matena8.fm Page 243 Tuesday, January 7, 2003 4:34 PM

CHAPTER 8

Entity Bean Application Example

THIS chapter uses an example of a distributed application to illustrate how enterprise applications use entity beans to model business entities. The example application uses entity beans to store the persistent state of the enterprise application. The chapter also shows how organizations develop the respective components of the application and how, ultimately, the customer deploys the entire application. The example application illustrates

- **The implementation of several entity beans to highlight the various issues in managing persistence.** Entity beans use various styles to implement their persistence; thus, we illustrate the use of both CMP and BMP. We focus on the CMP programming model defined in the EJB 2.0 and 2.1 specifications. The example shows how to construct an abstract schema consisting of multiple entity beans related through their local interfaces, using container-managed relationships.
- **The use of local interfaces to develop lightweight entity beans.** The example contains several entity beans that have local interfaces. Local interfaces allow clients to access the entity beans in an efficient manner and also avoid the complexities of programming distributed objects.
- **The use of EJB QL to develop portable queries on the application's persistent state.** The example illustrates the use of EJB QL queries for find methods that can be directly invoked from clients, as well as for select methods, which are used internally by the bean.
- **The use of home business methods**. We show how home business methods

matena8.fm Page 244 Tuesday, January 7, 2003 4:34 PM

are used to model aggregate operations that do not operate on a specific entity bean instance.

- **The techniques for developing applications for different customers with different operational environments.** An ISV would like to sell the application to as broad a range of customers and operational environments as possible. Our example illustrates how the ISV (1) uses entity beans with CMP to integrate its application with the customer's existing applications and database and (2) uses remote interfaces to allow flexibility in the deployment of the application with respect to client applications.
- **The design issues for remote interfaces.** The developer should design the remote interface so that its methods take into account the costs of distribution.
- **The techniques for caching an entity object's persistent state.** The example illustrates how to use the instance variables of an entity bean class, along with the `ejbLoad` and `ejbStore` methods, to cache the entity object's persistent state.
- **The correct approach that a client application, such as `EnrollmentBean`, takes to use the entity bean client-view API.**
- **The techniques for "subclassing" an entity bean with CMP to create an entity bean with BMP.** The subclass implements the data access methods.
- **The packaging of enterprise beans into J2EE standard files.** The example illustrates the packaging of enterprise beans and their dependent parts into the standard ejb-jar file and the J2EE enterprise application archive file (`.ear` file).
- **The parts of an application that do *not* have to be developed.** The example code is also interesting in what it does not include— namely, database access code in the CMP entity beans and no transaction or security management–related code. The deployment descriptors describe declaratively the transaction and security requirements for entity beans. Transaction management is described in Chapter 10, Understanding Transactions; security management, in Chapter 11, Managing Security.

This chapter begins with the description of the problem. Then, to give you a feel for the scope of the application, the application components are described from a high level, followed by detailed information on each part of the application, from the perspective of the vendor that developed the part.

matena8.fm Page 245 Tuesday, January 7, 2003 4:34 PM

8.1 Application Overview

Our example application illustrates the development and deployment of an enterprise application that consists of components developed by multiple vendors.

8.1.1 Problem Description

The example entity bean application implements a benefits self-service application. An employee uses this application to select and enroll in the benefits plans offered by the company. From the end-user perspective, the application is identical to the benefits application built using session beans, described in Chapter 4, Working with Session Beans. However, the design of the two applications differs as follows:

- The application uses entity beans with CMP to manage their persistent state. Because the entity beans use CMP, the benefits application contains no explicit database access code, and the amount of code in each entity bean is reduced drastically. Using CMP also facilitates storing the application's persistent state in a wide variety of persistence stores, including relational databases.
- Wombat Inc. developed the benefits application. Wombat is an ISV that specializes in the development of benefits applications used by enterprises. Because Wombat wants to sell its application to as many different enterprises as it can, its application must work in a myriad of operational environments. In contrast, Star Enterprise's IT department developed the application illustrated in Chapter 4. Because it was intended to be used only within Star Enterprise's own environment, that application was developed with no regard for the application's portability to other operational environments.
- The application described in this chapter allows dynamic changes to the configuration of the available medical and dental plans. For example, a benefits administrator at Star Enterprise can add and remove medical and dental plans to the benefits application. In contrast, the application in Chapter 4 requires redeployment to change the configuration of the available plans.

8.1.2 Main Parts of the Application

The example application presented here consists of multiple enterprise beans, Web applications, and databases. Typical for an application such as this, some parts

matena8.fm Page 246 Tuesday, January 7, 2003 4:34 PM

already existed at Star Enterprise, whereas outside organizations developed the other parts. Figure 8.1 illustrates the logical parts of the application.

The application consists of two principal parts, which come from two sources:

1. The preexisting employee and payroll databases and PayrollEJB bean in the Star Enterprise operational environment
2. The Wombat benefits application, which consists of multiple enterprise beans and Web applications

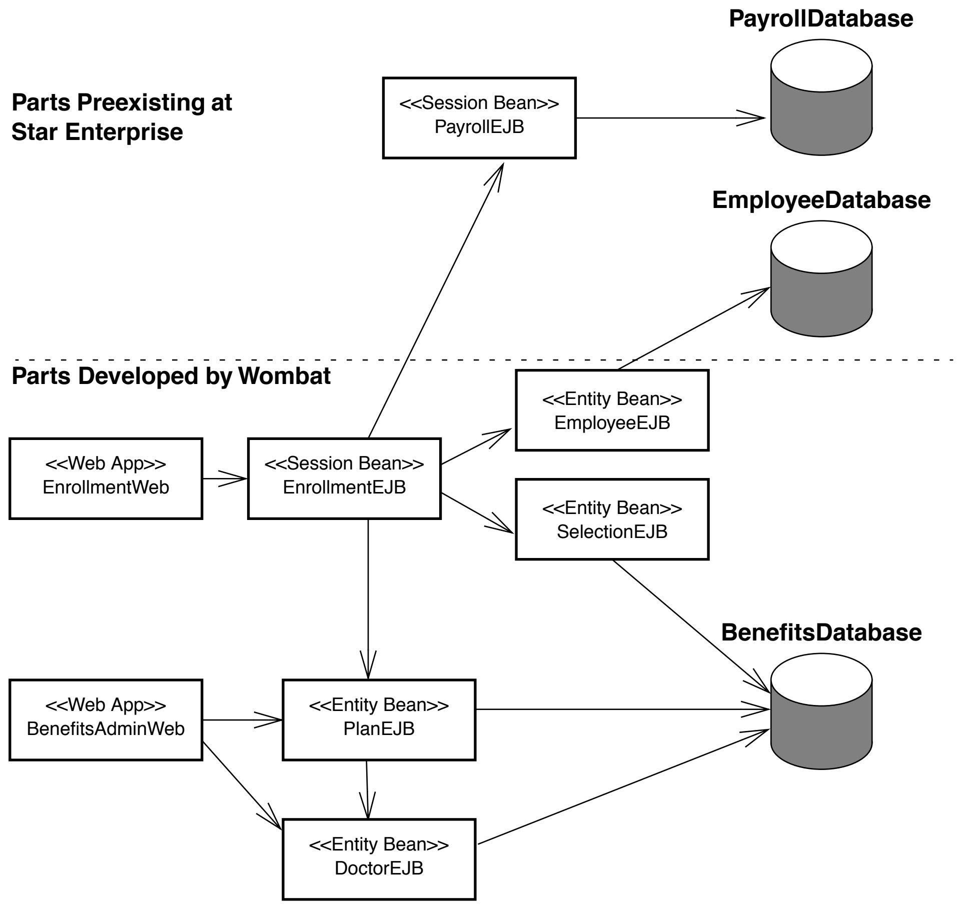
PayrollDatabase
Parts Preexisting at Star Enterprise
<<Session Bean>> PayrollEJB
EmployeeDatabase
Parts Developed by Wombat
<<Entity Bean>> EmployeeEJB
<<Web App>> EnrollmentWeb
<<Session Bean>> EnrollmentEJB
<<Entity Bean>> SelectionEJB
BenefitsDatabase
<<Web App>> BenefitsAdminWeb
<<Entity Bean>> PlanEJB
<<Entity Bean>> DoctorEJB

Figure 8.1 Logical Parts of the Entity Bean Benefits Application

matena8.fm Page 247 Tuesday, January 7, 2003 4:34 PM

Prior to the deployment of Wombat's benefits application, Star Enterprise used the `EmployeeDatabase`, `PayrollDatabase`, and PayrollEJB parts. These parts pertain to the following aspects of Star Enterprise's business:

- `EmployeeDatabase` contains information about Star Enterprise employees.
- `PayrollDatabase` contains payroll information about Star Enterprise.
- PayrollEJB is a stateless session bean that provides nonpayroll applications with secure access to the payroll database. Nonpayroll applications, including Wombat's benefits application, use PayrollEJB as the payroll integration interface.

Wombat, an ISV, has implemented the bulk of the benefits application. Wombat develops multiple Web applications and enterprise beans, as follows:

- EnrollmentWeb is a Web application that implements the presentation logic for the benefits enrollment process. A Wombat customer's employees, such as Star Enterprise employees when the application is deployed at Star Enterprise, access EnrollmentWeb via a browser.
- BenefitsAdminWeb is a Web application that implements the presentation logic for business processes used by the customer's benefits administration department. The benefits administration department uses BenefitsAdminWeb, for example, to customize the portfolio of plans offered to the employees.
- EnrollmentEJB, a stateful session bean that implements the benefits enrollment business process, uses several entity beans to perform its function.
- EmployeeEJB, an entity bean that encapsulates access to the customer's—Star Enterprise, in this example—employee information, uses CMP, and its main role is to allow deployment binding with the customer's employee database.
- SelectionEJB, PlanEJB, and DoctorEJB are entity beans that encapsulate the benefits selections, medical and dental plan information, and physician information, respectively.
- `BenefitsDatabase` stores the information used by the SelectionEJB, PlanEJB, and DoctorEJB entity beans.

matena8.fm Page 248 Tuesday, January 7, 2003 4:34 PM

8.1.3 Distributed Deployment

The EJB architecture provides the power and flexibility necessary to enable distributed deployment of various components in an enterprise application. Although it is possible to deploy all the Web and EJB components in a single J2EE server and to aggregate all the databases into a single database, the traditional division of "information ownership" by multiple departments within a large enterprise leads to a distributed deployment scenario illustrated in Figure 8.2. Star Enterprise has deployed the benefits application across multiple servers, including six servers within its own enterprise intranet.

The benefits department has deployed the EnrollmentWeb and BenefitsWeb Web applications on the Benefits Web server, and the enterprise beans EnrollmentEJB, SelectionEJB, EmployeeEJB, PlanEJB, and DoctorEJB on the Benefits App server. `BenefitsDatabase` is stored on the Benefits Database server. The enterprise bean PayrollEJB is deployed on the Payroll App server, which in turn provides access to the `PayrollDatabase` server. `EmployeeDatabase` is stored on the Human Resources (HR) Database server.

