# Mapping UML Class Diagrams to CMP Entity Beans

**Gail C. Anderson**
**Paul L. Anderson**

Copyright © 2002 InformIT, http://www.informit.com/

## Introduction

The Enterprise JavaBeans 2.0 specification defines three types of JavaBeans: session, entity, and message-driven. Of the three, we use entity beans to model business data. This article describes how a UML class diagram can accurately model business data. We also discuss how to map a class diagram to entity beans with Container-Managed Persistence (CMP) and Container-Managed Relationships (CMR).

## What Is UML?

The Unified Modeling Language (UML) is an industry-standard notation for describing and analyzing object oriented systems. UML is independent of the process you use to design and build systems. Its main advantage is that with a standard notation, you can describe your system to a wide audience and obtain a consistent understanding.

UML consists of a set of diagrams that communicate a view of a system. For example, a class diagram describes objects within a system and their relationships to other objects. A use case diagram describes the functionality of a system—how someone uses the system. An activity diagram describes the steps that are required to do something: complete a use case, deploy an application, or make broccoli and garlic penne, for example. Although UML has a rich set of diagrams, we will use only class diagrams for this discussion.

## What Are Entity Beans?

Entity beans are J2EE components that execute under the control of a J2EE application server. The application server uses an Enterprise JavaBeans (EJB) container to handle system services such as thread control, security, transaction management, database connections, and life cycle management on behalf of each bean. Entity beans reflect business data; they mirror persistent data from an underlying database. Entity beans in general are not used to fulfill a business process; their role is limited to modeling data.

There are two types of entity beans: Bean-Managed Persistence (BMP) and Container- Managed Persistence (CMP). With BMP entity beans, the bean developer writes the database access code which provides entity bean persistence. With CMP entity beans, the application server generates the required database access code for entity bean persistence. Furthermore, the 2.0 specification includes local interfaces and container-managed relationships that allow CMP entity beans to model real-world enterprise data requirements. This includes entity beans with relationships to other entity beans. Therefore, we use CMP entity beans for this discussion.

## State–Capital City Example

UML class diagrams describe objects and the associations between them. Associations detail how objects are related to each other, how many objects are involved in a relationship (multiplicity), whether you can determine the related objects of an entity relationship (navigability), and the life cycle characteristics (when relationships begin and when they end). These details are important to describe CMP entity bean persistent field methods, relationship field methods, and deployment descriptor declarations.

Let's start with a simple example. The idea is that every state has one capital city and each capital city belongs to only one state. Figure 1 shows a class diagram modeling these entities and their association (relationship).

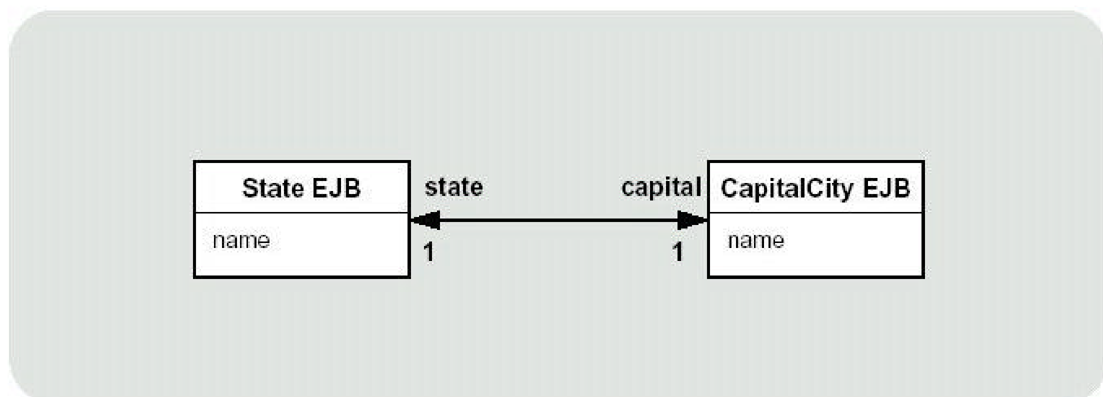


**Figure 1. State EJB and CapitalCity EJB Class Diagram**

### Persistent Fields

This UML class diagram depicts State EJB with a single attribute, name. Object attributes model persistent data; thus State EJB has a single persistent data field. Furthermore, all entity beans have a primary key, which we do not need to model in a class diagram. Its existence is implied. CapitalCity EJB also contains a single name attribute and an implied primary key.

