



especialista universitario en

Tecnologías Java Enterprise
para Aplicaciones Web

Enterprise JavaBeans

ENTERPRISE JAVABEANS

1

TEMA 1. INTRODUCCIÓN A LA TECNOLOGÍA ENTERPRISE JAVABEANS

1.1. ARQUITECTURAS DE APLICACIONES DE EMPRESA	7
1.1.1. ARQUITECTURA DE DOS CAPAS	8
1.1.2. ARQUITECTURA DE TRES CAPAS	9
1.1.3 ARQUITECTURA DE TRES CAPAS CON COMPONENTES DISTRIBUIDOS	10
1.1.4 ARQUITECTURA INICIAL DE APLICACIONES WEB	10
1.1.5 ARQUITECTURA DE APLICACIONES J2EE	11
1.2. LA ARQUITECTURA ENTERPRISE JAVABEANS	13
1.2.1 ESPECIFICACIONES DE LA ARQUITECTURA ENTERPRISE JAVABEANS	13
1.2.2 ROLES EN LA ARQUITECTURA ENTERPRISE JAVABEANS	15
1.2.3. CONTENDOR DE BEANS	15
1.2.4. COMPONENTES ENTERPRISE BEANS	17
1.2.6. VENTAJAS DE LA ARQUITECTURA ENTERPRISE JAVABEANS	18

TEMA 2. INTRODUCCIÓN A LOS ENTERPRISE BEANS

21

2.1 TIPOS DE ENTERPRISE BEANS	22
2.1.1 BEANS DE SESIÓN	22
2.1.2 BEANS DE ENTIDAD	24
2.1.3 BEANS DIRIGIDOS POR MENSAJES	27
2.2 IMPLEMENTACIÓN DE UN ENTERPRISE BEAN	28
INTERFAZ REMOTA O LOCAL	30
INTERFAZ HOME	31
CLASE ENTERPRISE BEAN	31
FICHERO XML DESCRIPTOR DEL DESPLIEGUE	32
FICHERO EJB JAR QUE EMPAQUETA EL BEAN	32
2.3 APLICACIONES CLIENTES	33
2.3.1 CLIENTE JAVA	33
2.3.2 CLIENTES WEB: SERVLET Y JSP	34
2.4 ACCESO REMOTO Y LOCAL A LOS BEANS	37
2.4.1. ACCESO REMOTO	37
2.4.2 ACCESO LOCAL	37
2.4.3. INTERFACES LOCALES Y RELACIONES GESTIONADAS POR EL CONTENEDOR	38

TEMA 3: EJBS DE SESIÓN CON ESTADO

39

3.1 LA CLASE BEAN DE SESIÓN (SESSIONBEAN)	40
3.1.1 LAS VARIABLES DE INSTANCIA	41
3.1.2 LA INTERFAZ SESSIONBEAN	41
3.1.3 LOS MÉTODOS EJBCreate	41
3.1.4 MÉTODOS DE NEGOCIO	42
3.2 LA INTERFAZ HOME	44
3.3 INTERFAZ REMOTE	45
3.4 EL FICHERO DESCRIPTOR DEL DESPLIEGUE	45

3.5 CLASES DE APOYO	46
3.6 EL FICHERO EJB JAR	46
3.7 LA APLICACIÓN CLIENTE	47
3.8 OTRAS CARACTERÍSTICAS DE LOS ENTERPRISE BEANS	48
3.8.1 CÓMO ACCEDER A ENTRADAS DEL ENTORNO	48
3.8.2 CÓMO COMPARAR ENTERPRISE BEANS	49
3.8.3 CÓMO DEVOLVER UNA REFERENCIA A UN ENTERPRISE BEAN	50

TEMA 4: EJBS DE ENTIDAD CON PERSISTENCIA GESTIONADA POR EL BEAN (BMP)

51

4.1 CLASE ENTITYBEAN	52
4.1.1 LA INTERFAZ ENTITYBEAN	53
4.1.2 EL MÉTODO EJBCREATE	53
4.1.3 EL MÉTODO EJBPOSTCREATE	55
4.1.4 EL MÉTODO EJBREMOVE	55
4.1.5 LOS MÉTODOS EJBLOAD Y EJBSTORE	56
4.1.6 LOS MÉTODOS DE BÚSQUEDA	57
4.1.7 LOS MÉTODOS DE NEGOCIO	58
4.1.8 LOS MÉTODOS HOME	59
4.1.9 LLAMADAS A LA BASE DE DATOS	61
4.2 INTERFAZ HOME	62
4.2.1 DEFINICIÓN DE MÉTODOS CREATE	63
4.2.2 DEFINICIÓN DE LOS MÉTODOS DE BÚSQUEDA	63
4.2.3 DEFINICIONES DE MÉTODOS HOME	64
4.3 INTERFAZ REMOTA	64
4.4 EL DESCRIPTOR DE DESPLIEGUE	65

TEMA 5: BEANS DE ENTIDAD CON PERSISTENCIA GESTIONADA POR EL CONTENEDOR

67

5.1 INTRODUCCIÓN	67
5.1.1 EL MODELO ABSTRACTO DE PROGRAMACIÓN	67
5.1.2 EL ESQUEMA ABSTRACTO DE PERSISTENCIA	68
5.1.3 HERRAMIENTAS DEL CONTENEDOR Y PERSISTENCIA	68
5.2 EL EJB CUSTOMER	69
5.2.2 LA CLASE CUSTOMERBEAN	69
5.2.3 LA INTERFAZ REMOTA	72
5.2.4 LA INTERFAZ HOME REMOTA	73
5.2.5 EL DESCRIPTOR DE DESPLIEGUE XML	74
5.2.6 EL FICHERO WEBLOGIC-EJB-JAR.XML	76
5.2.7 EL FICHERO WEBLOGIC-CMP-RBMS-JAR.XML	77
5.2.7 EL FICHERO EJB JAR	78
5.2.8 LA APLICACIÓN CLIENTE	78
5.3 CAMPOS DE PERSISTENCIA	79

TEMA 6: RELACIONES ENTRE BEANS DE ENTIDAD

80

6.1 DEFINICIÓN DEL BEAN DEPENDIENTE	81
6.2. DEFINICIÓN DE LA RELACIÓN EN EL BEAN QUE REALIZA LA REFERENCIA	84
6.2.1. CAMPO DE RELACIÓN EN LA CLASE BEAN	84
6.2.2 DEFINICIÓN DE LA RELACIÓN EN LA TABLA DE LA BASE DE DATOS	84
6.3 ACTUALIZACIÓN DE LA RELACIÓN DESDE LOS CLIENTES	85
6.3.1 DEFINICIÓN/MODIFICACIÓN DE UN OBJETO RELACIONADO	86
6.3.2. OBTENCIÓN DE LA INFORMACIÓN DEL OBJETO RELACIONADO	87
6.4 EL MODELO ABSTRACTO DE PROGRAMACIÓN	91
6.5 EL ESQUEMA ABSTRACTO DE PERSISTENCIA	91
6.6 MODELADO DE LA BASE DE DATOS	95
6.7 RELACIÓN UNO-A-UNO UNIDIRECCIONAL	95
6.7.1 ESQUEMA DE BASE DE DATOS RELACIONAL	95
6.7.2 MODELO ABSTRACTO DE PROGRAMACIÓN	95
6.7.3 EL ESQUEMA ABSTRACTO DE PERSISTENCIA	97
6.8 RELACIÓN UNO-A-UNO BIDIRECCIONAL	97
6.8.1 ESQUEMA DE BASE DE DATOS RELACIONAL	98
6.8.2 MODELO ABSTRACTO DE PROGRAMACIÓN	98
6.8.3 ESQUEMA ABSTRACTO DE PERSISTENCIA	100
6.9 RELACIÓN UNO-A-MUCHOS UNIDIRECCIONAL	100
6.9.1 ESQUEMA DE BASE DE DATOS RELACIONAL	101
6.9.2 MODELO ABSTRACTO DE PROGRAMACIÓN	102
6.9.3 ES ESQUEMA ABSTRACTO DE PERSISTENCIA	105

TEMA 7: GESTIÓN DE TRANSACCIONES **107**

7.1 INTRODUCCIÓN	107
7.1.1 TRANSACCIONES	107
7.1.2 UN EJEMPLO CON EJBs	109
7.1.3 GESTIÓN DECLARATIVA DE LAS TRANSACCIONES	110
7.2 ALCANCE DE LA TRANSACCIÓN	111
7.3 ATRIBUTOS DE TRANSACCIÓN	112
7.3.1 DESCRIPTOR DEL DESPLIEGUE	112
7.3.2 DEFINICIÓN DE LOS ATRIBUTOS DE TRANSACCIÓN	113
7.4 PROPAGACIÓN DE LA TRANSACCIÓN	114
7.5 RELACIONES BASADAS EN COLECCIONES Y TRANSACCIONES	116

TEMA 8: SEGURIDAD **117**

8.1 INTRODUCCIÓN A LA SEGURIDAD EN EJBs	117
8.2 CONTROL DE ACCESO BASADO EN ROLES	119
8.3 MÉTODOS NO-CHEQUEADOS	121
8.4 LA IDENTIDAD DE SEGURIDAD RUNAs	121

TEMA 9: BUENAS PRÁCTICAS CON EJBs **124**

9.1 DISEÑO	124
9.1.1 ASEGURARSE DE QUE LOS EJBs SON NECESARIOS	124
9.1.2 USAR ARQUITECTURAS DE DISEÑO ESTÁNDAR	124

9.1.3 USAR BEANS DE ENTIDAD CMP	126
9.1.4 USAR PATRONES DE DISEÑO	126
SESSION FAÇADE	126
VALUE OBJECTS	127
9.2 IMPLEMENTACIÓN	130
9.2.1 USAR INTERFACES LOCALES PARA LOS BEANS DE ENTIDAD	130
9.2.3 MANEJAR LAS EXCEPCIONES CORRECTAMENTE	131
REMOTEEXCEPTION	131
EJBEXCEPTION Y SUS SUBCLASES	132
EXCEPCIONES DEFINIDAS EN LA APLICACIÓN	132
9.2.4 CACHEAR LOS OBJETOS JNDI LOOKUP	132
9.2.5 USAR BUSINESS DELEGATES COMO CLIENTES	133
9.3 DESPLIEGUE	134
9.3.1 CREAR UN ENTORNO DE CONSTRUCCIÓN	134
9.3.2 ESCRIBIR CÓDIGO DE PRUEBA	135

EJERCICIOS **136**

EJERCICIO 1.1: INSTALACIÓN DE WEBLOGIC 7.0	136
CREAR EL DOMINIO VACÍO <i>MODULOEJB</i>	136
ARRANCAR EL SERVIDOR DE APLICACIONES	140
COMPROBAR QUE EL SERVIDOR ESTÁ EN MARCHA	140
EJERCICIO 1.2: COMPILAR Y DESPLEGAR EL BEAN DE SESIÓN	142
CHEQUEAR LAS VARIABLES DE ENTORNO	143
OBTENER EL CÓDIGO FUENTE CON LOS EJEMPLOS	143
EMPAQUETAR EL BEAN	144
DESPLEGAR EL BEAN DE SESIÓN USANDO EL WEBLOGIC BUILDER	144
PROBAR LA APLICACIÓN CLIENTE	148
DESPLEGAR EL BEAN USANDO LA CONSOLA DE ADMINISTRACIÓN	148
EJERCICIO 3: MODIFICAR EL CÓDIGO FUENTE DEL BEAN	151
EJERCICIO 2.1: COMPILACIÓN, DESPLIEGUE Y PRUEBA DE UNA APLICACIÓN WEB QUE USA UN BEAN DE SESIÓN SIN ESTADO	152
PASOS PREVIOS	152
COMPILACIÓN Y CREACIÓN DEL FICHERO JAR CON EL BEAN	152
DESPLEGAR LA APLICACIÓN	154
EJERCICIO 2.2: CAMBIAR ALGUNOS DESCRIPTORES DEL DESPLIEGUE	155
URLS DE LA APLICACIÓN	155
REFERENCIAS A BEANS EN LA APLICACIÓN WEB	155
EJERCICIO 3: COMPILACIÓN Y PRUEBA DE UN BEAN DE SESIÓN CON ESTADO	157
EJERCICIO 4.1: DESPLIEGUE Y PRUEBA DEL BEAN SAVINGSACCOUNT EJB	158
CREACIÓN DEL DOMINIO <i>EJEMPLOS</i>	158
CREACIÓN DE LA TABLA <i>SAVINGSACCOUNT</i>	159
DESPLIEGUE DEL BEAN	160
COMPILACIÓN Y PRUEBA DE LA APLICACIÓN CLIENTE	160
EJERCICIO 4.2: MODIFICACIÓN DEL BEAN	161
EJERCICIO 5.1: DESPLIEGUE Y PRUEBA DEL BEAN CUSTOMER EJB	162
CREACIÓN DE LA TABLA <i>CUSTOMER</i>	162
DESPLIEGUE DEL BEAN	163
COMPILACIÓN Y PRUEBA DE LA APLICACIÓN CLIENTE	163

EJERCICIO 5.2: MODIFICACIÓN DEL BEAN	163
EJERCICIO 7: DESCARGAR Y PROBAR EL EJEMPLO7	165
EJERCICIO 7.1	166
EJERCICIO 8: SEGURIDAD Y CONTROL DE ACCESO	167
DESCARGAR Y PROBAR EL EJB SAVINGSACCOUNT	167
CREAR DE NUEVOS USUARIOS EN EL SERVIDOR WEBLOGIC	167
MODIFICACIÓN DEL BEAN SAVINGSACCOUNT	168
EJERCICIO 9: UN EJEMPLO COMPLETO CON EJBS	171
 BIBLIOGRAFÍA	 174

Tema 1. Introducción a la tecnología Enterprise JavaBeans

La tecnología Enterprise JavaBeans es la propuesta del mundo Java (Sun y un amplio grupo de empresas colaboradoras entre las que no se encuentra, evidentemente, Microsoft) para el desarrollo de aplicaciones de empresa. Entendemos por aplicaciones de empresa las aplicaciones informáticas de gestión que se implantan en grandes corporaciones con múltiples oficinas distribuidas por toda la geografía nacional (o internacional). Suelen ser aplicaciones distribuidas, usadas por múltiples clientes, que hacen un uso intensivo de transacciones, bases de datos y políticas de seguridad.

La arquitectura Enterprise JavaBeans es el núcleo de un conjunto de tecnologías Java que recibe el nombre de J2EE (Java 2 Enterprise Edition) y hace uso intensivo de librerías incluidas en esta plataforma. Algunos ejemplos de APIs Java usados por la arquitectura EJB son los siguientes:

- JNDI para acceder a recursos
- JMS para tratar con mensajes
- JTA para programar explícitamente las transacciones

La tecnología Enterprise JavaBeans esta siendo apoyada por la mayor parte de las empresas de informática que se orientan al mundo de la empresa (IBM, ORACLE, SUN,...) y también por los clientes de estas empresas, que valoran de forma muy positiva, entre otros aspectos, el carácter abierto de la arquitectura.

En este tema veremos una introducción a las distintas arquitecturas que se han ido proponiendo para las aplicaciones de empresa, y situaremos en este contexto la propuesta de arquitectura Enterprise JavaBeans. Comentaremos las características principales de la propuesta, dando algunos ejemplos prácticos de componentes Enterprise JavaBeans (EJB). Como parte práctica, explicaremos cómo realizar un despliegue de componentes EJB en el servidor de aplicaciones Weblogic.

1.1. Arquitecturas de aplicaciones de empresa

Las aplicaciones de empresa han sufrido una transformación extensa. La primera generación de aplicaciones de empresa consistían en aplicaciones centralizadas usando computadores mainframes. A finales de 1980 y comienzos de 1990, la mayoría de las nuevas aplicaciones de empresa se construyeron siguiendo una arquitectura de dos capas (también conocida como arquitectura cliente/servidor). Después, las aplicaciones de empresa evolucionaron a arquitecturas de tres capas, después a arquitecturas basadas en Web. El estado de evolución actual está representado por la arquitectura de aplicaciones J2EE.

El párrafo anterior se aplica a países con un alto grado de desarrollo tecnológico (léase Estados Unidos, y poco más). En países como España nos encontramos en la mayoría de los casos todavía usando aplicaciones cliente/servidor, y comenzando a implantar soluciones basadas en Web.

1.1.1. Arquitectura de dos capas

Con una aplicación de dos capas, un sistema de negocio se estructura como un conjunto de aplicaciones a nivel de sistema operativo que se ejecutan en la máquina cliente, típicamente un PC. Las aplicaciones se comunican a través de la red con el servidor de bases de datos, que almacena la información corporativa y que es el que implementa el manejo de transacciones y la seguridad. La comunicación entre las aplicaciones y el servidor de base de datos se realiza normalmente usando el lenguaje SQL.

Las aplicaciones cliente implementan uno o varios procesos de negocio, junto con la interfaz de usuario que proporciona la lógica de presentación que realiza la interacción entre el usuario y el proceso de negocio. El concepto de proceso de negocio encapsula una interacción del usuario con alguna información de la empresa.

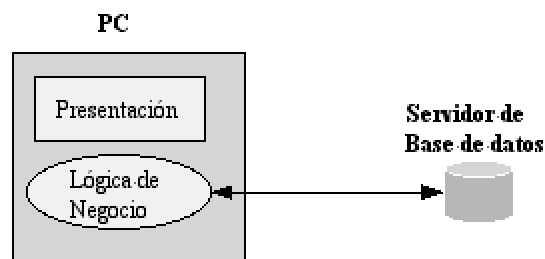


Figura 1: arquitectura de aplicaciones de dos capas.

Entre las ventajas de esta arquitectura se encuentra la facilidad de desarrollo debido a que la presentación y la lógica de negocio residen en la misma aplicación. Los inconvenientes son mucho más numerosos. Entre ellos destacan :

- Dificultad de administrar las aplicaciones en grandes empresas
- La integridad de la base de datos se puede comprometer fácilmente
- Dificultad de mantener el código
- Expone las aplicaciones a violaciones de seguridad
- Su escalabilidad es limitada, es difícil escalar a un alto número de usuarios
- Requiere una estructura homogénea de las máquinas cliente
- Liga una aplicación a un tipo particular de presentación
- Incompatibilidad con la Web (uno de los inconvenientes más importantes en la actualidad)

1.1.2. Arquitectura de tres capas

El avance fundamental de las arquitecturas de tres capas es la separación de la lógica de presentación de la lógica de negocio. Para ello se introduce una capa intermedia que se encarga específicamente de la lógica de negocio y de la gestión de transacciones y de seguridad. Se introducen monitores de procesamiento de transacciones (TP, transaction processing, monitors) como CICS o Tuxedo que dan soporte a esta capa intermedia. La mayoría de los sistemas ERP* (Enterprise Resource Planning) usan esta arquitectura.

*ERP (Enterprise resource planning) es un término de la industria para un amplio conjunto de actividades soportadas por software de aplicación que facilitan a un fabricante o a otro negocio manejar partes importantes de su negocio, incluyendo planificación de productos, compra de material, mantenimiento de inventario, interacción con suministradores, servicios al cliente y seguimiento de pedidos. Los ERP también incluyen módulos de aplicación para los aspectos de finanzas y de recursos humanos del negocio. Normalmente, un sistema ERP usa o está integrado con una base de datos relacional. El despliegue de sistemas ERP obliga normalmente a realizar un considerable análisis de los procesos de negocio, así como la formación de los empleados y nuevos procedimientos de trabajo. Entre las empresas que desarrollan y despliegan ERP se encuentran SAP o Peoplesoft. (definición cortesía de whatis.com).

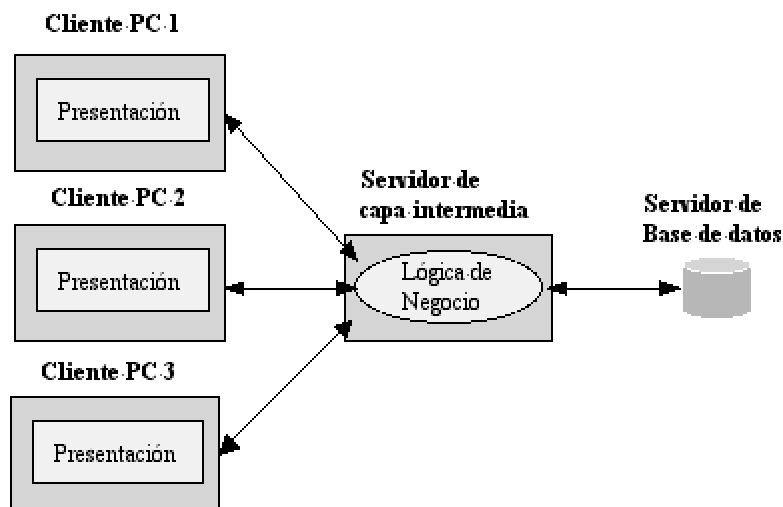


Figura 2: Arquitectura de aplicaciones de tres capas

Aunque esta arquitectura elimina algunos de los inconvenientes de la arquitectura de dos capas, todavía sufre de inconvenientes:

- Complejidad
- Ausencia de portabilidad de la aplicación
- Incompatibilidad entre fabricantes
- Falta de difusión y compartición de conocimientos y experiencias
- Incompatibilidad con la Web

1.1.3 Arquitectura de tres capas con componentes distribuidos

Los sistemas de componentes distribuidos representan un enfoque más moderno a la arquitectura de tres capas. Estos sistemas definen objetos o componentes que se ejecutan en la capa intermedia y que están disponibles para servir a otros procesos a través de proxies remotos. Estos proxies remotos comunican peticiones a los componentes distribuidos a lo largo de la red. Ejemplo de sistemas de componentes distribuidos son RMI, CORBA y DCOM.

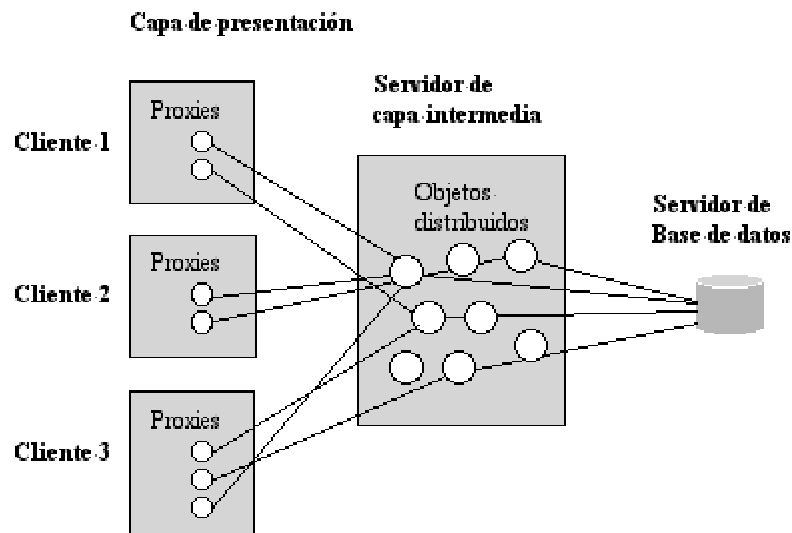


Figura 3: Arquitectura de tres capas con componentes distribuidos

Los componentes distribuidos son más reusables y flexibles que las aplicaciones basadas en procedimientos usadas en los monitores TP tradicionales, debido a que pueden ser ensamblados en una gran variedad de combinaciones para el desarrollo de aplicaciones de negocio, pero pueden ser complicados de programar y carecen de la infraestructura robusta que ofrecen los monitores TP.

1.1.4 Arquitectura inicial de Aplicaciones Web

La introducción de la Web lo cambió todo. El modelo simplificado de funcionamiento consiste en incorporar extensiones a los servidores Web. Estas extensiones invocan programas en el servidor que construyen dinámicamente documentos HTML a partir de la información almacenada en la base de datos corporativa. Además, estas extensiones en el servidor también introducen cambios en las bases de datos corporativas a partir de la información de formularios HTML.

Un ejemplo de estas extensiones son los scripts cgi-bin. Aunque estos mecanismos han permitido a los desarrolladores corporativos construir aplicaciones Web simples, el enfoque no escala bien para aplicaciones de empresa más complejas por las siguientes razones:

- Los scripts cgi-bin no proporcionan una buena encapsulación de los procesos o entidades de negocio.
- Los scripts cgi-bin son complicados de desarrollar, probar y mantener. Las herramientas de desarrollo de alto nivel no soportan bien estos scripts.
- Los scripts cgi-bin mezclan la lógica de negocio con la lógica de presentación.
- Los scripts cgi-bin no proporcionan soporte para el manejo de transacciones ni de seguridad.
- Los scripts cgi-bin no promueven el mantenimiento de la integridad de las reglas de negocio.

1.1.5 Arquitectura de aplicaciones J2EE

J2EE es una arquitectura estándar orientada específicamente hacia el desarrollo y el despliegue de aplicaciones de empresa orientadas a Web usando el lenguaje de programación Java. Las empresas y los vendedores independientes de software (ISV, en inglés) pueden usar la arquitectura J2EE tanto para el desarrollo y despliegue de aplicaciones intranet, sustituyendo de esta forma las aplicaciones de dos capas y de tres capas, y para el desarrollo de aplicaciones Web, sustituyendo el enfoque basado en cgi.

La siguiente figura ilustra el modelo de programación propuesto por J2EE para el desarrollo de aplicaciones cliente/servidor. La Máquina Virtual Java hace de Contenedor de Aplicaciones Cliente.

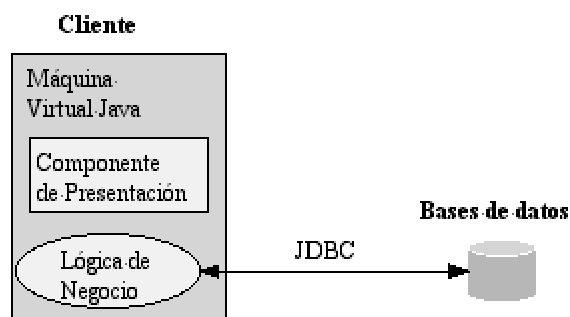


Figura 4: Modelo de programación cliente/servidor de las aplicaciones J2EE

La siguiente figura ilustra el modelo de programación propuesto por J2EE para el desarrollo de aplicaciones de tres capas.

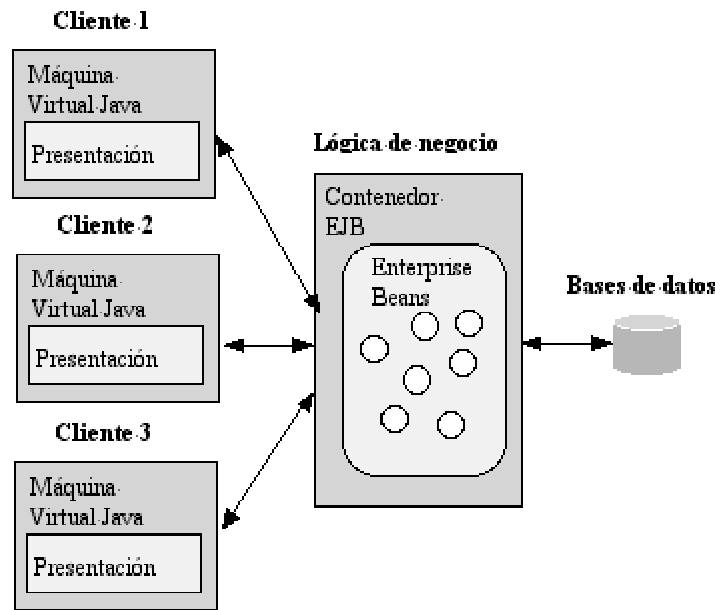


Figura 5: Modelo de programación de tres capas de las aplicaciones J2EE

La siguiente figura ilustra el modelo de programación para las aplicaciones basadas en Web:

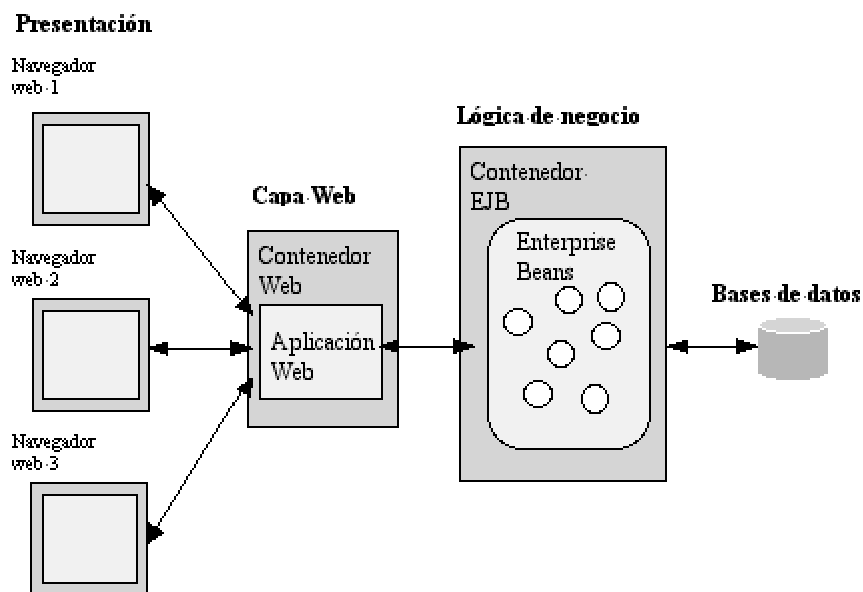


Figura 6: Modelo de programación de aplicaciones J2EE para aplicaciones basadas en Web.

La plataforma J2EE consiste en cuatro entornos de programación, llamados contenedores:

- **El contenedor EJB:** proporciona el entorno para el desarrollo, despliegue, y manejo en tiempo de ejecución de enterprise beans. Los enterprise beans son componentes que implementan los procesos y las entidades de negocio.
- **El contenedor Web:** proporciona el entorno para el desarrollo, despliegue y manejo en tiempo de ejecución de servlets y páginas JSP

(JavaServer Pages). Los servlets y las páginas JSP se agrupan en unidades desplegables llamadas aplicaciones Web. Una aplicación Web implementa la lógica de presentación de una aplicación de empresa.

- **El contenedor de aplicaciones cliente:** se trata de la máquina virtual Java. Proporciona el entorno para la ejecución de aplicaciones cliente J2EE.
- **El contenedor de applets:** proporciona el entorno para ejecutar applets Java. Este entorno está embebido típicamente en un navegador Web.

En módulos anteriores del curso hemos estudiado las partes de J2EE que configuran una aplicación Web. Vamos a centrarnos en este módulo en el estudio de los componentes enterprise beans y de los contenedores EJB.

1.2. La arquitectura Enterprise JavaBeans

La arquitectura de componentes Enterprise JavaBeans (arquitectura EJB) es el núcleo de la plataforma J2EE. Esta arquitectura de componentes combina lo mejor de los monitores de procesamiento de transacciones y de los componentes distribuidos, proporcionando un entorno al estilo de los monitores de transacciones orientado a componentes distribuidos. La arquitectura EJB está constituida por un conjunto de componentes software llamados enterprise beans y un contenedor EJB que da soporte de ejecución a estos componentes.

En este apartado realizaremos una breve introducción a las características generales de la arquitectura.

1.2.1 Especificaciones de la arquitectura Enterprise JavaBeans

En Marzo de 1998 Sun Microsystems propone la especificación 1.0 de la arquitectura Enterprise JavaBeans. Esta especificación comienza con la siguiente definición:

La arquitectura Enterprise JavaBeans es una arquitectura de componentes para el desarrollo y despliegue de aplicaciones de empresa distribuidas y orientadas a objetos. Las aplicaciones escritas usando la arquitectura Enterprise JavaBeans son escalables, transaccionales y seguras para multi usuarios. Estas aplicaciones pueden escribirse una vez, y luego desplegarse en cualquier servidor que soporte la especificación Enterprise JavaBeans.

Aunque se han introducido nuevas versiones de la especificación, que incorporan muchas mejoras a la propuesta inicial, la definición de la arquitectura sigue siendo la misma. La siguiente tabla muestra las distintas revisiones que ha sufrido la especificación de la arquitectura EJB.

Especificación EJB	Fecha	Principales novedades
EJB 1.0	Marzo 1998	Propuesta inicial de la arquitectura EJB. Se introducen los beans de sesión y los de entidad (de implementación opcional). Persistencia manejada por el contenedor en los beans de entidad. Manejo de transacciones. Manejo de seguridad.
EJB 1.1	Diciembre 1999	Implementación obligatoria de los beans de entidad. Acceso al entorno de los beans mediante JNDI.
EJB 2.0	Agosto 2001	Manejo de mensajes con los beans dirigidos por mensajes. Relaciones entre beans manejadas por el contenedor. Uso de interfaces locales entre beans que se encuentran en el mismo servidor. Consultas de beans declarativas, usando el EJB QL.
EJB 2.1	Agosto 2002	Soporte para servicios web. Temporizador manejado por el contenedor de beans. Mejora en el EJB QL.

Tabla 1: Evolución de las especificaciones de la arquitectura Enterprise JavaBeans

Entre los objetivos que se enumeraban en ese primer documento de especificación 1.0 se encuentran los siguientes:

- Definir una arquitectura de componentes estándar con la que construir aplicaciones de gestión (business applications) distribuidas y orientadas a objetos en el lenguaje de programación Java. Enterprise JavaBeans permitirá construir aplicaciones combinando componentes desarrollados por herramientas de distintas compañías.
- La arquitectura Enterprise JavaBeans hará fácil la construcción de aplicaciones. Los desarrolladores de aplicaciones no tendrán que

entender los detalles de bajo nivel referidos al manejo de transacciones, manejo de estado, multi threading, pooling de recursos y otras APIs complejas de bajo nivel.

- Las aplicaciones Enterprise JavaBeans seguirán la filosofía “escribe una vez, ejecuta en cualquier sitio” del lenguaje de programación Java. Un enterprise bean puede desarrollarse una vez, y después desplegado en múltiples plataformas sin necesidad de recompilarlo o modificar su código fuente.

1.2.2 Roles en la arquitectura Enterprise JavaBeans

La arquitectura EJB define seis papeles principales. Brevemente, son:

- **Desarrollador de beans:** desarrolla los componentes enterprise beans.
- **Ensamblador de aplicaciones:** compone los enterprise beans y las aplicaciones cliente para conformar una aplicación completa
- **Desplegador:** despliega la aplicación en un entorno operacional particular (servidor de aplicaciones)
- **Administrador del sistema:** configura y administra la infraestructura de computación y de red del negocio
- **Proporcionador del Contenedor EJB y Proporcionador del Servidor EJB:** un fabricante (o fabricantes) especializado en manejo de transacciones y de aplicaciones y otros servicios de bajo nivel. Desarrollan el servidor de aplicaciones.

1.2.3. Contenedor de beans

Los enterprise beans (que también se denominan beans) son componentes software que se ejecutan en un entorno especial denominado contenedor EJB. El contenedor hospeda y maneja un enterprise bean de la misma forma que un servidor web Java hospeda un servlet o un navegador hospeda un applet. Un enterprise bean no puede funcionar fuera de un contenedor EJB. El contenedor EJB maneja cualquier aspecto del bean en tiempo de ejecución, incluyendo acceso remoto al bean, seguridad, persistencia, transacciones, concurrencia y acceso y pooling de recursos. El contenedor EJB también suele proporcionar servicios relacionados con la escalabilidad de la aplicación, como son la definición de clusters de contenedores, el balanceo de carga o la tolerancia a fallos.

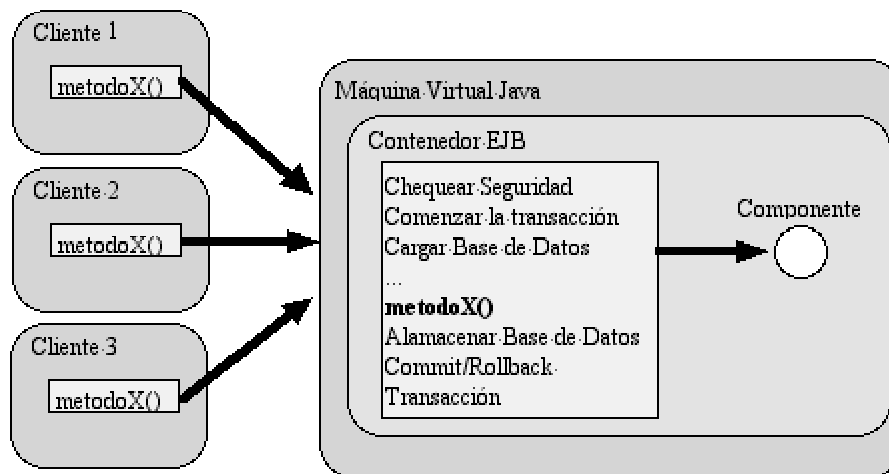


Figura 7: El contenedor EJB captura la petición del cliente y realiza servicios de bajo nivel antes y después de pedir al bean que ejecute la lógica de negocio de la petición.

El contenedor aísla el enterprise bean del acceso directo de las aplicaciones cliente. Cuando una aplicación cliente invoca un método remoto en un enterprise bean, el contenedor intercepta la invocación para asegurar que la persistencia, transacciones y seguridad se aplican correctamente. De esta forma, el desarrollador de beans puede concentrarse en encapsular correctamente la lógica de negocio, mientras que el contenedor se encarga de todo lo demás.

El contenedor EJB se ejecuta a su vez en una máquina virtual Java, con lo que tiene acceso a toda la infraestructura proporcionada por este lenguaje de programación.

Los contenedores manejan muchos beans simultáneamente, en la misma forma que un servidor web Java maneja muchos servlets. Para reducir el consumo de memoria y el procesamiento, los contenedores almacenan (pool) los recursos y los ciclos de vida de los beans de forma muy cuidadosa. Cuando un bean no está siendo usado, el contenedor lo colocará en un almacén para ser reusado por otro cliente, o lo eliminará de la memoria y sólo lo recuperará cuando sea necesario. Debido a que las aplicaciones clientes no tienen acceso a los beans, la aplicación cliente desconoce completamente las actividades de manejo de recursos del bean. Por ejemplo, un bean que no está siendo usado puede ser eliminado de la memoria, mientras que su referencia remota en el cliente permanece intacta. Cuando el cliente invoca un método sobre la referencia remota, el contenedor simplemente recupera el bean para dar servicio a la petición. La aplicación cliente no se da cuenta de todo este proceso.

Un enterprise bean depende del contenedor para todo lo que necesita. Si un enterprise bean tiene que acceder a una conexión JDBC o a otro enterprise bean, lo hace a través del contenedor; si un enterprise bean tiene que acceder a la identidad de su llamador, obtener una referencia a él mismo, o acceder a propiedades, lo hace a través del contenedor.

En las próximas sesiones veremos cómo se implementa todo este funcionamiento. Veremos cómo el contenedor EJB intercepta las llamadas y cómo los beans pueden acceder a los servicios que proporciona el contenedor.

1.2.4. Componentes enterprise beans

Los componentes enterprise bean encapsulan típicamente un proceso o una entidad de negocio. Un enterprise bean, por ejemplo, podría calcular los pagos de interés de un préstamo, o encapsular la información sobre una cuenta bancaria que se encuentra físicamente en una base de datos relacional. Un cliente que necesita información llama a los métodos de negocio en el bean y esta llamada provoca una invocación remota que llega al contenedor EJB.

Hay tres tipos de enterprise beans: beans de sesión, beans de entidad y beans dirigidos por mensajes. En la siguiente sesión los presentaremos con más detalle. Por ahora, vamos a adelantar sólo algunas de sus características

En la especificación 2.0 de la arquitectura EJB se definen tres tipos de beans: beans de sesión, beans de entidad y beans dirigidos por mensaje:

- Los beans de sesión realizan una tarea a petición de un cliente, pero no se corresponden con ninguna entidad persistente de negocio. Pueden ser a su vez de dos tipos: con estado y sin estado. El estado lo almacenan localmente en la memoria, no en almacenamiento secundario, y dura el tiempo que está activa la sesión con el cliente.
- Los beans de entidad representan objetos persistentes de negocio, normalmente datos existentes en alguna o algunas bases de datos del negocio. Típicamente, un bean de entidad representa una fila de una tabla de una base de datos relacional.
- Los beans dirigidos por mensajes permiten procesar mensajes asíncronos generados por otros beans o por aplicaciones externas que necesitamos conectar al sistema.
- El cliente que usa los beans es una aplicación Java independiente, un servlet o una página JSP. En cualquier caso se trata de código Java que tiene acceso a las interfaces definidas para cada bean. También lo veremos con más detalle en la próxima sesión, pero adelantemos que existen dos tipos de acceso a los beans:
- Acceso remoto. El cliente usa RMI-IIOP para comunicarse con el bean. El desarrollador del bean define dos tipos de interfaces: la mencionada interfaz remota, en la que se definen la interfaz de los procesos de negocio, y la interfaz home, en la que se definen la interfaz de los métodos de gestión (creación, borrado, etc.) de las instancias del bean.
- Acceso local. Si el cliente se ejecuta en la misma máquina virtual Java que el bean, puede acceder a versiones locales de las interfaces. De esta forma no es necesario serializar los parámetros ni los valores devueltos por el bean, mejorándose la eficiencia de las llamadas.

1.2.6.Ventajas de la arquitectura Enterprise JavaBeans

La arquitectura EJB proporciona beneficios a todos los papeles que hemos mencionado previamente (desarrolladores, ensambladores de aplicaciones, administradores, desplegados, fabricantes de servidores). Vamos en enumerar las ventajas que obtendrán los desarrolladores de aplicaciones y los clientes finales.

Las ventajas que ofrece la arquitectura Enterprise JavaBeans a un desarrollador de aplicaciones se listan a continuación.