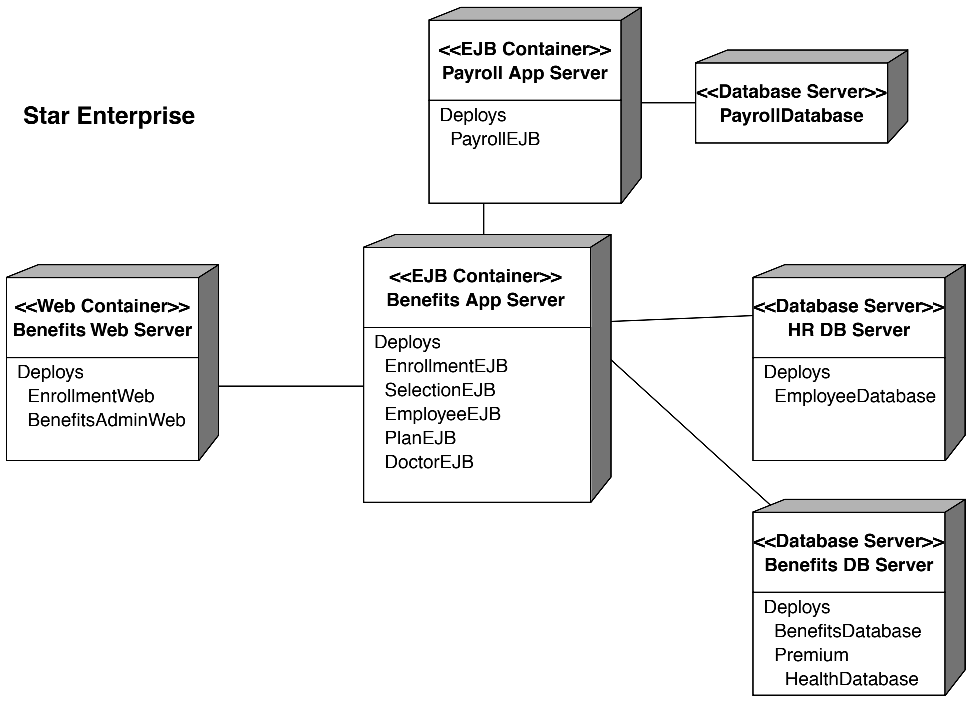

Figure 8.2 Benefits Application Deployment

matena8.fm Page 249 Tuesday, January 7, 2003 4:34 PM

8.2 Parts Developed by Wombat

Wombat Inc. is an ISV specializing in the development of applications for enterprises to use to administer benefits plans, such as medical and dental insurance plans. One Wombat application is a Web-based self-service Benefits Enrollment application. Employees of an enterprise use the application to make selections from multiple medical and dental plans offered to them by their employer.

Wombat's goal is to develop a single, generic Benefits Enrollment application and sell it to many customer enterprises. The Benefits Enrollment application is not an isolated application: It uses data provided by other applications or databases that exist in the customers' operational environment. This presents a challenge for Wombat: Every customer is likely to have a different implementation of the application or data with which the Benefits Enrollment application needs to integrate. For example, the enrollment application needs access to a database that contains information about employees, as well as access to the payroll system so that it can update benefits-related paycheck deductions. In addition, the enrollment application needs to have access to the plan-specific information provided by the insurance companies.

8.2.1 Overview of the Wombat Parts

Wombat develops the Web applications and enterprise beans, which are illustrated in Figure 8.3.

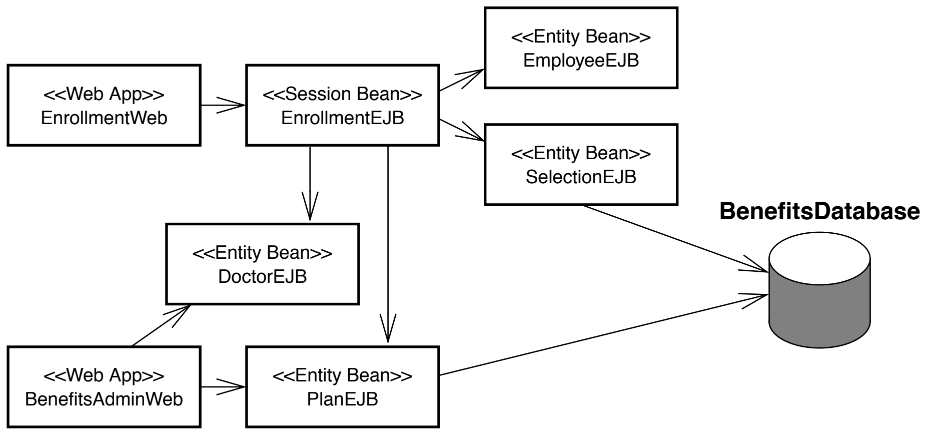

Figure 8.3 Web Applications and Enterprise Beans Developed by Wombat

matena8.fm Page 250 Tuesday, January 7, 2003 4:34 PM

Each of the following applications and enterprise beans is detailed throughout this section:

- **EnrollmentWeb**—A Web application that implements the presentation logic for the benefits enrollment process. A customer's employees use EnrollmentWeb to enroll into the offered medical and dental plans.
- **BenefitsAdminWeb**—A Web application that implements the presentation logic for business processes used by the customer's benefits administration department to configure and customize the medical and dental plans offered to the employees.
- **EnrollmentEJB**—A stateful session bean that implements the benefits enrollment business process.
- **EmployeeEJB**—An entity bean with CMP that encapsulates access to a customer's employee information so that it can accommodate the different representations of employee databases at different customer sites. CMP allows deployment binding with the customer's employee database. CMP also allows customers to implement their own database access code by writing a BMP entity bean that subclasses the CMP entity bean class.
- **SelectionEJB**—A CMP entity bean that encapsulates the benefits selections chosen by each employee.
- **PlanEJB—**A CMP entity bean that stores information about medical and dental plans and provides operations to search through plans.
- **DoctorEJB**—A CMP entity bean that stores information about physicians and dentists and provides operations to search for doctors, based on specified criteria.

The enterprise beans developed by Wombat store information in `BenefitsDatabase`. Wombat designed the `BenefitsDatabase` schema, and at deployment the customer creates the database at the customer site. Wombat also allows the customer to choose a different schema as a deployment option.

The following describe in greater detail the parts developed by Wombat.

matena8.fm Page 251 Tuesday, January 7, 2003 4:34 PM

8.2.2 EnrollmentEJB Session Bean

EnrollmentEJB is a stateful session bean that implements the benefits enrollment business process. EnrollmentEJB's home and component interfaces are the same as in the example in Chapter 4, Working with Session Beans. However, although an IT developer at Star Enterprise defined the interfaces in that chapter, Wombat defined the home and component interfaces shown in this chapter's alternative approach. Code Example 8.1 shows the EnrollmentEJB's home interface definition:

```
package com.wombat.benefits;

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

public interface EnrollmentHome extends EJBHome {
   Enrollment create(int emplnum) throws RemoteException,
   CreateException, EnrollmentException;
}
```

Code Example 8.1 The `EnrollmentHome` Home Interface Defined by Wombat

Code Example 8.2 shows the definition of the EnrollmentEJB's remote component interface:

```
package com.wombat.benefits;

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

public interface Enrollment extends EJBObject {
   EmployeeInfo getEmployeeInfo()
          throws RemoteException, EnrollmentException;
   Options getCoverageOptions()
          throws RemoteException, EnrollmentException;
   void setCoverageOption(int choice)
          throws RemoteException, EnrollmentException;
   Options getMedicalOptions()
          throws RemoteException, EnrollmentException;
   void setMedicalOption(int choice)
          throws RemoteException, EnrollmentException;
   Options getDentalOptions()
          throws RemoteException, EnrollmentException;
```

matena8.fm Page 252 Tuesday, January 7, 2003 4:34 PM

```
        void setDentalOption(int choice)
                throws RemoteException, EnrollmentException;
        boolean isSmoker()
                throws RemoteException, EnrollmentException;
        void setSmoker(boolean status)
                throws RemoteException, EnrollmentException;
        Summary getSummary()
                throws RemoteException, EnrollmentException;
        void commitSelections()
                throws RemoteException, EnrollmentException;
    }
```

Code Example 8.2 The `Enrollment` Remote Interface Defined by Wombat

The `Enrollment` and `EnrollmentHome` interfaces presented in this chapter are similar to those presented in Chapter 4. However, the interfaces presented here are implemented as remote interfaces, extending `javax.ejb.EJBObject` and `javax.ejb.EJBHome`, respectively. (The interfaces in Chapter 4 extended `javax.ejb.EJBLocalObject` and `javax.ejb.EJBLocalHome`.)

Wombat's developers consciously decided to make these interfaces remote instead of local when they designed the EnrollmentEJB entity bean. Wombat chose to use remote interfaces because they provide a location-independent client view. As a result, Wombat's customers, such as Star Enterprise, can deploy the EnrollmentEJB in a server other than the EnrollmentWeb application, thus allowing more flexible deployment choices. Moreover, the Benefits Enrollment application can be accessed by "rich client" applications and other enterprise applications, regardless of their location on the network. Such deployment and client access flexibility was critical for Wombat to achieve its goal of selling its benefits application to as many customers as possible.

However, the use of remote interfaces introduces new issues that need to be reflected in the design of the interfaces themselves:

- **Remote access is expensive.** Remote calls require more resources and use more overhead: They may traverse a network, they require client-side and server-side software to provide the distributed object invocation infrastructures, and they require arguments to be "deep copied" even when the client is in the same Java virtual machine. Hence, the cost of a remote call may be a

matena8.fm Page 253 Tuesday, January 7, 2003 4:34 PM

hundred to a thousand times more than the cost of a local call.

Accordingly, the `Enrollment` remote interface must be designed to avoid a large number of fine-grained calls from the client and instead use a small number of large- or coarse-grained calls. Hence, the `Enrollment` interface defines the `getEmployeeInfo` method, which returns all employee information to the client in one call, using the serializable value class `EmployeeInfo`. Similarly, the `getSummary` method returns all benefits summary information in one method call, using the serializable value class `Summary`.

- **Remote access can lead to errors that are not encountered in local calls.** Because remote calls make requests on another computer over the network, any number of reasons can cause the remote call to fail, including network problems, resource/memory limits on the target computer, software errors, and so forth. The client must always be prepared to receive such exceptions and handle them if necessary. (For example, a Web component might handle an error by sending a meaningful error page to the browser.) To help identify these remote errors, the methods of the `Enrollment` and `EnrollmentHome` interface throw `java.rmi.RemoteException` as required for all remote interfaces.
- **State cannot be shared directly.** Because remote calls involve copying all arguments and return values, both the bean and the client code cannot assume that they have a reference to the actual Java object used by the other side. Thus, the bean and the client never share any state directly; they can have only copies of each other's state.
- **Remote references behave slightly differently from local references**. In particular, casting a remote reference to a derived type requires the use of the `javax.rmi.PortableRemoteObject.narrow` operation. Thus, the EnrollmentEJB's client—the EnrollmentWeb application—must use this `narrow` operation to cast `EJBHome` reference objects that are obtained through JNDI lookup to the `EnrollmentHome` interface.

`EnrollmentBean` Implementation Class

The implementation of the `EnrollmentBean` session bean class is similar to the implementation illustrated in Chapter 4. However, there are some key differences, as follows:

- EnrollmentEJB in Chapter 4 uses command beans to access the employee's database. EnrollmentEJB in this chapter uses the EmployeeEJB entity bean to encapsulate access to the employee information. Because the EmployeeEJB

matena8.fm Page 254 Tuesday, January 7, 2003 4:34 PM

entity bean is implemented with CMP, the deployer can bind the EmployeeEJB bean with the customer's employee database in a standard way.