### Mapping Persistent Fields

Every CMP entity bean has a local home interface, a local interface, and a bean implementation class. As a bean developer, you must write both interfaces as well as the bean implementation class. Here, we will focus only on the CMP bean implementation class and the modeling information provided by a UML class diagram. Ultimately, the data modeling depicted in the UML class diagram maps to methods in the bean implementation class. We map class diagram attributes as abstract access methods for persistent fields. Each attribute has a corresponding setter and getter access method. We also create access methods for the primary key. Here are the attributes for entity beans State EJB and CapitalCity EJB.

```
// StateBean.java
// Bean implementation class for State entity bean
public abstract class StateBean implements EntityBean {

   // Access methods for primary key
   public abstract String getStateID();
   public abstract void setStateID(String id);

   // Access methods for class attributes (persistent fields)
   public abstract String getName();
   public abstract void setName(String name);
   . . .
}

// CapitalCityBean.java
// Bean implementation class for CapitalCity entity bean
public abstract class CapitalCityBean implements EntityBean {

   // Access methods for primary key
   public abstract String getCapitalCityID();
   public abstract void setCapitalCityID(String id);

   // Access methods for class attributes (persistent fields)
   public abstract String getName();
   public abstract void setName(String name);
   . . .
}
```

Note that each bean implementation class is abstract because the application server generates database access code as well as the code to implement the abstract access methods.

### Relationships

What else does Figure 1 tell us about state and capital city objects? The objects are connected by a line, denoting an association or relationship. We describe an entity bean relationship by specifying a name (*role*) on each end of the line and the multiplicity, navigability, and life cycle requirements for the relationship. Each state has exactly one capital city and each capital city has exactly one state, denoted by the 1 next to the relationship arrow. Thus, the relationship multiplicity between state and capital city

is one to one. When a relationship multiplicity is either one or optional (zero or one) we call it *singular*. Furthermore, this relationship is bidirectional, meaning that you can navigate in both directions. You can determine the capital city from the state and the state from the capital city (denoted by the double-headed arrow). A bidirectional relationship is also called two-way navigation. Unidirectional relationships provide one- way navigation.

### Mapping Relationships

Each association in a class diagram maps to an entity bean relationship. In CMR, the container maintains the relationship fields for you, but you must provide abstract access methods for those relationships that are navigable. Singular relationships (as in our example) specify the local interface of the related entity bean for arguments and return types. The bidirectional association between State EJB and CapitalCity EJB maps to the following abstract access methods in the corresponding bean implementation code. (We assume that the local interface for State EJB is `StateLocal` and the local interface for CapitalCity EJB is `CapitalCityLocal`.)

```
// StateBean.java
// Bean implementation class for State entity bean
public abstract class StateBean implements EntityBean {
   . . .
   // Access methods for relationship field
   public abstract CapitalCityLocal getCapital();
   public abstract void setCapital(CapitalCityLocal city);
   . . .
}

// CapitalCityBean.java
// Bean implementation class for CapitalCity entity bean
public abstract class CapitalCityBean implements EntityBean {
   . . .
   // Access methods for relationship field
   public abstract StateLocal getState();
   public abstract void setState(StateLocal state);
   . . .
}
```

### Relationship Life Cycle Issues

Because the multiplicity for both ends of the state-capital city relationship is exactly one, this implies that there must always be a capital city for a state and a state for a capital city *throughout the lifetime of both entities*. When a multiplicity is *mandatory* (meaning at least one), this implies that a capital city cannot exist without a state and a state cannot exist without a capital city. Mandatory multiplicities have two requirements:

- You must define a mandatory relationship throughout the lifetime of the entity. This means that the relationship must be initialized during the create process. Typically, this is done in bean implementation method `ejbPostCreate()`. When you create an entity for state New York, you must also create and initialize its relationship with capital city Albany.
- If you destroy an entity that has a mandatory relationship, then you must delete the related entity. This is called *cascading delete*. Thus, if you delete State EJB Nebraska, then you must also delete CapitalCity EJB Lincoln.

