Apuntes

de

J2EE

Tema 4:
EJB

Uploaded by

Ingteleco

http://ingteleco.webcindario.com

ingtelecoweb@hotmail.com

La dirección URL puede sufrir modificaciones en el futuro. Si no funciona contacta por email

4.- MODELO DE COMPONENTES EJB

4.1.- ENTERPRISE JAVABEANS

La arquitectura Enterprise JavaBeans (EJB a partir de ahora) define un modelo de componentes que permite el desarrollo de componentes Java destinados a ejecutarse en la parte servidora de una aplicación.

Estos componentes, denominados también Enterprise JavaBeans, son componentes remotos, escritos en Java y que se ejecutan remotamente en un servidor, denominado Servidor de EJBs, fuera del cual no tienen ningún sentido.

Gracias a estos componentes, la tecnología EJB permite el desarrollo de aplicaciones distribuidas multicapa basadas en componentes, en las que la lógica de negocio de la aplicación queda soportada mediante un conjunto de componentes EJB. La utilidad fundamental de los componentes EJB es, por tanto, soportar la lógica de negocio de las aplicaciones.

Además, el modelo EJB proporciona un framework de servicios para que sea usado por los componentes EJB. Así, el desarrollador puede implementar sus componentes sin necesidad de tener que implementar todo el middleware de servicios subyacentes. El desarrollador simplemente se tienen que preocupar de implementar los procesos de negocio de sus componentes, haciendo uso de toda la infraestructura de servicios que le proporciona la arquitectura EJB.

A continuación se expondrán más en detalle algunas de las caracteristicas más importantes de la tecnología EJB.

4.1.1.- Portabilidad

El modelo de componentes EJB asegura la portabilidad de sus componentes a lo largo de diferentes servidores de EJBs, aun cuando la implementación que cada servidor haga de los servicios que tiene que proporcionar a sus componentes sea diferente.

Esta portabilidad se consigue gracias a que la especificación de los EJBs define un contrato entre los componentes EJB y el contenedor sobre el que se ejecutan basado en el uso de un API estándar. Cada componente EJB, según este contrato, debe implementar un conjunto de interfaces que permitan al contenedor de EJBs manipular y gestionar los componentes que se ejecutan sobre él de una manera estándar. A su vez, el contenedor debe comprometerse a invocar los interfaces de los componentes según una serie de normas dictadas por la especificación y a proporcionar, a su vez, una serie de interfaces estándar que permitan a los componentes acceder a los servicios del servidor donde se ejecutan.

De esta forma, si tanto el desarrollador de componentes EJB como la compañía que implementa el servidor de EJBs siguen los contratos que marca la especificación, la portabilidad de los componentes entre distintos servidores está garantizada ya que tanto el acceso a la infraestructura de servicios del servidor por parte de los componentes, como la manipulación de los componentes por parte del servidor se realizará de una forma estándar: a través del API estándar.

Por tanto, podemos concluir que el modelo EJB permite el desarrollo de componentes portables entre distintos servidores de EJBs, siempre y cuando estos últimos cumplan las normas que dicta la especificación de los EJBs.

4.1.2.- Reusabilidad

Un componente EJB es una unidad de software reusable que se ejecuta en el servidor y que lleva a cabo un conjunto bien definido de funciones que son publicadas y hechas accesibles al resto de componentes y aplicaciones a través de un interfaz.

De esta forma, un componente EJB se puede implementar una única vez para que realice una determinada función y ser reutilizado posteriormente en múltiples aplicaciones que requieran esa funcionalidad.

Mediante el desarrollo de EJBs las empresas pueden crear sus propias librerías de componentes reusables, de manera que a partir de ese momento el desarrollo de aplicaciones para esa empresa se simplifique, hasta el punto en que se reduzca a ir seleccionando de estas librerías aquellos componentes reusables que implementen la funcionalidad requerida por la aplicación y a ir combinándolos hasta formar la aplicación que se quiere desarrollar. El desarrollo de aplicaciones se reduce a la combinación de componentes reusables ya implementados.

El modelo EJB, gracias a que esta basado en el lenguaje Java y gracias a los contratos estándar definidos en su especificación, nos permite crear componentes portables y reusables que se escriben una única vez y que pueden ser reutilizados en múltiples aplicaciones y ejecutados en cualquier servidor que esté acorde con la especificación. (Write Once, Run Anywhere)

4.1.3.- Simplifica el desarrollo: Framework de servicios

La arquitectura EJB proporciona un framework de servicios que simplifica enormemente el proceso de desarrollo de aplicaciones basadas en componentes EJB. El servidor de EJBs sobre el que se ejecutan los componentes proporciona a éstos un importante número de servicios middleware, de tal forma que el desarrollador de componentes puede centrarse unicamente en el desarrollo de los procesos de negocio que implementan sus componentes, sin necesidad de tener que preocuparse de implementar todo el middleware de servicios subyacente. La arquitectura EJB delega todos estos servicios en el servidor de EJBs.

Este hecho permite agilizar y simplificar el proceso de desarrollo de componentes, al mismo tiempo que logra que el código de éstos sea de mayor calidad al centrarse sólo en los procesos de negocio y hacer uso de los servicios proporcionados por el servidor.

Basándose en la especificación de los EJBs, donde se indican los servicios básicos que debe proporcionar un servidor de EJBs, las compañías software implementan los servidores de EJBs comerciales sobre los que luego el desarrollador de aplicaciones despliega sus componentes.

4.1.4.- Roles en el desarrollo de aplicaciones

La especificación de los EJBs asigna roles específicos a los participantes en el desarrollo de una aplicación mediante componentes EJB. Algunos de estos roles son los siguientes:

- **Desarrollador de EJBs:** se encarga de implementar la funcionalidad de cada componente EJB.

- **Ensamblador de componentes:** se encarga de combinar los componentes que le suministra el desarrollador para que proporcionen el funcionamiento que requiere la aplicación.

- **Encargado del despliegue de componentes:** se encarga de instalar los componentes sobre el servidor de EJBs, encargándose de proporcionar al servidor los parámetros adecuados para que éste maneje a los componentes y les proporcione los servicios requeridos de la forma adecuada.

- **Compañía comercial:** se encarga de implementar el servidor de EJBs sobre el que ejecutaremos los componentes.

4.1.5.- Soporte para transacciones distribuidas

De entre los múltiples servicios que proporciona la arquitectura EJB cabe destacar el servicio de transacciones. Los componentes EJB son componentes transaccionales.

El modelo EJB, a través del servidor de EJBs, proporciona control de transacciones a sus componentes así como soporte para transacciones distribuidas de forma transparente. Esto quiere decir que el desarrollador puede implementar sus componentes sin necesidad de tener que escribir una sola línea de código que haga referencia al tratamiento de transacciones, y sin embargo, sus componentes gozarán de soporte para transacciones y podrán participar en transacciones distribuidas gracias a que su servidor se encarga de proporcionarles dicho soporte.

4.1.6.- Fácil integración con la tecnología CORBA

La tecnología EJB se complementa muy fácilmente con la tecnología CORBA gracias a la compatibilidad con el protocolo IIOP (Interner Inter-ORB Protocol)

Así, un cliente CORBA puede acceder a la funcionalidad que proporciona un componente EJB como si de un componente EJB se tratase y, a su vez, en un futuro los servidores de EJBs encapsularán (a través de un API estándar) servicios proporcionados por la arquitectura CORBA, de tal manera que podrán ser accedidos por los componentes EJB como si de objetos CORBA se tratasen.

4.2.- EL MODELO EJB

La arquitectura básica EJB consta de los siguiente elementos:

- Un **Servidor de EJBs.**

- Uno o varios **Contenedores de EJBs** que se ejecutan dentro del servidor.

- El objeto **HomeObject**, el objeto **EJBObject** y el **Enterprise JavaBean** que se ejecutan dentro del contenedor.

- El **Interfaz Home** y el **Interfaz Remoto.**

- El **Cliente EJB.**

- El **Conjunto de Servicios** implementados por el servidor.

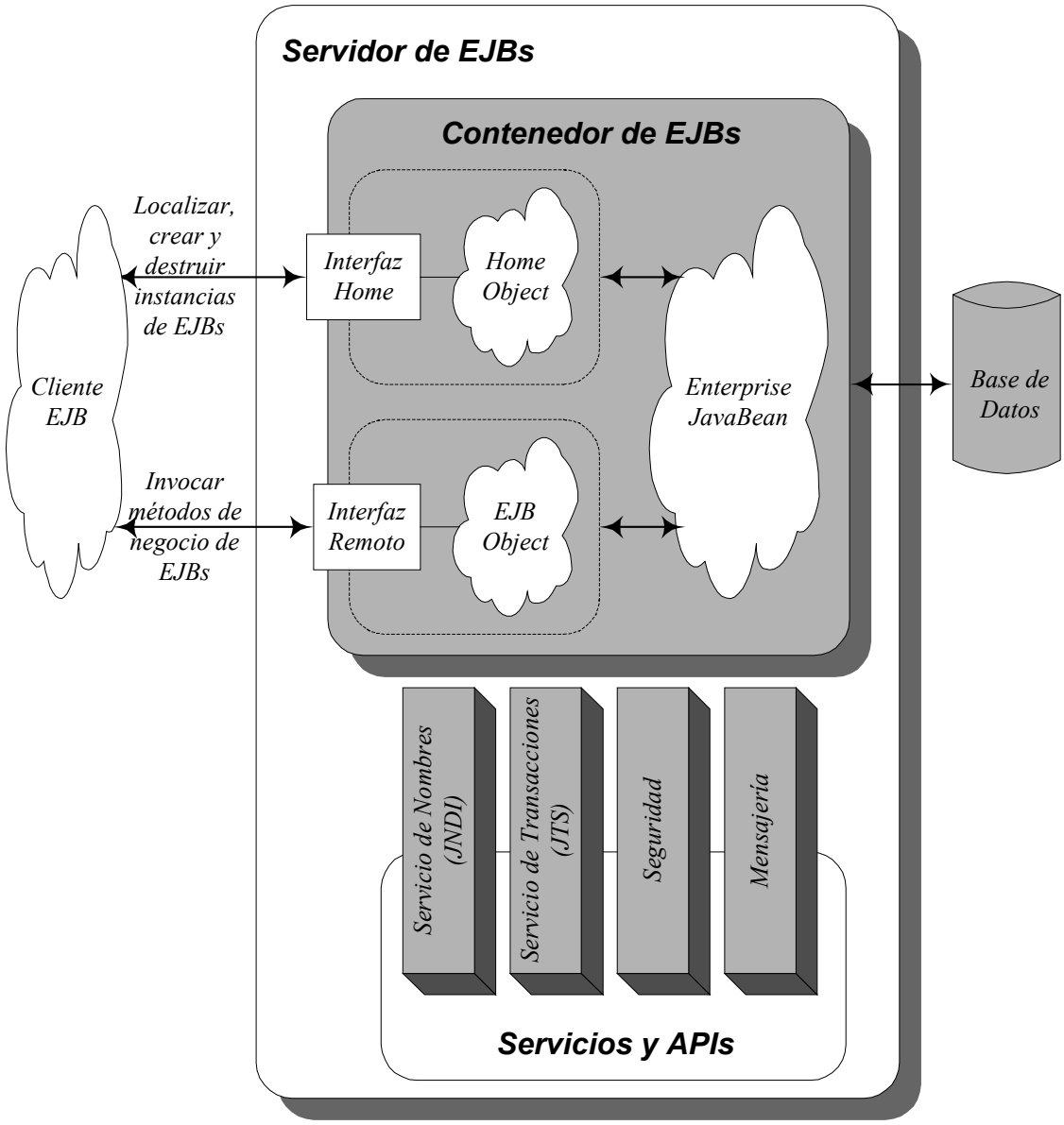

Figura 4.1: Arquitectura básica Enterprise JavaBean

4.2.1.- Servidor de EJBs

El servidor de EJBs proporciona un entorno de ejecución en el que residen uno o más contenedores EJB. Un servidor de EJBs implementa una serie de servicios tales como el servicio de nombres y directorio (JNDI), el servicio de transacciones (JTS), servicios de seguridad,... y se encarga de hacerlos accesibles a los contenedores a través de una serie de interfaces.