- EnrollmentEJB in Chapter 4 uses command beans to access the benefits selections in `BenefitsDatabase`. EnrollmentEJB in this chapter uses the SelectionEJB entity bean to encapsulate the access to the employees' current selections. In addition, because Wombat provides a CMP version of the SelectionEJB entity bean—the `SelectionBean` class—a customer can customize the format in which the selections are stored or even store them in a nonrelational database.
- EnrollmentEJB uses the PlanEJB entity bean to access the medical and dental plans offered to the employees. PlanEJB allows the medical and dental plan information to be dynamically updated by Wombat's customers through the BenefitsAdmin Web application. The EnrollmentEJB implementation in Chapter 4 relied on Java classes that hard-coded the plan information. This hard-coding prevented the dynamic update of plan information after the benefits application was deployed.
- EnrollmentEJB uses a command bean to update an employee's payroll with the deduction based on his or her benefits choices. Because each customer of Wombat may have a different payroll system, Wombat defines only a command bean interface: `DeductionUpdateBean`. The actual implementation class for this bean is provided by Wombat's customer, such as Star Enterprise. The class name is then set in the EnrollmentEJB's environment by the deployer.

Code Example A.4 on page 382 illustrates the source code for the `EnrollmentBean` session bean class as it has been implemented for the example in this chapter. (Note that this implementation of `EnrollmentBean` differs from that in Chapter 4.)

Using Entity Bean Client-View Interfaces

The `EnrollmentBean` class illustrates how applications typically use the entity bean client-view interfaces. Recall that EnrollmentEJB is a client of the EmployeeEJB, SelectionEJB, and PlanEJB entity beans.

For example, let's look at how EnrollmentEJB uses the SelectionEJB entity bean. In the `ejbCreate` method, note that EnrollmentEJB uses the `findByPrimaryKey` method to look up an existing `Selection` object, as follows:

```
selection = selectionHome.findByPrimaryKey(
                     new Integer(employeeNumber));
```

matena8.fm Page 255 Tuesday, January 7, 2003 4:34 PM

After obtaining an object reference to the `Selection` object, EnrollmentEJB invokes a business method on the object. Here, it invokes the `Selection` object's `getCopy` method to read the current benefits selection values:

```
selCopy = selection.getCopy();
```

In the `commitSelections` method, EnrollmentEJB either creates a new `Selection` object by invoking the `create` method on the `SelectionHome` interface, or it updates the existing `Selection` object by invoking the `updateFromCopy` business method on the `Selection` object, as follows:

```
if (recordDoesNotExist) {
    selection = selectionHome.create(selCopy);
    recordDoesNotExist = false;
} else {
    selection.updateFromCopy(selCopy);
}
```

Note that EnrollmentEJB does not need to remove `Selection` objects. If it did, however, it would use the following code fragment:

```
selection.remove();
```

Alternatively, EnrollmentEJB could use the `SelectionHome` interface to remove a `Selection` object identified by its primary key:

```
selectionHome.remove(new Integer(employeeNumber));
```

EnrollmentEJB uses the other entity beans in much the same manner.

8.2.3 EmployeeEJB Entity Bean

EmployeeEJB is an entity bean that uses the container-managed persistence model defined in the EJB 2.0 and 2.1 specifications. EmployeeEJB provides an object-oriented view of the employee data used by the Benefits Enrollment application. The main role of this entity bean is to allow the integration between the benefits application and the customer's employee data.

Because Wombat does not impose rules about how a customer stores the information about its employees, customers must have the means to integrate the application with their employee data. Wombat uses the CMP mechanism to allow the deployer to bind EmployeeEJB with an existing employee database.

matena8.fm Page 256 Tuesday, January 7, 2003 4:34 PM

Wombat's customers have two choices while deploying the EmployeeEJB bean:

- **Using an object-relational mapping tool**—A customer can use an object-relational mapping tool to create a mapping from the Employee bean's abstract persistence schema—that is, its CMP and CMR fields—to the physical database schema in the preexisting employee database. In this case, the object-relational mapping tool generates the database access code.
- **Using bean-managed persistence**—Alternatively, a customer could develop a BMP entity bean that subclasses the CMP `EmployeeBean` class. The customer then would write the database access code.

However, Wombat designs the `Employee` and `EmployeeHome` interfaces to meet the needs of the Benefits Enrollment application.

EmployeeEJB's Primary Key

Wombat uses the employee number as the primary key for the EmployeeEJB entity bean. Its type is the class `java.lang.Integer`. Note that it would be an error if the employee number were the Java primitive `int`, because the EJB specification requires that the primary key type for an entity bean be a Java class. Furthermore, this requirement implies that primitive types that are not serializable Java classes cannot be used directly for the primary key type.

`Employee` Local Interface

Because EmployeeEJB is accessed only from EnrollmentEJB and both are packaged in the same benefits application, EmployeeEJB defines only local interfaces. Code Example 8.3 shows the `Employee` local interface definition:

```
public interface Employee extends EJBLocalObject {
    Integer getEmployeeNumber();
    String getFirstName();
    String getLastName();
    Date getBirthDate();
}
```

Code Example 8.3 The `Employee` Local Interface

matena8.fm Page 257 Tuesday, January 7, 2003 4:34 PM

The `Employee` local interface defines methods that allow its client, the EnrollmentEJB bean, to access each employee's employee number, first and last names, and date of birth. Note that these methods are fine-grained: One method returns one piece of data rather than a single method returning an aggregation of all the pieces of data. Typically, the EnrollmentEJB bean would make several calls on the `Employee` interface to eventually retrieve all fields for an employee. Making multiple method calls, with each method call retrieving individual data fields, is effective because local EJB calls are very efficient in typical application server implementations.

Note that the `Employee` local interface does not define any methods that allow clients to modify `EmployeeBean`'s state. This implies that the EmployeeEJB is a *read-only* bean. Most EJB containers are designed to efficiently manage such read-only beans.

The `EmployeeHome` Home Interface

Code Example 8.4 shows the `EmployeeHome` interface definition:

```
package com.wombat.benefits;
import javax.ejb.*;

public interface EmployeeHome extends EJBLocalHome {
    // find methods
    Employee findByPrimaryKey(Integer employeeNumber)
            throws FinderException;
}
```

Code Example 8.4 The `EmployeeHome` Home Interface

The `EmployeeHome` interface defines only the mandatory `findByPrimaryKey` method. It defines no create methods, because the Wombat Benefits Enrollment application does not need to create new employee objects in the customer databases.

The `EmployeeBean` Entity Bean Class

The `EmployeeBean` class illustrates how simple it is to develop an entity bean with CMP. Note that the entity bean contains no database operations. Section 8.3.1, The Employee Database and Deployment of EmployeeEJB, on page 286 explains how

matena8.fm Page 258 Tuesday, January 7, 2003 4:34 PM

the deployer binds the container-managed fields with the columns of the preexisting `EmployeeDatabase` at Star Enterprise. Note also that the entity bean class contains no implementations of any find methods, although the home interface defines the `findByPrimaryKey` method. Because EmployeeEJB is an entity bean with CMP, the EJB container supplies the implementations of its find methods. Code Example 8.5 shows the source code of the `EmployeeBean` entity bean class:

```
package com.wombat.benefits;

import javax.ejb.*;
import java.util.Date;
import com.wombat.AbstractEntityBean;

public abstract class EmployeeBean extends AbstractEntityBean {
    // Container-managed fields

    public abstract Integer getEmployeeNumber(); // primary key field
    public abstract void setEmployeeNumber(Integer n);

    public abstract String getFirstName();
    public abstract void setFirstName(String s);

    public abstract String getLastName();
    public abstract void setLastName(String s);

    public abstract Date getBirthDate();
    public abstract void setBirthDate(Date d);
}
```

Code Example 8.5 `EmployeeBean` Class Implementation

The `EmployeeBean` class, essentially a pure data object that models a business entity, is one of the simplest examples of a CMP entity bean class. The `Employee-Bean` class contains only CMP fields, and these fields define the persistent state of the employee. The class has no create methods and no relationships with other beans.

The `AbstractEntityBean` class, which `EmployeeBean` imports, is a utility class developed by Wombat. This class contains default implementations of the

matena8.fm Page 259 Tuesday, January 7, 2003 4:34 PM

`javax.ejb.EntityBean` methods. The class was developed to make it easy to implement entity bean classes that do not need to use methods in the `EntityBean` interface.

There are some important things to note about the implementation of an entity bean class, particularly concerning container-managed fields and the primary key. The `employeeNumber`, `lastName`, `firstName`, and `birthDate` fields are container-managed fields of the entity bean class and are declared as such in the EmployeeEJB's deployment descriptor. These fields are also *virtual* fields. Virtual fields are represented in the `EmployeeBean` class as pairs of get and set methods. The method names are derived from the CMP field names and follow the Java-Beans design pattern. The EJB specification mandates that the get and set methods be defined as `public`, even though a client program never directly accesses them. The container uses these methods to synchronize the content of the fields they represent with the information in the database. These methods must be `public` so that the container can move the data between the fields and the database to keep them synchronized.

8.2.4 SelectionEJB Entity Bean

The SelectionEJB entity bean stores an employee's benefits selections, using CMP. By using CMP, the bean can be developed with no database dependences: Wombat's customers can use any database of their choice, including nonrelational databases, to store the persistent state of the bean.

SelectionEJB's Primary Key

Wombat uses the employee number as the primary key for the SelectionEJB entity bean. The type of the primary key is `java.lang.Integer`.

`Selection` Local Interface

Wombat designed SelectionEJB's local interface to meet the needs of its client, the EnrollmentEJB session bean. The local interface uses the `SelectionCopy` value object to pass the information between the SelectionEJB entity bean and its client, through its `getCopy` and `updateFromCopy` business methods. Code Example 8.6 shows the `Selection` interface definition:

```
package com.wombat.benefits;
```

matena8.fm Page 260 Tuesday, January 7, 2003 4:34 PM

```
import javax.ejb.*;

public interface Selection extends EJBLocalObject {
   SelectionCopy getCopy()
      throws SelectionException;
   void updateFromCopy(SelectionCopy copy)
      throws SelectionException;
}
```

Code Example 8.6 The `Selection` Interface Definition

The `Selection` interface defines methods that its clients use to obtain and update an employee's benefits selection. In the Wombat benefits application, the EnrollmentEJB session bean is the client of SelectionEJB. EnrollmentEJB uses the `Selection` interface's `getCopy` method to obtain a transient copy of the employee's benefits selection. The `SelectionCopy` object stores this data in memory while the employee selects the benefits options. When the employee commits those selections, EnrollmentEJB uses the `updateFromCopy` method to write the contents of the `SelectionCopy` object to the persistent fields of the SelectionEJB entity bean, thus saving the selection information to the database. Code Example 8.7 shows the code for the `SelectionCopy` class:

```
package com.wombat.benefits;

import com.wombat.plan.Plan;

public class SelectionCopy {
   private Employee employee;
   private int coverage;
   private Plan medicalPlan;
   private Plan dentalPlan;
   private boolean smokerStatus;

   public Employee getEmployee() { return employee; }
   public int getCoverage() { return coverage; }
   public Plan getMedicalPlan() { return medicalPlan; }
   public Plan getDentalPlan() { return dentalPlan; }
   public boolean isSmoker() { return smokerStatus; }
```

matena8.fm Page 261 Tuesday, January 7, 2003 4:34 PM

```
        public void setEmployee(Employee v) { employee = v; }
        public void setCoverage(int v) { coverage = v; }
        public void setMedicalPlan(Plan v) { medicalPlan = v; }
        public void setDentalPlan(Plan v) { dentalPlan = v; }
        public void setSmoker(boolean v) { smokerStatus = v; }
    }
```

Code Example 8.7 The `SelectionCopy` Value Object

You might wonder why an entity bean with a local client view uses a value object such as `SelectionCopy`. Value objects are a useful way to avoid fine-grained data access, something you might want to consider when access is across a distributed network. A bean can also use a value object to temporarily hold information that the bean collects in the course of its processing but that the bean may not ultimately save to a database.

