Using Enterprise JavaBeans in a Computer Science Curriculum

Rodney S. Tosten, rtosten@gettysburg.edu
Jeannie R. Albrecht, s412270@gettysburg.edu
Christyann Ferraro, s384898@gettysburg.edu
Gettysburg College
Gettysburg, PA 17325
(717) 337-6630

ABSTRACT

This paper introduces the principles behind Enterprise JavaBeans including examples of both session and entity beans. It also discusses techniques for incorporating Enterprise JavaBeans into a computer science curriculum specifically within a distributed processing course and database course. The paper ends with an evaluation and summary of Enterprise JavaBeans.

Keywords: *deploy, Enterprise JavaBeans, transactions, database, Distributed processing*

1. INTRODUCTION

In the early 1990's, traditional enterprise information system providers began shifting from two-tier client server application models to multi-tier models. The new models separated business logic from system services and user interfaces, placing a new middle tier between the two. The development of these middleware services, combined with the growing popularity of the Internet, has created a demand for simple, portable, easy to deploy applications.

Until recently, the task of creating these middle business applications was fairly difficult. There were two basic types of services from which to choose. The first service was Transaction Processing Monitors. The purpose of a Transaction Processing Monitor was essentially to manage and oversee all actions. They were often responsible for performing complex tasks. One problem with Transaction Processing Monitors was that they were not object oriented and had no sense of identity. They worked solely with procedural code. The second service was Object Request Brokers. The goal of these was to establish the communication backbones that were used to interact with unique objects [1]. They did have a sense of identity, however they left concurrency, transactions, and fault tolerance up to the developer. Because programmers had to choose one of these two services, programming a stable application often became very complicated. Enterprise JavaBeans (EJB) combine ideas from both Transaction Processing Monitors and Object Request Brokers, offering a simple and platform independent solution to programmers' problems.

Enterprise based technologies, like EJB, will play a large role in the future of computing, and therefore also in computer science education. EJB incorporates concepts taught in both Database Systems courses and Parallel and Distributed Processing courses. Thus, we believe that EJB should be featured in a computer science curriculum.

2. WHAT ARE EJB?

The Enterprise JavaBeans (EJB) technology provides an adaptable architecture for distributed business objects that can automatically manage transactions, object distribution, concurrency, security, and persistence. This technology essentially combines the fundamental concepts of Transaction Processing Monitors and Object Request Brokers. By using EJB, programmers can create portable, customizable, platform independent applications to control database access and operations.

There are two main categories of EJB: session beans and entity beans. The development and functionality of the two are

very different. Entity beans are persistent objects, while session beans are transient. Generally, entity beans model business concepts that can be expressed as nouns. They represent the state and behavior of real world objects, and are persistent records in a database. In fact, each row in a database can be viewed as an entity bean. Session beans, on the other hand, are responsible for executing processes or completing tasks. They model activities that are fundamentally transient, and do not represent anything in a database. Session beans are frequently used to describe interactions or implement tasks for a group of entity beans.

There are two different types of session beans that can be used when creating an application. Stateless session beans are the simplest type of bean. They have no knowledge of past method calls or requests, and they do not maintain a conversational state with the client. Due to their non-persistence and non-dedicated properties, few server resources are required, making them very efficient. Basically any activity that can be performed in one method call is a good candidate to be a stateless session bean. The other type of session bean is a stateful session bean. These beans offer an alternative somewhere between entity beans and stateless session beans. They still do not represent any information in a database, however they are dedicated to one client for the life of the bean instance. They maintain a conversational state with their client, causing them to require more server resources.

Like session beans, entity beans are also divided into two subcategories. The first and less complex type of entity bean is the container-managed bean. With container-managed beans, the container automatically regulates the coordination of data represented by a bean instance with the database. Container-managed beans are very flexible and reusable, however they require more sophisticated mapping tools to access the database. This generally means that extra steps are required when setting up the bean within the deployment tool. The other type of entity bean is the bean-managed bean. These beans force the developer to handle the inserting, updating, and deleting of data explicitly. Programmers must supply code for database manipulation, so there is no need for the sophisticated mapping tools.

3. DEVELOPING EJB

The basic structure of an Enterprise JavaBean is the same regardless of the type of bean being programmed. Two interfaces and one class are needed to define EJB. All three of these applications reside within an EJB container. Containers are abstract entities that are responsible for managing beans and providing an environment in which enterprise beans can run [2]. The interfaces present the methods of the bean to clients outside of the container, while the class contains the Java code for these methods.

