MindTelligent, Inc.

OC4J and EJB 2.0 Container Managed Relationships

EJB 2.0 & OC4J
(Oracle Components for Java)

Container Managed Relationships

A Technical White Paper

Published by MindTelligent, Inc. 2034, Lamego Way, El Dorado Hills, CA 95762
Copyright © 2003 by MindTelligent, Inc. All rights reserved.

No part of this tutorial may be reproduced or utilized in any form or by any means, electronic or mechanical including photo-copying, recording, or by an information storage retrieval system, without permission in writing from the publisher. Inquiries should be addressed to MindTelligent, Inc. 2034 Lamego Way El Dorado Hills, CA 95762. (916)-595-1884.

All brand names, product names, trademarks, and registered trademarks are the property of their respective owners.

INTRODUCTION

One of the most challenging tasks in working in the object-relational mapping area has proven to be selecting the appropriate modeling of relationships between objects and their mapping into corresponding table relations in he database. While EJB 1.1 didn't provide any formal support for relationships, EJB 2.0 addresses this by introducing the concept of Container Managed Relationships (CMR). Before we go into more details of EJB 2.0 CMR, let's review some of the limitations in EJB 1.1.

EJB 1.1 – A REVIEW

Entity Enterprise JavaBeans (EJBs) in the EJB 1.1 specification were fairly simple representations of business objects in a storage mechanism, usually a database. There was no defined standard for defining relationships between objects. Developers had to code the relationships into their classes. If the underlying data structure changed, developers would have to re-code the relationships as well.

A typical implementation of a relationship between two EJBs would involve either coding each side of the relationship into the respective EJBs, or coding the relationship into another bean, which managed the participants in the relationship. J2EE design patterns emerged to provide repeatable ways to do this manually, for example the Session Façade pattern.

Alternatively many J2EE container vendors provided a non-standard way of declaring relationships between entity beans. For example in OC4J 1.0.2.2.2, Object-Relational mappings could be declared in the container specific file `orion-ejb-jar.xml`. However this mapping declaration and behavior was not portable across containers from different vendors as each container requires a different configuration file.

So how has that changed in EJB 2.0?

EJB 2.0

The EJB 2.0 specification addresses many of the flaws in the EJB 1.1 specification by standardizing many aspects of EJB behavior. The specification takes many good points from vendor specific enhancements and incorporates them into the standard.

New features in the EJB 2.0 specification include local interfaces for improved performance, an abstract persistence schema for CMP classes, easing development and portability, message-driven beans, and of course, Container Managed Relationships. CMR enables a developer to declare the relationship between two entity EJBs in the `ejb-jar.xml` deployment descriptor. There is no coding of the relationship required. The developer simply includes the relevant accessor methods in the EJB and then declares the relationship in the `ejb-jar.xml` using the `<relationships>` element.

EJB RELATIONSHIPS

Speaking in academic terms, relationships are categorized by their cardinality (also referred to as multiplicity) and by their direction. What does that mean? It's actually only half as complicated as it might sound at first glance.

The cardinality simply describes the number of instances on each side of the relationship. Let' say you have a relationship between a department and all employees working in the department. You choose to

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

model it in such a way that each employee can only work in one department, and each department can have many employees. That would be considered a one-to-many relationship between a department and its employees.

The direction describes who is associated with whom in a relationship. Again, it's best explained by a simple example. In the previous example if an employee would know his department, but the department wouldn't directly know of all the employees working for it, then that would be a unidirectional relationship. But if you would design the relationship in such a way, that employees would know their department and departments would also know all the employees that work for them, then it would be a bi-directional relationship. It sounds more complicated than it is, doesn't it?

Now that we've covered the dry theory behind relationships, let's examine a few common relationship examples including their cardinality and their direction.

One-to-One

One-to-one (1:1) relationships are more common in the object world than the relational world since 1:1 entities are usually stored in one table for performance reasons. In the object world you may want to use the entities in different areas of an application so creating separate entities makes more sense.

