CHAPTER 7

Entity Bean Application Example

THIS chapter uses an example of a distributed application to illustrate how enterprise applications use entity beans. The example application is typical of applications that include components built by multiple organizations. The chapter shows how the different organizations develop the respective components of the application and how, ultimately, the customer deploys the entire application.

The example application illustrates the following:

- **The techniques for using entity beans to develop applications for different customers with different operational environments.** An Independent Software Vendor (ISV) uses entity beans in the development of an application so that it can sell the application to as broad a possible range of customers and operational environments. Our example illustrates:
  - How the ISV uses entity beans with container-managed persistence to integrate its application with the customer's existing applications and database.
  - How the ISV uses the entity bean client-view API as the integration points with the different third party components.
- **The techniques for integrating application software parts from multiple vendors into a single application.** We illustrate:
  - How an ISV enables the customer to extend the application by adding components from other vendors.
  - How other vendors structure their components to facilitate the integration of their components with the first ISV.

- **The implementation of several entity beans to illustrate the various issues of the entity bean architecture.** Entity beans use different styles to implement their persistence; thus, we illustrate the use of both container-managed and bean-managed persistence. In addition, we illustrate different strategies for implementing entity bean state, including:
  - An entity bean whose state is implemented in a database using JDBC. This is the PremiumHealthPlanEJB entity bean.
  - An entity bean whose state is implemented by other entity beans. This is the WrapperPlanEJB entity bean.
  - An entity bean whose state is stored in a remote Web server accessible using XML over HTTP. This is the ProvidencePlanBeanEJB entity bean.
- **Various design approaches for an entity bean's remote interface, plus the advantages each approach offers to different applications.** The developer should design the remote interface such that its methods support the intended client use cases.
- **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 use of an entity bean to aggregate the function of multiple other entity beans into a single component.** WrapperPlanEJB is an entity bean that performs just such a function. Aggregation simplifies the development of an application that needs to access multiple entity beans because the application can access their functions through the single aggregate bean.
- **The correspondence of entity beans to database rows.** In particular, an entity object does not always correspond to a single row in a database. The example illustrates that the state of an entity object can map to multiple rows in possibly multiple tables in the database. In addition, we show that it is not necessary to expose as entity beans the business objects which are not intended to be invoked from other applications. These objects could be implemented as dependent objects of the entity beans with which they are associated. For example, the Doctor object is implemented as a dependent object of the Plan entity object.
- **The techniques for subclassing an entity bean with container-managed**

persistence to create an entity bean with bean-managed persistence. The subclass implements the data access methods.

- **The packaging of enterprise beans into J2EE standard files.** The example illustrates packaging of the entity 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 which do *not* have to be developed.** The example code is also interesting in what it does not include. The reader will see no transaction nor security management-related code. The deployment descriptors describe declaratively the transaction and security requirements for entity beans. The transaction management is described in Chapter 8, Understanding Transactions, and Chapter 9, Managing Security, describes security management.
- **A discussion on application integration techniques involving entity beans.**

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

7.1 Application Overview

Our example application illustrates the development and deployment of an enterprise application that consists of components developed by multiple vendors. We illustrate how using entity beans in the application facilitates the integration of the components from different vendors.

7.1.1 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 application described in Chapter 4, Working with Session Beans, which in that case was a benefits application built using session beans. However, the following are the key differences in the design of the two applications:

- Wombat Inc. developed the core of the application. Wombat is an ISV that spe-

cializes in the development of benefits applications used by enterprises. Wombat wants to sell its application to as many different enterprises as it can; this means that its application must work in a myriad of different operational environments. In contrast, Star Enterprise's IT department developed the application illustrated in Chapter 4. Because that application was only intended to be used within Star Enterprise's own environment, it was developed with no regard for the application's portability to operational environments other than that of Star Enterprise.

- 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.
- Star Enterprise uses a payroll system from Aardvark (another ISV) that is a mainframe application. External programs cannot directly access the payroll system or its database. The PayrollEJB enterprise bean from Aardvark provides the integration interfaces so that other applications can access the payroll system.
- The application described in this chapter dynamically interacts with the plan providers; that is, with the insurance companies. For example, a plan provider can change the premium calculation formula and manage the list of doctors participating in the offered plans, and the application reflects these changes. In contrast, the application in Chapter 4 requires a redeployment to change this information.

7.1.2 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 already existed at Star Enterprise, while multiple outside organizations developed the other parts. Figure 7.1 illustrates the logical parts of the application. It is followed by a brief description.

The section “Summary of the Integration Techniques” on page 362 discusses how to use the EJB architecture to integrate these parts at deployment time.

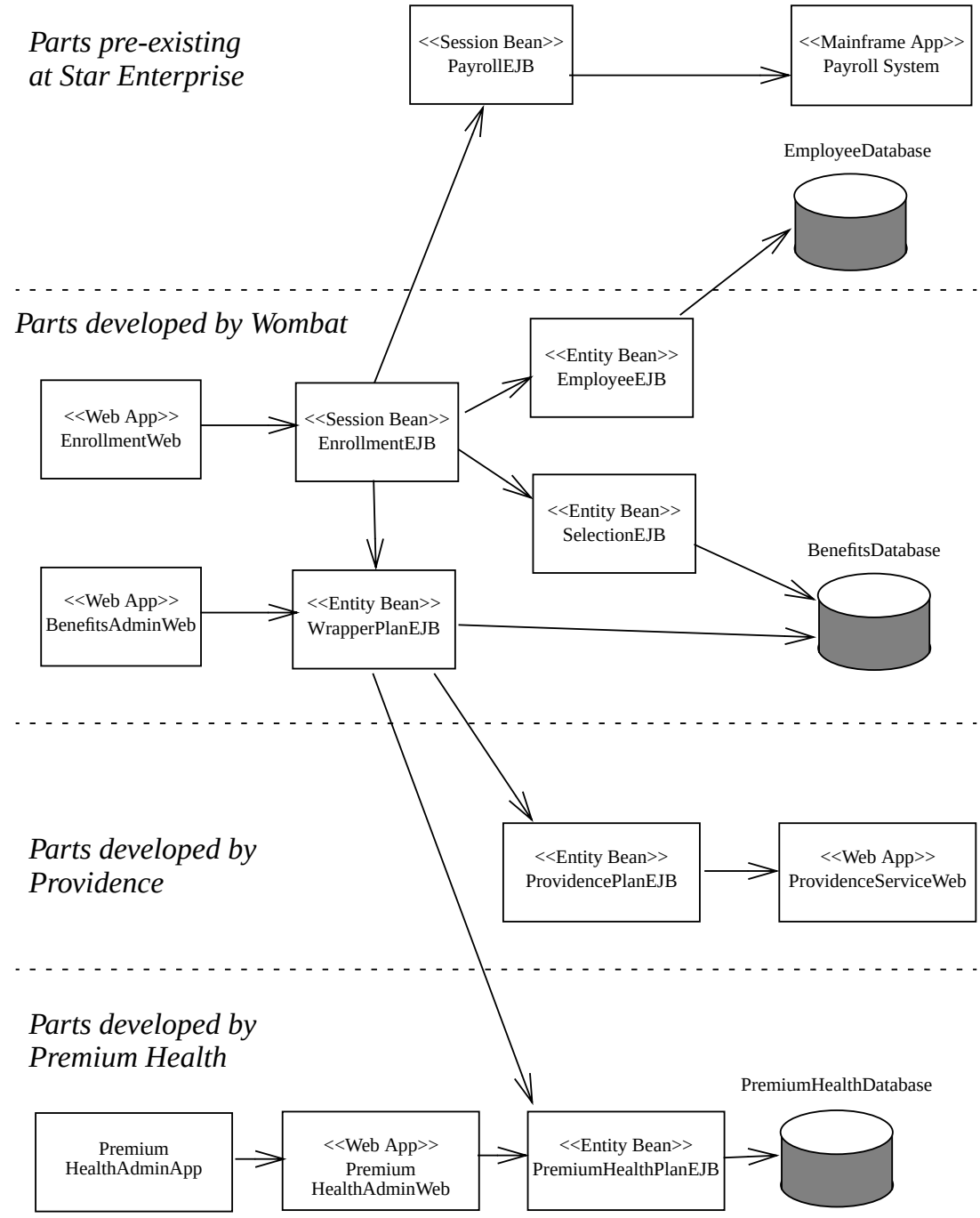

Figure 7.1 Logical Parts of the Entity Bean Benefits Application

The application consists of four principal parts, and these parts come from four different sources:

- The pre-existing employee and payroll parts in the Star Enterprise's operational environment
- The Wombat Benefits application, which consists of multiple enterprise beans and Web applications
- The enterprise bean and Web application parts developed by Premium Health Insurance
- The enterprise bean and Web application parts developed by Providence Insurance

Prior to the deployment of Wombat's Benefits application, Star Enterprise utilized the EmployeeDatabase, Payroll System, and PayrollEJB parts. These parts pertain to the following aspects of Star Enterprise's business:

- EmployeeDatabase contains the information about Star Enterprise's employees.
- Payroll System is a mainframe application that Star enterprise uses for its payroll.
- PayrollEJB is a stateless session bean that provides non-payroll applications with secure access to the payroll system. Non-payroll 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's 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 is a stateful session bean that implements the benefits enrollment business process. It uses several entity beans to perform its function.
- EmployeeEJB is an entity bean that encapsulates access to the customer's (Star Enterprise, in this example) employee information. It is an entity bean with container-managed persistence and its main role is to allow deployment-time binding with the customer's employee database.
- SelectionEJB is an entity bean that encapsulates the benefits selections chosen by each employee.
- WrapperPlanEJB is an entity bean that aggregates the medical and dental plans from multiple providers (that is, from the insurance companies). It is the integration point between the benefits application and the plan entity beans provided by the insurance companies.
- BenefitsDatabase stores the information used by the SelectionEJB and WrapperPlanEJB entity beans.

Wombat defines what an insurance provider needs to do to integrate its plan information with Wombat's Benefits application. This integration technique is explained in the section "Benefits Plan Integration Approach" on page 295. Our example shows the parts developed by two insurance providers: Premium Health and Providence. For the sake of illustration, Premium Health and Providence use very different techniques to implement access to their plan information.

Premium Health provides PremiumHealthPlanEJB, PremiumHealthAdminWeb, PremiumHealthDatabase, and PremiumHealthAdminApp.

- PremiumHealthPlanEJB is an entity bean that implements the interfaces defined by Wombat. PremiumHealthPlanEJB allows the Wombat Benefits application to access the information about the medical and dental plans offered by Premium Health.
- PremiumHealthDatabase is a database residing at Star Enterprise that stores the data used by PremiumHealthPlanEJB.
- PremiumHealthAdminWeb is a Web application that allows Premium Health to remotely update the benefits information stored in PremiumHealthDatabase at Star Enterprise.

- PremiumHealthAdminApp is an application that Premium Health uses to upload and modify the plan information in PremiumHealthDatabase.

Providence provides the ProvidencePlanEJB entity bean and ProvidenceServiceWeb Web application.

- ProvidencePlanEJB allows the Wombat Benefits application to access information about the medical and dental plans offered by Providence. ProvidencePlanEJB communicates using XML over HTTP with the ProvidenceServiceWeb located at Providence. This enables the Benefits application to get online the information about the medical and dental plans offered by Providence.
- ProvidenceServiceWeb is a Web application located at the Providence site. It is used by applications running at customer sites, including the Benefits application running at Star Enterprise, to obtain the current plan information for the plans offered by Providence.

7.1.3 Distributed Deployment

Star Enterprise has deployed the Benefits application across multiple servers, including six servers within its own enterprise intranet. The Benefits application also communicates with two servers located at the insurance companies. The deployment configuration is shown in Figure 7.2:

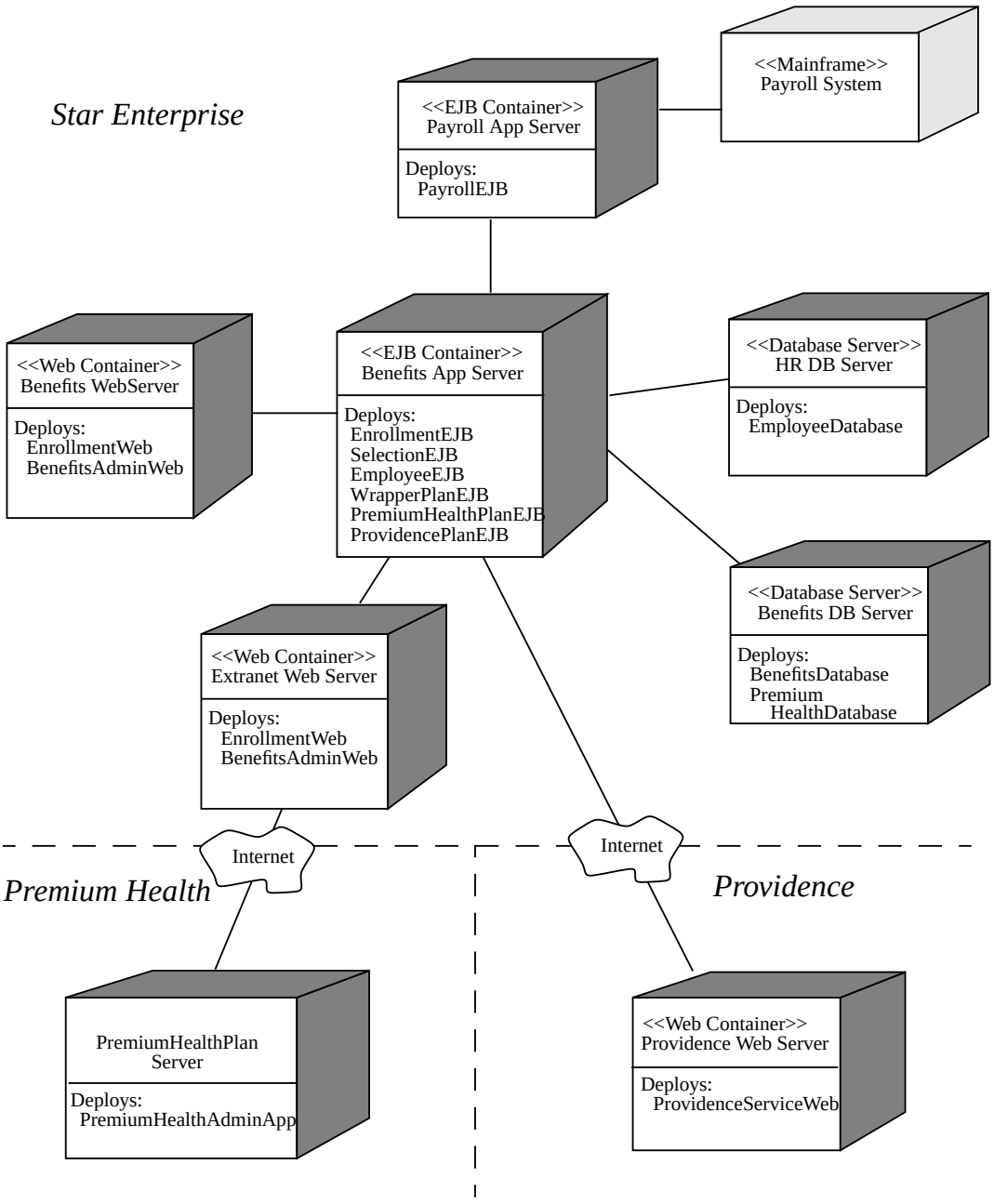

Figure 7.2 Benefits Application Deployment

The Benefits application is spread across six servers within Star Enterprise's own physical confines. The benefits department has deployed the EnrollmentWeb

and BenefitsWeb Web applications on the Benefits Web Server. It has deployed the enterprise beans EnrollmentEJB, SelectionEJB, EmployeeEJB, WrapperPlanEJB, PremiumHealthPlanEJB, and ProvidencePlanEJB on the Benefits App Server. BenefitsDatabase and PremiumHealthDatabase are stored on the Benefits DB Server. The enterprise bean PayrollEJB is deployed on the Payroll App Server, which in turn provides the connection to the payroll system on the mainframe. EmployeeDatabase is stored on the Human Resources (HR) DB Server. The PremiumHealthAdminWeb Web application is deployed on the Extranet Web Server that Star Enterprise uses for communication with its trading partners.

The deployment diagram also illustrates the two servers that are physically located outside of Star Enterprise. The Premium Plan Admin App Server is located at Premium Health and the Providence Web Server is located at Providence. The Premium Plan Admin App Server runs the PremiumHealthAdminApp application that communicates with the PremiumHealthAdminWeb located at Star Enterprise; this application updates the health plan information stored at Star Enterprise. The Providence Web Server hosts the ProvidenceServiceWeb Web application, which is invoked by the ProvidencePlanEJB deployed at Star Enterprise.

Figure 7.2 illustrates deploying Web applications and enterprise beans across multiple server machines. This is just one deployment scenario. It is possible to deploy the application on fewer machines without modification to its components. For example, the Benefits Web Server, Benefits App Server, Benefits DB Server, and HR DB Server can all be part of the same J2EE server installed on a single server machine. The example illustrates the more distributed deployment scenario because it reflects the traditional division of “information ownership” by multiple departments within a large enterprise. It also illustrates the power and flexibility of the EJB architecture for developing and deploying distributed applications.

7.2 Pre-existing Parts at Star Enterprise

This section describes the relevant software and systems at Star Enterprise that preexisted the deployment of Wombat’s Benefits application.

7.2.1 EmployeeDatabase

The HR Department at Star Enterprise maintains the information about employees and the company departments in EmployeeDatabase. The information is stored in

multiple tables. The Employees table within the database is relevant to the Benefits application. Code Example 7.1 shows the SQL `CREATE` statement defining this table:

```
CREATE TABLE Employees (
    empl_id INT,
    empl_first_name VARCHAR(32),
    empl_last_name VARCHAR(32),
    empl_addr_street VARCHAR(32),
    empl_addr_city VARCHAR(32),
    empl_addr_zip VARCHAR(10),
    empl_addr_state VARCHAR(2),
    empl_dept_id VARCHAR(10),
    empl_start_date DATE,
    empl_position VARCHAR(5),
    empl_birth_date DATE,
    ...
    PRIMARY KEY ( empl_id )
)
```

Code Example 7.1 Employees Table Definition

Note that Wombat has no knowledge of the schema of EmployeeDatabase, nor does it need that knowledge. Wombat must code its Benefits application as generically as possible. Its primary consideration is that the application work regardless of an individual customer's schema and type of database management system. If Wombat coded the Benefits application using the Star Enterprise's schema, the application would be unusable by other customers who are likely to have a different schema or even a different type of database management system.

7.2.2 Payroll System

Star Enterprise's payroll system is a mainframe application. It consists of a collection of CICS Transaction Processing (TP) programs.

Prior to the deployment of Wombat's Benefits application, Star Enterprise needed to give its non-mainframe applications access to the payroll information. To accomplish this, Star Enterprise purchased a mainframe connectivity product from vendor Aardvark.

Aardvark sells enterprise beans that provide access to popular mainframe applications, including the payroll system used by Star Enterprise. Star Enterprise bought Aardvark's PayrollEJB enterprise bean to enable access to the payroll system from Java applications.

Aardvark packages the client-view interface files for its enterprise bean products into Java ARchive (JAR) files. Because Aardvark wants to encourage others to build products utilizing Aardvark's enterprise beans, it makes these client-view JAR files available to other ISVs. With the JAR files, other vendors can create Java applications that communicate with Aardvark's enterprise beans. The JAR file `payroll_ejb_client.jar` contains the client view of PayrollEJB. Wombat, an ISV, uses the `payroll_ejb_client.jar` file in its Benefits application to access the customer's payroll system.

Star Enterprise deployed Aardvark's mainframe connectivity product and PayrollEJB.

7.2.2.1 PayrollEJB Enterprise Bean

Aardvark supplies PayrollEJB, which is a stateless session bean. The external interface of the mainframe payroll system is a procedural interface, and therefore Aardvark defined its Java representation using a stateless session bean.

Code Example 7.2 shows the definition for the Payroll interface which is the PayrollEJB's remote interface:

```
package com.aardvark.payroll;

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

public interface Payroll extends EJBObject {
    void setBenefitsDeduction(int emplNumber, double deduction)
        throws RemoteException, PayrollException;
    double getBenefitsDeduction(int emplNumber)
        throws RemoteException, PayrollException;
    double getSalary(int emplNumber)
        throws RemoteException, PayrollException;
    void setSalary(int emplNumber, double salary)
        throws RemoteException, PayrollException;
}
```

Code Example 7.2 Payroll Remote Interface

Code Example 7.3 gives the definition of the PayrollHome home interface:

```
package com.aardvark.payroll;

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

public interface PayrollHome extends EJBHome {
    Payroll create() throws RemoteException, CreateException;
}
```

Code Example 7.3 PayrollHome Home Interface

The implementation of PayrollEJB's PayrollBean class uses Aardvark's mainframe connectivity product. The mainframe connectivity product integrates with the EJB Container as a resource adapter using the Connector Architecture to communicate with the payroll system transaction processing programs on the mainframe. (Refer to the *Java 2 platform, Enterprise Edition Connector Specification*. See "Other Sources of Information" on page xv for the complete reference to this specification. Also note that the Connector API, while public, is not yet the final version.)