- **Simplicidad.** Debido a que el contenedor de aplicaciones libera al programador de realizar las tareas del nivel del sistema, la escritura de un enterprise bean es casi tan sencilla como la escritura de una clase Java. El desarrollador no tiene que preocuparse de temas de nivel de sistema como la seguridad, transacciones, multi-threading o la programación distribuida. Como resultado, el desarrollador de aplicaciones se concentra en la lógica de negocio y en el dominio específico de la aplicación.
- **Portabilidad de la aplicación.** Una aplicación EJB puede ser desplegada en cualquier servidor de aplicaciones que soporte J2EE.
- **Reusabilidad de componentes.** Una aplicación EJB está formada por componentes enterprise beans. Cada enterprise bean es un bloque de construcción reusable. Hay dos formas esenciales de reusar un enterprise bean a nivel de desarrollo y a nivel de aplicación cliente. Un bean desarrollado puede desplegarse en distintas aplicaciones, adaptando sus características a las necesidades de las mismas. También un bean desplegado puede ser usado por múltiples aplicaciones cliente.
- **Posibilidad de construcción de aplicaciones complejas.** La arquitectura EJB simplifica la construcción de aplicaciones complejas. Al estar basada en componentes y en un conjunto claro y bien establecido de interfaces, se facilita el desarrollo en equipo de la aplicación.
- **Separación de la lógica de presentación de la lógica de negocio.** Un enterprise bean encapsula típicamente un proceso o una entidad de negocio. (un objeto que representa datos del negocio), haciéndolo independiente de la lógica de presentación. El programador de gestión no necesita de preocuparse de cómo formatear la salida; será el programador que desarrolle la página Web el que se ocupe de ello usando los datos de salida que proporcionará el bean. Esta separación hace posible desarrollar distintas lógicas de presentación para la misma lógica de negocio o cambiar la lógica de presentación sin modificar el código que implementa el proceso de negocio.
- **Despliegue en muchos entornos operativos.** Entendemos por entornos operativos el conjunto de aplicaciones y sistemas (bases de datos, sistemas operativos, aplicaciones ya en marcha, etc.) que están instaladas en una empresa. Al detallarse claramente todas las posibilidades de despliegue de las aplicaciones, se facilita el desarrollo

de herramientas que asistan y automaticen este proceso. La arquitectura permite que los beans de entidad se conecten a distintos tipos de sistemas de bases de datos.

- **Despliegue distribuido.** La arquitectura EJB hace posible que las aplicaciones se desplieguen de forma distribuida entre distintos servidores de una red. El desarrollador de beans no necesita considerar la topología del despliegue. Escribe el mismo código independientemente de si el bean se va a desplegar en una máquina o en otra (cuidado: con la especificación 2.0 esto se modifica ligeramente, al introducirse la posibilidad de los interfaces locales).
- **Interoperabilidad entre aplicaciones.** La arquitectura EJB hace más fácil la integración de múltiples aplicaciones de diferentes vendedores. El interfaz del enterprise bean con el cliente sirve como un punto bien definido de integración entre aplicaciones.
- **Integración con sistemas no-Java.** Las APIs relacionadas, como las especificaciones Connector y Java Message Service (JMS), así como los beans manejados por mensajes, hacen posible la integración de los enterprise beans con sistemas no Java, como sistemas ERP o aplicaciones mainframes.
- **Recursos educativos y herramientas de desarrollo.** El hecho de que la especificación EJB sea un estándar hace que exista una creciente oferta de herramientas y formación que facilita el trabajo del desarrollador de aplicaciones EJB.
- Entre las ventajas que aporta esta arquitectura al cliente final, destacamos la posibilidad de elección del servidor, la mejora en la gestión de las aplicaciones, la integración con las aplicaciones y datos ya existentes y la seguridad.
- **Elección del servidor.** Debido a que las aplicaciones EJB pueden ser ejecutadas en cualquier servidor J2EE, el cliente no queda ligado a un vendedor de servidores. Antes de que existiera la arquitectura EJB era muy difícil que una aplicación desarrollada para una determinada capa intermedia (Tuxedo, por ejemplo) pudiera portarse a otro servidor. Con la arquitectura EJB, sin embargo, el cliente deja de estar atado a un vendedor y puede cambiar de servidor cuando sus necesidades de escalabilidad, integración, precio, seguridad, etc. lo requieran. Existen en el mercado algunos servidores de aplicaciones gratuitos (JBOSS, el servidor de aplicaciones de Sun, etc.) con los que sería posible hacer unas primeras pruebas del sistema, para después pasar a un servidor de aplicaciones con más funcionalidades.
- **Gestión de las aplicaciones.** Las aplicaciones son mucho más sencillas de manejar (arrancar, parar, configurar, etc.) debido a que existen herramientas de control más elaboradas.
- **Integración con aplicaciones y datos ya existentes.** La arquitectura EJB y otras APIs de J2EE simplifican y estandarizan la integración de aplicaciones EJB con aplicaciones no Java y sistemas en el entorno operativo del cliente. Por ejemplo, un cliente no tiene que cambiar un esquema de base de datos para encajar en una aplicación. En lugar de

ello, se puede construir una aplicación EJB que encaje en el esquema cuando sea desplegada.

- **Seguridad.** La arquitectura EJB traslada la mayor parte de la responsabilidad de la seguridad de una aplicación de el desarrollador de aplicaciones al vendedor del servidor, el Administrador de Sistemas y al Desplegador (papeles de la especificación EJB) La gente que ejecuta esos papeles están más cualificados que el desarrollador de aplicaciones para hacer segura la aplicación. Esto lleva a una mejor seguridad en las aplicaciones operacionales.

Tema 2. Introducción a los Enterprise Beans

Un enterprise bean es un componente (objeto) distribuido gestionado por un servidor, escrito en el lenguaje Java y que implementa la lógica de negocio de una aplicación. Por ejemplo, en una aplicación de control de inventario, un enterprise bean podría implementar la lógica de negocio mediante métodos llamados `checkInventoryLevel` y `orderProduct`. Mediante la invocación de estos productos, los clientes remotos podrían acceder a los servicios de inventario proporcionados por la aplicación.

Los enterprise beans simplifican el desarrollo de aplicaciones distribuidas de gran tamaño por varias razones:

- Debido a que el contenedor EJB proporciona servicios de nivel de sistema a los enterprise beans, el desarrollador de los beans puede concentrarse en la resolución de los problemas de negocio. El contenedor EJB se responsabiliza de servicios de nivel de sistema como gestión de transacciones o seguridad.
- Debido a que los beans, y no el cliente, contienen la lógica de negocio de la aplicación, el desarrollador de la aplicación cliente puede concentrarse en la presentación del cliente y no se debe preocupar de acceder a bases de datos ni de reglas de negocio. Como resultado, los clientes son más ligeros, un beneficio que es particularmente importante en clientes que corren en dispositivos pequeños (palm computers o teléfonos Java).
- Debido a que los enterprise beans son componentes portables, el ensamblador de aplicaciones puede construir nuevas aplicaciones a partir de los beans existentes. Estas aplicaciones pueden ejecutarse en cualquier servidor J2EE compatible.

Es recomendable usar enterprise beans cuando la aplicación que deseamos desarrollar cumple alguno de los siguientes requisitos:

- La aplicación debe escalable. Si se hace necesario mejorar la eficiencia de la aplicación para cubrir las necesidades de un número creciente de clientes, es muy útil poder distribuir la aplicación entre múltiples máquinas. La arquitectura EJB está especialmente indicada para esto, ya que los enterprise beans pueden moverse de una máquina a otra y replicarse, todo ello de forma totalmente transparente a los clientes finales.
- Se requieren transacciones para asegurar la integridad de los datos. Los enterprise beans soportan transacciones para gestionar el acceso concurrente a objetos compartidos.
- La aplicación tendrá una variedad de clientes. Con sólo unas pocas líneas de código, los clientes remotos pueden localizar fácilmente

enterprise beans en el contenedor. Estos clientes pueden ser ligeros, variados y numerosos.

Existen tres tipos de enterprise beans:

- **Beans de sesión:** ejecutan una tarea para un cliente.
- **Beans de entidad:** representan un objeto entidad de negocio que existe en un almacenamiento persistente.
- **Beans dirigidos por mensajes:** actúan como escuchadores (listeners) del API Java Message Service, procesando mensajes asíncronamente.

2.1 Tipos de enterprise beans

2.1.1 Beans de sesión

Los beans de sesión representan sesiones interactivas con uno o más clientes. Los bean de sesión pueden mantener un estado, pero sólo durante el tiempo que el cliente interactúa con el bean. Esto significa que los beans de sesión no almacenan sus datos en una base de datos después de que el cliente termine el proceso. Por ello se suele decir que los beans de sesión no son persistentes.

A diferencia de los bean de entidad, los beans de sesión no se comparten entre más de un cliente, sino que existe una correspondencia uno-uno entre beans de sesión y clientes. Por esto, el contenedor EJB no necesita implementar mecanismos de manejo de concurrencia en el acceso a estos beans.

Existen dos tipos de beans de sesión: con estado y sin él.

Beans de sesión sin estado

Los beans de sesión sin estado no se modifican con las llamadas de los clientes. Los métodos que ponen a disposición de las aplicaciones clientes son llamadas procedurales que reciben datos y devuelven resultados, pero que no modifican internamente el estado del bean. Esta propiedad permite que el contenedor EJB pueda crear un almacén (*pool*) de instancias, todas ellas del mismo bean de sesión sin estado y asignar cualquier instancia a cualquier cliente. Incluso un único bean puede estar asignado a múltiples clientes, ya que la asignación sólo dura el tiempo de invocación del método solicitado por el cliente.

Cuando un cliente invoca un método de un bean de sesión sin estado, el contenedor EJB obtiene una instancia del almacén. Cualquier instancia servirá, ya que el bean no puede guardar ninguna información referida al cliente. Tan pronto como el método termina su ejecución, la instancia del bean está disponible para otros clientes. Esta propiedad hace que los beans de sesión sin estado sean muy escalables para un gran número de clientes. El contenedor EJB no tiene que mover sesiones de la memoria a un almacenamiento secundario para liberar recursos, simplemente puede obtener recursos y memoria destruyendo las instancias.

Es apropiado usar beans de sesión sin estado cuando una tarea no está ligada a un cliente específico:

- Para enviar una confirmación por correo electrónico o calcular unas cuotas de un préstamo.
- Como un puente de acceso a una base de datos o a un bean de entidad. En una arquitectura cliente-servidor, el bean de sesión podría proporcionar al interfaz de usuario del cliente los datos necesarios, así como modificar objetos de negocio (base de datos o bean de entidad) a petición de la interfaz.

Algunos ejemplos de bean de sesión sin estado podrían ser:

- Un enterprise bean que comprueba si un símbolo de compañía está disponible en el mercado de valores y devuelve la última cotización registrada.
- Un enterprise bean que calcula la cuota del seguro de un cliente, basándose en los datos que se le pasa del cliente.

Beans de sesión con estado

En un bean de sesión con estado, las *variables de instancia* del bean almacenan datos específicos obtenidos durante la conexión con el cliente. Cada bean de sesión con estado, por tanto, almacena el estado conversacional de un cliente que interactúa con el bean. Este estado conversacional se modifica conforme el cliente va realizando llamadas a los métodos de negocio del bean. El estado conversacional no se guarda cuando el cliente termina la sesión.

Algunos ejemplos de beans de sesión con estado podrían ser:

- Un ejemplo típico es un carrito de la compra, en donde el cliente va guardando uno a uno los items que va comprando.
- Un enterprise bean que reserva un vuelo y alquila un coche en un sitio Web de una agencia de viajes.

Debido a que el bean guarda el estado conversacional con un cliente determinado, no le es posible al contenedor crear un almacén de instancias y compartirlo entre muchos clientes. Por ello, el manejo de beans de sesión con estado es más pesado que el de beans de sesión sin estado.

Uso de beans de sesión

En general, se debería usar un bean de sesión con estado si se cumplen las siguientes circunstancias:

- El estado del bean representa la interacción entre el bean y un cliente específico.
- El bean necesita mantener información del cliente a lo largo de un conjunto de invocaciones de métodos.
- El bean hace de intermediario entre el cliente y otros componentes de la aplicación, presentando una vista simplificada al cliente.

Para mejorar la eficiencia de la aplicación, se debería usar beans de sesión sin estado si se cumple alguna de las siguientes condiciones:

- El estado del bean no tiene ningún dato para un cliente específico.
- En una única invocación de un método, el bean realiza una tarea genérica que puede ser solicitada por cualquier cliente. Por ejemplo, se podría usar un bean sin estado para enviar un e-mail que confirme un pedido on-line.
- El bean obtiene de una base de datos un conjunto de datos de sólo lectura que se usan a menudo por los clientes. Un bean de este tipo, por ejemplo, podría obtener las filas de tabla que representan los productos que están a la venta esta semana.

2.1.2 Beans de entidad

Los beans de entidad modelan conceptos de negocio que puede expresarse como nombres. Esto es una regla sencilla más que un requisito formal, pero ayuda a determinar cuándo un concepto de negocio puede ser implementado como un bean de entidad. Los beans de entidad representan “cosas”: objetos del mundo real como hoteles, habitaciones, expedientes, estudiantes, y demás. Un bean de entidad puede representar incluso cosas abstractas como una reserva. Los beans de entidad describen tanto el estado como la conducta de objetos del mundo real y permiten a los desarrolladores encapsular las reglas de datos y de negocio asociadas con un concepto específico. Por ejemplo un EJB Estudiante encapsula los datos y reglas de negocio asociadas a un estudiante. Esto hace posible manejar de forma consistente y segura los datos asociados a un concepto.

Los beans de entidad se corresponden con datos en un almacenamiento persistente (base de datos, sistema de ficheros, etc.). Las variables de instancia del bean representan los datos en las columnas de la base de datos. El contenedor debe sincronizar las variables de instancia del bean con la base de datos. Los beans de entidad se diferencian de los beans de sesión en que las variables de instancia se almacenan de forma persistente.

Son muchas las ventajas de usar beans de entidad en lugar de acceder a la base de datos directamente. El uso de beans de entidad para transformar en objetos los datos proporciona a los programadores un mecanismo más simple para acceder y modificar los datos. Es mucho más fácil, por ejemplo, cambiar el nombre de un estudiante llamando a `student.setName()` que ejecutando un comando SQL contra la base de datos. Además, el uso de objetos favorece la reutilización del software. Una vez que un bean de entidad se ha definido, su definición puede usarse a lo largo de todo el sistema de forma consistente. Un EJB Estudiante proporciona una forma completa de acceder a la información del estudiante y eso asegura que el acceso a la información es consistente y simple.

La representación de los datos como beans de entidad puede hacer que el desarrollo sea más sencillo y menos costoso.

Diferencias con los beans de sesión

Los beans de entidad se diferencian de los beans de sesión, principalmente, en que son persistentes, permiten el acceso compartido, tienen clave primaria y pueden participar en relaciones con otros beans de entidad:

Persistencia

Debido a que un bean de entidad se guarda en un mecanismo de almacenamiento se dice que es persistente. Persistente significa que el estado del bean de entidad existe más tiempo que la duración de la aplicación o del proceso del servidor J2EE. Un ejemplo de datos persistentes son los datos que se almacenan en una base de datos.

Los beans de entidad tienen dos tipos de persistencia: **Persistencia Gestionada por el Bean** (BMP, *Bean-Managed Persistence*) y **Persistencia Gestionada por el Contenedor** (CMP, *Container-Managed Persistence*). En el primer caso (BMP) el bean de entidad contiene el código que accede a la base de datos. En el segundo caso (CMP) la relación entre las columnas de la base de datos y el bean se describe en el fichero de propiedades del bean, y el contenedor EJB se ocupa de la implementación.

Acceso compartido

Los clientes pueden compartir beans de entidad, con lo que el contenedor EJB debe gestionar el acceso concurrente a los mismos y por ello debe usar transacciones. La forma de hacerlo dependerá de la política que se especifique en los descriptores del bean.

Clave primaria

Cada bean de entidad tiene un identificador único. Un bean de entidad alumno, por ejemplo, puede identificarse por su número de expediente.

Este identificador único, o *clave primaria*, permite al cliente localizar a un bean de entidad particular.

Relaciones

De la misma forma que una tabla en una base de datos relacional, un bean de entidad puede estar relacionado con otros EJB. Por ejemplo, en una aplicación de gestión administrativa de una universidad, el bean `alumnoEJB` y el bean `actaEJB` estarían relacionados porque un alumno aparece en un acta con una calificación determinada.

Las relaciones se implementan de forma distinta según se esté usando la persistencia manejada por el bean o por el contenedor. En el primer caso, al igual que la persistencia, el desarrollador debe programar y gestionar las relaciones. En el segundo caso es el contenedor el que se hace cargo de la gestión de las relaciones. Por ello, estas últimas se denominan a veces relaciones gestionadas por el contenedor.

Persistencia manejada por el contenedor

El término persistencia manejada por el contenedor significa que el contenedor EJB maneja todos los accesos a bases de datos requeridos por el bean. El código del bean no contiene ninguna llamada SQL. Como resultado, el código del bean no está atado a un tipo específico de almacenamiento persistente. A causa de esta flexibilidad, incluso si se vuelve a desplegar el bean en un servidor distinto de J2EE que use una base de datos distinta, no hará falta recompilar el código del bean. En breve, los beans son más portables.

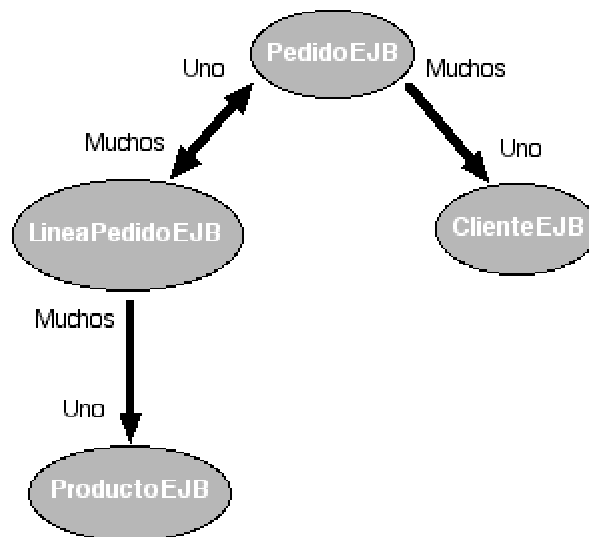
Para poder generar las llamadas de acceso a los datos, el contenedor necesita información que se proporciona en el esquema abstracto del bean de entidad

Esquema abstracto

El esquema abstracto es una parte del descriptor de despliegue de un bean de entidad. El término abstracto distingue este esquema del esquema físico del sistema de almacenamiento de datos subyacente. En una base de datos relacional, por ejemplo, el esquema físico está formado por estructuras como tablas y columnas.

En el esquema abstracto se define un nombre del enterprise bean. Este nombre puede ser usado luego como referencia en preguntas escritas en el lenguaje EJB QL (*Enterprise JavaBeans Query Language*). Es necesario definir una consulta EJB QL para cada método de búsqueda del bean (excepto `findByPrimaryKey`). Esta consulta EJB QL es ejecutada por el contenedor EJB cada vez que se invoca el método de búsqueda.

La siguiente figura muestra un esquema abstracto sencillo que describe las relaciones entre cuatro beans de entidad.



Campos persistentes

Los campos persistentes de un bean de entidad se almacenan en el almacén de datos subyacente. El conjunto de campos persistentes representa el estado del bean. En tiempo de ejecución, el contenedor EJB sincroniza automáticamente este estado con la base de datos. Durante el despliegue, el contenedor suele hacer corresponder el bean de entidad con una tabla de la base de datos y hacer corresponder los campos persistentes con las columnas de la tabla.

Por ejemplo, el bean `ClienteEJB` podría tener campos persistentes como nombre, apellidos, direcciónCorreo y telefono. En persistencia gestionada por el contenedor estos campos son virtuales. Se declaran en el esquema abstracto, pero no se codifican como variables de instancia en el bean. En su lugar, los campos persistentes se identifican en el código con métodos de acceso (de tipo *get* y *set*).

Campos de relación

Un campo de relación es como una clave externa en una base de datos: identifica a un bean relacionado. Al igual que un campo persistente, un campo de relación es virtual y se define en el enterprise bean con un método de acceso. Pero a diferencia de los campos persistentes, un campo de relación no representa el estado del bean.

2.1.3 Beans dirigidos por mensajes

Son el tercer tipo de beans propuestos por la última especificación de EJB. Estos beans permiten que las aplicaciones J2EE reciban mensajes JMS de forma asíncrona. Así, el hilo de ejecución de un cliente no se bloquea cuando está esperando que se complete algún método de negocio de otro enterprise bean. Los mensajes pueden enviarse desde cualquier componente J2EE (una aplicación cliente, otro enterprise bean, o un componente Web) o por una aplicación o sistema JMS que no use la tecnología J2EE.

Diferencias con los beans de sesión y de entidad

La diferencia más visible es que los clientes no acceden a los beans dirigidos por mensajes mediante interfaces (explicaremos esto con más detalle más adelante), sino que un bean dirigido por mensajes sólo tienen una clase bean.

En muchos aspectos, un bean dirigido por mensajes es parecido a un bean de sesión sin estado.

- Las instancias de un bean dirigido por mensajes no almacenan ningún estado conversacional ni datos de clientes.
- Todas las instancias de los beans dirigidos por mensajes son equivalentes, lo que permite al contenedor EJB asignar un mensaje a cualquier instancia. El contenedor puede almacenar estas instancias para permitir que los streams de mensajes sean procesados de forma concurrente.
- Un único bean dirigido por mensajes puede procesar mensajes de múltiples clientes.

Las variables de instancia de estos beans pueden contener algún estado referido al manejo de los mensajes de los clientes. Por ejemplo, pueden contener una conexión JMS, una conexión de base de datos o una referencia a un objeto enterprise bean.

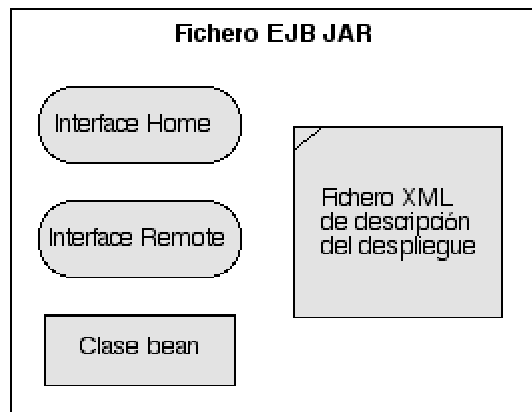
Cuando llega un mensaje, el contenedor llama al método `onMessage` del bean. El método `onMessage` suele realizar un casting del mensaje a uno de los cinco tipos de mensajes de JMS y manejarlo de forma acorde con la lógica de negocio de la aplicación. El método `onMessage` puede llamar a métodos auxiliares, o puede invocar a un bean de sesión o de entidad para procesar la información del mensaje o para almacenarlo en una base de datos.

Un mensaje puede enviarse a un bean dirigido por mensajes dentro de un contexto de transacción, por lo que todas las operaciones dentro del método `onMessage` son parte de una única transacción.

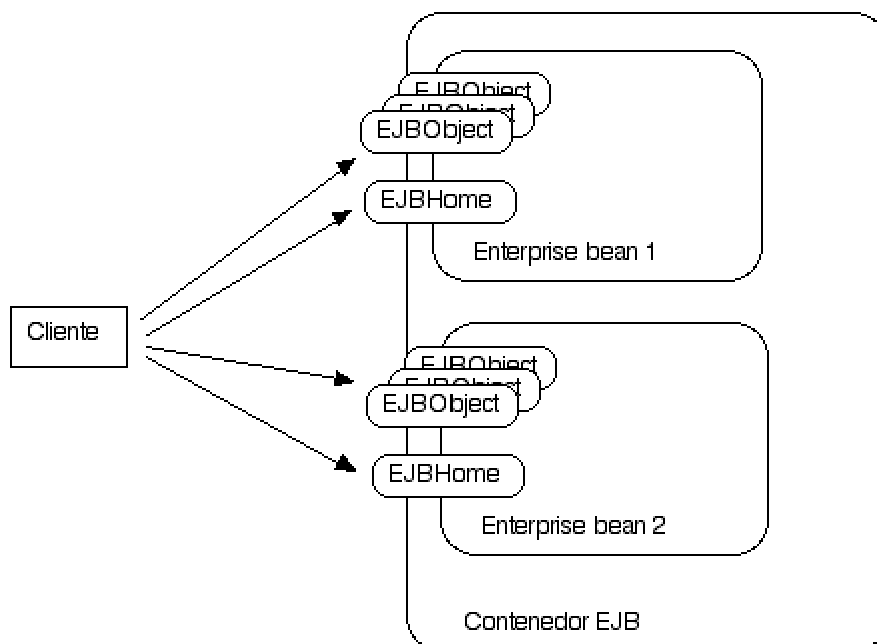
2.2 Implementación de un enterprise bean

Para implementar un enterprise bean, el desarrollador debe proporcionar cuatro elementos: una interfaz remota del bean, una interfaz home, una clase enterprise bean y un fichero de descripción del despliegue. La interfaz home declara métodos de fabricación (factory methods) que usan los clientes Java para crear nuevos enterprise beans, localizar enterprise beans existentes y destruir los beans. La interfaz remota declara los métodos de negocio del enterprise bean, que usan los clientes Java en tiempo de ejecución. La clase enterprise bean encapsula la lógica del bean e implementa los métodos de negocio definidos en la interfaz remota. El descriptor de despliegue es un fichero de configuración XML que describe al servidor EJB el enterprise bean y sus atributos de tiempo de ejecución. Es responsabilidad del desarrollador del

bean crear estos cuatro elementos y empaquetarlos en un fichero JAR listo para el despliegue.



Cuando se despliega un enterprise bean, el desplegador usa las herramientas proporcionadas por el servidor EJB para abrir el fichero JAR y leer el fichero XML descriptor del despliegue. La información de este fichero se usa para configurar la conducta en tiempo de ejecución del enterprise bean y distintos elementos relacionados con el servidor. El enterprise bean se despliega en un contenedor, que es el nombre que recibe la parte del servidor de aplicaciones que almacena instancias de enterprise beans. El contenedor se responsabiliza de gestionar las transacciones, seguridad, concurrencia, persistencia y recursos que usa el enterprise bean en tiempo de ejecución. Además, el contenedor genera los proxies EJBHome y EJBObject que implementan las interfaces home y remota del bean (los stub, usando la terminología de RMI). Estos proxies proporcionan a los clientes Java distribuidos que se encuentran en la capa de presentación un acceso a los enterprise beans de la capa intermedia.



Los clientes Java usan JNDI para obtener una referencia remota al EJBHome de un enterprise bean. Después usa el EJBHome para crear o encontrar

enterprise beans específicos en el contenedor EJB. Cuando lo consigue, recibe una referencia remota a un EJBObject. El EJBObject implementa la interfaz remota del enterprise bean.

Los clientes Java acceden al enterprise bean a través de sus proxies remotos EJBHome y EJBObject, en lugar de directamente. El contenedor EJB intercepta las invocaciones de los métodos realizadas sobre los proxies remotos, de forma que pueda manejar el entorno de ejecución del bean asociado con la invocación. Las invocaciones de métodos sobre el EJBHome hacen que el contenedor localice o cree una instancia del bean y proporcione una referencia remota EJBObject del mismo al cliente. Las invocaciones de métodos sobre el EJBObject se delegan a una instancia del enterprise bean, que contiene la lógica de negocio necesaria para dar servicio a la petición. El contenedor EJB puede usar muchas instancias de la clase enterprise bean para dar soporte a los clientes, lo cual permite que sobrellevar incrementos puntuales de carga de clientes aumentando el número de instancias gestionadas e incluso construyendo clusters de contenedores y distribuyendo las instancias.

A continuación explicamos un poco más cada uno de los componentes del bean, dando también indicaciones de nomenclatura de los nombres de ficheros. Usaremos como ejemplo el bean HelloWorldEJB.

Interfaz remota o local

La **interfaz remota** (HelloWorld.java) o **local** (HelloWorldLocal.java) del bean son las interfaces que definen los métodos de acceso que un enterprise bean ofrece a sus clientes. Estas interfaces pueden definir un acceso remoto al bean, usando la interfaz Remote de RMI, o pueden definir un acceso local al bean para aquellos casos en que el bean va a ser usado por clientes en la misma máquina virtual Java. Esta interfaz será implementada por los métodos de negocio que se definen en el fichero de definición de la clase del bean y por el contenedor EJB, que añade los aspectos del nivel de sistema.

HelloWorld.java

```
package HelloWorldSLBean;

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

public interface HelloWorld extends EJBObject {
    public String sayHello() throws RemoteException;
}
```

HelloWorldLocal.java

```
package HelloWorldSLBean;

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

public interface HelloWorldLocal extends EJBLocalObject {
    public String hi() throws EJBException;
}
```

```
}
```

Interfaz Home

La **interfaz home** (`HelloWorldHome.java`) o la **interfaz home local** (`HelloWorldHomeLocal.java`) son las interfaces que definen los métodos de gestión del ciclo de vida de las instancias del bean. Si recordamos el módulo de RMI, esta interfaz es lo que en RMI se denomina una *clase factoría*. Algunos ejemplos de métodos de esta interfaz son los métodos `create`, `remove` o `findByPrimaryKey` que crean, eliminan o buscan una instancia de bean. Esta interfaz es implementada por el contenedor EJB. La interfaz home también puede ser remota o local, dependiendo si el acceso al bean va a ser desde otra máquina virtual Java o desde la misma.

`HelloWorldHome.java`

```
package HelloWorldSLBean;

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

public interface HelloWorldHome extends EJBHome {
    public HelloWorldRemote create() throws RemoteException,
    CreateException;
}
```

`HelloWorldHomeLocal.java`

```
package HelloWorldSLBean;

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

public interface HelloWorldHomeLocal extends EJBLocalHome {
    public HelloWorld create() throws CreateException, EJBException;
}
```

Clase enterprise bean

La clase **enterprise bean** (`HelloWorld.java`) implementa los métodos de negocio del bean. El fichero es el mismo, independientemente de que la implementación del bean sea local o remota.

`HelloWorldBean.java`

```
package HelloWorldSLBean;

import javax.ejb.*;

public class HelloWorldBean implements SessionBean {
    public String sayHello() {
        String ret = new String("Hola; soy Domingo");
        return ret;
    }
}
```

```
public void ejbCreate() {}  
public void ejbActivate() {}  
public void ejbPassivate() {}  
public void ejbRemove() {}  
public void setSessionContext(SessionContext cntx) {}  
}
```

Fichero XML descriptor del despliegue

El **fichero de descripción del despliegue** (`ejb-jar.xml`) es el fichero XML que especifica información declarativa sobre el enterprise bean que será usada por el contenedor EJB para manejar los aspectos de nivel de sistema (seguridad, transacciones, persistencia, etc.).

`ejb-jar.xml`

```
<?xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE ejb-jar PUBLIC  
'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN'  
'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>  
  
<ejb-jar>  
<display-name>Hello World</display-name>  
<enterprise-beans>  
<session>  
<description>Hello World EJB</description>  
<display-name>HelloWorld</display-name>  
<ejb-name>HelloWorldEJB</ejb-name>  
<home>HelloWorldSLBean.HelloWorldHome</home>  
<remote>HelloWorldSLBean.HelloWorld</remote>  
<ejb-class>HelloWorldSLBean.HelloWorldBean</ejb-class>  
<session-type>Stateless</session-type>  
<transaction-type>Container</transaction-type>  
</session>  
</enterprise-beans>  
</ejb-jar>
```

Fichero EJB JAR que empaqueta el bean

Como hemos visto en el tema anterior, todos los ficheros que constituyen el bean se empaquetan en un único fichero con la utilidad JAR. La estructura del fichero JAR es:

```
/META-INF/ejb-jar.xml  
/META-INF/weblogic-ejb-jar.xml  
/HelloWorldSLBean/HelloWorld.class  
/HelloWorldSLBean/HelloWorldHome.class  
/HelloWorldSLBean/HelloWorldBean.class
```

En el directorio META-INF se incluyen los ficheros de descripción del despliegue: el fichero estándar (`ejb-jar.xml`) y el personalizado para el servidor de aplicaciones (`weblogic-ejb-jar.xml`). La mayoría de servidores EJB proporcionan herramientas gráficas que crean el fichero de descripción y empaquetan el bean de forma automática. El resto de directorio comprende los paquetes que definen beans.

2.3 Aplicaciones clientes

2.3.1 Cliente Java

Un cliente puede acceder a un bean de sesión o bean de entidad a través de los métodos definidos en las interfaces del bean. Estas interfaces definen la vista que el cliente tiene del bean. El resto de aspectos del bean, como implementaciones de los métodos, características definidas por los descriptores de despliegue, esquemas abstractos o llamadas de acceso a las bases de datos, se esconden de los clientes.

Los clientes que acceden a los enterprise beans pueden ser aplicaciones Java usando JNDI y Java RMI-IIOP, componentes Web (páginas JSP o servlets) o también otros beans. En todos los casos el funcionamiento es el mismo:

- En primer lugar se obtiene un contexto JNDI del contenedor EJB.
- Después se debe localizar en este contexto la interfaz Home del bean, que implementa los métodos de búsqueda y creación de beans.
- Una vez obtenido el objeto de la interfaz `home` del bean, se invoca el método `create` (para obtener una instancia nueva del bean) o `findByPrimaryKey` (para obtener una instancia concreta, en el caso de un bean de entidad). Estos métodos devuelven un objeto que implementa la clase `Remote` de Java RMI.
- Una vez obtenido el objeto remoto que hace referencia al bean, ya se puede invocar cualquier operación definida en su interfaz remota.

Es fundamental esforzarse en realizar un buen diseño de las interfaces. El desarrollo y el mantenimiento de las aplicaciones J2EE dependen de ello. Unos interfaces limpios y claros permitirán ocultar a las aplicaciones clientes las complejidades internas de la capa EJB y también permitirán internamente cambiar la implementación de los beans sin afectar a esas aplicaciones. Por ejemplo, si se cambia el método de persistencia de persistencia gestionada por el bean a persistencia gestionada por el contenedor, no será necesario modificar el código de las aplicaciones clientes. Pero si se cambia la definición de los métodos de las interfaces, entonces será necesario modificar también el código de las aplicaciones clientes. Por ello, para aislar las aplicaciones clientes de posibles cambios en los beans, es importante que se diseñen las interfaces de forma cuidadosa.

HelloWorldClient.java

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import javax.rmi.*;
```

```
public class HelloWorldClient {

    public static void main(String [] args) {
        try {
            Context jndiContext = getInitialContext();
            Object ref = jndiContext.lookup("HelloWorldEJB");
            HelloWorldRemoteHome home = (HelloWorldRemoteHome)
                PortableRemoteObject.narrow(ref,
                HelloWorldRemoteHome.class);
            HelloWorldRemote hw = home.create();
            System.out.println("Voy a llamar al bean");
            System.out.println(hw.sayHello());
            System.out.println("Ya he llamado al bean");
        } catch (Exception e) {e.printStackTrace();}
    }

    public static Context getInitialContext()
    throws javax.naming.NamingException {
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
        "weblogic.jndi.WLInitialContextFactory");
        p.put(Context.PROVIDER_URL, "t3://localhost:7001");
        return new javax.naming.InitialContext(p);
    }
}
```

2.3.2 Clientes Web: Servlet y JSP

La diferencia fundamental entre un cliente Web (Servlet y JSP) y un cliente Java es que, en general, los clientes web se ejecutan en el mismo servidor de aplicaciones en donde reside en el EJB. Por ello, cambia la forma de localizar el bean: en lugar de usar el contexto genérico JNDI, se usa un contexto específico llamado JNDI ENC (Environment Naming Context). Para colocar una referencia al bean en este contexto se usa la entrada <ejb-ref> en el fichero de descripción de despliegue de la aplicación web.

HelloWorldServlet.java

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;
import javax.rmi.*;
import HelloWorldSLBean.*;

public class HelloWorldServlet extends HttpServlet {
    private HelloWorld hw = null;
    public HelloWorldServlet() throws NamingException {
        try {
            Context jndiContext = getInitialContext();
            Object ref =
jndiContext.lookup("java:comp/env/ejb/HelloWorldEJB");
            HelloWorldHome home = (HelloWorldHome)
                PortableRemoteObject.narrow(ref, HelloWorldHome.class);
            this.hw = home.create();
        } catch (java.rmi.RemoteException re){re.printStackTrace();}
    }
}
```

```

        catch (javax.naming.NamingException ne){ne.printStackTrace();}
        catch (javax.ejb.CreateException ce){ce.printStackTrace();}
    }

    public void doGet(HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hola</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<h1> HelloWorldEJB dice: ");
        out.println(this.hw.hi());
        out.println("</h1>");
        out.println("</body>");
        out.println("</html>");
    }

    public static Context getInitialContext()
        throws javax.naming.NamingException {
        Properties p = new Properties();
        return new javax.naming.InitialContext(p);
    }
}

HelloWorld.jsp

<%@ page import="HelloWorldSLBean.*, javax.naming.*,
javax.rmi.PortableRemoteObject,java.rmi.RemoteException,
javax.ejb.CreateException" %>
<% response.addHeader("Cache-Control", "no-cache, must-revalidate"); %>
<%@ include file="CommonFunctions.jsp" %>

<%!
    HelloWorld helloWorld = null;
%>

<html>
<head>
<title>HelloWorld</title>
</head>
<body bgcolor="#ffffff" LEFTMARGIN="10" RIGHTMARGIN="10"
    link="#3366cc" vlink="#9999cc" alink="#0000cc">

<%
    try {
        Context jndiContext = getInitialContext();
        Object ref =
jndiContext.lookup("java:comp/env/ejb/HelloWorldEJB");
        HelloWorldHome home = (HelloWorldHome)
        PortableRemoteObject.narrow(ref,HelloWorldHome.class);
        helloWorld = home.create();
    } catch (Throwable t) {
        t.printStackTrace();
        out.print(t.getMessage()+"<br>Stacktrace:<br>");
        t.printStackTrace(new PrintWriter(out,true));
    }
%>

```

```
<h1> HelloWorld bean dice: <%= helloWorld.hi()%> </h1>
```

```
</body>  
</html>
```

web.xml

```
<web-app>  
  <display-name>Hello World</display-name>  
  <description>Hello World</description>  
  
  <servlet>  
    <servlet-name>HelloWorldJSP</servlet-name>  
    <jsp-file>HelloWorld.jsp</jsp-file>  
  </servlet>  
  
  <servlet>  
    <servlet-name>HelloWorldServlet</servlet-name>  
    <servlet-class>HelloWorldServlet</servlet-class>  
  </servlet>  
  
  <servlet-mapping>  
    <servlet-name>HelloWorldServlet</servlet-name>  
    <url-pattern>/HolaMundo</url-pattern>  
  </servlet-mapping>  
  
  <servlet-mapping>  
    <servlet-name>HelloWorldJSP</servlet-name>  
    <url-pattern>/HolaMundo.jsp</url-pattern>  
  </servlet-mapping>  
  ...  
  
  <ejb-ref>  
    <ejb-ref-name>ejb/HelloWorldEJB</ejb-ref-name>  
    <ejb-ref-type>Session</ejb-ref-type>  
    <home>HelloWorldSLBean.HelloWorldHome</home>  
    <remote>HelloWorldSLBean.HelloWorld</remote>  
  </ejb-ref>  
</web-app>
```

weblogic.xml

```
<weblogic-web-app>  
  <reference-descriptor>  
    <ejb-reference-description>  
      <ejb-ref-name>ejb/HelloWorldEJB</ejb-ref-name>  
      <jndi-name>HelloWorldEJB</jndi-name>  
    </ejb-reference-description>  
  </reference-descriptor>  
</weblogic-web-app>
```

2.4 Acceso remoto y local a los beans

Al diseñar aplicaciones J2EE, una de las primeras decisiones que hay que tomar es el tipo de acceso que van a utilizar en los beans: remoto o local.

2.4.1. Acceso remoto

Un cliente remoto de un enterprise bean tiene las siguientes características

- Puede correr en una máquina física distinta o en una máquina virtual Java (JVM) distinta de aquella en la que se encuentra el bean al que está accediendo.
- Puede ser un componente Web, una aplicación cliente J2EE o también otro enterprise bean.

Para crear un enterprise bean con un acceso remoto es necesario codificar una interfaz remota y una interfaz home. La *interfaz remota* define los métodos de negocio que podrán ser respondidos por las instancias del bean. Por ejemplo, la interfaz remota de un bean llamado CuentaBancariaEJB podría tener métodos de negocio como transferir o credito. La *interfaz home* proporciona los métodos necesarios para gestionar el ciclo de vida de las instancias del bean, como create y remove. Para los beans de entidad, la interfaz home también define métodos de búsqueda y métodos home (equivalentes a los métodos de clase en Java). Los métodos de búsqueda se usan para localizar instancias de beans de entidad a partir de su clave primaria. Los métodos home son métodos de negocio cuya invocación afecta a todas las instancias de un bean.

2.4.2 Acceso local

Un cliente local tiene estas características:

- Debe correr en la misma JVM que el bean al que está accediendo.
- Puede ser un componente Web o también otro enterprise bean.
- Para el cliente local, la localización del enterprise bean al que está accediendo no es transparente.
- A menudo el cliente local es otro bean de entidad que tienen una relación gestionada por el contenedor con otro bean de entidad.

Al igual que en el acceso remoto, para construir un enterprise bean que permita acceso local se debe codificar la interfaz local y la interfaz home local. La interfaz local define los métodos de negocio del bean y la interfaz home local los métodos que gestionan el ciclo de vida de las instancias del bean.