`SelectionHome` Home Interface

Code Example 8.8 shows the definition for the `SelectionHome` home interface:

```
package com.wombat.benefits;

import javax.ejb.*;

public interface SelectionHome extends EJBLocalHome {
Selection create(SelectionCopy copy)
      throws CreateException;

Selection findByPrimaryKey(Integer emplNumber)
      throws FinderException;
}
```

Code Example 8.8 The `SelectionHome` Home Interface

Note how the `SelectionHome` interface uses the `SelectionCopy` object as the argument of the `create` method. A client uses this method to create an entity

matena8.fm Page 262 Tuesday, January 7, 2003 4:34 PM

object that stores an employee's benefits selections from a copy of the information passed by the client.

The `SelectionHome` interface also defines the mandatory `findByPrimaryKey` method. This method finds the `Selection` object by using the primary key, which is the employee number.

`SelectionBean` Entity Bean Class

Code Example 8.9 shows the abstract schema portion of the `SelectionBean` entity bean implementation:

```
public abstract class SelectionBean extends AbstractEntityBean {
   // Container-managed persistence fields
    public abstract Integer getEmployeeNumber(); // primary key field
    public abstract void setEmployeeNumber(Integer n);

    public abstract int getCoverage();
    public abstract void setCoverage(int c);

    public abstract boolean getSmokerStatus();
    public abstract void setSmokerStatus(boolean s);

    // Container-managed relationship fields
    public abstract Plan getMedicalPlan();
    public abstract void setMedicalPlan(Plan p);

    public abstract Plan getDentalPlan();
    public abstract void setDentalPlan(Plan p);
    ......
}
```

Code Example 8.9 Abstract Schema for the `SelectionBean` Entity Bean Class

The abstract schema defines the bean's persistent state. (For the complete listing of the source code for the `SelectionBean` entity bean class implementation, see Code Example A.5 on page 392.)

matena8.fm Page 263 Tuesday, January 7, 2003 4:34 PM

The `SelectionBean` class has three CMP fields: `coverage`, `smokerStatus`, and `employeeNumber`. These CMP fields are declared by using pairs of set and get methods.

The `SelectionBean` class also has two container-managed relationships, represented as CMR fields: `medicalPlan` and `dentalPlan`. Like CMP fields, CMR fields are declared by using pairs of set and get methods. The argument and return value types for CMR fields are the local interfaces of the related beans. The `medicalPlan` and `dentalPlan` CMR fields provide references to different instances of the PlanEJB bean. In the deployment descriptor, these fields are declared to be many-to-one relationships because many instances of a SelectionEJB bean are associated with a single medical plan or dental plan bean instance. Also in the deployment descriptor, the `medicalPlan` and `dentalPlan` relationships are declared to be unidirectional, implying that no corresponding CMR field in the PlanEJB bean refers to a SelectionEJB instance.

Using CMR fields to manage references to other enterprise beans has two benefits:

1. It simplifies the development of the `SelectionBean` methods because they can work directly with object references rather than having to convert object references to primary keys.
2. It avoids hard-coding into the `SelectionBean` class the database representation of the relationships to the other entity beans. As a result, a deployer is free to choose how to represent the relationships in the underlying database schema.

Note that although it could have done so, the SelectionEJB class does not define a container-managed relationship with the EmployeeEJB bean. At first glance, it seems natural to define this relationship as container managed because there is a SelectionEJB instance for each EmployeeEJB instance. In addition, SelectionEJB and EmployeeEJB use the same primary key: the employee number.

However, Wombat's developers want to give their customers the flexibility to store instances of these two beans in separate databases and not be compelled to store them together in the same database. To allow customers this flexibility, Wombat's developers chose not to define the relationship between the SelectionEJB and EmployeeEJB beans as a container-managed relationship. Generally, when two beans are related using a container-managed relationship, they must be stored in the same database. Keeping related beans in the same database allows it to manage the relationship more efficiently; for example, a relational database

matena8.fm Page 264 Tuesday, January 7, 2003 4:34 PM

might use a foreign key to model the relationship. Because Wombat knows that its customers are likely to have two separate databases—an employee database managed by the human resources department and a benefits database managed by the benefits department—the developers decided to keep the EmployeeEJB and SelectionEJB beans independent. By doing so, Wombat customers can store these bean instances in separate databases or together in one database. (Although CMP allows multiple databases to be used, EJB products that permit multiple databases are still evolving.)

Let's take a closer look at the implementation of the `SelectionBean` methods. The `SelectionBeanCMP` class implements three sets of methods:

1. The business methods defined in the `Selection` local interface
2. The `ejbCreate` and `ejbPostCreate` methods that correspond to the `create` method defined in the `SelectionHome` interface
3. The container callbacks defined in the `EntityBean` interface

The `SelectionBean` class follows the EJB specification rules and does not implement the `ejbFind` methods corresponding to the find methods defined in the `SelectionHome` interface.

Business Methods

The business methods `getCopy` and `updateFromCopy` read and write the container-managed fields. The container loads and stores the contents of the container-managed fields according to the rules defined in the EJB specification. The business methods can assume that the contents of the container-managed fields are always up-to-date, even if other transactions change the underlying selection record in the database.

The code for the business methods demonstrates how an enterprise might implement simple business rules. For example, the `updateCoverage` helper method checks that the value of the coverage field is an allowed value, whereas the `updateMedicalPlan` helper method optionally checks that the value of `medicalPlan` is indeed a medical plan rather than a dental plan.

The `ejbCreate` and `ejbPostCreate` Methods

The `ejbCreate` method sets values into `SelectionBean`'s container-managed persistence fields from the values passed to it in the method parameter. In particular, the method sets the primary key by using the `setEmployeeNumber` method. Setting the

matena8.fm Page 265 Tuesday, January 7, 2003 4:34 PM

primary key establishes the identity of the bean instance. After the `ejbCreate` method completes, the container extracts the values of the container-managed fields and creates a representation of the selection object in the database. Note that `ejb-Create` returns a **`null`** value even though the return value type is declared to be the primary key type. According to the EJB 2.1 specification, the container ignores the value returned from an `ejbCreate` method of an entity bean with container-managed persistence. However, the EJB 2.1 specification requires that the type of the `ejbCre-ate` method be the primary key type to allow a subclass of the `SelectionBean` class to be an entity bean with bean-managed persistence.

The `ejbPostCreate` method sets values of the CMR fields `medicalPlan` and `dentalPlan`. The EJB 2.1 specification does not allow CMR fields to be set in the `ejbCreate` method. Thus, CMR fields should be set either in the `ejbPostCreate` method or in a business method.

Life-Cycle Methods

The `SelectionBean` class inherits most of the default `EntityBean` life-cycle method implementations from the `AbstractEntityBean` class, providing an implementation only of the `setEntityContext` method. Entity bean classes typically use the `setEn-tityContext` method to query their environment and customize their business logic as part of their initialization. `SelectionBean`'s `setEntityContext` method calls the `readEnvironment` helper method, which accesses the environment entry available with the key `java:comp/env/checkPlanType` to do the `lookup` operation in Code Example 8.10:

```
private void readEnvironment() {
    try {
        Context ictx = new InitialContext();
        Boolean val = (Boolean)ictx.lookup(
                "java:comp/env/checkPlanType");
        checkPlanType = val.booleanValue();
        employeeHome = (EmployeeHome)ictx.lookup(
                "java:comp/env/ejb/EmployeeEJB");
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}
```

Code Example 8.10 `SelectionBean`'s `readEnvironment` Method

matena8.fm Page 266 Tuesday, January 7, 2003 4:34 PM

The value of the entry parameterizes the business logic of the bean. If the value of the environment entry is `true`, the `setMedicalPlan` and `setDentalPlan` methods check that the value of the plan to be set is indeed of the expected plan type. If the value is `false`, they do not perform these checks. The application assembler sets the value of the environment entry at application assembly. Wombat made the plan type checks optional to allow the application assembler to improve performance by omitting them if the clients of SelectionEJB are known to set the plan types correctly. We added this somewhat artificial optional check to illustrate how to use the enterprise bean environment entries to parameterize the business logic at application assembly or deployment.

8.2.5 PlanEJB Entity Bean

The PlanEJB entity bean represents medical and dental plan information that is obtained from insurance providers. This entity bean has been implemented to use container-managed persistence. PlanEJB is accessed from the EnrollmentEJB bean, as well as from the benefits administration Web application. This entity bean allows client applications to obtain details of medical and dental plans, as well as to run queries on all existing plan objects. The bean also has a facility, which uses the timer service, to e-mail to administrators on a daily basis statistics of employee enrollment in each plan.

The primary key for the PlanEJB entity bean is a unique plan identifier that is a `java.lang.String` type. The benefits department assigns unique identifiers to plan instances.

`Plan` Local Interface

Code Example 8.11 shows the methods defined by the `Plan` local interface:

```
public interface Plan extends EJBLocalObject {
     // values of planType CMP field
    public final int MEDICAL_PLAN = 1;
    public final int DENTAL_PLAN = 2;

    int getPlanType() throws PlanException;
    String getPlanId();
    String getPlanName();
    double getAgeFactor();
    void setAgeFactor(double a);
```

matena8.fm Page 267 Tuesday, January 7, 2003 4:34 PM

```
    double getCoverageFactor();
    void setCoverageFactor(double c);
    double getSmokerCost();
    void setSmokerCost(double cost);

    double getCost(int coverage, int age, boolean smokerStatus)
        throws PlanException;
    void addDoctor(Doctor doctor) throws PlanException;
    boolean removeDoctor(Doctor doctor) throws PlanException;
    Collection getAllDoctors() throws PlanException;
    Collection getDoctorsByName(Doctor template)
        throws PlanException;
    Collection getDoctorsBySpecialty(String specialty)
        throws PlanException;
}
```

Code Example 8.11 The `Plan` Local Interface

The `Plan` local interface methods perform the following operations:

- The `getPlanType` method returns an integer value that indicates the type of benefits plan. The value is equal to `Plan.MEDICAL_PLAN` if the plan is a medical plan and to `Plan.DENTAL_PLAN` if the plan is a dental plan.
- The `getPlanId` method returns the unique identifier—the primary key—of the plan.
- The `getPlanName` method returns the name of the medical or dental plan.
- The `getCost` method returns the monthly premium charged by the plan provider. The premium is determined by the benefits enrollee's coverage category, age, and smoker status.
- The `set/getSmokerCost`, `set/getAgeFactor`, `set/getCoverageFactor` methods retrieve and update the smoker cost, age factor, and coverage factor of the plan.
- The `addDoctor` method adds a doctor to the plan.
- The `removeDoctor` method removes a doctor from the plan.