Code Example 7.4 shows the implementation of the PayrollBean class.

```
package com.aardvark.payroll.impl;

import javax.ejb.*;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.Connection;
import javax.resource.cci.MappedRecord;
import javax.resource.cci.RecordFactory;
import javax.resource.cci.InteractionSpec;
import javax.resource.cci.Interaction;
```

```
import javax.resource.ResourceException;

import com.aardvark.payroll.PayrollException;

public class PayrollBean implements SessionBean {
    // Mainframe connection factory.
    private ConnectionFactory connectionFactory;

    public void setBenefitsDeduction(int emplNumber,
           double deduction) throws PayrollException
    {
        Connection cx = null;
        try {
            // Obtain connection to mainframe
            cx = getConnection();

            // Create an interaction object
            Interaction ix = cx.createInteraction();

            InteractionSpecImpl ixSpec = new InteractionSpecImpl();

            // Set the name of the TP program to be invoked
            ixSpec.setFunctionName("SETPAYROLL_DEDUCTION");

            // Specify that we will be sending input parameters
            // to the TP program, but are expecting to receive
            // no output parameters.
            ixSpec.setInteractionVerb(InteractionSpec.SYNC_SEND);

            RecordFactory rf = ix.getRecordFactory();

            // Create an object that knows how to
            // format the input parameters
            MappedRecord input =
                rf.createMappedRecord("PAYROLLINFO_DEDUCTION");
            input.put("EMPLOYEENUMBER", new Integer(emplNumber));
            input.put("DEDUCTION", new Double(deduction));

            // Execute invokes the TP program, passing it
```

```
        // the input parameters
        ix.execute(ixSpec, input);
    } catch (ResourceException ex) {
        throw new EJBException(ex);
    } finally {
        try {
            if (cx != null) cx.close();
        } catch (ResourceException ex) {
        }
    }
}

public double getBenefitsDeduction(int emplNumber)
    throws PayrollException
{
    Connection cx = null;
    try {
        cx = getConnection();
        Interaction ix = cx.createInteraction();

        InteractionSpecImpl ixSpec = new InteractionSpecImpl();
        ixSpec.setFunctionName("GETPAYROLLDATA");
        ixSpec.setInteractionVerb(
            InteractionSpec.SYNC_SEND_RECEIVE);

        RecordFactory rf = ix.getRecordFactory();

        MappedRecord input =
            rf.createMappedRecord("EMPLOYEEINFO");
        input.put("EMPLOYEENUMBER", new Integer(emplNumber));

        EmployeeRecord employee = new EmployeeRecordImpl();

        if (ix.execute(ixSpec, input, employee))
            return employee.getBenefitsDeduction();
        else
            throw new PayrollException(
                PayrollException.INVAL_EMPL_NUMBER);
    } catch (ResourceException ex) {
```

```
            throw new EJBException(ex);
        } finally {
            try {
                if (cx != null) cx.close();
            } catch (ResourceException ex) {
            }
        }
    }

    public void setSalary(int emplNumber, double salary)
        throws PayrollException
    {
        Connection cx = null;
        try {
            cx = getConnection();
            Interaction ix = cx.createInteraction();

            InteractionSpecImpl ixSpec = new InteractionSpecImpl();
            ixSpec.setFunctionName("SETPAYROLL_SALARY");
            ixSpec.setInteractionVerb(InteractionSpec.SYNC_SEND);

            RecordFactory rf = ix.getRecordFactory();

            MappedRecord input =
                rf.createMappedRecord("PAYROLLINFO_SALARY");
            input.put("EMPLOYEENUMBER", new Integer(emplNumber));
            input.put("SALARY", new Double(salary));

            ix.execute(ixSpec, input);
        } catch (ResourceException ex) {
            throw new EJBException(ex);
        } finally {
            try {
                if (cx != null) cx.close();
            } catch (ResourceException ex) {
            }
        }
    }
```

```
public double getSalary(int emplNumber)
    throws PayrollException
{
    Connection cx = null;
    try {
        cx = getConnection();
        Interaction ix = cx.createInteraction();

        InteractionSpecImpl ixSpec = new InteractionSpecImpl();
        ixSpec.setFunctionName("GETPAYROLLDATA");
        ixSpec.setInteractionVerb(
            InteractionSpec.SYNC_SEND_RECEIVE);

        RecordFactory rf = ix.getRecordFactory();

        MappedRecord input =
            rf.createMappedRecord("EMPLOYEEINFO");
        input.put("EMPLOYEENUMBER", new Integer(emplNumber));

        EmployeeRecord employee = new EmployeeRecordImpl();

        if (ix.execute(ixSpec, input, employee))
            return employee.getSalary();
        else
            throw new PayrollException(
                PayrollException.INVAL_EMPL_NUMBER);
    } catch (ResourceException ex) {
        throw new EJBException(ex);
    } finally {
        try {
            if (cx != null) cx.close();
        } catch (ResourceException ex) {
        }
    }
}

public void ejbCreate() {}
public void ejbRemove() {}
public void ejbPassivate() {}
```

```
    public void ejbActivate() {}
    public void setSessionContext(SessionContext sc) {}

    private Connection getConnection() {
        try {
            Connection cx = connectionFactory.getConnection();
            return cx;
        } catch (ResourceException ex) {
            throw new EJBException(ex);
        }
    }

    private void readEnvironment() {
        try {
            Context nc = new InitialContext();

            connectionFactory = (ConnectionFactory)nc.lookup(
                "java:comp/env/eis/ConnectionFactory");
        } catch (NamingException ex) {
            throw new EJBException(ex);
        }
    }
}
```

Code Example 7.4 PayrollBean Class

Let's look at the implementation of the `setBenefitsDeduction` method. It first obtains a `Connection` object to use for communication with the mainframe. It uses the `Connection` object to create an `Interaction` object. The `Interaction` object is used for a single interaction with the mainframe.

Then, the `setBenefitsDeduction` method creates an `InteractionSpecImpl` object and sets on it the name of the target TP mainframe program. It also specifies the direction of the interaction. In our case, the `SYNC_SEND` verb indicates that arguments pass only to the mainframe.

The `setBenefitsDeduction` method next creates a `MappedRecord` object and uses the `put` methods to set the values of input arguments to be sent to the target mainframe program.

Finally, the `setBenefitsDeduction` method invokes the `execute` method on the `Interaction` object, which causes the resource adapter to send the input arguments to the `SETPAYROLL_DEDUCTION` program on the mainframe. Under the covers, the EJB Container and the mainframe resource adaptor propagate the transaction from the PayrollBean instance to the mainframe TP program. The developer of the PayrollBean did not have to write any transaction-related code to enable this propagation of the transaction.

Refer to “CCI Interface Classes” on page 441 in Appendix B for a more detailed description of the classes that implement the CCI interface.

7.3 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 the information about employees. It also needs access to the payroll system so that it can update benefits-related paycheck deductions. In addition, the enrollment application needs to be integrated at the customer site with the plan-specific components provided by the insurance companies.

Wombat uses entity bean components to integrate the benefits enrollment application with applications or databases from other parties.

7.3.1 Overview of the Wombat Parts

Wombat develops the Web applications and enterprise beans, which are illustrated in Figure 7.3. Each application and enterprise bean is detailed throughout this section.

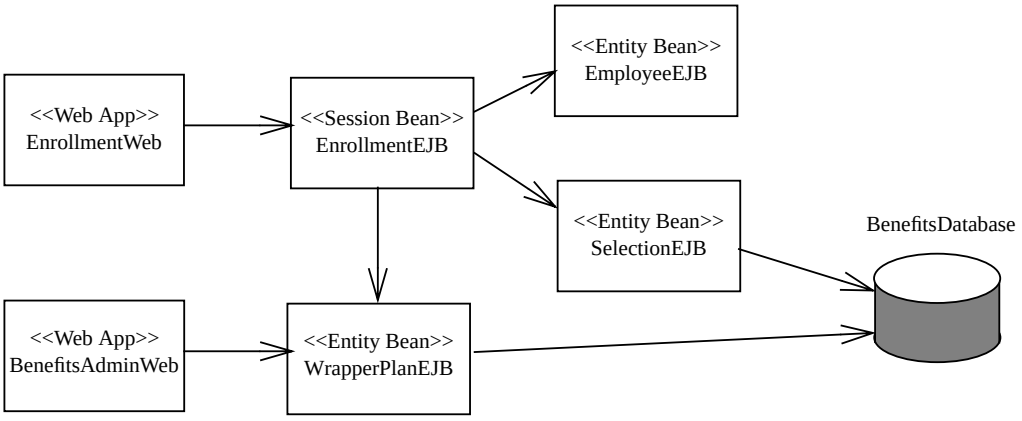

Figure 7.3 Web Applications and Enterprise Beans Developed by Wombat

- **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. The benefits administration department uses this web application 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 that encapsulates access to a customer's employee information. It is an entity bean with container-managed persistence so that it can accommodate the different representations of employee databases at different customer sites. Container-managed persistence allows deployment-time binding with the customer's employee database.

- **SelectionEJB**—An entity bean that encapsulates the benefits selections chosen by each employee.
- **WrapperPlanEJB**—An entity bean that aggregates the medical and dental plans from multiple providers (that is, insurance companies). It takes the different plans from multiple providers and "wraps" them into a single component, thus simplifying application development. WrapperPlanEJB insulates the rest of the enrollment application from the configuration of the individual medical and dental plans. Without this entity bean, the application would have to be coded to handle several different health plans; with this entity bean, the application has only to deal with a single component.

The enterprise beans developed by Wombat store information in BenefitsDatabase. Wombat designs the BenefitsDatabase schema and at deployment time the customer creates the database on his own site. Wombat also allows the customer to choose a different schema as a deployment option.

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

7.3.2 EnrollmentEJB Session Bean

EnrollmentEJB is a stateful session bean that implements the benefits enrollment business process. EnrollmentEJB's home and remote interfaces are the same as in the Chapter 4, Working with Session Beans, example. However, while an IT developer at Star Enterprise defined the interfaces in the session bean chapter, Wombat defined the home and remote interfaces shown in this chapter.

Code Example 7.5 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 7.5 EnrollmentHome Home Interface Defined by Wombat

Code Example 7.6 shows the definition of the EnrollmentEJB's remote 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;
    void setDentalOption(int choice)
        throws RemoteException, EnrollmentException;
    boolean getSmokerStatus()
        throws RemoteException, EnrollmentException;
    void setSmokerStatus(boolean status)
        throws RemoteException, EnrollmentException;
    Summary getSummary()
        throws RemoteException, EnrollmentException;
    void commitSelections()
        throws RemoteException, EnrollmentException;
```

```
}
```

Code Example 7.6 Enrollment Remote Interface Defined by Wombat

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:

- The EnrollmentEJB in Chapter 4 uses command beans to access the employee's database. The EnrollmentEJB in this chapter uses the EmployeeEJB entity bean to encapsulate access to the employee information. Because the EmployeeEJB entity bean is implemented with container-managed persistence, the Deployer can bind the EmployeeEJB bean with the customer's employee database in a standard way.
- The EnrollmentEJB in Chapter 4 uses command beans to access the benefits selection in the BenefitsDatabase. The EnrollmentEJB in this chapter uses the SelectionEJB entity bean to encapsulate the access to the employees' current selections. This approach is advantageous because the benefits selection access is available, via the Selection and SelectionHome client-view interfaces, to other applications regardless of their location on the network. For example, the Star Enterprise IT department can develop other applications that use the Selection and SelectionHome client-view interfaces to access the employees' benefits selections. In addition, because Wombat also provides a CMP version of the SelectionEJB entity bean (this is the SelectionBeanCMP class), a customer can customize the format in which the selections are stored, or even store them in a non-relational database.
- EnrollmentEJB uses the WrapperPlanEJB entity bean to access the medical and dental plans currently offered to the employees. WrapperPlanEJB integrates the medical and dental plans provided by multiple insurance companies with the enrollment application. The WrapperPlanEJB offers the same advantages as the SelectionEJB: other Star Enterprise applications can use the WrapperPlanEJB to access the available medical and dental plans. The applications do not have to be concerned with the multiple plan entity beans provided by the the individual insurance companies.

Code Example 7.7 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 the implementation in Chapter 4.)

```
package com.wombat.benefits;

import javax.ejb.*;

import javax.naming.Context;
import javax.naming.InitialContext;
import com.wombat.plan.Plan;
import com.wombat.plan.PlanHome;
import com.wombat.plan.PlanInfo;
import com.aardvark.payroll.Payroll;
import com.aardvark.payroll.PayrollHome;
import java.util.Collection;
import java.util.Iterator;
import javax.rmi.PortableRemoteObject;


// PlanCache is a helper class that encapsulates the access
// to a plan object. It caches the PlanInfo associated with
// the plan object to avoid remote calls to the plan.
//
class PlanCache {
    private Plan plan;
    private PlanInfo planInfo;

    public PlanCache(Plan plan, PlanInfo planInfo) {
        this.plan = plan;
        this.planInfo = planInfo;
    }

    public Plan getPlan() {
        return plan;
    }
    public PlanInfo getPlanInfo() {
```

```
        return planInfo;
    }

    public String getPlanId() {
        return planInfo.getPlanId();
    }
    public String getPlanName() {
        return planInfo.getPlanName();
    }
    public int getPlanType() {
        return planInfo.getPlanType();
    }
    public double getCost(int coverage, int age, boolean smoker) {
        try {
            return plan.getCost(coverage, age, smoker);
        } catch (Exception ex) {
            throw new EJBException("getCost failed");
        }
    }
}

// EnrollmentBean implements the benefits enrollment
// business process.
public class EnrollmentBean implements SessionBean
{
    private final static String[] coverageDescriptions = {
        "Employee Only",
        "Employee and Spouse",
        "Employee, Spouse, and Children"
    };

    // Tables of Java classes that are used for calculation of
    // of cost of medical and dental benefits
    private PlanCache[] medicalPlans;
    private PlanCache[] dentalPlans;

    // Portion of the benefits cost paid by the employee.
    // (A real-life application would read this value from some
    // benefits plan configuration database)
```

```
    private double employeeCostFactor = 0.10;

    // Employee number that uniquely identifies an employee
    private int employeeNumber;

    private EmployeeHome employeeHome;
    private Employee employee;
    private EmployeeCopy employeeCopy;

    private SelectionHome selectionHome;
    private Selection selection;
    private SelectionCopy selCopy;

    // Indication if a selection record exist for an employee
    private boolean recordDoesNotExist = false;

    // The following variables are calculated values and are
    // used for programming convenience.
    private int age;// employee's age
    private int medicalSelection = -1;// index to medicalPlans
    private int dentalSelection = -1;// index to dentalPlans
    private double totalCost;// total benefits cost
    private double payrollDeduction;// payroll deduction

    private Payroll payroll;
    private PayrollHome payrollHome;

    private PlanHome planHome;

    // public no-arg constructor
    public EnrollmentBean() { }

    // business methods follow

    // Get employee information.
    public EmployeeInfo getEmployeeInfo() {
        return new EmployeeInfo(employeeNumber,
            employeeCopy.getFirstName(),
            employeeCopy.getLastName());
```

```
}

// Get coverage options.
public Options getCoverageOptions() {
    Options opt = new Options(coverageDescriptions.length);
    opt.setOptionDescription(coverageDescriptions);
    opt.setSelectedOption(selCopy.getCoverage());
    return opt;
}

// Set selected coverage option.
public void setCoverageOption(int choice)
        throws EnrollmentException {
    if (choice >= 0 && choice < coverageDescriptions.length) {
        selCopy.setCoverage(choice);
    } else {
        throw new EnrollmentException(
            EnrollmentException.INVAL_PARAM);
    }
}

// Get list of available medical options.
public Options getMedicalOptions() {
    Options opt = new Options(medicalPlans.length);
    for (int i = 0; i < medicalPlans.length; i++) {
        PlanCache plan = medicalPlans[i];
        opt.setOptionDescription(i, plan.getPlanName());
        opt.setOptionCost(i, plan.getCost(
            selCopy.getCoverage(),
            age, selCopy.getSmokerStatus()));
    }
    opt.setSelectedOption(medicalSelection);
    return opt;
}

// Set selected medical option.
public void setMedicalOption(int choice)
        throws EnrollmentException
{
```

```
        if (choice >= 0 && choice < medicalPlans.length) {
            medicalSelection = choice;
            selCopy.setMedicalPlan(medicalPlans[choice].getPlan());
        } else {
            throw new EnrollmentException(
                EnrollmentException.INVAL_PARAM);
        }
    }

    // Get list of available dental options.
    public Options getDentalOptions() {
        Options opt = new Options(dentalPlans.length);
        for (int i = 0; i < dentalPlans.length; i++) {
            PlanCache plan = dentalPlans[i];
            opt.setOptionDescription(i, plan.getPlanName());
            opt.setOptionCost(i, plan.getCost(
                selCopy.getCoverage(),
                age, selCopy.getSmokerStatus()));
        }
        opt.setSelectedOption(dentalSelection);
        return opt;
    }

    // Set selected dental option.
    public void setDentalOption(int choice)
            throws EnrollmentException
    {
        if (choice >= 0 && choice < dentalPlans.length) {
            dentalSelection = choice;
            selCopy.setDentalPlan(dentalPlans[choice].getPlan());
        } else {
            throw new EnrollmentException(
                EnrollmentException.INVAL_PARAM);
        }
    }

    // Get smoker status.
    public boolean getSmokerStatus() {
        return selCopy.getSmokerStatus();
```

```
}

// Set smoker status.
public void setSmokerStatus(boolean status) {
    selCopy.setSmokerStatus(status);
}

// Get summary of selected options and their cost.
public Summary getSummary() {
    calculateTotalCostAndPayrollDeduction();
    try {
        Summary s = new Summary();
        s.setCoverageDescription(
            coverageDescriptions[selCopy.getCoverage()]);
        s.setSmokerStatus(selCopy.getSmokerStatus());
        s.setMedicalDescription(
            medicalPlans[medicalSelection].getPlanName());
        s.setMedicalCost(
            medicalPlans[medicalSelection].getCost(
                selCopy.getCoverage(),
                age, selCopy.getSmokerStatus()));
        s.setDentalDescription(
            dentalPlans[dentalSelection].getPlanName());
        s.setDentalCost(
            dentalPlans[dentalSelection].getCost(
                selCopy.getCoverage(),
                age, selCopy.getSmokerStatus()));
        s.setTotalCost(totalCost);
        s.setPayrollDeduction(payrollDeduction);
        return s;
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

// Update corporate databases with the new selections.
public void commitSelections() {
    try {
        if (recordDoesNotExist) {
```

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

            // Update information in the payroll system
            payroll.setBenefitsDeduction(employeeNumber,
                payrollDeduction);
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }

    // Initialize the state of the EmployeeBean instance.
    public void ejbCreate(int emplNum) throws EnrollmentException {
        employeeNumber = emplNum;

        // Obtain values from bean's environment.
        readEnvironmentEntries();

        try {
            Collection coll;
            Iterator it;

            coll = planHome.findMedicalPlans();
            medicalPlans = new PlanCache[coll.size()];
            it = coll.iterator();
            for (int i = 0; i < medicalPlans.length; i++) {
                Plan plan = (Plan)PortableRemoteObject.narrow(
                        it.next(), Plan.class);
                medicalPlans[i] = new PlanCache(
                    plan, plan.getPlanInfo());
            }

            coll = planHome.findDentalPlans();
            dentalPlans = new PlanCache[coll.size()];
            it = coll.iterator();
            for (int i = 0; i < dentalPlans.length; i++) {
```

```
    Plan plan = (Plan)PortableRemoteObject.narrow(
            it.next(), Plan.class);
    dentalPlans[i] = new PlanCache(
        plan, plan.getPlanInfo());
}

try {
    employee = employeeHome.findByPrimaryKey(
           new Integer(emplNum));
} catch (ObjectNotFoundException ex) {
    throw new EnrollmentException(
        "employee not found");
}
employeeCopy = employee.getCopy();

selection = selectionHome.findByEmployee(employee);
if (selection == null) {
    // This is the first time that the employee
    // runs this application. Use values
    // for the selections.
    selCopy = new SelectionCopy();
    selCopy.setEmployee(employee);
    selCopy.setCoverage(0);
    selCopy.setMedicalPlan(medicalPlans[0].getPlan());
    selCopy.setDentalPlan(dentalPlans[0].getPlan());
    selCopy.setSmokerStatus(false);
    recordDoesNotExist = true;
} else {
    selCopy = selection.getCopy();
}

// Calculate employee's age.
java.util.Date today = new java.util.Date();
age = (int)((today.getTime() -
    employeeCopy.getBirthDate().getTime()) /
      ((long)365 * 24 * 60 * 60 * 1000));

// Translate the medical plan id to an index
// into the medicalPlans table.
```

```
            String medicalPlanId = (String)
                selCopy.getMedicalPlan().getPrimaryKey();
            for (int i = 0; i < medicalPlans.length; i++) {
                if (medicalPlans[i].getPlanId().
                        equals(medicalPlanId)) {
                    medicalSelection = i;
                    break;
                }
            }

            // Translate the dental plan id to an index
            // into the dentalPlans table.
            String dentalPlanId = (String)
                selCopy.getMedicalPlan().getPrimaryKey();
            for (int i = 0; i < dentalPlans.length; i++) {
                if (dentalPlans[i].getPlanId().
                        equals(dentalPlanId)) {
                    dentalSelection = i;
                    break;
                }
            }

            // create a payroll session object
            payroll = (Payroll)payrollHome.create();
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }

    // Clean up any resource held by the instance.
    public void ejbRemove() {
        try {
            payroll.remove();
        } catch (Exception ex) {
        }
    }

    public void ejbPassivate() {}
    public void ejbActivate() {}
```

```
public void setSessionContext(SessionContext sc) {}

// Helper methods follow

// Calculate total benefits cost and payroll deduction
private void calculateTotalCostAndPayrollDeduction() {
    try {
        double medicalCost =
            medicalPlans[medicalSelection].getCost(
                selCopy.getCoverage(),
                age, selCopy.getSmokerStatus());

        double dentalCost =
            dentalPlans[dentalSelection].getCost(
                selCopy.getCoverage(),
                age, selCopy.getSmokerStatus());

        totalCost = medicalCost + dentalCost;
        payrollDeduction = totalCost * employeeCostFactor;
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

// Read and process enterprise bean's environment entries.
private void readEnvironmentEntries() {
    try {
        Context ictx = new InitialContext();
        planHome = (PlanHome)
                PortableRemoteObject.narrow(ictx.lookup(
                        "java:comp/env/ejb/PlanEJB"),
                    PlanHome.class);
        employeeHome = (EmployeeHome)
                PortableRemoteObject.narrow(ictx.lookup(
                        "java:comp/env/ejb/EmployeeEJB"),
                    EmployeeHome.class);
        selectionHome = (SelectionHome)
                PortableRemoteObject.narrow(ictx.lookup(
                    "java:comp/env/ejb/SelectionEJB"),
```

```
                    SelectionHome.class);
            payrollHome = (PayrollHome)
                    PortableRemoteObject.narrow(ictx.lookup(
                            "java:comp/env/ejb/PayrollEJB"),
                        PayrollHome.class);
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }
}
```

Code Example 7.7 EnrollmentBean Class Implementation

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 WrapperPlanEJB entity beans.