### Mapping Life Cycle Characteristics

When both ends of a relationship are mandatory, this produces the proverbial chicken and egg

dilemma. That is, you can't initialize the capital city until the corresponding state entity exists, and you can't initialize the state until the corresponding capital city exists. To resolve this dilemma, you pick one end to initialize during the create process; the good news is that the EJB container initializes the other end for you. Let's examine additional code snippets from the bean implementation class for State EJB, showing the initialization of its relationship with CapitalCity EJB.

```
// StateBean.java
// Bean implementation class for State entity bean
public abstract class StateBean implements EntityBean {
   . . .
   // EntityBean method ejbCreate()
   public String ejbCreate(String name, CapitalCityLocal capital)
   throws CreateException {
      String newKey = DBUtil.dbGetKey();
      setStateID(newKey);
      setName(name);
      return newKey;
   }

   // EntityBean method ejbPostCreate()
   public void ejbPostCreate(String name, CapitalCityLocal capital) {
      setCapital(capital);
   }
   . . .
}
```

The parameters for `ejbCreate()` and `ejbPostCreate()` must be the same. Note that we generate a primary key by calling `DBUtil.dbGetKey()`. Primary key generation is beyond the scope of this article. Many database servers provide primary key generation routines for you.

Method `ejbCreate()` initializes the entity bean's persistent fields by calling the set access methods we defined earlier. At the same time, the container inserts a matching row in the underlying database for the entity bean. Because the entity bean is not fully instantiated at this time, you cannot invoke any methods except the persistent data field access methods.

The EJB container invokes `ejbPostCreate()` after `ejbCreate()`. We use method `ejbPostCreate()` to initialize the relationship between State EJB and CapitalCity EJB by invoking the `setCapital()` access method we defined earlier. The EJB container initializes the other end of the relationship. The container also inserts matching rows in the underlying database to maintain the relationship fields (typically in a cross-reference database table).

### Mapping Declarative Information

Bean developers communicate much of an entity bean's characteristics through abstract access methods written in the bean implementation code. Deployment tools inspect this code and infer information about persistent fields and relationship fields. However, we must still identify primary key fields and specify characteristics of relationship fields. Armed with a class diagram that specifies the relationship, its multiplicity, and its navigability, this is quite straightforward. (And, application servers will typically provide a deployment tool that generates an XML deployment descriptor for you.)

Here are the deployment descriptor tags that describe the relationship between State EJB and CapitalCity EJB. (We `bold` the XML tag values to improve readability.)

```
<relationships>
   <ejb-relation>
      <ejb-relation-name></ejb-relation-name>
      <ejb-relationship-role>
         <ejb-relationship-role-name>CapitalCityBean
         </ejb-relationship-role-name>
```

```
        <multiplicity>One</multiplicity>
        <cascade-delete />
        <relationship-role-source>
          <ejb-name>CapitalCityBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>state</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>StateBean</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <cascade-delete />
        <relationship-role-source>
          <ejb-name>StateBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>capital</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
 </relationships>
```

Notes about the relationship described in the deployment descriptor:

- The relationship multiplicity <**multiplicity**> is One to One.
- Cascading deletes <**cascade-delete /**> apply in both directions (because the relationship is mandatory in both directions).
- The CMR fields <**cmr-field-name**> are **state** and **capital** for CapitalCity and State, respectively. Note that these are the labels (roles) we use on the relationship line in Figure 1.

## Customer–Orders Example

Now that we've covered a simple example that maps a UML class diagram to CMP entity bean implementation code, let's look at a more involved example. Figure 2 shows a class diagram modeling entities customer (Customer EJB), orders (Order EJB) and line items (LineItem EJB).