The simplest relationship is the unidirectional 1:1 relationship. One parent has a relationship with one child but the child is not related to the parent. An example is the relationship between an employee and his address. You would often want to find the address of an employee but you would probably not want to find an employee from an address. By modeling the relationship this way (unidirectional) you also ensure, that the address object can easily be reused by other components, since it doesn't implement a reference back to the employee object.

The next step up in complexity is a bi-directional 1:1 relationship. As already discussed, bi-directional means, that there is a relationship from the parent to the child and the child back to the parent. An example could be an employee and his office. If you want to find out whether a certain office is vacant, you would want to know whether an employee is currently associated with the office.

One-to-Many

One-to-many (1:M) relationships probably represent the most common form of relationship. They describe one parent and many children.

A unidirectional 1:M relationship is when one parent is related to one or more children but the children are not related to the parent. An example of this is an employee and their phone numbers. If you want to call a person you can find all of their phone numbers from the person. However you may not want to find a person by their phone number. And once again, this way the phone number can easily be reused in other models.

A bi-directional 1:M relationship is when a parent has one or more children and each of those children is related back to one parent. The DEPT-EMP tables in the SCOTT/TIGER schema of an Oracle database are the classic example. A department can have many employees but an employee can only belong to one department. You can find out what employees belong to a given department and also what department a given employee belongs to.

Many-to-Many

The final category of relationships is the many-to-many (M:N) relationship. In the M:N relationship a parent can have many children and each child can have many parents.

A unidirectional M:N relationship is best described with an example, the relationship between projects and employees. Let's say you wanted to model this relationship such that employees would know all the projects they are working at, but you wouldn't want to be able to tell what employees are working at a certain project. In that case you would model this as a unidirectional many-to-many relationship.

A bi-directional relationship enables the parents to query the children, but also the children to query back

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

to their parents. Let's take another look at the employees-projects example. Most likely you would model this as a bi-directional relationship, thereby enabling the user to query all employees working on a particular project.

EJB RELATIONSHIP EXAMPLE

Ready? Let's look at a common example. Assume you are modeling the relationship between departments and employees, as discussed earlier in this document. A department can have many employees working for it, but an employee can only work for one department at a time. Since you want to be able to query all employees who are working for a particular department and since you also want to be able to find the department an employee is currently working for, you would model the relationship as a bi-directional relationship. Therefore this is an example of a one-to-many, bi-directional relationship.

The employee is represented by the `EmpBean` EJB and the department by the `DeptBean` EJB.

Accessor Methods

For simplicity we will only concentrate on the accessor methods relevant to the relationship. The Department EJB needs accessor methods to get and set the employees. The method declarations in the bean implementation class would look like this:

```
public abstract Collection getEmployees();
public abstract void setEmployees(Collection employees);
```

Since the relationship is a 1:M each method deals with a `java.util.Collection`. This Collection object will be a collection of the Employee EJB local interfaces. If no employees are found an empty `Collection` is returned by the get method. If you want to remove all employees, an empty `Collection` is passed in via the set method. You can use the `java.util.Set` object instead of the `Collection` object. The difference is that a `Set` will contain unique objects where as the `Collection` may not.

If you examine the `ejb-jar.xml` for the Department EJB you would not find a reference to the Employees field in the `<cmp-field>` tag that's usually required for each abstract accessor. This is because the field is not a container managed persistence (CMP) field. It is a container managed relationship (CMR) field.

The Employee EJB needs to contain methods to access the parent, in this case the department. The accessors are shown below.

```
public abstract DeptLocal getDept();
public abstract void setDept(DeptLocal deptLocal);
```

This time the accessors deal with a single instance of `DeptLocal` (the local interface for the `DeptBean` EJB) rather than a `Set` or `Collection` as it was the case for employees. Why? Because we're dealing with a 1:M relationship, which means that one employee will return only one department, whereas one department will return a collection of employees.

One important point to note is that CMR managed relationships are only supported for local EJB interfaces. Remote interfaces and therefore distributed relationships across containers are not supported in the current specification.