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

```
selection = selectionHome.findByEmployee(employee);
```

After EnrollmentEJB obtains an object reference to the Selection object, it invokes a business method on the object. Here, it invokes the Selection object's `getCopy` method to read the current benefit 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.

7.3.3 EmployeeEJB Entity Bean

EmployeeEJB is an entity bean with container-managed persistence. It provides an object-oriented view of the employee data used by the benefits enrollment application. Its main role is to allow the integration between the benefits application and customer's employee data.

Since Wombat does not impose rules on a customer regarding how it stores the information about its employees, it must give its customers the means to integrate the application with the customer's employee data. Wombat uses the container-managed persistence mechanism to allow the Deployer to bind the EmployeeEJB with an existing employee database.

However, Wombat designs the Employee and EmployeeHome interfaces to meet the needs of the benefits enrollment application.

7.3.3.1 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 type `int`. This is because the EJB specification requires that the primary key type for an entity bean be a Java class. Furthermore, this specification implies that primitives types that are not Java classes cannot be used directly for the primary key type.

7.3.3.2 Employee Remote Interface

Code Example 7.8 shows the Employee remote interface definition.

```
package com.wombat.benefits;

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

public interface Employee extends EJBObject {
    EmployeeCopy getCopy()
        throws RemoteException, EmployeeException;
    void updateFromCopy(EmployeeCopy copy)
        throws RemoteException, EmployeeException;
}
```

Code Example 7.8 Employee Remote Interface

The enrollment application needs access to each employee's employee number, first name, last name, and date of birth. To efficiently exchange the information between the employee object and the client application (the client is the EnrollmentBean in our example application), Wombat defines the `getCopy` and `updateFromCopy` methods. The `getCopy` method returns all the employee attributes to the client, and the `updateFromCopy` method updates the employee object using the modified values passed by the client.

The `getCopy` and `updateFromCopy` methods use the EmployeeCopy class, which is defined as shown in Code Example 7.9.

```
package com.wombat.benefits;

import java.util.Date;

public class EmployeeCopy implements java.io.Serializable {
    private int employeeNumber;
    private String firstName;
    private String lastName;
    private Date birthDate;
```

```
    public int getEmployeeNumber() { return employeeNumber; }
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public Date getBirthDate() { return birthDate; }

    public void setEmployeeNumber(int v) { employeeNumber = v; }
    public void setFirstName(String v) { firstName = v; }
    public void setLastName(String v) { lastName = v; }
    public void setBirthDate(Date v) { birthDate = v; }
}
```

Code Example 7.9 EmployeeCopy Class

Note that the EmployeeCopy class implements the `java.io.Serializable` interface so that its instances can be passed by value over RMI-IIOP. Refer to the EmployeeBean class in Code Example 7.11 on page 276 to see how a client uses the information returned in the EmployeeCopy object.

7.3.3.3 EmployeeHome Home Interface

Code Example 7.10 shows the EmployeeHome interface definition:

```
package com.wombat.benefits;

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

public interface EmployeeHome extends EJBHome {
    // finder methods
    Employee findByPrimaryKey(Integer employeeNumber)
        throws RemoteException, FinderException;
}
```

Code Example 7.10 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.

7.3.3.4 EmployeeBean Entity Bean Class

The EmployeeBean class illustrates how simple it is to develop an entity bean with container-managed persistence. Notice that the entity bean contains no database operations. “Summary of the Integration Techniques” on page 362 describes how the deployer binds the container-managed fields with the columns of the pre-existing EmployeeDatabase at Star Enterprise. Note also that the entity bean class contains no implementations of any finder methods, although the home interface defines the `findByPrimaryKey` method. Because the EmployeeEJB is an entity bean with container-managed persistence, the implementations of finder methods have to be supplied at deployment time. Wombat describes in the deployment descriptor those objects that the finder methods are supposed to return. The Deployer then implements the finder methods to return the specified objects.

Code Example 7.11 below shows the source code of the EmployeeBean entity bean class.

```
package com.wombat.benefits;

import javax.ejb.*;
import java.util.Date;

public class EmployeeBean implements EntityBean {
    //
    // Container-managed fields
    //
    public int employeeNumber;
    public String firstName;
    public String lastName;
    public Date birthDate;

    public EmployeeCopy getCopy() {
        EmployeeCopy ec = new EmployeeCopy();
        ec.setEmployeeNumber(employeeNumber);
        ec.setFirstName(firstName);
```

```
        ec.setLastName(lastName);
        ec.setBirthDate(birthDate);
        return ec;
    }

    public void updateFromCopy(EmployeeCopy ec)
        throws EmployeeException
    {
        if (ec.getEmployeeNumber() != employeeNumber) {
            throw new EmployeeException(
                "can't change primary key");
        } else {
            firstName = ec.getFirstName();
            String newLastName = ec.getLastName();
            if (newLastName == null ||
                newLastName.length() == 0)
                throw new EmployeeException(
                        "last name can't be blank");
            lastName = newLastName;
            birthDate = ec.getBirthDate();
        }
    }

    //
    // There are no ejbCreate(...) methods.
    //

    //
    // Methods from EntityBean interface
    //
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbPassivate() {}
}
```

Code Example 7.11 EmployeeBean Class Implementation

The implementation of an entity bean class of an entity bean with container-managed persistence is generally straightforward. The EmployeeBean class is a good example of a straightforward implementation. Its `updateFromCopy` method provides a good illustration of a simple business rules validation. The method checks that the new last name is not blank. It throws an exception if the client passes an empty string for the value for last name.

However, 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. The EJB specification mandates that the container-managed fields be defined as public, even though a client program never directly accesses them. These fields must be public so that the container can move the data between the fields and the database to keep the content of the fields synchronized with the information in the database.

The example also illustrates that the primary key of an entity object is not allowed to change. Changing an entity object's primary key would result in confusion about object identity in a distributed application. The entity object should throw an exception if a client attempts to change the primary key. For example, the `updateFromCopy` method throws an `EmployeeException` if a client attempts to change its primary key.

7.3.4 SelectionEJB Entity Bean

There are two approaches for dealing with entity bean persistence. An entity bean can use either *bean-managed persistence* (BMP) or *container-managed persistence* (CMP). To provide a choice to its customers, Wombat developed two versions of the SelectionEJB entity bean. One uses CMP, and the other uses BMP.

Wombat first developed the SelectionEJBCMP entity bean as an entity bean with container-managed persistence. An entity bean with container-managed persistence can be developed independently from the underlying database schema. With container-managed persistence, the Deployer binds the database table columns to the entity bean fields at deployment time by using tools provided by the Container. Wombat's SelectionEJBCMP entity bean is independent from the schema of the database that stores the persistent state of the Selection objects. It is

even independent from the type of the database in which the state of the Selection objects are stored. Selection objects can be stored in a relational database or a non-relational database.

After developing the SelectionEJBCMP entity bean, Wombat defines the SelectionEJB entity bean with bean-managed persistence; this entity bean uses a default relational database schema defined by Wombat. The benefit of using the default schema is simpler deployment, because the Deployer does not have to set up the binding between the container-managed fields and the database schema. The SelectionBean entity bean class is a subclass of the SelectionBeanCMP entity bean class. The Java class hierarchy is illustrated in Figure 7.4:

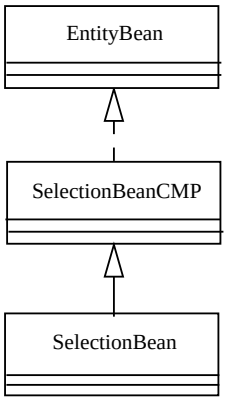

Figure 7.4 Entity Bean Persistence Class Hierarchy

Wombat defines both the container-managed and bean-managed entity beans so that it can support customers who want to use a custom database schema and those content to use the Wombat-supplied default database schema. Wombat expects that most customers will use the default schema. Customers using the default database schema can use the SelectionEJB entity bean. Customers who want to use a custom database schema can use the SelectionEJBCMP entity bean and define their own binding of the SelectionEJBCMP container-managed fields to the schema.

7.3.4.1 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`.

7.3.4.2 Selection Remote Interface

Wombat designed the SelectionEJB's remote interface to meet the needs of its client, the EnrollmentEJB session bean. The remote interface uses the SelectionCopy value object to pass the information between the SelectionEJB entity bean and its client. It uses the `getCopy` and `updateFromCopy` design pattern that was also used in the `Employee` remote interface.

The Selection interface definition is shown in Code Example 7.12:

```
package com.wombat.benefits;

import javax.ejb.*;
import java.rmi.RemoteException;
import com.wombat.plan.Plan;

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

Code Example 7.12 Selection Interface Definition

The Selection interface defines methods to obtain and update an employee's benefits selection. In the Wombat Benefits application, the EnrollmentEJB session bean, which is the client of SelectionEJB, uses the `getCopy` method to obtain a copy of the employee's benefits selection and the `updateFromCopy` method to change the selection.

The Selection interface uses the SelectionCopy value object, which is defined as shown in Code Example 7.13:

```
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 getSmokerStatus() { return smokerStatus; }
    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 setSmokerStatus(boolean v) { smokerStatus = v; }
}
```

Code Example 7.13 SelectionCopy Value Object

The SelectionCopy class defines private fields to hold the information about an employee's benefits selection. It also defines public accessor methods (that is, the getter and setter methods) for each field.

The CoverageCategory interface (see Code Example 7.14) defines the integer values that represent the different coverage categories.

```
package com.wombat.benefits;

public class CoverageCategory {
    public static final int EMPLOYEE_ONLY = 0;
    public static final int EMPLOYEE_SPOUSE = 1;
    public static final int EMPLOYEE_SPOUSE_CHILDREN = 2;
}
```

Code Example 7.14 CoverageCategory Interface

7.3.4.3 SelectionHome Home Interface

Code Example 7.15 shows the definition for the SelectionHome home interface.

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

```
import javax.ejb.*;
import java.rmi.RemoteException;
import com.wombat.plan.Plan;
import java.util.Collection;

public interface SelectionHome extends EJBHome {
    // create methods
    Selection create(SelectionCopy copy)
        throws RemoteException, CreateException;

    // finder methods
    Selection findByPrimaryKey(Integer emplNumber)
        throws RemoteException, FinderException;
    Selection findByEmployee(Employee employee)
        throws RemoteException, FinderException;
    Collection findByPlan(Plan plan)
        throws RemoteException, FinderException;
}
```

Code Example 7.15 SelectionHome Home Interface

Notice how the SelectionHome interface uses the SelectionCopy object as the argument of the `create` method. A client uses this method to create an entity object that stores an employee's benefits selections from a copy of the information passed by the client.

The SelectionHome interface also defines three entity bean finder methods. The mandatory `findByPrimaryKey` method finds the Selection object by the employee number. The `findByEmployee` method finds the Selection object for a given Employee object. The `findByPlan` finder methods finds a collection of Selection objects that contain a reference to a given Plan.

The Benefits application does not use the `findByEmployee` and `findByPlan` finder methods. Wombat includes these definitions so that other applications developed by Wombat customers can more easily reuse SelectionEJB. (We include them to illustrate how to implement more complex finder methods).

7.3.4.4 SelectionBeanCMP Entity Bean Class

Code Example 7.16 illustrates the SelectionBeanCMP entity bean class implementation.

```
package com.wombat.benefits;

import javax.ejb.*;

import javax.naming.Context;
import javax.naming.InitialContext;

import com.wombat.plan.Plan;
import com.wombat.plan.PlanType;

public class SelectionBeanCMP implements EntityBean
{
    // container-managed fields
    public int coverage;
    public boolean smokerStatus;
    public Employee employee;
    public Plan medicalPlan;
    public Plan dentalPlan;
    public Integer employeeNumber;// primary key field

    // values obtained from environment
    boolean checkPlanType;

    // Helper methods.
    //
    void updateCoverage(int v) throws SelectionException {
        switch (coverage) {
        case CoverageCategory.EMPLOYEE_ONLY:
        case CoverageCategory.EMPLOYEE_SPOUSE:
        case CoverageCategory.EMPLOYEE_SPOUSE_CHILDREN:
            coverage = v;
            break;
        default:
```

```
            throw new SelectionException(
                SelectionException.INVAL_COVERAGE);
        }
    }

    void updateMedicalPlan(Plan p) throws SelectionException {
        if (checkPlanType) {
            int type;
            try {
                type = p.getPlanType();
            } catch (Exception ex) {
                throw new EJBException(ex);
            }
            if (type != PlanType.MEDICAL)
                throw new SelectionException(
                    SelectionException.INVAL_PLAN_TYPE);
        }
        medicalPlan = p;
    }

    void updateDentalPlan(Plan p) throws SelectionException {
        if (checkPlanType) {
            int type;
            try {
                type = p.getPlanType();
            } catch (Exception ex) {
                throw new EJBException(ex);
            }
            if (type != PlanType.DENTAL)
                throw new SelectionException(
                    SelectionException.INVAL_PLAN_TYPE);
        }
        dentalPlan = p;
    }

    void updateSmokerStatus(boolean v) { smokerStatus = v; }

    // Business methods from remote interface
    //
```

```
public SelectionCopy getCopy() {
    SelectionCopy copy = new SelectionCopy();
    copy.setEmployee(employee);
    copy.setCoverage(coverage);
    copy.setMedicalPlan(medicalPlan);
    copy.setDentalPlan(dentalPlan);
    copy.setSmokerStatus(smokerStatus);
    return copy;
}

public void updateFromCopy(SelectionCopy copy)
    throws SelectionException
{
    try {
        if (!employeeNumber.equals(
            copy.getEmployee().getPrimaryKey()))
            throw new SelectionException(
                "can't change primary key");
    } catch (java.rmi.RemoteException ex) {
        throw new EJBException(ex);
    }
    updateMedicalPlan(copy.getMedicalPlan());
    updateDentalPlan(copy.getDentalPlan());
    updateSmokerStatus(copy.getSmokerStatus());
    updateCoverage(copy.getCoverage());
}

// create(...) methods from home interface
//
public Integer ejbCreate(SelectionCopy copy)
    throws SelectionException, CreateException
{
    employee = copy.getEmployee();
    try {
        employeeNumber = (Integer)employee.getPrimaryKey();
    } catch (java.rmi.RemoteException ex) {
        throw new EJBException(ex);
    }
    updateMedicalPlan(copy.getMedicalPlan());
```

```
            updateDentalPlan(copy.getDentalPlan());
            updateSmokerStatus(copy.getSmokerStatus());
            updateCoverage(copy.getCoverage());
            return null; // ejbCreate returns null in CMP beans
        }

        public void ejbPostCreate(SelectionCopy copy) {}

        // Methods from EntityBean interface
        //
        public void setEntityContext(EntityContext ctx) {
            readEnvironment();
        }
        public void unsetEntityContext() {}
        public void ejbRemove() {}
        public void ejbActivate() {}
        public void ejbLoad() {}
        public void ejbStore() {}
        public void ejbPassivate() {}

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

Code Example 7.16 SelectionBeanCMP Entity Bean Class Implementation

First, let's examine the container-managed fields: `coverage`, `smokerStatus`, `employee`, `medicalPlan`, `dentalPlan`, and `employeeNumber`. Notice that the `employee`, `medicalPlan`, and `dentalPlan` fields are references to other enterprise

beans (Employee and Plan, respectively). The EJB specification allows the container-managed fields to include references to other enterprise beans. Allowing container-managed fields to include references to other enterprise beans has two benefits, as follows:

- It simplifies the development of the SelectionBeanCMP methods because they can work directly with object references rather than having to convert object references to primary keys.
- It avoids hardcoding the database representation of the relationships to the other entity beans into the SelectionBeanCMP class; it leaves a Deployer free to choose how to represent the relationships in the underlying database schema.

See "SelectionBean Entity Bean Class" on page 287 for an explanation of how Wombat implemented the representation of the entity object relationships in the SelectionBean class.

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

- The business methods defined in the Selection remote interface
- The `ejbCreate` and `ejbPostCreate` methods that correspond to the `create` method defined in the SelectionHome interface
- The container callbacks defined in the EntityBean interface

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

The business methods and the `ejbCreate` method 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` method checks that the value of the coverage field is an allowed value, while the `updateMedicalPlan` method optionally checks that the value of `medicalPlan` is indeed a medical plan, rather than a dental plan. The `getCopy`, `updateFromCopy`, and `ejb-`

`Create` methods illustrate how the entity bean class can implement handling value objects passed between the entity bean and the client. The SelectionBeanCMP class (within its `setEmployee` and `updateFromCopy` methods) enforces the rule that a client cannot change the primary key of a Selection object.

The SelectionBeanCMP class (within the `readEnvironment` helper method) makes access to the environment entry available with the key `java:comp/env/checkPlanType`. 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 time. Wombat made the plan type checks optional to allow the application assembler to improve performance by omitting the checks 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 time).

The `ejbCreate` method sets up the container-managed fields from the values passed to it in the method parameter. 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 `ejbCreate` returns a null value even though the return value type is declared to be the primary key type. According to the EJB specification, the container ignores the value returned from an `ejbCreate` method of an entity bean with container-managed persistence. However, the EJB specification requires that the type of the `ejbCreate` method be the primary key type to allow a subclass of the SelectionBeanCMP class to be an entity bean with bean-managed persistence. The SelectionBean class illustrates this use of subclassing.

Notice that most of the container callbacks inherited from the EntityBean interface have an empty implementation. More complex entity beans with container-managed persistence might use non-empty implementations of these callback methods. See “WrapperPlanBeanCMP Entity Bean Class” on page 303 for an illustration of the use of the `ejbLoad` method in an entity bean with container-managed persistence.

7.3.4.5 SelectionBean Entity Bean Class

There are many possible ways to implement an entity bean's persistence; that is, its database access code. The SelectionBean class illustrates one such way: it uses subclassing to convert an entity bean with container-managed persistence into an entity bean with bean-managed persistence. This is accomplished by subclassing the SelectionBeanCMP entity bean class. While in our example we manually coded the SelectionBean class, most real development and deployment scenarios use tools to automatically generate the database access code.