The first interface is the remote interface. The bean's business methods that are present to the outside world are listed here. The remote interface always extends *javax.ejb.EJBObject*. The exact same methods that are listed in the remote interface will be found in the bean class. For example, if there is a method *getName( )* in the remote interface, an identical method *getName( )* must be in the bean class as well.

The second interface needed is the home interface. The bean's life cycle methods, such as it's methods for creating, removing, and finding beans, are defined in the home interface. It always extends *javax.ejb.EJBHome*. The methods found in the home interface correspond to those in the bean class, however they do not have exactly the same name. For example, the home interface always defines a *create( )* method. Instead of a *create( )* method in the bean class, there is a method called *ejbCreate( )*. They are not identical, but the correlation is obvious.

The class file that must be created defines the algorithms for the business operations. This class implements either *javax.ejb.SessionBean* or *javax.ejb.EntityBean* depending on the type of bean being built. The bean class contains the coding for all of the methods of the bean. It contains business methods that match those found in the remote interface, and life cycle methods that correspond to those found in the home interface.

Once all three of these programs are written and compiled with Java's *javac* compiler, the bean can be deployed. When a bean is deployed it is added to an EJB container so that it can be accessed as a distributed component [1]. There are several different applications available for the deployment of EJB. Starting with the release of Java 2 Enterprise Edition (J2EE), an EJB deployment tool (Figure 1) is part of the basic download. A J2EE server and Cloudscape database server are also included, so it is possible to develop EJB without any other software packages. Each deployment tool is slightly different, and in some cases it is necessary to create a primary key class for entity beans. Some packages may require the developer to create a deployment descriptor before the bean can successfully be deployed. With J2EE's tool, however, neither of these is needed. They are created during deployment.

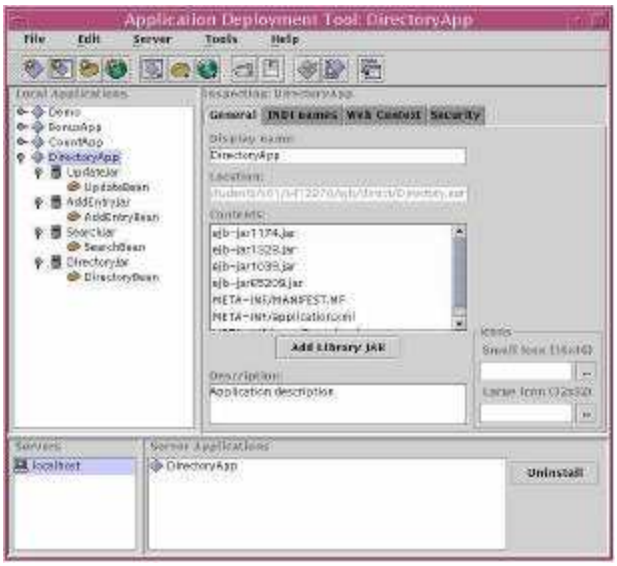

Figure 1. – J2EE Deployment Tool

In addition to the interfaces and the class, a client program is needed to reference the EJB. The client is separate from the bean, and would not be included in the EJB container. When a client wants to access a bean, it first locates the bean's home interface via the Java Naming and Directory Interface (JNDI). The home interface creates an instance of a bean, and returns the instance to the client. The client is then free to access the business methods of the bean via the remote interface (Figure 2). When the client is finished and wants to delete the bean, it notifies the home interface again, and the bean instance is removed.

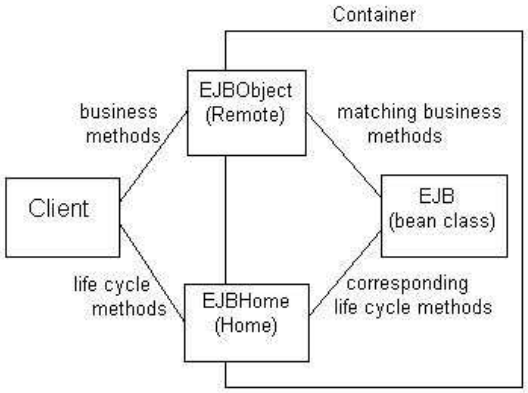

Figure 2. – EJB Overview

4. EJB IN THE CLASSROOM

EJB can be integrated into two courses in a computer science curriculum: Database Systems and Parallel and Distributed Processing. Both of these courses have the same prerequisite, namely Data Structures. Knowledge of both database systems and distributed processing are needed to fully understand the theory behind EJB, so there will be some overlap between the two classes.