2.4.3. Interfaces locales y relaciones gestionadas por el contenedor

Si un bean de entidad es el objetivo de una relación gestionada por el contenedor, entonces debe tener una interfaz local. La dirección de la relación determina si un bean es un objetivo o no lo es. En la figura 2.1, por ejemplo, `ProductoEJB` es el objetivo de una relación unidireccional con `LineaPedidoEJB`. Debido a que `LineaPedidoEJB` accede a `ProductoEJB` localmente, `ProductoEJB` debe tener una interfaz local. El bean `LineaPedidoEJB` también necesita una interfaz local ya que es el objetivo de una relación con `PedidoEJB`. Y como la relación entre `LineaPedidoEJB` y `PedidoEJB` es bidireccional ambos beans deben tener interfaces locales.

Todos los beans que participan en un grupo de relaciones manejadas por el contenedor deben residir en el mismo fichero JAR, ya que todos requieren interfaces locales.

El beneficio principal de las interfaces locales es una mejora en la eficiencia de las invocaciones entre beans, debido a que los parámetros no tienen que serializarse.

Tema 3: EJBs de sesión con estado

Vamos a construir en esta sesión un ejemplo completo de bean de sesión. En concreto, vamos a implementar un bean de sesión con estado que define un carrito de la compra de una librería.

El bean de sesión `CartEJB` representa un carrito de la compra de una librería online. El cliente del bean podrá añadir un libro al carrito, eliminar un libro o consultar el contenido del carrito. Tal y como adelantamos en el tema 2, para construir el enterprise bean `CartEJB`, se necesitan los siguientes ficheros:

- Clase de implementación bean (`CartBean`)
- Interface home (`CartHome`)
- Interface remota (`Cart`)

El estado del bean va a estar definido por sus variables de instancias. Estas variables de instancia deben declararse en la clase `CartBean` que implementa la lógica de negocio, y serán actualizadas por los métodos del bean, como consecuencia de la invocación remota del cliente.

Además de estas clases, vamos a necesitar dos clases de apoyo, la clase `BookException` y `IdVerifier`.

La siguiente tabla muestra cómo están relacionados los métodos en la interface home, la interfaz remota y la implementación.

	Método de la interfaz home	Método de la interfaz remota	Método de la clase de implementación
Creación	<code>Cart create(args) throws CreateException, RemoteException;</code>	No disponible	<code>public void ejbCreate(args)</code>
Uso	No disponible	Métodos de negocio (deben arrojar la excepción <code>RemoteException</code>)	Métodos de negocio con la misma signatura (arrojar <code>RemoteException</code> es opcional)
Borrado	<code>public void remove();</code>	<code>public void remove()</code>	<code>public void ejbRemove();</code>

El código fuente de este ejemplo está en el fichero `ejercicio3.zip` en la página web del módulo de EJB.

3.1 La clase bean de sesión (SessionBean)

Siguiendo las indicaciones de nomenclatura que comentamos en la sesión pasada, la clase bean se llama `CartBean`. Esta clase implementa la lógica de negocio del bean, y debe cumplir los siguientes requisitos:

- Implementa la interfaz `SessionBean`.
- La clase se define como `public`.
- La clase no puede definirse como `abstract` ni `final`.
- Implementa uno o más métodos `ejbCreate`.
- Implementa los métodos remotos definidos en la interfaz remota del bean.
- Contiene un constructor `public` sin parámetros.
- No debe definir el método `finalize`.

El código fuente sigue a continuación:

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

public class CartBean implements SessionBean {

    String customerName;
    String customerId;
    Vector contents;

    public void ejbCreate(String person)
        throws CreateException {

        if (person == null) {
            throw new CreateException("Null person not allowed.");
        }
        else {
            customerName = person;
        }

        customerId = "0";
        contents = new Vector();
    }

    public void ejbCreate(String person, String id)
        throws CreateException {

        if (person == null) {
            throw new CreateException("Null person not allowed.");
        }
        else {
            customerName = person;
        }

        IdVerifier idChecker = new IdVerifier();
        if (idChecker.validate(id)) {
            customerId = id;
        }
    }
}
```



```
    }
    else {
        throw new CreateException("Invalid id: "+ id);
    }

    contents = new Vector();
}

public void addBook(String title) {
    contents.addElement(title);
}

public void removeBook(String title) throws BookException {

    boolean result = contents.removeElement(title);
    if (result == false) {
        throw new BookException(title + "not in cart.");
    }
}

public Vector getContents() {
    return contents;
}

public CartBean() {}
public void ejbRemove() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void setSessionContext(SessionContext sc) {}
}
```

3.1.1 Las variables de instancia

Las variables de instancia definen el estado de una instancia de enterprise bean. En este caso se trata de `customerName`, `customerId` y `contents`. Las variables de instancia se inicializan al crearse una instancia del bean, copiándose en ellas los argumentos del método `ejbCreate`. Después son modificadas por las llamadas de los clientes a los métodos de negocio `addBook()` y `removeBook()`.

3.1.2 La interfaz `SessionBean`

La interfaz `SessionBean` extiende la interfaz `EnterpriseBean`, que a su vez extiende el interfaz `Serializable`. La interfaz `SessionBean` declare los métodos `ejbRemove`, `ejbActivate`, `ejbPassivate`, y `setSessionContext`. Estos métodos se llaman desde el contenedor EJB cuando el bean va pasando de un estado a otro en su ciclo de vida. La clase `CartBean` no usa estos métodos, pero debe implementarlos porque están declarados en la interfaz `SessionBean`.

3.1.3 Los métodos `ejbCreate`

Un cliente no puede instanciar directamente un bean, ya que éste reside en el contenedor EJB. Sólo el contenedor EJB puede realizar la instanciación. En el ejemplo del carrito de la compra esto sucede de la siguiente forma:

1. El cliente invoca un método `create` en el objeto `home`:

```
Cart shoppingCart = home.create("Duke DeEarl", "123");
```

2. El contenedor EJB instancia el enterprise bean.

3. El contenedor EJB invoca el método `ejbCreate` apropiado en `CartBean`:

```
public void ejbCreate(String person, String id)
    throws CreateException {

    if (person == null) {
        throw new CreateException("Null person not allowed.");
    }
    else {
        customerName = person;
    }

    IdVerifier idChecker = new IdVerifier();
    if (idChecker.validate(id)) {
        customerId = id;
    }
    else {
        throw new CreateException("Invalid id: " + id);
    }

    contents = new Vector();
}
```

Al tratarse de un bean de sesión con estado, el método `ejbCreate` inicializa este estado. El método `ejbCreate` anterior, por ejemplo, inicializa las variables `customerName` y `customerId` con los argumentos pasados por el método `create`.

Un enterprise bean debe tener uno o más métodos `ejbCreate`. Las signatures de estos métodos deben cumplir los siguientes requisitos:

- El modificador de control de acceso debe ser `public`.
- El tipo de retorno debe ser `void`.
- Si el bean permite un acceso remoto, los argumentos deben ser tipos legales de RMI.
- El modificador no puede ser `static` ni `final`.

La cláusula `throws` puede incluir la excepción `javax.ejb.CreateException` y cualquier otra excepción específica de la aplicación. El método `ejbCreate` arroja normalmente una excepción `CreateException` si un parámetro de entrada no es válido.

3.1.4 Métodos de negocio

El propósito fundamental de un bean de sesión es ejecutar tareas de negocio a petición de aplicaciones cliente. El cliente invoca métodos de negocio en la referencia remota del objeto que es devuelta por el método `create`. Desde la perspectiva del cliente, los métodos de negocio parecen ejecutarse de forma

local, pero realmente lo que se ejecuta es un método del stub que envía al skeleton (recordatorio de RMI!!) la petición de invocación remota, junto con todo el *marshaling* de parámetros.

Un ejemplo del punto de vista del cliente:

```
Cart shoppingCart = home.create("Duke DeEarl", "123");
...
shoppingCart.addBook("The Martian Chronicles");
shoppingCart.removeBook("Alice In Wonderland");
bookList = shoppingCart.getContents();
```

La clase `CartBean` implementa los métodos de negocio en el siguiente código:

```
public void addBook(String title) {
    contents.addElement(new String(title));
}

public void removeBook(String title) throws BookException {
    boolean result = contents.removeElement(title);
    if (result == false) {
        throw new BookException(title + "not in cart.");
    }
}

public Vector getContents() {
    return contents;
}
```

La signatura de un método de negocio debe cumplir los siguientes requisitos:

- El nombre del método no debe entrar en conflicto con ningún otro definido por la arquitectura EJB. Por ejemplo, un método de negocio no se puede llamar `ejbCreate` o `ejbActivate`.
- El modificador de control de acceso debe ser `public`.
- Si el bean permite acceso remoto, los argumentos y los tipos devueltos deben ser tipos legales de RMI.
- El modificador no puede ser `static` ni `final`.

La cláusula `throws` puede incluir excepciones que se definen en la aplicación. Por ejemplo, el método `removeBook` arroja la excepción `BookException` si el libro no está en el carrito.

Para indicar un problema de nivel de sistema, como la imposibilidad de conectar con una base de datos, un método de negocio debería arrojar una excepción `javax.ejb.EJBException`. Cuando un método de negocio arroja una excepción `EJBException`, el contenedor EJB la envuelve en una `RemoteException`, que es capturada por el cliente. El contenedor no envuelve excepciones de aplicación como `BookException`. Debido a que `EJBException` es una subclase de `RuntimeException`, no es necesario incluirla en la cláusula `throws` del método de negocio.

3.2 La interfaz home

La interfaz home extiende la interfaz `javax.ejb.EJBHome`. Para un bean de sesión, el propósito de la interfaz home es definir los métodos `create()` (puede haber más de uno, dependiendo de los argumentos) que el cliente remoto puede invocar para crear instancias del bean. El cliente `CartCliente`, por ejemplo, invoca este método `create`

```
(Cart shoppingCart = home.create("Duke DeEarl", "123");
```

Por cada método `create` en la interfaz home debe existir un método `ejbCreate` correspondiente en la clase de implementación del bean (clase `CartBean`). En nuestro ejemplo, definíamos dos firmas de métodos `ejbCreate`:

```
public void ejbCreate(String person) throws CreateException
...
public void ejbCreate(String person, String id)
    throws CreateException
```

Y el código correspondiente en el fichero `CartHome.java` es el siguiente:

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

public interface CartHome extends EJBHome {
    Cart create(String person) throws
        RemoteException, CreateException;
    Cart create(String person, String id) throws
        RemoteException, CreateException;
}
```

Las firmas de ambos conjuntos de métodos son similares, pero difieren en aspectos importantes. Un resumen de las reglas que debemos aplicar al crear ambas firmas es el siguiente:

- El número y tipos de argumentos en el método `create` debe corresponderse con los definidos en el método `ejbCreate` asociado.
- Los tipos de los argumentos y valores devueltos deben ser tipos válidos RMI.
- Cuando usemos llamadas remotas, los métodos `create` de la interfaz home siempre debe devolver el tipo de la interfaz remota del bean. Cuando un cliente invoque este método se le devolverá una referencia remota del bean recién creado. El método `ejbCreate`, sin embargo, debe devolver `void`. Es el contenedor EJB el que se encarga de crear el bean, llamar al método `ejbCreate` y devolver la referencia remota.
- La cláusula `throws` del método `create` debe incluir las excepciones `java.rmi.RemoteException` y `javax.ejb.CreateException`.

3.3 Interfaz Remote

La interfaz remota siempre debe extender la interfaz `javax.ejb.EJBObject`. Esta interfaz define los métodos de negocio que el cliente remoto puede invocar. A continuación se muestra el código fuente del fichero `Cart.java`:

```
import java.util.*;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Cart extends EJBObject {

    public void addBook(String title) throws RemoteException;
    public void removeBook(String title) throws
        BookException, RemoteException;
    public Vector getContents() throws RemoteException;
}
```

La definición de los métodos en la interfaz remota debe seguir estas reglas:

- Cada método en la interfaz remota debe corresponderse con un método equivalente en la clase del enterprise bean. Las firmas de ambos deben ser iguales.
- Los tipos de los argumentos y valores devueltos deben ser tipos válidos RMI.
- La cláusula `throws` debe incluir la excepción `java.rmi.RemoteException`.

3.4 El fichero descriptor del despliegue

El fichero `ejb-jar.xml` se muestra a continuación

```
<ejb-jar>
  <display-name>CartJAR</display-name>
  <enterprise-beans>
    <session>
      <display-name>CartEJB</display-name>
      <ejb-name>CartEJB</ejb-name>
      <home>CartHome</home>
      <remote>Cart</remote>
      <ejb-class>CartBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
      <security-identity>
        <use-caller-identity/>
      </security-identity>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>
```

Y el fichero `weblogic-ejb-jar.xml` es el siguiente. Los únicos datos que se definen son el nombre del EJB y el nombre JNDI del bean.

```
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>CartEJB</ejb-name>
    <stateful-session-descriptor><stateful-session-cache>
    </stateful-session-cache><stateful-session-clustering>
    </stateful-session-clustering></stateful-session-descriptor>
    <transaction-descriptor></transaction-descriptor>
    <jndi-name>CartEJB</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

3.5 Clases de apoyo

El bean de sesión **CartEJB** tiene dos clases de apoyo: `BookException` y `IdVerifier`. La excepción `BookException` es arrojada por el método `removeBook` y el `IdVerifier` valida el `clienteId` en uno de los métodos `ejbCreate`. Ambas clases deben residir en el mismo fichero JAR en el que se empaqueta todo el enterprise bean.

A continuación se muestran los códigos de ambas clases:

```
public class BookException extends Exception {

    public BookException() {
    }

    public BookException(String msg) {
        super(msg);
    }
}

public class IdVerifier {

    public IdVerifier() {
    }

    public boolean validate(String id) {

        boolean result = true;
        for (int i = 0; i < id.length(); i++) {
            if (Character.isDigit(id.charAt(i)) == false)
                result = false;
        }
        return result;
    }
}
```

3.6 El fichero EJB JAR

El fichero EJB JAR tiene la siguiente estructura, sencilla al no usar paquetes en la definición de las clases

```
extracted: META-INF/MANIFEST.MF
extracted: BookException.class
```

```
extracted: Cart.class
extracted: CartBean.class
extracted: CartHome.class
extracted: IdVerifier.class
  created: META-INF/
extracted: META-INF/ejb-jar.xml
extracted: META-INF/weblogic-ejb-jar.xml
```

3.7 La aplicación cliente

La aplicación cliente compra algunos libros y después elimina algún libro no existente en el carrito, lo que genera un error.

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

public class CartClient {

    public static void main(String[] args) {
        try {
            Context jndiContext = getInitialContext();
            Object objref = jndiContext.lookup("CartEJB");

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

            Cart shoppingCart = home.create("Duke DeEarl", "123");

            shoppingCart.addBook("The Martian Chronicles");
            shoppingCart.addBook("2001 A Space Odyssey");
            shoppingCart.addBook("The Left Hand of Darkness");

            Vector bookList = new Vector();
            bookList = shoppingCart.getContents();

            Enumeration enumer = bookList.elements();
            while (enumer.hasMoreElements()) {
                String title = (String) enumer.nextElement();
                System.out.println(title);
            }

            shoppingCart.removeBook("Alice in Wonderland");
            shoppingCart.remove();

            System.exit(0);

        } catch (BookException ex) {
            System.err.println("Caught a BookException: " +
                               ex.getMessage());
            System.exit(0);
        } catch (Exception ex) {
            System.err.println("Caught an unexpected exception!");
            ex.printStackTrace();
            System.exit(1);
        }
    }
}
```

```
public static Context getInitialContext()
    throws javax.naming.NamingException {
    Properties p = new Properties();
    p.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
    p.put(Context.PROVIDER_URL, "t3://localhost:7001");
    return new javax.naming.InitialContext(p);
}
}
```

3.8 Otras características de los enterprise beans

Los aspectos que siguen se aplican tanto a beans de sesión como a beans de entidad. Veremos en primer lugar cómo leer desde los beans entradas del entorno que se especifican de forma declarativa en la descripción de despliegue del bean. A continuación revisaremos cómo comparar enterprise beans. Terminaremos comentando cómo pasar al cliente referencias remotas de beans.

3.8.1 Cómo acceder a entradas del entorno

Una *entrada del entorno* (*environment entry*) es una pareja nombre-valor que se almacena en el fichero de descripción del despliegue y que permite modificar la lógica de negocio del bean sin cambiar su código fuente ni recompilarlo. El formato con el que se almacena en el fichero `ejb-jar.xml` es el siguiente

```
<env-entry>
  <description>Discount Level</description>
  <env-entry-name>DiscountLevel</env-entry-name>
  <env-entry-type>java.lang.Double</env-entry-type>
  <env-entry-value>300.0</env-entry-value>
</env-entry>
<env-entry>
  <description>Discount Percent</description>
  <env-entry-name>Discount Percent</env-entry-name>
  <env-entry-type>java.lang.Double</env-entry-type>
  <env-entry-value>0.15</env-entry-value>
</env-entry>
```

En el ejemplo se han definido las entradas del entorno `DiscountLevel` y `DiscountPercent`.

Una vez desplegado el enterprise bean en el contenedor EJB, estas entradas se almacenan como objetos JNDI en el contexto `java:comp/env`. Para usar estas entradas en los métodos de negocio del enterprise bean hay que usar JNDI y obtener el contexto de nombres del entorno invocando a `lookup` del contexto inicial con el parámetro `java:comp/env`. Luego ya se puede invocar a `lookup` del contexto de nombres pasándole como parámetro el string que define el nombre de la entrada del entorno.

El siguiente código proporciona un ejemplo práctico, el método de negocio `applyDiscount`:


```
public double applyDiscount(double amount) {  
    try {  
        double discount;  
  
        Context initial = new InitialContext();  
        Context environment =  
            (Context)initial.lookup("java:comp/env");  
  
        Double discountLevel =  
            (Double)environment.lookup("DiscountLevel");  
        Double discountPercent =  
            (Double)environment.lookup("DiscountPercent");  
  
        if (amount >= discountLevel.doubleValue()) {  
            discount = discountPercent.doubleValue();  
        }  
        else {  
            discount = 0.00;  
        }  
  
        return amount * (1.00 - discount);  
    } catch (NamingException ex) {  
        throw new EJBException("NamingException: "+  
            ex.getMessage());  
    }  
}
```

3.8.2 Cómo comparar enterprise beans

Un cliente puede determinar si un bean de sesión con estado ya ha sido definido y está residiendo en el contenedor de aplicaciones usando el método `isIdentical`:

```
bookCart = home.create("Bill Shakespeare");  
videoCart = home.create("Lefty Lee");  
...  
if (bookCart.isIdentical(bookCart)) {  
    // true ... }  
if (bookCart.isIdentical(videoCart)) {  
    // false ... }
```

Debido a que los beans de sesión sin estado siempre tienen la misma identidad como objetos, el método `isIdentical` siempre devolverá `true` cuando se comparan.

Para determinar si dos beans de entidad son idénticos, el cliente puede usar el método `isIdentical` o puede obtener y comparar las claves primarias de los beans:

```
String key1 = (String)accta.getPrimaryKey();  
String key2 = (String)acctb.getPrimaryKey();
```

```
if (key1.compareTo(key2) == 0)
    System.out.println("equal");
```

3.8.3 Cómo devolver una referencia a un enterprise bean

Supongamos que queremos pasar la referencia de un mismo bean a otro bean, o al cliente que invoca un método de negocio del bean. No podemos usar `this`, por que apunta a la instancia del bean, que está corriendo en el contenedor EJB.

Se obtiene llamando al método `getEJBObject` de la interfaz `SessionContext`. Un bean de entidad llamaría al método `getEJBObject` de la interfaz `EntityContext`. Estas interfaces proporcionan a los beans la posibilidad de acceder a contextos de instancias mantenidos por el contenedor EJB. Normalmente, el bean graba los contextos en el método `setSessionContext`. El siguiente código muestra un ejemplo de cómo usar estos métodos:

```
public class WagonBean implements SessionBean {

    SessionContext context;
    ...
    public void setSessionContext(SessionContext sc) {
        this.context = sc;
    }
    ...
    public void passItOn(Basket basket) {
        ...
        basket.copyItems(context.getEJBObject());
    }
    ...
}
```

Tema 4: EJBs de entidad con persistencia gestionada por el bean (BMP)

El bean de entidad presentado en este ejemplo representa una sencilla cuenta de banco. Los beans de entidad guardan su estado en algún dispositivo secundario (fichero o base de datos). En este caso, el estado de las instancias del bean `SavingsAccountEJB` se almacenan en la tabla `savingsaccount` de una base de datos relacional. Esta tabla se crea con la siguiente sentencia SQL

```
CREATE TABLE savingsaccount
(id VARCHAR(3)
CONSTRAINT pk_savingsaccount PRIMARY KEY,
firstname VARCHAR(24),
lastname VARCHAR(24),
balance NUMERIC(10,2));
```

El bean incluye métodos mediante los cuales las aplicaciones clientes pueden obtener instancias del bean (cuentas) que cumplan unos requisitos de balance, o cuentas de personas con un determinado apellido. También incluye un ejemplo de un método que se puede invocar desde la aplicación cliente para que se realice una determinada operación de mantenimiento sobre todas las instancias del bean.

Los beans de entidad tienen un conjunto de métodos que permiten el acceso a operaciones de bases de datos. La siguiente tabla resume los mismos:

La implementación remota del bean `SavingsAccountEJB` requiere las siguientes clases e interfaces:

- Clase `EntityBean` (`SavingsAccountBean`)
- Interfaz `Home` (`SavingsAccountHome`)
- Interfaz `Remote` (`SavingsAccountRemote`)

La lista de clases es la misma que para los beans de sesión, con la excepción de la clase `EntityBean`, que en lugar de implementar la interfaz `SessionBean` deberá implementar la interfaz `EntityBean`. A continuación se detallan estas clases, junto con alguna clase auxiliar y la aplicación ejemplo que va a usar el bean. En concreto, estas últimas son:

- Una clase auxiliar llamada `InsufficientBalanceException`
- La aplicación cliente se implementa con la clase llamada `SavingsAccountClient`

4.1 Clase EntityBean

La clase EntityBean se llama `SavingsAccountBean`. Esta clase define los métodos a los que va a llamar el contenedor EJB cuando se requiera el cliente realice una invocación a un método de la interfaz remota, de la interfaz home o cuando haya que realizar alguna operación de gestión del bean.

La diferencia fundamental con un bean de sesión son los métodos relacionados con con la persistencia de las instancias del bean. Entre ellas:

- Cargar y grabar el estado de la instancia del bean en la base de datos. Ambas operaciones se implementan en los métodos `ejbLoad` y `ejbStore`, que son métodos llamados por el contenedor EJB cuando es necesario.
- Métodos de búsqueda. Todos ellos comienzan por `ejbFind` y permiten la búsqueda de una instancia concreta o de una colección de instancias que cumplen una condición.
- Métodos de mantenimiento de todas las instancias. Comienzan por `ejbHome` y permiten realizar una operación que afecta a todas las instancias del bean.

La clase define los métodos públicos que agrupados en la siguiente tabla:

Métodos relacionados con la interfaz remota de las instancias del bean

```
public void debit(BigDecimal amount)
public void credit(BigDecimal amount)
public String getFirstName()
public String getLastName()
public BigDecimal getBalance()
```

Métodos relacionados con la interfaz home

```
public void ejbHomeChargeForLowBalance(BigDecimal
minimumBalance,
                                     BigDecimal charge)
    throws InsufficientBalanceException
public String ejbCreate(String id,
                        String firstName,
                        String lastName,
                        BigDecimal balance)
    throws CreateException
public String ejbFindByPrimaryKey(String primaryKey)
    throws FinderException
public Collection ejbFindByLastName(String lastName)
    throws FinderException
public Collection ejbFindInRange(BigDecimal low,
                                BigDecimal high)
    throws FinderException
```

Métodos relacionados con el ciclo de vida del bean

```
public void ejbRemove()
public void setEntityContext(EntityContext context)
public void unsetEntityContext()
public void ejbActivate()
public void ejbPassivate()
public void ejbLoad()
public void ejbStore()
public void ejbPostCreate(String id,
                          String firstName,
                          String lastName,
                          BigDecimal balance)
```

Resumiendo, la clase implementa:

- La interfaz `EntityBean`
- Los métodos `ejbCreate` y `ejbPostCreate`
- Métodos de búsqueda
- Métodos de negocio
- Métodos home

Además, un bean de entidad con persistencia gestionada por el bean debe cumplir los siguientes requisitos:

- La clase se define como `public`
- La clase no puede ser definida como `abstract` o `final`
- Contiene un constructor vacío
- No implementa el método `finalize`

4.1.1 La interfaz `EntityBean`

La interfaz `EntityBean` extiende la interfaz `EnterpriseBean`, la cual a su vez extiende la interfaz `Serializable`. La interfaz `EntityBean` declara una lista de métodos, como `ejbActivate` y `ejbLoad`, que deben implementarse en la clase bean entidad que estemos definiendo (`SavingsAccountBean`, en este caso). Estos métodos se explican más adelante.

4.1.2 El método `ejbCreate`

Cuando el cliente invoca un método `create` con unos determinados argumentos, el contenedor EJB invoca el método correspondiente `ejbCreate`. Normalmente, un método `ejbCreate` en un bean de entidad realiza las siguientes tareas:

- Inserta el estado de la entidad en la base de datos
- Inicializa las variables de instancia
- Devuelve la clave primaria

El método `ejbCreate` de `SavingsAccountBean` inserta el estado de la entidad en la base de datos invocando al método privado `insertRow`, el cual lanza una

sentencia SQL INSERT. A continuación se encuentra el código fuente del método `ejbCreate`

```
public String ejbCreate(String id, String firstName,
    String lastName, BigDecimal balance)
    throws CreateException {

    if (balance.signum() == -1) {
        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;
}
```

De la misma forma que ya vimos en el tema anterior, con el bean de sesión `CartEJB`, un bean de entidad puede tener múltiples métodos `ejbCreate` con distintas signaturas.

Cuando se escribe un método `ejbCreate` de un bean de entidad deben seguirse las siguientes reglas:

- El modificador de control de acceso al método debe ser `public`
- El tipo devuelto debe ser `String` y el objeto corresponde a la clave primaria
- Los argumentos deben ser tipos legales de RMI
- No se debe declarar el método `static` o `final`

La cláusula `throws` puede incluir la excepción `javax.ejb.CreateException` y excepciones que sean específicas a la aplicación. Un método `ejbCreate` normalmente arroja una excepción `CreateException` si un parámetro de entrada es inválido. Si un método `ejbCreate` no puede crear una entidad a causa de que ya existe otra entidad con la misma clave primaria, debería arrojar una excepción `javax.ejb.DuplicateKeyException` (una subclase de `CreateException`). Si un cliente recibe una excepción `CreateException` o `DuplicateKeyException`, debería asumir que la entidad no ha sido creada.

La clase de clave primaria se especifica en el descriptor de despliegue del bean de entidad. En la mayoría de los casos, la clase de clave primaria será un `String`, un `Integer` o alguna otra clase que pertenece a las librerías estándar de J2SE o J2EE. Sin embargo, en algunos casos será necesario definir una clase de clave primaria propia. Por ejemplo, si el bean tiene una clave primaria

compuesta (esto es, una compuesta de múltiples campos), entonces es necesario crear una clase de clave primaria.

Por último, el estado de un bean de entidad puede insertarse en la base de datos directamente, por una aplicación desconocida al servidor J2EE. Por ejemplo, un script SQL podría insertar una fila en la tabla `savingsaccount`. Aunque el bean de entidad correspondiente a esta fila no ha sido creado por un método `ejbCreate`, el bean puede ser obtenido y usado por un programa cliente.

4.1.3 El método `ejbPostCreate`

Para cada método `ejbCreate`, se debe escribir un método `ejbPostCreate` en la clase bean de entidad. El contenedor EJB invoca el método `ejbPostCreate` inmediatamente después de que llama a `ejbCreate`. A diferencia de `ejbCreate`, el método `ejbPostCreate` puede invocar los métodos `getPrimaryKey` y `getEJBObject` de la interfaz `EntityContext`. En el tema anterior se comentó brevemente el método `getEJBObject` en el apartado *Cómo pasar una referencia al objeto enterprise bean*. A menudo, los métodos `ejbPostCreate` se dejan vacíos.

La signatura de un método `ejbPostCreate` debe conformar los siguientes requisitos:

- El número y tipos de los argumentos debe corresponderse con los del método `ejbCreate` equivalente
- El modificador de control de acceso debe ser `public`.
- El modificador del método no puede ser `final` o `static`.
- El tipo devuelto debe ser `void`

La cláusula `throws` puede incluir una excepción `javax.ejb.CreateException` y excepciones que son específicas a la aplicación cliente.

4.1.4 El método `ejbRemove`

Un cliente borra un bean de entidad invocando el método `remove`. Esta invocación causa que el contenedor EJB llame al método `ejbRemove`, el cual borra el estado de la entidad de la base de datos. en la clase `SavingsAccountBean`, el método `ejbRemove` invoca un método privado llamado `deleteRow`, el cual envía una sentencia SQL `DELETE`. El método `ejbRemove` es corto:

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

Si el método `ejbRemove` encuentra un problema del sistema, debería arrojar la excepción `javax.ejb.EJBException`. Si encuentra un error de aplicación debería arrojar una excepción `javax.ejb.RemoveException`.

Un bean de entidad también puede ser borrado directamente mediante un borrado de la base de datos. Por ejemplo, si un script SQL borra una fila que contiene un bean entidad, entonces el bean de entidad es eliminado.

4.1.5 Los métodos `ejbLoad` y `ejbStore`

El contenedor EJB invoca a los métodos `ejbLoad` y `ejbStore` cuando necesita sincronizar las variables de instancia de los beans de entidad con los valores correspondientes almacenados en la base de datos. El método `ejbLoad` actualiza el valor de las variables de instancia con los valores de la base de datos, y el método `ejbStore` escribe las variables en la base de datos. El cliente no puede llamar directamente a estos métodos.

Si un método de negocio está asociado con una transacción, el contenedor invoca `ejbLoad` antes de que el método de negocio se ejecute. Inmediatamente después de que el método de negocio se ejecuta, el contenedor llama a `ejbStore`.

Si los métodos `ejbLoad` y `ejbStore` no pueden localizar una entidad en la base de datos, deberían arrojar la excepción `javax.ejb.NoSuchEntityException`. Esta excepción es una subclase de `EJBException`. Debido a que `EJBException` es una subclase de `RuntimeException` no es necesario incluirla en la cláusula `throws`. Cuando se lanza una excepción `NoSuchEntityException`, el contenedor la envuelve en una `RemoteException` antes de devolverla al cliente.

En la clase `SavingsAccountBean`, `ejbLoad` invoca el método `loadRow`, el cual lanza una sentencia SQL `SELECT` y asigna los datos devueltos a las variables de instancia. El método `ejbStore` llama al método privado `storeRow`, que a su vez almacena las variables de instancia en la base de datos con una sentencia SQL `UPDATE`. A continuación se muestra el código fuente de `ejbLoad` y `ejbStore`:

```
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("ejbStore: " +
            ex.getMessage());
    }
}
```



```
}
```

4.1.6 Los métodos de búsqueda

Los métodos de búsqueda (Finder) permiten a los clientes localizar beans de entidad. El programa `SavingsAccountClient` localiza beans de entidad con tres métodos de búsqueda:

```
SavingsAccount jones = home.findByPrimaryKey("836");  
...  
Collection c = home.findByLastName("Smith");  
...  
Collection c = home.findInRange(20.00, 99.00);
```

Por cada método de búsqueda disponible para un cliente, la clase de bean de entidad debe implementar un método correspondiente con el prefijo `ejbFind`. La clase `SavingsAccountBean`, por ejemplo, implementa el método `ejbFindByLastName` como sigue:

```
public Collection ejbFindByLastName(String lastName)  
    throws FinderException {  
  
    Collection result;  
  
    try {  
        result = selectByLastName(lastName);  
    } catch (Exception ex) {  
        throw new EJBException("ejbFindByLastName " +  
            ex.getMessage());  
    }  
    return result;  
}
```

Los métodos de búsqueda específicos de la aplicación, como `ejbFindByLastName` y `ejbFindInRange`, son opcionales. Pero el método `ejbFindByPrimaryKey` es obligatorio. Como su nombre implica, el método `ejbFindByPrimaryKey` acepta como argumento la clave primaria, que usa después para localizar el bean. En la clase `SavingsAccountBean`, la clave primaria es la variable `id`. Este es el código del método `ejbFindByPrimaryKey`:

```
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.");
    }
}
```

El método `ejbFindByPrimaryKey` puede parecer extraño a primera vista, debido a que usa una clave primaria como argumento y como valor devuelto. Sin embargo, hay que recordar que el cliente no llama a `ejbFindByPrimaryKey` directamente, sino que es el contenedor EJB el que llama al método. El cliente invoca el método `findByPrimaryKey`, que está definido en la interfaz `home`.

La siguiente lista resume las reglas que deben seguir los métodos `finder` que se implementan en un bean de entidad con persistencia gestionada por el bean:

- El método `ejbFindByPrimaryKey` debe implementarse obligatoriamente.
- Cualquier método de búsqueda debe comenzar con el prefijo `ejbFind`.
- El modificador de control de acceso debe ser público.
- El modificador del método no puede ser `final` o `static`.
- Los tipos de los argumentos y de los valores devueltos deben ser legales para RMI (este requisito sólo se aplica a métodos definidos en una interfaz remota --no local--).
- El tipo devuelto debe ser una clave primaria o una colección de claves primarias.

4.1.7 Los métodos de negocio

Los métodos de negocio contienen la lógica de negocio que es necesario encapsular dentro del bean de entidad. Normalmente, los métodos de negocio no acceden a la base de datos permitiendo separar la lógica de negocio de la implementación. La clase `SavingsAccountBean` contiene los siguientes métodos de negocio:

```
public void debit(BigDecimal amount)
    throws InsufficientBalanceException {

    if (balance.compareTo(amount) == -1) {
        throw new InsufficientBalanceException();
    }
    balance = balance.subtract(amount);
}

public void credit(BigDecimal amount) {
    balance = balance.add(amount);
}

public String getFirstName() {
    return firstName;
}

public String getLastName() {
    return lastName;
}
```

```
}  
  
public BigDecimal getBalance() {  
    return balance;  
}
```

El programa cliente `SavingsAccountClient` invoca los métodos de negocio de la siguiente forma:

```
BigDecimal zeroAmount = new BigDecimal("0.00");  
SavingsAccount duke = home.create("123", "Duke", "Earl",  
zeroAmount);  
...  
duke.credit(new BigDecimal("88.50"));  
duke.debit(new BigDecimal("20.25"));  
BigDecimal balance = duke.getBalance();
```

Los requisitos de las firmas de un método de negocio son los mismos tanto para beans de sesión como para los beans de entidad:

- El nombre del método no debe entrar en conflicto con un nombre de método definido en la arquitectura EJB. Por ejemplo, no puede existir un método de negocio llamado `ejbCreate` o `ejbActivate`.
- El modificador de control de acceso debe ser `public`.
- el modificador del método no puede ser `final` o `static`.
- Los argumentos y tipos devueltos deben ser legales para RMI. Este requisito se aplica sólo a los métodos definidos en una interfaz `home` remota, no local.

La cláusula `throws` puede incluir las excepciones propias de la aplicación. El método `debit`, por ejemplo, arroja la excepción `InsufficientBalanceException`. Para indicar un problema de nivel de sistema, un método de negocio debería arrojar la excepción `javax.ejb.EJBException`.

4.1.8 Los métodos Home

Un método `home` contiene la lógica de negocio que se aplica a todos los beans de entidad de una clase particular. Sin embargo, la lógica en un método de negocio se aplica a un bean de entidad particular, una instancia con una identidad única. Durante la invocación de un método `home`, la instancia o bien no tiene una identidad única o bien no tiene un estado con el que representar un objeto de negocio. Por ello, un método `home` no debe acceder a las variables de instancia del bean. En el caso de la persistencia gestionada por el contenedor, un método `home` no puede tampoco acceder a relaciones.

Normalmente, un método `home` localiza una colección de instancias de bean e invoca métodos de negocio de forma iterativa a todas las instancias. En el caso de la clase `SavingsAccountBean`, un ejemplo de método `home` es `ejbHomeChargeLowBalance`. Este método aplica un cargo por servicio a todas las cuentas con balances menores de una determinada cantidad. El método localiza estas cuentas invocando el método `findInRange`. Al tiempo que itera a

través de la colección de instancias `SavingAccounts`, el método `ejbHomeChargeForLowBalance` chequea el balance e invoca el método `debit`. He aquí el código fuente de este método:

```
public void ejbHomeChargeForLowBalance(
    BigDecimal minimumBalance, BigDecimal charge)
    throws InsufficientBalanceException {

    try {
        SavingsAccountHome home =
            (SavingsAccountHome)context.getEJBHome();
        Collection c = home.findInRange(new BigDecimal("0.00"),
            minimumBalance.subtract(new BigDecimal("0.01")));

        Iterator i = c.iterator();

        while (i.hasNext()) {
            SavingsAccount account = (SavingsAccount)i.next();
            if (account.getBalance().compareTo(charge) == 1) {
                account.debit(charge);
            }
        }
    } catch (Exception ex) {
        throw new EJBException("ejbHomeChargeForLowBalance: " +
            ex.getMessage());
    }
}
```

La interfaz `home` define un método correspondiente llamado `chargeForLowBalance`. Ya que esta interfaz `home` es la que proporciona la vista del cliente, el programa `SavingsAccountClient` invoca el método `home` como sigue:

```
SavingsAccountHome home;
...
home.chargeForLowBalance(new BigDecimal("10.00"),
    new BigDecimal("1.00"));
```

En una clase de bean de entidad, la implementación de un método `home` debe cumplir estas reglas:

- El nombre de un método `home` debe comenzar con el prefijo `ejbHome`
- El modificador de control de acceso debe ser `public`.
- El modificador del método no puede ser `static`.

La cláusula `throws` puede incluir excepciones específicas de la aplicación y no debe arrojar la excepción `java.rmi.RemoteException`.

4.1.9 Llamadas a la base de datos

La siguiente tabla resume las llamadas de acceso a la base de datos en la clase `SavingsAccountBean`. Los métodos de negocio de la clase `SavingsAccountBean` no aparecen en la tabla porque no accede a la base de datos. En su lugar, estos métodos de negocio actualizan las variables de instancias y la modificación de la base de datos sucede cuando el contenedor EJB llama a `ejbStore`. Se podría haber escogido otro enfoque, en el que los propios métodos de negocio acceden a la base de datos. La decisión de hacerlo de una forma u otra tiene que ver con la política de uso de transacciones.

Método	Sentencia SQL
<code>ejbCreate</code>	INSERT un registro nuevo Devuelve la clave primaria al contenedor
<code>ejbFindByPrimaryKey</code>	SELECT el registro con una clave primaria específica Devuelve la clave primaria al contenedor
<code>ejbFindXXX</code> (<code>ByLastName</code> , <code>InRange</code>)	SELECT uno o más registros Devuelve al contenedor la clave primaria o una colección de claves primarias
<code>ejbHomeXXX</code> (<code>ChargeForLowBalance</code>)	SELECT o UPDATE no ligado a una clave primaria específica
<code>ejbLoad</code>	SELECT (refresca variables de instancia con valores de base de datos)
<code>ejbRemove</code>	DELETE un registro
<code>ejbStore</code>	UPDATE (almacena variables de instancia en base de datos)

Antes de acceder a la base de datos hay que conectarse a ella. Eso se hace en el método `setEntityContext`. El nombre de la base de datos a la que conectarse se define como una referencia mediante nombre JNDI ENC. En el fichero de configuración de despliegue se define a qué nombre JNDI real corresponde esta referencia. El método `unsetEntityContext` se encarga de cerrar la conexión con la base de datos.

```
private String dbName = "java:comp/env/jdbc/SavingsAccountDB";
```

```

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

private void makeConnection() throws NamingException, SQLException {
    InitialContext ic = new InitialContext();
    DataSource ds = (DataSource) ic.lookup(dbName);
    con = ds.getConnection();
}

public void unsetEntityContext()
    try {
        con.close();
    } catch (SQLException ex) {
        throw new EJBException("unsetEntityContext: " +
ex.getMessage());
    }
}

```

Para acceder a la base de datos se han definido unos métodos privados, como insertRow o deleteRow. Por ejemplo, el código fuente de insertRow es el siguiente:

```

private void insertRow (String id, String firstName, String lastName,
    BigDecimal balance) throws SQLException {
    String insertStatement =
        "insert into savingsaccount values ( ? , ? , ? , ? )";
    PreparedStatement prepStmt =
        con.prepareStatement(insertStatement);

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

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

```

4.2 Interfaz Home

La interfaz home define los métodos que permiten a un cliente crear y encontrar un bean de entidad.

La interfaz SavingsAccountHome sigue a continuación:

```

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

```

```
public interface SavingsAccountHome extends EJBHome {
    public SavingsAccount create(String id, String firstName,
        String lastName, BigDecimal balance)
        throws RemoteException, CreateException;

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

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

    public Collection findInRange(BigDecimal low,
        BigDecimal high)
        throws FinderException, RemoteException;

    public void chargeForLowBalance(BigDecimal minimumBalance,
        BigDecimal charge)
        throws InsufficientBalanceException, RemoteException;
}
```

4.2.1 Definición de métodos create

Cada método `create` en la interfaz home debe cumplir los siguientes requisitos:

- Tiene la misma signatura que sus métodos correspondientes `ejbCreate` en la clase bean enterprise.
- Devuelve el tipo de la interfaz remota del enterprise bean
- La cláusula `throws` incluye las excepciones especificadas por la cláusula `throws` en los métodos correspondientes `ejbCreate` y `ejbPostCreate`.
- La cláusula `throws` incluye la excepción `javax.ejb.CreateException`.
- Si el método se define en una interfaz home remota -no local-, entonces la cláusula `throws` incluye la excepción `RemoteException`.