Code Example 7.17 shows the implementation of the SelectionBean class.

```
package com.wombat.benefits;

import javax.ejb.*;

import javax.naming.InitialContext;
import javax.naming.Context;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;
import java.util.Vector;
import java.util.Collection;
import javax.rmi.PortableRemoteObject;

import com.wombat.plan.Plan;
import com.wombat.plan.PlanHome;
import com.wombat.plan.PlanInfo;
import com.wombat.plan.PlanType;

public class SelectionBean extends SelectionBeanCMP
{
    private EntityContext entityContext;
    private DataSource ds;
    private EmployeeHome employeeHome;
    private PlanHome planHome;
```

```
//
// create(...) methods from home interface
//
public Integer ejbCreate(SelectionCopy copy)
    throws SelectionException, CreateException
{
    super.ejbCreate(copy);
    try {
        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
            "INSERT INTO Selections " +
            "VALUES (?, ?, ?, ?, ?)"
        );
        pstmt.setInt(1, employeeNumber.intValue());
        pstmt.setInt(2, coverage);
        pstmt.setString(3, (String)medicalPlan.getPrimaryKey());
        pstmt.setString(4, (String)dentalPlan.getPrimaryKey());
        pstmt.setString(5, smokerStatus ? "Y" : "N");
        if (pstmt.executeUpdate() == 1) {
            con.close();
            return employeeNumber;
        } else {
            con.close();
            throw new CreateException();
        }
    } catch (SQLException ex) {
        throw new EJBException(ex);
    } catch (java.rmi.RemoteException ex) {
        throw new EJBException(ex);
    }
}

public Integer ejbFindByPrimaryKey(Integer employeeNumber)
    throws FinderException
{
    try {
        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
            "SELECT sel_empl " +
```

```
            "FROM Selections " +
            "WHERE sel_empl = ?"
        );
        pstmt.setInt(1, employeeNumber.intValue());
        ResultSet rs = pstmt.executeQuery();
        if (rs.next()) {
            con.close();
            return employeeNumber;
        } else {
            con.close();
            throw new ObjectNotFoundException();
        }
    } catch (SQLException ex) {
        throw new EJBException(ex);
    }
}

public Integer ejbFindByEmployee(Employee employee)
         throws FinderException
{
    try {
        return ejbFindByPrimaryKey(
            (Integer)employee.getPrimaryKey());
    } catch (java.rmi.RemoteException ex) {
        throw new EJBException(ex);
    }
}

public Collection ejbFindByPlan(Plan plan)
{
    try {
        PlanInfo planInfo = plan.getPlanInfo();
        int planType = planInfo.getPlanType();
        String planId = (String)planInfo.getPlanId();
        String columnName = (planType == PlanType.MEDICAL) ?
            "sel_medical_plan" : "sel_dental_Plan";

        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
```

```
                "SELECT sel_empl " +
                "FROM Selections " +
                "WHERE " + columnName + " = ?"
            );
            pstmt.setString(1, planId);
            ResultSet rs = pstmt.executeQuery();

            Vector vec = new Vector();
            while (rs.next()) {
                int emplnum = rs.getInt(1);
                vec.add(new Integer(emplnum));
            }
            con.close();
            return vec;
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }

    //
    // Methods from EntityBean interface
    //

    public void setEntityContext(EntityContext ctx) {
        readEnvironment();
        super.setEntityContext(ctx);
    }

    public void ejbRemove() {
        super.ejbRemove();
        try {
            Connection con = getConnection();
            PreparedStatement pstmt = con.prepareStatement(
                "DELETE FROM Selections " +
                "WHERE sel_empl = ?"
            );
            pstmt.setInt(1, employeeNumber.intValue());
            pstmt.executeUpdate();
            con.close();
```

```
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

public void ejbLoad() {
    try {
        String medicalPlanId;
        String dentalPlanId;
        employeeNumber =
            (Integer)entityContext.getPrimaryKey();
        employee =
            employeeHome.findByPrimaryKey(employeeNumber);

        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
            “SELECT sel_coverage, sel_smoker, “ +
                “sel_medical_plan,  sel_dental_plan” +
            “FROM Selections “ +
            “WHERE sel_empl = ?”
        );
        pstmt.setInt(1, employeeNumber.intValue());
        ResultSet rs = pstmt.executeQuery();
        if (rs.next()) {
            coverage = rs.getInt(1);
            smokerStatus = rs.getString(2).equals(“Y”);
            medicalPlanId = rs.getString(3);
            dentalPlanId = rs.getString(4);
            con.close();
        } else {
            throw new NoSuchEntityException();
        }
        medicalPlan = planHome.findByPlanId(medicalPlanId);
        dentalPlan = planHome.findByPlanId(dentalPlanId);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
    super.ejbLoad();
}
```

```
public void ejbStore() {
    super.ejbStore();
    try {
        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
            “UPDATE Selections SET “ +
            “sel_coverage = ?, “ +
            “sel_medical_plan = ?, “ +
            “sel_dental_plan = ?, “ +
            “sel_smoker = ? “ +
            “WHERE sel_empl = ?”
        );
        pstmt.setInt(1, coverage);
        pstmt.setString(2, (String)medicalPlan.getPrimaryKey());
        pstmt.setString(3, (String)dentalPlan.getPrimaryKey());
        pstmt.setString(4, smokerStatus ? “Y” : “N”);
        pstmt.setInt(5, employeeNumber.intValue());
        pstmt.executeUpdate();
        con.close();
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

//
// Helper methods.
//

private Connection getConnection() {
    try {
        return ds.getConnection();
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

private void readEnvironment() {
    try {
```

```
            Context ictx = new InitialContext();

            planHome = (PlanHome)
                PortableRemoteObject.narrow(ictx.lookup(
                       “java:comp/env/ejb/PlanEJB”),
                    PlanHome.class);
            employeeHome = (EmployeeHome)
                PortableRemoteObject.narrow(ictx.lookup(
                       “java:comp/env/ejb/EmployeeEJB”),
                   EmployeeHome.class);
            ds = (DataSource)
               ictx.lookup(“java:comp/env/jdbc/BenefitsDB”);
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }
}
```

Code Example 7.17 SelectionBean Class Implementation

The SelectionBean class defines the following methods:

- The `ejbCreate` method. This method overrides the `ejbCreate` method in the SelectionBeanCMP superclass.
- All the finder methods defined in the SelectionHome interface.
- The `ejbLoad`, `ejbStore`, `ejbRemove`, and `setEntityContext` container callbacks inherited from the EntityBean interface. These callbacks override the same-named methods in the superclass.

7.3.4.5.1 Invoking Superclass Methods

The above methods (`ejbCreate`, `ejbLoad`, `ejbStore`, `ejbRemove`, and `setEntityContext`) invoke their corresponding overridden method in the superclass. The timing of the invocation of the same-named superclass method is significant—the order in which a method invokes the same-named method in the superclass relative to the database operations performed in the method is important.

The `ejbCreate`, `ejbRemove`, and `ejbStore` methods invoke the method in the superclass *before* they perform the database operations. The `ejbLoad` method calls the `ejbLoad` in the superclass *after* it performs the database operations. The order of operations ensures the proper synchronization between the values of the container-managed fields, as seen by the superclass methods, and the values of the fields stored in the database.

7.3.4.5.2 NoSuchEntityException

Note that the `ejbLoad` method throws the `NoSuchEntityException` if the representation of the entity object has been removed from the database and the entity bean expects that the representation exists. The `NoSuchEntityException` provides more information to the container than the generic `EJBException` thrown on other database errors. If the bean throws the `NoSuchEntityException`, the container throws the `java.rmi.NoSuchObjectException` to the client to indicate that the reason for a failed method was the removal of the entity object's representation from the database.

7.3.4.5.3 Representing Object References in a Database

Note how the `ejbCreate`, `ejbLoad`, and `ejbStore` methods handle object references to other entity beans maintained in the container-managed fields (that is, the `employee`, `medicalPlan`, and `dentalPlan` fields defined in the SelectionBeanCMP class). The object references are represented as foreign keys in the Selections database table. However, use of a foreign key is only one possible mechanism for representing an object reference. "WrapperPlanEJB Entity Bean" on page 295 describes an entity bean that illustrates how serialized object handles can be used to represent object references in persistent storage.

7.3.4.5.4 Automatic Closing of Connections on Uncaught Exceptions

While it is a good practice to ensure that a bean always closes its database connections, we illustrated that it is not necessary to close a database connection when the bean throws a runtime exception from a method. If the developer has not closed a database connection when a method throws a runtime exception (for example, the EJBException) and the exception is not caught within the bean, the EJB Container will automatically release the connection when it catches the exception. The Container will also remove the instance from its pool because the values of the instance's variables may be in an inconsistent state after the thrown exception.

7.3.5 WrapperPlanEJB Entity Bean

The WrapperPlanEJB entity bean aggregates the medical and dental plans provided by multiple insurance providers into a single entity bean. Aggregation hides the complexity of managing the configuration of plans from multiple providers, and is a benefit for applications that need to access these plans. An example of such an application is the EnrollmentEJB session bean.

Wombat uses the same approach with persistence for WrapperPlanEJB as it did for SelectionEJB. Wombat first defines the WrapperPlanEJBCMP entity bean with container-managed persistence. Wombat then subclasses the WrapperPlanBeanCMP entity bean class to create the WrapperPlanBean entity bean class with bean-managed persistence. This allows the customer to choose between using the WrapperPlanEJB entity bean, which uses the default BenefitsDatabase schema, and the WrapperPlanEJBCMP entity bean, which allows the customer to define a custom database schema, or store the entity bean's state in a non-relational database.

7.3.5.1 Benefits Plan Integration Approach

Wombat wants to allow its customers to configure the Benefits application with multiple medical and dental plans that may be provided by multiple insurance companies. Therefore, Wombat designs the Benefits application to access plan information from multiple sources.

Wombat uses EJB client-view interfaces as the contract between the Benefits application and the software from the individual insurance companies. Wombat defines the Plan remote interface and the PlanHome home interface to be the standard interfaces for integrating the plan information from an insurance company with the Benefits application. If an insurance provider wants to support the Benefits application, the insurance provider (or application integrator) must develop an enterprise bean that uses the Plan and PlanHome interfaces as its client-view interfaces.

Figure 7.5 illustrates the ProvidencePlanEJB developed by Providence (Providence is an insurance company). Note that the "interface implementation" notation in the diagram does not represent interface implementation in the Java language sense. Instead, it shows that the remote and home interfaces of the Prov-

idencePlanEJB entity bean are the Plan and PlanHome interfaces, or their sub-interfaces.

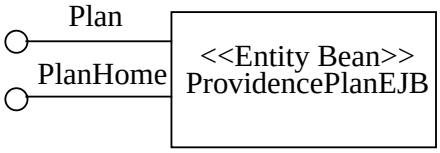

Figure 7.5 ProvidencePlanEJB

7.3.5.2 Aggregation of Plans into a Single Entity Bean

Though enterprise beans from all the insurance providers have a uniform client view (that is, they implement the Plan and PlanHome interfaces), dealing with these multiple enterprise beans can be a tedious task for application programmers. It is tedious because the programmer must know where to locate all the enterprise beans and how to deal with the situation in which enterprise beans are added or removed.

To simplify this task, Wombat develops the WrapperPlanEJB entity bean. WrapperPlanEJB aggregates all the configured enterprise beans from multiple insurance providers into what looks like a single entity bean to the application programmer. Figure 7.6 illustrates this.

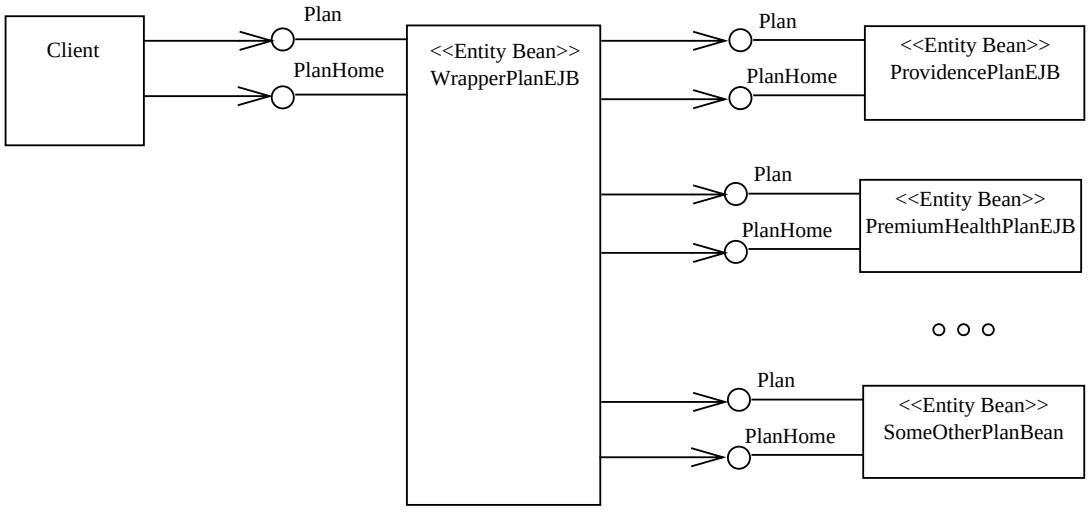

Figure 7.6 WrapperPlanEJB Aggregation of Enterprise Beans

The plan objects implemented by WrapperPlanEJB "wrapper" the plan objects implemented by the entity beans from the individual insurance providers. The implementation of the WrapperPlanEJB business methods delegates to the wrapped plan. The finders implemented in WrapperPlanEJB aggregate the finders in the individual plan entity beans. WrapperPlanEJB "hides" the primary key of the wrapped plan. The primary keys visible to a WrapperPlanEJB's client are customer-assigned plan identifiers that must be unique within the scope of the WrapperPlanHome interface. In our example, the Star Enterprise benefits administration department assigns these primary keys.

A client program deals only with the single WrapperPlanEJB, regardless of how many different insurance providers are used by the customer. A client of the WrapperPlanEJB entity bean uses the Plan and PlanHome interfaces to access the aggregated insurance plan information.

The EnrollmentBean class in "EnrollmentEJB Session Bean" on page 257, which is a WrapperPlanEJB's client, illustrates that the existence of multiple insurance providers is transparent to the client application. The following statement from the EnrollmentBean class finds all the configured medical plans (or, more accurately, their wrapper plan objects) for all the currently enabled medical plans.

```
coll = planHome.findMedicalPlans();
```

7.3.5.3 Plan Remote Interface

The plan entity beans provided by the individual insurance providers must use the Plan interface as their remote interfaces. The WrapperPlanEJB entity bean developed by Wombat also uses the Plan interface as its remote interface.

Code Example 7.18 illustrates the definition of the Plan interface.

```
package com.wombat.plan;

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

public interface Plan extends EJBObject {
    PlanInfo getPlanInfo()
        throws RemoteException, PlanException;
    int getPlanType() throws RemoteException, PlanException;
```

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

Code Example 7.18 Plan Remote Interface

The Plan interface methods do the following operations:

- The `getPlanInfo` method returns a PlanType value object that contains the plan attributes. The PlanType class is described later in this section. (See Code Example 7.20 on page 299.)
- The `getPlanType` method returns an integer value. The value is equal to `PlanType.MEDICAL` if the plan is a medical plan, and it is equal to `PlanType.DENTAL` if the plan is a dental plan.
- The `getCost` method returns the monthly premium charged by the plan provider. The premium depends on the coverage category, age, and smoker status.
- The `getAllDoctors` method returns a `Collection` of Doctor objects that participate in the plan. The `Doctor` class is described below.
- 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.

Code Example 7.19 illustrates the definition of the PlanInfo class.

```
package com.wombat.plan;
```

```
public class PlanInfo implements java.io.Serializable {
    String planId;
    String planName;
    int planType;

    public String getPlanId() { return planId; }
    public String getPlanName() { return planName; }
    public int getPlanType() { return planType; }

    public void setPlanId(String v) { planId = v; }
    public void setPlanName(String v) { planName = v; }
    public void setPlanType(int v) { planType = v; }
}
```

Code Example 7.19 PlanInfo Class

The PlanInfo class is part of the Plan remote interface. The `getPlanInfo` method uses the PlanInfo class to pass a set of plan attributes to the client. The attributes are:

- **`PlanId`**—This is the unique identifier of the plan. It should be the same as the primary key of the Plan entity object.
- **`PlanName`**—This is a short name for the plan. The GUI portion of the application uses this name for display.
- **`PlanType`**—This is either `PlanType.MEDICAL` or `PlanType.DENTAL`.

Code Example 7.20 illustrates the PlanType class, which defines the integer values representing the plan types.

```
package com.wombat.plan;

public class PlanType {
    public static final int MEDICAL = 1;
    public static final int DENTAL = 2;
}
```

Code Example 7.20 PlanType Class

The Doctor class is used to pass information about participating doctors. Note that we do not implement the Doctor class as an entity bean, but instead, implement it as a value object that is passed through the Plan interface. Why do we do this? Our application does not need to view a Doctor object as an entity object. For example, the client does not have the need to invoke methods on the Doctor objects, nor to compare Doctor objects for identity. Rather, the application sees a Doctor object as only a piece of information available through the Plan entity objects. Therefore, we decided not to implement the Doctor class as an entity bean. The Doctor class is shown in Code Example 7.21.

```
package com.wombat.plan;

public class Doctor implements java.io.Serializable {
    String lastName;
    String firstName;
    String specialty;
    String hospital;
    int practiceSince;

    // public get/set methods

    public String getLastName() { return lastName; }
    public void setLastName(String v) { lastName = v; }
    public String getFirstName() { return firstName; }
    public void setFirstName(String v) { firstName = v; }
    public String getSpecialty() { return specialty; }
    public void setSpecialty(String v) { specialty = v; }
    public String getHospital() { return hospital; }
    public void setHospital(String v) { hospital = v; }
    public int getPracticeSince() { return practiceSince; }
    public void setPracticeSince(int v) { practiceSince = v; }
}
```

Code Example 7.21 Implementation of the Doctor Class

7.3.5.4 PlanHome Home Interface

The plan entity beans provided by the individual insurance providers must extend the PlanHome interface in their home interfaces. At the same time, the WrapperPlanHome interface of the WrapperPlanEJB entity bean developed by Wombat also extends the PlanHome interface. Code Example 7.22 shows the definition of the PlanHome interface.

```
package com.wombat.plan;

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

public interface PlanHome extends EJBHome {
    // finder methods
    Plan findByPlanId(String planID)
        throws RemoteException, FinderException;
    Collection findMedicalPlans()
        throws RemoteException, FinderException;
    Collection findDentalPlans()
        throws RemoteException, FinderException;
    Collection findByDoctor(Doctor template)
        throws RemoteException, FinderException;
}
```

Code Example 7.22 PlanHome Home Interface

The PlanHome interface defines the finder methods used by the benefits application. The finder methods are:

- The `findByPlanId` method returns the Plan object for a given plan identifier. The plan identifier is a primary key that uniquely identifies the plan.

- The `findMedicalPlans` method returns all the medical plans configured in this home interface. The objects in the returned `Collection` implement the Plan interface.

- The `findDentalPlans` method returns all the dental plans configured in this

home interface. The objects in the returned `Collection` implement the Plan interface.

- The `findByDoctor` method returns all the plans configured in this home interface that include a specified doctor in their preferred doctors list. A template is used to specify the doctor and to perform a partial match using the first and last name. The objects in the returned `Collection` implement the Plan interface.

Note that the `PlanHome` interface does not define the `findByPrimaryKey` finder method. Although it is expected that most plan providers will use the plan identifier as the primary key, not defining the `findByPrimaryKey` in the `PlanHome` inteface allows the plan provider to choose the primary key type if necessary. The plan provider must define the `findByPrimaryKey` method in the subinterface of the `PlanHome` interface.

The PlanHome interface defines no `create` methods. Create methods are not needed because the Benefits application does not create new medical and dental plans in the individual entity beans supplied by the insurance companies. An insurance provider that wants to use the entity bean home interface as the mechanism for creating new insurance plans should define the `create` methods in a subinterface of the PlanHome interface. The PremiumHealthPlanHome home interface of the PremiumHealthPlanEJB entity bean illustrates extending the home interface for the purpose of creating a new insurance plan, as does the WrapperPlanHome home interface (described in the next section). See “PremiumHealthPlanHome Home Interface” on page 323.

7.3.5.5 WrapperPlanEJB’s Remote Interface

The WrapperPlanEJB uses the `com.wombat.plan.Plan` interface as its remote interface. See Code Example 7.18, which lists the `Plan` remote interface.