The <ejb-relation> tag

Once the accessor methods have been written, the relationship has to be declared using the `<ejb-relation>` tag. Below is the XML for our example. This is entered in the `ejb-jar.xml` deployment descriptor.

```
<ejb-relation>
 <ejb-relation-name>Dept-Emps</ejb-relation-name>
 <ejb-relationship-role>
```

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

```
  <ejb-relationship-role-name>Dept-has-Emps
  </ejb-relationship-role-name>
  <multiplicity>One</multiplicity>
  <relationship-role-source>
   <ejb-name>DeptBean</ejb-name>
  </relationship-role-source>
  <cmr-field>
   <cmr-field-name>employees</cmr-field-name>
   <cmr-field-type>java.util.Collection
   </cmr-field-type>
  </cmr-field>
 </ejb-relationship-role>
 <ejb-relationship-role>
  <ejb-relationship-role-name>Emps-have-Dept
  </ejb-relationship-role-name>
  <multiplicity>Many</multiplicity>
  <cascade-delete/>
  <relationship-role-source>
   <ejb-name>EmpBean</ejb-name>
  </relationship-role-source>
  <cmr-field>
   <cmr-field-name>dept</cmr-field-name>
  </cmr-field>
 </ejb-relationship-role>
</ejb-relation>
```

This XML snippet looks a bit daunting at first, so let's break it down into each tag.

<ejb-relation-name>

This optional descriptive tag is a simple display name for the relationship. You can leave it out altogether but it makes the XML more understandable for humans.

<ejb-relationship-role>

There are two of these tags, one for each side of the relationship. The sub tags include information on the type of relationship, the EJBs involved and the fields the relationship uses.

<ejb-relationship-role-name>

Again an optional descriptive name that can be included to describe each side of the relationship.

<multiplicity>

This tag defines what type of relationship is being declared, 1:1, 1:M or M:N. Possible values for this tag are `One` or `Many`. Our example is a one-to-many. `One` Department has `Many` Employees. Therefore on the Department side of the declaration the multiplicity is `One` and on the Employee side it is `Many`.

For a many-to-many relationship both sides would have a multiplicity of Many. For a one-to-one relationship both sides would have the value One. Note that this does not specify whether the relationship is unidirectional or bi-directional (see <cmr-field>).

<cascade-delete/>

In many situations child data has no meaning without parent data. If a parent is deleted from the system then the child record(s) should also be deleted. This behavior can be specified using the `<cascade-`

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

delete/> tag. The tag can only be assigned to a child entity bean with a multiplicity of one or many that has a parent with a multiplicity of one. Therefore it can only be assigned to the child of a 1:1 or 1:M relationship. You can't cascade-delete a child of a many-to-many relationship.

<relationship-role-source>

This tag defines the EJB that the relationship maps onto. It contains the `<ejb-name>` sub-tag that declares the EJB name, defined in the entity declaration. It can also contain an optional description

<cmr-field>

The parent entity of the relationship must declare a `<cmr-field>`. For the child it is dependent on whether the relationship is a unidirectional or a bi-directional relationship. For a unidirectional relationship the child must not declare the `<cmr-field>` element. For a bi-directional relationship it must declare the `<cmr-field>` element. Or in other words, at least one `<cmr-field>` tag must be present per relationship, the absence of the second one is what makes the relationship unidirectional.

The element has two possible sub-tags. The `<cmr-field-name>` tag declares the field name that the relationship uses. This has to map onto the accessor methods in the EJB. So for the Employee EJB, the accessors were `getDept()` and `setDept()`. Therefore the `<cmr-field-name>` has the value of `dept`.

The other possible sub tag is the `<cmr-field-type>`. This is only used if the accessors return a `Collection` or a `Set`. The only two values that this tag can have are `java.util.Collection` or `java.util.Set`. As you can see the Department relationship declaration uses the `<cmr-field-name>` of `employees` and the `<cmr-field-type>` of `java.util.Collection`. This maps back to the accessors `getEmployees()` and `setEmployees()`, which use a `Collection` object.