4.2.2 Definición de los métodos de búsqueda

Cada método de búsqueda en la interfaz home corresponde con un método de búsqueda en la clase de bean de entidad. El nombre del método `finder` en la interfaz home comienza con `find`, mientras que el correspondiente nombre en la clase de bean de entidad comienza por `ejbFind`. Por ejemplo, la clase `SavingsAccountHome` define el método `findByLastName`, y la clase `SavingsAccountBean` implementa el método `ejbFindByLastName`. Las reglas para definir las firmas de los métodos `finder` son similares a las de los métodos home:

- Tiene la misma signatura que sus métodos correspondientes en la clase bean enterprise.
- Devuelve el tipo de la interfaz remota del enterprise bean, o una colección de estos tipos
- La cláusula `throws` incluye las excepciones especificadas por la cláusula `throws` en los métodos correspondientes.
- La cláusula `throws` incluye la excepción `javax.ejb.FinderException`.

- Si el método se define en una interfaz home remota -no local-, entonces la cláusula throws incluye la excepción `RemoteException`.

4.2.3 Definiciones de métodos Home

Cada definición de método home en la interfaz home corresponde a un método home en la clase de bean de entidad. En la interfaz home, el nombre del método es arbitrario, siempre que no comience por create o find. En la clase bean, el nombre del método correspondiente es el mismo precedido por `ejbHome`. Por ejemplo, en la clase `SavingsAccountBean`, el nombre es `ejbHomeChargeForLowBalance`, mientras que en la interfaz `SavingsAccountHome` el nombre es `chargeForLowBalance`.

La signatura del método home debe seguir las mismas reglas especificadas para métodos finder en la sección previa (excepto que un método home no arroja una excepción `FinderException`).

4.3 Interfaz remota

La interfaz remota extiende `javax.ejb.EJBObject` y define los métodos de negocio que un cliente remoto puede invocar. He aquí la interfaz remota `SavingsAccountRemote`:

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.math.BigDecimal;

public interface SavingsAccountRemote extends EJBObject {

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

    public void credit(BigDecimal amount)
        throws RemoteException;

    public String getFirstName()
        throws RemoteException;

    public String getLastName()
        throws RemoteException;

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

Los requisitos para las definiciones de métodos en una interfaz remota son los mismos que hemos visto en beans de sesión y beans de entidad:

- Cada método en la interfaz remota debe emparejar un método en el bean de empresa.
- Devuelve el tipo de la interfaz remota del enterprise bean, o una colección de estos tipos

- La cláusula `throws` incluye las excepciones especificadas por la cláusula `throws` en los métodos correspondientes.
- La cláusula `throws` incluye la excepción `javax.ejb.FinderException`.
- Si el método se define en una interfaz home remota -no local-, entonces la cláusula `throws` incluye la excepción `RemoteException`.

Una interfaz local tiene los mismos requisitos, con las siguientes excepciones:

- Los tipos de los argumentos y de los valores devueltos no tienen por qué ser válidos RMI.
- La cláusula `throws` no incluye `java.rmi.RemoteException`

4.4 El descriptor de despliegue

En la primera parte del fichero de descripción del despliegue se definen las características básicas del despliegue del bean

```
<enterprise-beans>
  <entity>
    <display-name>SavingsAccountEJB</display-name>
    <ejb-name>SavingsAccountEJB</ejb-name>
    <home>SavingsAccountHome</home>
    <remote>SavingsAccount</remote>
    <ejb-class>SavingsAccountBean</ejb-class>
    <persistence-type>Bean</persistence-type>
    <prim-key-class>java.lang.String</prim-key-class>
    <reentrant>False</reentrant>
    <security-identity>
      <description></description>
      <use-caller-identity></use-caller-identity>
    </security-identity>
    <resource-ref>
      <res-ref-name>jdbc/SavingsAccountDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
      <res-sharing-scope>Shareable</res-sharing-scope>
    </resource-ref>
  </entity>
</enterprise-beans>
```

La segunda parte del fichero de descripción del despliegue contiene las características de seguridad y transacciones de cada uno de los métodos del EJB

```
<assembly-descriptor>
  <method-permission>
    <method>
      <ejb-name>SavingsAccountEJB</ejb-name>
      <method-intf>Home</method-intf>
      <method-name>create</method-name>
      <method-params>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.String</method-param>
        <method-param>java.lang.String</method-param>
        <method-param>java.math.BigDecimal</method-param>
      </method-params>
    </method>
  </method-permission>
</assembly-descriptor>
```

```
        </method-params>
    </method>
    ...
<container-transaction>
    <method>
        ...
    </method>
    <trans-attribute>Required</trans-attribute>
</assembly-descriptor>
```

Tema 5: Beans de entidad con persistencia gestionada por el contenedor

5.1 Introducción

Como hemos visto en el tema anterior, en los beans de entidad con persistencia gestionada por el bean hay que declarar en el código fuente del bean:

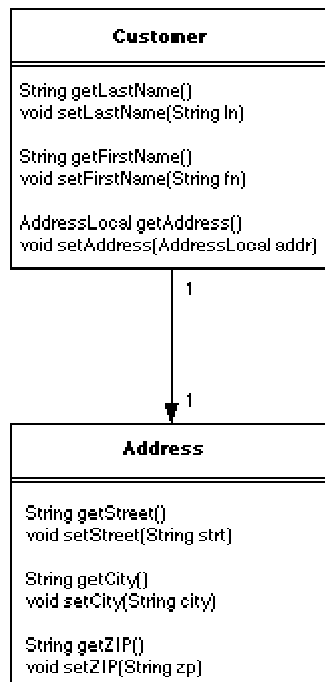
- funciones de conexión a la base de datos (abrir y cerrar base de datos)
- funciones para hacer operaciones `SELECT`, `INSERT`, `UPDATE` sobre la tabla de la base de datos que guarda la información del bean
- nombre de la base de datos
- nombre de la tabla donde se almacena el bean

En este tema describiremos los beans de entidad con persistencia gestionada por el contenedor. En ellos todas las funciones que antes había que codificar se definen de forma declarativa en el fichero de descripción del despliegue del bean. Veremos dos ejemplos -`Customer` y `Address`- con los que explicaremos cómo se definen los beans con persistencia gestionada por el contenedor y cómo funcionan en tiempo de ejecución. Veremos en el tema siguiente la definición de relaciones entre beans, que también son gestionadas automáticamente por el contenedor.

La persistencia gestionada por el contenedor se ha introducido en la especificación 2.0 de la arquitectura EJB. En la literatura en inglés se suele referir a ella como CMP 2.0 (Container Managed Persistence).

5.1.1 El modelo abstracto de programación

En CMP 2.0 el contenedor gestiona automáticamente el estado de los beans de entidad. El contenedor se encarga de las transacciones y de la comunicación con la base de datos. El desarrollador del bean describe los atributos y las relaciones de un bean de entidad usando campos de persistencia virtuales y campos de relación. Se llaman campos virtuales porque el desarrollador no declara estos campos explícitamente, sino que se definen métodos abstractos de acceso (`get` y `set`) en la clase bean de entidad. La implementación de estos métodos se genera en tiempo de despliegue por las herramientas del servidor de aplicaciones. Es importante recordar que los términos campo de relación y campo de persistencia se refieren a los métodos abstractos de acceso y no a los campos reales declarados en las clases.



En la figura anterior, el EJB `Customer` tiene seis métodos de acceso. Los primeros cuatro leen y actualizan los apellidos y el nombre del cliente. Estos son ejemplos de campos de persistencia: atributos directos del bean de entidad. Los últimos dos son métodos de acceso que obtienen y definen referencias al EJB `Address` a través de su interfaz local, `AddressLocal`. Este es un ejemplo de un campo de relación llamado `address`.

5.1.2 El esquema abstracto de persistencia

Las clases bean de entidad en CMP 2.0 se definen usando métodos de acceso que representan campos virtuales de relación y de persistencia. Como se ha mencionado antes, los campos reales no se declaran en las clases bean de entidad. En su lugar, las características de estos campos se describen en detalle en el fichero XML descriptor del despliegue que usa el bean. El esquema abstracto de persistencia es un conjunto de elementos XML que describen los campos de relación y los campos de persistencia. Junto con el modelo abstracto de programación (los métodos abstractos de acceso) y alguna ayuda del desplegador, la herramienta del contenedor tendrá suficiente información para hacer corresponder la entidad y sus relaciones con otros beans en la base de datos.

5.1.3 Herramientas del contenedor y persistencia

Una de las responsabilidades del vendedor de las herramientas de despliegue y del contenedor es la generación de implementaciones concretas de los beans de entidad abstractos. Las clases concretas generadas por el contenedor se llaman clases de persistencia. Las instancias de las clases de persistencia son responsables de trabajar con el contenedor para leer y escribir datos entre el bean de entidad y la base de datos en tiempo de ejecución. Una vez que las clases de persistencia se generan, pueden ser desplegadas en el contenedor EJB. El contenedor informa a las instancias de persistencia cuándo

deben leer y escribir en la base de datos. Las instancias de persistencia realizan la lectura y escritura de una forma optimizada para la base de datos que se está usando.

Las clases de persistencia incluirán lógica de acceso a la base de datos adaptada a una base de datos particular. por ejemplo, un producto EJB podría proporcionar un contenedor que pueda hacer corresponder un bean de entidad a una base de datos específica como una base de datos relacional Oracle o la base de datos de objetos POET. Esta especificidad permite a las clases de persistencia emplear optimizaciones personalizadas a bases de datos particulares.

La herramienta del contenedor genera toda la lógica de acceso a la base de datos en tiempo de despliegue y la embebe en las clases de persistencia. Esto significa que los desarrolladores no tienen que escribir la lógica de acceso a la base de datos ellos mismos, lo que ahorra un montón de trabajo; además este mecanismo puede resultar en un mejor rendimiento en los beans de entidad porque la implementación está optimizada. Un desarrollador de bean de entidad nunca tendrá que tratar con el código de acceso a la base de datos si está trabajando con entidades CMP 2.0. De hecho, ni siquiera tendrá acceso a las clases de persistencia que contienen esa lógica, porque se generan de forma automática por la herramienta. En la mayoría de los casos este código fuente no está disponible para los desarrolladores de beans.

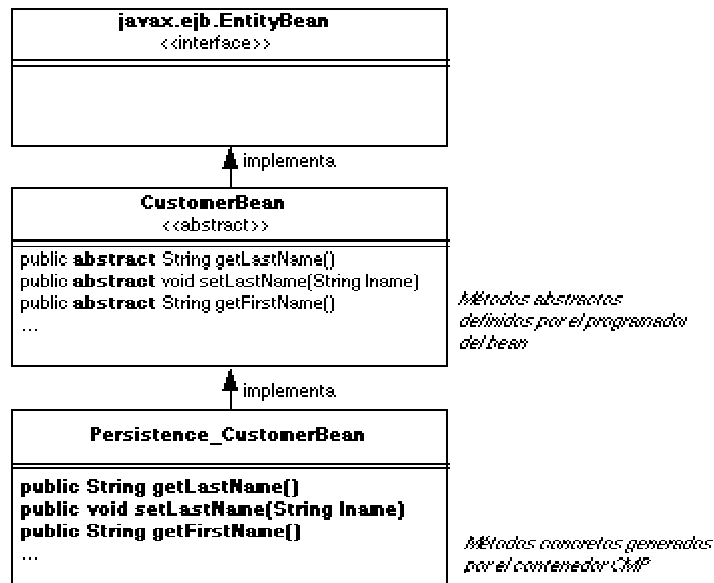
5.2 El EJB `Customer`

En el siguiente ejemplo, desarrollaremos un simple bean de entidad CMP 2.0 - el EJB `Customer`. Conforme avancemos con el tema, el ejemplo se hará más complejo para ilustrar los conceptos comentados en cada sección. Este ejemplo supone que estas usando una base de datos relacional. Necesitaremos una tabla llamada `CUSTOMER` a partir de la cual obtener los datos de clientes. La definición de la tabla se hace en SQL con la siguiente sentencia:

```
CREATE TABLE CUSTOMER
(ID INT PRIMARY KEY NOT NULL,
LAST_NAME CHAR(20),
FIRST_NAME CHAR(20))
```

5.2.2 La clase `CustomerBean`

La clase `CustomerBean` es una clase abstracta a partir de la cual la herramienta del contenedor generará una implementación concreta que se ejecutará en el contenedor EJB. El mecanismo por el que la herramienta del contenedor genera una clase entidad persistente a partir de la clase bean de entidad varía, pero la mayoría de fabricantes generan una subclase de la clase abstracta proporcionada por el desarrollador del bean (ver figura siguiente).



La clase abstracta del bean debe declarar métodos de acceso (`set` y `get`) para cada campo y relación de persistencia definidos en el esquema abstracto de persistencia del descriptor de despliegue. La herramienta del contenedor necesita tanto los métodos abstractos de acceso (definidos en la clase bean) como los elementos XML del descriptor de despliegue para describir completamente el esquema de persistencia del bean.

He aquí una definición muy simple de la clase `CustomerBean`.

```

import javax.ejb.EntityContext;

public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id) {
        setId(id);
        return null;
    }

    public void ejbPostCreate(Integer id) {}

    // abstract accessor methods

    public abstract Integer getId();
    public abstract void setId(Integer id);

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

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

    // standard callback methods

    public void setEntityContext(EntityContext ec) {}
    public void unsetEntityContext() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbActivate() {}
  
```

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

La clase `CustomerBean` se define como una clase abstracta. Esto es requerido por CMP 2.0 para reforzar la idea de que el `CustomerBean` no se despliega directamente en el sistema contenedor. Ya que las clases abstractas no pueden instanciarse, la herramienta del contenedor debe definir una subclase que es la que se instala en el contenedor. También se declaran abstractos los propios métodos de acceso, lo cual hace que la herramienta del contenedor los deba implementar.

Al igual que en la persistencia gestionada por el bean, la clase `CustomerBean` implementa la interfaz `javax.ejb.EntityBean`, que define bastantes métodos de callback, incluyendo `setEntityContext`, `unsetEntityContext`, `ejbLoad`, `ejbStore`, `ejbActivate`, `ejbPassivate` y `ejbRemove`.

El primer método en la clase bean de entidad es `ejbCreate`, que toma una referencia a un objeto `Integer` como único argumento. El método `ejbCreate` se llama cuando el cliente remoto invoca el método `create` en la interfaz home del bean de entidad. Este funcionamiento es similar al que ya hemos visto en el tema anterior. El método `ejbCreate` es responsable de la inicialización de los campos de persistencia antes de que el bean de entidad sea creado. En este ejemplo, el método `ejbCreate` se usa para inicializar el campo de persistencia `id`, que está representado por los métodos de acceso `setId/getId`.

El tipo devuelto por el método `ejbCreate` es un `Integer`, que es la clave primaria del bean de entidad. La clave primaria es un identificador único que puede tomar distintas formas, incluyendo un wrapper de tipos primitivos y clases definidas por el desarrollador. En este caso, la clave primaria (el `Integer`) se hace corresponder con el campo `ID` en la tabla `CUSTOMER`. Esto se verá con más claridad cuando definamos el descriptor de despliegue XML. Aunque el tipo devuelto por el método `ejbCreate` es la clave primaria, el valor realmente devuelto por el método `ejbCreate` es `null`. El contenedor EJB y la clase de persistencia extraerán la clave primaria del bean cuando sea necesario.

El método `ejbPostCreate` se usa para realizar una inicialización después de que el bean de entidad haya sido inicializado, pero antes de que sirva cualquier petición de ningún cliente. Este método normalmente se usa para realizar algunas operaciones con los campos de relación del bean de entidad, lo cual sólo es posible después de que el método `ejbCreate` del bean haya sido invocado y el bean esté en la base de datos. Para cada método `ejbCreate` debe haber un método correspondiente `ejbPostCreate` con los mismos argumentos pero devolviendo `void`. Este funcionamiento en pareja de los métodos `ejbCreate` y `ejbPostCreate` asegura que el contenedor llama a ambos métodos correctamente. Más adelante veremos algún ejemplo del uso de `ejbPostCreate`; por ahora dejaremos su implementación vacía.

Los métodos abstractos de acceso (`setLastName`, `getLastName`, `setFirstName`, `getFirstName`), representan los campos de persistencia de la clase `CustomerBean`. Estos métodos se definen como abstractos sin cuerpos de método. Como ya mencionamos, cuando el bean se procesa por una herramienta del contenedor, estos métodos se implementarán por una clase de persistencia basada en el esquema abstracto de persistencia (los elementos del descriptor de despliegue XML), el contenedor EJB y la base de datos usados. Básicamente, estos métodos obtienen y actualizan los valores en la base de datos y no se implementan por el desarrollador del bean.

5.2.3 La interfaz remota

Necesitaremos una interfaz remota `Customer` para el EJB `Customer`, ya que se accederá al bean por parte de clientes fuera del sistema del contenedor. La interfaz remota define, como siempre, los métodos que el cliente podrá usar para interactuar con las instancias del bean de entidad. La interfaz remota debería definir los métodos que modelan los aspectos públicos del concepto de negocio que se están modelando; esto es, aquellas conductas y datos que debería exponerse hacia las aplicaciones cliente. He aquí la interfaz remota `Customer`:

```
import java.rmi.RemoteException;

public interface Customer extends javax.ejb.EJBObject {

    public String getLastName() throws RemoteException;
    public void setLastName(String lname) throws RemoteException;

    public String getFirstName() throws RemoteException;
    public void setFirstName(String fname) throws RemoteException;
}
```

Las firmas de todos los métodos definidos en la interfaz remota debe corresponderse con las firmas de los métodos definidos en la clase bean. En este caso, los métodos de acceso en la interfaz `Customer` se corresponden con los métodos de acceso a los campos de persistencia en la clase `CustomerBean`. Cuando hacemos esto, el cliente tiene acceso directo a los campos de persistencia del bean de entidad.

Sin embargo, no se requiere que se emparejen todos los métodos abstractos de acceso con métodos en la interfaz remota. De hecho, se recomienda que la interfaz remota sea lo más independiente posible del modelo de programación abstracta.

La especificación EJB prohíbe que los métodos remotos se emparejen con los campos de relación, que definen el acceso a otros beans de entidad. Además, los métodos remotos no deben modificar ningún campo de persistencia gestionado por el contenedor que sea parte de la clave primaria de un bean de entidad. En este caso, la interfaz remota no define un método `setId`, ya que esto permitiría a los clientes modificar la clave primaria.

5.2.4 La interfaz Home remota

La interfaz home remota de cualquier bean de entidad se usa para crear, localizar y eliminar entidades del contenedor EJB. Cada tipo de bean de entidad debe tener su propia interfaz home remota, su interfaz home local, o ambas. De la misma forma que vimos en el tema anterior, las interfaces home definen tres clases de métodos home: métodos home de negocio, métodos de creación y métodos de búsqueda. Los métodos `create()` actúan como constructores remotos y definen cómo se crean nuevas instancias del bean de entidad. En este ejemplo, proporcionaremos un único método `create`, que se empareja con el método `ejbCreate` definido en la clase de bean de entidad. El método de búsqueda que vamos a definir se usa para localizar una instancia específica del bean `Customer` usando la clave primaria como único identificador.

He aquí la definición de la interfaz `CustomerHome` remota:

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

public interface CustomerHome extends javax.ejb.EJBHome {

    public Customer create(Integer id)
        throws CreateException, RemoteException;

    public Customer findByPrimaryKey(Integer id)
        throws FinderException, RemoteException;

}
```

Al igual que vimos con los beans de entidad con persistencia gestionada por el bean, es posible definir distintos métodos remotos `create` con distintas firmas siempre que en la clase bean se definan los métodos `ejbCreate` que emparejen con estas firmas. Incluso es posible definir distintos métodos `create` con la misma firma añadiendo un sufijo al nombre del método `create`. Así, por ejemplo, podríamos tener dos métodos `create` que toman como argumentos la clave primaria y un `String`. El argumento `String` podría ser el nombre en un caso o el DNI en otro:

```
public interface CustomerHom extends javax.ejb.EJBHome {

    public CustomerRemote createWithName (Integer id, String
socialSecurityNumber)

        throws CreateException, RemoteException;

    public CustomerRemote createWithDNI(Integer id, String
taxIdentificationNumber)
        throws CreateException, RemoteException;

    public CustomerRemote findByPrimaryKey(Integer id)
        throws FinderException, RemoteException;

}
```

```
public findBySocialSecurityNumber(String taxId)
    throws FinderException, RemoteException;

}
```

Los sufijos son útiles cuando necesitamos métodos `create` que sean más descriptivos o necesitamos diferenciar los métodos de creación para permitir la posterior sobrecarga del método. Cada método `createXXX` debe tener un método correspondiente `ejbCreateXXX` en la clase bean. Para mantener el ejemplo sencillo vamos a usar sólo un único método `create` y no vamos a definir sufijos. Los requisitos de los métodos `create` son los mismos que en los beans con persistencia gestionada por el bean:

- Tiene la misma signatura que sus métodos correspondientes `ejbCreate` en la clase bean enterprise.
- Devuelve el tipo de la interfaz remota del enterprise bean
- La cláusula `throws` incluye las excepciones especificadas por la cláusula `throws` en los métodos correspondientes `ejbCreate` y `ejbPostCreate`.
- La cláusula `throws` incluye la excepción `javax.ejb.CreateException`.
- Si el método se define en una interfaz home remota -no local-, entonces la cláusula `throws` incluye la excepción `RemoteException`.

Las interfaces home remotas deben definir un método `findByPrimaryKey` que tome como único argumento el tipo primario. A diferencia del tema anterior, no se debe especificar en la clase bean el método de búsqueda correspondiente, sino que el método que implemente la búsqueda será generado automáticamente por la herramienta de despliegue. En tiempo de ejecución el método `findByPrimaryKey` localizará automáticamente y devolverá una referencia remota al bean de entidad con la clave primaria correspondiente.

El desarrollador del bean también puede declarar otros métodos de búsqueda. En este caso se ha incluido el método `findBySocialSecurityNumber(String id)` que localiza a aquellos `Customer` con un número determinado de identificación. Estos tipos de métodos de búsqueda también son implementados automáticamente por la herramienta de despliegue basándose en la signatura del método y en una sentencia EJB QL, similar a SQL pero específica de los EJB, que se incluye en el descriptor de despliegue.

5.2.5 El descriptor de despliegue XML

Los beans de entidad CMP 2.0 deben empaquetarse para el despliegue junto con un descriptor de despliegue XML que describe el bean y su esquema abstracto de persistencia. En la mayoría de los casos, el desarrollador del bean no usará directamente el fichero XML, sino que usará una herramienta visual proporcionada por el contenedor de aplicaciones. Más adelante veremos algún ejemplo de herramienta visual, pero ahora detallaremos los campos XML para poder entender completamente la especificación y arquitectura.

El descriptor de despliegue XML para nuestro EJB `Customer` contiene muchos elementos que ya hemos visto previamente. Aparece a continuación:

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>CustomerEJB</ejb-name>
      <home>CustomerHomeRemote</home>
      <remote>CustomerRemote</remote>
      <ejb-class>CustomerBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>Customer</abstract-schema-name>
        <cmp-field><field-name>id</field-name></cmp-field>
        <cmp-field><field-name>lastName</field-name></cmp-field>
        <cmp-field><field-name>firstName</field-name></cmp-field>
        <cmp-field><field-name>hasGoodCredit</field-name></cmp-field>
      <primkey-field>id</primkey-field>
      <query>
        <query-method>
          <method-name>findByLastName</method-name>
          <method-params>
            <method-param>java.lang.String</method-param>
          </method-params>
        </query-method>
      <ejb-ql>
        SELECT OBJECT(c) FROM Customer c
        WHERE c.lastName = ?1
      </ejb-ql>
    </query>
  </entity>
</enterprise-beans>
```

Los primeros elementos, que declaran el nombre del EJB `Customer` (`CustomerEJB`) así como sus clases `home`, remotas y `bean`, deberían sernos ya familiares. El elemento `<security-identity>` lo describiremos en el tema en el que hablemos de seguridad. También trataremos más adelante el elemento `<assembly-descriptor>`, con el que se definen atributos de seguridad y de transacciones del `bean`. En este caso, define que todos los empleados pueden acceder cualquier método del `bean` y que todos los métodos van a usar el atributo `Required` para las transacciones.

Las entidades con persistencia gestionada por el contenedor también deben declarar su tipo de persistencia, versión y si son reentrantes. Estos elementos se declaran bajo el elemento `<entity>`.

El elemento `<persistence-type>` indica al sistema contenedor si el `bean` usará una persistencia gestionada por el contenedor o una persistencia gestionada por el `bean`. En este caso se trata de persistencia gestionada por el contenedor, por lo que usamos `Container`. En el tema pasado, usábamos `Bean`.

El elemento `<cmp-version>` indica al contenedor que versión se va a usar de persistencia gestionada por el contenedor. Los contenedor EJB 2.0 deben

soportar el nuevo modelo de persistencia, así como el antiguo definido en EJB 1.1. El valor de `<cmp-version>` puede ser o bien 2.x o 1.x para las versiones EJB 2.0 y 1.1 respectivamente. Este elemento es opcional y si no se declara su valor por defecto es 2.x.

El elemento `<reentrant>` indica si se permite o no una conducta reentrante. En este caso el valor es `False`, lo que indica que el `CustomerEJB` es no reentrante. Hablaremos de reentrada en el tema 7, cuando comentemos los aspectos relacionados con la concurrencia en el acceso a los beans de entidad.

El bean de entidad también debe declarar sus campos de persistencia gestionada por el contenedor y su clave primaria. Los campos de persistencia gestionada por el contenedor son `id`, `lastName` y `firstName`, como se indican en los elementos `<cmp-field>`. Los elementos `<cmp-field>` deben tener métodos de acceso correspondientes en la clase de bean de entidad. Como se puede ver en la tabla siguiente, los valores declarados en el elemento `<field-name>` corresponden con los nombres de los métodos de acceso abstractos que hemos declarado en la clase `CustomerBean`.

Campo CMP	Método abstracto de acceso
id	public abstract Integer getId() public abstract void setId(Integer id)
lastName	public abstract String getLastName() public abstract void setLastName(String lname)
firstName	public abstract String getFirstName() public abstract void setFirstName(String fname)

CMP 2.0 obliga a que los valores `<field-name>` comiencen con una letra minúscula, y a que sus correspondientes métodos de acceso tengan la forma `get<field-name>` y `set<field-name>`, con la primera letra de `<field-name>` convertida en mayúscula. El tipo devuelto por el método `get` y el parámetro del método `set` determinan el tipo del campo `<cmp-field>`.

Finalmente, definimos la sentencia EJB-QL que va a implementar la búsqueda `findByLastName(String)`.

5.2.6 El fichero `weblogic-ejb-jar.xml`

En el fichero de descripción del despliegue propio del servidor de aplicaciones, `weblogic-ejb-jar.xml`, se describen las características del bean que son particulares del servidor de aplicaciones. En este caso se trata de la correspondencia entre los campos de persistencia gestionada por el contenedor y los campos o los objetos de datos en la base de datos.

Además, es necesario corresponder los roles de seguridad con los sujetos del realm de seguridad del servidor de aplicaciones, así como asignar un nombre

JNDI al enterprise bean. En este caso no definimos ningún tipo de seguridad. Lo veremos en el tema 7.

```
<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>CustomerEJB</ejb-name>
    <entity-descriptor>
      <entity-cache>
        <max-beans-in-cache>100</max-beans-in-cache>
      </entity-cache>
      <persistence>
        <persistence-use>
          <type-identifier>WebLogic_CMP_RDBMS</type-identifier>
          <type-version>6.0</type-version>
          <type-storage>META-INF/weblogic-cmp-rdbms-jar.xml</type-
storage>
        </persistence-use>
      </persistence>
    </entity-descriptor>
    <jndi-name>CustomerHomeRemote</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

5.2.7 El fichero weblogic-cmp-rbms-jar.xml

Todos los beans de entidad con persistencia gestionada por el contenedor necesitan relacionar la base de datos con los campos de persistencia. Esto se realiza en un fichero de descripción del despliegue propio del servidor de aplicaciones. En este caso se llama `weblogic-cmp-rdbms-jar.xml`.

```
<weblogic-rdbms-jar>
  <weblogic-rdbms-bean>
    <ejb-name>CustomerEJB</ejb-name>
    <data-source-name>examples-datasource-demoPool</data-source-name>
    <table-map>
      <table-name>CUSTOMER</table-name>
      <field-map>
        <cmp-field>id</cmp-field>
        <dbms-column>ID</dbms-column>
      </field-map>
      <field-map>
        <cmp-field>lastName</cmp-field>
        <dbms-column>LAST_NAME</dbms-column>
      </field-map>
      <field-map>
        <cmp-field>firstName</cmp-field>
        <dbms-column>FIRST_NAME</dbms-column>
      </field-map>
      <field-map>
        <cmp-field>hasGoodCredit</cmp-field>
        <dbms-column>HAS_GOOD_CREDIT</dbms-column>
      </field-map>
    </table-map>
  </weblogic-rdbms-bean>
</weblogic-rdbms-jar>
```

5.2.7 El fichero EJB JAR

Ahora que ya tenemos definidas las interfaces, la clase bean y el descriptor de despliegue, estamos listos para empaquetar el bean y dejarlo preparado para el despliegue. Como ya hemos visto en otros ejemplos, lo único que tenemos que hacer es usar el comando jar para comprimir todos los ficheros con la estructura adecuada:

```
% jar cf customer.jar *.class META-INF/*.xml
```

Una vez creado el fichero JAR deberemos preparar este fichero para el despliegue en servidor de aplicaciones.

5.2.8 La aplicación cliente

La aplicación cliente es un cliente remoto del `CustomerEJB` que creará algunos clientes, los encontrará y después los eliminará. He aquí el código fuente de la aplicación `Client`:

```
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Properties;

public class Client {
    public static void main(String [] args) throws Exception {
        //obtengo el home remoto
        Context jndiContext = getInitialContext();
        Object obj=jndiContext.lookup("CustomerHomeRemote");
        CustomerHomeRemote home = (CustomerHomeRemote)

        javax.rmi.PortableRemoteObject.narrow(obj,CustomerHomeRemote.class);

        //creo Customers
        for(int i =0;i <args.length;i++){
            Integer primaryKey =new Integer(args [ i ]);
            String firstName = args [ i ];
            String lastName = args [ i ];
            CustomerRemote customer = home.create(primaryKey);
            customer.setFirstName(firstName);
            customer.setLastName(lastName);
        }

        //busco y encuentro Customer
        for(int i =0;i <args.length;i++){
            Integer primaryKey = new Integer(args [i ]);
            CustomerRemote customer = home.findByPrimaryKey(primaryKey);
            String lastName = customer.getLastName();
            String firstName = customer.getFirstName();
            System.out.print(primaryKey+"=");
            System.out.println(firstName+" "+lastName);

            //remove Customer
            customer.remove();
        }
    }
}
```

```
public static Context getInitialContext(  
    throws javax.naming.NamingException {  
    Properties p =new Properties();  
    //...Specify the JNDI properties specific to the vendor.  
    return new javax.naming.InitialContext(p);  
}  
}
```

5.3 Campos de persistencia

Los campos de persistencia gestionados por el contenedor (campos CMP en inglés) son campos virtuales cuyos valores se corresponden directamente con la base de datos. Los campos de persistencia pueden ser tipos serializables Java y tipos primitivos Java.

Los tipos serializables Java pueden ser cualquier clase que implemente la interfaz `java.io.Serializable`. La mayoría de herramientas de despliegue manejan los tipos `java.lang.String`, `java.util.Date` y los wrappers primitivos (`Byte`, `Boolean`, `Short`, `Integer`, `Long`, `Double` y `Float`) fácilmente, porque estos tipos de objetos son parte del núcleo de Java y se corresponden de forma natural con los campos de bases de datos relacionales.

El bean `CustomerEJB` declara tres campos serializables, `id`, `lastName` y `firstName`, que se corresponden de forma natural con los campos `INT` y `CHAR` de la tabla `CUSTOMER` en la base de datos.

Los objetos serializables siempre se devuelven como copias y no como referencias, por lo que la modificación de un objeto serializable no afectará a su valor en la base de datos. El valor debe actualizarse usando el método abstracto `set<field-name>`.

También se permite que sean campos CMP los tipos primitivos (`byte`, `short`, `int`, `long`, `double`, `float` y `boolean`). Estos tipos se pueden corresponder fácilmente con la base de datos y todas las herramientas de despliegue los soportan. Como ejemplo, el bean `CustomerEJB` debería haber declarado un `boolean` que represente si es adecuado conceder un crédito al cliente:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {  
  
    public Integer ejbCreate(Integer id){  
        setId(id);  
        return null;  
    }  
  
    // metodos abstractos de acceso  
  
    public abstract boolean getHasGoodCredit();  
    public abstract void setHasGoodCredit(boolean creditRating);  
}
```

Tema 6: Relaciones entre beans de entidad

Una importante característica de CMP 2.0 es la posibilidad de definir campos de relación entre beans. En este capítulo vamos a ver como ejemplo una relación uno-a-uno unidireccional entre los EJBs `Customer` y `Address`. El `Customer` tendrá una referencia a el `Address`, pero el `Address` no podrá referenciar hacia atrás el `Customer`. Son posibles también otras relaciones. Por ejemplo, cada `Address` podría referenciar su `Customer` con lo que tendríamos un ejemplo de una relación bidireccional uno-a-uno, en la que ambos EJBs mantienen referencias al otro. Además de relaciones uno-a-uno, los beans de entidad pueden tener relaciones uno-a-muchos, muchos-a-uno y muchos-a-muchos.

Existen en total siete tipo de relaciones posibles. Por un lado, tenemos cuatro combinaciones de cardinalidad: uno-a-uno, uno-a-muchos, muchos-a-uno y muchos-a-muchos. Y por otro lado tenemos dos posibles direcciones de la relación: unidireccional y bidireccional. Lo cual nos lleva a ocho posibles relaciones. Pero la relación bidireccional uno-a-muchos es obviamente idéntica a la relación muchos-a-uno, lo que nos deja con siete posibles relaciones. La mejor forma de entender estas relaciones es mediante algún ejemplo. Ahí van algunos sacados del dominio de negocio de las agencias de viajes.

Uno-a-uno, unidireccional

La relación entre un cliente y una dirección. Será necesario localizar una dirección de un cliente, pero probablemente no sea necesario localizar el alumno de una dirección.

Uno-a-uno, bidireccional

La relación entre un cliente y un número de tarjeta de crédito de la universidad. Dado un cliente, será necesario buscar su número de tarjeta de crédito. Dado un número de tarjeta de crédito también será necesario buscar el cliente que la tiene.

Uno-a-muchos, unidireccional

La relación entre un cliente y un número de teléfono. Un cliente puede tener muchos números de teléfono. Será necesario localizar un número de teléfono de un cliente, pero no será normal usar alguno de esos números para localizar al cliente.

Uno-a-muchos, bidireccional

La relación entre un crucero y una reserva. Dada una reserva, será necesario localizar el crucero para el que se ha hecho la reserva. Dado un crucero, será necesario buscar todas las reservas para ese crucero.

Muchos-a-uno, unidireccional

La relación entre un crucero y un barco. Será necesario saber qué barco ha sido usado para un crucero en particular. Y muchos cruceros usarán el mismo barco a lo largo del funcionamiento del mismo.

Muchos-a-muchos, unidireccional

La relación entre una reserva y un camarote. Es posible hacer una reserva para múltiples camarotes y va a ser necesario buscar el camarote asignado a una reserva particular.

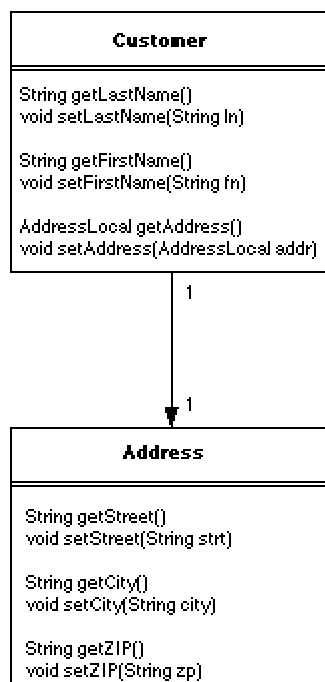
Muchos-a-muchos, bidireccional

La relación entre un crucero y un cliente. Un cliente puede hacer reservas en muchos cruceros, y cada crucero tiene muchos clientes. Será necesario buscar tanto los cruceros en los que ha embarcado un cliente como los clientes que van en un crucero dado.

En nuestro caso, la relación va a ser entre clientes y direcciones. Va a ser uno-a-uno y será unidireccional.

6.1 Definición del bean dependiente

Los beans de entidad pueden formar relaciones con otros beans de entidad. En la figura siguiente que ya mostramos en el tema anterior, el EJB `Customer` tiene una relación uno-a-uno con el EJB `Address`.



El EJB `Address` es un objeto de negocio de grano fino que debería usarse en el contexto de otro bean, lo que significa que debería tener sólo interfaces

locales. Un bean de entidad puede estar relacionado con muchos beans de entidad distintos al mismo tiempo. Por ejemplo, podríamos añadir fácilmente campos de relación para `Phone`, `CreditCard`, y otros beans de entidad al EJB `Customer`. No lo vamos a hacer, sin embargo, para mantener sencillo el ejemplo.

Siguiendo lo indicado en la figura anterior, definimos el EJB `Address` como sigue:

```
import javax.ejb.EntityContext;

public abstract class AddressBean implements javax.ejb.EntityBean {

    public Integer ejbCreateAddress(String street, String city, String
state, String zip) {
        setStreet(street);
        setCity(city);
        setState(state);
        setZip(zip);
        return null;
    }
    public void ejbPostCreateAddress(String street, String city, String
state, String zip) {
    }

    // campos de persistencia

    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract String getStreet();
    public abstract void setStreet(String street);
    public abstract String getCity();
    public abstract void setCity(String city);
    public abstract String getState();
    public abstract void setState(String state);
    public abstract String getZip();
    public abstract void setZip(String zip);

    // metodos estandard callback

    public void setEntityContext(EntityContext ec){}
    public void unsetEntityContext(){}
    public void ejbLoad(){}
    public void ejbStore(){}
    public void ejbActivate(){}
    public void ejbPassivate(){}
    public void ejbRemove(){}
}
```

La clase `AddressBean` define un método `ejbCreateAddress()`, al que se llama cuando se crea un nuevo EJB `Address`, así como bastantes campos de persistencia (`street`, `city`, `state`, and `zip`). Los campos de persistencia se representan por métodos de acceso abstractos, lo requerido para los campos de persistencia en todas las clases de bean de entidad. Estos métodos de acceso abstractos se corresponden con un conjunto de elementos XML definidos en el fichero de descripción del despliegue. En tiempo de despliegue

la herramienta de despliegue del contenedor hará corresponder los campos de persistencia del EJB `Customer` y del EJB `Address` con la base de datos. Esto significa que debe haber una tabla en nuestra base de datos relacional que contenga las columnas que emparejan con los campos de persistencia en el EJB `Address`. En este ejemplo, usaremos una tabla `ADDRESS` separada para almacenar la información sobre direcciones, pero los datos podrían haberse declarado fácilmente en la otra tabla:

```
CREATE TABLE ADDRESS (  
  ID INT PRIMARY KEY NOT NULL,  
  STREET CHAR(40),  
  CITY CHAR(20),  
  STATE CHAR(2),  
  ZIP CHAR(10)  
)
```

Los beans de entidad no tienen por qué definir todas las columnas de tablas correspondientes como campos de persistencia. De hecho, un bean de entidad puede incluso no tener una única tabla correspondiente; puede tener los campos de persistencia distribuidos entre varias tablas. Es la herramienta de despliegue del contenedor la que se encargará de definir el modo de persistencia del bean. En este caso, la columna `ID` es un campo auto-incrementado, creado automáticamente por el sistema de base de datos o el sistema contenedor. Es la clave primaria del EJB `Address`.

Una vez que se crea el bean, la clave primaria nunca puede ser modificada. Cuando las claves primarias son valores autogenerados, como la columna `ID` en la tabla `ADDRESS`, el contenedor EJB obtendrá el valor de la clave primaria a partir de la base de datos.

Además de la clase bean, definiremos el interfaz local para el EJB `Address`, lo que nos permitirá hacerlo accesible desde otros beans de entidad (esto es, desde el EJB `Customer`) dentro del mismo espacio de direcciones o proceso:

```
// Interfaz local del EJB Address  
public interface AddressLocal extends javax.ejb.EJBLocalObject {  
  public String getStreet();  
  public void setStreet(String street);  
  public String getCity();  
  public void setCity(String city);  
  public String getState();  
  public void setState(String state);  
  public String getZip();  
  public void setZip(String zip);  
}  
  
// Interfaz local home de Address Address EJB's local home interface  
public interface AddressHomeLocal extends javax.ejb.EJBLocalHome {  
  public AddressLocal createAddress(String street,String city,  
    String state,String zip) throws javax.ejb.CreateException;  
  public AddressLocal findByPrimaryKey(Integer primaryKey)  
    throws javax.ejb.FinderException;  
}
```

El método `ejbCreate()` de la clase `AddressBean` y el método `findByPrimaryKey()` de la interfaz `home` definen ambos el tipo de la clave primaria como `java.lang.Integer`. En este caso la clave primaria es autogenerada. La mayoría de fabricantes EJB 2.0 permitirán hacer corresponder claves primarias de beans de entidad con campos autogenerados. En caso de que la herramienta no lo permitiera, tendríamos que definir el valor de la clave primaria en el método `ejbCreate()`.