7.3.5.6 WrapperPlanHome Home Interface

The WrapperPlanHome interface is the home interface of the WrapperPlanEJB entity bean. Code Example 7.23 shows its definition.

```
package com.wombat.benefits;

import javax.ejb.*;
import java.rmi.RemoteException;
import com.wombat.plan.Plan;
```

```
import com.wombat.plan.PlanHome;

public interface WrapperPlanHome extends PlanHome {
    Plan create(Plan planRef, String wrapperPlanId)
        throws RemoteException, CreateException;
    Plan findByPrimaryKey(String pkey)
         throws RemoteException, FinderException;
}
```

Code Example 7.23 WrapperPlanHome Home Interface

The WrapperPlanHome interface extends the PlanHome interface and defines a single `create` method. The BenefitsAdminWeb Web application uses this `create` method to add a new plan to the plans currently configured for the Benefits enrollment application. The `plan` argument is an object reference to an actual medical or dental plan object provided by an insurance company. The `wrapperPlanId` is a primary key for the plan visible to the WrapperPlanEJB clients; the customer (Star Enterprise's Benefits Administrator in our example) chooses this key. See "BenefitsAdminWeb Web application" on page 316 for the description of the BenefitsAdminWeb Web application.

7.3.5.7 WrapperPlanBeanCMP Entity Bean Class

The WrapperPlanBeanCMP is an interesting entity bean class from the perspective of its persistence. Its persistence state consists of two parts. Container-managed fields implement one part of its persistent state (these fields are ultimately bound to columns in a database table, which we illustrate in the WrapperPlanBean class), while delegation to the wrapped actual plan object implements the other part. Our example implementation also illustrates caching of some of the state obtained from the wrapped bean. The delegation and caching can be considered as a form of bean-managed persistence in which the access to the state is made by calls to the wrapped plan objects.

Code Example 7.24 below illustrates the implementation of the WrapperPlanBeanCMP class.

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

```
import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.Collection;
import com.wombat.plan.Plan;
import com.wombat.plan.PlanInfo;
import com.wombat.plan.Doctor;
import com.wombat.plan.PlanException;

public class WrapperPlanBeanCMP implements EntityBean {

    // container-managed fields
    public String wrapperPlanId;
    public Plan plan;

    // cached attributes of the wrapped plan
    public int planType;
    public String planName;

    //
    // Business methods from WrapperPlan interface
    //

    public PlanInfo getPlanInfo() {
        PlanInfo pi = new PlanInfo();
        pi.setPlanId(wrapperPlanId);
        pi.setPlanType(planType);
        pi.setPlanName(planName);
        return pi;
    }

    public int getPlanType() throws PlanException {
        return planType;
    }

    public double getCost(int coverage, int age,
            boolean smokerStatus) throws PlanException {
        try {
            return plan.getCost(coverage, age, smokerStatus);
```

```
    } catch (RemoteException ex) {
        throw new EJBException(ex);
    }
}

public Collection getAllDoctors() throws PlanException {
    try {
        return plan.getAllDoctors();
    } catch (RemoteException ex) {
        throw new EJBException(ex);
    }
}

public Collection getDoctorsByName(Doctor template)
        throws PlanException {
    try {
        return plan.getDoctorsByName(template);
    } catch (RemoteException ex) {
        throw new EJBException(ex);
    }
}

public Collection getDoctorsBySpecialty(String specialty)
        throws PlanException {
    try {
        return plan.getDoctorsBySpecialty(specialty);
    } catch (RemoteException ex) {
        throw new EJBException(ex);
    }
}

//
// Methods from the home interface WrapperPlanHome
//
public String ejbCreate(Plan plan, String wrapperPlanId) {
    // set container-managed fields
    this.wrapperPlanId = wrapperPlanId;
    this.plan = plan;
    return null;
```

```
    }

    public void ejbPostCreate(Plan plan, String wrapperPlanId) {
        updateCachedFields();
    }

    //
    // Methods from javax.ejb.EntityBean interface
    //
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbLoad() {
        updateCachedFields();
    }
    public void ejbStore() {}
    public void ejbPassivate() {}

    private void updateCachedFields() {
        try {
            PlanInfo planInfo = plan.getPlanInfo();
            this.planType = planInfo.getPlanType();
            this.planName = planInfo.getPlanName();
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }
}
```

Code Example 7.24 WrapperPlanBeanCMP Class Implementation

The WrapperPlanBeanCMP class defines two container-managed fields: `wrapperPlanId` and `plan`. The `wrapperPlanId` field is a customer-assigned primary key that identifies a particular plan at a customer enterprise. The `plan` field is an object reference to the actual plan object implemented by an entity bean supplied by the insurance providers. In our example, this is one of the entity beans provided by PremiumHealth or Providence.

The WrapperPlanBeanCMP class also defines two fields that are cached values obtained from the wrapped actual plan. These fields are `planType` and `planName`. We discuss their use later in this section.

In addition to the fields, the WrapperPlanBeanCMP class implements the business methods defined in the PlanHome remote interface, the `ejbCreate` method that corresponds to the `create` method defined in the WrapperPlanHome interface, and the container callbacks defined in the EntityBean interface.

The `getCost`, `getAllDoctors`, `getDoctorsByName`, and `getDoctorsBySpecialty` methods delegate to the same-named method of the actual plan object using the `plan` object reference.

The `getPlanInfo` method returns the plan attributes cached in the instance fields. The `planId` in the PlanInfo object is the `wrapperPlanId`. Returning the key in this manner means that the client of the WrapperPlanEJB bean does not see the primary key of the actual plan object provided by the insurance company.

The `getPlanType` method returns the value of the `planType` field.

It is important to note how and when we set the values of the cached fields. Note how we manage the `planType` and `planName` fields. These fields contain cached values of the wrapped actual plan state. We set the cached fields in the `ejbPostCreate` and `ejbLoad` methods. We use the `ejbPostCreate` method to set the values for the first time. (Alternatively, we could have used the `ejbCreate` method to set the values.) Because other programs may change the objects from which the cached fields are obtained, we use the `ejbLoad` method to refresh the values of the cached fields. The container invokes the `ejbLoad` method before it dispatches the first business method on an instance of the WrapperPlanBeanCMP class in each transaction unless the Container is certain that the values have not changed in the underlying objects. The `ejbLoad` method ensures that the instance has a chance to refresh the values of the cached fields before the container dispatches a business method on the instance. See “Transaction Commit OID” on page 208 in Chapter 6, Understanding Entity Beans, for a discussion of transaction commit options.

7.3.5.8 WrapperPlanBean Entity Bean Class

The WrapperPlanBean class is the bean-managed persistence version of the WrapperPlanBeanCMP entity bean class. The WrapperPlanBean class extends the WrapperPlanBeanCMP class and implements the data access logic using the BenefitsDatabase default schema defined by Wombat (See “BenefitsDatabase” on page 317).

Code Example 7.25 shows the implementation of the WrapperPlanBean class.

```
package com.wombat.benefits;

import javax.ejb.*;

import javax.naming.InitialContext;
import javax.naming.Context;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;
import java.util.Vector;
import java.util.Collection;
import javax.rmi.PortableRemoteObject;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ByteArrayInputStream;

import com.wombat.plan.Plan;
import com.wombat.plan.PlanHome;
import com.wombat.plan.PlanInfo;
import com.wombat.plan.PlanType;
import com.wombat.plan.Doctor;

public class WrapperPlanBean extends WrapperPlanBeanCMP {
    private EntityContext entityContext;
    private DataSource ds;

    public String ejbCreate(Plan plan, String wrapperPlanId) {
        super.ejbCreate(plan, wrapperPlanId);

        // Perform database insert
        try {
            byte[] serHandle = planToSerHandle(plan);
```

```
        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
            “INSERT INTO Wrapper_Plans VALUES (?, ?, ?)”
        );
        pstmt.setString(1, this.wrapperPlanId);
        pstmt.setBytes(2, serHandle);
        pstmt.setInt(3, this.plan.getPlanType());
        pstmt.executeUpdate();
        con.close();
        return wrapperPlanId;
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

public String ejbFindByPrimaryKey(String planID)
    throws ObjectNotFoundException
{
    try {
        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
            “SELECT cfg_wrapper_planid “ +
            “FROM Wrapper_Plans “ +
            “WHERE cfg_wrapper_planid = ?”
        );
        pstmt.setString(1, planID);
        ResultSet rs = pstmt.executeQuery();
        if (rs.next()) {
            con.close();
            return planID;
        } else {
            con.close();
            throw new ObjectNotFoundException();
        }
    } catch (SQLException ex) {
        throw new EJBException(ex);
    }
}
```

```
public String ejbFindByPlanId(String planID)
    throws ObjectNotFoundException
{
     return ejbFindByPrimaryKey(planID);
}

public Collection ejbFindMedicalPlans() {
    return findPlans(PlanType.MEDICAL);
}

public Collection ejbFindDentalPlans() {
    return findPlans(PlanType.DENTAL);
}

public Collection ejbFindByDoctor(Doctor template) {
    try {
        Vector vec = new Vector();
        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
            “SELECT cfg_wrapper_planid, cfg_act_plan_handle “ +
            “FROM Wrapper_Plans “
        );
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            String wrapperPK = rs.getString(1);
            byte[] serHandle = rs.getBytes(2);
            Plan plan = serPlanHandleToRef(serHandle);
            Collection coll =
                plan.getDoctorsByName(template);
            if (coll.size() > 0)
                vec.add(wrapperPK);
        }
        con.close();
        return vec;
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}
```

```
public void setEntityContext(EntityContext ctx) {
    readEnvironment();
    super.setEntityContext(ctx);
}

public void ejbRemove() {
    super.ejbRemove();

    try {

        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
            “DELETE FROM Wrapper_Plans “ +
            “WHERE cfg_wrapper_planid = ?”);
        pstmt.setString(1, wrapperPlanId);
        pstmt.executeUpdate();
        con.close();
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

public void ejbLoad() {
    try {
        byte[] serHandle;

        wrapperPlanId =
            (String)entityContext.getPrimaryKey();

        // load Plan handle from database
        Connection con = getConnection();

        PreparedStatement pstmt = con.prepareStatement(
            “SELECT cfg_act_plan “ +
            “FROM Wrapper_Plans “ +
            “WHERE cfg_wrapper_planid = ?”);
        pstmt.setString(1, wrapperPlanId);
        ResultSet rs = pstmt.executeQuery();
```

```
            if (rs.next()) {
                serHandle = rs.getBytes(1);
                con.close();
            } else {
                throw new NoSuchEntityException();
            }

            // convert handle to Plan reference
            plan = serPlanHandleToRef(serHandle);
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
        super.ejbLoad();
    }

    //
    // Helper methods.
    //

    private Connection getConnection() {
        try {
            return ds.getConnection();
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }

    private void readEnvironment() {
        try {
            Context ictx = new InitialContext();
            ds = (DataSource)
                ictx.lookup("java:comp/env/jdbc/BenefitsDB");
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }

    private static Plan serPlanHandleToRef(byte[] serHandle) {
        try {
```

```
        ObjectInputStream inp = new ObjectInputStream(
            new ByteArrayInputStream(serHandle));
        Handle handle = (Handle)inp.readObject();
        return (Plan)PortableRemoteObject.narrow(
            handle.getEJBObject(), Plan.class);
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

private static byte[] planToSerHandle(Plan plan) {
    try {
        ByteArrayOutputStream barr =
            new ByteArrayOutputStream();
        ObjectOutputStream out =
            new ObjectOutputStream(barr);
        Handle handle = plan.getHandle();
        out.writeObject(handle);
        return barr.toByteArray();
    } catch (Exception ex) {
        throw new EJBException(ex);
    }
}

private Collection findPlans(int planType) {
    try {
        Vector vec = new Vector();
        Connection con = getConnection();
        PreparedStatement pstmt = con.prepareStatement(
            “SELECT cfg_wrapper_planid “ +
            “FROM Wrapper_Plans “ +
            “WHERE cfg_plan_type = ?”
        );
        pstmt.setInt(1, planType);
        ResultSet rs = pstmt.executeQuery();

        while (rs.next()) {
            String pkey = rs.getString(1);
            vec.add(pkey);
```

```
            }
            con.close();
            return vec;
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }
}
```

Code Example 7.25 WrapperPlanBean Class Implementation

The WrapperPlanBean class defines the following methods:

- The `ejbCreate` method that corresponds to the `create` method defined in the WrapperPlanHome interface
- All the finder methods defined in the PlanHome interface
- The `ejbLoad`, `ejbRemove`, and `setEntityContext` container callbacks that override the same-named callbacks in the superclass

The same principles apply to invoking the above methods for the WrapperPlanBean class as for the SelectionBean class. Refer to “Invoking Superclass Methods” on page 293, for more information.

The WrapperPlanBean class is of particular interest because it uses an entity object handle to deal with the persistence of the object references to the actual plan entity objects implemented by the insurance companies. The WrapperPlanBean class uses handles to store entity object references in persistent storage. Its `ejbCreate` and `ejbLoad` methods illustrate the code to persist an entity object handle. The `ejbCreate` method obtains a handle for the `plan` object reference, serializes it, and stores it in the `cfg_act_Plan_handle` database column. The `ejbLoad` method reads the handle from the `cfg_act_Plan_handle` database column and deserializes it to obtain the object reference to the `plan` entity object. The size of a serialized handle depends on how the container implements handles. It is typically on the order of hundreds of bytes.

Note that Wombat does not override the `ejbStore` method in the WrapperPlanBean class. This is because Wombat knows that the business methods of the WrapperPlanBeanCMP class do not modify the container-managed fields. There-

fore, there is no need to update the database when the Container invokes `ejbStore` on the instance.

Using a handle is not the only way to represent an entity object reference in persistent storage. A handle's main advantage is that the program storing the handle does not need to know the JNDI name of the home interface of the entity object. However, a handle's main disadvantage is that the handle is a binary object not understood outside of the Java environment that created the handle. (This means, for example, that the handle is stored as a VARBINARY data type which is opaque to SQL queries.) In addition, the serialized handle may become invalid after a container reconfiguration. Other ways of storing an object reference in persistent storage include:

- Storing the primary key of the entity object (if all the saved object references have the same home interface).
- Storing both the name of the JNDI home interface and the primary key of the entity object (if the saved object references have different home interfaces).

The second approach (storing both the JNDI home interface name and primary key) would be more suitable for our application because the wrapped actual plans have different home interfaces. A real-life application would probably use this approach rather than the handle-based approach to avoid storing opaque binary handles in the relational schema.

Notice that the `Wrapper_Plans` table maintains the plan type indication in the `cfg_plan_type` database column. Keeping the plan type in its own database column enables a more efficient implementation of the `ejbFindMedicalPlans` and `ejbFindDentalPlans` finder methods. Without this database column, the finder methods would have to iterate over all rows in the `Wrapper_Plans` table and invoke the `getPlanType` method on all the actual plan objects to find the set of all the configured medical or dental plans.

7.3.5.9 plan_intf. jar JAR File

A Java ARchive (JAR) file is a standard file format for packaging a collection of compiled Java classes into a unit that can be loaded and executed in a JVM. One use of a JAR file is for packaging libraries.

To allow the insurance companies or application integrators to develop the enterprise beans that implement the Plan and PlanHome interfaces, Wombat creates and publishes the `plan_intf.jar` JAR file. The `plan_intf.jar` contains all

the class files that an entity bean that implements the Plan and PlanHome interfaces needs at compile time and at runtime.

The `plan_intf.jar` file contains the following class files:

```
com.wombat.plan.Plan.class
com.wombat.plan.PlanHome.class
com.wombat.plan.PlanType.class
com.wombat.plan.PlanInfo.class
com.wombat.plan.Doctor.class
com.wombat.plan.PlanException.class
```

7.3.6 EnrollmentWeb Web application

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

7.3.7 BenefitsAdminWeb Web application

The BenefitsAdminWeb Web application is a set of JavaServer Pages used by the customer's benefits administration department to administer its benefits plans.

The BenefitsAdminWeb Web application does the following work:

- It finds all the deployed plan beans from the insurance companies.
- It assumes that all the home interfaces of the deployed plans are located in a specified JNDI context.
- It uses the finder methods in the WrapperPlanHome interface to find all the plans that are currently configured for use by the Benefits enrollment application.
- It displays the information to the plan administrator and lets her modify the set of configured plans.

If the plan administrator modifies the set of plans, the BenefitsAdminWeb Web application uses the WrapperPlanHome interface to add and remove the configured plans. The skeleton code for adding a plan is illustrated below (we show only the parts that are relevant to the use of the WrapperPlanEJB entity bean):

```
...
// Select a plan to add to the set of configured plans. The plan
// is the object reference to the plan to be added.
Plan plan = ...;

// Let the plans administrator to select a primary key that will
// be seen by the clients.
String wrapperPkey = ...;

// Create a wrapper entity object for the plan.
Plan wrapperPlan = WrapperPlanHome.create(wrapperPkey, plan);
...
```

After the `create` method completes, the created wrapper plan entity object becomes available to the Benefits enrollment application.

If a plan administrator wishes to remove a plan from the list of configured plans, she invokes the `remove` method on the WrapperPlanHome interface, passing it the wrapper plan primary key, as follows:

```
...
WrapperPlanHome.remove(wrapperPkey);
...
```

7.3.8 BenefitsDatabase

Wombat defines the default schema for BenefitsDatabase, which stores the persistent state of SelectionEJB and WrapperPlanEJB. BenefitsDatabase contains two tables: the Selections table and the Wrapper_Plans table. The SelectionEJB and WrapperPlanEJB entity beans access these tables.

If a customer chooses to use a different database schema than the default schema, the customer needs to use the entity beans with container-managed persistence rather than the entity beans with bean-managed persistence. That is, the customer uses the SelectionEJBCMP and WrapperPlanEJBCMP entity beans instead of the SelectionEJB and WrapperPlanEJB beans. In addition, the customer must define a custom persistence mechanism that does not use BenefitsDatabase.

The Selections table contains the employee's current benefits selections. Code Example 7.26 shows the SQL definition of this table.

```
CREATE TABLE Selections (
    sel_empl INT,
    sel_coverage INT,
    sel_medical_plan VARCHAR(32),
    sel_dental_plan VARCHAR(32),
    sel_smoker INT,
    PRIMARY KEY (sel_empl)
)
```

Code Example 7.26 Selections Table

The columns of the Selections table contain the following data:

- **`sel_empl`**—The employee number identifying the employee.
- **`sel_coverage`**—The coverage selection. `sel_coverage` accepts three possible values:
  - 0 means “Employee Only”
  - 1 means “Employee and Spouse”
  - 2 means “Employee, Spouse, and Children”
- **`sel_medical_plan`**—The primary key of the medical plan selected by the employee.
- **`sel_dental_plan`**—The primary key of the dental plan selected by the employee.
- **`sel_smoker`**—The indication whether the employee is a smoker or not. The allowed values are ‘Y’ and ‘N’.

The Wrapper_Plans table contains information about the plans configured for the Benefits enrollment application. Code Example 7.27 shows the SQL definition for this table.

```
CREATE TABLE Wrapper_Plans (
    cfg_wrapper_planid VARCHAR(32),
    cfg_act_plan VARBINARY(1024),
    cfg_plan_type INT,
```

```
    PRIMARY KEY (cfg_wrapper_planid)
)
```

Code Example 7.27 Wrapper_Plans Table

The columns of the Wrapper_Plans table contain the following data:

- **`cfg_wrapper_planid`**—Identifies the primary key for the wrapper plan. The customer's benefits department (such as Star Enterprise) assigns the wrapper primary key.
- **`cfg_act_plan`**—Identifies the serialized handle of the entity object representing the actual medical or dental plan implemented by an entity bean provided by an insurance company.
- **`cfg_plan_type`**—Provides a means for the WrapperPlanEJB entity bean to optimize the implementation of the finder methods. There are two possible values for `cfg_plan_type`:
  - 1 indicates the plan is a medical plan
  - 2 indicates the plan is a dental plan

7.3.9 Packaging of Parts

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

7.3.9.1 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 7.7 depicts the contents of the `benefits.ear` file.