At Gettysburg College, Parallel and Distributed Processing covers both parallel and distributed systems and architectures. In the beginning of the course, students work on SIMD and numerical parallel processing. The SIMD based programming language Parallaxis is used to practice developing programs like sorting algorithms. This helps to familiarize students with the theory of parallel processing. The second part of the course focuses on MIMD and distributed processing. Java Threads and RMI (Remote Method Invocation) are used to introduce students to these topics.

This second section is also the part of the course where EJB are discussed. EJB are a direct extension and application of JavaRMI. RMI is used in the development and deployment of EJB. The stubs and skeletons needed are

actually generated by the *rmic* compiler, although the deployment tool runs the command automatically during the deployment of the bean. EJB communicate using RMI. Hence all beans and interfaces import either *java.rmi.** or *javax.rmi.**, just as all other RMI applications do. EJB allow students to see some of the real world applications of RMI, and it gives them another chance to work with concepts that were covered throughout the course.

EJB fits naturally into a database course. An instructor can discuss both data modeling and data implementation techniques within the context of EJB. EJB allows for the abstraction of data by wrapping the data representing an object into one bean. Students can study and analyze an object by the operations and major data groups of the object.

For example, students can model a person object by using the person's name and address. The students discuss the public methods used in the entity and session beans associated with a person. When it is time to implement the beans and connect their abstraction to a database, then the discussion can turn to specifics associated with entity-relationship diagrams. Student must determine and store the actual facts and data comprising a name and address. Most people have several names that makeup their full name. Likewise, addresses are normally comprised of several pieces of data: street, city, state, and country. At this level, an instructor can discuss the SQL commands necessary to implement the data abstraction layer.

5. EXAMPLE BEANS

This section discusses three sample beans that illustrate the elementary concepts of EJB.

5.1 Stateless Session Bean

The first class example demonstrates a simple stateless session bean called *DemoBean* that does nothing more than return a String ("Hello World! We're back in business!") back to the client. The example is presented first with a demonstration followed by an explanation of the two interfaces, ending with a description of the bean class and a client.

The code below defines the remote interface, *Demo.java*. The important features to point out are that it extends *EJBObject*, and defines the bean's only business method, *demoSelect( )*.

```
//Remote interface
package demo;
import java.rmi.*;
import javax.ejb.*;

public interface Demo extends  EJBObject {
   public String demoSelect() throws
     RemoteException; }
```

The following code defines the home interface of the bean, *DemoHome.java*. Note that it extends *EJBHome* and defines the bean's only life cycle method, the *create( )* method.

```
//Home interface
package demo;
import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface DemoHome extends EJBHome {
   public Demo create() throws
     RemoteException, CreateException; }
```

The code below describes the bean class, *DemoBean.java*. This is where the algorithms and coding for the bean's methods are located. The first item to point out is that it implements *SessionBean*. Also notice that by implementing *SessionBean* all of the methods except for *demoSelect( )* are required as part of the interface. These methods are the life cycle methods that correspond to the ones found in the home interface. All beans contain them. The method *demoSelect( )* defines the only business method of *DemoBean*. It simply returns a String back to the client, or throws a *RemoteException*.

```
//Bean class
package demo;
import javax.ejb.*;
import java.rmi.*;
import java.util.*;
import java.io.*;

public class DemoBean implements SessionBean {

 public void ejbActivate() {
    System.out.println("ejbActivate
      called"); }

 public void ejbRemove() {
    System.out.println("ejbRemove
      called"); }
```

```
public void ejbPassivate() {
   System.out.println("ejbPassivate
     called"); }

public void
  setSessionContext(SessionContext ctx){
   System.out.println("SessionContext
     set"); }

public void ejbCreate () {
   System.out.println("ejbCreate
     called"); }

public String demoSelect() throws
  RemoteException  {
   return("Hello World!  We're back in
     business!"); } }
```