6.2. Definición de la relación en el bean que realiza la referencia

6.2.1. Campo de relación en la clase bean

El campo de relación para el EJB `Address` se define en la clase `CustomerBean` usando un método de acceso abstracto, de la misma forma que se declaran los campos de persistencias. En el código que sigue, la clase `CustomerBean` aparece modificada para incluir el EJB `Address` como un campo de relación

```
import javax.ejb.EntityContext;
import javax.ejb.CreateException;

public abstract class CustomerBean implements javax.ejb.EntityBean {
    ...

    // relaciones de persistencia
    public abstract AddressLocal getAddress();
    public abstract void setAddress(AddressLocal address);

    // campos de persistencia
    public abstract boolean getHasGoodCredit();
    public abstract void setHasGoodCredit(boolean creditRating);
    ...
}
```

Los métodos `getAddress()` y `setAddress()` son auto-explicativos; permiten al bean acceder y modificar su relación `address`. Estos métodos de acceso representan un campo de relación, que es un campo virtual que referencia a otro bean de entidad. El nombre del método de acceso viene determinado por el nombre del campo de relación, tal y como está definido en el fichero XML de descripción del despliegue. En este caso hemos llamado a la dirección del cliente `address`, por lo que los nombres de los métodos de acceso correspondientes serán `getAddress()` y `setAddress()`.

6.2.2 Definición de la relación en la tabla de la base de datos

Para incluir la relación entre el EJB `Customer` y el EJB `Address`, añadiremos una clave ajena, `ADDRESS_ID`, en la tabla `CUSTOMER`. La clave ajena apuntará al registro `ADDRESS`.

```
CREATE TABLE CUSTOMER (
    ID INT PRIMARY KEY NOT NULL,
    LAST_NAME CHAR(20),
    FIRST_NAME CHAR(20),
```

```
ADDRESS_ID INT
)
```

Cuando se crea un nuevo EJB `Address` y se actualiza en una relación `address` de un EJB `Customer`, la clave primaria del EJB `Address` se colocará en la columna `ADDRESS_ID` de la tabla `CUSTOMER`, creando una relación en la base de datos:

```
// obtener referencia
AddressLocal address = ...

// establecer la relacion
customer.setAddress(address);
```

6.3 Actualización de la relación desde los clientes

Para dar información al bean `Customer` una dirección, necesitamos proporcionar al bean la información sobre la dirección. Esto podría parecer tan sencillo como declarar en la interfaz remota del bean `Customer` una pareja de métodos de acceso correspondientes, tales como `setAddress()/getAddress()`, pero no es así. Mientras que es sencillo hacer disponible a los clientes remotos campos de persistencia, no es tan sencillo hacer lo mismo con los campos de relación.

La interfaz remota de un bean no puede mostrar sus campos de relación. En el caso del campo `address`, hemos declarado el tipo como `AddressLocal`, el cual es una interfaz local, por lo que los métodos de acceso `setAddress()/getAddress()` no pueden declararse en la interfaz remota del EJB `Customer`. La razón de esta restricción sobre interfaces remotas es sencilla: el `EJBLocalObject`, que implementa la interfaz local, está optimizado para ser usado dentro del mismo espacio de direcciones que la instancia del bean y no está preparado para ser usado a través de la red. En otras palabras, las referencias que implementan la interfaz local de un bean no pueden pasarse a través de la red, por lo que una interfaz local no puede declararse como tipo devuelto o como un parámetro de una interfaz remota.

Las interfaces locales (interfaces que extienden `javax.ejb.EJBLocalObject`), por otro lado, pueden mostrar cualquier tipo de campo de relación. Con interfaces locales, el cliente y el bean que está siendo usado están localizados en el mismo espacio de direcciones, por lo que pueden pasarse referencias locales sin problemas. Por ejemplo, si hubiéramos definido un interfaz local para el EJB `Customer`, podría incluir un método que permita a los clientes locales el acceso directo a su campo de relación `address`:

```
public interface CustomerLocal extends javax.ejb.EJBLocalObject {
    public AddressLocal getAddress();
    public void setAddress(AddressLocal address);
}
```

6.3.1 Definición/modificación de un objeto relacionado

En lo que se refiere al EJB `Address`, es mejor definir una interfaz local sólo debido a que es un bean de grano fino. Para solucionar las restricciones de las interfaces remotas, los métodos de negocio en la clase bean pueden intercambiarse datos, en lugar de referencias a `Address`. Por ejemplo, podemos declarar un método que permita a un cliente enviar información para crear una dirección para el `Customer`:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id) {
        setId(id);
        return null;
    }

    public void ejbPostCreate(Integer id) {}

    // metodos de negocio
    public void setAddress(String street,String city,String state,String
zip) {
        try {
            AddressLocal addr = this.getAddress();
            if (addr == null) {
                // Customer no tiene todavia una direccion. Crear una.
                InitialContext cntx = new InitialContext();
                AddressHomeLocal addrHome =
                    (AddressHomeLocal)cntx.lookup("AddressHomeLocal");
                addr = addrHome.createAddress(street,city,state,zip);
                this.setAddress(addr);
            } else {
                // Customer ya tiene una direccion. Cambiar sus campos.
                addr.setStreet(street);
                addr.setCity(city);
                addr.setState(state);
                addr.setZip(zip);
            }
        } catch (Exception e) {
            throw new EJBException(e);
        }
    }

    ...
}
```

El método de negocio `setAddress()` en la clase `CustomerBean` también se declara en la interfaz remota del EJB `Customer`, por lo que podrá ser llamado por los clientes remotos:

```
public interface Customer extends javax.ejb.EJBObject {
    public void setAddress(String street,String city,String state,String
zip);
    public Name getName() throws RemoteException;
    public void setName(Name name) throws RemoteException;

    public boolean getHasGoodCredit() throws RemoteException;
    public void setHasGoodCredit(boolean creditRating) throws
RemoteException;
}
```

```
}
```

Cuando el método de negocio `CustomerRemote.setAddress()` se invoca en el `CustomerBean`, se usan los argumentos del método para crear un nuevo EJB `Address` y para definirlo como el campo de relación `address`, en el caso de que no exista ya ninguno. Si el EJB `Customer` ya tiene una relación `address`, el EJB `Address` se modifica para reflejar la nueva información.

Cuando creamos un nuevo EJB `Address`, el objeto home se obtiene a través del JNDI ENC (environment naming context) y se llama a su método `createAddress()`. Esto tiene como resultado la creación de un nuevo EJB `Address` y la inserción de el correspondiente registro `ADDRESS` en la base de datos. Después de haber creado el EJB `Address`, se usa en el método `setAddress()`. La clase `CustomerBean` debe llamar explícitamente al método `setAddress()`, o la nueva dirección no se asignará al cliente. De hecho, si creamos simplemente un EJB `Address` sin asignárselo al cliente con el método `setAddress()` estaremos creando un EJB `Address` desconectado. Detallando más, tendrá como efecto la creación de un registro `ADDRESS` de la base de datos que no será referenciado por ningún registro `CUSTOMER`.

La viabilidad de entidades desconectadas depende, en parte, de la integridad referencial de la base de datos. Por ejemplo, si la integridad referencial de la base de datos permite sólo valores no nulos para la columna de clave ajena, la creación de una entidad desconectada puede resultar en un error de la base de datos.

Cuando se invoca el método local `setAddress()`, el contenedor enlaza automáticamente el registro `ADDRESS` y el registro `CUSTOMER`. En este caso, coloca la clave primaria `ADDRESS` en el campo `ADDRESS_ID` del registro `CUSTOMER` y crea una referencia desde el registro `CUSTOMER` al registro `ADDRESS`.

Si el EJB `Customer` ya tiene un `address`, queremos cambiar sus valores en lugar de crear un nuevo EJB `Address`. No necesitamos usar el método local `setAddress()` si vamos a actualizar los valores de un EJB `Address` ya existente, porque el EJB `Address` que modifiquemos ya tiene una relación con el bean de entidad.

6.3.2. Obtención de la información del objeto relacionado

También queremos proporcionar a los clientes un método de negocio para obtener la información de la dirección de un EJB `Customer`. Ya que no podemos enviar directamente una instancia del EJB `Address` (porque es una interfaz local), debemos empaquetar los datos de la dirección de alguna otra forma para enviárselos al cliente. Hay dos soluciones a este problema: adquirir la interfaz remota del EJB `Address` y devolverla o devolver los datos como un objeto valor dependiente (dependent value object).

6.3.2.1 Devolución de una interfaz remota

Podemos obtener la interfaz remota del EJB `Address` sólo se ha sido definida. Los beans de entidad pueden tener un conjunto de interfaces remotas, locales o ambas. En la situación actual, el EJB `Address` tiene un grano demasiado fino para justificar la creación de una interfaz remota, pero en muchas otras circunstancias podemos tener realmente una interfaz remota. Por ejemplo, si el EJB `Customer` referenciara a un EJB `SalesPerson`, el `CustomerBean` podría convertir la referencia local en una referencia remota. Esto se haría accediendo al objeto EJB local, obteniendo su clave primaria (`EJBLocalObject.getPrimaryKey()`), obteniendo el home remoto del EJB `SalesPerson` a partir del JNDI ENC, y luego usando la clave primaria y la referencia home remota para encontrar una referencia a la interfaz remota:

```
public SalesRemote getSalesRep() {
    SalesLocal local = getSalesPerson();
    Integer primKey = local.getPrimaryKey();

    Object ref = jndiEnc.lookup("SalesHomeRemote");
    SalesHomeRemote home = (SalesHomeRemote)
        PortableRemoteObject.narrow(ref, SalesHomeRemote.class);

    SalesRemote remote = home.findByPrimaryKey( primKey );
    return remote;
}
```

6.3.2.2 Devolución de un objeto de valor dependiente

La otra opción es usar un valor dependiente para pasar los datos del EJB `Address` entre el EJB `Customer` y los clientes remotos. Este es el enfoque recomendado para los beans de grano fino como el EJB `Address`. El siguiente código muestra como se usa la clase de valor dependiente `AddressDO` junto con las componentes locales del EJB `Address` (el sufijo “DO” en `AddressDO` es una convención que se suele usar para denominar a las clases que definen valores dependientes; significa “dependent object”):

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(Integer id) {
        setId(id);
        return null;
    }

    public void ejbPostCreate(Integer id) {
    }

    // metodos de negocio

    public AddressDO getAddress() {
        AddressLocal addrLocal = getHomeAddress();
        if(addrLocal == null) return null;
        String street = addrLocal.getStreet();
        String city = addrLocal.getCity();
        String state = addrLocal.getState();
        String zip = addrLocal.getZip();
    }
}
```



```
        AddressDO addrValue = new AddressDO(street,city,state,zip);
        return addrValue;
    }

    public void setAddress(AddressDO addrValue)
        throws EJBException {
        String street = addrValue.getStreet();
        String city = addrValue.getCity();
        String state = addrValue.getState();
        String zip = addrValue.getZip();

        AddressLocal addr = getHomeAddress();

        try {

            if(addr == null) {
                // Customer no tiene una direccion. Crear una nuave.
                InitialContext cntx = new InitialContext();
                AddressHomeLocal addrHome = (AddressHomeLocal)cntx.lookup(
                    "AddressHomeLocal");
                addr = addrHome.createAddress(street, city, state, zip);
                this.setHomeAddress(addr);
            } else {
                // Customer ya tiene una relacion. Modificar sus campos.
                addr.setStreet(street);
                addr.setCity(city);
                addr.setState(state);
                addr.setZip(zip);
            }
        } catch(NamingException ne) {
            throw new EJBException(ne);
        } catch(CreateException ce) {
            throw new EJBException(ce);
        }
    }
    ...
}
```

He aquí la definición de una clase de valor dependiente `AddressDO`, que usa el enterprise bean para enviar información de la dirección al cliente:

```
public class AddressDO implements java.io.Serializable {
    private String street;
    private String city;
    private String state;
    private String zip;

    public AddressDO(String street, String city, String state, String
zip ) {
        this.street = street;
        this.city = city;
        this.state = state;
        this.zip = zip;
    }

    public String getStreet() {
        return street;
    }

    public String getCity() {
        return city;
    }
}
```

```
}

public String getState() {
    return state;
}

public String getZip() {
    return zip;
}
}
```

El valor dependiente `AddressDO` es inmutable, lo que significa que no puede ser alterado una vez que ha sido creado. La inmutabilidad ayuda a reforzar el hecho de que la clase de valor dependiente es una copia, no una referencia remota.

Ahora podemos usar una aplicación cliente para probar la relación del EJB `Customer` con el EJB `Address`. A continuación se encuentra el código del cliente que crea un nuevo cliente, le proporciona una dirección y luego le cambia la dirección.

```
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import javax.naming.Context;
import javax.naming.NamingException;
import java.util.Properties;

public class Client {
    public static void main(String [] args) throws Exception {

        // obtener CustomerHome
        Context jndiContext = getInitialContext();
        Object obj=jndiContext.lookup("CustomerHomeRemote");
        CustomerHome home = (CustomerHomeRemote)
            javax.rmi.PortableRemoteObject.narrow(obj,
CustomerHomeRemote.class);

        // crear un Customer
        Integer primaryKey = new Integer(1);
        Customer customer = home.create(primaryKey);

        // crear un AddressDO
        AddressDO address = new AddressDO("1010 Colorado","Austin",
                                         "TX", "78701");

        // establecer la direccion
        customer.setAddress(address);
        address = customer.getAddress();
        System.out.print(primaryKey+" = ");
        System.out.println(address.getStreet());
        System.out.println(address.getCity()+", "+
            address.getState()+" "+
            address.getZip());

        // crear una nueva direccion
        address = new AddressDO("1600 Pennsylvania Avenue NW",
                                "DC", "WA", "20500");

        // cambiar la direccion del customer
```

```
customer.setAddress(address);

address = customer.getAddress();

System.out.print(primaryKey+" = ");
System.out.println(address.getStreet());
System.out.println(address.getCity()+" "+
                    address.getState()+" "+
                    address.getZip());

// borrar Customer
customer.remove();
}

public static Context getInitialContext()
throws javax.naming.NamingException {
    Properties p = new Properties();
    // ... Specify the JNDI properties specific to the vendor.
    //return new javax.naming.InitialContext(p);
    return null;
}
}
```

6.4 El modelo abstracto de programación

EL modelo abstracto de programación define los métodos de negocio que están asociados a una relación entre EJBs. Por ejemplo, supongamos una relación uno-a-uno unidireccional entre Cliente y Direccion. Cada cliente tiene una dirección asociada. Para poder usar esta relación necesitaremos dos métodos de negocio: uno para definir el bean Direccion asociado a un Cliente y otro para obtenerlo. O supongamos una relación uno-a-muchos unidireccional entre Cliente y NumeroTelefono. Necesitaremos también una pareja de métodos de negocio para, dado un cliente, obtener o añadir números de teléfono asociados. La signature de los métodos de negocio va a depender del tipo de relación que se haya definido.

Debido a la necesidad de que las relaciones se implementen de forma eficiente, los métodos de negocio van a usar siempre interfaces locales de los EJB asociados. Es necesario por ello que todos los EJB que participan en una relación residan en el mismo contenedor EJB.

Los métodos de negocio deben definirse en la interfaz remota (y/o local) del EJB origen de la relación, además de en el fichero de implementación del EJB, donde deben declararse como abstractos.

6.5 El esquema abstracto de persistencia

Además de declarar métodos de acceso abstractos, el desarrollador del bean debe describir la cardinalidad y la dirección de las relaciones entre beans en el fichero de descripción del despliegue. Llamaremos *esquema abstracto de persistencia* a los elementos XML del fichero de descripción del despliegue que describen la relación.

Un esquema abstracto de persistencia de un bean se define en la sección <relationships> del descriptor XML del despliegue del bean. Dentro del elemento <relationships>, cada relación entidad-a-entidad se define en un elemento separado <ejb-relation>:

```
<ejb-jar>
  <enterprise-beans>
    ...
  </enterprise-beans>
  <relationships>
    <ejb-relation>
      ...
    </ejb-relation>
    <ejb-relation>
      ...
    </ejb-relation>
  </relationships>
  <assembly-descriptor>
    ...
  </assembly-descriptor>
</ejb-jar>
```

Los elementos <ejb-relation> complementan el modelo abstracto de programación. Para cada par de métodos de acceso abstractos que definen un compo de relación, hay un elemento <ejb-relation> en el descriptor del despliegue. Además, todos los beans de entidad que participan en una relación se deben definir en el mismo descriptor de despliegue XML.

A continuación se lista parcialmente el descriptor de despliegue de los EJBs Customer y Address, haciendo énfasis en los elementos que definen la relación:

```
<ejb-jar>
  ...
  <enterprise-beans>
    <entity>
      <ejb-name>CustomerEJB</ejb-name>
      <local-home>.CusomterHomeLocal</local-home>
      <local>CustomerLocal</local>
      ...
    </entity>
    <entity>
      <ejb-name>AddressEJB</ejb-name>
      <local-home>AddressHomeLocal</local-home>
      <local>AddressLocal</local>
      ...
    </entity>
    ...
  </enterprise-beans>

  <relationships>
    <ejb-relation>
      <ejb-relation-name>Customer-Address</ejb-relation-name>
      <ejb-relationship-role>
        <ejb-relationship-role-name>
          Customer-has-an-Address
        </ejb-relationship-role-name>
      </ejb-relationship-role>
    </ejb-relation>
  </relationships>
</ejb-jar>
```

```
<multiplicity>One</multiplicity>
<relationship-role-source>
  <ejb-name>CustomerEJB</ejb-name>
</relationship-role-source>
<cmr-field>
  <cmr-field-name>homeAddress</cmr-field-name>
</cmr-field>
</ejb-relationship-role>
<ejb-relationship-role>
  <ejb-relationship-role-name>
    Address-belongs-to-Customer
  </ejb-relationship-role-name>
  <multiplicity>One</multiplicity>
  <relationship-role-source>
    <ejb-name>AddressEJB</ejb-name>
  </relationship-role-source>
</ejb-relationship-role>
</ejb-relation>
</relationships>
</ejb-jar>
```

Todas las relaciones entre el EJB Customer y otros beans, como CreditCard, Address y Phone, requieren que definamos un elemento `<ejb-relationZ>` para complementar los métodos abstractos de acceso.

Cada relación puede tener un nombre, que se declara en el elemento `<ejb-relation-name>`. Esto sirve para identificar la relación para las herramientas de despliegue, pero no es obligatorio.

Cada elemento `<ejb-relation>` tiene exactamente dos elementos `<ejb-relationship-role>`, uno por cada participante en la relación. En el ejemplo anterior, el primer `<ejb-relationship-role>` declara el papel del EJB Customer en la relación. Sabemos esto porque el elemento `<relationship-role-source>` especifica el nombre del bean `<ejb-name>` como CustomerEJB, el nombre del bean Customer tal y como aparece en la sección `<enterprise-beans>`.

El elemento `<ejb-relationship-role>` también declara la cardinalidad del papel. El elemento `<multiplicity>` puede ser o bien One o Many. En este caso, el elemento `<multiplicity>` del EJB Customer tiene un valor de One, lo que quiere decir que cada EJB Address tiene una relación con exactamente un EJB Customer. El elemento `<multiplicity>` del EJB Address también señala One, lo que significa que cada EJB Customer tiene una relación con exactamente un EJB Address. Si el EJB Customer tuviera una relación con muchos EJBs Address, el elemento `<multiplicity>` del EJB Address debería definirse como Many.

En la sesión anterior definimos métodos abstractos de acceso en el EJB Customer para obtener y definir el EJB Address en el campo address, pero el EJB Address no tenía métodos abstractos de acceso para el EJB Customer. En este caso estamos definiendo una relación unidireccional, lo que significa que sólo uno de los beans de la relación mantiene un campo de relación gestionado por el contenedor.

Si el bean descrito por la relación <ejb-relationship-role> mantiene una referencia a otro bean en la relación, esa referencia debe declararse como un campo de relación gestionado por el contenedor en el elemento <cmr-field>. Este elemento se declara bajo el elemento <ejb-relationship-role>:

```
<ejb-relationship-role>
  <ejb-relationship-role-name>
    Customer-has-an-Address
  </ejb-relationship-role-name>
  <multiplicity>One</multiplicity>
  <relationship-role-source>
    <ejb-name>CustomerEJB</ejb-name>
  </relationship-role-source>
  <cmr-field>
    <cmr-field-name>address</cmr-field-name>
  </cmr-field>
</ejb-relationship-role>
```

La especificación EJB 2.0 requiere que el <cmr-field-name> comience con una letra minúscula. Para cada campo de relación definido debe haber una pareja de métodos de acceso en la clase bean. Un método de la pareja debe definirse con el nombre set<cmr-field-name>(), con la primera letra del <cmr-field-name> cambiada a mayúscula. El otro método se define como get<cmr-field-name>(), también con la primera letra del <cmr-field-name> cambiada a mayúsculas.

En el ejemplo previo, el campo <cmr-field-name> es address, con lo que se deben definir los métodos abstractos getAddress() y setAddress():

```
// bean class code
public abstract void setAddress(AddressLocal address);
public abstract AddressLocal getAddress();

// XML deployment descriptor declaration
<cmr-field>
  <cmr-field-name>address</cmr-field-name>
</cmr-field>
```

El tipo devuelto por el método get<cmr-field-name>() y el tipo del parámetro del método set<cmr-field-name>() deben ser el mismo. El tipo debe ser la interfaz local de bean de entidad al que se hace referencia o uno de los dos tipos java.util.Collection. En el caso del campo de relación address, usamos la interfaz local del EJB Address, AddressLocal. Los tipos Collection se usan cuando aparece la cardinalidad Many en la relación. Los veremos más adelante.

Es importante hacer notar de nuevo que aunque los beans de entidad pueden tener tanto interfaces locales como remotas, una relación gestionada por el contenedor puede usar sólo las interfaces locales del bean cuando hace persistente una relación. Por ello, por ejemplo, es ilegal definir un método abstracto de acceso que tenga un argumento de tipo javax.ejb.EJBObject (un

tipo de interfaz remoto). Todas las relaciones gestionadas por el contenedor se basan en tipos `javax.ejb.EJBLocalObject` (interfaz local).

6.6 Modelado de la base de datos

A lo largo de este tema se muestran distintos esquemas de tablas de bases de datos. La intención de estos esquemas es sólo demostrar posibles relaciones entre entidades en la base de datos; no son obligatorios. Por ejemplo, la relación Dirección-Cliente se pone de manifiesto introduciendo en la tabla CUSTOMER una clave foránea a la tabla ADDRESS. Esta no es la forma habitual de organización de la mayoría de bases de datos. En lugar de esto, probablemente usan una tabla de enlace o hacen que la tabla ADDRESS mantenga una relación foránea a CUSTOMER. Sin embargo, este esquema muestra cómo la persistencia gestionada por el contenedor puede soportar diferentes organizaciones de la base de datos.

A lo largo de este tema, suponemos que las tablas de las bases de datos se crean antes de la aplicación EJB. Algunos fabricantes ofrecen herramientas que generan las tablas automáticamente a partir de las relaciones definidas entre los bean de entidad. Estas herramientas pueden crear esquemas que son muy distintos de los que se muestran aquí. En otros casos, los fabricantes que soportan esquemas de bases de datos ya establecidos pueden no tener la flexibilidad necesaria para soportar los esquemas ilustrados en este tema. Como un desarrollador EJB, debes ser lo suficientemente flexible para adaptarte a las facilidades proporcionadas por tu fabricante de EJB.

6.7 Relación Uno-a-Uno Unidireccional

Un ejemplo de una relación uno-a-uno es la relación entre el EJB Customer y el EJB Address que vimos en el tema pasado. En este caso, cada Customer tiene exactamente un Address y cada Address tiene exactamente un Customer. Qué bean referencia a qué otro determina la dirección de la relación. En este caso el Customer tiene una referencia al Address, pero no al contrario. Es una relación unidireccional porque sólo puedes ir del Customer al Address, y no en la otra dirección. En otras palabras, un EJB Address no tiene idea de a quién pertenece.

6.7.1 Esquema de base de datos relacional

La relación uno-a-uno unidireccional usa un esquema de base de datos típico en el que una tabla contiene una clave foránea a otra tabla. En este caso, la tabla CUSTOMER contiene una clave foránea a la tabla ADDRESS, pero no al contrario. Esto permite que los registros de la tabla ADDRESS sean compartido por otras tablas. el hecho de que el esquema de base de datos no sea el mismo que el esquema abstracto de persistencia muestra que son independientes hasta cierto punto.

6.7.2 Modelo abstracto de programación

Como hemos visto en el tema pasado, los métodos abstractos de acceso se usan para definir campos de relación en la clase del bean. Cuando un bean de entidad mantiene una referencia a otro bean, define un par de métodos abstractos de acceso para modelar esa referencia. En relaciones unidireccionales, que pueden navegarse sólo en una dirección, sólo uno de los enterprise beans define estos métodos abstractos de acceso. En este caso, en la clase CustomerBean se definen los métodos getAddress()/setAddress() para acceder a los EJBs Address, pero no se define ningún método de este tipo en la clase AddressBean para acceder el EJB Customer.

Un EJB Address puede ser compartido entre campos de relación del mismo enterprise bean, pero no puede ser compartido entre distintos EJBs Customer. Si, por ejemplo, el EJB Customer definiera dos campos de relación, billingAddress and homeAddress, como relaciones uno-a-uno unidireccionales con el EJB Address, estos dos campos podría referenciar al mismo EJB Address:

```
public class CustomerBean implements javax.ejb.EntityBean {
    ...
    public void setAddress(String street,String city,String
state,String zip) {
        ...

        address = addressHome.createAddress(street, city, state, zip);

        this.setHomeAddress(address);
        this.setBillingAddress(address);

        AddressLocal billAddr = this.getBillingAddress();
        AddressLocal homeAddr = this.getHomeAddress();

        if(billAddr.isIdentical(homeAddr))
            // always true

        ...
    }
    ...
}
```

Si en cualquier momento quisiéremos distinguir el billingAddress del homeAddress, simplemente tendríamos que llamar al método set con un EJB distinto. Para poder implementar estas relaciones, habría que modificar la tabla CUSTOMER:

```
CREATE TABLE CUSTOMER
(
    ID INT PRIMARY KEY,
    LAST_NAME CHAR(20),
    FIRST_NAME CHAR(20),
    ADDRESS_ID INT,
    BILLING_ADDRESS_ID INT
)
```


Tal y como se ha mencionado, no es posible compartir un mismo EJB Address entre dos EJBs distintos Customer. Si, por ejemplo, el EJB Address que está asignado al Customer A se asignara al Customer B, el Address se movería de un Customer a otro y el Customer A dejaría de estar relacionado con ningún EJB Address. Este efecto, en apariencia extraño, es sencillamente un resultado natural de la definición uno-a-uno de la relación.

Si el EJB Customer no tienen un EJB Address asociado, el método getAddress() devolverá null.

6.7.3 El esquema abstracto de persistencia

Hemos definido ya los elementos XML de la relación Customer-Address. We defined the XML elements for the Customer-Address. El elemento <ejb-relation> que usábamos declaraba una relación uno-a-uno unidireccional. Si necesitáramos usar dos campos de relación con el EJB Address, los campos homeAddress y billingAddress, cada una de estas relaciones tendría que ser descrita en su propio elemento <ejb-relation>:

```
<relationships>
  <ejb-relation>
    <ejb-relation-name>Customer-HomeAddress</ejb-relation-name>
    <ejb-relationship-role>
      ...
      <cmr-field>
        <cmr-field-name>homeAddress</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      ...
    </ejb-relationship-role>
  </ejb-relation>
  <ejb-relation>
    <ejb-relation-name>Customer-BillingAddress</ejb-relation-name>
    <ejb-relationship-role>
      ...
      <cmr-field>
        <cmr-field-name>billingAddress</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      ...
    </ejb-relationship-role>
  </ejb-relation>
</relationships>
```

6.8 Relación Uno-a-Uno Bidireccional

Podemos ahora ampliar nuestro EJB Customer para incluir una referencia a un EJB CreditCard. Además, queremos que el EJB CreditCard mantenga también la referencia con el Customer que lo tiene asociado. Esto parece una buena decisión de diseño, ya que habrá bastantes aplicaciones en las que necesitaremos, a partir de una instancia del EJB CreditCard, conocer cuál es el

Customer que la posee. Debido a que las referencias se realizan en ambos sentidos, tendremos una relación uno-a-uno bidimensional

6.8.1 Esquema de base de datos relacional

El EJB CreditCard tiene una tabla correspondiente en la base de datos, que se llama CREDIT_CARD. Necesitaremos, evidentemente, la clave foránea que liga los Customer con los CreditCard. Además, al ser una relación bidireccional, necesitaremos también añadir una clave foránea en la tabla CREDIT_CARD hacia la tabla CUSTOMER::

```
CREATE TABLE CREDIT_CARD
(
    ID INT PRIMARY KEY NOT NULL,
    EXP_DATE DATE,
    NUMBER CHAR(20),
    NAME CHAR(40),
    ORGANIZATION CHAR(20),
    CUSTOMER_ID INT
)

CREATE TABLE CUSTOMER
(
    ID INT PRIMARY KEY,
    LAST_NAME CHAR(20),
    FIRST_NAME CHAR(20),
    ADDRESS_ID INT,
    CREDIT_CARD_ID INT
)
```

También es posible establecer una relación bidireccional uno-a-uno a través de una tabla de enlace, en la que cada columna de clave foránea debe ser única. Esto es conveniente cuando no quieres imponer relaciones sobre las tablas originales. Las tablas de enlace se usan también en las relaciones una-a-muchos y muchos-a-muchos, pero es importante recordar que el esquema de base de datos usado en estos ejemplos es puramente ilustrativo. El esquema abstracto de persistencia de un bean de entidad se puede corresponder con una diversidad de esquemas de bases de datos; los esquemas de base de datos usado en estos ejemplos son sólo una posibilidad.

6.8.2 Modelo abstracto de programación

Para modelar la relación entre los EJB Customer y CreditCard, tendremos que declarar un campo de relación llamado customer en la clase CreditCardBean:

```
public abstract class CreditCardBean extends javax.ejb.EntityBean {

    ...

    // relationship fields
    public abstract CustomerLocal getCustomer();
    public abstract void setCustomer(CustomerLocal local);
}
```

```
// persistence fields
public abstract Integer getId();
public abstract void setId(Integer id);
public abstract Date getExpirationDate();
public abstract void setExpirationDate(Date date);
public abstract String getNumber();
public abstract void setNumber(String number);
public abstract String getNameOnCard();
public abstract void setNameOnCard(String name);
public abstract String getCreditOrganization();
public abstract void setCreditOrganization(String org);

// standard callback methods
...
}
```

En este caso, usamos la interfaz local del EJB Customer (asumiendo que se ha creado) debido a que los campos de relación requieren tipos de interfaces locales. La limitación de usar interfaces locales en lugar de interfaces remotas es que perdemos la transparencia de la ubicación. Todos los beans de entidad deben estar ubicados en el mismo proceso o la misma Máquina Virtual Java (JVM).

También podemos añadir un conjunto de métodos abstractos de acceso en la clase CustomerBean para el campo de relación creditCard:

```
public class CustomerBean implements javax.ejb.EntityBean {
    ...
    public abstract void setCreditCard(CreditCardLocal card);
    public abstract CreditCardLocal getCreditCard();
    ...
}
```

Aunque está disponible el método setCustomer() en el CreditCardBean, no tenemos que establecer explícitamente la referencia al Customer en el EJB CreditCard. Cuando una referencia a un EJB CreditCard se pasa en el método setCreditCard() en la clase CustomerBean, el contenedor EJB establecerá automáticamente la relación customer en el EJB CreditCard para que apunte al EJB Customer:

```
public class CustomerBean implements javax.ejb.EntityBean {
    ...
    public void setCreditCard(Date exp, String numb, String name,
String org)
        throws CreateException {
        ...

        card = creditCardHome.create(exp,numb,name,org);

        // el campo customer del EJB CreditCard se establece
        automáticamente
        this.setCreditCard(card);
        Customer customer = card.getCustomer();
    }
}
```

```

        if(customer.isIdentical(ejbContext.getEJBLocalObject()))
            // always true

        ...
    }
    ...
}

```

Las reglas para compartir un bean individual en una relación una-a-una bidireccional son las mismas que las de una relación uno-a-uno unidireccional. Mientras que un EJB CreditCard puede compartirse entre campos de relación en el mismo EJB Customer, no puede compartirse entre diferentes EJBs Customer. La asignación del CreditCard del Customer A al Customer B elimina la asociación entre el CreditCard y el Customer A y la mueve al Customer B.

6.8.3 Esquem abstracto de persistencia

El elemento `<ejb-relation>` que define la relación Customer-a-CreditCard es similar al usado para la relación Customer-a-Address, con una diferencia importante: los dos elementos `<ejb-relationship-role>` tienen un `<cmr-field>`:

```

<relationships>
  <ejb-relation>
    <ejb-relation-name>Customer-CreditCard</ejb-relation-name>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        Customer-has-a-CreditCard
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>CustomerEJB</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>creditCard</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
      <ejb-relationship-role-name>
        CreditCard-belongs-to-Customer
      </ejb-relationship-role-name>
      <multiplicity>One</multiplicity>
      <relationship-role-source>
        <ejb-name>CreditCardEJB</ejb-name>
      </relationship-role-source>
      <cmr-field>
        <cmr-field-name>customer</cmr-field-name>
      </cmr-field>
    </ejb-relationship-role>
  </ejb-relation>
</relationships>

```

El hecho de que ambos participantes en la relación definan elementos `<cmr-field>` (campos de relación) nos indica que la relación es bidireccional.

6.9 Relación Uno-a-Muchos unidireccional

Los beans de entidad también pueden mantener relaciones de cardinalidad mayor de uno. Esto significa que un bean de entidad puede contener muchos otros beans de entidad. Por ejemplo, el EJB Customer puede estar relacionado con muchos EJBs Phone, cada uno de los cuales representa un número de teléfono. Esto es bastante distinto de las relaciones sencillas uno-a-uno. Las relaciones uno-a-muchos y muchos-a-muchos requieren que el desarrollador trabaje con una colección de referencias cuando accede al campo de relación.

6.9.1 Esquema de base de datos relacional

Para ilustrar la relación una-a-muchos unidireccional, usaremos un nuevo bean de entidad, el EJB Phone, para el que deberemos definir una tabla, la tabla PHONE:

```
CREATE TABLE PHONE
(
    ID INT PRIMARY KEY NOT NULL,
    NUMBER CHAR(20),
    TYPE INT,
    CUSTOMER_ID INT
)
```

Las relaciones uno-a-muchos entre las tablas CUSTOMER y PHONE podrían realizarse de distintas formas en una base de datos relacional. Para este ejemplo hemos decidido incluir en la tabla PHONE una clave foránea a la tabla CUSTOMER.

La tabla con los datos añadidos puede mantener una columna de claves foráneas no únicas. En el caso de los EJBs Customer y Phone, la tabla PHONE mantiene una clave foránea a la tabla CUSTOMER, y uno o más registros PHONE pueden contener claves foráneas hacia el mismo registro CUSTOMER. En otras palabras, en la base de datos los registros PHONE apuntan a los registros CUSTOMER. En el modelo abstracto de programación, sin embargo, es el EJB Customer el que apunta a los EJBs Phone. ¿Cómo funciona esto? El sistema contenedor esconde el puntero reverso de forma que parece que es el Customer el que conoce los EJB Phone asociados y no al revés. Cuando le pedimos al contenedor que devuelva una Collection de EJBs Phone (mediante la invocación del método `getPhoneNumbers()`), preguntará a la tabla PHONE por todos los registros con una clave foránea que se corresponda con la clave primaria del EJB Customer.

Una implementación más sencilla de la relación Customer-Phone podría usar una tabla de enlace que mantenga dos columnas con claves foráneas apuntando tanto a los registros CUSTOMER y PHONE. Podríamos entonces colocar una restricción de unicidad en la columna de clave foránea PHONE para asegurarnos de que sólo contiene entradas únicas (esto es, de que cada teléfono sólo pertenece a un cliente), mientras que podríamos permitir que la columna de clave foránea CUSTOMER tuviera duplicados. La ventaja de la tabla de enlace es que no se modifica ninguna de las dos tablas originales CUSTOMER y PHONE.

6.9.2 Modelo abstracto de programación

En el modelo abstracto de programación representamos una relación a muchos definiendo un campo de relación que pueda apuntar a muchos beans de entidad. Para conseguir esto, empleamos los mismos métodos abstractos de acceso que ya hemos usado en las relaciones uno-a-uno, pero esta vez definimos el tipo del campo como `java.util.Collection` o `java.util.Set`. La colección mantiene un grupo homogéneo de referencias a interfaces locales del objeto EJB.

Por ejemplo, un EJB Customer puede estar relacionado con muchos números de teléfono, cada uno de ellos representado por un EJB Phone. En lugar de tener distintos campos de relación para cada uno de ellos, el EJB Customer mantiene todos los EJBs Phone en un campo de relación basado en colecciones, al que se puede acceder a través de los métodos abstractos de acceso:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {
    ...
    // relationship fields
    public java.util.Collection getPhoneNumbers();
    public void setPhoneNumbers(java.util.Collection phones);

    public AddressLocal getHomeAddress();
    public void setHomeAddress(AddressLocal local);
    ...
}
```

El EJB Phone debe definirse como un bean de entidad con una interfaz local. Al ser la relación unidireccional, no es necesario definir los campos de relación en la clase bean:

```
// the local interface for the Phone EJB
public interface PhoneLocal extends javax.ejb.EJBLocalObject {
    public String getNumber();
    public void setNumber(String number);
    public byte getType();
    public void setType(byte type);
}

// the bean class for the Phone EJB
public class PhoneBean implements javax.ejb.EntityBean {

    public Integer ejbCreate(String number, byte type) {
        setNumber(number);
        setType(type);
        return null;
    }
    public void ejbPostCreate(String number, byte type) {
    }

    // persistence fields
    public abstract Integer getId();
    public abstract void setId(Integer id);
    public abstract String getNumber();
}
```

```
public abstract void setNumber(String number);
public abstract byte getType();
public abstract void setType(byte type);

// standard callback methods
...
}
```

Para ilustrar cómo un bean de entidad usa un campo de relación basado en una colección, definiremos un método en la clase CustomerBean que permita a los clientes remotos añadir nuevos números de teléfono. El método, addPhoneNumber(), usa los argumentos para crear un nuevo EJB Phone y luego añade ese EJB Phone a un campo de relación basado en una colección llamado phoneNumbers:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    // business methods
    public void addPhoneNumber(String number, String type) {

        InitialContext jndiEnc = new InitialContext();
        PhoneHomeLocal phoneHome = jndiEnc.lookup("PhoneHomeLocal");
        PhoneLocal phone = phoneHome.create(number,type);

        Collection phoneNumbers = this.getPhoneNumbers();
        phoneNumbers.add(phone);

    }
    ...
    // relationship fields
    public java.util.Collection getPhoneNumbers();
    public void setPhoneNumbers(java.util.Collection phones);

    ...
}
```

Hemos creado primero el EJB Phone y luego lo hemos añadido a la relación añadiéndolo a la colección que define esta relación. La colección phoneNumbers la obtenemos a partir del método de acceso getPhoneNumbers(). Al añadir el EJB Phone a la Collection hace que el contenedor establezca la clave foránea en el nuevo registro PHONE de forma que apunte al registro CUSTOMER del EJB Customer. Si hubiéramos usado una tabla de enlace, se habría creado un nuevo registro. A partir de este momento, el nuevo EJB Phone estará disponible en la relación basada en la colección phoneNumbers.

También es posible actualizar o borrar referencias en una relación basada en una colección usando el método de acceso de la relación. Por ejemplo, el siguiente código define dos métodos en la clase CustomerBean que permiten a los clientes borrar o actualizar números de teléfono en el campo de relación del bean phoneNumbers:

```
public abstract class CustomerBean implements javax.ejb.EntityBean {

    // business methods
```

```
public void removePhoneNumber(byte typeToRemove) {

    Collection phoneNumbers = this.getPhoneNumbers();
    Iterator iterator = phoneNumbers.iterator();
    while(iterator.hasNext()) {
        PhoneLocal phone = (PhoneLocal)iterator.next();
        if(phone.getType() == typeToRemove) {
            iterator.remove(phone);
            break;
        }
    }
}

public void updatePhoneNumber(String number, byte typeToUpdate) {
    Collection phoneNumbers = this.getPhoneNumbers();
    Iterator iterator = phoneNumbers.iterator();
    while(iterator.hasNext()) {
        PhoneLocal phone = (PhoneLocal)iterator.next();
        if(phone.getType() == typeToUpdate) {
            phone.setNumber(number);
            break;
        }
    }
}

...
// relationship fields
public java.util.Collection getPhoneNumbers();
public void setPhoneNumbers(java.util.Collection phones);
```

En el método de negocio `removePhoneNumber()`, se encuentra un EJB Phone con el mismo tipo y se borra de la relación basada en una colección. El EJB no se borra de la base de datos, sino que se elimina su asociación con el EJB Customer.

El método `updatePhoneNumber()` realmente modifica un EJB Phone existente, cambiando su estado en la base de datos. El EJB Phone sigue siendo referenciado por la relación basada en una colección, pero sus datos han cambiado.

Los métodos `removePhoneNumber()` y `updatePhoneNumber()` ilustran que una relación basada en una colección puede accederse y actualizarse exactamente igual que cualquier otro objeto `Collection`. Además, es posible obtener un `java.util.Iterator` a partir del objeto `Collection` y realizar iteraciones con él. Sin embargo, es necesaria cierta precaución cuando se usa un iterador sobre una relación basada en una colección. No se deben añadir o eliminar elementos del objeto `Collection` mientras que se está usando el iterador. La única excepción a esta regla es que el método `Iterator.remove()` puede usarse para eliminar una entrada. Aunque los métodos `Collection.add()` y `Collection.remove()` pueden usarse en otras circunstancias, si se llaman mientras que el iterador está en uso se generará una excepción `java.util.IllegalStateException`.