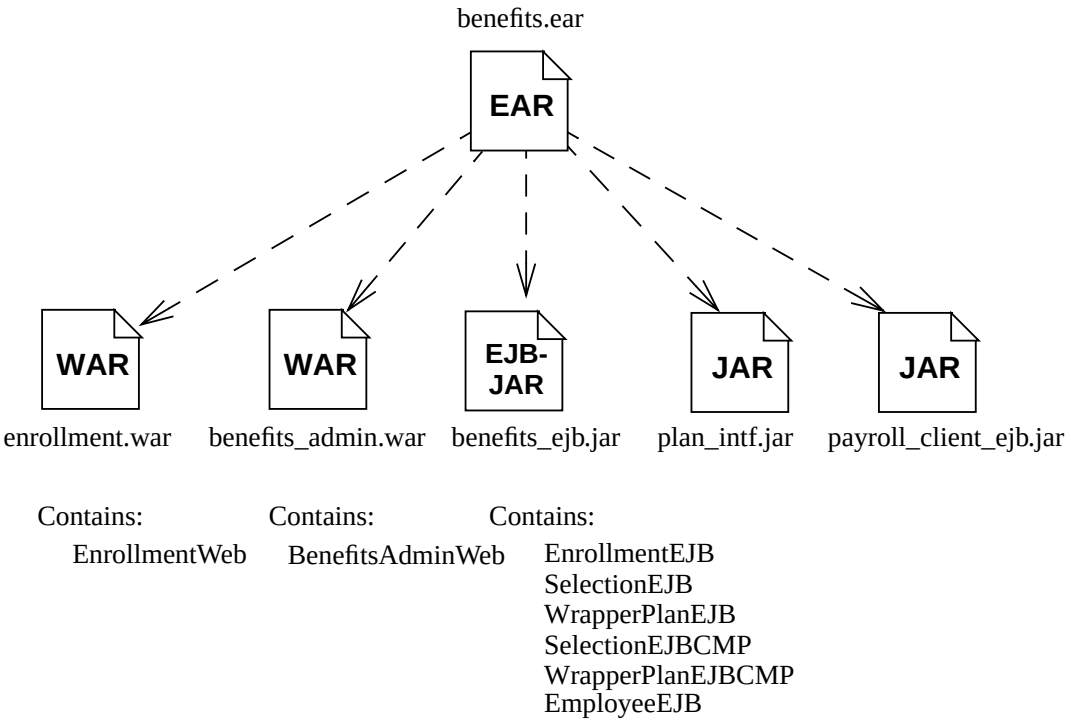

Figure 7.7 Contents of the benefits.ear File

The `benefits.ear` file contains the following parts:

- The `enrollment.war` file with the EnrollmentWeb Web application. (A `war` file is a Web archive file.) The EnrollmentWeb Web application consists of several Java Server Pages.

- The `benefits_admin.war file` with the BenefitsAdminWeb Web application. The BenefitsAdminWeb Web application consists of several JavaServer Pages.

- The `benefits_ejb.jar` file. This is the ejb-jar file that contains the enterprise beans developed by Wombat. See “benefits-ejb.jar File” on page 321.

- The `payroll_client_ejb.jar` file. This is the JAR file with the client-view of the Aardvark PayrollEJB session bean. Wombat needs this file to compile the EnrollmentEJB classes, and the EnrollmentEJB classes will need this JAR file at runtime. The `benefits_ejb.jar` file should note this dependency in the Manifest file of the `benefits.ejb` jar file.
- The `plan_intf.jar` JAR file that contains the Plan and PlanHome interface files with their dependent class and interfaces. The `benefits_ejb.jar` file should note the dependency on this JAR file in the Manifest file of the `benefits.ejb` jar file

7.3.9.2 benefits-ejb.jar File

The `benefits-ejb.jar` file is an ejb-jar file that contains the enterprise beans developed by Wombat. Code Example 7.28 lists the classes that the file contains.

```
com.wombat.benefits.CoverageCategory.class
com.wombat.benefits.Employee.class
com.wombat.benefits.EmployeeBean.class
com.wombat.benefits.EmployeeCopy.class
com.wombat.benefits.EmployeeException.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
com.wombat.benefits.Options.class
com.wombat.benefits.Selection.class
com.wombat.benefits.SelectionBean.class
com.wombat.benefits.SelectionBeanCMP.class
com.wombat.benefits.SelectionCopy.class
com.wombat.benefits.SelectionException.class
com.wombat.benefits.SelectionHome.class
com.wombat.benefits.Summary.class
com.wombat.benefits.WrapperPlanBean.class
com.wombat.benefits.WrapperPlanBeanCMP.class
com.wombat.benefits.WrapperPlanHome.class
```

Code Example 7.28 Contents of the benefits-ejb.jar File

7.4 Parts Developed by Premium Health

Premium Health is one of the insurance companies that provides insurance plans to the employees of Star Enterprise. It develops its own Web application and enterprise beans that are intended for deployment at its customer sites.

7.4.1 Overview

Premium Health develops the parts illustrated in the Figure 7.8.

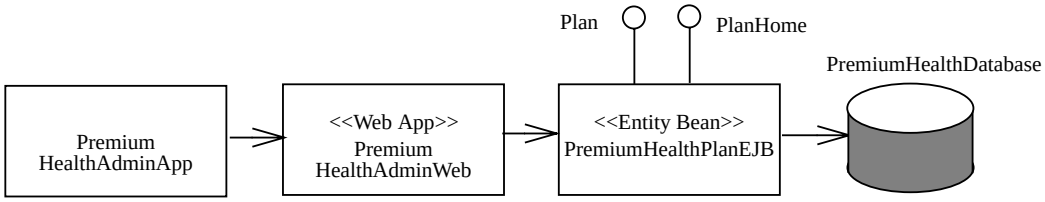

Figure 7.8 Parts Developed by Premium Health

Premium Health expects its customers to deploy the PremiumHealthAdminWeb, PremiumHealthPlanEJB, and PremiumHealthDatabase parts at their own customer sites. A company such as Star Enterprise that contracts with Premium Health deploys the three parts at its own site. However, the PremiumHealthAdminApp is deployed at the Premium Health site, not at the customer site.

PremiumHealthAdminWeb is a Web application deployed at the customer site. Premium Health uses it to remotely administer the plan information stored and used at the customer site.

PremiumHealthPlanEJB is an entity bean that implements the Plan and PlanHome interfaces. Recall that Wombat developed these interfaces to provide a standard way for an insurance company to integrate its coverage plans with Wombat's Benefits application.

PremiumHealthDatabase is a database that stores the plan information used by the PremiumHealthPlan bean.

PremiumHealthAdminApp is an application deployed at the Premium Health site. Premium Health uses the application to upload and modify the plan information at the customer sites.

7.4.2 PremiumHealthPlanEJB EntityBean

The PremiumHealthPlanEJB entity bean implements the business logic for the medical and dental plans provided by Premium Health to its customers. It uses the Plan remote interface and the PremiumHealthPlanHome home interface. This section describes the remote and home interfaces, and the entity bean implementation class.

7.4.2.1 PremiumHealthPlanEJB Remote Interface

PremiumHealthPlanEJB uses the `com.wombat.plan.Plan` interface as its remote interface. See Code Example 7.18.

7.4.2.2 PremiumHealthPlanHome Home Interface

The PremiumHealthPlanHome interface extends the PlanHome interface and adds a single `create` method. Code Example 7.29 shows the definition of this home interface:

```
package com.premiumhealth.plan;

import javax.ejb.*;
import java.rmi.RemoteException;
import com.wombat.plan.Plan;
import com.wombat.plan.PlanHome;

public interface PremiumHealthPlanHome extends PlanHome {
    Plan create(Plan planRef, String planName,
        int planType)throws RemoteException, CreateException;
    Plan findByPrimaryKey(String planId)
        throws RemoteException, FinderException;
}
```

Code Example 7.29 PremiumHealthPlanHome Home Interface

The PremiumHealthAdminWeb Web application invokes the `create` method when adding a new plan to the list of the available plans administered by Premium Health and offered to a given customer enterprise.

7.4.2.3 PremiumHealthPlanBean Entity Bean Class

The PremiumHealthPlanBean entity bean class defines the implementation of the plan business logic. Code Example 7.30 shows the implementation of the PremiumHealthPlanBean class.

```
package com.premiumhealth.plan;

import javax.ejb.*;
import java.util.Vector;
import java.util.Collection;
import com.wombat.plan.Plan;
import com.wombat.plan.PlanHome;
import com.wombat.plan.PlanInfo;
import com.wombat.plan.PlanType;
import com.wombat.plan.PlanException;
import com.wombat.plan.Doctor;
import java.sql.SQLException;
import javax.sql.DataSource;
import javax.naming.InitialContext;
import javax.naming.Context;

import com.premiumhealth.plan.db.DBDeletePremiumByPlanId;
import com.premiumhealth.plan.db.DBQueryPremium;

import com.premiumhealth.plan.db.DBDeleteDoctorByPlanId;
import com.premiumhealth.plan.db.DBQueryAllDoctors;
import com.premiumhealth.plan.db.DBQueryDoctorByName;
import com.premiumhealth.plan.db.DBQueryDoctorBySpecialty;

import com.premiumhealth.plan.db.DBInsertPlan;
import com.premiumhealth.plan.db.DBDeletePlan;
import com.premiumhealth.plan.db.DBUpdatePlan;
import com.premiumhealth.plan.db.DBQueryPlan;
import com.premiumhealth.plan.db.DBQueryPlanByDoctor;
import com.premiumhealth.plan.db.DBQueryPlanByType;

public class PremiumHealthPlanBean implements EntityBean {
```

```
// cached persistent state
String planId;// primary key
int planType;
String planName;

// other instance variables
DataSource ds;
EntityContext ctx;

// Business methods from the remote interface
//  com.wombat.plan.Plan

public PlanInfo getPlanInfo() {
    PlanInfo pi = new PlanInfo();
    pi.setPlanId(planId);
    pi.setPlanType(planType);
    pi.setPlanName(planName);
    return pi;
}

public int getPlanType() { return planType; }

public double getCost(int coverage, int age,
       boolean smokerStatus) throws PlanException
{
   DBQueryPremium cmd = null;
   try {
       cmd = new DBQueryPremium(ds);
       cmd.setPlanId(planId);
       cmd.setAge(age);
       cmd.setSmokerStatus(smokerStatus ? “Y” : “N”);
       cmd.setCoverage(coverage);
       cmd.execute();
       if (!cmd.next()) {
           throw new PlanException(
               “premium information unavailable”);
       }
       double amount = cmd.getAmount();
```

```
            return amount;
        } catch (SQLException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    public Collection getAllDoctors()
    {
        DBQueryAllDoctors cmd = null;
        try {
            cmd = new DBQueryAllDoctors(ds);
            cmd.setPlanId(planId);
            cmd.execute();

            Vector vec = new Vector();
            while (cmd.next()) {
                Doctor doc = new Doctor();
                doc.setFirstName(cmd.getFirstName());
                doc.setLastName(cmd.getLastName());
                doc.setSpecialty(cmd.getSpecialty());
                doc.setHospital(cmd.getHospital());
                doc.setPracticeSince(cmd.getPracticeSince());
                vec.add(doc);
            }
            return vec;
        } catch (SQLException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    public Collection getDoctorsByName(Doctor template)
    {
        DBQueryDoctorByName cmd = null;
        try {
            cmd = new DBQueryDoctorByName(ds);
```

```
            cmd.setPlanId(planId);
            cmd.setFirstName(template.getFirstName() + "%");
            cmd.setLastName(template.getLastName() + "%");
            cmd.execute();

            Vector vec = new Vector();
            while (cmd.next()) {
                Doctor doc = new Doctor();
                doc.setFirstName(cmd.getFirstName());
                doc.setLastName(cmd.getLastName());
                doc.setSpecialty(cmd.getSpecialty());
                doc.setHospital(cmd.getHospital());
                doc.setPracticeSince(cmd.getPracticeSince());
                vec.add(doc);
            }
            return vec;
        } catch (SQLException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    public Collection getDoctorsBySpecialty(String specialty)
    {
        DBQueryDoctorBySpecialty cmd = null;
        try {
            cmd = new DBQueryDoctorBySpecialty(ds);
            cmd.setPlanId(planId);
            cmd.setSpecialty(specialty);
            cmd.execute();

            Vector vec = new Vector();
            while (cmd.next()) {
                Doctor doc = new Doctor();
                doc.setFirstName(cmd.getFirstName());
                doc.setLastName(cmd.getLastName());
                doc.setSpecialty(cmd.getSpecialty());
                doc.setHospital(cmd.getHospital());
```

```
                doc.setPracticeSince(cmd.getPracticeSince());
                vec.add(doc);
            }
            return vec;
        } catch (SQLException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    // Methods from the home interface
    //  com.wombat.plan.PremiumHealthPlanHome

    public String ejbCreate(String planId, String planName,
            int planType)
    {
        this.planId = planId;
        this.planName = planName;
        this.planType = planType;

        DBInsertPlan cmd = null;
        try {
            cmd = new DBInsertPlan(ds);
            cmd.setPlanId(planId);
            cmd.setPlanType(planType);
            cmd.setPlanDescr(planName);
            cmd.execute();
            return planId;
        } catch (SQLException ex) {
            throw new EJBException(ex);
        } finally {
             if (cmd != null) cmd.release();
        }
    }

    public void ejbPostCreate(String planId, String planName,
            int planType) {}
```

```
// Finder methods

public String ejbFindByPrimaryKey(String planId)
    throws FinderException
{
    DBQueryPlan cmd = null;
    try {
        cmd = new DBQueryPlan(ds);
        cmd.setPlanId(planId);
        cmd.execute();

        Vector vec = new Vector();
        if (cmd.next()) {
            return planId;
        } else {
            throw new ObjectNotFoundException();
        }
    } catch (SQLException ex) {
        throw new EJBException(ex);
    } finally {
        if (cmd != null) cmd.release();
    }
}

public String ejbFindByPlanId(String planId)
    throws FinderException
{
    return ejbFindByPrimaryKey(planId);
}

public Collection ejbFindMedicalPlans()
{
    DBQueryPlanByType cmd = null;
    try {
        cmd = new DBQueryPlanByType(ds);
        cmd.setPlanType(PlanType.MEDICAL);
        cmd.execute();

        Vector vec = new Vector();
```

```
            while (cmd.next()) {
                vec.add(cmd.getPlanId());
            }
            return vec;
        } catch (SQLException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    public Collection ejbFindDentalPlans()
    {
        DBQueryPlanByType cmd = null;
        try {
            cmd = new DBQueryPlanByType(ds);
            cmd.setPlanType(PlanType.DENTAL);
            cmd.execute();

            Vector vec = new Vector();
            while (cmd.next()) {
                vec.add(cmd.getPlanId());
            }
            return vec;
        } catch (SQLException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    public Collection ejbFindByDoctor(Doctor template)
    {
        DBQueryPlanByDoctor cmd = null;
        try {
            cmd = new DBQueryPlanByDoctor(ds);
            cmd.setFirstName(template.getFirstName() + "%");
            cmd.setLastName(template.getLastName() + "%");
            cmd.execute();
```

```
        Vector vec = new Vector();
        while (cmd.next()) {
            vec.add(cmd.getPlanId());
        }
        return vec;
    } catch (SQLException ex) {
        throw new EJBException(ex);
    } finally {
        if (cmd != null) cmd.release();
    }
}

//
// Methods from javax.ejb.EntityBean interface
//

public void setEntityContext(EntityContext ctx) {
    this.ctx = ctx;
    readEnvironment();
}

public void unsetEntityContext() {}

public void ejbRemove()
{
    DBDeletePlan cmd = null;
    DBDeletePremiumByPlanId cmd2 = null;
    DBDeleteDoctorByPlanId cmd3 = null;

    try {
        // Delete dependent premium information.
        cmd2 = new DBDeletePremiumByPlanId(ds);
        cmd2.setPlanId(planId);
        cmd2.execute();

        // Delete dependent doctor information.
        cmd3 = new DBDeleteDoctorByPlanId(ds);
        cmd3.setPlanId(planId);
```

```
            cmd3.execute();

            // Delete plan record.
            cmd = new DBDeletePlan(ds);
            cmd.setPlanId(planId);
            cmd.execute();
        } catch (SQLException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
            if (cmd2 != null) cmd2.release();
            if (cmd3 != null) cmd3.release();
        }
    }

    public void ejbActivate() {
        planId = (String)ctx.getPrimaryKey();
    }

    public void ejbPassivate() { planId = null; }

    public void ejbLoad()
    {
        DBQueryPlan cmd = null;
        try {
            cmd = new DBQueryPlan(ds);
            cmd.setPlanId(planId);
            cmd.execute();
            if (!cmd.next()) {
                throw new NoSuchEntityException();
            }
            planName = cmd.getPlanDescr();
            planType = cmd.getPlanType();
        } catch (SQLException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
         }
    }
```

```
    public void ejbStore()
    {
        DBUpdatePlan cmd = null;
        try {
            cmd = new DBUpdatePlan(ds);
            cmd.setPlanId(planId);
            cmd.setPlanType(planType);
            cmd.setPlanDescr(planName);
            cmd.execute();
         } catch (SQLException ex) {
             throw new EJBException(ex);
         } finally {
            if (cmd != null) cmd.release();
        }
    }

    private void readEnvironment() {
        try {
            Context ictx = new InitialContext();
            ds = (DataSource)ictx.lookup(
                "java:comp/env/jdbc/PremiumHealthDB");
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }
}
```

Code Example 7.30 PremiumHealthPlanBean Entity Bean Class

The PremiumHealthPlanBean class is a typical entity bean class with bean-managed persistence. As such, it manages its own database access by making use of command beans. See “Data Access Command Beans” on page 112 for an explanation of command beans.

The entity object caches some parts of the persistent state (`planType` and `planName`) to reduce the number of database accesses. The entity object does not cache other parts of the PremiumHealthPlanEJB state. For example, it does not

cache the Doctor information. Instead, the entity object's business methods access the database directly when Doctor information is needed.

Refer to "Data Access Command Beans" on page 446 to see the implementations of the different data access command beans used by the entity example.

7.4.3 HelperEJB Session Bean

The HelperEJB is a stateless session bean which the PremiumHealthAdmin Web application uses to create and maintain the benefits information in the database. We do not show its implementation in the book because it is not relevant to the rest of the application.

7.4.4 PremiumHealthAdminWeb Web Application

The PremiumHealthAdminWeb Web application is a set of servlets which the Premium Health insurance company uses to maintain plan information at its customer sites. The PremiumHealthAdminWeb Web application is deployed at the customer site. Premium Health sends XML requests over the Internet to make online updates to the plan information at the customer site.

The PremiumHealthAdminWeb Web application uses the PremiumHealthPlanEJB and HelperEJB to perform the updates to the plan information.

We briefly mention the Web application only because it is one part of the entire example. The Web application consists of servlets, and these servlets are important to EJB. We do not discuss the Web application and servlets in more detail because the focus of this book is Enterprise JavaBeans.

7.4.5 PremiumHealthAdminApp

PremiumHealthAdminApp is an application deployed at Premium Health. PremiumHealthAdminApp acts as the client of the PremiumHealthAdminWeb Web application. It is used to upload and maintain the plan information at the customer sites.

We do not describe the architecture and implementation of PremiumHealthAdminApp in this book.

7.4.6 PremiumHealthDatabase

PremiumHealthDatabase is a relational database that exists at the customer site. It contains the plan information needed by the PremiumHealthPlanEJB entity bean.

Using the PremiumHealthAdminWeb Web application, the Premium Health insurance company updates the plan information in the database via the Internet.

The PremiumHealthDatabase schema defines three tables. The Plans table contains the plans offered by Premium Health. The Premium table contains the information about the plan premium. The Doctor table contains the information about the participating doctors.

Code Example 7.31 shows the definition of the Plans table.

```
CREATE TABLE Plans (
    plan_id VARCHAR(32),
    plan_descr VARCHAR(32),
    plan_type INT,
    PRIMARY KEY (plan_id)
)
```

Code Example 7.31 Plans Table

The columns of the Plans table contain the following data:

- **`plan_id` column**—Contains the unique identifier for the plan.
- **`plan_descr` column**—Holds a short description of the plan suitable for display within a GUI form.
- **`plan_type` column**—Indicates whether the plan is a medical or dental plan. When equal to 1, the plan is a medical plan; when equal to 2, the plan is a dental plan.

Code Example 7.32 shows the SQL statements defining the Premium table:

```
CREATE TABLE Premium (
    prem_plan_id VARCHAR(32),
    prem_age INT,
    prem_smoker_status CHAR(1),
    prem_coverage INT,
    prem_amount DECIMAL(6.2),
    PRIMARY KEY (prem_plan_id, prem_age, prem_smoker_status,
```

```
        prem_coverage)
)
```

Code Example 7.32 Premium Table

The columns of the Premium table contain information that the application uses to calculate an employee's premium deduction. An employee's premium amount is determined by the selected plan, his or her age, smoker status, and extent of coverage. The columns of the Premium table hold the following data:

- **`prem_plan_id` column**—Holds the unique identifier of the plan.
- **`prem_age` column**—Indicates the age of the employee.
- **`prem_smoker_status` column**—Indicates whether the employee is a smoker or not. The allowed values are 'Y' and 'N'.
- **`prem_coverage` column**—Indicates the coverage category selected by the employee.
- **`prem_amount` column**—Holds the amount of the plan premium.