MORE EXAMPLES

One-to-One unidirectional

Below are the elements from the ejb-jar.xml file that describe the one-to-one unidirectional relationship between an employee and an address. The employee is represented by the `EmpBean` EJB and the address by the `AddressBean` EJB.

```
<ejb-relation>
 <ejb-relation-name>Emp-Address</ejb-relation-name>
 <ejb-relationship-role>
  <ejb-relationship-role-name>Emp-has-Address
  </ejb-relationship-role-name>
  <multiplicity>One</multiplicity>
  <relationship-role-source>
   <ejb-name>EmpBean</ejb-name>
  </relationship-role-source>
  <cmr-field>
   <cmr-field-name>address</cmr-field-name>
  </cmr-field>
 </ejb-relationship-role>
 <ejb-relationship-role>
  <ejb-relationship-role-name>Address-has-Emp
  </ejb-relationship-role-name>
  <multiplicity>One</multiplicity>
  <cascade-delete/>
  <relationship-role-source>
   <ejb-name>AddressBean</ejb-name>
  </relationship-role-source>
```

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

```
 </ejb-relationship-role>
</ejb-relation>
```

As you can see there are two main differences between this example and the one-to-many relationship. Both sides of the relationship have the multiplicity of `One` to define the one-to-one nature. Secondly the Address relation does not have a `<cmr-field>` element. This means that the relationship is unidirectional. Lastly, this relationship has a `<cascade-delete>`, which indicates that removal of the `EmpBean` will also remove the related `AddressBean`.

Many-to-Many bi-directional

The relationship defined below is a many-to-many bi-directional relationship between employees (`EmpBean` EJB) and projects (`ProjectBean` EJB).

```
<ejb-relation>
 <ejb-relation-name>Emps-Projects</ejb-relation-name>
 <ejb-relationship-role>
  <ejb-relationship-role-name>Projects-have-Emps
  </ejb-relationship-role-name>
  <multiplicity>Many</multiplicity>
  <relationship-role-source>
   <ejb-name>ProjectBean</ejb-name>
  </relationship-role-source>
  <cmr-field>
   <cmr-field-name>employees</cmr-field-name>
   <cmr-field-type>java.util.Collection
   </cmr-field-type>
  </cmr-field>
 </ejb-relationship-role>
 <ejb-relationship-role>
  <ejb-relationship-role-name>Emps-have-Projects
  </ejb-relationship-role-name>
  <multiplicity>Many</multiplicity>
  <relationship-role-source>
   <ejb-name>EmpBean</ejb-name>
  </relationship-role-source>
  <cmr-field>
   <cmr-field-name>projects</cmr-field-name>
   <cmr-field-type>java.util.Collection
   </cmr-field-type>
  </cmr-field>
 </ejb-relationship-role>
</ejb-relation>
```

This time you can see that both sides of the relationship have the multiplicity of `Many` and also that both sides of the relationship have a `<cmr-field>` declaration, signifying the bi-directional relationship of the many-to-many relationship

EJB—DATABASE MAPPING

You might have noticed that so far all we did was to describe the object relationships in the `ejb-xml.jar` deployment descriptor based on the EJB 2.0 specification. We have not yet specified how the relationship between the EJBs maps into the corresponding table relations in the database. Interestingly enough, the EJB 2.0 specification doesn't describe this mapping, but rather leaves it to the individual

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

containers to implement the mappings in vendor specific deployment descriptors. In the case of OC4J, this vendor specific mapping information is stored in the `orion-ejb-jar.xml` deployment descriptor.