Además, los servidores de EJBs se pueden encargar de proporcionar servicios que permitan automatizar aspectos complejos de las aplicaciones distribuidas como pueden ser la administración y reciclaje de recursos (conexiones a bases de datos, conexiones de red, memoria,...) o el balanceo de cargas entre varios servidores de EJBs.

4.2.2.- Contenedor de EJBs

El contenedor de EJBs proporciona un entorno de ejecución para los componentes EJB, de la misma manera que un servidor web proporciona un entorno para la ejecución de Servlets o un navegador para la ejecución de Applets.

El contenedor de EJBs se encarga de gestionar ciertos aspectos de la ejecución de los componentes que se encuentran dentro de él, proporcionándoles una serie de servicios. Algunos de los servicios que el contenedor proporciona a sus componentes son:

Gestión del ciclo de vida

El contenedor se encarga de gestionar de manera automática el ciclo de vida de los componentes que se ejecutan dentro de él. Se encarga de la creación y destrucción de instancias, asignación y liberación de recursos (memoria, conexiones,...), etc.

Como parte del ciclo de vida de los EJB se incluye la activación y desactivación de los objetos. Cuando el contenedor lo crea oportuno, generalmente porque el objeto lleva mucho tiempo sin ser usado, el contenedor se encarga de salvar el estado del objeto en un almacenamiento secundario, eliminándolo de memoria y liberando de esta forma recursos. Cuando ese objeto vuelva a ser requerido por un cliente, el contenedor se encargará de volver a traer ese objeto desde el almacenamiento secundario hasta memoria.

El proceso por el cual el estado de un EJB es salvado a un almacenamiento persistente y posteriormente eliminado de memoria, se denomina ***Passivation***. El proceso contrario en el que el estado de un EJB es recuperado del almacenamiento persistente e instanciado en memoria, se denomina ***Activation***.

Gestión del estado

El contenedor se encarga automáticamente de mantener o eliminar el estado de los componentes después de cada invocación de un método por parte del cliente. El contenedor tomará una decisión u otra en función de las propiedades que se le hayan indicado en el momento del despliegue del componente. (***Stateless vs Statefull***)

Gestión de transacciones

El contenedor proporciona soporte para transacciones a todos los componentes que se ejecutan dentro de él. De esta forma, los componentes EJB se convierten en componentes transacciones que pueden verse involucrados en transacciones distribuidas sin necesidad de que el desarrollador escriba una sola línea de código relativa al tratamiento de las transacciones. Será el contenedor quien de forma transparente soporte las transacciones de sus componentes, encargándose de establecer los límites de cada transacción así como de efectuar de manera correcta el comienzo y la finalización de cada transacción (Commit o Rollback).

Gestión de persistencia

El contenedor de EJBs se encarga de gestionar la persistencia de todos sus componentes. El contenedor se encargará de determinar cuando el estado de un EJB debe ser salvado en la base de

datos (o cualquier otro soporte de almacenamiento) así como cuando la información que está en la base de datos debe ser recuperada para refrescar el estado del componente que está en memoria.

En este sentido, el contenedor es quien tiene la responsabilidad de que la información de los EJBs que estén en memoria esté en todo momento sincronizada con la información que se encuentra almacenada en la base de datos.

Existen distintos grados en los que un contenedor puede gestionar la persistencia de los componentes EJB. Podemos tener un control total de la persistencia por parte del contenedor, de tal forma que éste decida cuando un EJB debe salvar o recuperar su información de la base de datos y sea el propio contenedor quien se encargue del almacenamiento o recuperación de la misma. Y por otra parte, podemos tener un control de la persistencia parcial, en la que el contenedor decida cuando salvar o recuperar la información de la base de datos, pero por contra, él no realice la actualización de la información, sino que simplemente se encargue de invocar el código que el desarrollador ha implementado para tal efecto, siendo, por tanto, el código del desarrollador quien realice los accesos a la base de datos y no el propio contenedor.

Estos conceptos se verán más adelante cuando se traten las configuraciones ***Bean Managed Persistence*** y ***Container Managed Persistence***.

Seguridad

El contenedor se puede encargar de garantizar la seguridad de los EJBs mediante chequeos en los accesos que los clientes realicen sobre los componentes.

Todos estos servicios permiten que los EJBs disfruten de todas estas facilidades sin necesidad de que el desarrollador escriba una sola línea de código para ello.

Logicamente, el contenedor no trata de la misma forma a todos sus componentes, sino que proporcionará estos servicios de distinta manera a cada componente en función de cómo de le haya indicado que lo haga. Para ello, en el momento en que un componente es desplegado sobre el contenedor, se proporciona a éste un archivo llamado ***Deployment Descriptor*** en el que se indica el modo en que el contenedor debe tratar el EJB, de tal manera que sepa qué tipo de seguridad le debe proporcionar, como debe gestionar su persistencia, como debe gestionar su estado, etc.

Según todo lo anterior, el contenedor es el corazón de cualquier entorno EJB. El contenedor se encarga de registrar los abjetos EJB para que puedan ser accedidos remotamente, gestiona las transacciones entre los clientes y los EJBs, proporciona control de acceso sobre ciertos métodos de los EJBs, controla la creación, activación, desactivación y destrucción de objetos EJB, administra los recursos usados por los componentes, etc. Además y para que todo esto sea posible, el contenedor se encargará en el momento del despliegue de un componente de generar una serie de objetos (se verán posteriormente) que le ayudarán a gestionar y dar servicio a ese componente que está siendo desplegado.

El contenedor de EJBs envuelve a los componentes que residen dentro de él, no permitiendose el acceso directo con cualquier elemento del exterior. En este sentido, el contenedor actúa como un interface entre los componentes EJB y el exterior, de tal forma que cualquier acceso al exterior que quiera ser realizado por un componente, los deberá realizar a través del contenedor.

Cuando un cliente invoca un método de un EJB, el contenedor se encarga de interceptar la invocación, asegurandose de esta forma que la invocación pasa a través de él. Si un componente quiere acceder a alguno de los servicios proporcionados por el servidor de EJBs (por ejemplo el servicio JNDI) lo deberá haver a través del API estandar que le proporciona el contenedor.

A su vez, un cliente EJB que quiera acceder a un componente EJB nunca lo hará directamente sino a través del contenedor que contiene ese EJB. Cada vez que un cliente invoque un método de un EJB, el contenedor interceptará la invocación, de tal forma que se encargará de proporcionar todos aquellos servicios de transacciones, seguridad, persistencia,... que esa llamada requiera, para posteriormente pasar la invocación del método al componente para el que iba destinada inicialmente.

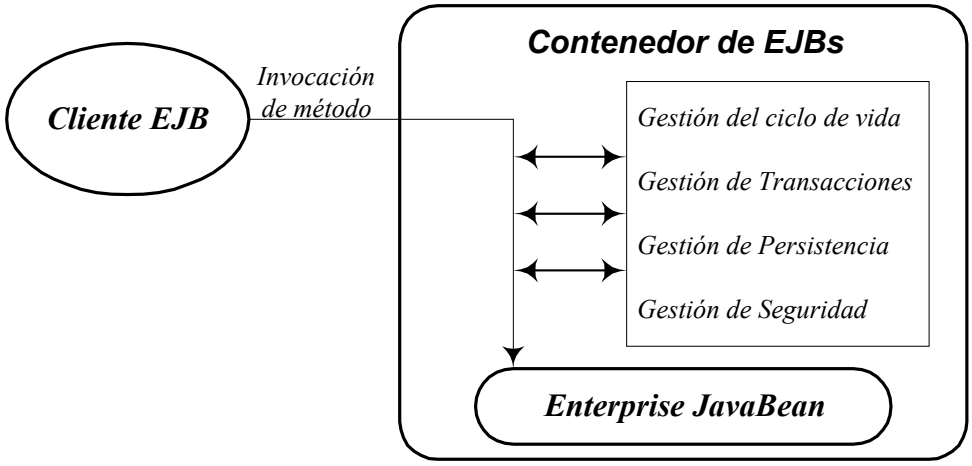

Figura 4.2: Invocación de un método del EJB a través del contenedor

El contenedor, interceptando estas invocaciones, se asegura de que todos los accesos a sus EJBs pasan a través de él y de esta forma puede proporcionar a sus componentes todos los servicios de manera transparente. La captura de la invocación se realizará mediante dos objetos generados automáticamente por él durante el despliegue de los componentes, como son el objeto EJBHome y el objeto EJBObject que se estudiarán posteriormente.

La especificación de los EJBs define una serie de contratos que indican como debe ser la relación del contenedor con el resto de elementos:

Contrato entre el contenedor y el servidor de EJBs

El servidor de EJBs implementará una serie de servicios a los que el contenedor deberá acceder para poder proporcionárselos él a los componentes que gestiona. Por ejemplo, el contenedor para poder dar soporte a las transacciones de sus componentes deberá hacer uso del servicio de transacciones que tiene implementado el servidor donde reside.

La diferencia fundamental entre el servidor y el contenedor radica en que el contenedor se encarga de gestionar y dar servicio a los EJBs que se ejecutan dentro de él, y el servidor de EJBs se encarga de implementar una serie de servicios externos a los componentes EJB. El contenedor hará uso de los servicios del servidor para servir luego él a sus EJBs.

En la práctica, la diferencia entre el servidor y el contenedor no siempre está tan clara y suele usarse un término u otro indistintamente.

Contrato entre el contenedor y el cliente EJB

El contenedor deberá presentar a los clientes de EJBs una serie de interfaces que permitan a los clientes utilizar los EJBs. Deberá proporcionar interfaces que permitan crear, buscar, destruir y utilizar los componentes. Así como mecanismos (JNDI) para obtener estos interfaces.