Code Example 7.33 shows the SQL definition of the Doctors table.

```
CREATE TABLE Doctors (
    doc_id INT,
    doc_plan_id VARCHAR(32),
    doc_first_name VARCHAR(32),
    doc_last_name VARCHAR(32),
    doc_specialty VARCHAR(32),
    doc_hospital VARCHAR(32),
    doc_practice_since INT,
    PRIMARY KEY (doc_id, doc_plan_id)
)
```

Code Example 7.33 Doctors Table

The columns of the Doctors table hold information about the different doctors affiliated with a particular plan. These columns contain the following data:

- **`doc_id` column**—An identifier for a particular doctor
- **`doc_plan_id` column**—The identifier for a particular plan
- **`doc_first_name` column**—The doctor's first name
- **`doc_last_name` column**—The doctor's last name
- **`doc_specialty` column**—A short description of the doctor's field of specialty
- **`doc_hospital` column**—The name of the hospital to which the doctor is affiliated
- **`doc_practice_since` column**—The year when the doctor first began his practice

The `doc_id` and `doc_plan_id` columns together uniquely identify a row in the Doctors table, and this is noted as the primary key. Note that the database is not in normalized form. Because of this, it would be possible (though not desirable) to have the same doctor listed in different plans with inconsistent information.

7.4.7 Packaging

This section describes the packaging of the parts developed by Premium Health.

7.4.7.1 premiumhealth.ear Archive File

Premium Health packages the parts that support the benefits application as a single J2EE Enterprise Application Archive file which it names `premiumhealth.ear`. Figure 7.9 depicts the contents of the `premiumhealth.ear` file.

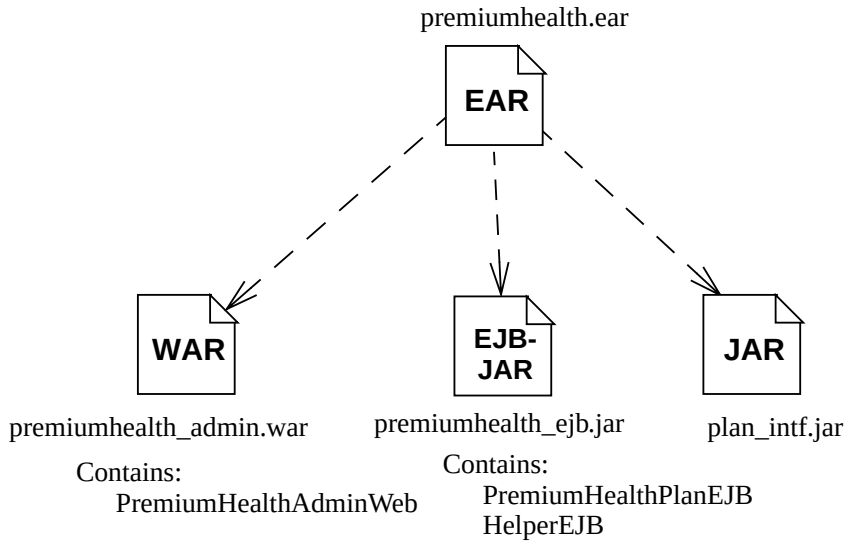

Figure 7.9 Premium Health's Enterprise Application Archive File

The `premiumhealth.ear` file contains the following parts:

- **`premiumhealth_admin.war` file**—This Web Application Archive file contains the PremiumHealthAdminWeb Web application. The Web application consists of several servlets. A `war` file is a Web archive resource file that contains the class files for servlets and JSPs.
- **`premiumhealth_ejb.jar` file**—This ejb-jar file contains the enterprise beans (PremiumHealthPlanEJB and the HelperEJB) developed by Premium Health.
- **`plan_intf.jar` file**—This is the JAR file published by Wombat. Because the PremiumHealthPlanEJB depends on the contents of this JAR file at runtime, it is included in the `premiumhealth.ear` file. The `Class-Path` attribute in the Manifest of the `premiumhealth_ejb.jar` file lists this dependency on the `plan_intf.jar` file.

7.4.7.2 premiumhealth_ejb.jar File

The `premiumhealth_ejb.jar` file is an ejb-jar file that contains the enterprise beans developed by Premium Health. Code Example 7.34 shows the Java class files contained by the file.

```
com.premiumhealth.plan.PremiumHealthPlan.class
com.premiumhealth.plan.PremiumHealthPlanBean.class
com.premiumhealth.plan.PremiumHealthPlanException.class
com.premiumhealth.plan.PremiumHealthPlanHome.class

com.premiumhealth.plan.db.DBDeletePlan.class
com.premiumhealth.plan.db.DBDeletePremiumByPlanId.class
com.premiumhealth.plan.db.DBDeleteDoctorByPlanId.class
com.premiumhealth.plan.db.DBInsertPlan.class
com.premiumhealth.plan.db.DBQueryAllDoctors.class
com.premiumhealth.plan.db.DBQueryBean.class
com.premiumhealth.plan.db.DBQueryDoctorsByName.class
com.premiumhealth.plan.db.DBQueryDoctorsBySpecialty.class
com.premiumhealth.plan.db.DBQueryPlan.class
com.premiumhealth.plan.db.DBQueryPlanByDoctor.class
com.premiumhealth.plan.db.DBQueryPlanByType.class
com.premiumhealth.plan.db.DBQueryPremium.class
com.premiumhealth.plan.db.DBUpdateBean.class
com.premiumhealth.plan.db.DBUpdatePlan.class

com.premiumhealth.helper.Helper.class
com.premiumhealth.helper.HelperBean.class
com.premiumhealth.helper.HelperException.class
com.premiumhealth.helper.HelperHome.class
```

Code Example 7.34 premiumhealth_ejb.jar File

7.5 Parts Developed by Providence

Providence is another insurance provider that interoperates with Wombat's benefits application. Providence, however, differs radically from Premium Health in its approach to exchanging information with its customers. While Premium Health maintains a database with the plan information at each customer site, Providence

maintains the plan data at its own site. An application that needs to access the Providence plan data must access the Providence Web site to obtain the plan data.

7.5.1 Overview

Figure 7.10 illustrates the main parts developed by Providence:

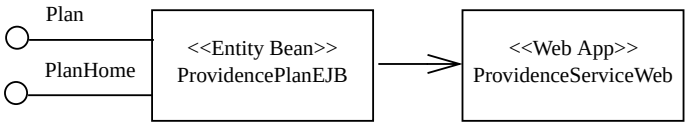

Figure 7.10 Providence's Benefits Plan Parts

ProvidencePlanEJB is an entity bean that enables Providence's plan to be integrated with the Wombat Benefits application. Providence accomplishes this by defining ProvidencePlanEJB's remote and home interfaces to be compatible with the Plan and PlanHome interfaces defined by Wombat. ProvidencePlanEJB uses HTTP and XML to communicate with ProvidenceServiceWeb to obtain the Providence plan information.

ProvidenceServiceWeb is a Web application deployed at the Providence site. Customers access it remotely from their own sites to obtain the up-to-date plan information.

7.5.2 ProvidencePlanEJB EntityBean

The ProvidencePlanEJB entity bean uses the Plan remote interface and the ProvidencePlanHome home interface. This section describes these two interfaces as well as the entity bean implementation class.

7.5.2.1 ProvidencePlanEJB Remote Interface

ProvidencePlanEJB uses the `com.wombat.plan.Plan` interface as its remote interface. See Code Example 7.18.

7.5.2.2 ProvidencePlanHome Home Interface

The ProvidencePlanHome interface is the home interface for ProvidencePlanEJB. This interface extends the Wombat PlanHome interface. It neither defines nor overrides any methods. See Code Example 7.35.

```
package com.providence.plan;

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

public interface ProvidencePlanHome
        extends com.wombat.plan.PlanHome {
    Plan ejbFindByPrimaryKey(String planId)
        throws RemoteException, FinderException;
}
```

Code Example 7.35 ProvidencePlanHome Home Interface

7.5.2.3 ProvidencePlanBean Entity Bean Class

The ProvidencePlanBean class demonstrates the implementation of an entity bean whose persistent state is not stored directly in a local database. Instead, the state is obtained by making requests to a remote application. In our case, the remote application is the ProvidenceServiceWeb Web application at the Providence site. The communication is over HTTP in the XML format. Code Example 7.36 shows the implementation of the ProvidencePlanBean entity bean class.

```
package com.providence.plan;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.Vector;
import java.util.Collection;
import java.util.Iterator;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import com.wombat.plan.Plan;
import com.wombat.plan.PlanHome;
import com.wombat.plan.PlanInfo;
import com.wombat.plan.PlanType;
import com.wombat.plan.PlanException;
import com.wombat.plan.Doctor;

import com.providence.plan.cb.GetCostBean;
import com.providence.plan.cb.GetPlansBean;
import com.providence.plan.cb.GetPlanInfoBean;
import com.providence.plan.cb.GetDoctorsBean;
import com.providence.plan.cb.HttpCommandBeanException;

public class ProvidencePlanBean implements EntityBean {
    EntityContext ctx;
    String baseURL;

    String planId;
    String planName;
    int planType;

    // Business methods from the remote interface
    //  com.wombat.plan.Plan

    public PlanInfo getPlanInfo() {
        PlanInfo pi = new PlanInfo();
        pi.setPlanId(planId);
        pi.setPlanName(planName);
        pi.setPlanType(planType);
        return pi;
    }

    public int getPlanType() { return planType; }

    public double getCost(int coverage, int age,
           boolean smokerStatus)
    {
        GetCostBean cmd = null;
        try {
```

```
        cmd = new GetCostBean();
        cmd.setBaseURL(baseURL);
        cmd.setPlanId(planId);
        cmd.setCoverage(coverage);
        cmd.setAge(age);
        cmd.setSmoker(smokerStatus);
        cmd.execute();
        return cmd.getCost();
    } catch (HttpCommandBeanException ex) {
        throw new EJBException(ex);
    } finally {
        if (cmd != null) cmd.release();
    }
}

public Collection getAllDoctors()
{
    GetDoctorsBean cmd = null;
    try {
        cmd = new GetDoctorsBean();
        cmd.setBaseURL(baseURL);
        cmd.setPlanId(planId);
        cmd.execute();

        Vector vec = new Vector();
        while (cmd.next()) {
            Doctor doc = new Doctor();
            doc.setLastName(cmd.getLastName());
            doc.setFirstName(cmd.getFirstName());
            doc.setSpecialty(cmd.getSpecialty());
            doc.setPracticeSince(cmd.getPracticeSince());
            doc.setHospital(cmd.getHospital());
            vec.add(doc);
        }
        return vec;
    } catch (HttpCommandBeanException ex) {
        throw new EJBException(ex);
    } finally {
        if (cmd != null)
```

```
            cmd.release();
        }
    }

    public Collection getDoctorsByName(Doctor template) {
        GetDoctorsBean cmd = null;
        try {
            cmd = new GetDoctorsBean();
            cmd.setBaseURL(baseURL);
            cmd.setPlanId(planId);
            cmd.setLastName(template.getLastName());
            cmd.setFirstName(template.getFirstName());

            cmd.execute();
            Vector vec = new Vector();
            while (cmd.next()) {
                Doctor doc = new Doctor();
                doc.setLastName(cmd.getLastName());
                doc.setFirstName(cmd.getFirstName());
                doc.setSpecialty(cmd.getSpecialty());
                doc.setPracticeSince(cmd.getPracticeSince());
                doc.setHospital(cmd.getHospital());
                vec.add(doc);
            }
            return vec;
        } catch (HttpCommandBeanException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null)
                cmd.release();
        }
    }

    public Collection getDoctorsBySpecialty(String specialty) {
        GetDoctorsBean cmd = null;
        try {
            cmd = new GetDoctorsBean();
            cmd.setBaseURL(baseURL);
            cmd.setPlanId(planId);
```

```
            cmd.setSpecialty(specialty);
            cmd.execute();

            Vector vec = new Vector();
            while (cmd.next()) {
                Doctor doc = new Doctor();
                doc.setLastName(cmd.getLastName());
                doc.setFirstName(cmd.getFirstName());
                doc.setSpecialty(cmd.getSpecialty());
                doc.setPracticeSince(cmd.getPracticeSince());
                doc.setHospital(cmd.getHospital());
                vec.add(doc);
            }
            return vec;
        } catch (HttpCommandBeanException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    // Finder methods

    public String ejbFindByPrimaryKey(String planId)
        throws ObjectNotFoundException
    {
        GetPlansBean cmd = null;
        try {
            cmd = new GetPlansBean();
            cmd.setBaseURL(baseURL);
            cmd.setPlanId(planId);
            cmd.execute();

            Vector vec = new Vector();
            if (cmd.next()) {
                return planId;
            } else {
                throw new ObjectNotFoundException();
            }
```

```
        } catch (HttpCommandBeanException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    public String ejbFindByPlanId(String planId)
        throws ObjectNotFoundException
    {
        return ejbFindByPrimaryKey(planId);
    }

    public Collection ejbFindMedicalPlans()
    {
        GetPlansBean cmd = null;
        try {
            cmd = new GetPlansBean();
            cmd.setBaseURL(baseURL);
            cmd.setPlanType("medical");
            cmd.execute();

            Vector vec = new Vector();
            while (cmd.next()) {
                vec.add(cmd.getPlanId());
            }
            return vec;
        } catch (HttpCommandBeanException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    public Collection ejbFindDentalPlans()
    {
        GetPlansBean cmd = null;
        try {
            cmd = new GetPlansBean();
```

```
            cmd.setBaseURL(baseURL);
            cmd.setPlanType("dental");
            cmd.execute();

            Vector vec = new Vector();
            while (cmd.next()) {
                vec.add(cmd.getPlanId());
            }
            return vec;
        } catch (HttpCommandBeanException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    public Collection ejbFindByDoctor(Doctor template)
    {
        GetPlansBean cmd = null;
        try {
            cmd = new GetPlansBean();
            cmd.setBaseURL(baseURL);
            cmd.setLastName(template.getLastName());
            cmd.setFirstName(template.getFirstName());
            cmd.execute();

            Vector vec = new Vector();
            while (cmd.next()) {
                vec.add(cmd.getPlanId());
            }
            return vec;
        } catch (HttpCommandBeanException ex) {
            throw new EJBException(ex);
        } finally {
            if (cmd != null) cmd.release();
        }
    }

    //
```

```
// Methods from javax.ejb.EntityBean interface
//

public void setEntityContext(EntityContext ctx) {
    this.ctx = ctx;
    readEnvironment();
}

public void unsetEntityContext() {}

public void ejbRemove() throws RemoveException {
    throw new RemoveException(
        "application is not allowed to remove plan");
}

public void ejbActivate() {
    planId = (String)ctx.getPrimaryKey();
}

public void ejbPassivate() { planId = null; }

public void ejbLoad()
{
    GetPlanInfoBean cmd = null;
    try {
        cmd = new GetPlanInfoBean();
        cmd.setBaseURL(baseURL);
        cmd.setPlanId(planId);
        cmd.execute();
        planName = cmd.getPlanName();
        planType = cmd.getPlanType();
    } catch (HttpCommandBeanException ex) {
        throw new EJBException(ex);
    } finally {
        if (cmd != null) cmd.release();
    }
}

public void ejbStore() {}
```

```
    private void readEnvironment() {
        try {
            Context ictx = new InitialContext();
            baseURL = (String)ictx.lookup(
                "java:comp/env/ProvidenceServiceWebURL");
        } catch (Exception ex) {
            throw new EJBException(ex);
        }
    }
}
```

Code Example 7.36 ProvidencePlanBean Class

The ProvidencePlanBean class depends on several command beans to perform the invocation of the HTTP requests to the ProvidenceServiceWeb.

The `ejbLoad` method loads the basic information about a plan and caches it in the `planName` and `planType` instance fields. Although the ProvidencePlanBean methods are assigned the NotSupported transaction attribute, we use the `ejbLoad` method to cache the plan information. Though in general it is not possible to use the `ejbLoad` and `ejbStore` methods to cache information for entity beans with non-transactional methods (see "Using the ejbLoad and ejbStore Methods" on page 217), it is possible to use the `ejbLoad` method to cache information that is immutable, such as the plan name and plan type.

A business method obtains other information, such as Doctor lists, directly from ProvidenceServiceWeb when needed. The ProvidencePlanBean class also implements the finder methods defined in the PlanHome interface.

Note that the ProvidencePlanBean class obtains the URL of the ProvidenceServiceWeb from an environment entry of the ProvidencePlanEJB entity bean.

7.5.2.4 HTTP Command Beans

Code Example 7.38 through Code Example 7.42 illustrate the implementation of the command beans that make the HTTP calls to the ProvidenceServiceWeb.

The command beans use the APIs defined in the `org.w3.dom` and `org.xml.sax` packages to process XML. They use the `java.net package` to make HTTP calls to

the ProvidenceServiceWeb. This book does not describe the use of these APIs. “XML Message Formats” on page 358 describes the format of the message.

```
package com.providence.plan.cb;

import java.io.InputStream;
import java.net.URL;
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;
import org.xml.sax.Parser;
import org.xml.sax.InputSource;

import com.sun.xml.tree.XmlDocumentBuilder;

public class HttpCommandBean {
    private String baseURL;
    private String params = ““;
    private int paramCount = 0;
    private Element elem;

    public void setBaseURL(String url) {
        this.baseURL = url;
    }

    public void release() {
    }

    protected void addParam(String name, String value) {
        if (paramCount == 0)
            params = “?” + name + “=” + value;
        else
            params = params + “&” + name + “=” + value;
        paramCount++;
    }

    protected void addParam(String name, int value) {
        addParam(name, String.valueOf(value));
    }
```

```
    protected void addParam(String name, double value) {
        addParam(name, String.valueOf(value));
    }

    protected void callServer() throws HttpCommandBeanException
    {
        try {
            URL u = new URL(baseURL + params);
            Object xmlReply = u.getContent();

            XmlDocumentBuilder builder =
                new XmlDocumentBuilder();
            Parser parser = new com.sun.xml.parser.Parser();

            parser.setDocumentHandler(builder);
            builder.setParser(parser);
            builder.setDisableNamespaces(true);
            parser.parse(new InputSource((InputStream)xmlReply));
            elem = builder.getDocument().getDocumentElement();
        } catch (Exception ex) {
            throw new HttpCommandBeanException(ex);
        }
    }

    protected static String getElementValue(Element elem,
            String tagName)
    {
        NodeList li = elem.getElementsByTagName(tagName);
        if (li.getLength() == 1) {
            return li.item(0).getFirstChild().getNodeValue();
        } else {
            return null;
        }
    }

    protected Element getTopElement() {
        return elem;
    }
}
```

Code Example 7.37 HttpCommandBean Command Bean

```
package com.providence.plan.cb;

import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class GetCostBean extends HttpCommandBean {
    public GetCostBean() {
        super();
        addParam(“method”, “calc-cost”);
    }

    public void setCoverage(int coverage) {
        addParam(“coverage”, coverage);
    }
    public void setPlanId(String planId) {
        addParam(“plan-id”, planId);
    }
    public void setAge(int age) {
        addParam(“age”, age);
    }
    public void setSmoker(boolean smoker) {
        addParam(“smoker”, smoker ? “Y” : “N”);
    }

    public void execute() throws HttpCommandBeanException {
        callServer();
    }

    public double getCost() {
        String s = getElementValue(getTopElement(), “cost”);
        return Double.parseDouble(s);
    }
}
```

Code Example 7.38 GetCostBean Command Bean

```
package com.providence.plan.cb;

import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class GetDoctorsBean extends HttpCommandBean {
    private NodeList list;
    private int listLength;
    private int listIndex = -1;
    private Element elem;

    public GetDoctorsBean() {
        super();
        addParam(“method”, “get-doctors”);
    }

    public void setPlanId(String planId) {
        addParam(“plan-id”, planId);
    }
    public void setLastName(String lastName) {
        addParam(“last-name”, lastName);
    }
    public void setFirstName(String firstName) {
        addParam(“first-name”, firstName);
    }
    public void setSpecialty(String specialty) {
        addParam(“specialty”, specialty);
    }

    public void execute() throws HttpCommandBeanException {
        callServer();
        list = getTopElement().getElementsByTagName(“doctor”);
        listLength = list.getLength();
    }

    public String getLastName() {
        return getElementValue(elem, “last-name”);
    }
    public String getFirstName() {
```

```
            return getElementValue(elem, “first-name”);
        }
        public String getSpecialty() {
            return getElementValue(elem, “specialty”);
        }
        public int getPracticeSince() {
            return Integer.parseInt(
                getElementValue(elem, “practice-since”));
        }
        public String getHospital() {
            return getElementValue(elem, “hospital”);
        }

        public boolean next() {
            if (++listIndex < listLength) {
                elem = (Element)list.item(listIndex);
                return true;
            } else {
                return false;
            }
        }
    }
```

Code Example 7.39 GetDoctorsBean Command Bean

```
package com.providence.plan.cb;

import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

import com.wombat.plan.PlanType;

public class GetPlanInfoBean extends HttpCommandBean {
    public GetPlanInfoBean() {
        super();
        addParam(“method”, “get-plan-info”);
    }
```

```
    public void setPlanId(String planId) {
        addParam("plan-id", planId);
    }

    public void execute() throws HttpCommandBeanException {
        callServer();
    }

    public String getPlanId() {
        return getElementValue(getTopElement(), "plan-id");
    }
    public String getPlanName() {
        return getElementValue(getTopElement(), "plan-name");
    }
    public int getPlanType()
        throws HttpCommandBeanException
    {
        return convertPlanType(
            getElementValue(getTopElement(), "plan-type"));
    }

    static private int convertPlanType(String s)
        throws HttpCommandBeanException
    {
        if (s.equals("Medical"))
            return PlanType.MEDICAL;
        else if (s.equals("Dental"))
            return PlanType.DENTAL;
        else
            throw new HttpCommandBeanException(
                "Bad plan type from server");
    }
}
```

Code Example 7.40 GetPlanInfoBean Command Bean

```
package com.providence.plan.cb;
```

```
import org.w3c.dom.Element;
import org.w3c.dom.NodeList;

public class GetPlansBean extends HttpCommandBean {
    private NodeList list;
    private int listLength;
    private int listIndex = -1;
    private Element elem;

    public GetPlansBean() {
        super();
        addParam("method", "get-plans");
    }

    public void setLastName(String lastName) {
        addParam("last-name", lastName);
    }
    public void setFirstName(String firstName) {
        addParam("first-name", firstName);
    }
    public void setSpecialty(String specialty) {
        addParam("specialty", specialty);
    }
    public void setPlanId(String planId) {
        addParam("plan-id", planId);
    }
    public void setPlanType(String planType) {
        addParam("plan-type", planType);
    }

    public void execute() throws HttpCommandBeanException {
        callServer();
        list = getTopElement().getElementsByTagName("plan-id");
        listLength = list.getLength();
    }

    public String getPlanId() {
        return elem.getFirstChild().getNodeValue();
    }
```

```
    public boolean next() {
        if (++listIndex < listLength) {
            elem = (Element)list.item(listIndex);
            return true;
        } else {
            return false;
        }
    }
}
```

Code Example 7.41 GetPlansBean Command Bean

```
package com.providence.plan.cb;

public class HttpCommandBeanException extends java.lang.Exception {
    private Exception causeException = null;

    public HttpCommandBeanException() {
    }

    public HttpCommandBeanException(String message) {
        super(message);
    }

    public HttpCommandBeanException(Exception  ex) {
        super();
        causeException = ex;
    }

    public Exception getCausedByException() {
        return causeException;
    }
}
```

Code Example 7.42 HttpCommandBeanException Command Bean

7.5.2.5 XML Message Formats

XML stands for eXtensible Markup Language. XML is the industry standard used both for documents and for exchange of data between organizations. The structure of an XML document is defined using Document Type Description (DTD).

