException;
}
```

You can't inherit the Home interface because of a problem with the create() methods; that is, the child will need to supply the same create() methods as the parent, but the methods will return different values (or remote interfaces). Unfortunately, the Java language doesn't permit a class to have two methods that differ in signature only by return type, so inheriting from Home interfaces is out of the question.

You can, however, inherit implementations. You could have a ParentEJBean implementation class and declare a ChildEJBean as:

```
public class ChildEJBean extends ParentEJBean {

}
```

You can thus reuse implementation code. But with this approach you'll run into the differing-only-by-return-type problem in the ejbCreate() methods of bean-managed persistence (see the sections on Types and Persistence). This is because the methods will take the same arguments, but will return different primary keys. Since create() methods in Session beans return a void, they don't encounter this problem. Container-managed EJB's ejbCreate() methods also return a void. Even though they compile, there may be problems when the EJB container generates code that actually returns the primary key.

Inheritance in EJB is tricky, and you're better off using containment instead.

Visibility

JB

A bean can be visible, invisible or both. A Stopwatch bean, for instance, can have the GUI shown when it runs on the client side and turned off if it's running on the server side. Even if a bean is invisible, it can be customized, serialized and connected to other beans in visual builder tools.

EJB

An EJB is a nonvisual remote object that resides only on the server side.

Types

JB

Beans that conform to JavaBeans 1.0 specs aren't typed. The Glasgow spec, however, allows two types of beans: Simple and Participant. A Simple bean isn't aware of its container, whereas a Participant bean actively participates in its container. A bean that conforms only to the original JavaBeans 1.0 specs falls under the Simple bean category. A Participant bean, however, conforms to "Extensible Runtime Containment and Services Protocol" specs in addition to the original bean specs. A Participant bean can also discover and utilize arbitrary services from its container.

Table 1 gives a comparison of Simple and Participant beans.

EJB

There are two primary types of EJBs: Session and Entity beans. While an Entity bean has a unique identity defined by its primary key class, a Session bean has no unique identity. Multiple clients can thus share an Entity bean. A Session bean, on the other hand, is created, used and destroyed by the client that created it. Each bean has associated with it a context object (Session context or Entity context) for its lifetime.

Table 2 compares Session and Entity beans.

There are two types of Session beans: stateful and stateless. Similarly, there are two types of Entity beans: container-managed persistent entities and bean-managed persistent entities. (See the following section for more details.)

Persistence

JB

Beans are persistent. In the context of JavaBeans, beans are objects that can be saved and resurrected. Persistence is achieved by saving a bean's internal states through serialization. As mentioned before, a resurrected serialized prototype of a bean can be included in an application.

EJB

Stateful Session beans may have internal states. Therefore, they need to handle activation and passivation. Passivation is the process by which the state of a bean is serialized out into secondary storage. Activation is the process by which it is deserialized from secondary storage. These types of EJBs can be serialized and restored across client sessions. To serialize, a call to the bean's getHandle() method returns a handle object. To restore, a call to the handle object's getEJBObject() method is used to return a bean reference.

Entity beans are inherently persistent beans. There are two types of persistence in Entity Beans:

- ***Bean-managed persistence:*** In BMP the Entity bean is directly responsible for saving its own state. The container doesn't need to generate any database calls. Hence, the programmer needs to hard-code persistence into the bean through explicit JDBC or embedded SQL calls.
- ***Container-managed persistence:*** In CMP the EJB container is responsible for saving the bean's state. Since it's container managed, the implementation is independent of the data source. The container-managed fields need to be specified in the deployment descriptor and the EJB container automatically handles persistence.

Customization

JB

Beans are visually customizable. You can customize a bean by editing its properties. Visual builder tools typically present property sheets for this purpose. To generate property sheets, builder tools use introspection. The JavaBeans spec also provides an alternative to property sheets. It specifies an interface called Customizer to enable bean providers to build bean-specific customizers. Such a customizer can also be invoked at runtime. (See L. Rodrigues's article, "On JavaBeans Customization," ***JDJ*** Vol. 4, issue 5, for more details.)

EJB

EJB customization is a bit different from JavaBean customization. There's no concept of a property sheet or a custom-written customizer for an EJB. EJBs are customized using deployment descriptors, which define the contract between the ejb-jar provider and the EJB consumer. It captures the declarative information (information not included directly in the EJB code) that's intended for the consumer of the ejb-jar file.

The two types of information in the deployment descriptors are the EJB's structural information and application assembly information. In EJB 1.1 XML is used to define the deployment descriptors. EJB vendors may provide tools that can be used by the ejb-jar provider to create deployment descriptors.

Containment and Nesting

JB

A bean can contain another bean. The original JavaBeans 1.0 specs didn't explicitly address containment-related issues. The Glasgow specification defines the notion of a logical bean container or BeanContext (see Figure 1). A child bean in a container can itself be a BeanContext, thus allowing nesting of beans. In a BeanContext child beans (Simple and Participant) can be dynamically added and removed. They can also access arbitrary services from the container.

EJB

EJBs always run within an EJB container. EJBs request different services from their containers and are aware of their environment. Containers can't contain other containers and therefore there's no concept of nesting in EJBs. Each EJB is associated with a context object (either a SessionContext or an EntityContext that provides information about the EJB). The context object is the component's handle on the container, through which the component can get transaction information, security information and information from the component's deployment descriptor. The EJB component calls into the Context object through the SessionContext or EntityContext interface.

Packaging and Deployment

JB

Beans are packaged in JAR files. The bean provider has to provide a manifest file with JavaBeans-related attributes in order to identify the bean class. A JAR file can hold more than one bean. However, the JAR entry for each bean class should have the Java-Bean attribute set to true. An example:

Name: Spreadsheet.class Java-Bean: True

Two more JavaBean-related attributes are Depends-On and Design-Time-Only. A typical bean JAR file contains design-time and runtime bean classes, documentation, resources such as images, and sound files.

EJB
EJBs are also packaged in JAR files. To identify the EJB class, the bean provider has to provide a manifest file in which the jar-entry for the EJB class should have the Enterprise-JavaBean attribute set to true. An example:

Name: ~gopalan/BankAccountDeployment.ser Enterprise-Bean: True

Application Assembly
JB
You can compose applications by visually connecting beans in a builder tool or manually by writing connection programs. The application so developed can be an applet or a stand-alone application. Beans for an application need not come from the same vendor because beans can be developed independent of one another.

As we alluded to before, events act as interfaces between beans. Bean connections are performed at design time and are unidirectional. The source bean fires an event and the target bean receives it. When the application assembler chooses a source bean for connection, the builder tool discovers through introspection the events fired by that bean. When the application assembler chooses a target bean for the selected source bean, the builder tool discovers the compatible methods in the target bean, again through introspection.

EJB
EJBs are assembled into larger deployable applications. The input of application assembly is one or more ejb-jar files produced by different providers. The output is one or more ejb-jar files that contain Enterprise beans with their assembly instructions. As we mentioned in the Customization section, the application assembly instructions have been inserted into the deployment descriptors. EJBs too can be developed independent of one another. Once an EJB's home and remote interfaces are known, you can use them to create or find them and to invoke methods on them.

Execution
JB
The execution phase consists of the instantiation and running of beans.

Instantiation
Even though a bean is an object, it is instantiated differently. Instead of the new operation, beans are instantiated using the Beans.intantiate() method. There are many flavors of this method. A bean instantiation example:

```
// Obtain the Class Loader
ClassLoader loader =(Account.class).getClassLoader ();

// Instantiate the Account Bean
Account account = (Account)Beans.instantiate (loader, ÒAccount-)
```

Running
As mentioned earlier, a bean is like any other object when it's running in an application. Normally, methods in a bean aren't directly invoked from other beans. The bean connections determine what methods need to be invoked. Event adapters enable indirect method invocation.

Beans can also be executed in a builder tool at design time. As mentioned before, an important requirement for a bean to run in a builder tool is for the bean class to have a constructor with no arguments. This is because the builder tool can't provide the constructor parameters. When there are many constructors, it can't decide which one to use.

Java 2 has new Beans.instantiate() methods to facilitate the instantiation of applets and BeanContexts.

EJB
The EJB execution phase consists of locating, instantiating and running.

Locating the EJB
EJB clients locate the specific EJB container that contains the enterprise Bean through the JNDI. They then make use of the EJB container to invoke bean methods. As you may be aware, JNDI allows multiple directory services to coexist and even cooperate within the same JNDI client. Using JNDI, a user can navigate across several directory and naming services while seeming to work with only one logical federated naming service.

Instantiating the EJBean
Once EJB clients obtain a reference to the Home object, they can create the EJB by calling its create() method or find EJBs by calling its find methods. This creates the EJBObject and the EJB component inside the EJB container.

Invoking Methods on the EJBean
The EJB client can now use the remote object reference to invoke methods on the EJBean by invoking its remote methods, which form the business logic of the component. For example:

```
// get the JNDI naming context
Context initialCtx = new InitialContext ();