Contrato entre el contenedor y los componentes EJB

Por una parte, el contenedor debe disponer de un interfaz estándar (definido en la especificación) que permita a los componentes EJB acceder a los servicios que proporciona el contenedor. Por otra parte, debe existir un mecanismo que permita al contenedor informar a sus componentes EJB de todos aquellos eventos que ocurran durante su ciclo de vida. Este mecanismo se denomina ***callback*** y se basa en que los componentes implementen una serie de métodos, llamados ***métodos callback*** y definidos en la especificación, que serán invocados por el contenedor para informarles de ciertos eventos. Cada método ***callback*** alertará al EJB de un evento diferente en su ciclo de vida y el contenedor los invocará para notificarles, por ejemplo, que el EJB va a ser salvado a la base de datos, o que va a ser desactivado de memoria.

4.2.3.- Interfaz Home y objeto HomeObject

El ***Interfaz Home*** define una serie de métodos que nos permiten crear, localizar y destruir instancias de un EJB. Contiene, por tanto, métodos relacionados con el ciclo de vida de los componentes EJB. Todos los componentes EJB deben tener un interfaz Home y éste debe ser definido por el desarrollador del componente.

El ***objeto HomeObject*** es un objeto que implementa el interfaz Home de un componente EJB. Sin embargo, mientras que el interfaz Home debe ser definido por el desarrollador del componente, el objeto HomeObject es generado automáticamente por el contenedor cada vez que un EJB es desplegado sobre él. De esta forma, cada vez que un componente se despliega sobre un contenedor, el contenedor genera una clase que implementa el interfaz Home del componente, crea un instancia de esta clase y la hace accesible a los clientes. Esta instancia es el objeto HomeObject.

Cada vez que un cliente quiera hacer uso de los servicios de un EJB, éste deberá crear o localizar uno mediante su interfaz Home. El cliente localizará el objeto HomeObject creado por el contenedor (este objeto implementa el interfaz Home) y lo usará para crear un nuevo componente o localizar uno ya existente.

4.2.4.- Interfaz Remoto y objeto EJBObject

El ***Interfaz Remoto*** de un componente contiene los métodos de negocio que ese componente expone a sus clientes. Todo componente EJB debe poseer un interfaz remoto con sus métodos de negocio. El interfaz remoto debe ser definido por el desarrollador del componente. El cliente de un componente sólo podrá invocar aquellos métodos del componente que estén en su interfaz remoto.

El ***objeto EJBObject*** es un objeto generado automáticamente por el contenedor que implementa el interfaz remoto de un componente. Por tanto, el desarrollador es responsable de definir el interfaz remoto de un componente, mientras que el contenedor es el encargado, en el momento del despliegue del componente, de generar un objeto que implemente ese interfaz remoto.

Cuando un cliente quiere usar un componente EJB, obtiene una referencia a su interfaz remoto (al objeto EJBObject que la implementa) e invoca los métodos de ese interfaz. Esto quiere decir que el cliente siempre que invoca un método del EJB lo hace sobre el objeto EJBObject y no sobre el propio componente. El cliente nunca trabaja directamente sobre el EJB sino que lo hace sobre el objeto EJBObject. El objeto EJBObject es la vista que el cliente tiene del EJB y el objeto que usa para invocar los métodos del componente, nunca los invoca directamente.

Cuando hablábamos del contenedor, deciamos que éste se encargaba de interceptar todos las llamadas que el cliente hacía sobre los componente EJB. Es precisamente a través de los objetos EJBObject y EJBHome como el contenedor logra interceptar todas la invocaciones. El cliente nunca logra invocar directamente a un componente EJB sino que los objetos EJBObject y EJBHome siempre se ponen por medio e interceptan la llamada.

De esta forma, cuando un cliente invoca un método de negocio de un EJB, éste lo hace sobre el objeto EJBObject en lugar de hacerlo sobre el propio componente. El objeto EJBObject se pone en contacto con el contenedor para informarle de que ese método va a ser invocado y para que así el contenedor aporte todos los servicios implicitos que esa invocación requiera. Una vez que el contenedor ha hecho su labor, el objeto EJBObject delega la invocación del método sobre el componente EJB sobre el que iba destinada inicialmente.

Dado un tipo de componente EJB, existirá un único objeto HomeObject compartido por todas las instancias de ese componente y un objeto EJBObject para cada instancia de ese componente.

4.2.5.- El Enterprise JavaBean

Todo componente EJB posee una clase que implementa toda la funcionalidad que ese componente debe proporcionar.

La clase que implementa un Enterprise JavaBean debe proporcionar la implementación de los métodos de negocio definidos en el interfaz remoto, así como otra serie de métodos usados por el contenedor para gestionar el EJB y definidos en el contrato que une el contenedor con el componente EJB.

Es, por tanto, en esta clase en la que el desarrollador del componente EJB deberá escribir la mayor parte del código que forma el EJB. Cada instancia de esta clase será una nueva instancia de un componente EJB.

Como ya se ha comentado, el cliente no accede nunca directamente a los EJBs sino que siempre lo hace a través del contenedor.

4.2.6.- El cliente EJB

Un cliente EJB es cualquier entidad software que accede a los servicios proporcionados por un componente EJB.

El cliente hará uso del interfaz Home (implementado por el objeto EJBHome) para localizar, crear y destruir instancias de EJBs y del interfaz remoto (implementado por el objeto EJBObject) para invocar los métodos de negocio de una instancia del componente.

El cliente obtendrá la referencia al interfaz Home de un EJB haciendo uso del servicio JNDI proporcionado por el servidor de EJBs. Cuando un componente EJB es desplegado sobre un

contenedor, el contenedor se encargará de generar un objeto HomeObject para ese componente y de registrarlo en el servicio de nombres JNDI con un determinado nombre. Luego los clientes podrán localizarlo haciendo uso de este servicio.

4.2.7.- Interacción típica de un cliente EJB con un componente EJB

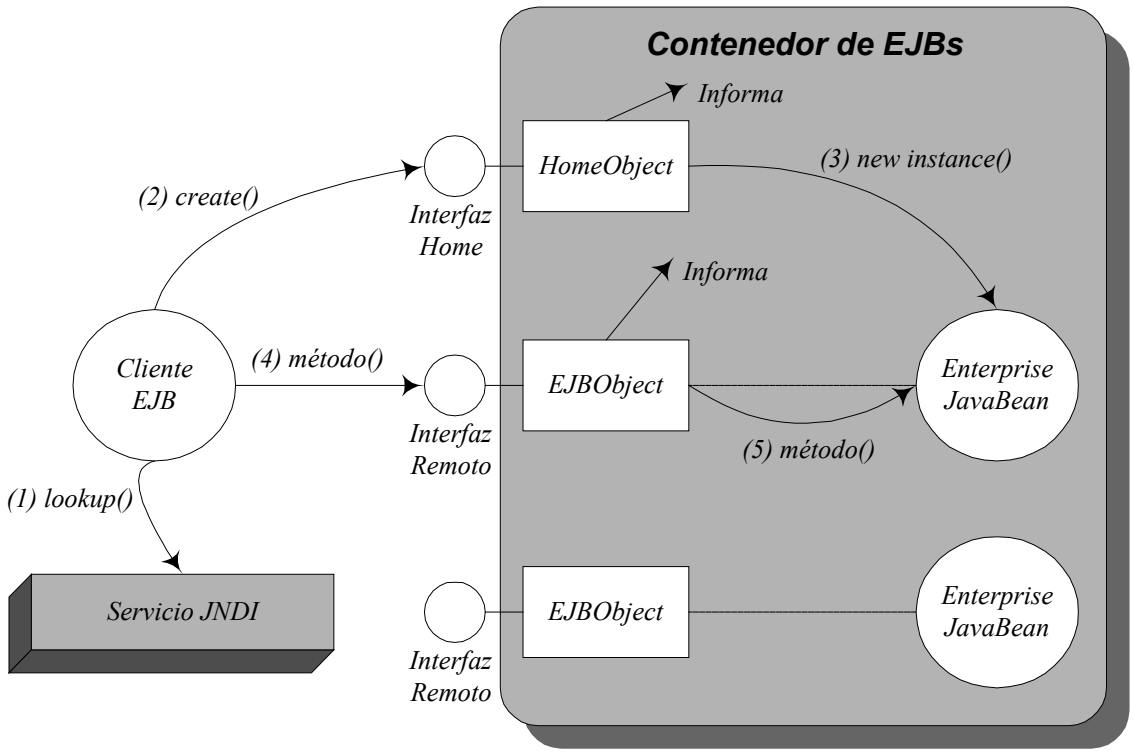

Figura 4.3: Interacción típica de un cliente con un componente EJB

A grandes rasgos, los pasos que un cliente EJB sigue para usar los servicios de un componente EJB son los siguientes:

1. El cliente accede al servicio JNDI y obtiene una referencia al interfaz Home del EJB (una referencia al objeto HomeObject).

2. El cliente invoca el método `create()` del interfaz Home para crear una nueva instancia del EJB.

3. El objeto HomeObject recibe la invocación, se pone en contacto con el contenedor para informarle de que se va a crear una nueva instancia del componente EJB, crea la nueva instancia y devuelve al cliente una referencia al interfaz remoto de esa instancia (una referencia al nuevo objeto EJBObject que también de habrá creado).

4. El cliente invoca uno de los métodos de negocio del interfaz remoto.

5. El objeto EJBObject recibe la invocación, informa al contenedor para que aporte toda la funcionalidad requerida y, finalmente, delega la ejecución del método al EJB, que será quien contenga la funcionalidad implementada por el desarrollador del componente.

4.3.- COMPONENTES EJB

De todos los elementos que constituyen el modelo EJB, los únicos que el desarrollador debe implementar a la hora de crear un componente EJB son el ***Interfaz Home***, el ***Interfaz Remoto*** y la ***Clase EJB***. El resto de elementos de la arquitectura son proporcionados por el contenedor y por tanto, el desarrollador no tiene que ocuparse de ellos.

A partir de este momento nos centraremos únicamente en los tres elementos que un desarrollador debe implementar para crear un componente EJB: ***Interfaz Home***, ***Interfaz Remoto*** y ***Clase EJB***.

Nos basaremos para su explicación en el desarrollo de un EJB simple llamado ***Adder***, que tiene como única finalidad ir sumando una serie de números que le vayamos proporcionando.

4.3.1.- Interfaz Home

El interfaz Home de un EJB contiene una serie de métodos que permiten crear, localizar y destruir instancias de ese EJB.

A continuación se muestra el interfaz Home de nuestro EJB *Adder*:

```
package adderExample;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface AdderHome extends EJBHome
{
   Adder create() throws RemoteException, CreateException;
   Adder create(int initial) throws RemoteException, CreateException;
}
```

Figura 4.4: Código del Interfaz Home AdderHome

A continuación se indican algunas características que cabe destacar de este interfaz:

- El interfaz Home siempre extiende del interfaz `javax.ejb.EJBHome`.
- Un interfaz Home puede contener múltiples métodos `create()` que permiten al cliente crear nuevas instancias del EJB e inicializarlas.

  En nuestro ejemplo, el interfaz Home contiene dos métodos `create()`, uno de ellos sin argumentos y otro con un argumento de tipo `int`. Ambos métodos nos permiten crear nuevas instancias del EJB *Adder*. El segundo de ellos nos permitirá además inicializarle con el valor pasado como parámetro.
- Todos los métodos del interfaz Home deben lanzar la excepción `java.rmi.RemoteException`, que se producirá cuando ocurra algún error en la comunicación durante la invocación del método.
- Además, los métodos `create()` del interfaz deben lanzar la excepción `javax.ejb.CreateException`, que se producirá cuando ocurra algún error durante la creación del EJB.