- The `getAllDoctors` method returns a collection of `Doctor` objects that participate in the plan. The `Doctor` class is described later.
- The `getDoctorsByName` method returns a collection of participating doctors whose names match the information in the template supplied as a method argument.
- The `getDoctorsBySpecialty` method returns all the doctors of a given specialty.

`PlanHome` Home Interface

Code Example 8.12 shows the definition of the `PlanHome` home interface:

```
public interface PlanHome extends EJBLocalHome {
    // create methods
    Plan create(String planId, String planName, int planType,
        double coverageFactor, double ageFactor, double smokerCost)
        throws CreateException;

    // find methods
    Plan findByPrimaryKey(String planID) throws FinderException;
    Collection findMedicalPlans() throws FinderException;
    Collection findDentalPlans() throws FinderException;
    Collection findByDoctor(String firstName, String lastName)
        throws FinderException;

    // home business methods
    void updateSmokerCosts(double cost) throws FinderException;
    String[] getMedicalPlanNames() throws FinderException;
    String[] getDentalPlanNames() throws FinderException;
}
```

Code Example 8.12 The `PlanHome` Home Interface

The `PlanHome` interface defines one `create` method. The benefits administration Web application uses this `create` method to add a new medical or dental plan to the benefit options provided to employees. The `PlanHome` interface also defines the find methods used by the benefits application. The find methods that can

matena8.fm Page 269 Tuesday, January 7, 2003 4:34 PM

potentially return more than one object return these objects as a Java `Collection` and are also implemented as EJB QL queries in the deployment descriptor for PlanEJB.

- The `findByPrimaryKey` method returns the `Plan` object for a given plan identifier. The plan identifier is the primary key that uniquely identifies the plan.
- The `findMedicalPlans` method returns as a `Collection` all the medical plans configured in this home interface. Each object in the returned `Collection` implements the `Plan` interface. The EJB QL query for this method is provided in PlanEJB's deployment descriptor. The query is as follows:

```
SELECT DISTINCT OBJECT(p) FROM PlanBean p WHERE p.planType = 1
```

- The `findDentalPlans` method returns all the dental plans configured in this home interface. Each object in the returned `Collection` implements the `Plan` interface. The EJB QL query for this method is as follows:

```
SELECT DISTINCT OBJECT(p) FROM PlanBean p WHERE p.planType = 2
```

- The `findByDoctor` method returns configured in this home interface all the plans that include a specified doctor in their doctors list. This find method is implemented with an EJB QL query that navigates from the PlanEJB bean to the DoctorEJB bean. The EJB QL query for the `findByDoctor` method is as follows:

```
SELECT DISTINCT OBJECT(p) FROM PlanBean p, IN(p.doctors) d
   WHERE d.firstName = ?1 AND d.lastName = ?2
```

In this query, the `FROM` clause declares an identification variable `d` whose values are DoctorEJB instances belonging to PlanEJB's one-to-many CMR field `doctors`. The `WHERE` clause restricts the set of doctors to those having the requested first and last names. The `SELECT` clause thus returns PlanEJB instances containing doctors satisfying the condition in the `WHERE` clause.

In addition, the `PlanHome` interface defines the home business methods, whose operations are not restricted to a particular bean instance. Instead, home business

matena8.fm Page 270 Tuesday, January 7, 2003 4:34 PM

methods are used to implement aggregate operations or queries on an entire set of benefits plans. The home business methods are

- **`updateSmokerCost`**—updates the smoker cost for all plans
- **`getMedicalPlanNames`**—returns an array of medical plan names
- **`getDentalPlanNames`**—returns an array of dental plan names

`PlanBean` Implementation Class

The `PlanBean` class is an abstract class that follows the requirements for an entity bean class with container-managed persistence. Code Example A.6 on page 396 lists the complete source code for the `PlanBean` entity bean class implementation.

Code Example 8.13 shows just the abstract schema for `PlanBean`. The abstract schema defines the bean's persistent state:

```
public abstract class PlanBean extends AbstractEntityBean implements
       TimedObject {
   // Container-managed persistence fields
   public abstract String getPlanId();
   public abstract void setPlanId(String s);

   public abstract String getPlanName();
   public abstract void setPlanName(String s);

   public abstract int getPlanType();
   public abstract void setPlanType(int s);

   public abstract double getCoverageFactor();
   public abstract void setCoverageFactor(double s);

   public abstract double getAgeFactor();
   public abstract void setAgeFactor(double s);

   public abstract double getSmokerCost();
   public abstract void setSmokerCost(double s);

   // container-managed relationships (CMR fields)
   public abstract Collection getDoctors();
```

matena8.fm Page 271 Tuesday, January 7, 2003 4:34 PM

```
    public abstract void setDoctors(Collection doctors);
        ...
}
```

Code Example 8.13 Abstract Schema for the `PlanBean` Entity Bean Class

`PlanBean` has six CMP fields that represent the persistent state of PlanEJB. These fields, declared in the deployment descriptor, are `planId`, `planName`, `planType`, `coverageFactor`, `ageFactor`, and `smokerCost`. In the `PlanBean` class, the CMP fields are represented as pairs of get and set methods.

`PlanBean` also has one CMR field, `doctors`, which represents a container-managed relationship to the DoctorEJB bean. This relationship is a many-to-many relationship: Each plan is associated with multiple doctors, and each doctor may participate in several plans. As a result, the `setDoctors` and `getDoctors` methods have argument and return types that are `java.util.Collection` types. The doctor/plan relationship is also declared as a bidirectional relationship. In a bidirectional relationship between two entity beans, each bean has a CMR field referencing the other bean. For the doctor/plan bidirectional relationship, PlanEJB declares a CMR field `doctors` that references DoctorEJB, which in turn has a CMR field referencing back to PlanEJB. Section 8.2.6, DoctorEJB Entity Bean, on page 279 discusses this in more detail.

Remember from the previous section that SelectionEJB declared two one-to-one relationships to PlanEJB. However, those were declared as unidirectional relationships from SelectionEJB to PlanEJB. Hence the `PlanBean` class does not declare a CMR field referencing back to SelectionEJB.

`PlanBean` has five types of methods:

- **Business methods**—`PlanBean` implements the business methods declared in the local interface.
- **Home business methods**—`PlanBean` implements the home business methods from the home interface.
- **Select methods**—`PlanBean` includes `ejbSelect` methods that are used to declare and invoke EJB QL queries from other methods of `PlanBean`.
- **`ejbTimeout` method**—`PlanBean` includes the `ejbTimeout` method, which is called when the bean's timer expires.

matena8.fm Page 272 Tuesday, January 7, 2003 4:34 PM

- **Life-cycle methods**—`PlanBean` includes the life-cycle methods `ejbCreate` and `ejbPostCreate`.

Business Methods

The business methods implement the operations declared in the local interface. The business methods—`getCost`, `addDoctor`, `removeDoctor`, `getAllDoctors`, `get-DoctorsByName`, and `getDoctorsBySpecialty`—are called from EnrollmentEJB and from the benefits administration Web application.

The `addDoctor` and `removeDoctor` methods operate on the `Collection` returned from the `getDoctor` method. The code for these methods is shown in Code Example 8.14:

```
public void addDoctor(Doctor doctor) throws PlanException {
    Collection doctors = getDoctors();
    doctors.add(doctor);
}

public boolean removeDoctor(Doctor doctor) throws PlanException {
    Collection doctors = getDoctors();
    return doctors.remove(doctor);
}
```

Code Example 8.14 Implementation of the `addDoctor` and `removeDoctor` Methods

The `addDoctor` method first obtains the `doctors` *managed* `Collection`, a live `Collection`; any changes made to the `Collection` cause the bean's persistent state to be changed. The `addDoctor` method simply adds the new DoctorEJB instance represented by the `doctor` argument to the managed `Collection`. Note that it is not necessary to add the PlanEJB instance to the corresponding plans `Collection` held by the DoctorEJB instance; the container automatically sets the other side of a bidirectional relationship when the first side is set.

The `removeDoctor` method obtains the `doctors` managed `Collection` and removes the `doctor` argument from the `Collection`. Again, it is not necessary to remove this PlanEJB instance from the corresponding `Collection` held by the DoctorEJB instance, because the container automatically does this for you.

The `getAllDoctors` method returns a `Collection` of all DoctorEJB instances related to a PlanEJB instance—that is, all the doctors participating in a particular insurance plan. Code Example 8.15 shows the code for `getAllDoctors`:

matena8.fm Page 273 Tuesday, January 7, 2003 4:34 PM

```
public Collection getAllDoctors() throws FinderException {
        Collection doctors = getDoctors();
        Collection doctorsCopy = new ArrayList(doctors);
    return doctorsCopy;
}
```

Code Example 8.15 Implementation of the `getAllDoctors` Method

You might wonder why the `Plan` local interface does not expose the `getDoctors` method. You might also wonder why the `getAllDoctors` method returns a new `Collection` rather than returning the `doctors` managed collection directly. The answer lies in the behavior of managed collections and transactions.

Managed collections are live collections whose state needs to be saved to the database when the transaction to which they are involved commits. Thus, managed collections are valid only in the transaction in which they were obtained. If a client of PlanEJB does not have an active transaction, the container starts a new transaction before calling the `getAllDoctors` method and commits the transaction immediately after the `getAllDoctors` method ends. In this situation, if it were given the `doctors` managed collection directly, the client would not be able to access the collection, as the transaction would have already committed. Operations on the collection at this point, such as obtaining an iterator, are out of the context of a transaction and would throw `IllegalStateException`.

To avoid such problems, Wombat's developers instead return a copy of the managed collection `doctors` to the client. The `doctorsCopy` collection is an *unmanaged* collection. A client can access an unmanaged collection at any time, without having to be within the context of a transaction, and changes to the unmanaged collection are not saved to the persistent state of the PlanEJB instance.

The `getDoctorsByName` method returns all doctors with the given first name and last name. To do this, the method invokes the `ejbSelectDoctorsByName` method. Similarly, the `getDoctorsBySpecialty` method returns all doctors with the given specialty. The method invokes the `ejbSelectDoctorsBySpecialty` method to accomplish this.

Home Business Methods

The home business methods implement the methods declared in the home interface. These methods execute on an instance that is not associated with a specific identity. As a result, they can perform only the logic that does not operate on a specific bean

matena8.fm Page 274 Tuesday, January 7, 2003 4:34 PM

instance. They cannot access the identity of the bean instance that is executing the method through the `getPrimaryKey`, `getEJBObject`, and `getEJBLocalObject` methods on the `EntityContext` interface. Usually, the home business methods implement aggregate operations or queries on an entire set of beans.

The `ejbHomeUpdateSmokerCosts` method implements the `updateSmokerCosts` method declared in the `PlanHome` home interface (Code Example 8.16):

```
public void ejbHomeUpdateSmokerCosts(double cost)
          throws FinderException {
    Collection allPlans = ejbSelectAllPlans();
    Iterator itr = allPlans.iterator();
    while ( itr.hasNext() ) {
        Plan plan = (Plan)itr.next();
        plan.setSmokerCost(cost);
    }
}
```

Code Example 8.16 Implementation of the `ejbHomeUpdateSmokerCosts` Method

This method uses the `cost` argument value to set one smoker cost for all plans. The method first obtains a `Collection` of all plans, using the `ejbSelectAllPlans` method, and then iterates over all the plans, setting the cost for each plan.