Si no se ha añadido ningún bean al campo de relación `phoneNumbers`, el método `getPhoneNumbers()` devolverá un objeto `Collection` vacío. Los campos de relación de tipo `<multiplicidad>` nunca devuelve `null`. El objeto `Collection` usado con el campo de relación está implementado por el sistema contenedor y está estrechamente acoplado al funcionamiento interno del contenedor. Esto

permite al contenedor EJB implementar mejoras en la eficiencia, como la concurrencia optimista, sin exponer estos mecanismos propietarios al desarrollador del bean (un objeto Collection obtenido a partir de una relación basada en una colección que se materializa en una transacción no puede modificarse fuera del alcance de esas transacción). Los objetos Collection definidos en la aplicación pueden usarse con campos de relación gestionados por el contenedor sólo si los elementos son del tipo apropiado. Por ejemplo, es legar crear un nuevo objeto Collection y luego añadir ese objeto Collection al EJB Customer usando el método `setPhoneNumbers()`:

```
public void addPhoneNumber(String number, String type) {  
  
    ...  
    PhoneLocal phone = phoneHome.create(number,type);  
  
    Collection phoneNumbers = java.util.Vector();  
    phoneNumbers.add(phone);  
  
    // This is allowed  
    this.setPhoneNumbers(phoneNumbers);  
}  
  
// relationship fields  
public java.util.Collection getPhoneNumbers();  
  
public void setPhoneNumbers(java.util.Collection phones);
```

Si el EJB Customer tuviera una colección de EJB Phone previamente asociados, dejarían de estarlo.

6.9.3 Es esquema abstracto de persistencia

El esquema abstracto de persistencia para las relaciones unidireccionales uno-a-muchos tiene unas cuantas diferencias significativas con respecto a los elementos `<ejb-relation>` vistos hasta el momento:

```
<relationships>  
  <ejb-relation>  
    <ejb-relation-name>Customer-Phones</ejb-relation-name>  
    <ejb-relationship-role>  
      <ejb-relationship-role-name>  
        Customer-has-many-Phone-numbers  
      </ejb-relationship-role-name>  
      <multiplicity>One</multiplicity>  
      <relationship-role-source>  
        <ejb-name>CustomerEJB</ejb-name>  
      </relationship-role-source>  
      <cmr-field>  
        <cmr-field-name>phoneNumbers</cmr-field-name>  
        <cmr-field-type>java.util.Collection</cmr-field-type>  
      </cmr-field>  
    </ejb-relationship-role>  
    <ejb-relationship-role>  
      <ejb-relationship-role-name>
```

```
        Phone-belongs-to-Customer
    </ejb-relationship-role-name>
    <multiplicity>Many</multiplicity>
    <relationship-role-source>
        <ejb-name>PhoneEJB</ejb-name>
    </relationship-role-source>
</ejb-relationship-role>
</ejb-relation>
</relationships>
```

En el elemento `<ejb-relation>`, la multiplicidad del EJB Customer se declara como One, mientras que la multiplicidad del `<ejb-relationship-role>` del EJB Phone es Many. Esto establece obviamente la relación como uno-a-muchos. El hecho de que el `<ejb-relationship-role>` del EJB Phone no especifique un elemento `<cmr-field>` indica que la relación uno-a-muchos es unidireccional.

El cambio más interesante es el añadido del elemento `<cmr-field-type>` en la declaración del `<cmr-field>` del EJB Customer. El `<cmr-field-type>` debe especificarse para un bean que tiene un campo de relación basado en una colección (en este caso, el campo `phoneNumbers` mantenido por el EJB Customer). El campo `<cmr-field-type>` sólo puede tener dos valores, `java.util.Collection` o `java.util.Set`, que son los tipos permitidos en relaciones basadas en colecciones.

Tema 7: Gestión de transacciones

7.1 Introducción

7.1.1 Transacciones

En los negocios, una transacción incluye un intercambio entre dos partes. Cuando compras un periódico, intercambias dinero por un objeto; cuando trabajas para una compañía, intercambias conocimiento y tiempo por dinero. Si nos encontramos en uno de estos intercambios, siempre tenemos cuidado de asegurarnos que no sucede nada extraño. Si le damos al quiosquero un billete de 5€, esperamos que nos devuelva 4€ junto con el periódico, que vale 1€. Monitorizamos la seguridad de la transacción para asegurarnos de que cumple todas las restricciones que conlleva.

En software de negocios, una transacción incluye el concepto de un intercambio comercial. Una transacción de un sistema de negocios (transacción para abreviar) es la ejecución de una unidad-de-trabajo que accede uno o más recursos compartidos, normalmente bases de datos. Una unidad-de-trabajo es un conjunto de actividades que se relacionan mutuamente y que deben ser realizadas completamente. Por ejemplo, una operación de reserva de un billete de avión en un sistema informático puede estar formada por la selección del asiento a reservar, el cargo en una tarjeta de crédito y la generación de un billete. Todas estas acciones forman una unidad-de-trabajo que no puede romperse.

Las transacciones forman parte de distintos tipos de sistemas. En cada transacción el objetivo es el mismo: ejecutar una unidad-de-trabajo que resulte en un intercambio fiable. He aquí algunos ejemplos de sistemas de negocios que usan transacciones:

Cajeros automáticos

Los cajeros automáticos son un ejemplo típico de sistema en el que es fundamental el uso de transacciones. Cuando sacas dinero del cajero, por ejemplo, se debe chequear que tienes dinero suficiente en la cuenta corriente, después se debe entregar el dinero y por último se debe realizar un cargo en la cuenta.

Compra on-line

En una compra on-line también se debe hacer un uso intensivo de las transacciones. Cuando realizas una compra on-line debes proporcionar el número de tarjeta de crédito, éste debe validarse y después debe cargarse el precio de la compra. Luego se debe emitir un pedido al almacén para que realice el envío de la compra. Todas estas acciones deben ser una unidad-de-trabajo, una transacción que se debe ejecutar de forma indivisible.

Los sistemas que necesitan usar transacciones normalmente son complejos y realizan operaciones que conllevan el uso grandes cantidades de datos. Las transacciones deben por ello preservar la integridad de los datos, lo que significa que todas las operaciones que forman las transacciones deben funcionar perfectamente o la que la transacción no se debe ejecutar en absoluto.

En el campo de la gestión de transacciones se han identificado cuatro características que las transacciones deben cumplir para que el sistema sea considerado seguro. Las transacciones deben ser atómicas, consistentes, aisladas y duraderas (ACID, en inglés). A continuación describimos estos términos:

Atómica

Para ser atómica, una transacción debe ejecutarse totalmente o no ejecutarse en absoluto. Esto significa que todas las tareas dentro de una unidad-de-trabajo deben ejecutarse sin error. Si alguna de estas tareas falla, la transacción completa se debe abortar y todos los cambios que se han realizado en los datos deben deshacerse. Si todas las tareas se ejecutan correctamente, la transacción se comete (commit), lo que significa que los cambios realizados en los datos se hacen permanentes o duraderos.

Consistente

La consistencia es una característica transaccional que debe ser impuesta tanto por el sistema transaccional como por el desarrollador de la aplicación. La consistencia se refiere a la integridad del almacén de datos. El sistema transaccional cumple su obligación de consistencia asegurando que una transacción es atómica, aislada y duradera. El desarrollador de la aplicación debe asegurarse que la base de datos tiene las restricciones apropiadas (claves primarias, integridad referencial, y otras) y que la unidad-de-trabajo, la lógica de negocio, no resulta en datos inconsistentes (esto es, los datos no se corresponden con lo que representan del mundo real). En una transferencias entre cuentas, por ejemplo, un cargo en una cuenta debe ser igual a un ingreso en otra.

Aislada

Una transacción debe poder ejecutarse sin interferencia de otros procesos o transacciones. En otras palabras, los datos a los que accede una transacción no pueden ser modificados por ninguna otra parte del sistema hasta que la transacción se completa.

Duradera

Durabilidad significa que todos los cambios en los datos realizados durante el curso de una transacción deben escribirse en algún tipo de almacenamiento físico antes de que la transacción concluya con éxito. Esto asegura que los cambios no se pierden si el sistema se cae.

7.1.2 Un ejemplo con EJBs

Supongamos el siguiente método de un EJB que llamamos TravelAgent

```
public TicketDO bookPassage(CreditCardDO card, double price)
    throws IncompleteConversationalState {

    if (customer == null || cruise == null || cabin == null) {
        throw new IncompleteConversationalState();
    }
    try {
        ReservationHomeLocal resHome = (ReservationHomeLocal)
            jndiContext.lookup("java:comp/env/ejb/ReservationHomeLocal");
        ReservationLocal reservation =
            resHome.create(customer, cruise, cabin, price);
        Object ref = jndiContext.lookup
            ("java:comp/env/ejb/ProcessPaymentHomeRemote");
        ProcessPaymentHomeRemote ppHome = (ProcessPaymentHomeRemote)
            PortableRemoteObject.narrow(ref,
            ProcessPaymentHomeRemote.class);
        ProcessPaymentRemote process = ppHome.create();
        process.byCredit(customer, card, price);

        TicketDO ticket = new TicketDO(customer, cruise, cabin, price);

        return ticket;
    } catch (Exception e) {
        throw new EJBException(e);
    }
}
```

¿Es fiable el método? Una primera medida de la fiabilidad del EJB TravelAgent es su atomicidad: ¿asegura que la transacción se ejecuta completamente o no se ejecuta en absoluto? Para responder debemos concentrarnos en las tareas críticas que modifican o crea información nueva. En el método bookPassage(), se crea un EJB Reservation, el EJB ProcessPayment realiza un cargo en una tarjeta de crédito, y se crea un objeto TicketDO. Todas estas tareas deben tener éxito para que la transacción completa lo tenga a su vez.

Para entender la importancia de la característica de atomicidad podríamos imaginar qué sucedería si cualquiera de estas tareas fallara. Si, por ejemplo, la creación de un EJB Reservation fallara pero todas las otras tareas tuvieran éxito, el cliente terminaría probablemente expulsado del crucero o compartiendo el camarote con un extraño. En lo que concierne a la agencia de viajes, el método bookPassage() habría sido ejecutado con éxito porque se habría generado un TicketDO. Si se genera un billete sin la creación de una reserva, el estado del sistema de negocio se convierte en inconsistente con la realidad, porque el cliente ha pagado por un billete pero la reserva no se ha registrado. De la misma forma, si el EJB ProcessPayment falla al cargar la tarjeta de crédito del cliente, el cliente obtiene un crucero gratis. Seguro que él se quedará contento, pero gerencia no. Por último, si el TicketDO no se crea nunca, el cliente no tendrá ningún registro de la transacción y probablemente no podrá subir al crucero.

Por ello, la única forma de que pueda completarse la operación `bookPassage()` es si todas sus partes críticas se ejecutan con éxito. Si algo va mal, el proceso completo debe abortarse. Abortar una transacción requiere más que simplemente no finalizar las tareas; además, todas las tareas dentro de la transacción deben deshacerse. Si, por ejemplo, la creación de los EJB `Reservation` y `ProcessPayment` se realiza con éxito, pero la creación del `TicketDO` falla, los registros `reservation` y `payment` no deben añadirse a la base de datos.

Para que la operación `bookPassage()` sea completamente segura debe cumplir los otros requisitos de una transacción: debe ser consistente, aislada y duradera.

Para mantener la consistencia de las operaciones desde el punto de vista de la lógica del negocio, es necesario que se cumplan las otras tres propiedades y además que el desarrollador de la aplicación sea estricto a la hora de aplicar las restricciones de integridad en toda la implementación de la aplicación. Por ejemplo, de nada serviría que el sistema transaccional asegurase la atomicidad de la operación `bookPassage()` si el desarrollador no incluyera dentro del método una llamada a la consulta de la tarjeta de crédito y devolviera directamente el `TicketDO`. Desde el punto de vista del negocio, la transacción habría fallado, ya que se ha emitido un ticket sin realizar un cobro.

El aislamiento de la operación tiene que asegurar que otros procesos no van a modificar los datos de los EJBs mientras que la transacción está desarrollándose.

Por último, la durabilidad de la transacción obliga a que todas las operaciones hayan sido hechas persistentes antes de dar la transacción por terminada.

Asegurarnos que las transacciones se adhieren a los principios ACID requiere un diseño cuidadoso. El sistema tiene que monitorizar el progreso de una transacción, para asegurarse de que todo funciona correctamente, de que los datos se modifican de forma correcta, de que las transacciones no interfieren entre ellas y de que los cambios pueden sobrevivir a una caída del sistema. Comprobar todas estas condiciones conlleva un montón de trabajo. Afortunadamente, la arquitectura EJB soporta de forma automática el manejo de transacciones.

7.1.3 Gestión declarativa de las transacciones

Una de las ventajas principales de la arquitectura Enterprise JavaBeans es que permite la gestión declarativa de transacciones. Sin esta característica las transacciones deberían controlarse usando una demarcación explícita de la transacción. Esto conlleva el uso de APIs bastantes complejos como el `Object Transaction Service (OTS)` de `OMG`, o su implementación Java, el `Java Transaction Service (JTS)`. La demarcación explícita es difícil de usar correctamente para los desarrolladores, sobre todo si no están habituados a la programación de sistemas transaccionales. Además, la demarcación explícita requiere que el código transaccional se escriba junto con la lógica de negocio,

lo que reduce la claridad del código y, más importante, crea objetos distribuidos inflexibles. Una vez que la demarcación de la transacción está grabada en el objeto de negocio, los cambios en la conducta de transacción obligan a cambios en la misma lógica de negocio.

Con la gestión declarativa de transacciones, la conducta transaccional de los EJBs puede controlarse usando el descriptor del despliegue, que establece atributos de transacción para los métodos individuales del enterprise bean. Esto significa que la conducta transaccional de un EJB puede modificarse sin cambiar la lógica de negocio del EJB. Además, un EJB desplegado en una aplicación puede definirse con una conducta transaccional distinta que la del mismo bean desplegado en otra aplicación. La gestión declarativa de las transacciones reduce la complejidad del manejo de las transacciones para los desarrolladores de EJB y de aplicaciones y hace más sencilla la creación de aplicaciones transaccionales robustas.

El resto de este tema examina cómo los EJB soportan implícitamente las transacciones a través de atributos declarativos de transacciones.

7.2 Alcance de la transacción

El alcance de una transacción es un concepto crucial para comprender las transacciones. En este contexto, el alcance una transacción consiste en aquellos EJBs de entidad y de sesión que están participando en una transacción particular.

En el método `bookPassage()` del EJB `TravelAgent`, todos los EJBs que participan son parte del mismo alcance de transacción. El alcance de la transacción comienza cuando el cliente invoca el método `bookPassage()` del EJB `TravelAgent`. Una vez que el alcance de la transacción ha comenzado, éste se propaga a los dos EJB que se crean: `Reservation EJB` y `ProcessPayment EJB`.

Como ya hemos comentado, una transacción es una unidad-de-trabajo constituida por una o más tareas. En una transacción, todas las tareas que forman la unidad-de-trabajo deben ser un éxito para que la transacción en su totalidad tenga éxito; la transacción debe ser atómica. Si alguna tarea falla, las actualizaciones realizadas por las otras tareas deben desacerse. En EJB, las tareas se expresan como métodos de los enterprise bean, y una unidad-de-trabajo consiste en un conjunto de invocaciones a métodos los enterprise bean. El alcance de la transacción incluye todos los EJB que participan en la unidad de trabajo.

Es fácil trazar el alcance de una transacción siguiendo el hilo de ejecución. Si la invocación del método `bookPassage()` comienza una transacción, entonces de forma lógica, la transacción termina cuando el método se completa. El alcance de la transacción `bookPassage()` incluiría los EJB `TravelAgent`, `Reservation` y `ProcessPayment`.

Una transacción puede terminar si se arroja una excepción mientras que el método `bookPassage()` está en ejecución. La excepción puede arrojarse desde uno de los otros EJBs o desde el mismo método `bookPassage()`. Una excepción puede causar o no causar `rollback` (vuelta a los valores iniciales de los datos) dependiendo de su tipo. Lo veremos más adelante.

7.3 Atributos de transacción

Como desarrolladores de aplicaciones, no es necesario que controlemos explícitamente las transacciones cuando estamos usando un servidor EJB. Los servidores EJB pueden manejar las transacciones de forma implícita, basándose en los atributos transaccionales establecidos para los EJB en el momento del despliegue. La posibilidad de especificar cómo participan en las transacciones los objetos de negocio mediante programación basada en atributos es una característica común de los monitores de transacciones, y una de las características más importantes del modelo de componentes EJB.

Cuando un EJB se despliega, podemos establecer su atributo de transacción a uno de los siguientes valores:

- `NotSupported`
- `Supports`
- `Required`
- `RequiresNew`
- `Mandatory`
- `Never`

La especificación EJB 2.0 aconseja fuertemente que los beans de entidad con persistencia gestionada por el contenedor usen sólo los atributos `Required`, `RequiresNew` y `Mandatory`. Esta restricción asegura que todos los accesos a bases de datos suceden en el contexto de una transacción, lo cual es importante cuando el contenedor está gestionando automáticamente la persistencia.

Podemos establecer un atributo de transacción para el EJB completo (en cuyo caso se aplica a todos los métodos) o establecer distintos atributos de transacción para los métodos individuales. Lo primero es mucho más sencillo y conlleva menos riesgo de errores, pero lo segundo ofrece mayor flexibilidad. Los fragmentos de código de los siguientes apartados muestran cómo establecer los atributos de transacciones de un EJB en el descriptor del despliegue del EJB.

7.3.1 Descriptor del despliegue

En el descriptor XML del despliegue, un elemento `<container-transaction>` especifica los atributos de transacción para los EJBs descritos en el descriptor de despliegue:

```
<ejb-jar>
```



```
...
<assembly-descriptor>
  ...
  <container-transaction>
    <method>
      <ejb-name>TravelAgentEJB</ejb-name>
      <method-name> * </method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>TravelAgentEJB</ejb-name>
      <method-name>listAvailableCabins</method-name>
    </method>
    <trans-attribute>Supports</trans-attribute>
  </container-transaction>
  ...
</assembly-descriptor>
...
</ejb-jar>
```

Este descriptor de despliegue especifica los atributos de transacción para el EJB TravelAgent. Cada elemento `<container-transaction>` especifica un método y el atributo de transacción que debería aplicarse a ese método. El primer elemento `<container-transaction>` especifica que todos los métodos tengan por defecto un atributo de transacción de Required; el carácter * es un comodín que indica todos los métodos del EJB TravelAgent. El segundo elemento `<container-transaction>` hace caso omiso del atributo por defecto para especificar que el método `listAvailableCabins()` tendrá un atributo de transacción Supports. Tenemos que especificar a qué EJB nos estamos refiriendo con el elemento `<ejb-name>`, ya que un descriptor XML de despliegue puede contener muchos EJBs.

7.3.2 Definición de los atributos de transacción

Vamos a definir ahora los atributos de transacción que hemos declarado previamente. Tal y como comentamos arriba, los atributos recomendados por la especificación EJB 2.0 son Required, RequiresNew y Mandatory

Required

Este atributo significa que el método del enterprise bean debe invocarse dentro del alcance de una transacción. Si el cliente o el EJB que realiza la llamada es parte de una transacción el EJB con el atributo Required automáticamente se incluye en el alcance de esa transacción. Si, sin embargo, el cliente o EJB que realiza la llamada no está incluido en una transacción, el EJB con el atributo Required comienza su propia transacción. Esta nueva transacción concluye cuando el EJB termina.

RequiresNew

Este atributo significa que se debe comenzar siempre una nueva transacción. Independientemente de si el cliente o EJB que realiza la

llamada es parte de una transacción, un método con el atributo `RequiresNew` siempre comienza una nueva transacción. Si el cliente que realiza la llamada ya está incluido en una transacción, esa transacción se suspende hasta que la llamada al método con el atributo `RequiresNew` finaliza. En ese momento la transacción original vuelve a estar activa.

Mandatory

Este atributo significa que el método del enterprise bean debe siempre ser parte del alcance de la transacción del cliente que realiza la llamada. Si este cliente o EJB no es parte de una transacción, la invocación fallará, arrojándose una excepción `javax.transaction.TransactionRequiredException` a los clientes remotos o una excepción `javax.ejb.TransactionRequiredLocalException` a los clientes locales.

7.4 Propagación de la transacción

Para ilustrar el impacto de los atributos de transacción sobre los métodos del enterprise bean, miraremos una vez más al método `bookPassage()` del EJB `TravelAgent`.

Para que `bookPassage()` se ejecute como una transacción con éxito, tanto la creación del EJB `Reservation` como el cargo a la tarjeta de crédito deben también terminar con éxito. Esto significa que ambas operaciones deben incluirse en la misma transacción. Si alguna operación falla, la transacción completa falla. Podríamos haber especificado un atributo de transacción `Required` como el atributo por defecto para todos los EJBs incluidos, porque ese atributo refuerza la política deseada de que todos los EJBs deben ejecutarse dentro de una transacción y por ello asegura la consistencia de los datos.

Como un monitor de transacciones, el servidor EJB vigila cada llamada a un método en la transacción. Si cualquiera de las actualizaciones falla, todas las actualizaciones a todos los EJBs serán rebobinadas o recuperadas (rolled back). Una recuperación (rollback) es como un comando undo. Si hemos trabajado con bases de datos relacionales, el concepto de una recuperación debe sernos familiar. Una vez que se ejecuta una actualización (update), podemos o bien asegurar esa actualización (commit) o bien recuperar los datos iniciales (rollback). El uso de EJBs transaccionales nos proporciona el mismo tipo de control rollback/commit. Por ejemplo, si el EJB `Reservation` no pudiera crearse, el cargo realizado por el EJB `ProcessPayment` es descontado (rolled back).

En los casos en los que el contenedor gestiona implícitamente la transacción, las decisiones de commit y rollback se manejan de forma automática. Supongamos que el EJB `TravelAgent` se crea y se usa en el cliente como sigue:

```
TravelAgent agent = agentHome.create(customer);
agent.setCabinID(cabin_id);
agent.setCruiseID(cruise_id);
try {
    agent.bookPassage(card, price);
} catch (Exception e) {
    System.out.println("Transaction failed!");
}
```

Más aún, asumamos que el método `bookPassage()` tienen un atributo de transacción `RequiresNew`. En este caso, el cliente que invoca el método `bookPassage()` no es parte de la transacción. Cuando se invoca a `bookPassage()` en el EJB `TravelAgent`, se crea una nueva transacción, tal y como dicta el atributo `RequiresNew`. Esto significa que el EJB `TravelAgent` se registra en el gestor de transacciones del servidor de EJB, el cual gestionará la transacción automáticamente. El gestor de transacciones coordina transacciones, propagando el alcance de la transacción desde un EJB al siguiente para asegurarse de que todos los EJBs tocados por una transacción se incluyen en la unidad-de-trabajo de la transacción. De esta forma, el gestor de transacciones puede monitorizar las actualizaciones realizadas por cada enterprise bean y decidir, basándose en el éxito de estas actualizaciones, si hacer permanentes los cambios hechos por todos los enterprise beans en las bases de datos o si echarlos atrás y deshacerlos. Si una excepción del sistema se arroja en el método `bookPassage()`, la transacción es automática deshecha (rolled back).

Cuando el método `byCredit()` se invoca dentro del método `bookPassage()`, el EJB `ProcessPayment` se registra en el gestor de transacciones bajo el contexto transaccional que se creó para el EJB `TravelAgent`. Cuando se crea el nuevo EJB `Reservation`, también se registra en el gestor de transacciones bajo la misma transacción. Cuando se registran todos los EJBs y se realizan todas las actualizaciones, el gestor de transacciones chequea todo para asegurarse de que sus actualizaciones funcionarán. Si uno de los EJB devuelve un error o falla, los cambios realizados por los EJB `ProcessPayment` o `Reservation` se deshacen por el gestor de transacciones.

Además de gestionar las transacciones en su mismo entorno, un servidor EJB puede coordinarse con otros sistemas transaccionales. Si, por ejemplo, el EJB `ProcessPayment` realmente viniera de un servidor EJB distinto, los dos servidores EJB cooperarían para gestionar la transacción como una unidad-de-trabajo. Esto se llama una transacción distribuida.

Una transacción distribuida es mucho más complicada, y requiere lo que se llama two-phase commit (2-PC o TPC). 2-PC es un mecanismo que permite que una transacción sea gestionada a través de distintos servidores y recursos (por ejemplo, bases de datos y proveedores JMS). Los detalles de un sistema 2-PC están más allá del alcance de este tema. Si se soportan las transacciones distribuidas, se soportará el mismo protocolo para propagarlas. No nos daremos cuenta, como desarrolladores de EJBs o de aplicaciones, de las diferencias entre transacciones locales y distribuidas.

7.5 Relaciones basadas en colecciones y transacciones

En la persistencia gestionada por el contenedor EJB 2.0, las relaciones basadas en colecciones pueden accederse sólo dentro de una única transacción. En otras palabras, no es legal obtener un objeto Collection de un campo de relación basado en colecciones en una transacción y usarlo en otra.

Por ejemplo, si un enterprise bean accede a otro campo de relación basado en una colección a través de su interface local, el objeto Collection devuelto del método de acceso sólo puede usarse dentro de la misma transacción:

```
public class HypotheticalBean implements javax.ejb.EntityBean {  
  
    public void methodX(CustomerLocal customer) {  
  
        Collection reservations = customer.getReservations();  
        Iterator iterator = reservations.iterator();  
        while(iterator.hasNext()) {  
            ...  
        }  
        ...  
    }  
    ...  
}
```

Si se hubiera declarado como RequiresNew el atributo del método getReservations() del EJB Customer, cualquier intento de invocar algún método en el objeto Collection, incluyendo el método iterator(), resultará en una java.lang.IllegalStateException. Esta excepción se arroja debido a que el objeto Collection fue creado dentro del alcance de la transacción getReservations(), no en el alcance de la transacción del metodoX().

El objeto Collection de un bean de entidad puede usarse por otro bean co-ubicado sólo si se obtiene y se accede en el mismo contexto de transacción. En tanto que el método getReservations() del EJB Customer propage el contexto de transacción del metodoX(), el objeto Collection se puede usar sin problemas. Esto se puede conseguir cambiando el método getReservations() para que se declare su atributo de transacción como Required o Mandatory.

Tema 8: Seguridad

8.1 Introducción a la seguridad en EJBs

Los servidores Enterprise JavaBeans pueden soportar tres tipos de seguridad: autenticación, comunicación segura y control de acceso. En el módulo de seguridad se ha hecho hincapié en los dos primeros tipos de servicios usando las APIs que proporciona Java. En este apartado veremos qué mecanismos define la especificación EJB para el control de acceso a los EJBs. Revisemos brevemente cada uno de los tipos de seguridad.

Autenticación

Dicho sencillamente, la autenticación valida la identidad del usuario. La forma más común de autenticación es una simple ventana de login que pide un nombre de usuario y una contraseña. Una vez que los usuarios han pasado a través del sistema de autenticación, pueden usar el sistema libremente, hasta el nivel que les permita el control de acceso. La autenticación se puede basar también en tarjetas de identificación, certificados y en otros tipos de identificación.

Comunicación segura

Los canales de comunicación entre un cliente y un servidor son un elemento muy importante en la seguridad del sistema. Un canal de comunicación puede hacerse seguro mediante aislamiento físico (por ejemplo, via una conexión de red dedicada) o por medio de la encriptación de la comunicación entre el cliente y el servidor. El aislamiento físico es caro, limita las posibilidades del sistema y es casi imposible en Internet, por lo que lo más usual es la encriptación. Cuando la comunicación se asegura mediante la encriptación, los mensajes se codifican de forma que no puedan ser leídos ni manipulados por individuos no autorizados. Esto se suele conseguir mediante el intercambio de claves criptográficas entre el cliente y el servidor. Las claves permiten al receptor del mensaje decodificarlo y leerlo.

Control de acceso

El control de acceso (también conocido como autorización) aplica políticas de seguridad que regulan lo que un usuario específico puede y no puede hacer en el sistema. El control de acceso asegura que los usuarios accedan sólo a aquellos recursos a los que se les ha dado permiso. El control de acceso puede restringir el acceso de un usuario a subistemas, datos, y objetos de negocio. Por ejemplo, a algunos usuarios se les puede dar permiso de modificar información, mientras que otros sólo tienen permiso de visualizarla.

La mayoría de los servidores EJB soportan la comunicación segura a través del protocolo SSL (Secure Socket Layer) y proporcionan algún mecanismo de autenticación, pero la especificación Enterprise JavaBeans sólo especifica el control de acceso a los EJBs.

Aunque la autenticación no se especifica en EJB, a menudo se consigue usando el API JNDI. Un cliente que usa JNDI puede proporcionar información de autenticación usando este API para acceder a los recursos del servidor. Esta información se suele pasar cuando el cliente intenta iniciar una conexión JNDI con el servidor EJB. El siguiente código muestra cómo se añaden la contraseña y el nombre de usuario a las propiedades de la conexión que se usan para obtener una conexión JNDI con el servidor EJB:

```
properties.put(Context.SECURITY_PRINCIPAL, userName );
properties.put(Context.SECURITY_CREDENTIALS, userPassword);

javax.naming.Context jndiContext = new
javax.naming.InitialContext(properties);
Object ref= jndiContext.lookup("AccountEJB");
AccountHome accountHome = (AccountHome)
    PortableRemoteObject.narrow(ref, AccountHomeRemote.class);
```

EJB especifica que todas las aplicaciones clientes que acceden a un sistema EJB deben estar asociadas con una identidad de seguridad. La identidad de seguridad representa el cliente o bien como un usuario o bien como un rol. Un usuario podría ser una persona, una credencial de seguridad, un computador o incluso una tarjeta inteligente. Normalmente, el usuario es una persona a la que se le asigna una identidad cuando entra en el sistema. Un rol especifica una categoría de acceso a la que pueden pertenecer distintos usuarios. Por ejemplo, el rol "ReadOnly" definiría una categoría en la que sólo se permite el acceso a operaciones de lectura de datos.

Normalmente el servidor EJB permite definir grupos de usuarios a los que se les va a asignar las mismas restricciones de seguridad. ¿Cuál es la diferencia entre roles y grupos? La diferencia fundamental es que los roles son dependientes de la aplicación que se despliega, mientras que los grupos son dependientes de la organización. Después de desplegar una aplicación es necesario realizar una asignación de roles a usuarios o grupos de usuarios. Esta separación entre roles y usuarios permite hacer la aplicación portable e independiente de la organización en la que se despliega.

Cuando un cliente remoto se autentifica en el sistema EJB, se le asocia una identidad de seguridad que dura el tiempo que está activa la sesión. Esta identidad se encuentra en una base de datos o directorio específico de la plataforma EJB. Esta base de datos o directorio es responsable de almacenar las identidades individuales, su pertenencia a grupos y las asignaciones de roles a grupos e individuos.

Una vez que se le ha asociado una identidad de seguridad a un cliente remoto, está listo para usar los beans. El servidor EJB realiza un seguimiento de cada cliente y de su identidad. Cuando un cliente invoca un método de un EJB, el

servidor EJB pasa implícitamente la identidad del cliente junto con la invocación al método. Cuando el objeto EJB o el EJB home recibe la invocación debe chequear la identidad para asegurarse de que ese cliente puede usar ese método.

8.2 Control de acceso basado en roles

En Enterprise JavaBeans, la identidad de seguridad se representa por un objeto `java.security.Principal`. Este objeto actúa como representante de usuarios, grupos, organizaciones, tarjetas inteligentes, etc. frente a la arquitectura de control de acceso de los EJB.

Los descriptores del despliegue incluyen elementos que declaran a qué roles lógicos se permite el acceso a los métodos de los enterprise beans. Estos roles se consideran lógicos porque no reflejan directamente usuarios, grupos o cualquier otra identidad de seguridad en un entorno operacional específico. Los roles se hacen corresponder con grupos de usuarios o usuarios del mundo real cuando el bean se despliega. Esto permite que el bean sea portable ya que cada vez que el bean se despliega en un nuevo sistema, los roles se asignan a usuarios y grupos específicos de ese entorno operacional.

He aquí una porción de un fichero de despliegue de un EJB que define dos roles de seguridad, `ReadOnly` y `Administrator`:

```
<security-role>
  <description>
    A este rol se le permite ejecutar cualquier método del
    bean y leer y escribir cualquier dato del bean.
  </description>
  <role-name>
    Administrator
  </role-name>
</security-role>

<security-role>
  <description>
    A este rol se le permite localizar y leer información del
    bean.
    A este rol no se le permite modificar los datos del bean.
  </description>
  <role-name>
    ReadOnly
  </role-name>
</security-role>
```

Los nombres de los roles en este descriptor no son nombres reservados o especiales con significados predefinidos; son simplemente nombres lógicos escogidos por el ensamblador del bean.

Una vez que se declaran los roles, pueden asociarse con métodos en los beans. El elemento `<method-permission>` es en el que definimos a qué métodos puede acceder cada uno de los roles.

```
<method-permission>
  <role-name>Administrator</role-name>
  <method>
    <ejb-name>CustomerEJB</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
<method-permission>
  <role-name>ReadOnly</role-name>
  <method>
    <ejb-name>CustomerEJB</ejb-name>
    <method-name>getName</method-name>
  </method>
  <method>
    <ejb-name>CustomerEJB</ejb-name>
    <method-name>getAddress</method-name>
  </method>
  <method>
    <ejb-name>CustomerEJB</ejb-name>
    <method-name>findByPrimaryKey</method-name>
  </method>
</method-permission>
```

En el primer `<method-permission>` el rol `Administrator` se asocia con todos los métodos del EJB `CustomerEJB`. En el segundo, se limita el acceso del rol `ReadOnly` sólo a tres métodos: `getName()`, `getAddress()` y `findByPrimaryKey()`. Cualquier intento de un rol `ReadOnly` de acceder a un método que no sea éstos resultará en una excepción.

Al desplegar el bean debemos examinar el bean y asignar cada rol lógico a cada grupo de usuarios en el entorno operacional, sin preocuparnos de qué métodos son accesibles por cada rol. Esto muestra una vez más las ventajas de la separación de papeles que promueve la arquitectura EJB. El desarrollador del bean es el que debe preocuparse de definir los roles y restringir el acceso a los métodos, mientras que la persona que hace el despliegue se ocupa de asignar los roles a usuarios y grupos.

Una vez que el bean se despliega, el contenedor se encarga de comprobar que los usuarios acceden únicamente a aquellos métodos a los que tienen permisos. Esto se consigue propagando la identidad de seguridad, el objeto `Principal`, en cada invocación del cliente al bean. Cuando el cliente invoca un método de un bean, el objeto `Principal` del cliente se chequea para comprobar si el rol asociado tiene los privilegios necesarios.

Si un bean intenta acceder a cualquier otro bean mientras que está sirviendo a un cliente, pasará al segundo bean la identidad del cliente para que se realiza un control de acceso en este segundo bean. De esta manera, el `Principal` del cliente se propaga de un bean al siguiente, asegurando que se controla el acceso acceda o no directamente al EJB.

8.3 Métodos no-chequeados

En EJB 2.0 un conjunto de métodos se puede designar como `unchecked`, que significa que los permisos de seguridad no se chequean. Un método no chequeado puede invocarse por cualquier cliente, sin importancia de el rol que ese cliente esté usando. A continuación hay un ejemplo de declaración de métodos no-chequeados:

```
<method-permission>
  <unchecked/>
  <method>
    <ejb-name>CabinEJB</ejb-name>
    <method-name>*</method-name>
  </method>
  <method>
    <ejb-name>CustomerEJB</ejb-name>
    <method-name>findByPrimaryKey</method-name>
  </method>
</method-permission>
<method-permission>
  <role-name>administrator</role-name>
  <method>
    <ejb-name>CabinEJB</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

Esta declaración nos dice que todos los métodos del EJB Cabin, junto con el método `findByPrimaryKey()` del EJB Customer, son no-chequeados. Un segundo elemento le da al administrador permiso para acceder a todos los métodos del EJB Cabin. El permiso no-chequeado siempre tiene prioridad sobre cualquier otro permiso.

8.4 La identidad de seguridad runAs

Mientras que los elementos `<method-permission>` especifican qué `Principals` tienen acceso a qué métodos del bean, el elemento `<security-identity>` especifica bajo qué `Principal` el método va a ejecutarse. En otras palabras, el objeto `Principal runAs` se usa como la identidad del enterprise bean cuando éste intenta invocar métodos en otros beans. Esta identidad no tiene por qué ser la misma que la identidad que accede al bean por primera vez.

Por ejemplo, los siguientes elementos del descriptor de despliegue declaran que el método `create()` sólo puede accederse por el rol `JimSmith` y que el EJB Cabin siempre se va ejecutar con una identidad `Administrator`.