Generally speaking there are two ways to map your EJBs and their CMRs to corresponding tables and relationships in the database.

a) You can tell OC4J to auto-generate the tables and table relations during the deployment of your EJB module. OC4J will also automatically map all EJBs and EJB relationships to the underlying database objects by auto-generating the `orion-ejb-jar.xml` file. For more information on naming conventions, generation and mapping rules, and other useful information, please refer to the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide.* While this option is particularly useful for Rapid Application Development (RAD) like development scenarios and prototyping, it usually isn't the right way for production level application development, where you usually need to map your object world to an already existing database schema. This is where the next scenario comes into play.

b) You can map your EJBs and EJB relationships to an already existing database schema. In this case you have to either manually create the `orion-ejb-jar.xml` file from scratch, which is a rather error prone process, or follow the following recommended steps:

1. Deploy your EJB module with the `ejb-jar.xml` configured as described in previous chapters of this document, but without the `orion-ejb-jar.xml` deployment descriptor. Note: Make sure you disable the automatic table generation feature by setting the `autocreate-tables="false"` parameter in the `<orion-application>` tag in `$OC4J_HOME/j2ee/home/config/application.xml`.

2. Copy the auto-generated `orion-ejb-jar.xml` file located in `$OC4J_HOME/j2ee/home/application-deployments/`*<application>*`/`*<ejb-module>* to your development environment.

3. Modify the `orion-ejb-jar.xml` file based on your database schema.

4. Redeploy the application, this time with the modified `orion-ejb-jar.xml` file included in the EJB archive (in the `META-INF` directory, along with the `ejb-jar.xml` file.)

Let's take a closer look at how we need to modify the `orion-ejb-jar.xml` in our particular example.

MAPPING EXAMPLE

In our case, we want to create a one-to-many, bi-directional relationship between the Department EJB (`DeptBean`) and the Employee EJB (`EmpBean`). Assuming that our existing schema is designed such that the EMP table has a foreign key (FK) pointing to the primary key (PK) of the DEPT table, we can actually bypass use of an association table based on the PK-FK relationship defined between both tables. Please refer to the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide* for more details on association tables, PK and FK constraints, and more.

The <entity-deployment> tag

Principally all the mapping related information in `orion-ejb-jar.xml` can be found encapsulated in the `<entity-deployment>` element, along other data.

An extract of the corresponding XML for our example is shown below (Please note that only relationship relevant data is shown):

```
<entity-deployment name="DeptBean" … location="DeptBean" … table="DEPT" …>
…
 <cmp-field-mapping name="employees">
  <collection-mapping table="EMP">
   <primkey-mapping>
```

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

```
    <cmp-field-mapping name="deptNo"
     persistence-name="DEPTNO" />
   </primkey-mapping>
   <value-mapping type="package.EmpLocal">
    <cmp-field-mapping name="employees">
     <entity-ref home="EmpBean">
      <cmp-field-mapping name="empNo"
       persistence-name="EMPNO" />
     </entity-ref>
    </cmp-field-mapping>
   </value-mapping>
  </collection-mapping>
 </cmp-field-mapping>
…
</entity-deployment>
<entity-deployment name="EmpBean" … location="EmpBean" … table="EMP" …>
…
 <cmp-field-mapping name="dept">
  <entity-ref home="DeptBean">
   <cmp-field-mapping name="deptNo"
    persistence-name="DEPTNO" />
  </entity-ref>
 </cmp-field-mapping>
…
</entity-deployment>
```

Just as the `<ejb-relation>` tag XML snippet, this snippet might look a bit daunting at first, so let's go ahead and also break it down into each tag.

<cmp-field-mapping>

This tag maps CMP and CMR (relationship) fields. In the case of a CMR field, it will either contain a `<entity-ref>` tag for one-to-one mappings or a `<collection-mapping>` tag for 1:M, M:1, and M:N mappings. The `name` attribute specifies the CMP/CMR field to be mapped. The `persistence-name` attribute identifies the database column.

<collection-mapping>

This element maps the "many" side of a relationship. In our one-to-many, bi-directional case, the `table` attribute identifies the "many" table, meaning the EMP table. It contains two sub-elements, the `<primkey-mapping>` and the `<value-mapping>` element, which identify the primary keys for each object of the relationship.

<primkey-mapping>

The `<primkey-mapping>` element identifies the primary key mapping for the source object of the relationship, in our case the DeptBean EJB. It uses a `<cmp-field-mapping>` sub-element to do so.