Two other home business methods—the `ejbHomeGetMedicalPlanNames` and the `ejbHomeGetDentalPlanNames` methods—are shown in Code Example 8.17:

```
// get all medical plan names
public String[] ejbHomeGetMedicalPlanNames()
          throws FinderException {
    Collection names = ejbSelectPlanNames(Plan.MEDICAL_PLAN);
    return (String[])names.toArray(new String[names.size()]);
}

// get all dental plan names
public String[] ejbHomeGetDentalPlanNames()
          throws FinderException {
    Collection names = ejbSelectPlanNames(Plan.DENTAL_PLAN);
```

matena8.fm Page 275 Tuesday, January 7, 2003 4:34 PM

```
        return (String[])names.toArray(new String[names.size()]);
    }
```

Code Example 8.17 Implementation of Home Business Methods

Depending on the application's needs, it can be more efficient to use a home business method than a find method. For example, a client can use the home business method `ejbHomeGetMedicalPlanNames` to retrieve an array of all medical plan names in one operation. The home business method returns a `Collection` or array of fields of a bean. This is often more efficient than using a find method, which can return only a `Collection` of enterprise bean local or remote interfaces. If the client instead uses the `PlanHome` `findMedicalPlans` method, the client gets back a `Collection` of enterprise bean interfaces. The client then has to iterate over the `Collection` and extract each plan name one at a time.

Select Methods

The `ejbSelect` methods declare EJB QL queries that are used internally by a bean class. The queries themselves are defined in the deployment descriptor. The `PlanBean` class declares five `ejbSelect` methods, shown in Code Example 8.18:

```
public abstract Collection ejbSelectAllPlans()
        throws FinderException;
public abstract Collection ejbSelectPlanNames(int planType)
        throws FinderException;
public abstract Collection ejbSelectDoctorsByName
        (String planId, String fname, String lname)
        throws FinderException;
public abstract Collection ejbSelectDoctorsBySpecialty
        (String planId, String specialty) throws FinderException;
public abstract long ejbSelectNumEmployeesInPlan(Plan plan)
        throws FinderException;
```

Code Example 8.18 The `ejbSelect` Method Declarations in the `PlanBean` Class

The `ejbSelectAllPlans` method returns all PlanEJB instances. The EJB QL query for this select method does not have a `WHERE` clause:

matena8.fm Page 276 Tuesday, January 7, 2003 4:34 PM

```
SELECT DISTINCT OBJECT(p) FROM PlanBean p
```

The `ejbSelectPlanNames` method returns the names of all plans of a given plan type. The EJB QL query for this method is

```
SELECT p.planName FROM PlanBean p WHERE p.planType = ?1
```

The `ejbSelectDoctorsByName` method returns for a given plan all participating doctors whose first and last names match the specified name parameters. The EJB QL query for this method is

```
SELECT DISTINCT OBJECT(d) FROM PlanBean p, IN(p.doctors) d WHERE
p.planId = ?1 AND d.firstName = ?2 AND d.lastName = ?3
```

The `ejbSelectDoctorsBySpecialty` method returns for a given plan all participating doctors with a particular specialty. The returned `Collection` of doctors is ordered by each doctor's last name. The EJB QL query for this method is

```
SELECT DISTINCT OBJECT(d) FROM PlanBean p, IN(p.doctors) d WHERE
p.planId = ?1 AND d.specialty = ?2 ORDER BY d.lastName
```

The `ejbSelectNumEmployeesInPlan` method returns the number of employees who have chosen a given plan. The EJB QL query for this method is

```
SELECT COUNT(s) FROM SelectionBean s WHERE s.medicalPlan = ?1
OR s.dentalPlan = ?1
```

Life-Cycle Methods

There are two life-cycle methods in the PlanEJB bean. The `ejbCreate` method initializes the container-managed persistence fields of `PlanBean`, using arguments that the client passes to the method. The `ejbPostCreate` initializes the timer that provides daily statistics to plan administrators. Code Example 8.19 shows the code used to create a timer:

```
TimerService timerService = entityContext.getTimerService();
timerService.createTimer(midnight, interval, null);
```

Code Example 8.19 Creating a Timer

To create a timer, the entity bean's identity—its primary key—needs to be available, because a `Timer` object is associated with the instance of the entity bean that created it. Hence, the timer can be created in the `ejbPostCreate` method but not in the `ejbCreate` method.

The `TimerService.createTimer` method used in Code Example 8.19 takes three parameters, two of which have values:

- **`midnight`**—A `java.util.Date` object whose value corresponds to midnight on the day the bean instance was created
- **`interval`**—A `long` type whose value represents the number of milliseconds in one day

Together, these parameters indicate that the timer will first expire at midnight on the first day and then at midnight every day thereafter. Let's now look at what happens when the timer expires.

`ejbTimeout` Method

The `PlanBean` class has an `ejbTimeout` method. This method is defined in the `Time-dObject` interface, which the `PlanBean` class implements. This method is called by the EJB container when the timer expires—at midnight every day. The code in the `ejbTimeout` method gets statistics about the `PlanBean` instances—the number of employees who have subscribed to each plan—and sends them to the administrators. It obtains these numbers by calling the select method `ejbSelectNumEmploy-eesInPlan`. Code Example 8.20 shows the code for the `ejbTimeout` method:

```
public void ejbTimeout(javax.ejb.Timer timer) {
    try {
        // get the number of employees who have subscribed to
        // this plan
        long numEmployeesInThisPlan = ejbSelectNumEmployeesInPlan(
                (Plan)entityContext.getEJBLocalObject());

        String emailText = "Plan " + getPlanName() + " has "
                + numEmployeesInThisPlan +" employees.";

        // email the text
        InitialContext ic = new InitialContext();
        Session session = (Session)ic.lookup(
```

matena8.fm Page 278 Tuesday, January 7, 2003 4:34 PM

```
                "java:comp/env/MailSession");
            String toAddress = (String)ic.lookup(
                "java:comp/env/toAddress");
            String fromAddress = (String)ic.lookup(
                "java:comp/env/fromAddress");

            Message msg = new MimeMessage(session);
            msg.setFrom(new InternetAddress(fromAddress));
            msg.addRecipient(Message.RecipientType.TO,
                new InternetAddress(toAddress));
            msg.setSubject("Statistics");
            msg.setText(emailText);

            Transport.send(msg);
        } catch ( Exception ex ) {
            throw new EJBException(ex);
        }
    }
```

Code Example 8.20 The `ejbTimeout` Method

The `ejbTimeout` method obtains the statistics by calling a select method and composing a text message string to be sent by e-mail to the plan administrator. It sends the e-mail message using the JavaMail™ APIs. Because these APIs are a standard part of the J2EE platform, they are available in all application servers that support J2EE. Typically, application servers provide an implementation of these APIs, which can send messages using SMTP (Simple Mail Transfer Protocol). The `ejbTimeout` method first looks up a JavaMail `Session` object using JNDI and then uses that object to create a JavaMail `MimeMessage`, which represents an e-mail message with MIME (Multipurpose Internet Mail Extensions) attachments. Next, the method sets in the `MimeMessage` object the sender, receiver, subject, and content text of the e-mail message. Finally, it sends the message using the default mail transport, usually SMTP. For more information about JavaMail, refer to `http://java.sun.com/products/javamail`.

matena8.fm Page 279 Tuesday, January 7, 2003 4:34 PM

8.2.6 DoctorEJB Entity Bean

The DoctorEJB entity bean, like PlanEJB, manages its state and relationships by using container-managed persistence. This entity bean essentially functions as a data object that stores information about physicians and dentists.

DoctorEJB Primary Key

The DoctorEJB primary key sets this bean apart from other entity beans in the benefits application. The DoctorEJB entity bean has a *composite* primary key consisting of two CMP fields—`firstName` and `lastName`—and also uses a special primary key class, `DoctorPkey`, created by Wombat's developers. Code Example 8.21 shows the code for `DoctorPkey`:

```
public class DoctorPkey implements java.io.Serializable {
   public String firstName;
   public String lastName;

   public boolean equals(Object other) {
      if ( other instanceof DoctorPkey ) {
         DoctorPkey pkey = (DoctorPkey)other;
         return (firstName.equals(pkey.firstName) &&
                 lastName.equals (pkey.lastName));
      }
      return false;
    }

   public int hashCode() {
      return (firstName.hashCode() | lastName.hashCode());
   }
}
```

Code Example 8.21 The `DoctorPkey` Primary Key Class

Because it uses a composite primary key, DoctorEJB needs to declare these two CMP fields as `public` Java fields in its `DoctorPkey` primary key class. The names of the fields in the primary key class need to be a subset of the names of the CMP fields in the bean class. In addition, the `DoctorPkey` class needs to provide

matena8.fm Page 280 Tuesday, January 7, 2003 4:34 PM

implementations of the `equals` and `hashCode` methods, which allow the EJB container to use `DoctorPkey` as a key into the container's internal data structures, such as hash tables.

`Doctor` Local Interface

The DoctorEJB entity bean is accessed from EnrollmentEJB and from the benefits administration Web application, both of which are packaged in the same benefits application. This copackaging of the bean with its clients ensures that only local access is required. As a result, the DoctorEJB entity bean defines only local interfaces. Code Example 8.22 shows the `Doctor` local interface definition:

```
public interface Doctor extends EJBLocalObject {
    String getLastName();
    String getFirstName();
    String getSpecialty();
    String[] getHospitals();
    int getPracticeSince();
}
```

Code Example 8.22 The `Doctor` Local Interface

The `Doctor` local interface defines methods that allow clients to view its persistent fields but does not define any other business methods. Note that the local interface does not allow clients to modify any information in the bean. Because the DoctorEJB bean is a *read-only* bean, the EJB container can manage it more efficiently.

`DoctorHome` Home Interface

The `DoctorHome` interface defines a `create` method that allows the benefits administration Web application to create DoctorEJB instances when doctors are added to a plan. This interface also defines the mandatory `findByPrimaryKey` method, which takes the `DoctorPkey` primary key class as its argument. Code Example 8.23 shows the `DoctorHome` interface definition:

```
public interface DoctorHome extends EJBLocalHome {
    Doctor create(String firstName, String lastName,
            String specialty, String hospital, int practiceSince)
```

matena8.fm Page 281 Tuesday, January 7, 2003 4:34 PM

```
            throws CreateException;
    Doctor findByPrimaryKey(DoctorPkey pkey) throws FinderException;
}
```

Code Example 8.23 The `DoctorHome` Home Interface

`DoctorBean` Entity Bean Class

Code Example 8.24 shows the source code of the `DoctorBean` entity bean class:

```
public abstract class DoctorBean extends AbstractEntityBean {
    public abstract String getFirstName();
    public abstract void setFirstName(String v);

    public abstract String getLastName();
    public abstract void setLastName(String v);

    public abstract String getSpecialty();
    public abstract void setSpecialty(String v);

    public abstract String[] getHospitals();
    public abstract void setHospitals(String[] v);

    public abstract int getPracticeSince();
    public abstract void setPracticeSince(int v);

    // CMR fields
    public abstract Collection getPlans();
    public abstract void setPlans(Collection c);

    // Life-cycle methods
    public DoctorPkey ejbCreate(String firstName, String lastName,
            String specialty, String[] hospitals, int practiceSince)
            throws CreateException {
        setFirstName(firstName);
        setLastName(lastName);
        setSpecialty(specialty);
        setHospitals(hospitals);
        setPracticeSince(practiceSince);
```

matena8.fm Page 282 Tuesday, January 7, 2003 4:34 PM

```
        return null;
    }

    public void ejbPostCreate(String firstName, String lastName,
            String specialty, String[] hospitals, int practiceSince)
            throws CreateException {}
}
```

Code Example 8.24 The `DoctorBean` Class Implementation

The `DoctorBean` class is a simple entity bean class that uses container-managed persistence: CMP fields define the persistent state of each DoctorEJB instance. The CMP fields, declared in the deployment descriptor, are `firstName`, `lastName`, `specialty`, `hospitals`, and `practiceSince`. Each field is represented as a pair of get and set methods in the `DoctorBean` class.