- Todos los argumentos y valores de retorno de los métodos del interfaz Home deben ser tipos primitivos, objetos de tipo *Serializable* u objetos remotos RMI.

- Los métodos `create()` del interfaz Home devuelven como retorno una referencia al interfaz remoto del objeto EJB que crean. En nuestro ejemplo, los métodos `create()` devuelven una referencia de tipo `Adder` que es precisamente el tipo del interfaz remoto de nuestro EJB *Adder*.

- Cada método `create()` del interfaz Home debe tener su correspondiente método `ejbCreate()` en la clase EJB.

  Cada vez que el cliente invoque el método `create()` del interfaz Home, el contenedor se encargará de crear una nueva instancia del EJB y posteriormente invocará su método `ejbCreate()` correspondiente.

  Cada método `create()` se emparejará con un método `ejbCreate()` que tendrá los mismos argumentos. El método `ejbCreate()` será donde el desarrollador escriba el código que quiere que se ejecute cuando se cree el nuevo EJB.

4.3.2.- Interfaz Remoto

El interfaz Remoto define los métodos de negocio del EJB que van a poder ser invocados por el cliente.

En nuestro ejemplo, el EJB *Adder* posee dos métodos de negocio: uno que permite darle un número para que lo sume, y otro que permite consultar la cantidad total que lleva acumulada el componente *Adder*.

A continuación se muestra su interfaz Remoto:

```
package adderExample;

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

public interface Adder extends EJBObject
{
   void add(int number) throws RemoteException;
   int getTotal() throws RemoteException;
}
```

Figura 4.5: Código del Interfaz Remoto Adder

A continuación se indican las características más destacadas de este interfaz:

- El interfaz Remoto siempre extiende del interfaz `javax.ejb.EJBObject`.

- Todos los métodos del interfaz Remoto deben lanzar la excepción `java.rmi.RemoteException`, que se producirá en caso de que ocurra algún error en la comunicación durante la invocación del método.

- Los argumentos y tipo de retorno de los métodos del interfaz Remoto deben ser tipos primitivos, objetos de tipo *Serializable* u objetos remotos RMI.

- Los métodos del interfaz Remoto deben coincidir exactamente con métodos implementados en la clase EJB. A diferencia de lo que ocurría con el interfaz Home, donde cada método tenía un método en la clase EJB con el que se emparejaba pero sin ser su signatura idéntica, en el interfaz Remoto todos sus métodos deben tener un método en la clase EJB con el que coincidan exactamente en su signatura.

- Como se puede apreciar, el interfaz Remoto de nuestro EJB se denomina `Adder`, que es precisamente el tipo de la referencia que nos devolvían los métodos `create()` del interfaz `AdderHome`. Los métodos `create()` del interfaz `AdderHome` nos devuelven una referencia a este interfaz Remoto para que luego el cliente, con esa referencia, pueda invocar cualquiera de sus métodos: `add()` o `getTotal()`.

4.3.3.- La Clase EJB

Una vez que tenemos definido el interfaz Home que nos permite crear nuevas instancias del EJB y el interfaz Remoto que define los métodos que nuestro EJB va a hacer accesible a los clientes, debemos crear una clase que implemente el componente EJB propiamente dicho. Esta clase es la que denominamos ***Clase EJB***.

La clase EJB debe implementar todos los métodos de negocio definidos en el interfaz Remoto del EJB, proporcionar métodos que se correspondan con los métodos definidos en el interfaz Home para crear y localizar EJBs, así como implementar una serie de métodos que serán usados por el contenedor para gestionar y mantener informado al componente (***métodos callback***).

A continuación se muestra el código de la clase EJB de nuestro *Adder*:

```
package adderExample;

import javax.ejb.*;

public class AdderEJB implements SessionBean
{
   int total;

   // Implementación de los métodos del interfaz home
   public void ejbCreate() {
      total = 0;
   }

   public void ejbCreate(int initial) {
      total = initial;
   }

   // Implementación de los métodos de negocio del interfaz remoto
   public void add(int number) {
      total += number;
   }
   public int getTotal() {
      return total;
   }

   // Implementación de los métodos callback
   public AdderEJB() {}
   public void setSessionContext(SessionContext sc) {}
```

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

} // AdderEJB
```

Figura 4.6: Código de la clase EJB AdderEJB

Las características que cabe destacar de la clase EJB son las siguientes:

- Se puede apreciar como la clase `AdderEJB` implementa todos los métodos definidos en el interfaz Remoto `Adder`: `add()` y `getTotal()`.

  Sin embargo, la clase EJB no implementa directamente el interfaz Remoto (no declara la cláusula `implements Adder`) sino que quien verdaderamente implementa el interfaz Remoto será un objeto EJBObject generado por el contenedor. Este objeto lo que hará es delegar la ejecución de los métodos en la clase EJB, que es quien verdaderamente contiene la implementación de los métodos que ha proporcionado el desarrollador.

- La clase `AdderEJB` contiene dos métodos `ejbCreate()` que se corresponden con los dos métodos `create()` definidos en el interfaz Home. Como se puede apreciar, los métodos `ejbCreate()` de la clase EJB deben tener los mismos parámetros que los métodos `create()` del interfaz Home. No ocurre, sin embargo, lo mismo con el valor de retorno, que en el caso de los métodos `create()` es el interfaz Remoto del componente (`Adder`) y en el caso de `ejbCreate()` es `void`.

- La clase EJB debe ser declarada `public` para que el contenedor pueda acceder a los métodos del EJB cuando sea necesario. Lógicamente, sus métodos también deben ser `public`.

- Toda clase EJB debe proporcionar un constructor sin parámetros, en este caso `public AdderEJB()`.

- El resto de métodos son los denominados ***métodos callback***. Estos métodos son invocados por el contenedor para informar al EJB de ciertos eventos que ocurran durante su ciclo de vida. De esta forma, el desarrollador puede incluir en ellos el código que quiere que se ejecute cuando ocurra un determinado evento. Más adelante se explicará el significado de cada uno de estos métodos.

4.3.4.- Tipos de componentes EJB

Los componentes EJB son entidades software que implementan pequeños módulos de lógica de negocio y que pueden ser combinados para construir complejas aplicaciones multicapa. Existen dos tipos de componentes EJB: ***Session Beans*** y ***Entity Beans***.

4.3.4.1.- Entity Beans

Los Entity Beans representan datos persistentes que se encuantran almacenados en una base de datos (o en cualquier otro almacenamiento persistente).

El estado de un Entity Bean siempre se corresponde con la información que está en la base de datos y que él representa. En un ámbito relacional, un instancia de un Entity Bean se correspondería con una fila de una tabla de la base de datos.

Los Entity Bean son por naturaleza persistentes. Su existencia se extiende más allá del tiempo de vida de una aplicación o del propio servidor sobre el que se ejecutan. En caso de que el servidor falle, al estar la información del Entity Bean almacenada en la bases de datos, cuando el servidor se vuelva a poner en funcionamiento, la información se recuperará de la base de datos y el Entity Bean seguirá existiendo.

Al representar datos almacenados en una base de datos, los Entity Bean pueden ser compartidos por varios clientes, de tal forma que varios clientes pueden acceder a un mismo Entity Bean simultaneamente.

Todo Entity Bean debe poseer una clave primaria que le distinga del resto de instancias de ese Entity Bean.

Los Entity Bean, al representar datos almacenados en una base de datos, necesitan de un control de persistencia que les permita tener en todo momento su estado sincronizado con el de la base de datos. Existen dos maneras distintas de gestionar la persistencia de un Entity Bean: ***Container Managed Persistence (CMP)*** y ***Bean Managed Persistence (BMP)***.

4.3.4.1.1.- Container Managed Persistence (CMP)

La persistencia del Entity Bean es gestionada completamente por el contenedor, de tal manera que toda la lógica necesaria para sincronizar el estado del Entity Bean con el de la base de datos es proporcionada automáticamente por el contenedor.

El contenedor se encarga tanto de decidir cuando el estado del Entity Bean debe salvarse o recuperarse de la base de datos como de realizar los correspondientes accesos a la base de datos. El desarrollador no necesita escribir código para realizar los accesos a la base de datos, sino que éste es proporcionado de manera transparente por el contenedor.

4.3.4.1.2.- Bean Managed Persistence (BMP)

La persistencia del Entity Bean es gestionada a medias entre el contenedor y el propio Entity Bean.

En este caso, el Entity Bean tiene que encargarse de realizar los accesos a la base de datos necesarios para escribir su estado en la base de datos o para recuperarlo de ella. Por tanto, en este caso, el desarrollador tiene que encargarse de escribir dentro del propio Bean el código JDBC necesario para realizar los accesos a la base de datos.

El contenedor, sin embargo, seguirá ayudando al Entity Bean a gestionar su persistencia. El contenedor se encargará de decidir en qué momento es necesario leer o escribir el estado del Entity Bean en la base de datos e invocará el código de acceso a la base de datos (implementado por el desarrollador) cuando sea necesario.

Ejemplos de Entity Beans

Candidato a ser un Entity Bean puede ser cualquier entidad cuya información sea susceptible de ser almacenada en una base de datos, como por ejemplo los productos de un supermercado, los clientes de una tienda, las reservas de un hotel,...

4.3.4.2.- Session Beans

Los Session Beans no son objetos persistentes, no representan datos persistentes almacenados en una base de datos como los Entity Bean. En su lugar, representan tareas o procesos de negocio que se realizan a petición de un cliente.

Los Session Beans suelen usarse para coordinar las interacciones entre Entity Beans, coordinar otros Session Beans, realizar accesos a bases de datos y, en general, realizar cualquier tipo de tarea a petición de un cliente.

En este sentido, un Session Bean es una extensión de un cliente que se ejcuta remotamente en un servidor de EJBs. Habitualmente, un Session Bean es usado por un único cliente y la información de estado que mantiene es, generalmente, información perteneciente a ese cliente. Los Session Bean no suelen ser compartidos por más de un cliente.

Los Session Bean no son persistentes, suelen ser creados por el cliente que los va a usar y su vida suele estar limitada a la vida de ese cliente. En cualquier caso, la existencia un Session Bean nunca perdura más allá de la del servidor sobre el que se ejecutan.

Existen dos tipos de Session Bean: ***Stateless Session Bean*** y ***Stateful Session Bean***.

4.3.4.2.1.- Stateless Session Bean

Los Stateless Session Bean son Session Bean sin estado. Los Stateless Session Bean no mantienen información de estado entre invocaciones de métodos.

Cada invocación de un método en un Stateless Session Bean es independiente de todas las invocaciones anteriores. El Bean no recuerda nada de una invocación a la siguiente ya que el contenedor se encarga de eliminar la información de estado del Bean al final de cada invocación. Si un cliente realiza una serie de invocaciones sobre un Stateless Session Bean, el Bean se encontrará siempre en el mismo estado al principio de cada invocación.

Puesto que no tienen estado, todas las instancias de un Stateless Session Bean serán idénticas y no habrá forma de diferenciar unas de otras. Si un cliente invoca un método sobre una instancias de un Stateless Session Bean en un determinado momento, obtendrá el mismo resultado que si invoca ese mismo método sobre cualquier otra instancias y en cualquier otro momento. Todas las instancias son identicas y, por tanto, proporcionan el mismo servicio.

Esto da lugar a que se formen *Pools* donde se ubican Stateless Session Beans para que sean compartidos por varios clientes.

Ejemplos de Session Beans

Son candidatos a ser Session Beans aquella entidades que tengan como finalidad la realización de procesos de negocio y cuyos datos no se tengan que almacenar en una base de datos. Por ejemplo: una entidad que haga reservas en un hotel, una entidad que realice la facturación de una compra,...

4.3.4.2.2.- Stateful Session Bean

Los Stateful Session Bean son Session Bean con estado que sí mantienen información de una invocación de método a otra.

Cada Stateful Session Bean suele estar dedicado en exclusiva a un solo cliente (no se comparte entre varios clientes) y la información de estado que almacena suele ser relativa a ese cliente.

4.3.4.3.- Diferencias entre Session Beans y Entity Beans

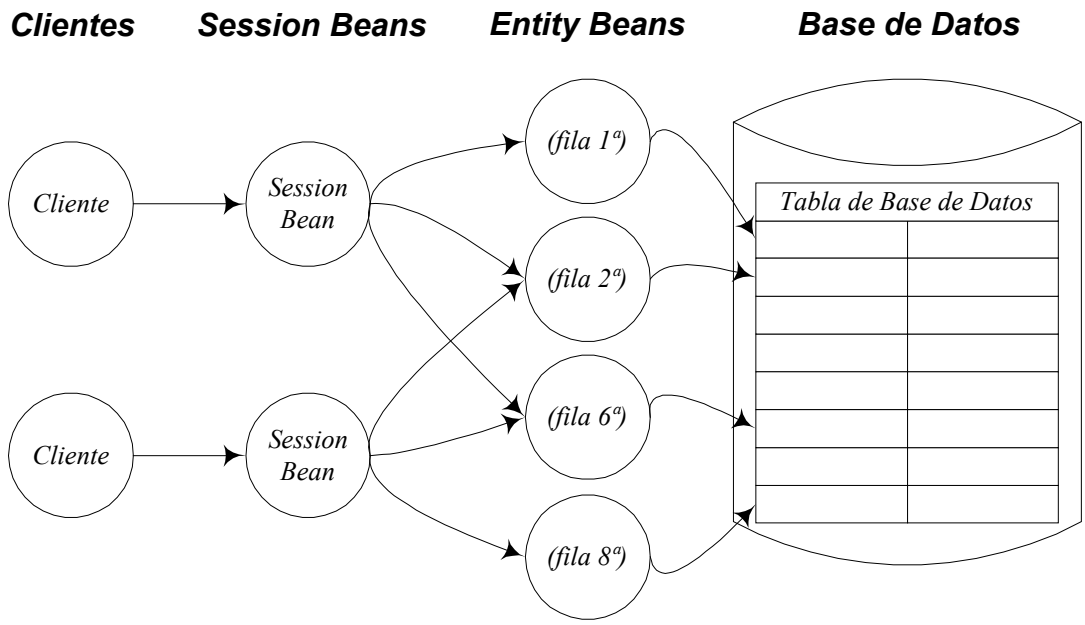

Figura 4.7: Relación entre Session Beans y Entity Beans

Session Beans	Entity Bean
La información de estado de un Session Bean no representa datos almacenados en una base de datos.	La información de estado de un Entity Bean representa datos almacenados en una base de datos.
Un Session Bean se encuantra unido a un único cliente y, por tanto, está capacitado para almacenar información perteneciente a ese cliente.	Un Entity Bean es compartido por varios clientes y, por tanto, no puede almacenar información relativa a todos los clientes.
La relación entre un Session Bean y su cliente es uno-a-uno.	La relación entre un Entity Bean y una fila de una tabla relacional es uno-a-uno.
La vida de un Session Bean está limitada a la vida de su cliente.	Un Entity Bean perdura tanto tiempo como los datos que representa existan en la base de datos.
Los Session Bean no sobreviven a fallos en su servidor.	Los Entity Bean sobreviven a fallos en su servidor.

4.4.- IMPLEMENTACIÓN DE SESSION BEANS

En este apartado se indicarán las nociones básica necesarias a la hora de implementar Session Beans. Para ello, nos basaremos en dos ejemplos: uno de un Stateless Session Bean y otro de un Stateful Session Bean.

4.4.1.- Implementación de un Stateless Session Bean

Se tomará como ejemplo un Stateless Session Bean llamado ***Converter***. Este EJB nos permite realizar conversiones monetarias, permitiéndonos convertir de Dolars a Yens y de Yens a Euros.

A continuación se muestra el código completo del *Bean Converter*:

```
package converterExample;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface ConverterHome extends EJBHome {

    Converter create() throws RemoteException, CreateException;
}
```

Figura 4.8: Código del Interfaz Home CoverterHome

```
package converterExample;

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

public interface Converter extends EJBObject {

   public double dollarToYen(double dollars) throws RemoteException;
   public double yenToEuro(double yen) throws RemoteException;
}
```

Figura 4.9: Código del Interfaz Remoto Converter

```
package converterExample;

import java.rmi.RemoteException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class ConverterEJB implements SessionBean {

   // Implementación de los métodos de negocio del interfaz Remoto
   public double dollarToYen(double dollars) {
      return dollars * 121.6000;
   }

   public double yenToEuro(double yen) {
      return yen * 0.0077;
   }
```

```
   // Implementación de los métodos del interfaz home
   public void ejbCreate() {}

   // Implementación de los métodos callback
   public ConverterEJB() {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void setSessionContext(SessionContext sc) {}

} // ConverterEJB
```

Figura 4.10: Código de la clase EJB ConverterEJB

Como se puede apreciar, el *Bean Converter* no tiene estado, de hecho ni siquiera tiene atributos que le permitan mantener un estado interno. Los métodos `dolarToYen()` y `yenToEuro()` usan únicamente la información que reciben como parámetros para proporcionar su respuesta y por tanto, siempre que sean invocados con los mismos parámetros, devolverán la misma respuesta. El *Bean Converter* es, por tanto, un Stateless Session Bean.

Si observamos el código del interfaz Home `ConverterHome` y del interfaz Remoto `Converter`, vemos que no aporta ninguna novedad frente a lo que ya conocíamos, sigue exactamente las mismas reglas que indicamos en apartados anteriores.

Sin embargo, el código de la clase EJB `ConverterEJB` presenta ciertos elementos que anteriormente dejamos sin explicar y que ahora es el momento de hacerlo:

- La clase EJB de todo Session Bean debe implementar el interfaz `javax.ejb.SessionBean`. Nuestro *Bean Converter* es un Session Bean y por tanto, la clase `ConverterEJB` implementa este interfaz.

- El interfaz `SessionBean` define una serie de métodos, llamados ***métodos callback***, que deben ser implementados por la clase `ConverterEJB`. Estos métodos serán invocados por el contenedor para informar al EJB de eventos que ocurran durante su ciclo de vida. Los métodos del interfaz `SessionBean` y su significado son los siguientes:

  - **`ejbActivate()`**: Este método es invocado por el contenedor inmediatamente después de que el EJB sea traído a memoria desde el almacenamiento secundario en el que anteriormente había sido desactivado. Se invoca inmediatamente después de ser activado (*Swap in*).

    En este método el desarrollador puede escribir código que permita recuperar los recursos que el EJB liberó cuando fue desactivado de memoria.

  - **`ejbPasivate()`**: Este método es invocado por el contenedor justo antes de que el EJB vaya a ser eliminado de memoria y llevado a un almacenamiento secundario. Se invoca justamente antes de que sea desactivado (*Swap out*).

    En este método el desarrollador puede escribir código que permita liberar los recursos que tenga asignado el EJB.

- **`ejbRemove()`**: Este método es invocado por el contenedor justamente antes de que el EJB sea destruido.

- **`setSessionContext()`**: Este método es invocado por el contenedor inmediatamente después de que el EJB sea creado. Este método recibe como parámetro un objeto de tipo `SessionContext`. El objeto `SessionContext` es un interfaz con el contenedor que permite al Session Bean comunicarse con él y obtener información variada.

Mediante estos métodos el contenedor puede informar al EJB de eventos que ocurran durante su ciclo de vida y el desarrollador puede ubicar dentro de ellos lo que quiere que se ejecute cuando ocurra un determinado evento.

4.4.2.- Implementación de un Stateful Session Bean

Como ejemplo de un Stateful Session Bean tomaremos el ***Bean Adder*** que ya vimos en apartados anteriores. A continuación se vuelve a mostrar el código de este EJB:

```
package adderExample;

import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface AdderHome extends EJBHome
{
   Adder create() throws RemoteException, CreateException;
   Adder create(int initial) throws RemoteException, CreateException;
}
```

Figura 4.11: Código del Interfaz Home AdderHome

```
package adderExample;

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

public interface Adder extends EJBObject
{
   void add(int number) throws RemoteException;
   int getTotal() throws RemoteException;
}
```

Figura 4.12: Código del Interfaz Remoto Adder

```
package adderExample;

import javax.ejb.*;

public class AdderEJB implements SessionBean
{
   int total;

   // Implementación de los métodos del interfaz home
   public void ejbCreate() {
```

```
      total = 0;
   }

   public void ejbCreate(int initial) {
      total = initial;
   }

   // Implementación de los métodos de negocio del interfaz remoto
   public void add(int number) {
      total += number;
   }

   public int getTotal() {
      return total;
   }

   // Implementación de los métodos callback
   public AdderEJB() {}
   public void setSessionContext(SessionContext sc) {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}

} // AdderEJB
```

Figura 4.13: Código de la clase EJB AdderEJB

El EJB *Adder* es un bean con estado. Tiene un atributo `total` que le permite almacenar la cantidad total que lleva acumulada el bean y que constituye su estado interno. Es, por tanto, un Stateful Session Bean.

El código del *Bean Adder* no presenta ninguna novedad que merezca la pena destacar, es muy similar al del *Bean Converter* y cumple todas las características descritas en apartados anteriores.

Como se puede apreciar, el código de un Session Bean no presenta ninguna característica especial que nos permita diferenciar si es un Stateless Session Bean o un Stateful Session Bean. El hecho de que un bean sea Stateless Session Bean o Stateful Session Bean no tiene ningún reflejo en la forma de implementar el bean, ésta es una característica que se determina en el momento del despliegue del bean, donde se indica al contenedor si queremos que nos gestione el bean como un Stateless Session Bean o como un Stateful Session Bean.

4.5.- IMPLEMENTACIÓN DE ENTITY BEANS

En este apartado se estudiarán los conceptos básicos involucrados en la implementación de Entity Beans. Para ello, se tomarán como ejemplo dos Entity Beans que gestionan su persistencia de distinta manera: uno de ellos ***Container Managed Persitence (CMP)*** y el otro ***Bean Managed Persistence (BMP)***.

4.5.1.- Implementación de un Entity Bean con CMP

El ***Bean Product*** que tomaremos como ejemplo es un Entity Bean cuyo modo de gestión de persistencia es ***Container Managed Persistence***.

El *Bean Product* representa los productos que están almacenados en un base de datos. Los datos que el *Bean Product* contiene de cada producto son el **identificativo del producto**, la **descripción** y su **precio**. Mediante este bean el cliente podrá localizar una serie de productos, consultar su información, modificar sus datos, así como crear nuevas instancias.

El interfaz Home del *Bean Product* es el siguiente:

```
package productExample;

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

public interface ProductHome extends EJBHome {

    public Product create(String productId, String description,
        double balance) throws RemoteException, CreateException;

    public Product findByPrimaryKey(String productId)
        throws FinderException, RemoteException;

    public Collection findByDescription(String description)
        throws FinderException, RemoteException;

    public Collection findInRange(double low, double high)
        throws FinderException, RemoteException;
}
```

Figura 4.14: Código del Interfaz Home ProductHome

Este interfaz tiene una serie de elementos que son característicos de los Entity Bean: ***Los métodos de busqueda***.

- Si tenemos en cuenta que los Entity Bean son elementos que existen en una base de datos y que pueden ser compartidos por múltiples clientes, tendremos que proporcionar a los clientes de algún mecanismo que les permita acceder a Entity Beans ya existentes. Este mecanismos son los métodos de búsqueda.

- Los métodos de búsqueda permiten al cliente de los Entity Bean localizar una serie de instancias de ese Entity Bean en función de una serie de criterios.

- Todo método de búsqueda tiene la forma `findXXX`, recibe como parámetro el criterio por el cual va a realizar la búsqueda (generalmente un atributo del bean) y devuelve una referencia al interfaz Remoto del bean, o una colección de referencias en función de si el criterio de búsqueda puede seleccionar a una o varias instancias del bean.

  En nuestro ejemplo, el método `findByPrimaryKey()` es un método de búsqueda predefinido que localiza a un bean dada su clave primaria. Este método recibe como parámetro la clave primaria y devuelve como resultado una referencia al interfaz Remoto de la instancia que tenga esa clave primaria. Todo Entity Bean está obligado a proporcionar este método.

  El método `findByDescription()` localiza a todas aquellas instancias que tengan una determinada descripción. Recibe como parámetro la descripción a partir de la cual se quiere realizar la

búsqueda y devuelve una colección de referencias al interfaz Remoto de todas aquellas instancias que cumplan con esa descripción.

- Todos los métodos de búsqueda deben lanzar la excepción java.rmi.RemoteException, que se producirá cuando ocurra algún fallo en la comunicación durante la invocación del método.

- Todos los métodos de búsqueda deben lanzar la excepción `javax.ejb.FinderException`, que se producirá cuando no exista el Entity Bean que se está buscando.

Es importante darse cuenta de lo que implica la existencia de los métodos de búsqueda. Los Session Bean al ser no persistentes y crearse en exclusiva para servir a un único cliente, la única manera en que un cliente podía usar un Session Bean era creándose una instancia mediante el método `create()` y usándola en exclusiva. En cambio, los Entity Bean al ser entidades persistentes que residen en una base de datos y que pueden ser compartidas por varios clientes, deben permitir que un cliente, además de crear un nuevo Entity Bean, pueda acceder a un Entity Bean ya existente. Por esa razón, los Entity Bean requieren de la existencia de métodos de búsqueda.

Tampoco hay que perder de vista que los Entity Bean son entidades persistentes cuyos datos residen en una base de datos. Esto implica que tanto los métodos de creación (`create()`) como los de búsqueda (`findXXX()`) del interfaz Home, conllevan sus correspondientes accesos a la base de datos. Así, la creación de una instancia de un Entity Bean implicará la creación de una nueva fila en una tabla de la base de datos y la busqueda de instancias de un Entity Bean implicará recorrer la base de datos en busca de ellas.

Con el modelo de persistencia Container Managed Persistence, todos los accesos a la base de datos así como la implementación de los métodos de búsqueda es llevada a cabo por el contenedor de forma transparente. De este modo, evita al desarrollador escribir todo el código encargado de acceder a la base de datos.

El interfaz Remoto del *Bean Product* es el siguiente:

```
package productExample;

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

public interface Product extends EJBObject {

   public void setPrice(double price) throws RemoteException;

   public double getPrice() throws RemoteException;

   public String getDescription() throws RemoteException;
}
```

Figura 4.15: Código del Interfaz Remoto Product

Este interfaz contiene los métodos de negocio que nos permiten consultar y modificar la información del producto.

La clase EJB que implementa el *Bean Product* es la siguiente:

```
package productExample;

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

public class ProductEJB implements EntityBean {

   public String productId;
   public String description;
   public double price;

   private EntityContext context;

   // Implementación de los métodos de negocio del interfaz remoto
   public void setPrice(double price) {
      this.price = price;
   }

   public double getPrice() {
      return price;
   }

   public String getDescription() {
      return description;
   }

   // Implementación de los métodos del interfaz home
   public String ejbCreate(String productId, String description,
      double price) throws CreateException {

      if (productId == null) {
         throw new CreateException("The productId is required.");
      }

      this.productId = productId;
      this.description = description;
      this.price = price;

      return null;
   }

  // Implementación de los métodos callback
   public void setEntityContext(EntityContext context) {
      this.context = context;
   }

   public void ejbActivate() {
      productId = (String)context.getPrimaryKey();
   }

   public void ejbPassivate() {
      productId = null;
      description = null;
   }

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

```
   public void ejbPostCreate(String productId, String description,
      double balance) { }

} // ProductEJB
```

Figura 4.16: Código de la clase EJB ProductEJB

Veamos cuales son las características más destacadas de esta clase:

- Posee una serie de atributos que le permiten almacenar la información del producto. Esta información se encontrará al mismo tiempo almacenada en la base de datos.

- Todo Entity Bean debe tener una clave primaria que identifique de forma unívoca a las instancias de ese Entity Bean. El único requisito que marca la especificación de los EJBs para la clave primaria es que debe ser un objeto (no puede ser un tipo primitivo) y debe ser de tipo *Serializable* (implementar el interfaz `Serializable`).

  Por simplicidad, usaremos siempre como clave primaria para nuestros Entity Beans un objeto de tipo `String`. En el ejemplo del *Bean Product* así se ha hecho y se ha puesto como clave primaria el String `productID`.

- La clase `ProductEJB` implementa los métodos de negocio del interfaz Remoto `Product`. Además, proporciona un método `ejbCreate()` que se corresponde con el método `create()` del interfaz `ProductHome`. Hasta aquí, todo parece igual que con los Session Bean.

  Sin embargo, si nos fijamos en el método `ejbCreate()` observamos que no devuelve `void` como ocurría en los Session Bean, sino que devuelve el tipo de la clave primaria (un `String`) y el valor devuelto por el método siempre es `null`.

  Los métodos `ejbCreate()` de un Entity Bean con Container Managed Persistence tienen dos opciones:

  - Pueden devolver `void` tal y como ocurría en los Session Bean.

  - Pueden devolver el tipo de la clave primaria (`String`) y dentro del método devolver siempre el valor `null`.

  En realidad, lo que devuelvan los métodos `ejbCreate()` en los Entity Bean con Container Managed Persistence no tiene demasiada importancia ya que el contenedor no hace ningún uso del valor devuelto. No ocurrirá lo mismo con los Entity Bean con Bean Managed Persistence como explicaremos más adelante.

- Toda clase EJB de un Entity Bean debe implementar el interfaz `javax.ejb.EntityBean`. Nuestro *Bean Product* es un Entity Bean y por tanto, la clase `ProductEJB` implementa este interfaz.

- El interfaz `EntityBean` define una serie de métodos (***métodos callback***) que deben ser implementados por la clase `ProductEJB`.

  Los métodos del interfaz `EntityBean` y su significado son los siguientes:

- **`ejbActivate()`**: Su significado es parecido al que tenía en los Session Bean. Será invocado por el contenedor inmediatamente después de que el bean sea activado en memoria. (*Activation*)

- **`ejbPasivate()`**: Su significado es parecido al que tenía en los Session Bean. Será invocado por el contenedor justamente antes de que el bean sea desactivado de memoria. (*Passivation*)

- **`ejbRemove()`**: Su significado es idéntico al que tenía en los Session Bean. Será invocado por el contenedor justo antes de que el Entity Bean sea destruido.

- **`ejbLoad()`**: Este método es invocado por el contenedor inmediatamente después de que el estado del bean haya sido recuperado de la base de datos. El contenedor se encargará de acceder a la base de datos, recuperar los datos del bean e inmediatamente después invocar este método.

- **`ejbStore()`**: Este método es invocado por el contenedor justamente antes de que el estado del bean vaya a ser salvado en la base de datos. El contenedor invocará este método e inmediatamente después de su finalización se encargará de almacenar el estado del Entity Bean en la base de datos.

- **`setEntityContext()`**: Este método es invocado por el contenedor inmediatamente después de que el Entity Bean sea creado.

  Este método recibe como parámetro un objeto de tipo `EntityContext`. El objeto `EntityContext` es un interfaz con el contenedor que permite al Entity Bean comunicarse con él y obtener información variada.

- **`unsetEntityContext()`**: Este método es invocado por el contenedor justo antes de que el Entity Bean vaya a ser definido. Su finalidad es eliminar el objeto `EntityContext` que tenga asignado el Entity Bean.

  Como se puede comprobar, en los Entity Bean con CMP la mayoría de estos métodos suelen estar vacíos. Al encargarse el contenedor de todos los accesos a la base de datos, estos métodos no suele hacer falta que contengan nada de código.

- El método **`ejbPostCreate()`** es un método opcional que puede ser implementado o no por la clase EJB de un Entity Bean. En el caso de que se decida implementarlo, deberá estar emparejado con alguno de los métodos `ejbCreate()` que tenga la clase (deberá tener sus mismos argumentos) y será invocado inmediatamente después de la finalización del método `ejbCreate()` con el que se encuentre emparejado.

Como se puede comprobar, la implementación de nuestro Entity Bean *Product* no contiene código JDBC que realice accesos a la base de datos. El desarrollador puede despreocuparse por completo de los accesos a la base de datos. Esto se debe a que el modo de gestión de persistencia usado es Container Managed Persistence y en este modo el contenedor es quien se encarga de proporcionar la lógica de acceso a la base de datos. El contenedor se encarga de implementar los métodos de búsqueda, así como de los accesos a la base de datos necesarios para recuperar, salvar y eliminar el estado del objeto en la base de datos. Esto supone una gran comodidad para el desarrollador.

4.5.2.- Implementación de un Entity Bean con BMP

El ***Bean Account*** que tomaremos como ejemplo es un Entity Bean cuyo modo de gestión de persistencia es ***Bean Managed Persistence***.

El *Bean Account* contiene la información de una serie de cuantas bancarias que se encuentran almacenadas en una base de datos. Los datos que contiene cada cuenta bancaria son el **nombre** y **apellido** del titular de la cuenta y el **saldo** de la cuenta.