```
<enterprise-beans>
...
  <entity>
    <ejb-name>CabinEJB</ejb-name>
    ...
    <security-identity>
```

```

        <run-as>
            <role-name>Administrator</role-name>
        </run-as>
    </security-identity>
    ...
</entity>
...
</enterprise-beans>
<assembly-descriptor>
<security-role>
    <role-name>Administrator</role-name>
</security-role>
<security-role>
    <role-name>JimSmith</role-name>
</security-role>
...
<method-permission>
    <role-name>JimSmith</role-name>
    <method>
        <ejb-name>CabinEJB</ejb-name>
        <method-name>create</method-name>
    </method>
</method-permission>
...
</assembly-descriptor>

```

Esta clase de configuración es útil cuando el enterprise bean o los recursos accedidos en el cuerpo del método requieren un `Principal` distinto del que ha sido usado para obtener acceso al método. Por ejemplo, el método `create()` podría llamar a un método en el enterprise bean X que requiera la identidad de seguridad de `Administrator`. Si quisiéramos usar el enterprise bean X en el método `create()`, pero sólo permitimos a Jim Smith la creación de nuevos camarotes, usaríamos conjuntamente los elementos `<method-permission>` y `<security-identity>` para conseguirlo. El `<method-permission>` para `create()` especificaría que sólo Jim Smith puede invocar el método, y el elemento `<security-identity>` especificaría que el enterprise bean siempre se debe ejecutar bajo la identidad de seguridad `Administrator`. Para especificar que un enterprise bean debe ejecutarse bajo la identidad de quien hace la llamada, el rol `<security-identity>` contiene un único elemento vacío, el elemento `<use-caller-identity>`. Por ejemplo las siguientes declaraciones especifican que el EJB Cabin se ejecute siempre bajo la identidad de quien hace la llamada, por lo que si Jim Smith invoca el método `create()`, el bean se ejecutará bajo la identidad de seguridad `JimSmith`:

```

<enterprise-beans>
...
    <entity>
        <ejb-name>CabinEJB</ejb-name>
        ...
        <security-identity>
            <use-caller-identity/>
        </security-identity>
        ...
    </entity>
...

```

```
</enterprise-beans>
```

Podemos entender la secuencia de cambio de identidades de la siguiente forma:

1. El cliente invoca el método X del bean con una identidad `Id1`.
2. El bean comprueba si la identidad `Id1` tiene permiso para ejecutar el método X. La tiene.
3. El bean consulta el elemento `<security-identity>` y cambia la identidad a la que indica ese elemento. Supongamos que es la identidad `Id2`.
4. El bean realiza las llamadas del método X con la identidad `Id2`.

Tema 9: Buenas prácticas con EJBs

9.1 Diseño

El diseño de la aplicación es clave en el desarrollo de la misma. Un error de implementación es sencillo de reparar, pero un error de diseño obliga a replantear completamente el proyecto. A continuación se enumeran unas recomendaciones sobre el diseño de aplicaciones con EJBs.

9.1.1 Asegurarse de que los EJBs son necesarios

El uso de EJBs obliga a ciertos compromisos de eficiencia, implementación o costes de desarrollo. Tenemos que asegurarnos de que la aplicación realmente necesita estos componentes distribuidos. Algunas condiciones necesarias para que la aplicación use EJBs son las siguientes.

Condiciones de diseño

La aplicación debe separarse de forma natural en capas estándar de persistencia, objetos de dominio, lógica de negocio y presentación. La aplicación debe tener componentes de intranet y de extranet.

Condiciones de implementación

La implementación de los EJBs es algo costosa. Un EJB típico consta de cuatro ficheros (interfaz home, remota, implementación y fichero descriptor del despliegue). Debemos tener cierto tiempo para poder acometer esa implementación. Si necesitamos hacer un prototipo rápido es posible usar JSPs y servlets o JavaBeans y después reescribir el código para incluir los EJBs.

Condiciones de eficiencia

Los servidores de aplicaciones están diseñados para proporcionar escalabilidad. Los servicios que proporcionan (manejo de transacciones, pooling de objetos, seguridad) son muy útiles para implementar aplicaciones escalables, pero también usan una gran cantidad de recursos. Si la aplicación no necesita de estos servicios y de esta escalabilidad, el uso de EJBs puede ralentizar la aplicación.

9.1.2 Usar arquitecturas de diseño estándar

La importancia de la realización de un diseño previo de la aplicación, antes de proceder a su implementación, es aun mayor en la arquitectura J2EE. La gran cantidad de distintos componentes que es posible usar en este tipo de aplicaciones obliga a pensar detenidamente cómo dividir la aplicación y dónde usar cada tipo de componente.

El primer principio de diseño J2EE es el de usar una estructura en capas, con los siguientes componentes:

Capa de presentación

Consiste en la interfaz de usuario de la aplicación. Normalmente contiene ficheros JSP, applets y alguna lógica de presentación que se puede resolver con etiquetas JSP o con alguna solución propia que incluya XML para garantizar la portabilidad de esta capa. Esta capa se considera un cliente de la capa de negocio, porque usa exclusivamente esta capa para completar sus operaciones.

Capa de lógica de negocio

Esta es la capa más importante de la aplicación, al menos desde el punto de vista del programador de EJBs. Esta capa contiene el flujo de trabajo del negocio y servicios varios usados por el cliente. Se basa en la capa de persistencia para el almacenamiento y recuperación de los datos.

Capa de persistencia

La capa de persistencia se ocupa del almacenamiento y la recuperación de los datos en la base de datos del sistema informático. La mayoría del código de esta capa se implementa con EJBs de entidad, y posiblemente algunos tipos de datos relacionados con estos beans de entidad, como son los objetos de datos de acceso (data access objects).

El segundo principio de diseño tiene que ver con el orden en el se desarrollan las capas de la aplicación. El orden más común es el que sigue:

1. Definir los requerimientos y los casos de uso. Esto ayuda a entender qué es lo que la aplicación tiene que hacer y cómo de flexible debe ser, así como puede ayudar a decidir qué tecnologías usar.
2. Definir claramente los objetos del dominio y las interfaces de negocio que serán expuestas a los clientes.
3. Escribir los stubs de algunos componentes, para poder comenzar el desarrollo.
4. Implementar la capa de persistencia.
5. Definir y escribir *servicios*, que son componentes independientes del sistema que se usan en la implementación.
6. Implementar la capa de lógica de negocio.
7. Implementar la capa de presentación.

9.1.3 Usar Beans de entidad CMP

Siempre que sea posible hay que intentar usar persistencia gestionada por el contenedor (CMP) para los beans de entidad. La mayoría de servidores de aplicaciones tienen mecanismos optimizados para el manejo de este tipo de beans. Entre los beneficios del uso de la CMP se encuentran:

- Gestión optimizada de las transacciones
- No uso de SQL ni de lógica de persistencia en el código fuente
- Relaciones gestionadas por el contenedor entre los beans de entidad

9.1.4 Usar patrones de diseño

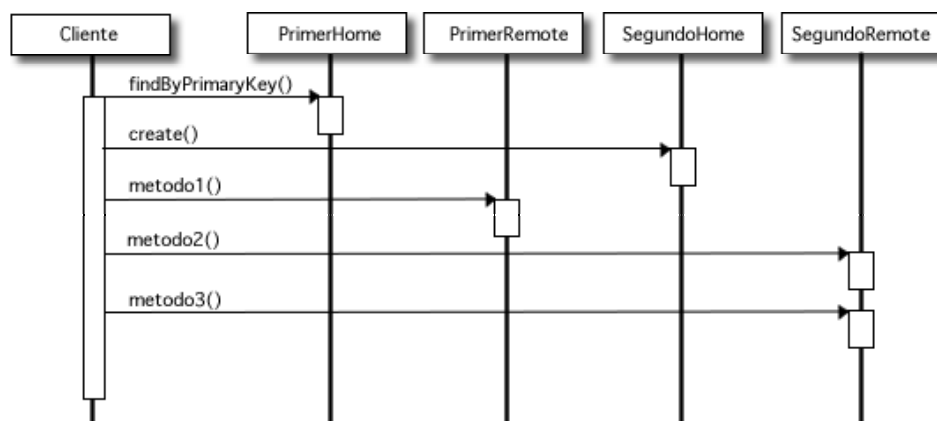
Los patrones de diseño son imprescindibles; hay que aprender y usar tantos patrones de diseño de EJBs como sea posible. La mayoría de los patrones ahorrarán tiempo de desarrollo, mejorarán la eficiencia de la aplicación y la harán más mantenible. Casi es posible asegurar que sin patrones de diseño sería casi imposible escribir buenas aplicaciones J2EE.

Como ejemplo, veamos rápidamente dos patrones muy comunes: *session façade* y *value objects*.

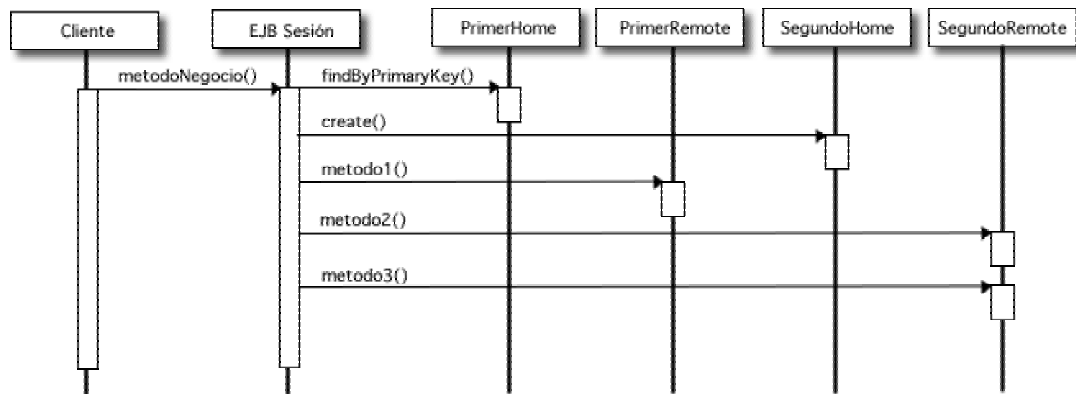
Session façade

El patrón *session façade* es el patrón de diseño EJB que se usa más frecuentemente. Representa una forma de encapsular la lógica de negocio para obtener mejor eficiencia y un código más mantenible. La idea básica es muy simple: poner toda la lógica de negocio en beans de sesión sin estado y hacer que los clientes llamen a estos beans, en lugar de a los distintos componentes de la aplicación. Las figuras siguientes explican este concepto.

Antes de aplicar el patrón:



Después de aplicar el patrón:

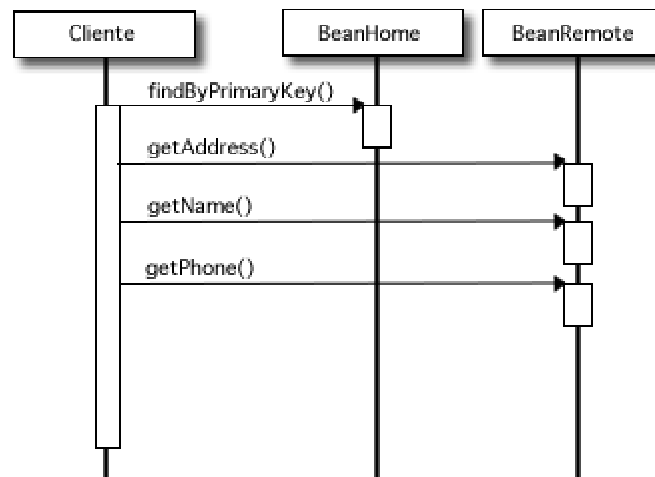


Este patrón tiene muchas ventajas. La más significativa es que al mover toda la lógica de negocio a una capa adicional obtenemos código mucho más limpio y manejable. Debido a que cada operación de flujo de trabajo se corresponde con un método de negocio en el bean de sesión, la ejecución de dicha operación se realiza bajo una transacción. Además, al definir el bean de sesión, podemos usar interfaces locales para todas las operaciones entre éste y los beans de entidad y sólo mostrar las interfaces remotas del bean de sesión.

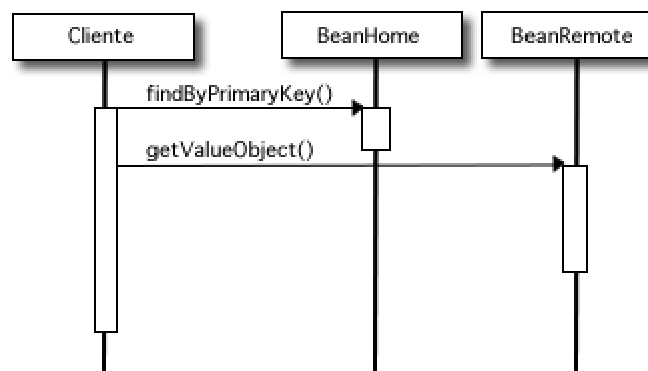
Value objects

El patrón *value objects* (objetos de valor) o *data transfer objects* (objetos de transferencia de datos), representa una forma de transferir datos entre los componentes remotos. Por ejemplo, supongamos que definimos un bean de entidad `User`. Para obtener el nombre del usuario, su apellido, su teléfono, su dirección, y otros datos, deberíamos usar al método `get` del bean para cada dato que queramos obtener. Si hacemos esto a través de la interfaz remota, la sobrecarga en la red será muy grande. La solución natural para este problema es crear un objeto que pueda mantener todos los datos que necesitamos, y usar ese objeto para transferir los datos. Esto es exactamente lo que hace un *value object*.

Antes de aplicar el patrón:



Después de aplicar el patrón:



Una práctica común es usar el *value object* en el bean de entidad, en lugar de construirlo cada vez que el cliente lo pide. Otra práctica común es exponer distintos *value objects* de un simple bean de entidad, de forma que los clientes sólo obtengan los datos en que están interesados.

Cuando tenemos un número grande de value objects y su manejo se vuelve tedioso, podría ser interesante implementar un value object genérico `HashMap` que pueda transmitir datos arbitrarios. Sin embargo, un value object genérico tendría sólo una pareja de métodos `getField/setField`, y sería complicado de implementar algún tipo de validación que nos asegurara que se están definiendo los campos correctos en cada momento. Por el contrario, un value object normal tiene una pareja de métodos `getXxx/setXxx` para cada campo, lo que hace más difícil cometer errores. Un compromiso es implementar todos los *value objects* como interfaces y luego usar un *DynamicProxy* con un `HashMap` para almacenar los campos de los value objects. El ejemplo siguiente muestra una implementación de un proxy dinámico y cómo se usa en un bean de entidad.


```

public class ValueObjectProxy implements InvocationHandler,
Serializable
{
    protected HashMap fieldMap;
    protected Class valueObjectClass;

    protected ValueObjectProxy (Class valueObjectClass) {
        this.valueObjectClass = valueObjectClass;
        fieldMap = new HashMap( );
    }

    public static Object createValueObject (Class valueObjectClass) {
        return Proxy.newProxyInstance (
            valueObjectClass.getClassLoader( ),
            new Class[ ] {valueObjectClass},
            new ValueObjectProxy(valueObjectClass));
    }

    public Object invoke (Object proxy, Method method, Object[ ] args)
    throws Exception {
        String methodName = method.getName( );

        if (methodName.startsWith ("get")) {
            // Remove "get" to get the field name.
            String fieldName = methodName.substring(3);

            // It's a get, so return the value.
            if (!fieldMap.containsKey ("fieldName"))
                throw new ValueObjectException ("Field " +
fieldName
+ " does not
exist");

            return fieldMap.get(fieldName);

        } else if (methodName.startsWith ("set")) {
            // Remove "set" to get the field name.
            String fieldName = methodName.substring(3);

            // Put it into the hashmap.
            // Assume we received one argument in the set method.
            fieldMap.put (fieldName, args[0]);

            // It's neither a get nor a set.
        } else {
            throw ValueObjectException ("Invalid method");
        }
    }
}

public SomeBean implements EntityBean
{
    // Skipping irrelevant methods . . .

    public SomeValueObject getValueObject( )
    {
        // Create the value object.
        SomeValueObject vo = (SomeValueObject)

```

```
        ValueObjectProxy.createValueObject  
(SomeValueObject.class);  
  
        // Set its values.  
        vo.setName ("John Smith");  
        vo.setAddress ("140 Maple Drive");  
  
        return vo;  
    }  
}
```

9.2 Implementación

Algunas reglas importantes que conviene seguir cuando estamos implementando EJBs son las siguientes.

9.2.1 Usar interfaces locales para los Beans de Entidad

Tal y como hemos comentado en el patrón session façade, es una buena idea usar beans de sesión para que hagan de interfaz de negocio de los beans de entidad. De esta forma, dado que todos los beans de entidad van a comunicarse entre ellos o ser usados por beans de sesión, es conveniente usar interfaces locales en todos los beans de entidad.

9.2.2 Usar interfaces de negocio

Ya hemos comentado que la implementación de un EJB es compleja: consta de cuatro ficheros. Es muy usual cometer errores cuando se escribe el fichero de implementación y el fichero remoto (o local, en el caso de los EJBs de entidad) y llamar a los métodos de forma distinta o usar distintos tipos de parámetros. Este error sólo se va a hacer evidente en tiempo de despliegue. ¿Cómo conseguir que estos errores aparezcan en tiempo de compilación? Una idea es la de usar las denominadas interfaces de negocio.

En una interfaz de negocio definimos los perfiles de los métodos de negocio, que se deben corresponder también con los métodos de la clase de implementación. Después hacemos que la interfaz remota o local extienda la interfaz de negocio y que la clase bean implemente esta interfaz de negocio. Debido a que las interfaces remotas arrojan una `RemoteException`, la interfaz de negocio también debe hacerlo. También, todos los parámetros de los métodos deben ser serializables.

```
// Interfaz de negocio  
  
public interface Order  
{  
    public int getQuantity( ) throws RemoteException;  
    public void setQuantity (int quantity) throws RemoteException;  
    public double getPricePerItem( ) throws RemoteException;  
    public void setPricePerItem (double price) throws RemoteException;  
  
    public double getTotalPrice( ) throws RemoteException;  
}
```

```
// Interfaz remota

public interface OrderRemote extends Order, EJBObject
{
    // Todos los metodos se heredan de Order y EJBObject.
}

// Implementacion

public class OrderBean extends EntityBean implements Order
{
    private int quantity;
    private double pricePerItem;

    // Business interface implementation

    public int getQuantity( ) {
        return quantity;
    }
    public void setQuantity (int quantity) {
        this.quantity = quantity;
    }
    public double getPricePerItem( ) {
        return pricePerItem;
    }
    public void setPricePerItem (double price) {
        this.pricePerItem = pricePerItem;
    }
    public double getTotalPrice( ) {
        return quantity*pricePerItem;
    }

    // Other EntityBean methods go here . . .
}
```

Es importante tener cuidado en no usar las interfaces de negocio en el lado del cliente. Allí debemos seguir usando la interfaz `OrderRemote` y la clase `OrderBean`. Sólo debemos usar esta técnica si queremos asegurarnos de detectar los errores en tiempo de compilación.

9.2.3 Manejar las excepciones correctamente

El manejo de las excepciones en un entorno distribuido J2EE puede ser confuso y complicado. Debido a que la mayoría de las excepciones nunca se disparan en tiempo de producción, se suele tender a ignorarlas o, sencillamente, a imprimirlas. Sin embargo, si queremos escribir aplicaciones robustas, debemos realizar un control estricto de las condiciones en las que pueden aparecer estas excepciones.

Es útil separar las distintas excepciones de los EJB en tres tipos básicos.

RemoteException

Esta excepción se declara en todos los interfaces remotos expuestos por un EJB. Debe ser capturada por los clientes de los EJBs y normalmente se originan por un problema de conexión a la red. Las clases que implementan

EJBs no pueden arrojar este tipo de excepción. Si se usa una conexión remota entre EJBs y sucede esta excepción, se debería propagar la excepción al cliente envolviéndola en una excepción `EJBException`.

EJBException y sus subclases

Esta excepción se lanza por el desarrollador en la implementación del EJB y es capturada por el contenedor. Normalmente, lanzaremos una excepción de este tipo cuando se ha producido un error importante, en cuyo caso el contenedor siempre hará un rollback de la transacción actual. Esta excepción se debería tratar como una `NullPointerException`: es una excepción generada en tiempo de ejecución y en general no debe ser capturada por el desarrollador.

Excepciones definidas en la aplicación

A diferencia de las excepciones previas, las excepciones definidas por la aplicación deben ser consideradas como normales desde el punto de vista del contenedor. Un EJB puede declarar y lanzar estas excepciones, y los clientes las capturarán como si fueran una excepción Java normal. Debido a que las excepciones se deben mover por la red (desde el EJB hasta el cliente) deben implementar la interfaz `Serializable`. Ejemplos de este tipo de excepciones son las que ya vienen predefinidas en la arquitectura EJB, como `CreateException` o `FinderException`.

9.2.4 Cachear los objetos JNDI Lookup

Para obtener un `DataSource` o un interfaz home de un EJB, es necesario crear un `InitialContext`, y después buscar (lookup) en él recurso que necesitamos. Estas operaciones son normalmente muy costosas, considerando el hecho de que se deben repetir frecuentemente. Una forma de optimizar estas peticiones es realizar la búsqueda sólo una vez y guardar en una caché el resultado. El siguiente código hace esto.

```
public class EJBHomeCache
{
    private static EJBHomeCache instance;

    protected Context ctx = null;
    protected FirstEJBHome firstHome = null;
    protected SecondEJBHome secondHome = null;

    private EJBHomeCache( )
    {
        try {
            ctx = new InitialContext( );
            firstHome = (FirstEJBHome)PortableRemoteObject.narrow
(
                ctx.lookup ("java:comp/env/FirstEJBHome"),
                FirstEJBHome.class);
            secondHome =
(SecondEJBHome)PortableRemoteObject.narrow (
                ctx.lookup ("java:comp/env/SecondEJBHome"),
                FirstEJBHome.class);
```

```
        } catch (Exception e) {
            // Handle JNDI exceptions here, and maybe throw
            // application-level exception.
        }
    }
    public static synchronized EJBHomeCache getInstance( )
    {
        if (instance == null) instance = new EJBHomeCache( );
        return instance;
    }
    public FirstEJBHome getFirstEJBHome( )
    {
        return firstHome;
    }

    public SecondEJBHome getSecondEJBHome( )
    {
        return secondHome;
    }
}
```

9.2.5 Usar Business Delegates como clientes

Un *business delegate* es una simple clase Java que delega todas las llamadas a un EJB. Podría parecer demasiado simple para ser útil, pero sin embargo es un patrón de diseño muy usado. La principal razón para usar un business delegate es separar la lógica de manejo del EJB (por ejemplo, la obtención de la interfaz remota, el manejo de las excepciones remotas, etc.) de los clientes de forma que los desarrolladores que trabajan en el lado del cliente no tengan que conocer y preocuparse de los EJBs.

Otro beneficio del patrón de diseño es la posibilidad de realizar prototipado rápido de la aplicación, escribiendo en los métodos código de prueba, o incluso llamadas a JDBC. Incluso en una aplicación completa J2EE, el uso de este patrón puede ser útil para encapsular o usar almacenar en cachés los resultados de las llamadas a los EJBs.

En el siguiente ejemplo, se usa un business delegate para implementar una política de reintento de conexión en caso de problemas de red.

```
public class BeanDelegate
{
    private static final int NETWORK_RETRIES = 3;
    private BeanRemote bean;

    public void create( ) throws ApplicationError
    {
        // Here you get a bean instance.
        try {
            InitialContext ctx = new InitialContext( );
            BeanHome home = (BeanHome)
PortableRemoteObject.narrow (
                                ctx.lookup ("ejb/BeanExample"),
                                BeanHome.class);
        }
    }
}
```

```

        // Retry in case of network problems.
        for (int i=0; i<NETWORK_RETRIES; i++)
            try {
                bean = home.create( );
                break;
            } catch (RemoteException e) {
                if (i+1 < NETWORK_RETRIES) continue;
                throw new ApplicationError ("Network
problem "
                                + e.toString( ));
            }
        } catch (NamingException e) {
            throw new ApplicationError ("Error with bean");
        }
    }

    public void remove( ) throws ApplicationError
    {
        // Release the session bean here.

        // Retry in case of network problems
        for (int i=0; i<NETWORK_RETRIES; i++)
            try {
                bean.remove( );
                break;
            } catch (RemoteException e) {
                if (i+1 < NETWORK_RETRIES) continue;
                throw new ApplicationError ("Network problem "
                                + e.toString( ));
            }
    }

    public int doBusinessMethod (String param) throws ApplicationError
    {
        // Call a bean method here.
        for (int i=0; i<NETWORK_RETRIES; i++)
            try {
                return bean.doBusinessMethod (param);
            } catch (RemoteException e) {
                if (i+1 < NETWORK_RETRIES) continue;
                throw new ApplicationError ("Network problem "
                                + e.toString( ));
            }
    }
}

```

9.3 Despliegue

Por último veamos algunas recomendaciones sobre despliegue de una aplicación J2EE.

9.3.1 Crear un entorno de construcción

Es muy importante definir un entorno de construcción de las aplicaciones que desarrollemos. Entendemos por entorno de construcción un conjunto de comandos que simplifiquen la compilación, empaquetamiento y despliegue de las aplicaciones. Incluso si estamos usando un entorno integrado de desarrollo (IDE) para crear los ficheros EJB JARs y WARs, es conveniente desarrollar este tipo de entornos.

El principal beneficio es el de tener el control total sobre la estructura de los ficheros producidos, sin depender de las características particulares de una herramienta de desarrollo concreta.

Una herramienta de código abierto que cada vez se usa más es Ant. Está escrita en Java y es fácilmente personalizable para cualquier tarea, ya sea compilar aplicaciones, empaquetarlas, desplegarlas, etc. Es muy recomendable estudiar su uso.

9.3.2 Escribir código de prueba

Esta es una de las reglas principales que se suelen ignorar cuando se desarrollan aplicaciones con EJBs. La razón es muy simple: las aplicaciones J2EE son grandes y normalmente son más difíciles de depurar que las aplicaciones normales.

Por ello es muy útil usar un framework de casos de prueba como JUnit (<http://www.junit.org>) para probar las distintas capas de la aplicación de forma separada. Al probar las capas de forma independiente es posible aislar mejor los errores.

Es también una buena práctica el incluir código de diagnóstico en tiempo de ejecución en la versión final de la aplicación. Para que la aplicación funcione correctamente se deben desplegar muchos componentes y sistemas, y es útil generar un log en el que se registren los problemas que se detectan. De esta forma, por ejemplo, se podrían chequear fácilmente problemas ocasionados por la no conexión de los recursos necesarios para que la aplicación funcione.

Ejercicios

Ejercicio 1.1: Instalación de WebLogic 7.0

Suponemos que el servidor de aplicaciones está instalado en el directorio del usuario. Por ejemplo, en el directorio

```
/home/domingo/bea
```

Denominaremos este directorio `WL_INSTALL`. Vamos a necesitar que las variables de entorno tengan los valores que aparecen en la siguiente lista

1. Edita el fichero `$WL_INSTALL/weblogic700/server/bin/setEnv.sh` y modifica, en su primera línea de código, la variable `WL_HOME` con el valor donde está instalado el servidor de aplicaciones. Modifica también la variable `JAVA_HOME` con el directorio donde está instalada la máquina virtual Java (jdk 1.3.1).

2. Actualiza las variables de entorno ejecutando el script `setEnv`. Actualiza el valor de `CLASSPATH` para incluir el directorio actual.

```
% cd $WL_INSTALL/weblogic700/server/bin
% ./setEnv
% export CLASSPATH=%CLASSPATH:.
```

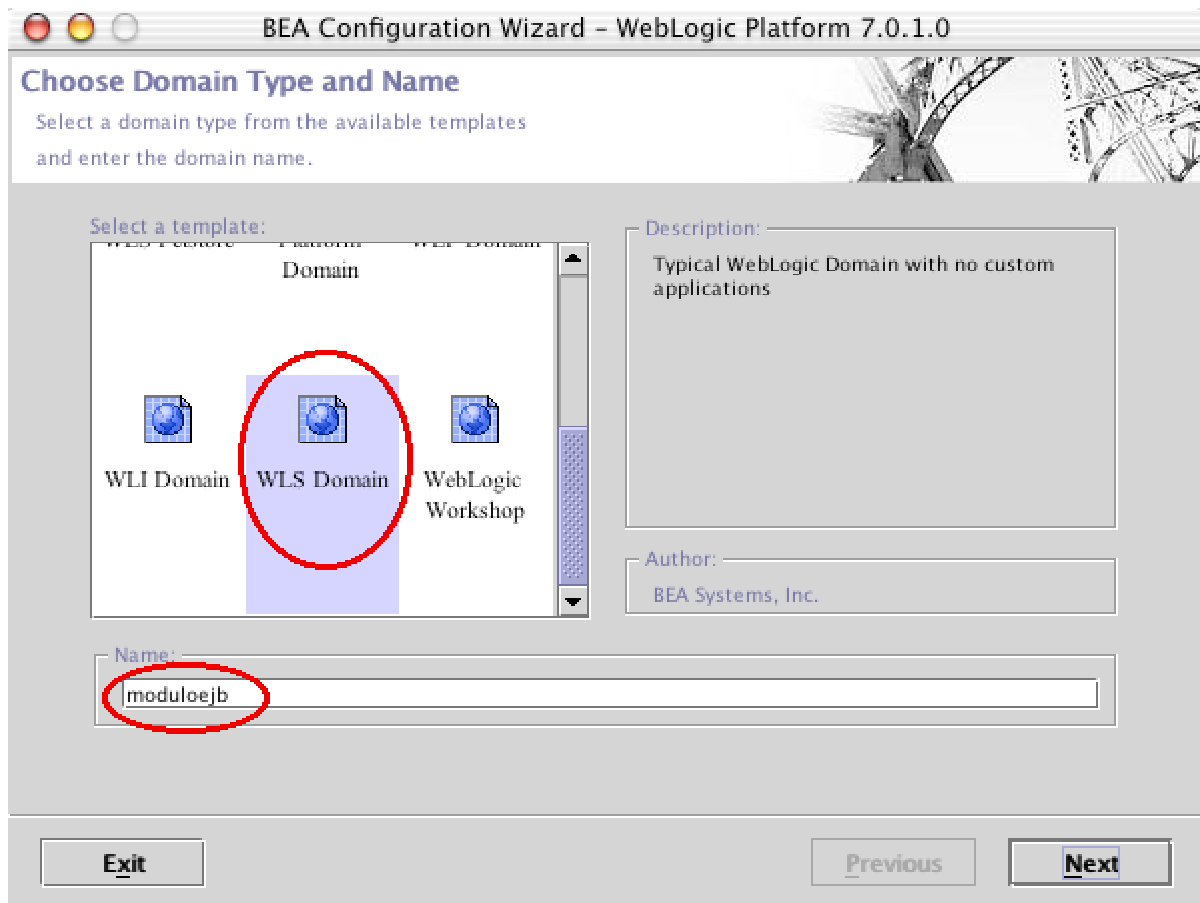
Crear el dominio vacío *moduloejb*

El servidor WebLogic 7.0 proporciona un Asistente de Configuración de Dominio que debe usarse para crear dominios vacíos. Ejecuta el asistente con los comandos

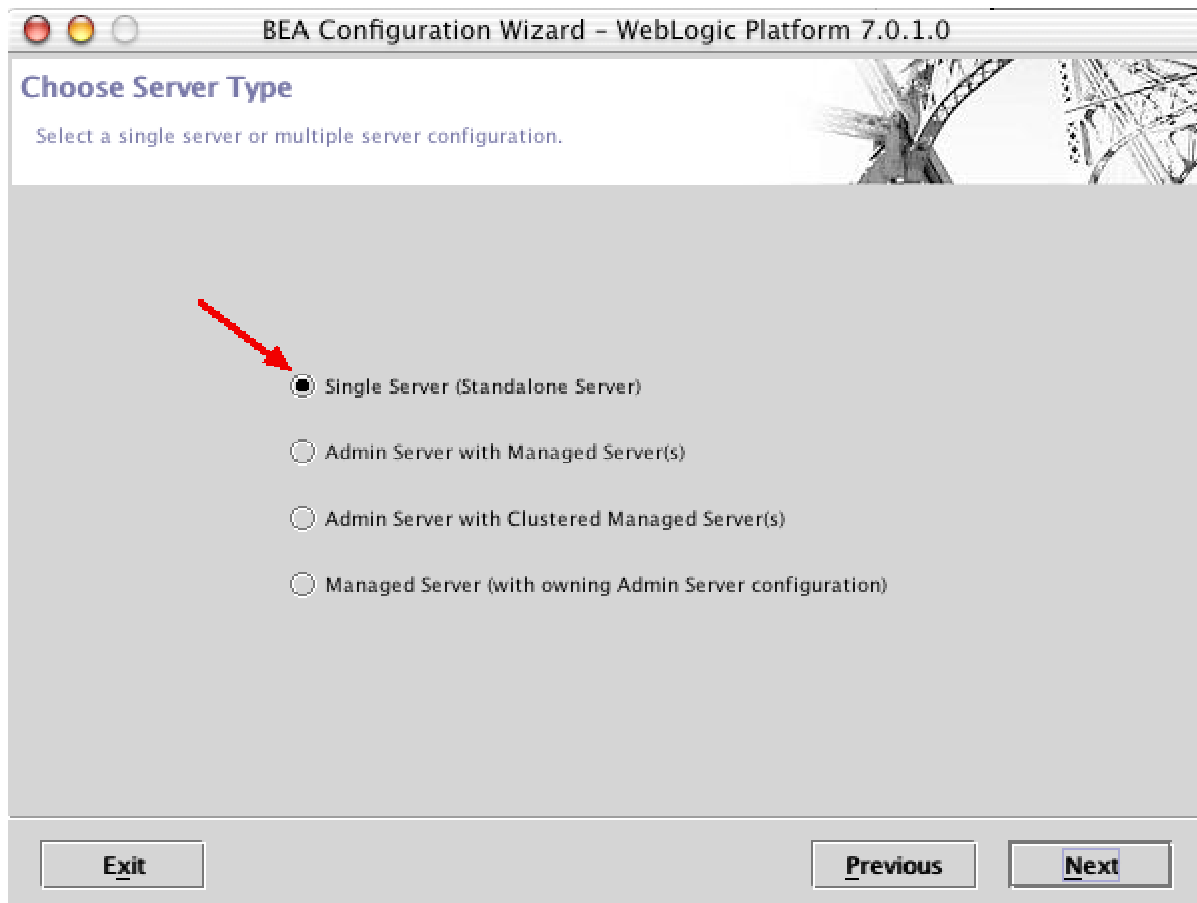
```
% cd $WL_INSTALL/weblogic70/common/bin
% ./dmwiz.sh
```

y crea un dominio vacío de un Standalone Server llamado *moduloejb* siguiendo los sencillos pasos ilustrados en las siguientes figuras:

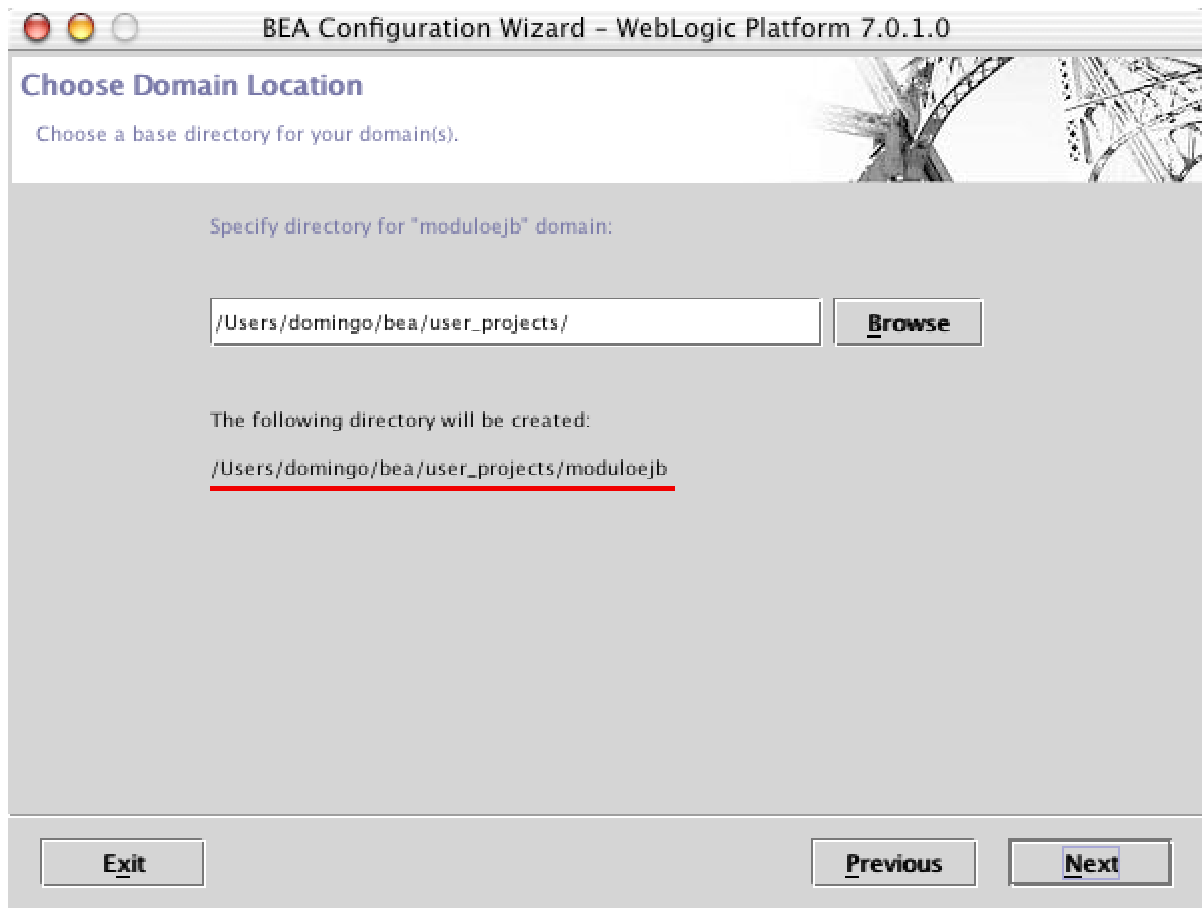
1. Define el tipo de dominio (WLS Domain sin aplicaciones) y el nombre (moduloejb):



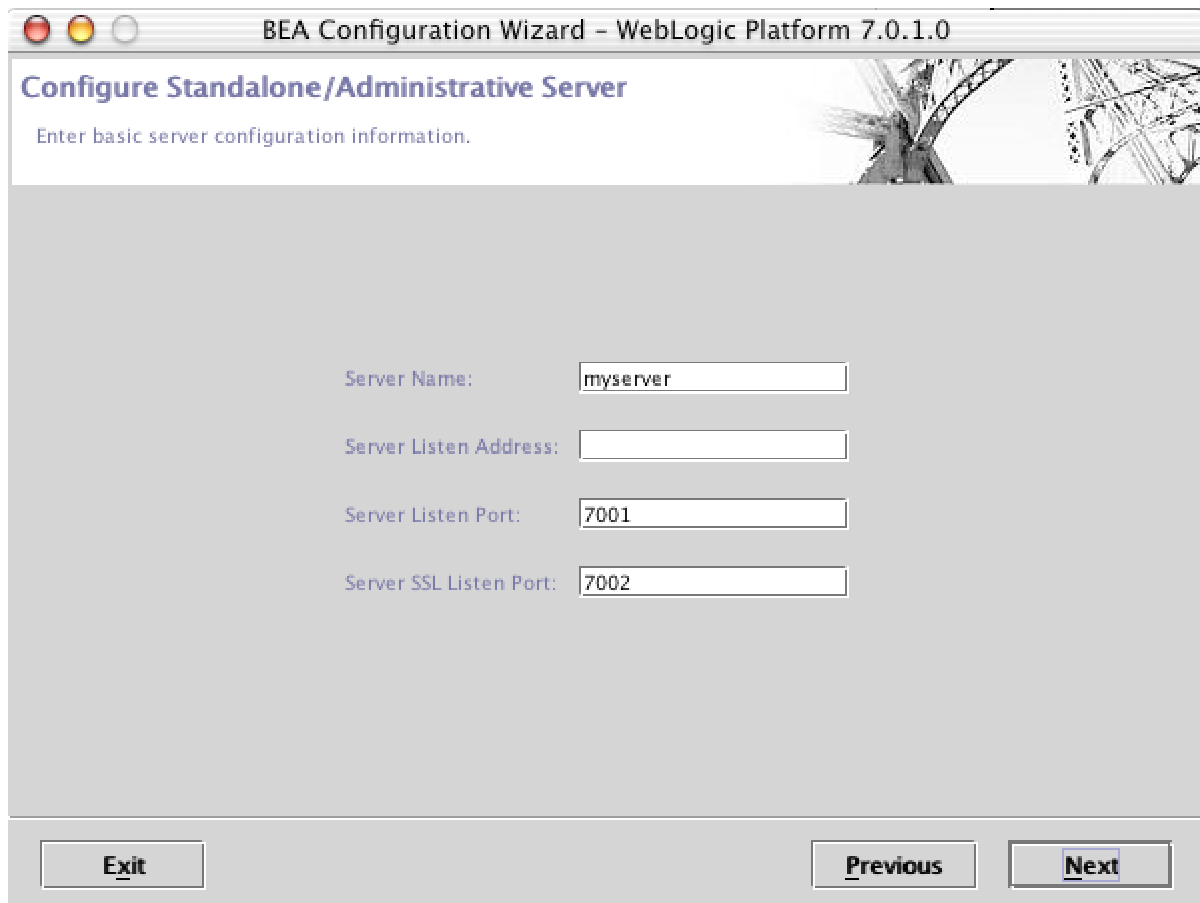
2. Define el funcionamiento del servidor. Por sencillez, vamos a preferir que sea standalone. Después, todo lo que probemos de esta forma va a poder funcionar en modo gestionado o incluso en clusters de servidores.



3. Define la localización del directorio donde se va a instalar el dominio. Acepta la opción por defecto (`$WL_INSTALL/user_projects/moduloejb`)



4. Define un nombre del servidor y unos puertos de escucha del mismo. Acepta la opción por defecto.



BEA Configuration Wizard - WebLogic Platform 7.0.1.0

Configure Standalone/Administrative Server
Enter basic server configuration information.

Server Name:

Server Listen Address:

Server Listen Port:

Server SSL Listen Port:

5. Define el login y password del administrador. Por ejemplo login: `system`, password: `weblogic`.

6. Termina aceptando las dos pantallas informativas que vienen a continuación. El dominio habrá sido creado en el directorio `/home/domingo/bea/user_projects/moduloejb` y en ese directorio estará instalado el servidor de aplicaciones preparado para ese dominio.

En su momento, cuando los ejercicios lo requieran, deberás crear los recursos: pool JDBC, TX DataSource y JMS.

Arrancar el servidor de aplicaciones

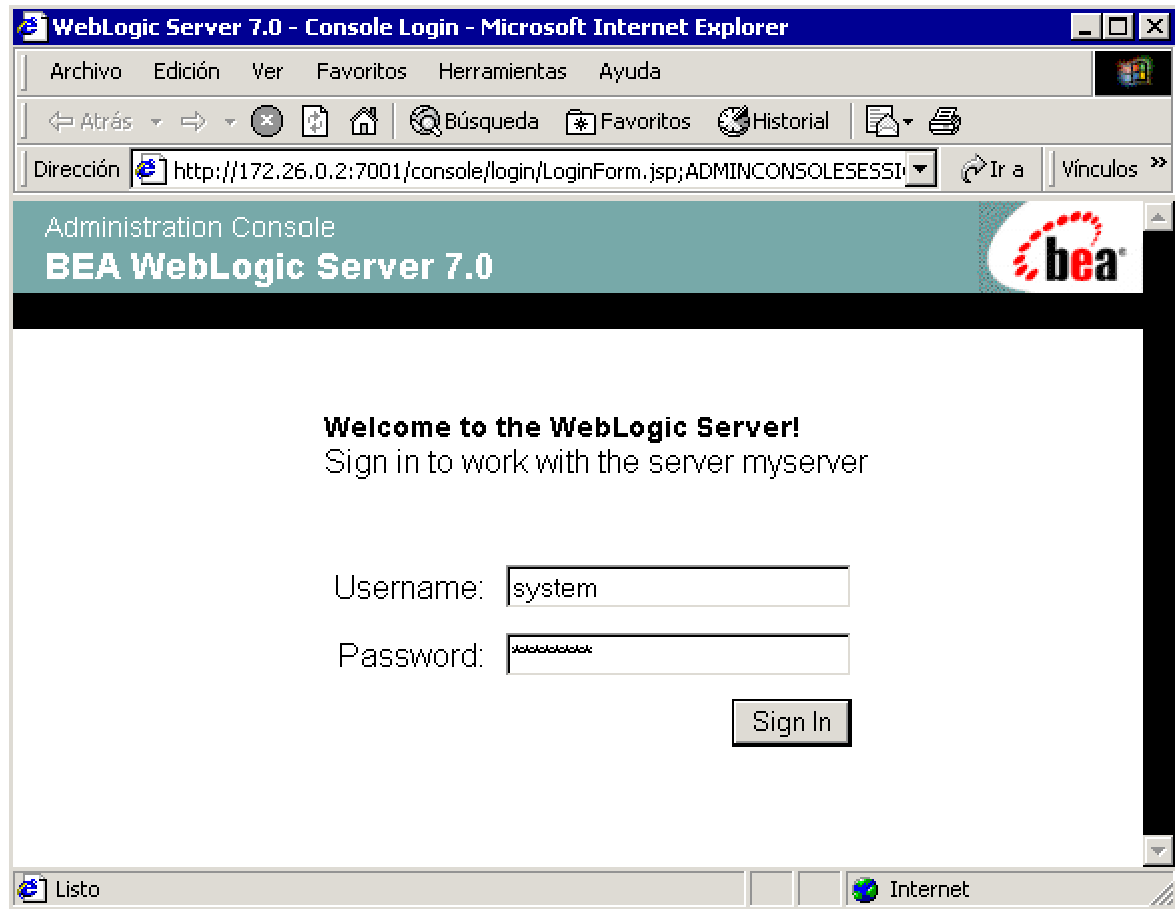
Una vez creado el dominio, ya puedes arrancar el servidor de aplicaciones, con los siguientes comandos en una ventana del sistema:

```
% cd $WL_INSTALL/user_projects/moduloejb
% ./startWebLogic.sh
```

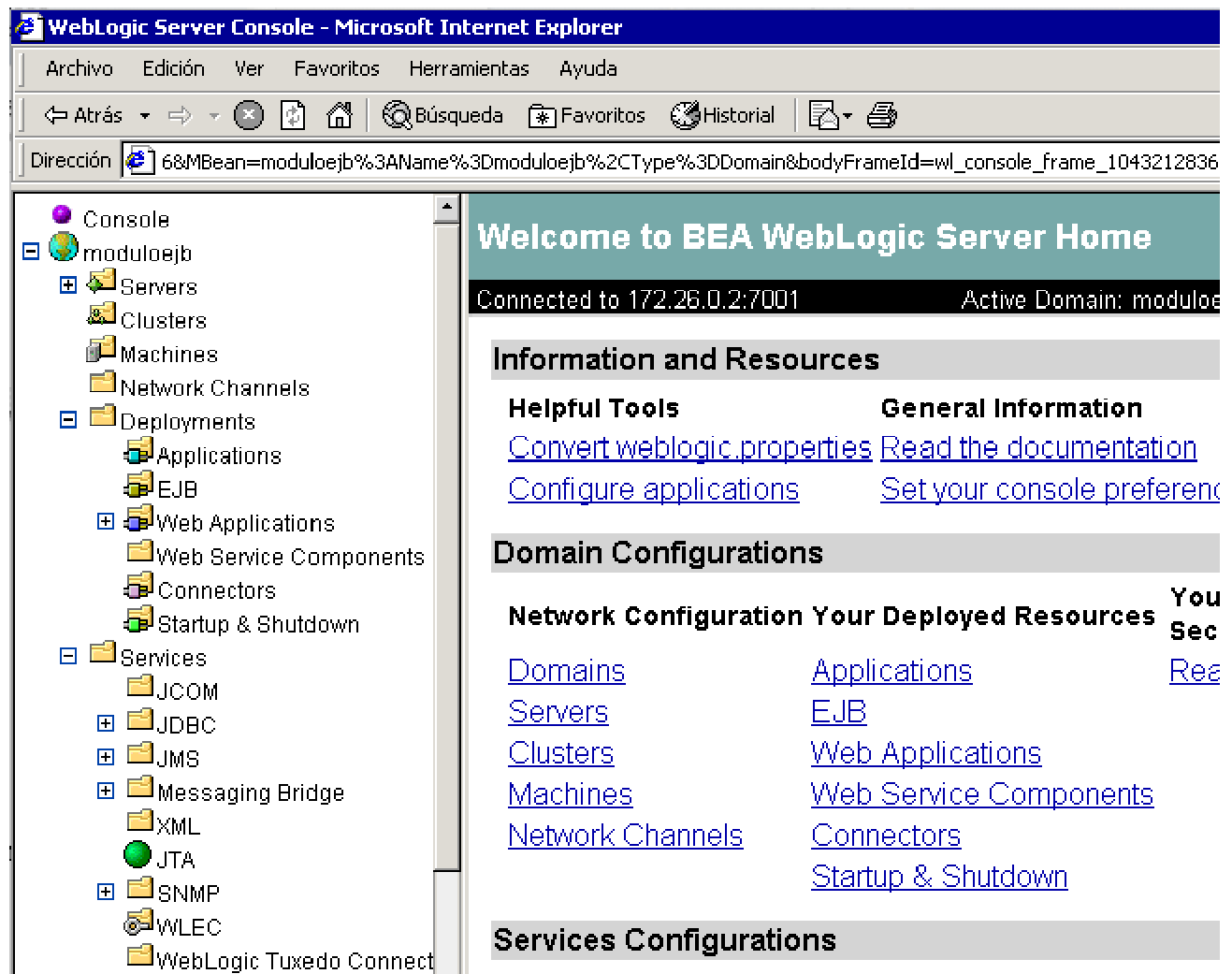
En la misma ventana del sistema te pedirá el login y contraseña de administrador. Teclea lo mismo que en la ventana de inicialización.

Comprobar que el servidor está en marcha

1. Desde un ordenador cualquiera lanza un navegador y conéctate a la URL `http://servidor:7001/console`, siendo servidor el ordenador en el que acabamos de arrancar el servidor de aplicaciones. Por ejemplo:



2. Introduce el login y la contraseña con la que hemos instalado el servidor de aplicaciones. Entrarás en la aplicación web con la que podemos administrar el servidor de aplicaciones.



Ejercicio 1.2: Compilar y desplegar el bean de sesión

El proceso de compilación, empaquetamiento, despliegue y prueba de un EJB de sesión y una aplicación cliente consta de los siguientes pasos:

1. Compilar los ficheros JAVA que forman el EJB y la aplicación cliente
2. Empaquetar los ficheros CLASS y el descriptor de despliegue en un fichero EJB JAR
3. Personalizar el fichero EJB JAR para el WebLogic con el WebLogic Builder
4. Desplegar el EJB JAR en el servidor de aplicaciones
5. Probar la aplicación cliente

Pasos previos:

- Chequear las variables de entorno
- Descargar el fichero ZIP con los ejemplos y descomprimirlo

Chequear las variables de entorno

Asegurate de que las variables de entorno tienen los valores correctos, tal y como vimos al comienzo del ejercicio anterior.

Obtener el código fuente con los ejemplos

El código fuente para los componentes se encuentra en la misma página donde se encuentran estos apuntes, en el fichero *ejercicio1.zip*.

1. Supongamos que el directorio de trabajo es

```
/home/domingo/ejb
```

Creamos una variable de entorno para definir el directorio de trabajo.