The `DoctorBean` class also defines a CMR field for its relationship with the PlanEJB entity bean. This CMR field, `plans`, is represented by the methods `setPlans` and `getPlans`. (Recall that the description of the `PlanBean` class discussed this relationship. See the section `PlanBean` Implementation Class on page 270.) The relationship between `DoctorBean` and `PlanBean` is bidirectional; hence, both classes have a CMR field to refer to each other. Because this relationship is many-to-many, the get and set methods in `DoctorBean` operate on a `Collection` of `Plan` objects.

The `DoctorBean` class defines an `ejbCreate` method that initializes the values of the CMP fields, including the two primary key fields `firstName` and `lastName`. The class also has an empty `ejbPostCreate` method.

8.2.7 EnrollmentWeb Web Application

The EnrollmentWeb Web application is a set of JSPs. See the example in Chapter 4, Working with Session Beans, for a description of the EnrollmentWeb Web application.

matena8.fm Page 283 Tuesday, January 7, 2003 4:34 PM

8.2.8 BenefitsAdminWeb Web Application

The BenefitsAdminWeb Web application is a set of JSPs used by the customer's benefits administration department to administer its benefits plans. The Benefits-AdminWeb Web application does the following work:

- Uses the find methods on the `PlanHome` interface to find the deployed plan beans from the respective insurance companies
- Allows the plan administrator to use the business methods on the `Plan` interface to query the doctors associated with a plan
- Uses the business methods on the `Plan` interface to allow doctors to be added and removed from a plan
- Allows the plan administrator to add or remove plans from the set of configured plans

Code Example 8.25 shows the skeleton code for adding an insurance plan to the set of configured plans. (Note that the example shows only those parts relevant to using the PlanEJB entity bean.)

```
...
// Create an entity object for the new plan.
InitialContext initialContext = new InitialContext();
PlanHome planHome = (PlanHome)initialContext.lookup
    ("java:comp/env/ejb/Plan");
Plan plan = planHome.create
    (planId, planName, planType, coverageFactor, ageFactor,
    smokerCost);
...
```

Code Example 8.25 Code for Adding a Plan

After the `create` method completes, the created plan entity object is saved to the database and becomes available to the Benefits Enrollment application. A plan administrator wanting to remove a plan from the list of configured plans invokes the `remove` method on the `PlanHome` interface and passes it the plan identifier, `planId`, which is PlanEJB's primary key:

matena8.fm Page 284 Tuesday, January 7, 2003 4:34 PM

```
...
planHome.remove(planId);
...
```

8.2.9 The Benefits Database

`BenefitsDatabase` stores the persistent state of the three entity beans SelectionEJB, PlanEJB, and DoctorEJB. All three beans use container-managed persistence, so Wombat does not need to specify a schema for this database. The schema is created by Wombat's customer, which in this case is Star Enterprise, using the object-relational mapping tools provided by the J2EE application server product on which these CMP entity beans are deployed.

8.2.10 Packaging of Parts

This section describes how Wombat packages its benefits application for distribution to customers.

`benefits.ear` File

Wombat packages the benefits application as a single J2EE enterprise application archive file, which it names `benefits.ear`. (An `.ear` file is an enterprise application archive resource file.) Figure 8.4 depicts the contents of the `benefits.ear` file.

The `benefits.ear` file contains

- The `enrollment.war` file with the EnrollmentWeb Web application. (A `.war` file is a Web archive file.) The EnrollmentWeb Web application consists of several JSPs.
- The `benefits_admin.war` file with the BenefitsAdminWeb Web application. The BenefitsAdminWeb Web application consists of several JSPs.
- The `benefits_ejb.jar` file. This is the ejb-jar file that contains the enterprise beans developed by Wombat.

matena8.fm Page 285 Tuesday, January 7, 2003 4:34 PM

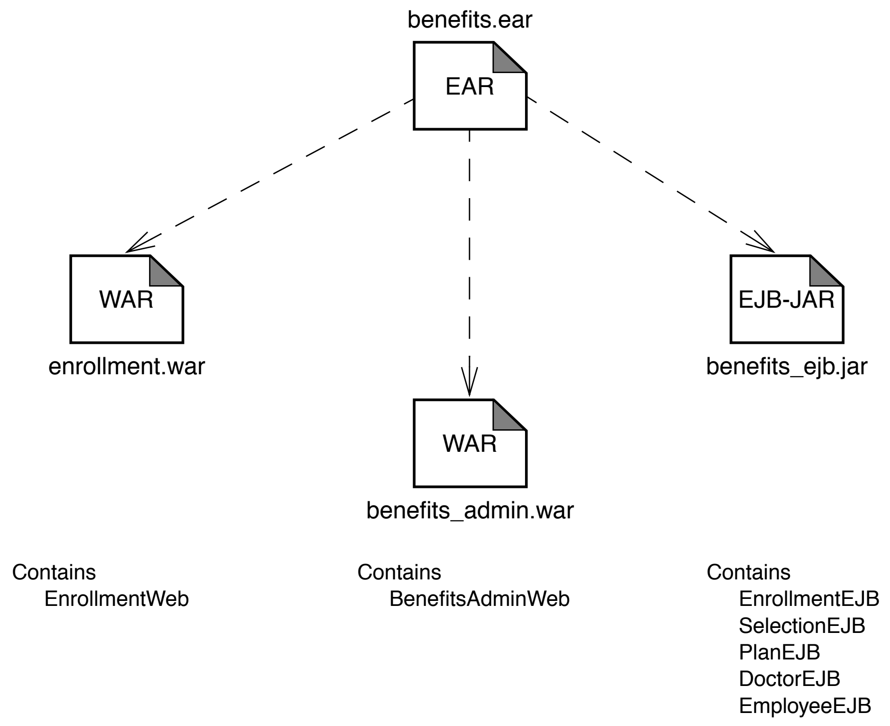

Figure 8.4 Contents of the `benefits.ear` File

`benefits_ejb.jar` File

The `benefits_ejb.jar` file contains the enterprise beans developed by Wombat. Code Example 8.26 lists the classes that the file contains:

```
com/wombat/AbstractEntityBean.class
com/wombat/benefits/DeductionUpdateBean.class
com/wombat/benefits/Employee.class
com/wombat/benefits/EmployeeBean.class
com/wombat/benefits/EmployeeHome.class
com/wombat/benefits/EmployeeInfo.class
com/wombat/benefits/Enrollment.class
com/wombat/benefits/EnrollmentBean.class
com/wombat/benefits/EnrollmentException.class
com/wombat/benefits/EnrollmentHome.class
```

matena8.fm Page 286 Tuesday, January 7, 2003 4:34 PM

```
com/wombat/benefits/Options.class
com/wombat/benefits/Selection.class
com/wombat/benefits/SelectionBean.class
com/wombat/benefits/SelectionCopy.class
com/wombat/benefits/SelectionException.class
com/wombat/benefits/SelectionHome.class
com/wombat/benefits/Summary.class
com/wombat/plan/Doctor.class
com/wombat/plan/DoctorBean.class
com/wombat/plan/DoctorHome.class
com/wombat/plan/DoctorPkey.class
com/wombat/plan/Plan.class
com/wombat/plan/PlanBean.class
com/wombat/plan/PlanException.class
com/wombat/plan/PlanHome.class
```

Code Example 8.26 Contents of the `benefits_ejb.jar` File

8.3 Parts Developed at Star Enterprise

Prior to the deployment of Wombat's benefits application, Star Enterprise already had a Benefits Enrollment application that it had developed internally. (See Chapter 4 for the description of the Benefits Enrollment application using session beans.)

With the deployment of Wombat's benefits application, Star Enterprise needs to integrate some parts of its application into Wombat's application. This section addresses these issues.

8.3.1 The Employee Database and Deployment of EmployeeEJB

The human resources department at Star Enterprise maintains the information about employees and company departments in `EmployeeDatabase`. The information is stored in multiple tables. The `Employees` table within the database is relevant to the benefits application. The schema for `EmployeeDatabase` is described in Section 4.6.1, The Employee Database, on page 120.

Note that as an ISV, Wombat had no knowledge of the schema of `EmployeeDa-tabase` and did not need that knowledge. Coding its benefits application as generically as possible, Wombat's primary consideration was that the application work regardless of an individual customer's schema and type of DBMS. If Wombat

matena8.fm Page 287 Tuesday, January 7, 2003 4:34 PM

coded the benefits application according to the Star Enterprise schema, the application would be unusable by customers having a different schema or even a different type of DBMS.

Star Enterprise has two choices for deploying the EmployeeEJB entity bean developed by the ISV Wombat:

1. It can use an object-relational mapping tool to map the CMP fields of the EmployeeEJB to the columns of the `Employees` table in its relational database `EmployeeDatabase`. An example of such a mapping of fields follows:

CMP Field in EmployeeBean	Column Name in `Employees` Table
`employeeNumber`	`empl_id`
`firstName`	`empl_first_name`
`lastName`	`empl_last_name`
`birthDate`	`empl_birth_date`

2. It can develop an entity bean by using bean-managed persistence and can write the database access code in the BMP bean class. This approach is described in the next section.

8.3.2 `EmployeeBeanBMP` Entity Bean Class

The `EmployeeBeanBMP` bean class uses bean-managed persistence to manage the state of the EmployeeEJB entity bean. This class uses the same local interface `Employee` and local home interface `EmployeeHome` as in the EmployeeEJB entity bean. (See Section 8.2.3, EmployeeEJB Entity Bean, on page 255.) The `EmployeeBeanBMP` class subclasses the `EmployeeBean` container-managed persistence class.

This section discusses some of the important issues that need to be kept in mind when developing entity beans with bean-managed persistence. Portions of code from the `EmployeeBeanBMP` class are used to illustrate these points. See Section A.7, `EmployeeBeanBMP` Class, on page 402 for the complete code for the `EmployeeBeanBMP` class.