Mediante este bean el cliente podrá localizar una serie de cuentas bancarias, consultar sus datos y realizar operaciones que incrementen y decrementen el saldo de la cuenta.

El código completo que implementa este Entity Bean es el siguiente:

```
package accountExample;

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

public interface AccountHome extends EJBHome {

    public Account create(String id, String firstName,
        String lastName, double balance)
        throws RemoteException, CreateException;

    public Account findByPrimaryKey(String id)
        throws FinderException, RemoteException;

    public Collection findByLastName(String lastName)
        throws FinderException, RemoteException;

    public Collection findInRange(double low, double high)
        throws FinderException, RemoteException;
}
```

Figura 4.17: Código del Interfaz Home AccountHome

```
package accountExample;

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

public interface Account extends EJBObject {

    public void debit(double amount)
        throws InsufficientBalanceException, RemoteException;

    public void credit(double amount)
        throws RemoteException;

    public String getFirstName()
        throws RemoteException;

    public String getLastName()
        throws RemoteException;
```

```
    public double getBalance()
        throws RemoteException;
}
```

Figura 4.18: Código del Interfaz Remoto Account

```
package accountExample;

public class InsufficientBalanceException extends Exception {

   public InsufficientBalanceException() { }

   public InsufficientBalanceException(String msg) {
      super(msg);
   }
}
```

Figura 4.19: Código de la excepción InsufficientBalanceException

```
package accountExample;

import java.sql.*;
import javax.sql.*;
import java.util.*;
import javax.ejb.*;
import javax.naming.*;

public class AccountEJB implements EntityBean {

    private String id;
    private String firstName;
    private String lastName;
    private double balance;
    private EntityContext context;
    private Connection con;
    private String dbName = "java:comp/env/jdbc/AccountDB";

   // Implementación de los métodos de negocio del interfaz remoto
    public void debit(double amount)
       throws InsufficientBalanceException {

       if (balance - amount < 0) {
           throw new InsufficientBalanceException();
       }
       balance -= amount;
    }

    public void credit(double amount) {

       balance += amount;
    }

    public String getFirstName() {

       return firstName;
    }

    public String getLastName() {
```

```
        return lastName;
    }

    public double getBalance() {

        return balance;
    }

   // Implementación de los métodos del interfaz home
    public String ejbCreate(String id, String firstName,
        String lastName, double balance)
        throws CreateException {

        if (balance < 0.00) {
            throw new CreateException
                ("A negative initial balance is not allowed.");
        }

        try {
            insertRow(id, firstName, lastName, balance);
        } catch (Exception ex) {
            throw new EJBException("ejbCreate: " +
                ex.getMessage());
        }

        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
        this.balance = balance;

        return id;
    }

   public String ejbFindByPrimaryKey(String primaryKey)
        throws FinderException {

        boolean result;

        try {
            result = selectByPrimaryKey(primaryKey);
        } catch (Exception ex) {
            throw new EJBException("ejbFindByPrimaryKey: " +
                ex.getMessage());
        }
        if (result) {
            return primaryKey;
        }
        else {
            throw new ObjectNotFoundException
                ("Row for id " + primaryKey + " not found.");
        }
   }

   public Collection ejbFindByLastName(String lastName)
        throws FinderException {

        Collection result;

        try {
            result = selectByLastName(lastName);
```

```
     } catch (Exception ex) {
         throw new EJBException("ejbFindByLastName " +
            ex.getMessage());
     }

    if (result.isEmpty()) {
       throw new ObjectNotFoundException("No rows found.");
    }
    else {
       return result;
    }
 }

 public Collection ejbFindInRange(double low, double high)
    throws FinderException {

    Collection result;

    try {
       result = selectInRange(low, high);

     } catch (Exception ex) {
         throw new EJBException("ejbFindInRange: " +
            ex.getMessage());
     }
    if (result.isEmpty()) {
       throw new ObjectNotFoundException("No rows found.");
    }
    else {
       return result;
    }
 }

 // Implementación de los métodos callback
 public void ejbRemove() {

    try {
       deleteRow(id);
     } catch (Exception ex) {
         throw new EJBException("ejbRemove: " +
            ex.getMessage());
     }
 }

 public void setEntityContext(EntityContext context) {

    this.context = context;
    try {
       makeConnection();
    } catch (Exception ex) {
        throw new EJBException("Unable to connect to database. " +
           ex.getMessage());
    }
 }

 public void unsetEntityContext() {

    try {
       con.close();
    } catch (SQLException ex) {
```

```
            throw new EJBException("unsetEntityContext: " + ex.getMessage());
        }
    }

    public void ejbActivate() {

        id = (String)context.getPrimaryKey();
    }

    public void ejbPassivate() {

        id = null;
    }

    public void ejbLoad() {

        try {
            loadRow();
         } catch (Exception ex) {
             throw new EJBException("ejbLoad: " +
                ex.getMessage());
         }
    }

    public void ejbStore() {

        try {
            storeRow();
         } catch (Exception ex) {
             throw new EJBException("ejbLoad: " +
                ex.getMessage());
         }
    }


    public void ejbPostCreate(String id, String firstName,
        String lastName, double balance) { }


/********************** Database Routines ************************/

    private void makeConnection() throws NamingException, SQLException {

        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource) ic.lookup(dbName);
        con =  ds.getConnection();
    }

    private void insertRow (String id, String firstName, String lastName,
                            double balance) throws SQLException {

            String insertStatement =
                  "insert into account values ( ? , ? , ? , ? )";
            PreparedStatement prepStmt =
                  con.prepareStatement(insertStatement);

            prepStmt.setString(1, id);
            prepStmt.setString(2, firstName);
            prepStmt.setString(3, lastName);
            prepStmt.setDouble(4, balance);
```

```
        prepStmt.executeUpdate();
        prepStmt.close();
}

private void deleteRow(String id) throws SQLException {

   String deleteStatement =
         "delete from account where id = ? ";
   PreparedStatement prepStmt =
         con.prepareStatement(deleteStatement);

   prepStmt.setString(1, id);
   prepStmt.executeUpdate();
   prepStmt.close();
}

private boolean selectByPrimaryKey(String primaryKey)
   throws SQLException {

   String selectStatement =
         "select id " +
         "from account where id = ? ";
   PreparedStatement prepStmt =
         con.prepareStatement(selectStatement);
   prepStmt.setString(1, primaryKey);

   ResultSet rs = prepStmt.executeQuery();
   boolean result = rs.next();
   prepStmt.close();
   return result;
}

private Collection selectByLastName(String lastName)
   throws SQLException {

   String selectStatement =
         "select id " +
         "from account where lastname = ? ";
   PreparedStatement prepStmt =
         con.prepareStatement(selectStatement);

   prepStmt.setString(1, lastName);
   ResultSet rs = prepStmt.executeQuery();
   ArrayList a = new ArrayList();

   while (rs.next()) {
      String id = rs.getString(1);
      a.add(id);
   }

   prepStmt.close();
   return a;
}

private Collection selectInRange(double low, double high)
   throws SQLException {

   String selectStatement =
         "select id from account " +
```

```
            "where balance between  ? and ?";
      PreparedStatement prepStmt =
            con.prepareStatement(selectStatement);

      prepStmt.setDouble(1, low);
      prepStmt.setDouble(2, high);
      ResultSet rs = prepStmt.executeQuery();
      ArrayList a = new ArrayList();

      while (rs.next()) {
         String id = rs.getString(1);
         a.add(id);
      }

      prepStmt.close();
      return a;
   }

   private void loadRow() throws SQLException {

      String selectStatement =
            "select firstname, lastname, balance " +
            "from account where id = ? ";
      PreparedStatement prepStmt =
            con.prepareStatement(selectStatement);

      prepStmt.setString(1, this.id);

      ResultSet rs = prepStmt.executeQuery();

      if (rs.next()) {
         this.firstName = rs.getString(1);
         this.lastName = rs.getString(2);
         this.balance = rs.getDouble(3);
         prepStmt.close();
      }
      else {
         prepStmt.close();
         throw new NoSuchEntityException("Row for id " + id +
            " not found in database.");
      }
   }


   private void storeRow() throws SQLException {

      String updateStatement =
            "update account set firstname =  ? ," +
            "lastname = ? , balance = ? " +
            "where id = ?";
      PreparedStatement prepStmt =
            con.prepareStatement(updateStatement);

      prepStmt.setString(1, firstName);
      prepStmt.setString(2, lastName);
      prepStmt.setDouble(3, balance);
      prepStmt.setString(4, id);
      int rowCount = prepStmt.executeUpdate();
      prepStmt.close();
```

```
      if (rowCount == 0) {
         throw new EJBException("Storing row for id " + id + " failed.");
      }
   }

} // AccountEJB
```

Figura 4.20: Código de la clase EJB AccountEJB

El código del interfaz Home `AccountHome` y del interfaz Remoto `Account` no aporta nada novedoso que merezca la pena destacar.

El interfaz `AccountHome` contiene métodos `create()` y métodos de búsqueda (`findXXX()`) exactamente igual que todos los Entity Bean.

El interfaz Remoto `Account` contiene los métodos de negocio del Entity Bean, que permiten consultar la información de la cuenta así como incrementar y decrementar su saldo.

Es la clase EJB `AccountEJB` la que presenta una serie de características propias de todos los Entity Bean con Bean Managed Persistence. A continuación se comentan estas características:

- Se puede observar como existe código JDBC que se encarga de realizar operaciones sobre la base de datos. Al ser un Entity Bean con Bean Managed Persistence, el desarrollador se tiene que encargar de proporcionar el código que se encargue de acceder a la base de datos para salvar, recuperar y eliminar los datos del Entity Bean.

  En los Entity Bean con Container Managed Persistence, el contenedor se encargaba de realizar los accesos a la base de datos y el bean no necesitaba tener implementado este código.

- Los métodos `ejbCreate()` deben contener código JDBC que se encargue de crear una nueva fila en la base de datos para la nueva instancia creada y de almacenar sus datos en ella.

  Además, el método `ejbCreate()` siempre debe devolver la clave primaria (`String`) del nuevo Entity Bean que se ha creado. La clave primaria de nuestro *Bean Account* es el String `id`.

  Esto es completamente diferente a lo que ocurría con los Entity Bean con Container Managed Persistence. En éstos el método `ejbCreate()` no se tenía que encargar de crear una nueva fila en la base de datos (lo hacía el contenedor) y no hacía falta que devolviésemos la clave primaria (devolvíamos null o void).

- Todo Entity Bean con Bean Managed Persistence debe implementar los métodos de busqueda (`findXXX()`) del interfaz Home.

  La clase `AccountEJB` debe proporcionar un método `ejbFindXXX()` por cada método `findXXX()` del interfaz Home.

  Cada método `findXXX()` del interfaz Home debe tener un método `ejbFindXXX()` en la clase EJB con sus mismos argumentos que implemente la operación de búsqueda.

  Cada vez que un cliente invoque un método `findXXX()` del interfaz Home, el contenedor localizará en la clase EJB el método `ejbFindXXX()` con el que se empareje y lo invocará.