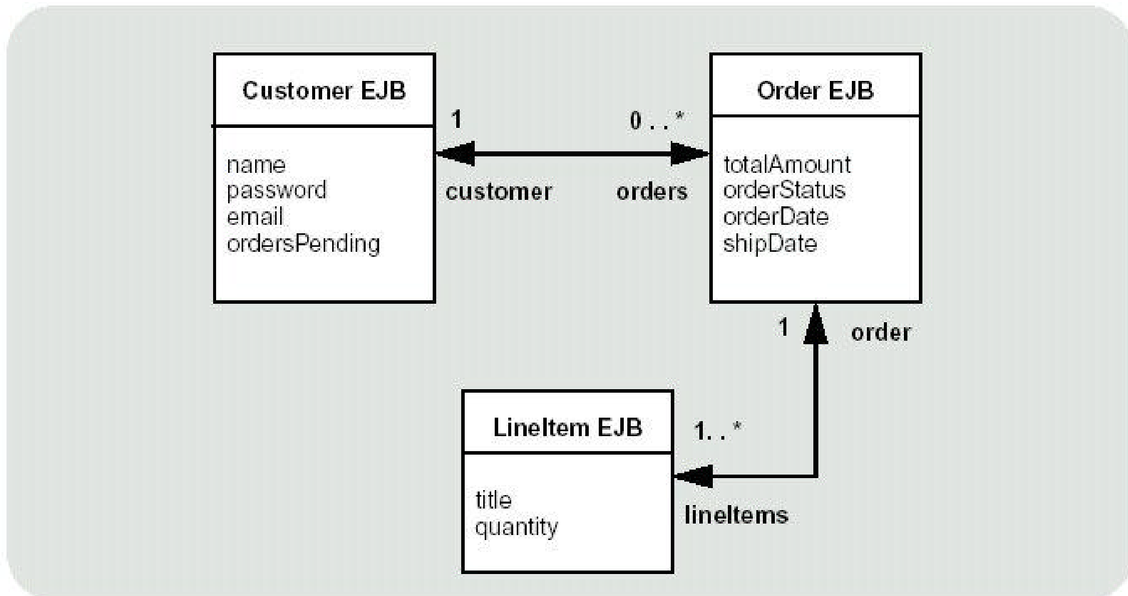


**Figure 2. Customer EJB, Order EJB, and LineItem EJB Class Diagram**

### Persistent Fields

Customer EJB has several attributes, including name and email address, which map to corresponding persistent fields. Likewise, Order EJB has attributes that include total amount (a double) and an order status (an integer). Let's look at a portion of the bean implementation code for Customer EJB, showing the mapping for attributes name and email, as well as the implied primary key.

```
public abstract class CustomerBean implements EntityBean {

   // Access methods for primary key
   public abstract String getCustomerID();
   public abstract void setCustomerID(String id);

   // Access methods for attributes
   public abstract String getName();
   public abstract void setName(String name);

   public abstract String getEmail();
   public abstract void setEmail(String email);
   . . .
}
```

Remember, both the class and the methods are abstract because the application server generates the implementation code for these methods. Here is a portion of the bean implementation code for Order EJB.

```
public abstract class OrderBean implements EntityBean {

   // Access methods for primary key
   public abstract String getOrderID();
   public abstract void setOrderID(String id);

   // Access methods for attributes
   public abstract double getTotalAmount();

   public abstract void setTotalAmount(double amount);

   public abstract int getOrderStatus();
   public abstract void setOrderStatus(int status);
   . . .
}
```

### Relationships

What sorts of relationships does the class diagram in Figure 2 show? You'll note that a customer may have more than one order associated with it. The notation `0..*` means many, where many may be zero or more. This means that it is perfectly okay for a customer to have no orders, as well as having multiple orders. When a relationship allows many, we use a Java collection (`java.util.Collection`) for the relationship type. Here is a portion of the bean implementation code for Customer EJB that shows the mapping for the relationship with Order EJB.

```
public abstract class CustomerBean implements EntityBean {
   . . .
   // Access methods for relationship fields
   public abstract Collection getOrders();
   public abstract void setOrders(Collection orders);
   . . .
}
```

Now let's look at the Order EJB. First, an order must have exactly one customer. (This is mandatory, meaning that an order must have an associated customer at all times.) What is the multiplicity between order and line item? An order has many line items, where many is one or more (`1..*`), and a line item has exactly one order. One or more means mandatory, which indicates that an order must have at least one line item at all times (zero is not allowed). Similarly, a line item must always have one associated order. As we learned earlier, mandatory means that these relationships are initialized during the create process. Here is a portion of the bean implementation code for Order EJB that shows the mapping for the order relationship with customer and line item.

```
public abstract class OrderBean implements EntityBean {
   . . .
   // Access methods for relationship fields
   public abstract CustomerLocal getCustomer();
   public abstract void setCustomer(CustomerLocal customer);

   public abstract Collection getLineItems();
   public abstract void setLineItems(Collection lineItems);
   . . .
}
```

Note that we use the local interface object (`CustomerLocal`) for the singular customer relationship and `java.util.Collection` for the many line item relationship.