<value-mapping>

This element defines the primary key mapping for the target object of the relationship, meaning the EmpBean EJB. It uses a `<cmp-field-mapping>` element with an embedded `<entity-ref>` element (see below) to do so. The `type` attribute specifies the fully qualified name (including package) of the local interface of the target EJB.

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

<entity-ref>

This element identifies the target EJB of a relationship and its primary key mapping, using a `<cmp-field-mapping>` element. The `home` attribute specifies the target EJB name, as specified in the `<ejb-name>` element in the `ejb-jar.xml` deployment descriptor.

Please refer to the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide* for more explicit mapping examples (e.g. one-to-one or many-to-many relationships, and others.) A few general concepts to keep in mind are:

- Many-to-many relationships must be represented using an association table.
- The OC4J default behavior for 1:M bi-directional relationships is to generate an association table unless the system property `associateUsingThirdTable` is set to `false` on OC4J startup (e.g. `java -DassociateUsingThirdTable=false -jar oc4j.jar`). The same effect is automatically the case if the `orion-ejb-jar.xml` deployment descriptor has the `<collection-mapping>` table set to the same table as the many table, as it is the case for the department-employee example earlier in this section.

OC4J - EJB CONTAINER FEATURES

Optimizations

The implementation of declared relationships is left to individual containers. OC4J makes several optimizations when implementing container-managed relationships. The main optimization is lazy loading. For example, take a M:N relationship. If the container does not implement any form of optimization you could very easily load the contents of both tables into memory by accident. Say you have two entities, A and B and the container provides no lazy loading option. If you load in one object from A it could have multiple references to elements in B. Therefore you have to load in all of the related instances of B. However each B object could have multiple A's associated with it, each of which could have multiple B's and so on. You can very quickly run out of memory and every time you look up one object, you could end up making multiple queries to load each related object.

Lazy loading is a means of circumventing this potential issue. Going back to the A-B example, with lazy loading the container loads the required element of A. The container then loads in the primary keys of each element in B related to A and no more. If one of the related B elements is required, a `findByPrimaryKey` is used to load the required element when and only when it is required. This saves the container loading in multiple entities therefore saving time and resources.

Locking

When an entity EJB is instantiated by the container it can be locked, depending on the locking mode specified, e.g., pessimistic or optimistic. If a parent EJB is locked it has the potential to lock all of its children as well. This could cause a large amount of locking and hence increase transaction times waiting for the lock to be released or possibly other negative side effects like deadlocks.

OC4J does not initially have to lock the child records even if the parent is locked. Due to the lazy loading optimization above, the child would have to be instantiated to update it and this would enable OC4J to lock the object at that time. This behavior is not affected by any other locking behavior specified in the container. Child records are only locked when the child is lazily loaded.

ORACLE JDEVELOPER FEATURES

JDeveloper's comprehensive features cover all aspects of EJB development from conception to implementation. EJB support includes modeling (see Figure 1) and wizard-based development of Session, Entity and Message Drive Beans, the ability to add, edit, and delete EJBs along with their properties using the EJB Module Editor (see Figure 2), reverse-engineering of database tables and foreign key relationships

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

as CMP Entity Beans and Container Managed Relationships, the ability to test EJBs locally in the IDE, verify the EJBs for deployment errors and inconsistencies using EJB Verifier and generation of standard EJB deployment archives.

The following figure shows a UML representation with CMR relationships reverse engineered from the database using JDeveloper's extensive EJB modeling capabilities.