- Los métodos `ejbFindXXX()` deben devolver la clave primaria del bean buscado, o una colección de claves primarias en el caso de que la busqueda localice más de una instancia. Esto quiere decir que el código que implementa los métodos `ejbFindXXX()` debe acceder a la base de datos, localizar los Entity Beans buscados y devolver sus claves primarias. No es necesario que este código recupere los datos de la base de datos, sólo es necesario que compruebe que existen y en su caso, que devuelva las claves primarias.

  El contenedor tomará las claves primarias devueltas por estos métodos, instanciará los Entity Beans correspondientes y devolverá sus referencias al cliente que invocó el método de búsqueda.

- En la clase `AccountEJB` tenemos, por ejemplo, el método `ejbFindByPrimaryKey()`. Este método es un método obligatorio que debe ser implementado por todos los Entity Beans con Bean Managed Persistence.

  El método `ejbFindByPrimaryKey()` implementa el método `findByPrimaryKey()` del interfaz `AccountHome` y su parámetro y valor de retorno es siempre el mismo: la clase primaria.

- Todos los Entity Beans con Bean Managed Persistence implementan, al igual que cualquier Entity Bean, el interfaz `javax.ejb.EntityBean`.

  El significado de los métodos de este interfaz es básicamente el mismo que el que vimos cuando explicamos los Entity Bean con Container Managed Persistence. Sólamente tres de estos métodos varían ligeramente su significado para los Entity Bean con Bean Managed Persistence:

  - **`ejbLoad()`**: Este método será invocado por el contenedor cada vez que sea necesario recuperar el estado del Entity Bean de la base de datos. Este método debe tener implementado el código JDBC encargado de acceder a la base de datos y recuperar la información del Entity Bean.

    La diferencia con los beans Container Managed Persistence radica en que en éstos el contenedor se encargaba de recuperar los datos de la base de datos y una vez que los había recuperado, invocaba este método para informar al bean de que su estado había sido recuperado. Este método, por tanto, no tenía código de acceso a bases de datos.

    Con Bean Managed Persistence, el contenedor no accede a la base de datos sino que invoca a este método para que se realice dicho acceso.

  - **`ejbStore()`**: Este método será invocado por el contenedor cada vez que sea necesario salvar el estado del Entity bean en la base de datos. Este método debe tener implementado el código JDBC encargado de almacenar la información en la base de datos.

    La diferencia con los Entity Bean con Container Managed Persistence es similar a la que se da con el método `ejbLoad()`.

  - **`ejbRemove()`**: Este método será invocado por el contenedor siempre que el Entity Bean vaya a ser destruido. Este método debe tener implementado el código JDBC que se encargue de eliminar la información del Entity Bean de la base de datos (eliminará una fila de una tabla de la base de datos).

- Al final de la clase `AccountEJB` aparecen una serie de rutinas de acceso a bases de datos que pueden ser reutilizadas en la implementación de Entity Beans con Bean Managed Persistence. La reutilización de estas rutinas facilitará enormemente la implementación de este tipo de Entity Beans.

Como se puede apreciar, la diferencia fundamental entre los Entity Bean con CMP y los Entity Bean con BMP es que en estos últimos el desarrollador se tiene que encargar de implementar todos los accesos a la base de datos mediante código JDBC, siendo el contenedor responsable únicamente de invocar los métodos del Entity Bean que ejecuten el código JDBC. Mientras que en los Entity Bean con CMP el contenedor se encarga de realizar todos los accesos a la base de datos y el desarrollador no tiene que preocuparse de escribir ni una sola línea de código JDBC.

4.6.- CLIENTES EJB

Un cliente EJB es cualquier entidad software que acceda a los servicios de un componente EJB.

Una vez que conocemos como desarrollar componentes EJB, es el momento de estudiar como estos clientes pueden hacer uso de los servicios que los componentes EJB proporcionan.

Un cliente que quiera acceder a los servicios de un componente EJB debe seguir los siguientes pasos:

1. Obtener una referencia al interfaz Home del componente a traves del servicio JNDI.
2. Crear o localizar instancias del componente EJB.
3. Invocar los métodos de negocio de esas instancias.

4.6.1.- Obtener una referencia al Interfaz Home a traves de JNDI

En el momento en que un EJB es desplegado sobre un contenedor, el contenedor se encarga de registrar el interfaz Home del EJB en el servicio de nombres y directorio (JNDI) del servidor bajo un determinado nombre.

El cliente EJB si conoce el nombre con el que el interfaz Home ha sido registrado, podrá acceder al servicio JNDI y obtener una referencia a este interfaz.

Supongamos que el EJB *Product* que tomamos como ejemplo en apartados anteriores, ha sido registrado en el servicio JNDI con el nombre "MyProduct". El código del cliente que obtendrá una referencia al interfaz Home de este EJB será el siguiente:

```
Properties env = new Properties();
env.put("java.naming.factory.initial", "com.sun.jndi.cosnaming.CNCtxFactory");
env.put("java.naming.provider.url", "iiop://localhost:1050");

InitialContext initial = new InitialContext(env);
Object objref = initial.lookup("MyProduct");
ProductHome home =
     (ProductHome)PortableRemoteObject.narrow(objref,ProductHome.class);
```

Figura 4.21: Código que obtiene una referencia al interfaz home mediante JNDI

Este fragmento de código se interpreta de la siguiente forma:

- El cliente configura una serie de propiedades de entorno vinculadas a JNDI. Estas propiedades serán necesarias para crear posteriormente el contexto JNDI. Concretamente las propiedades a configurar son dos:
  - La propiedad *java.naming.factory.initial* que permite indicar la clase factoría a usar para la creación del contexto JNDI. En este caso usaremos una clase factoría que está disponible dentro de las librerías base de Java 2 como es *com.sun.jndi.cosnaming.CNCtxFactory*.
  - La propiedad *java.naming.provider.url* que permite indicar la ubicación del servicio de JNDI al que se quiere acceder. En este caso se indica mediante la URL *iiop://localhost:1050* (el servicio JNDI se encuentra por tanto en la máquina local, en el puerto 1050 y el protocolo de acceso a usar es IIOP).
- El cliente, en base a estas propiedades de entorno, crea un contexto JNDI que le permita acceder al servicio de JNDI del servidor de EJBs.
- Haciendo uso del método `lookup()` que le proporciona este contexto y del nombre con el que está registrado el EJB, el cliente obtiene una referencia al interfaz Home del componente *Product*.
- Finalmente, se convierte la referencia de tipo `Object` que hemos obtenido al tipo del interfaz Home de nuestro componente. En este caso, el tipo `ProductHome`.

4.6.2.- Crear o localizar instancias del componente EJB

Una vez que tenemos una referencia al interfaz Home del EJB, podremos usar los métodos que este interfaz nos proporciona para crear nuevas instancias (`create()`) o para localizar instancias ya existentes (`findXXX()`).

Siguiendo con el ejemplo del EJB *Product*, usaremos la referencia al interfaz `ProductHome` que obtuvimos para:

- Crear un nuevo producto con las siguientes características:
  - Identificativo: “456”
  - Descripción: “Wooden Duck”
  - Precio: 13.00
- Localizar el producto que tenga como identificativo (clave primaria) el código “876”.

Este es el código:

```
Product producto = home.create("456", "Wooden Duck", 13.00);
Product producto2 = home.findByPrimaryKey("876");
```

Figura 4.22: Código que crea y localiza instancias de un EJB

Como se puede observar, cada vez que el cliente crea o localiza una instancia de un componente EJB, éste obtiene una referencia al interfaz Remoto del componente. En este caso, al interfaz `Product`.

4.6.3.- Invocar los métodos de negocio de esas instancias

Una vez que el cliente tiene una referencia al interfaz Remoto de un componente, ya se encuentra en disposición de poder hacer uso de los servicios del componente.

En nuestro ejemplo, invocaremos los métodos de negocio de las dos instancias que obtuvimos para modificar y consultar sus datos:

```
producto.setPrice(14.00):
double price = producto2.getPrice();
```

Figura 4.23: Código que invoca los métodos de negocio de dos EJBs

Por último, se muestra a modo de ejemplo el código integro de dos clientes: el cliente del EJB *Product* y el cliente del EJB *Converter*.

4.6.4.- Cliente del EJB Product

```
package productExample;

import java.util.*;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class ProductClient {

   public static void main(String[] args) {

       try {
           Properties env = new Properties();
           env.put("java.naming.factory.initial",
                                "com.sun.jndi.cosnaming.CNCtxFactory");
           env.put("java.naming.provider.url", "iiop://localhost:1050");

           InitialContext initial = new InitialContext(env);
           Object objref = initial.lookup("MyProduct");

           ProductHome home =
               (ProductHome)PortableRemoteObject.narrow(objref,
                                          ProductHome.class);

           Product duke = home.create("123", "Ceramic Dog", 10.00);
           System.out.println(duke.getDescription() + ": " + duke.getPrice());
           duke.setPrice(14.00);
           System.out.println(duke.getDescription() + ": " + duke.getPrice());

           duke = home.create("456", "Wooden Duck", 13.00);
           duke = home.create("999", "Ivory Cat", 19.00);
           duke = home.create("789", "Ivory Cat", 33.00);
           duke = home.create("876", "Chrome Fish", 22.00);

           Product earl = home.findByPrimaryKey("876");
```

```
            System.out.println(earl.getDescription() + ": " + earl.getPrice());

            Collection c = home.findByDescription("Ivory Cat");
            Iterator i = c.iterator();

            while (i.hasNext()) {
               Product product = (Product)i.next();
               String productId = (String)product.getPrimaryKey();
               String description = product.getDescription();
               double price = product.getPrice();
               System.out.println(productId + ": " + description + " " + price);
            }

            c = home.findInRange(10.00, 20.00);
            i = c.iterator();

            while (i.hasNext()) {
               Product product = (Product)i.next();
               String productId = (String)product.getPrimaryKey();
               double price = product.getPrice();
               System.out.println(productId + ": " + price);
            }

        } catch (Exception ex) {
            System.err.println("Caught an exception." );
            ex.printStackTrace();
        }
    }
}
```

Figura 4.24: Código del cliente ProductClient

4.6.5.- Cliente del EJB Converter

```
package converterExample;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

import Converter;
import ConverterHome;

public class ConverterClient {

   public static void main(String[] args) {
        try {
            Properties env = new Properties();
            env.put("java.naming.factory.initial",
                                "com.sun.jndi.cosnaming.CNCtxFactory");
            env.put("java.naming.provider.url", "iiop://localhost:1050");

            InitialContext initial = new InitialContext(env);
            Object objref = initial.lookup("MyConverter");

            ConverterHome home =
                (ConverterHome)PortableRemoteObject.narrow(objref,
                                            ConverterHome.class);
```

```
            Converter currencyConverter = home.create();
            double amount = currencyConverter.dollarToYen(100.00);
            System.out.println(String.valueOf(amount));
            amount = currencyConverter.yenToEuro(100.00);
            System.out.println(String.valueOf(amount));

        } catch (Exception ex) {
            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
        }
    }
}
```

Figura 4.25: Código del cliente ConverterClient