The following DTD defines the formats of the XML messages returned from ProvidenceServiceWeb to ProvidencePlanEJB. Code Example 7.43 shows the DTD for Providence's messages.

```
<!ELEMENT plan-ids (plan-id*)>
<!ELEMENT plan-id (#PCDATA)>

<!ELEMENT plans (plan*)>
<!ELEMENT plan (plan-id, plan-name, plan-type)>
<!ELEMENT plan-name (#PCDATA)>
<!ELEMENT plan-type (#PCDATA)>

<!ELEMENT premium-quote (coverage, age, smoker, cost)>
<!ELEMENT coverage (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT smoker (#PCDATA)>
<!ELEMENT cost (#PCDATA)>

<!ELEMENT doctors (doctor*)>
<!ELEMENT doctor (last-name, first-name, specialty,
       practice-since, hospital)>
<!ELEMENT last-name (#PCDATA)>
<!ELEMENT first-name (#PCDATA)>
<!ELEMENT specialty (#PCDATA)>
<!ELEMENT practice-since (#PCDATA)>
<!ELEMENT hospital (#PCDATA)>
```

Code Example 7.43 XML DTD for Providence

In conjunction with the above DTD, Table 7.1 describes the HTTP requests and XML replies between the ProvidencePlanBean and the ProvidenceServiceWeb Web application.

Table 7.1 XML Replies to Providence HTTP Requests

Method	Query String in Request	Returned XML
`ejbFindByPrimaryKey`	`method=get-plans&plan-id=ID`	`plan-ids`
`ejbFindMedicalPlans`	`method=get-plans&plan-type=medical`	`plan-ids`
`ejbFindDentalPlans`	`method=get-plans&plan-type=dental`	`plan-ids`
`ejbFindByDoctor`	`method=get-plans?last-name=LN&first-name=FN`	`plan-ids`
`ejbLoad`	`method=get-plan-info&plan-id=ID`	`plan`
`getAllDoctors`	`method=get-doctors&plan-id=ID`	`doctors`
`getDoctorsByName`	`method=get-doctors&plan-id=ID&last-name=LN&firstname=FN`	`doctors`
`getDoctorsBySpecialty`	`method=get-doctors&plan-id=ID&specialty=SP`	`doctors`
`getCost`	`method=calc-cost&plan-id=ID&coverage=CV&age=AG&smoker=SM`	`cost`

The table describes the HTTP requests sent by the ProvidencePlanBean class methods and the corresponding XML replies sent by ProvidenceServiceWeb. The arguments `ID`, `LN`, `FN`, `SP`, `CV`, `AG`, and `SM` are, respectively, the plan identifier, doctor's last name, doctor's first name, doctor's specialty, coverage category, age, and smoker status.

For example, the `getDoctorsByName` bean method might send the following request to ProvidenceServiceWeb to find all the doctors whose last names start with the string "Ford" and who participate in the plan identified by plan id `PRUD-01`.

```
http://serviceweb.Providence.com/star/plans?method=get-doc-
tors&plan-id=PRUD-01&last-name=Ford&first-name=
```

ProvidenceServiceWeb might send the following reply to the above request.

```
<doctors>
        <doctor>
                <last-name>Ford</last-name>
                <first-name>Harrison</first-name>
                <specialty>Surgeon</specialty>
                <practice-since>1990</practice-since>
                <hospital>Stanford</hospital>
        </doctor>
        <doctor>
                <last-name>Ford</last-name>
                <first-name>Betty</first-name>
                <specialty>Pediatrics</specialty>
                <practice-since>1992</practice-since>
                <hospital>Palo Alto Clinic</hospital>
        </doctor>
</doctors>
```

We assumed in the example that the environment entry `java:comp/env/ProvidenceServiceWebURL` of ProvidencePlanEJB was set to `http://service-web.providence.com/star/plans`, the URL at which ProvidenceServiceWeb maintains the plan information for the plans that it offers to Star Enterprise.

7.5.3 ProvidenceServiceWeb Web Application

ProvidenceServiceWeb is a Web application which consists of a number of servlets. The servlets handle the HTTP requests sent by ProvidencePlanBean instances; they return the requested plan information in XML format. This book does not describe the design of ProvidenceServiceWeb.

7.5.4 Packaging

This section described the packaging of the parts developed by Providence.

7.5.4.1 providence.ear Archive File

Providence packages the parts that support the benefits application as a single J2EE Enterprise Application Archive file which it names `providence.ear`. Figure 7.11 depicts the contents of the `providence.ear` file.

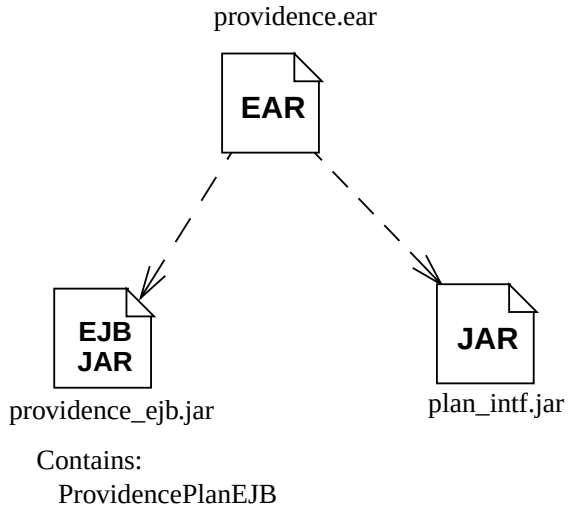

Figure 7.11 Providence's Enterprise Application Archive File

The `providence.ear` file contains two parts:

- **`providence_ejb.jar` file**—This ejb-jar file contains the ProvidencePlanEJB enterprise bean developed by Providence.
- **`plan_intf.jar` file**—This is the JAR file published by Wombat. Because the ProvidencePlanEJB depends on the contents of this JAR file at runtime, it is included in the `providence.ear` file. The `Class-Path` attribute in the Manifest of the `providence_ejb.jar` file lists this dependency on the `plan_intf.jar` file.

7.5.4.2 providence_ejb.jar File

The `providence_ejb.jar` file is an ejb-jar file that contains the enterprise beans developed by Providence. Code Example 7.44 shows the Java class files contained by the file.

```
com.providence.plan.ProvidencePlan.class
com.providence.plan.ProvidencePlanBean.class
com.providence.plan.ProvidencePlanHome.class

com.providence.plan.db.GetCostBean.class
com.providence.plan.db.GetDoctorsBean.class
com.providence.plan.db.GetPlanInfoBean.class
com.providence.plan.db.GetPlansBean.class
com.providence.plan.db.HttpCommandBean.class
com.providence.plan.db.HttpCommandBeanException.class
```

Code Example 7.44 providence_ejb.jar File

7.6 Summary of the Integration Techniques

The complete benefits application includes parts developed by several companies. This section discusses the techniques used to integrate these different parts into an operational application. As the discussion proceeds, notice how the EJB architecture simplifies the integration of components from multiple organizations.

We start the discussion by focusing on the key integration points. These are the points where the parts developed by different organizations come together. Figure 7.12 illustrates these integration points.

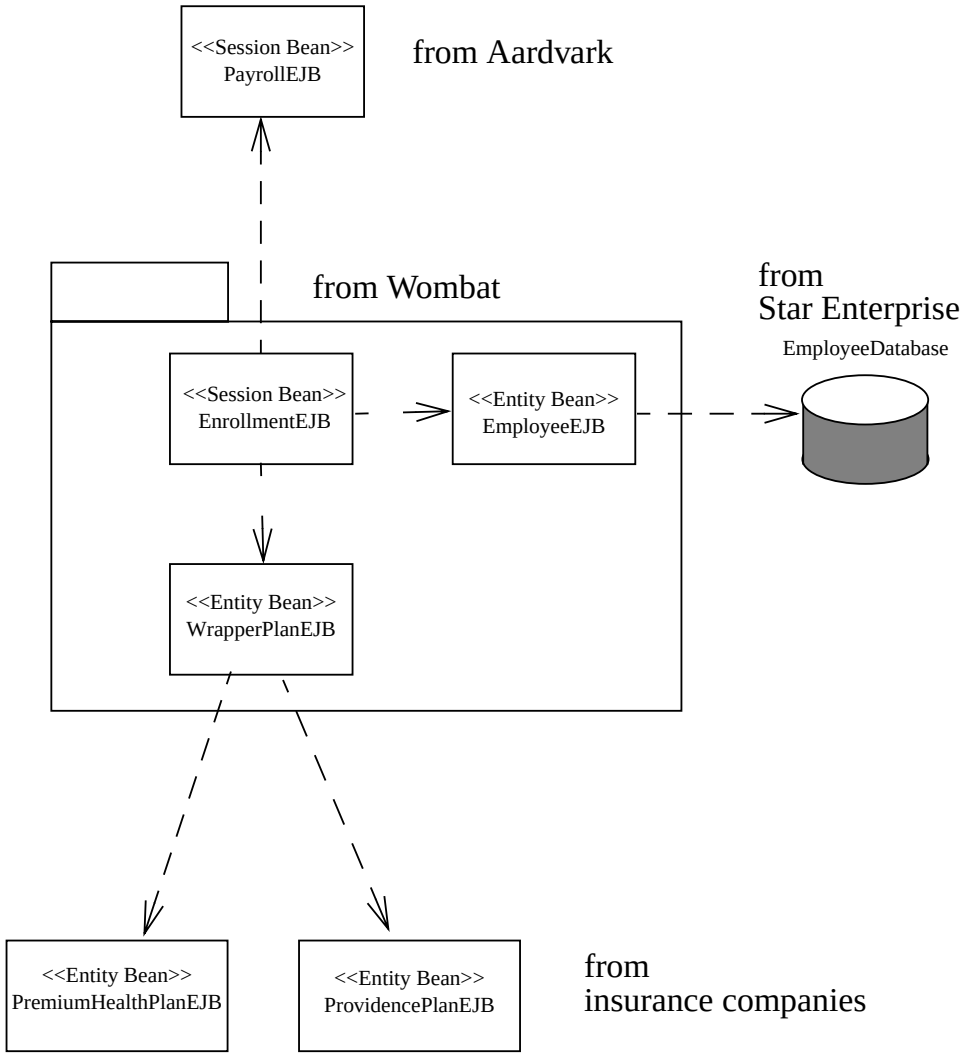

Figure 7.12 Integration of the Benefits Application Parts

The application illustrates the following three different techniques for integrating components.

- The client developer relies on the Bean Provider to publish the EJB client-view interfaces. This is the case of the Wombat's EnrollmentEJB, which is depen-

dent on PayrollEJB from Aardvark. EnrollmentEJB's dependency is as a client of PayrollEJB.

- The Bean Provider relies on the client provider to publish the EJB client-view interfaces so that it can develop an entity bean that can be integrated with the client application. This is the case of the integration of the WrapperPlanEJB from Wombat with the individual plan entity beans, such as ProvidencePlanEJB and PremiumHealthEJB, from the insurance companies. WrapperPlanEJB is the client of the ProvidencePlanEJB and PremiumHealthEJB entity beans.
- The Bean Provider uses container-managed persistence to allow the integration with its customers' existing databases or applications. This is the case of the integration of Wombat's EmployeeEJB with the pre-existing EmployeeDatabase at Star Enterprise.

The following subsections explain these different techniques in more detail.

7.6.1 Bean Provider Publishes EJB Client-view Interfaces

Let's look at how to integrate EnrollmentEJB from Wombat with PayrollEJB from vendor Aardvark. PayrollEJB implements the remote access to Star Enterprise's payroll system and can be used by non-payroll applications to access the payroll system. For example, EnrollmentEJB uses PayrollEJB to update the payroll information at the end of the benefits enrollment process.

The following six steps describe the integration of these two enterprise beans. Note that in our scenario, PayrollEJB pre-existed the development of Wombat's Benefits application.

1. Aardvark implements PayrollEJB.
2. Aardvark creates the `payroll_ejb_client.jar` JAR file, which contains the client-view interfaces and classes for PayrollEJB's client view. Aardvark makes the `payroll_ejb_client.jar` file available to companies that have a need to invoke PayrollEJB.
3. Star Enterprise deploys PayrollEJB at its site. When it deploys PayrollEJB, it also makes the PayrollEJB's home object available to other applications in the JNDI name space.
4. Wombat obtains the `payroll_ejb_client.jar` file from Aardvark. The class EnrollmentBean imports the package `com.aardvark.payroll` and uses the Payroll and PayrollHome interfaces to invoke PayrollEJB.

5. Wombat declares the dependency on PayrollEJB interfaces using the EJB reference mechanism. The EJB reference is an environment entry in the EnrollmentEJB's deployment descriptor.
6. When the Deployer at Star Enterprise deploys the application from Wombat, he discovers the EJB reference to PayrollEJB. He resolves the EJB reference by linking it to the PayrollEJB's home object, which was created in step 3.

Keep in mind that the `payroll_ejb_client.jar` file created by Aardvark and used by Wombat contains all the Java class files that comprise the PayrollEJB's client view. Wombat needs the `payroll_ejb_client.jar` when compiling the EnrollmentBean class. Because the EnrollmentEJB classes need access to the `payroll_ejb_client.jar` file at runtime, Wombat includes the `payroll_ejb_client.jar` file in the `benefits.ear` file. (The `benefits.ear` file contains the parts of the applications developed by Wombat.)

7.6.2 Client Provider Publishes EJB Client-view Interfaces

Let's look at how Wombat's Benefits enrollment application integrates with the individual plan entity beans provided by the insurance companies. This is a good illustration of how a bean developed by one vendor implements the EJB client-view interfaces defined by another vendor.

1. Wombat designs the Plan and PlanHome interfaces. Wombat assumes that these interfaces will be implemented by the plan entity beans developed by the individual insurance companies. Wombat packages the class files for these interfaces with their dependent class files into the `plan_intf.jar` JAR file, and makes the `plan_intf.jar` file available to the insurance companies.
2. Wombat develops the Benefits enrollment application. The WrapperPlanBean component functions as the client of the plan entity beans developed by the insurance providers. WrapperPlanEJB uses the Plan and PlanHome interfaces to invoke the individual plan beans.[1]
3. An insurance company, such as Providence, obtains the `plan_intf.jar` file from Wombat. Providence implements the ProvidencePlanEJB entity bean such that its remote and home interfaces are compatible with the Plan and PlanHome interfaces.

---

[1] The fact that WrapperPlanBean also implements the Plan and PlanHome interface in its client-view is irrelevant to the discussion here.

4. The Deployer at Star Enterprise deploys the Wombat Benefits enrollment application.
5. The Deployer at Star Enterprise deploys the ProvidencePlanEJB entity bean from Providence so that the home interface of ProvidencePlanEJB is made available to the BenefitsAdminWeb Web application. The Deployer does this by binding the home interface object to an entry in the JNDI context that BenefitsAdminWeb uses to find all the deployed plan beans.
6. The BenefitsAdminWeb Web application finds the home interface of ProvidencePlanEJB at runtime, displays all the individual plans offered by Providence to Star Enterprise, and allows the Star Enterprise's benefits plan administrator to add one or more of the benefits plans to the list of plans aggregated by the WrapperPlanEJB entity bean. The benefits plan administrator adds the Providence benefits plans by calling the `create` method on the WrapperPlanEJB's home interface. This creates wrapper plan objects that are then used by the EnrollmentEJB bean.

While the discussion of the integration of the benefits application with the individual plan beans seems complicated, in reality it actually is not. The apparent complication is due to Wombat's action combining the integration of the individual plans with their aggregation into a single entity bean (WrapperPlanEJB) at one time.

7.6.3 Use of Container-managed Persistence

In this section, we look at how the Bean Provider can use container-managed persistence to integrate the application with existing databases and customer applications.

1. The customer, Star Enterprise in this case, has EmployeeDatabase. Star Enterprise created EmployeeDatabase before the deployment of the Wombat Benefits application. The designer of the EmployeeDatabase schema had no knowledge of the Wombat application. Many Star Enterprise applications use the EmployeeDatabase to obtain employee information.
2. Wombat develops the Benefits application. It develops the EmployeeEJB entity bean that provides the application's interface to the customer's employee data. The remainder of the Wombat application, including the EnrollmentEJB session bean, uses the EmployeeEJB client-view interfaces to access the customer's employee data. It is important to note that Wombat had no knowledge of the Star Enterprise's EmployeeDatabase schema when developing the Em-

ployeeEJB entity bean—Wombat had to design the EmployeeEJB bean such that it works with many customers' employee databases.

3. When Star Enterprise deploys the Wombat application, the Deployer has to bind the container-managed fields of EmployeeEJB to columns in the EmployeeDatabase database, which stores the Star Enterprise's employee information. The Deployer may either use the tools provided by the container vendor or manually write code that performs the binding. For example, the Deployer could write code to subclass the EmployeeBean class to create an entity bean with bean-managed persistence, as illustrated with the SelectionEJB and WrapperPlanEJB entity beans.

Container-managed persistence allows not only the integration of an EJB application with a relational database, but also the integration of an EJB application with non-EJB applications. For example, the EmployeeEJB bean can be integrated with an Enterprise Resource Planning (ERP) system that a customer uses to manage its employee information. Because most ERP systems do not expose the database schema to third party applications, the third party applications need to invoke the integration interfaces published by the ERP vendor rather than accessing the ERP system's database directly. For example, if the benefits application needed to be integrated with an ERP system, the deployment-generated classes that implement the EmployeeEJB bean's container-managed persistence would make calls to the ERP applications rather than directly accessing the database used by the ERP system.

7.7 Conclusion

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

The example application clearly illustrates the differences, from a developer's point of view, of using entity beans. It focused on the various techniques for working with entity beans, such as caching persistent state, aggregating application functionality into a single component, subclassing techniques, among others, and how best to use the features of these types of beans.

The chapter showed how entity beans are more appropriate for applications that must be easily adapted for different customers with different operational envi-

ronments. Typically, these are applications built by ISVs rather than applications developed by an enterprise's in-house IT department.