// use the context to lookup the EJB Home interface
AccountHome home=(AccountHome)initialCtx.lookup ("com/gopalan/Account");

// use the Home Interface to create a Session Bean object
Account account = home.create (1234, "Athul", 1000671.54d);

// invoke business methods account.credit (1000001.55d);
```

Transactions
JB
There is no explicit transactional support.

EJB

- ***Declarative transaction management:*** The EJB container vendor is required to provide transaction control. The EJB developer who is writing the business functionality needn't worry about starting and terminating transactions. However, for maximum flexibility, the EJB spec provides for declarative transaction management. Six declarative modes can be specified by the deployer: TX_NOT_SUPPORTED, TX_BEAN_MANAGED, TX_REQUIRED, TX_SUPPORTS, TX_REQUIRES_NEW, TX_MANDATORY.
- ***Distributed transactional support:*** EJB provides transparency for distributed transactions. This means that a client can begin a transaction and then invoke methods on EJBs present within two different servers running on different machines, platforms or JVMs. Methods in one EJB can call methods in the other EJB with the assurance that they'll execute in the same transaction context.

Security Services
JB
There are no special security APIs for JavaBeans.

EJB
EJB provides authorization using the Java security model. EJB server implementations may choose to use connection-based authentication in which the client program establishes a connection to the EJB server. The client's identity is attached to the connection at connection establishment time. The EJB/CORBA mapping specifies that the CORBA principal propagation mechanism be used. This means that the client ORB adds the client's principal to each client request. The communication mechanism between the client and the server propagates the client's identity to the server. Security in EJB 1.1 is declaratively defined in the deployment descriptors and is role based.

Interoperability

JB

JavaBeans can interact with components built using other models, which includes the widely used Component Object Model (COM). Using The Beans-ActiveX Bridge, a bean can be converted to an ActiveX control. A converted ActiveX control can interoperate with other ActiveX controls in an ActiveX container.

EJB

While The EJB spec allows the EJB server vendors to use any communication protocol between the client and the server, The EJB/CORBA mapping document is prescriptive with respect to what goes on the wire. This allows both system-level and application -level interoperability between products from vendors who choose to implement the EJB/CORBA protocol as the underlying communication protocol.

Java clients will optionally communicate with server components using IIOP. They'll have a choice of API's - either the Java RMI or the Java mapping of the CORBA IDL interface. Non-Java clients communicate with server components using IIOP and the appropriate language mapping. Clients wishing to use the COM+ protocol communicate with the server component through a COM-CORBA bridge. Also realize that the client of an EJB can itself be a server component (for instance, Java Server Pages or a servlet), so an HTTP-only Web client can use a servlet to make EJB invocations.

An EJB can't be deployed as an ACtiveX control because those controls are intended to run at the desktop and EJB's are server-side components. CORBA-IIOP compatibility via the EJB-to-CORBA mapping is defined by the OMG. However, EJB components may be able to communicate with DCOM servers using a DCOM-CORBA bridge.

Summary

Table 3 summarizes the characteristics of JavaBeans and Enterprise JavaBeans.

Conclusion

JavaBeans technology, now in its third year, has undergone the acid test of the industry. The original spec has gone through a couple of iterations since its introduction. Although the spec may undergo further iterations, many aspects of JavaBeans have been firmed up.

On the other hand, the EJB spec is just more than a year old and still evolving. Now at version 1.1, it provides an excellent architectural foundation for building distributed enterprise-level business object systems. Some areas in the spec need to be examined closely, however - most notably in the EJB model for handling persistent objects. Standardizing the contract between development tools and systems to provide a uniform debugging interface for all development environments is being considered as well. The specification will still go though more iterations before becoming final.

The other issue is compatibility. There are two areas where compatibility is an issue. One is what actually constitutes an "EJB-compatible" server. The other is guaranteeing that EJBs developed on servers from different vendors can interoperate.

References

1. Cable, L. (1997). Glasgow Specification Version 99A. Sun Microsystems, July.
2. Enterprise JavaBeans Specification 1.1. Sun Microsystems, June 1999.
3. Hamilton, G., ed. (1997). JavaBeans Specification, Version 1.02. Sun Microsystems, July.
4. Rodrigues, L. (1997). "Java, The Next Generation: JavaBeans." *Java Developer's Journal*, Vol. 2, issue 1, January.
5. —(1998). *The Awesome Power of JavaBeans*. Manning.
6. Seshadri, G., and Raj, G.S. (1999). *Enterprise Java Computing - Applications and Architecture*. SIGS/Cambridge University Press.