The final component of this demonstration is an analysis of the client, *DemoClient.java*. The client starts out by locating the object using the JNDI name specified in the deployment tool. In this case, the name is "*dhome*." Once the object is found and returned to the client, the object is cast to the type *DemoHome*. The *create( )* method is called, and a bean object is created and returned to the client. At this point the client is free to use the business methods of the bean. In this example, *demoSelect( )* is called and a String is returned to the client. The results are displayed and then the client ends.

```
// Client
package demo;
import javax.ejb.*;
import javax.naming.*;
import javax.rmi.*;
import java.io.*;
import java.util.*;

public class DemoClient {
 static DemoHome dhome1;
 static Demo demo1;

 public static void main(String[] args) {
  System.out.println("\nBegin Stateless
    Session DemoClient...\n");

  try {
    Properties p = System.getProperties();
    Context ctx = new InitialContext(p);

    System.out.println("Looking for DemoHome
      class...");
    Object objref = ctx.lookup("dhome");
    dhome1 = (DemoHome)
      PortableRemoteObject.narrow
      objref,DemoHome.class);

    System.out.println("Creating a
      DemoBean\n");
    demo1 = dhome1.create();

    System.out.println("Bean created");
    System.out.println("The result is:
      " + demo1.demoSelect());    }
  catch (Exception e) {
    e.printStackTrace();  }

  System.out.println("\nEnd
    DemoClient...\n");  }  }
```

5.2 Stateful Session Bean

The next example is a stateful session bean called *CountBean*. It is another very basic example, but it demonstrates how stateful session beans maintain a conversational state with the client. The goal of this bean is simply to count by one. Thus there will only be one business method, *count( )*, which simply returns an integer to the client after incrementing it. In this case, the value of the returned integer actually represents the conversational state of the bean.

The interfaces of this bean are similar to the interfaces of *DemoBean*. The remote interface defines the *count( )* method, and the home interface contains the *create(int val)* method. Notice that this method *create(int val)* cannot be included in a stateless session bean. Stateless session beans forbid parameters in the *create( )* method, whereas stateful session beans allow parameters. In this example *val* is the starting state of the counter.

The code of the bean class is also similar to *DemoBean.java* with one main difference. Since we added a parameter to the *create(int val)* method, the *ejbCreate(int val)* initialization method now takes *val* as a parameter as well. Again, this would not be allowed for stateless beans. Another important part of this bean class is that it contains an instance variable that gets initialized to *val*. It is this instance variable that gets incremented and returned to the client in the bean's business method, *count( )*.

The main lesson to be learned in this demonstration is the difference between stateless and stateful session beans. By using simple examples like *DemoBean* and *CountBean*, it is sometimes difficult to understand the full effects of maintaining a conversational state. An interesting assignment to illustrate this difference is to deploy the *CountBean* as both a stateful and stateless session bean. This requires making a few changes to the code, namely removing the parameters in the *create( )* and

ejbCreate( ) methods. Once the parameters are removed the bean will deploy as either a stateless or stateful session bean.

Once the bean is changed and deployed, a client program is needed that creates multiple instances of *CountBean*. Have each instance call the *count( )* method at least twice, and print out the value that is returned to the client. In the case of the stateless session bean, the value returned is incremented each time, regardless of the fact that different instances are calling the method. The results are 1, 2, 3, 4, 5, 6 and so on. For the stateful beans however, each bean instance has a different "copy" of the shared variable, so the incrementing that takes place is specific to each instance. For example, if three bean instances are created in the client program, the results are 1, 1, 1, 2, 2, 2, etc. This comparison gives students a more precise and accurate idea of the difference between the two types of session beans.

5.3 Container-managed Entity Bean

The last example is a container-managed entity bean. This bean will represent a row in a database that consists of six fields: login, email address, phone number, IP address, socket address, and last name. The structure of the remote interface is exactly the same as the previous two beans. The home interface is essentially the same as well, with the addition of the *findByPrimaryKey(Object primarykey)* method. This is used for searching the database.