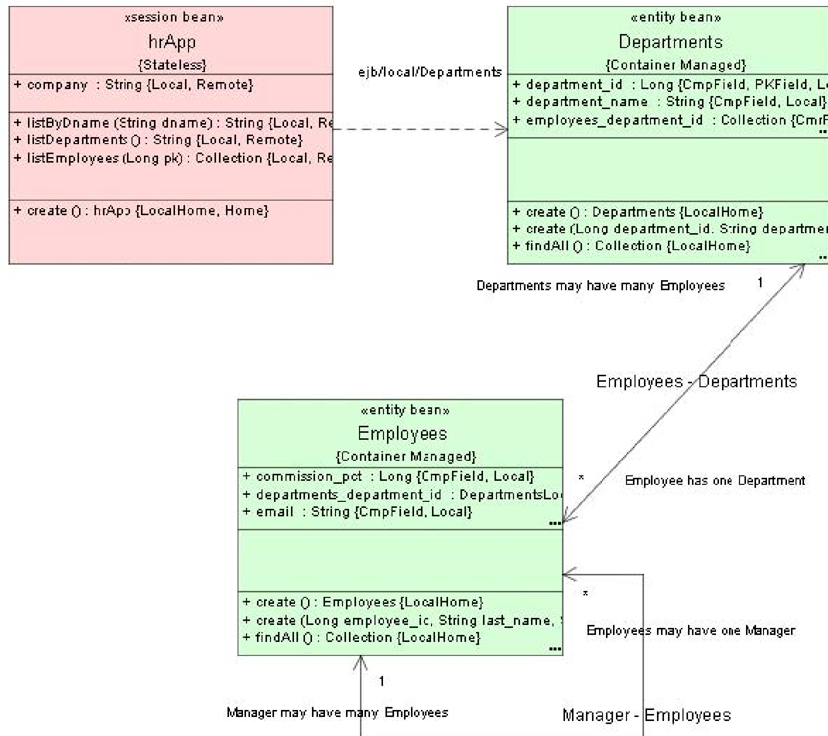

Figure 1 - EJB UML Model with CMR relationships

The next figure illustrates the EJB Module Editor that provides a common user interface for editing all the EJBs in the ejb-jar.xml deployment descriptor. The Relationships panel in the editor lets developers add, edit and delete the CMR relationships between Entity beans in an extremely intuitive way.

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0

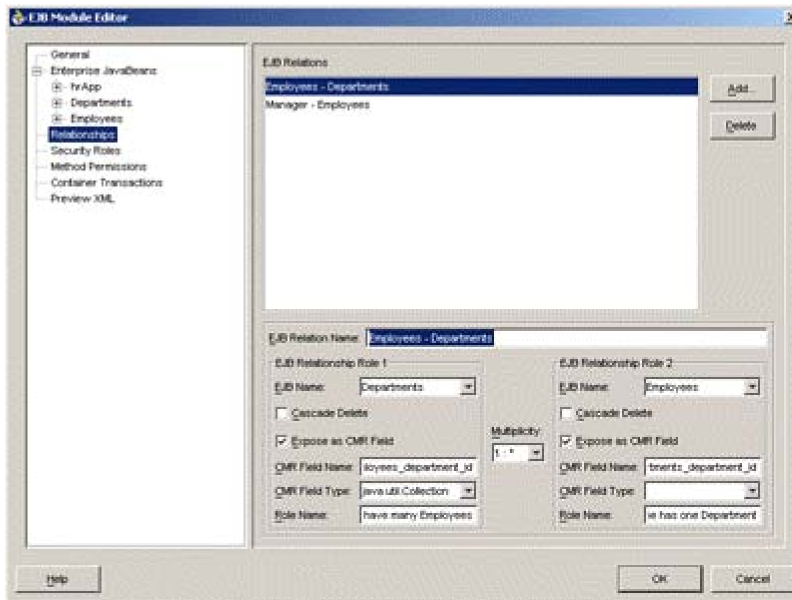

Figure 2 - EJB Module Editor

Overall JDeveloper allows the developer to concentrate on the business and development problems at hand rather than spending unnecessary time dealing with the complexity of deployment descriptors.

CONCLUSION

Modeling, developing and maintaining relationships between EJBs has been extremely difficult with EJB 1.1. As of EJB 2.0 this has been addressed. This document provides a short introduction into EJB 2.0 CMR and explains how this new technology can be used in OC4J in order to make modeling EJB relationships easy. Please refer to the *Oracle9iAS Containers for J2EE Enterprise JavaBeans Developer's Guide* for more detailed documentation regarding EJB development in OC4J.

Copyright 2003 MindTelligent, Inc. All Rights Reserved. MindTelligent Training Services. Revision A.2.0