The `EmployeeBeanBMP` class has three types of methods:

- Get and set methods for CMP fields
- `EntityBean` interface life-cycle methods
- Database access helper methods

CMP Field Methods

The EmployeeBeanBMP class needs to implement all abstract methods defined in the EmployeeBean class. These methods correspond to the abstract get and set methods for the CMP fields declared in the EmployeeBean CMP class. These methods get and set concrete fields in the EmployeeBeanBMP class corresponding to the CMP fields. Code Example 8.27 shows the code for these methods:

```
public class EmployeeBeanBMP extends EmployeeBean {
    // this field holds the JDBC DataSource for the employee database
    private DataSource dataSource;
    // the following fields hold the persistent state of
    // the EmployeeBean.
    private Integer employeeNumber;
    private String firstName;
    private String lastName;
    private Date birthDate;

    // The following methods implement the abstract CMP
    // field getters/setters declared in the EmployeeBean class.
    public Integer getEmployeeNumber() {
        return employeeNumber;
    }
    public void setEmployeeNumber(Integer n) {
        employeeNumber = n;
    }

    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String s) {
        firstName = s;
    }

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String s) {
        lastName = s;
    }
    public Date getBirthDate() {
        return birthDate;
    }
    public void setBirthDate(Date d) {
        birthDate = d;
    }
    ....
}
```

Code Example 8.27 Implementation of `EmployeeBeanBMP` Get and Set Methods

Life-Cycle Methods

The `EmployeeBeanBMP` class also implements the entity bean life-cycle methods, including `ejbCreate` and `ejbPostCreate`, and methods of the `javax.ejb.EntityBean` interface. Code Example 8.28 shows the code for the `ejbCreate` and `ejbPostCreate` methods:

```
public Integer ejbCreate(int emplNumber, String fname, String lname,
                    Date birthDate) throws CreateException {
    // this sets all the CMP fields
    super.ejbCreate(emplNumber, fname, lname, birthDate);

    // check if the primary key exists
    if ( primaryKeyExists(emplNumber) ) {
        throw new DuplicateKeyException("Employee number " +
                emplNumber + " already exists in database");
    }

    // create a row for this bean instance
    createRow();

    // return the primary key
    return new Integer(emplNumber);
}

public void ejbPostCreate(int emplNumber, String fname,
            String lname, Date birthDate) throws CreateException
{}
```

Code Example 8.28 `EmployeeBeanBMP` `ejbCreate` and `ejbPostCreate` Methods

`EmployeeBeanBMP`'s `ejbCreate` method first calls its superclass `EmployeeBean`'s `ejbCreate` method, which sets the values of the CMP fields by calling the respective set methods. This operation sets values in the Java fields in the `EmployeeBeanBMP` class. `EmployeeBeanBMP`'s `ejbCreate` method then checks whether the primary key for the employee, which is the employee number—`emplNumber`—already exists in the database. If it does, the method throws `DuplicateKeyException` to indicate to the client that an application-level error has occurred. If the employee number does not exist, the `ejbCreate` method creates a row for the employee in the database by calling the `createRow` helper method. Finally, the `ejbCreate` method returns the new bean instance's primary key, as required for BMP entity beans. The container converts this primary key to an `Employee` reference and returns it to the client.

Two additional entity bean life-cycle methods are in the `EmployeeBeanBMP` class: `setEntityContext` and `unsetEntityContext`. Code Example 8.29 shows the code for these methods:

```
public void setEntityContext(EntityContext c) {
    super.setEntityContext(c);
    String dataSourceName = "java:comp/env/jdbc/EmployeeDatabase";
    try {
        Context ctx = new InitialContext();
        dataSource = (DataSource)ctx.lookup(dataSourceName);
    } catch ( Exception ex ) {
        throw new EJBException("Unable to look up dataSource "
                + dataSourceName);
    }
}

public void unsetEntityContext() {
    dataSource = null;
```

matena8.fm Page 291 Tuesday, January 7, 2003 4:34 PM

```
        super.unsetEntityContext();
    }
```

Code Example 8.29 The `setEntityContext` and `unsetEntityContext` Methods

The `setEntityContext` method is called immediately after the `EmployeeBean-BMP` class is instantiated. The method first calls the `EmployeeBeanBMP` superclass's `setEntityContext` method to allow the superclass to do any necessary initialization. The method then looks up the JDBC `DataSource` object for `EmployeeData-base` from the JNDI environment.

The `unsetEntityContext` is the last method called before the `EmployeeBean-BMP` instance is destroyed. This method clears the value of the `DataSource` field and calls the superclass's `unsetEntityContext` method to allow the superclass to perform any required cleanup.

Database Access Methods

The `EmployeeBeanBMP` class implements the `ejbFindByPrimaryKey` method (Code Example 8.30):

```
    public Integer ejbFindByPrimaryKey(Integer emplNum)
              throws FinderException {
        // Try to load the row for this primary key
        if ( !primaryKeyExists(emplNum.intValue()) ) {
            throw new ObjectNotFoundException("Primary key " + primaryKey
                        + " not found");
        }
        return emplNum;
    }
```

Code Example 8.30 The `EmployeeBeanBMP` Class `ejbFindByPrimaryKey` Method

The `ejbFindByPrimaryKey` method checks whether the `emplNum` argument—the primary key for the EmployeeEJB bean—exists in the database, doing so by calling the helper method `primaryKeyExists`. If the employee number does not exist, the method throws `ObjectNotFoundException` to inform the client. If the

matena8.fm Page 292 Tuesday, January 7, 2003 4:34 PM

employee number does exist, the method returns the primary key. The container converts this primary key to an `Employee` reference and returns it to the client.

The `EmployeeBeanBMP` class also implements the `ejbRemove` method, as shown in Code Example 8.31:

```
public void ejbRemove() throws RemoveException {
    super.ejbRemove();

    // remove the row for this primary key
    removeRow();

    // clear all CMP fields
    employeeNumber = null;
    firstName = null;
    lastName = null;
    birthDate = null;
}
```

Code Example 8.31 The `ejbRemove` Method in `EmployeeBeanBMP`

The `ejbRemove` method first calls the `ejbRemove` method of the superclass `EmployeeBean`. This allows the superclass's `ejbRemove` implementation to do any needed work. Then `EmployeeBeanBMP`'s `ejbRemove` method calls the helper method `removeRow` to remove the row for this bean instance from the database. After this, the `ejbRemove` method clears all employee-specific fields. This is an important step because the bean instance goes into the container's pool after removal and can be used for another employee with a different identity and different fields. The `ejbRemove` method needs to clean up all fields that are specific to the particular employee. Note that `ejbRemove` does not need to clear the `dataSource` field, because that field's value does not depend on any particular employee.

Now let's examine the `ejbLoad` and `ejbStore` methods, which are used to synchronize the state of the bean with the persistent state in the database. The `ejbLoad` method is usually called at the beginning of a transaction, before any business methods are called. The `ejbStore` method is usually called at the end of a transaction, when a transaction is committed. Code Example 8.32 shows the code for these methods:

matena8.fm Page 293 Tuesday, January 7, 2003 4:34 PM

```
public void ejbLoad() {
    try {
        loadRow();
    } catch ( Exception ex ) {
        throw new NoSuchEntityException(
            "Exception caught in ejbLoad: "+ex);
    }
    super.ejbLoad();
}

public void ejbStore() {
    super.ejbStore();
    try {
        storeRow();
    } catch ( Exception ex ) {
        throw new EJBException("Exception caught in ejbStore ", ex);
    }
}
```

Code Example 8.32 `EmployeeBeanBMP`'s `ejbLoad` and `ejbStore` Methods

The `ejbLoad` method calls the `loadRow` helper method to load the state of the bean. The `loadRow` helper method does a database read, using the employee number primary key; retrieves the row for the employee; and sets the CMP fields in the `EmployeeBeanBMP` class. After the `loadRow` method completes successfully, the `ejbLoad` method calls the superclass `EmployeeBean`'s `ejbLoad` method to allow the superclass to initialize any cached transient variables from the CMP fields.

The `ejbStore` method first calls the superclass `EmployeeBean`'s `ejbStore` method. This allows the superclass to save any cached data to the CMP fields. After this, the `ejbStore` method stores the state of `EmployeeBeanBMP` by calling the `storeRow` helper method, which writes all the CMP fields to the database row representing the employee. For better performance, `EmployeeBeanBMP` should perform this database write only if the bean's CMP fields have been modified since the last time they were loaded. This optimization can be done by maintaining a flag in the `EmployeeBeanBMP`'s instance variable that indicates whether the bean's state is "dirty"—that is, changed. The flag needs to be set in every set method that modifies the bean's state.

matena8.fm Page 294 Tuesday, January 7, 2003 4:34 PM

An entity bean with bean-managed persistence must also implement the `ejbPassivate` and `ejbActivate` methods. Code Example 8.33 shows how `EmployeeBeanBMP` implemented these two methods:

```
public void ejbActivate() {
    employeeNumber = (Integer)entityContext.getPrimaryKey();
}

public void ejbPassivate() {
    // clear all CMP fields
    employeeNumber = null;
    firstName = null;
    lastName = null;
    birthDate = null;
}
```

Code Example 8.33 `EmployeeBeanBMP ejbActivate` and `ejbPassivate` Methods

The `ejbActivate` method is called when an entity bean instance is being associated with a specific primary key value. The `EmployeeBeanBMP`'s `ejbActivate` method initializes the `employeeNumber` field to the instance's primary key value. This association step is very important because it allows subsequent methods, such as `ejbLoad`, to operate on the correct primary key. In fact, `ejbActivate` is typically called immediately before `ejbLoad` at the beginning of a transaction.

The `ejbPassivate` method is called when the container wants to reclaim the memory associated with a bean instance and return the bean instance to its pool. The `ejbPassivate` method clears the values of all employee-specific fields—that is, the CMP fields. After the `ejbPassivate` method completes, the bean instance is no longer associated with an identity and, by calling `ejbActivate`, can be used to service a request on behalf of a different employee.

8.3.3 Payroll System

Prior to the deployment of Wombat's benefits application, Star Enterprise's payroll department developed a payroll application to give its enterprise applications access to payroll information. The payroll application consists of the PayrollEJB stateless session bean, described in Section 4.5, PayrollEJB Stateless Session Bean, on page

matena8.fm Page 295 Tuesday, January 7, 2003 4:34 PM

110. The payroll information is stored in `PayrollDatabase`, whose schema is described in Section 4.6.3, The Payroll Database, on page 123.

To integrate the PayrollEJB with the Benefits Enrollment application developed by Wombat, Star Enterprise's IT department develops an implementation of the `DeductionUpdateBean` command bean interface provided by Wombat. The deployer then sets the class name of this interface in the EnrollmentEJB's environment. This allows EnrollmentEJB to instantiate the `DeductionUpdateBean` implementation and use it to update the benefits deduction in Star Enterprise's payroll system.

8.4 Conclusion

We have now completed our examination of entity beans. This chapter presented an employee Benefits Enrollment application that was similar to the example presented earlier. However, this example was built and deployed using entity beans when appropriate rather than relying completely on session beans.

The example application clearly illustrated the differences, from a developer's point of view, of using entity beans. The application focused on the various techniques for working with entity beans, such as using container-managed persistence and container-managed relationships, caching persistent state, subclassing techniques, and so forth, and how best to use the features of these types of beans.

The example application also illustrated how to use the EJB timer service and the JavaMail APIs to provide plan administrators with statistics about the application by e-mail on a regular basis.

This chapter showed how entity beans are more appropriate for applications that must be easily adapted for different customers with different operational environments. Typically, these are applications built by ISVs rather than by an enterprise's in-house IT department.

matena8.fm Page 296 Tuesday, January 7, 2003 4:34 PM