The most apparent differences between the two types of beans appear in the code of the bean class. The code for *DirectoryBean.java* is shown below. Notice that this bean implements *EntityBean* instead of *SessionBean*, and defines six instance variables that correspond to the six fields of the database. Also, the *ejbCreate( )* method requires six parameters that represent the field values and initialize the instance variables. In entity beans, each *ejbCreate( )* method (there may be more than one) has an *ejbPostCreate( )* method that accepts the same parameters. Most entity beans include "set" and "get" methods for the instance variables, too.

```
//Container managed entity bean
package direct;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

public class DirectoryBean implements
EntityBean {
  public String login;  public String email;
  public String voice;  public String ip;
  public String sockad; public String name;

  public String getLogin() {
        return this.login;  }

  public String getEmail() {
        return this.email;  }
  public String getVoice() {
        return this.voice;  }
  public String getIp() {
        return this.ip;  }
  public String getSockad() {
        return this.sockad;  }
  public String getName() {
        return this.name;  }

  public String ejbCreate(String login,
    String email, String voice, String
    ip, String sockad, String name)
    throws CreateException{
      this.login=login;   this.email=email;
      this.voice=voice;   this.ip=ip;
      this.sockad=sockad; this.name=name;
      return null;  }

  public void setEntityContext(
    javax.ejb.EntityContext ctx){ }
  public void ejbActivate() { }
  public void ejbPassivate() { }

  public void ejbPostCreate(String
    login, String email, String voice,
    String ip, String sockad, String
    name) {}

  public void ejbRemove() throws
    RemoteException { }

  public void ejbLoad() { }
  public void ejbStore() { }
  public void unsetEntityContext(){}   }
```

Before a bean can be accessed, it must be compiled and deployed. During the deployment process, the primary key field should be specified as *login*. In order to create new rows in the database, a client is needed to create new bean instances. The code for the client is simple. It asks the user for six field values, and creates a new bean instance using the entered data. The login name should be used as the primary key, and the client should be prepared to handle an exception thrown by the bean class if a repeated login name is entered. Another client is needed to search the database. This client will ask the user for a login name to

search for, and then locate the desired bean using the *findMyPrimaryKey(String login)* method.

6. EJB PROJECT

After experimenting with some of the simple beans described in the previous section, it may be desirable for students to work on a larger project involving EJB. The project is slightly more difficult and includes both stateless session beans and container-managed entity beans. The goal is to create an information directory application that stores data in the same six fields as before: login, email address, phone number, IP address, socket address, and last name. Within this directory, it is possible to update the stored information, create new entries, and search for entries by login name. The assignment is to use three clients, three session beans, and one entity bean to model this scenario.

The directory structure itself can be created using entity beans just like the one described in the preceding example. Each entry in the directory is represented by an instance of this entity bean. Instead of using clients to add new entries and search the database as before, session beans should be used. Coding the beans should not be overly difficult for students, however making the beans communicate with one another can be complicated at times. One approach is to have the clients locate both the entity and session beans' home interfaces using JNDI. A reference to the entity bean should then be passed to the session beans. Each session bean must have it's own client, although they will be very similar in structure. This project allows students to gain experience with session beans, entity beans, and clients that interact with both types of beans at once.

7. EVALUATION OF EJB EXPERIENCES

Learning and using EJB was a very challenging task. The coding itself was not difficult, however getting the deployment tool configured so that the beans were deployed correctly was extremely frustrating at times. The problem was that every deployment tool was different, and some settings were machine dependent. It was also hard to find documentation that answered my questions. I often resorted to trial and error. However once I got the first bean to deploy, getting the others to work was much easier. The more I worked with EJB, the more comfortable I was using it, and the more I understood. It was a great feeling to finally get a series of interacting beans up and running smoothly. –Jeannie Albrecht

8. SUMMARY

There is a growing demand for transferable, platform independent, database managing applications in the business world today. EJB provide a simple and effective alternative to previous solutions by incorporating the portability of JavaBeans, the database managing concepts of JDBC (Java Database Connectivity), and distributed processing theories of JavaRMI into one revolutionary new technology. Students enjoy learning about real world applications of the concepts taught in both Database Systems and Parallel and Distributed Processing courses.

9. ACKNOWLEDGEMENTS

The authors express their appreciation to Gettysburg College for partially funding this project.

10. REFERENCES

[1] Monson-Haefel, Richard. *Enterprise JavaBeans*. O'Reilly & Associates, 1999.

[2] Roman, Ed. *Mastering Enterprise JavaBeans*. Wiley Computer Publishing, 1999.

[3] Simplified Guide to the Java 2 Platform, Enterprise Edition, Sunsoft, http://java.sun.com/j2ee/j2sdkee/techdocs/guides/j2ee-overview/cover.fm.html

[4] Enterprise JavaBeans Tutorial, Sunsoft, http://developer.java.sun.com/developer/onlineTraining/Beans/EJBTutorial/index.html

[5] Enterprise JavaBeans FAQ, Sunsoft, http://java.sun.com/products/ejb/faq.html