### Relationship Life Cycle Issues

Since the relationship between customer and order is one to many (where many is zero or more), a Customer EJB does not necessarily have an associated order. This means the relationship cannot be initialized during Customer EJB's create process. However, since an order must have an associated customer, the initialization does occur during Order EJB's create process. Let's see how this is done with the `ejbCreate()` and `ejbPostCreate()` methods for Order EJB.

```
public abstract class OrderBean implements EntityBean {
   . . .
   public String ejbCreate(double totalAmount,
      int orderStatus, long orderDate, long shipDate,
      CustomerLocal customer)
      throws CreateException {

      String newKey = DBUtil.dbGetKey();

      // initialize all persistent fields
      setOrderID(newKey);
      setTotalAmount(totalAmount);
      setOrderStatus(orderStatus);
      setOrderDate(orderDate);
      setShipDate(shipDate);
      return newKey;
   }

   public void ejbPostCreate(double totalAmount,
         int orderStatus, long orderDate, long shipDate,
         CustomerLocal customer) {

      // initialize relationship with customer
      setCustomer(customer);
   }
   . . .
}
```

Note that `ejbCreate()`'s arguments match those of `ejbPostCreate()`. As before we generate the primary key internally. In `ejbPostCreate()`, we initialize the relationship with Customer EJB. The container takes care of initializing the other end of the relationship.

Where is the relationship between order and line item initialized? Since this relationship is mandatory in both directions, we must choose one side and provide the initialization in `ejbPostCreate()`. For order and line item, initialization occurs in LineItem EJB, as follows.

```
public abstract class LineItemBean implements EntityBean {
   . . .
   public String ejbCreate(String title, int quantity,
        OrderLocal order) throws CreateException {
      String newKey = DBUtil.dbGetKey();
```

```
        setLineItemID(newKey);
        setTitle(title);
        setQuantity(quantity);
        return newKey;
    }

    public void ejbPostCreate(String title, int quantity,
            OrderLocal order) {
        // associate this LineItem EJB with Order EJB
        setOrder(order);
    }
    . . .
}
```

### Mapping Declarative Information

Our customer–order example has two associations (customer–orders and order–line items). From the information in our class diagram, we can specify declarative information. Here are the deployment descriptor tags that describe the relationship between Customer EJB and Order EJB. This relationship is one to many and cascading deletes apply to Order EJB (if we delete a customer, its associated orders must also be deleted since an order cannot exist without a customer). Note that CMR field `orders` is type `java.util.Collection` (since it is many).

```
 <relationships>
    <ejb-relation>
      <ejb-relation-name></ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>CustomerBean</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>CustomerBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>orders</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>OrderBean</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <cascade-delete />
        <relationship-role-source>
          <ejb-name>OrderBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>customer</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
```

Here are the deployment descriptor tags that describe the relationship between Order EJB and LineItem EJB. This relationship is one to many and cascading deletes apply to LineItem EJB (if we delete an order, its associated line items must also be deleted). CMR field `lineItems` is type `java.util.Collection`.

```
    <ejb-relation>
      <ejb-relation-name></ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>OrderBean</ejb-relationship-role-name>
        <multiplicity>One</multiplicity>
        <relationship-role-source>
          <ejb-name>OrderBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>lineItems</cmr-field-name>
          <cmr-field-type>java.util.Collection</cmr-field-type>
```

```
        </cmr-field>
      </ejb-relationship-role>
      <ejb-relationship-role>
        <ejb-relationship-role-name>LineItemBean</ejb-relationship-role-name>
        <multiplicity>Many</multiplicity>
        <cascade-delete />
        <relationship-role-source>
          <ejb-name>LineItemBean</ejb-name>
        </relationship-role-source>
        <cmr-field>
          <cmr-field-name>order</cmr-field-name>
        </cmr-field>
      </ejb-relationship-role>
    </ejb-relation>
 </relationships>
```

## Conclusion

UML class diagrams are useful when modeling business data. By accurately modeling attributes and associations of class entities, we can easily map these class diagram specifications to entity beans with CMP. Class attributes map to abstract access methods for persistent fields and association roles map to abstract access methods for relationship fields. Navigability determines whether relationship access methods appear in both related entity beans, or just one. Furthermore, multiplicity notation determines the correct type for relationship fields, life cycle issues, and cascading delete characteristics.