```
% export WORK=/home/domingo/ejb
```

2. Copiamos y descomprimos el directorio de ejemplos en el directorio de trabajo

```
% cp ejercicio1.zip $WORK  
% cd $WORK  
% unzip ejercicio1.zip
```

Los ficheros que se han descomprimido son:

```
HelloWorldSLBean/HelloWorldBean.java  
HelloWorldSLBean/HelloWorldRemoteHome.java  
HelloWorldSLBean/HelloWorldRemote.java  
HelloWorldClient.java  
ejb-jar.xml
```

En la próxima sesión explicaremos con más detalle el contenido de todos estos ficheros. Todos ellos implementan un bean de sesión sin estado con un único método, `hi()`, que devuelve un string con el típico saludo “Hello, world”. Vamos a ver en los pasos siguientes cómo compilar los ficheros, empaquetar el bean, desplegarlo en el servidor de aplicaciones y probarlo desde una aplicación cliente. Aunque se pueden utilizar herramientas avanzadas, como el `ant`, para compilar los beans, hemos preferido hacerlo de la manera más simple posible. Así sabemos en todo momento qué es lo que estamos haciendo.

1. Para compilar los ficheros vamos al directorio de ejemplo y llamamos a `javac` (ya hemos actualizado el `CLASSPATH` en un paso anterior):

```
% cd $WORK/HelloWorld  
% javac HelloWorldSLBean/*.java
```

2. Compilar y ejecutar la aplicación cliente. Si miramos el código fuente de la aplicación, lo único que hace es conectarse con el bean (la llamada clase

Home del bean), crear una instancia del bean y pedir el método `hi()` de esa instancia. Al no estar el bean desplegado en el servidor de aplicaciones, la aplicación cliente generará un error:

```
% javac HelloWorldClient.java
% java HelloWorldClient
javax.naming.NameNotFoundException: Unable to resolve 'HelloWorldEJB'
Resolved: ''
Unresolved:'HelloWorldEJB' ; remaining name 'HelloWorldEJB'
    at
weblogic.rmi.internal.BasicOutboundRequest.sendReceive(BasicOutboundRe
quest.java:109)
    at
weblogic.rmi.cluster.ReplicaAwareRemoteRef.invoke(ReplicaAwareRemoteRe
f.java:262)
    at
weblogic.rmi.cluster.ReplicaAwareRemoteRef.invoke(ReplicaAwareRemoteRe
f.java:229)
    at
weblogic.jndi.internal.ServerNamingNode_WLStub.lookup(Unknown Source)
    at
weblogic.jndi.internal.WLContextImpl.lookup(WLContextImpl.java:337)
    at
weblogic.jndi.internal.WLContextImpl.lookup(WLContextImpl.java:332)
    at javax.naming.InitialContext.lookup(InitialContext.java:350)
    at HelloWorldClient.main(HelloWorldClient.java:15)
```

Empaquetar el bean

Debemos construir un fichero JAR, que llamaremos `helloworld-ejb.jar`, con el bean compilado. Para ello:

```
% mkdir META-INF
% mv ejb-jar.xml META-INF/
% jar cvf helloworld-ejb.jar META-INF/* HelloWorldSLBean/*.class
manifest agregado
agregando: META-INF/ejb-jar.xml(entrada = 663) (salida=
330) (desinflado 50%)
agregando: HelloWorldSLBean/HelloWorld.class(entrada = 240) (salida=
182) (desinflado 24%)
agregando: HelloWorldSLBean/HelloWorldBean.class(entrada = 710)
(salida= 363) (desinflado 48%)
agregando: HelloWorldSLBean/HelloWorldHome.class(entrada = 288)
(salida= 199) (desinflado 30%)
```

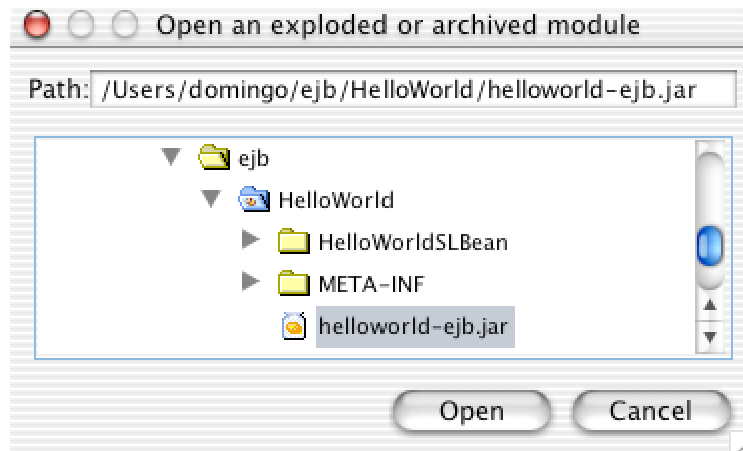
Desplegar el bean de sesión usando el WebLogic Builder

El WebLogic Builder es una aplicación Java que se distribuye junto con el servidor WebLogic y que tiene como principal utilidad la configuración de ficheros JAR, WAR y EAR para el posterior despliegue en el servidor. Vamos a ver cómo funciona, desplegando el fichero `helloworld-ejb.jar`

1. Lanzar la aplicación WebLogic Builder:

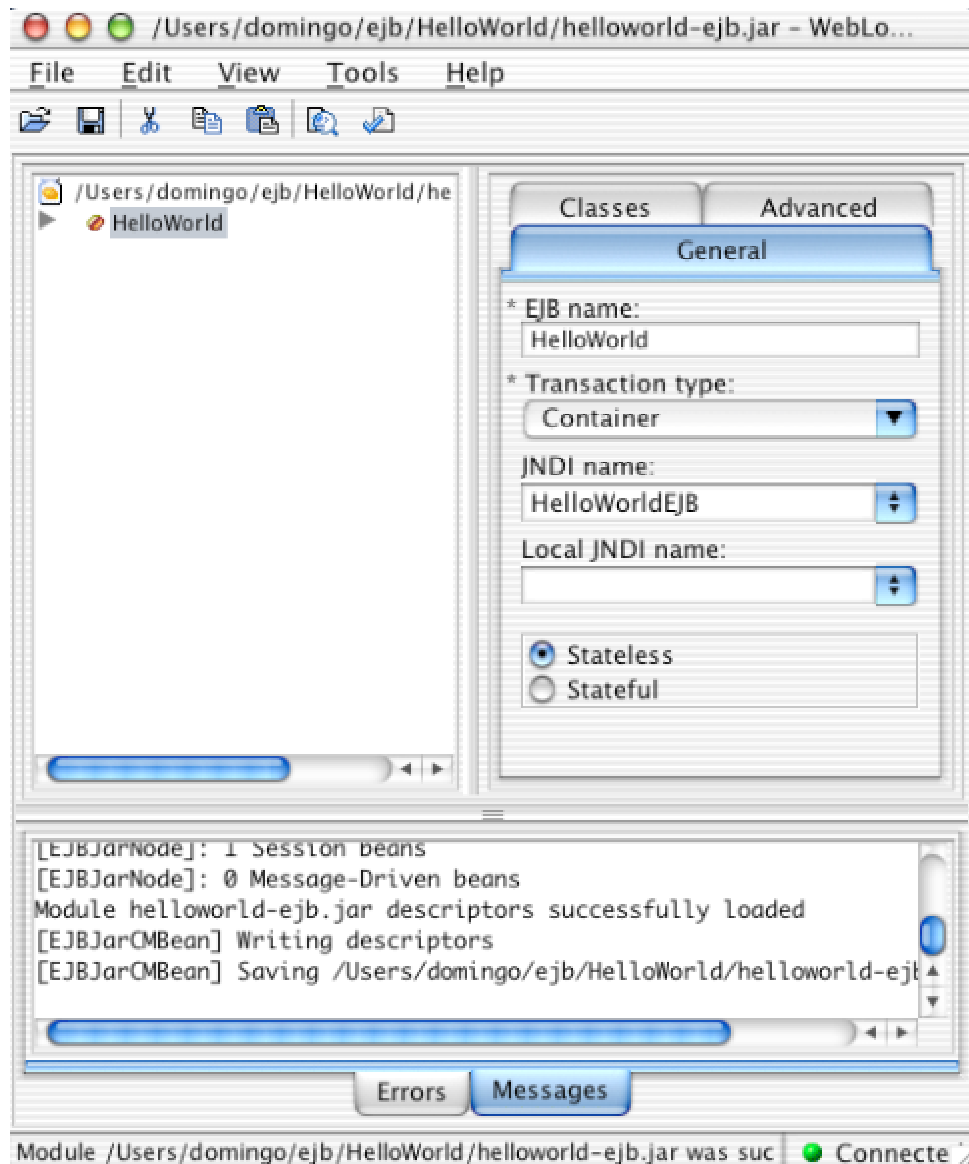

```
% cd $WL_INSTALL/weblogic700/server/bin  
% ./startWLBuilder.sh
```

2. Abrir el fichero helloworld-ejb.jar con el programa WebLogic Builder



Aceptar cuando pregunta si debe crear el fichero de descripciones `weblogic-ejb-jar.xml` (IMPORTANTE: el fichero `weblogic-ejb-jar.xml` es necesario para que el bean se despliegue correctamente en el servidor de aplicaciones. Salvar el paquete con la opción **File > Save** para que el fichero `helloworld-ejb.jar` se actualize con este fichero).

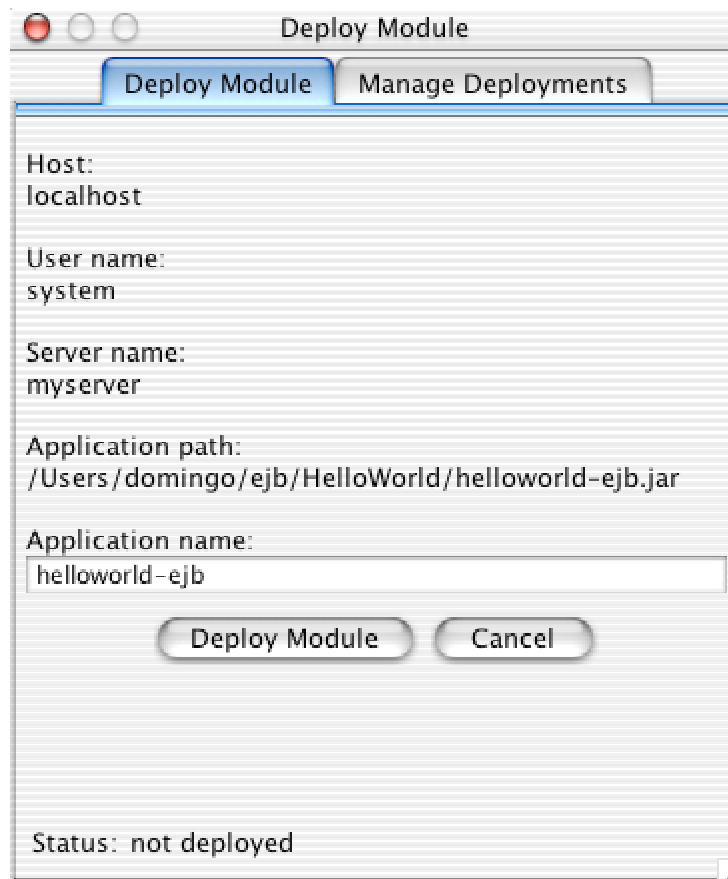
3. Cambiar el nombre JNDI del bean. Por defecto, el nombre JNDI es el mismo que el nombre del bean (`HelloWorld`). Vamos a cambiarlo, poniendo como nombre `HelloWorldEJB`. Este nombre es al que se hace referencia en la aplicación cliente. Para cambiar el nombre pincha en el bean `HelloWorld` que aparece en la parte superior derecha de la ventana. La ventana principal de la aplicación cambiará y te permitirá modificar ciertas propiedades del bean, entre ellas el nombre JNDI. Pon como nombre `HelloWorldEJB` y a continuación salva el bean.



3. Para desplegar el bean en el servidor, primero hay que conectarse a él con la aplicación. Selecciona la opción **Tools > Connect to server**. Aparece una ventana de diálogo en la que puedes dar la dirección de red donde se encuentra el servidor, el nombre del servidor, puerto, login y password. Completa el login y password de administrador.



4. Selecciona la opción **Tools > Deploy Module**.



Pulsa el botón Deploy Module. Si todo funciona correctamente el bean se desplegará en el servidor.

5. Comprueba que el bean se ha desplegado en el servidor de aplicaciones. Para ello conéctate con el navegador a la dirección *http://localhost:7001/console* y selecciona el apartado EJB. Allí debes encontrar el bean que acabas de desplegar.

Probar la aplicación cliente

Ahora la aplicación cliente debe funcionar correctamente. ¡Pruébala!:

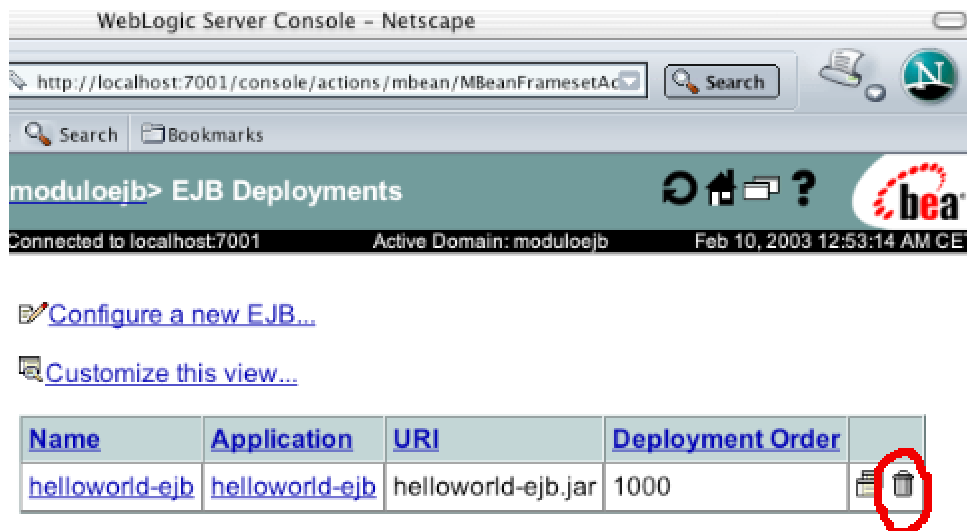
```
cd $WORK/ejb/HelloWorld
% java HelloWorldClient
Voy a llamar al bean
Hola; soy Domingo
He llamado al bean
```

Desplegar el bean usando la consola de administración

Vamos a ver otra forma de desplegar el bean; ahora usando la consola de administración.

1. Vamos a volver al instante en que el fichero EJB JAR no estaba desplegado en el servidor de aplicaciones. Para ello elimina el bean desplegado usando la

consola de administración. Debes pinchar en el cubo de basura que hay a la derecha del nombre del bean, en la pantalla **moduloejb > EJB Deployments**



Acepta en la pantalla de confirmación. Con ello el EJB se habrá eliminado del servidor de aplicaciones. Si ejecutas la aplicación cliente, verás que aparece el mismo error que la primera vez que lo hicimos.

2. En la consola de administración escoger la opción **EJB** y **Configure a new EJB**. Entrarás en la pantalla **Locate Application or Component to Configure**. En esta pantalla puedes subir un EJB JAR al servidor y después seleccionarlo para desplegarlo. Como el fichero EJB JAR ya está en el propio servidor (la máquina localhost) sólo debes seleccionarlo.

Step 2. Select the .ear, .war, .jar, or .rar file you would like to configure and deploy. Note that you may also configure an 'exploded' application or component by simply selecting its root directory. Click on the [select] link to the left of the desired directory or file to choose it and proceed to the next step.

Listing of [localhost/Users/domingo/ejb/HelloWorld](#)

[\[Up to parent directory\]](#)
[\[select\]](#) [HelloWorldSLBean](#)
[\[select\]](#) [META-INF](#)
[\[select\]](#) [helloworld-ejb.jar](#)

Selecciona ahora el servidor en el que quieres desplegar el bean (myserver), acepta el nombre por defecto de la aplicación y pulsa en **Configure and Deploy**:

Step 3. You have chosen to configure

`/Users/domingo/ejb/HelloWorld/helloworld-ejb.jar`

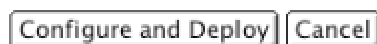
Select the Servers and/or Clusters on which you would like to deploy this application initially. (You can reconfigure deployment targets later if you wish).



Step 4. Enter a name for this application.

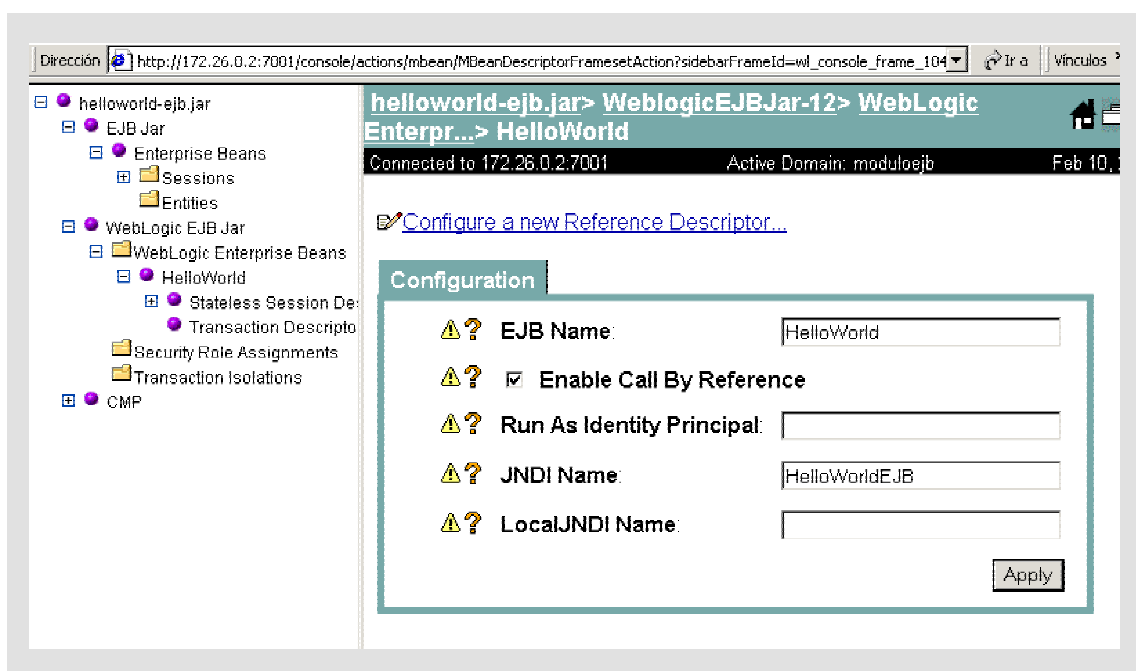
`helloworld-ejb`

Step 5. Press 'Configure and Deploy' to configure and deploy the application, or 'Cancel' to leave the Domain unchanged.



3. Prueba la aplicación cliente. Te seguirá dando error. ¿Porqué?

4. Para cambiar el nombre **JNDI** desde la consola de administración, selecciona la opción de Edit Deployment Descriptor. Se abrirá una nueva ventana del navegador en la que podrás modificar los distintos aspectos de configuración del EJB. Despliega el árbol WebLogic EJB Jar, WebLogic Enterprise Beans y selecciona el objeto HelloWorld. Aparece una pantalla en la que podrás modificar el **JNDI Name**. Escribe HelloWorldEJB y Apply Changes:



Comenzará a parpadear el icono con la admiración. Eso significa que debes reiniciar el servidor. Pero primero debes grabar la nueva configuración del EJB: pulsa en el árbol helloworld-ejb.jar y cambiará la pantalla de la derecha. Pulsa la opción “Persist”, con lo que se estará grabando el fichero helloworld-ejb.jar modificado. Para no tener que personalizar más este fichero, puedes descomprimirlo con el comando:

```
% jar xvf helloworld-ejb.jar
```

Si ahora miras en el directorio META-INF verás que está el fichero weblogic-ejb-jar.xml. Este fichero es el que ha creado el WebLogic Builder y ha modificado la consola de administración. Si lo editas, verás que el elemento XML jndi-name tiene el valor HelloWorldEJB.

Reinicia el servidor y prueba la aplicación cliente. Ahora sí que debe funcionar.

Ejercicio 3: Modificar el código fuente del bean

Modifica el código fuente del bean para que en lugar de devolver el mensaje “Hola, soy Domingo”, devuelva tu nombre. Para ello:

- Edita el fichero HelloWorldBean.java
- Compila todos los ficheros del bean
- Empaqueta el bean, ahora en el directorio META-INF ya está el fichero de personalización para el servidor WebLogic
- Arranca el WebLogic Builder
- Carga el EJB JAR
- Conecta el servidor y despliega el EJB JAR en el servidor
- Prueba la aplicación cliente

Por último, modifica la aplicación cliente, para que en lugar de conectarse al bean de tu propio servidor de aplicaciones se conecte al de un compañero.

Ejercicio 2.1: Compilación, despliegue y prueba de una aplicación web que usa un bean de sesión sin estado

Pasos previos

Antes de realizar el ejercicio, vamos a asegurarnos de que todo está configurado correctamente:

- El dominio `moduloejb` debe estar creado. Compruébalo mirando si existe un directorio con ese nombre en el directorio `bea/user_projects`.
- Actualiza la variable `CLASSPATH`:

```
% export PATH=$PATH:/home/j2ee/bea/weblogic700/server/bin
% . setWLSEnv.sh
% export CLASSPATH=$CLASSPATH:.
```

- Descarga el fichero ZIP con el código fuente del ejercicio (`ejercicio2.zip`) en el directorio de trabajo `/home/j2ee/ejb`

Compilación y creación del fichero JAR con el bean

Al descomprimir el fichero ZIP debes tener la siguiente estructura de ficheros:

```
HelloWorldWeb/jsp/CommonFunctions.jsp
HelloWorldWeb/jsp/HelloWorld.jsp

HelloWorldWeb/src/HelloWorldServlet.java
HelloWorldWeb/src/HelloWorldSLBean/HelloWorld.java
HelloWorldWeb/src/HelloWorldSLBean/HelloWorldBean.java
HelloWorldWeb/src/HelloWorldSLBean/HelloWorldHome.java

HelloWorldWeb/xml/application.xml
HelloWorldWeb/xml/ejb-jar.xml
HelloWorldWeb/xml/web.xml
HelloWorldWeb/xml/weblogic-application.xml
HelloWorldWeb/xml/weblogic-ejb-jar.xml
HelloWorldWeb/xml/weblogic.xml
```

1. Compila los ficheros del bean:

```
% cd HelloWorldWeb/src
% javac HelloWorldSLBean/*.java
```

2. Crea el fichero EJB JAR

```
% cd ..
% mkdir jar-ejb
% cd jar-ejb
```



```
% mkdir META-INF
% cp ../xml/ejb-jar.xml META-INF
% cp ../xml/weblogic-ejb-jar.xml META-INF
% mkdir HelloWorldSLBean
% cp ../src/HelloWorldSLBean/*.class HelloWorldSLBean
% jar cvf helloworld-ejb.jar *
manifest agregado
agregando: META-INF/ejb-jar.xml
agregando: META-INF/weblogic-ejb-jar.xml
agregando: HelloWorldSLBean/HelloWorld.class
agregando: HelloWorldSLBean/HelloWorldBean.class
agregando: HelloWorldSLBean/HelloWorldHome.class
```

3. Compila el servlet

```
% cd ../src
% javac HelloWorldServlet.java
```

4. Crea el fichero WAR JAR con la aplicación web

```
% cd ..
% mkdir jar-war
% cd jar-war
% mkdir WEB-INF
% cp ../xml/web.xml WEB-INF
% cp ../xml/weblogic.xml WEB-INF
% cp ../jsp/* .
% mkdir WEB-INF/classes
% mkdir WEB-INF/classes/HelloWorldSLBean
% cp ../src/HelloWorldSLBean/HelloWorld.class WEB-INF/classes/HelloWorldSLBean/
% cp ../src/HelloWorldSLBean/HelloWorldHome.class WEB-INF/classes/HelloWorldSLBean/
% cp ../src/HelloWorldServlet.class WEB-INF/classes
% jar cvf helloworld.war *
manifest agregado
agregando: CommonFunctions.jsp
agregando: HelloWorld.jsp
agregando: WEB-INF/
agregando: WEB-INF/classes/
agregando: WEB-INF/classes/HelloWorldServlet.class
agregando: WEB-INF/classes/HelloWorldSLBean/
agregando: WEB-INF/classes/HelloWorldSLBean/HelloWorld.class
agregando: WEB-INF/classes/HelloWorldSLBean/HelloWorldHome.class
agregando: WEB-INF/web.xml
agregando: WEB-INF/weblogic.xml
```

5. Por último, vamos a crear el fichero JAR con la aplicación (fichero EAR):

```
% cd ..
% mkdir jar-ear
% cd jar-ear
% mkdir META-INF
% cp ../xml/application.xml META-INF
% cp ../xml/weblogic-application.xml META-INF
% cp ../jar-ejb/helloworld-ejb.jar .
```

```
% cp ../jar-war/helloworld.war .
% jar cvf helloworld.ear *
manifest agregado
agregando: META-INF/
agregando: META-INF/application.xml
agregando: META-INF/weblogic-application.xml
agregando: helloworld-ejb.jar
agregando: helloworld.war
```

Desplegar la aplicación

1. Arranca el servidor de aplicaciones instalado en el ejercicio 1 de la sesión anterior
2. Usando la consola de administración, elimina el EJB HelloWorld existente (si es que hubiera alguno).
3. Despliega la aplicación. **Importante:** hazlo sin subir el fichero al servidor de aplicaciones, porque ya lo tienes allí:
 - Selecciona la opción *Applications > Configure new a new Application*.
 - Muévete por el árbol de directorios y selecciona el fichero helloworld.ear

Step 2. Select the .ear, .war, .jar, or .rar file you would like to configure and also configure an 'exploded' application or component by simply selecting the [select] link to the left of the desired directory or file to choose it and

Listing of [localhost/Users/domingo/ejb/HelloWorldWeb/jar-ear](#)

[\[Up to parent directory\]](#)
[\[select\]](#) META-INF
[\[select\]](#) helloworld-ejb.jar
[\[select\]](#) helloworld.ear
[\[select\]](#) helloworld.war

Probar la aplicación

1. Prueba el servlet y la página JSP

```
http://localhost:7001/HelloWorld/HolaMundo
http://localhost:7001/HelloWorld/HolaMundo.jsp
```

2. Compila el cliente y Pruébalo.

```
% cd src
% javac HelloWorldClient.java
% java HelloWorldClient
```

Ejercicio 2.2: cambiar algunos descriptores del despliegue

URLs de la aplicación

1. Las URL públicas de la aplicación se encuentran definidas en los elementos `url-pattern` del fichero de despliegue de la aplicación web (`web.xml`). El directorio raíz de la aplicación se encuentra definido en el elemento `context-root` del fichero de descripción del despliegue `application.xml`.

2. Cambia el nombre de estos elementos para que el acceso al servlet y a la página JSP se a las siguientes direcciones

```
http://localhost:7001/hw/hw
http://localhost:7001/hw/hw.jsp
```

3. Reconstruye los ficheros `helloworld.war` y `helloworld.ear` y redespiega la aplicación. Pruébala para asegurarte que funciona correctamente.

Referencias a beans en la aplicación web

1. En la aplicación web se accede al bean `HelloWorld` de forma indirecta, usando la dirección `java:comp/env/ejb/HelloWorldEJB` del espacio local de nombres JNDI ENC. Esto permite definir de forma declarativa el enlace entre la aplicación y el bean hasta el momento del despliegue y cambiar el mismo sin tener que recompilar el EJB.

- La definición de qué EJB concreto se liga a la dirección `ejb/HelloWorldEJB` se realiza en el fichero `weblogic.xml`, asociando un nombre JNDI concreto a esa dirección.
- La definición del nombre JNDI del bean `HelloWorld` se realiza en el fichero `weblogic-ejb-jar.xml`.

2. Cambia el nombre JNDI del bean y llámalo `HW`, ahora usando la consola de administración (seleccionando el bean *HelloWorld* y pulsando en la opción *Edit EJB Descriptor*). En el árbol con los elementos XML de la ventana del navegador que se acaba de abrir despliega la opción *WebLogic EJB JAR* y luego *Weblogic Enterprise Beans*. Pincha en el bean y cambia su nombre JNDI. Pincha en el padre de todo el árbol de elementos XML (con el nombre *helloworld-ejb.jar*) y pulsa la opción *Persist*. De esta forma se habrá guardado en el fichero de descripción del despliegue el nuevo nombre JNDI del bean. Puedes cerrar la ventana del navegador.

3. Redespiega la aplicación. En la ventana principal de la consola de administración pulsa la opción *Applications* y selecciona la aplicación *HelloWorld*. Pulsa la opción *Deploy Application* para volver a desplegar la aplicación.

Configuration | **Deploy** | Notes

Deployment Status by Target:

Component	Component Type	Target	Target Type	Deployed	
helloworld.war	Web Application	myserver	Server	true	<input type="button" value="Undeploy"/> <input type="button" value="Redeploy"/>
helloworld-ejb.jar	EJB	myserver	Server	true	<input type="button" value="Undeploy"/> <input type="button" value="Redeploy"/>

Undeploy the entire application

Deploy or redeploy the entire application

4. Prueba ahora cualquiera de las URLs y verás que aparece un error.
5. Modifica el código fuente de la aplicación cliente para que se conecte al bean con el nuevo nombre JNDI. Recompila y comprueba que ahora sí funciona.
6. Modifica el nombre JNDI de la referencia `ejb/HelloWorldEJB` usando también la consola de administración. Graba el cambio con la opción *Persist* y redespiega la aplicación. Prueba ahora las URLs.

Ejercicio 3: compilación y prueba de un bean de sesión con estado

1. Compila, despliega y prueba el bean cart que hay en el fichero ejercicio3.zip
2. Incluye en el bean el método applyDiscount, y modifica la aplicación cliente para poder probarlo
3. Implementa un servlet que use el bean, y crea una aplicación web. Desplégala y pruébala en el servidor de aplicaciones

Ejercicio 4.1: Despliegue y prueba del bean SavingsAccountEJB

En este primer ejercicio probaremos y desplegaremos el bean SavingsAccountEJB. Se trata de un bean de entidad con persistencia gestionada por el bean.

El ejercicio consta de los siguientes pasos:

1. Creación del dominio weblogic "ejemplos", basado en el servidor de ejemplos.
2. Creación de la tabla SAVINGSACCOUNT de la base de datos Demo. Esta base de datos está manejada por el servidor PointBase, un servidor de bases de datos escrito en Java que viene incluido en la distribución de Weblogic.
3. Despliegue del bean.
4. Compilación y prueba de la aplicación cliente.

Creación del dominio `ejemplos`

Para crear el dominio `ejemplos` debemos ejecutar el comando

```
% WL_HOME/boa/weblogic700/common/bin/dmwwiz.sh
```

donde `WL_HOME` es el directorio donde se encuentra instalado el servidor de aplicaciones. Aparecerán las ventanas de ayuda del asistente de creación de dominios que nos irá guiando para seleccionar las opciones adecuadas. Debemos seleccionar la plantilla *WLS Examples* y asignarle como nombre `ejemplos`. Escogeremos después la opción *Single Server (Standalone Server)* y aceptaremos el directorio por defecto (`WL_HOME\user_projects\ejemplos`). Aceptaremos el nombre de servidor por defecto (`myserver`) y los puertos de escucha. Dejaremos en blanco la dirección de escucha del servidor. Como usuario de administración podemos escribir `system` y `weblogic` como contraseña.

Una vez instalado el dominio de ejemplos, lanzaremos el servidor de aplicaciones

```
% WL_HOME/user_projects/ejemplos/startExamplesServer.sh
```

y nos conectaremos con el navegador a la dirección `http://localhost:7001/examplesWebApp/index.jsp`. Allí se encuentra la página principal en la que se pueden probar distintos tipos de ejemplos proporcionados por el servidor de aplicaciones.

Creación de la tabla `savingsaccount`

Para usar EJB de entidad es necesario crear tablas en alguna base de datos. El servidor de aplicaciones weblogic proporciona el gestor de bases de datos PointBase, escrito en Java y completamente integrado en el servidor de ejemplos. Cuando lanzamos el servidor de aplicaciones, automáticamente se arranca el servidor de bases de datos.

Podemos comprobar cuál es el nombre JNDI de la fuente de datos que usa el servidor de ejemplos. Para ello, entramos en la consola de administración (<http://localhost:7001/console>) y seleccionamos la opción *Tx Data Sources*, bajo el apartado *Services Configurations*. Aparecerán distintas fuentes de datos. La que vamos a usar en los ejemplos es la fuente de datos *examples-dataSource-demoPool* que tiene ese mismo nombre JNDI. Si seleccionamos el Connection Pool asociado a esa fuente de datos (demoPool) veremos que las características de la base de datos son las siguientes:

- URL: `jdbc:pointbase:server://localhost/demo`
- Driver class name: `com.pointbase.jdbc.jdbcUniversalDriver`
- Usuario y password: `examples, examples`

Vamos a crear la tabla que necesitamos en la base de datos `demo`:

1. Descargamos el fichero `ejercicio4.zip`, y lo descomprimos en el directorio de trabajo `$HOME/ejb`

2. Se habrá creado el directorio `$HOME/ejb/savingsaccount`. En ese directorio se encuentran los subdirectorios `java`, `xml` y `sql`. Dentro del directorio `sql`, se encuentra el script `savingsaccount.sql` con el que vamos a crear la base de datos.

3. Copiar ese script al directorio

```
$WL_HOME/weblogic700/samples/server/eval/pointbase/tools
```

```
% cp savingsaccount/sql/savingsaccount.sql
$WL_HOME/weblogic700/samples/server/eval/pointbase/tools
```

4. Vamos a llamar a una aplicación cliente que nos permite ejecutar comandos SQL contra la base de datos. No se encuentra directamente instalada en el directorio `tools`, pero podemos crearla fácilmente. Nos movemos al directorio `tools` y copiamos el fichero `startPointBaseConsole.sh` con el nombre `startPointBaseCommander.sh`

```
% cd $WL_HOME/weblogic700/samples/server/eval/pointbase/tools% cp
startPointBaseConsole.sh startPointBaseCommander.sh
```

y modificamos su última línea de código. Donde pone

```
"${JAVA_HOME}/bin/java" -classpath "${CLASSPATH}"
com.pointbase.tools.toolsConsole
```

lo cambiamos por

```
"${JAVA_HOME}/bin/java" -classpath "${CLASSPATH}"
com.pointbase.tools.toolsCommander
```

5. Ejecutamos la aplicación cliente del gestor de bases de datos con la instrucción

```
./startPointBaseCommander.sh
```

Introducimos los siguientes valores

```
Do you wish to create a "New/Overwrite" Database? [default: N]:
Select product to connect with: Embedded (E), or Server (S)? [default:
E]: S Please enter the driver to use: [default:
[com.pointbase.jdbc.jdbcUniversalDriver]: Please enter the
database URL to use: [default:
[jdbc:pointbase:server://localhost/sample]:
jdbc:pointbase:server://localhost/demo Username: [default: PBPUBLIC]:
examples Password: [default: PBPUBLIC]: examples PointBase Commander
4.2ECF build 183 SERVER Interactive SQL command language. Mac OS
X/10.1.5(C) Copyright 1998 - 2002 PointBase(R), Inc. All rights
reserved SQL>
```

6. Una vez en la consola SQL, ejecutaremos el comando

```
SQL > run "savingsaccount.sql";
```

(Recuerda el punto y coma!!). Ahora ya se habrá creado la tabla necesaria para el probar el bean de entidad.

Despliegue del bean

1. Vuelve al directorio `ejb`, donde se encuentran todos los ficheros necesarios para el bean y comprueba que se encuentra el fichero EJB JAR `savingsaccount-ejb.jar`.

2. Accede desde el navegador a la consola de administración y realiza el despliegue del bean `savingsaccount-ejb.jar`. Debes seleccionar la opción **EJB > Configure a new EJB** y pasar directamente al paso 2 para moverte por el árbol de directorios hasta el directorio `$HOME/ejb/savingsaccount` y seleccionar el fichero `savingsaccount-ejb.jar`. Selecciona el servidor `myserver` y pulsa **Configure and Deploy**.

Compilación y prueba de la aplicación cliente

1. Añade el directorio `WL_HOME/weblogic700/server/bin` al path:

```
% export PATH=$PATH:WL_HOME/weblogic700/server/bin
```

2. Actualiza las variables del entorno

```
% . setWLSEnv.sh% export CLASSPATH=$CLASSPATH:.
```


3. Vuelve al directorio donde se encuentra los fuentes y compila todos los ficheros

```
% cd $HOME/ejb/savingsaccount/src % javac *.java
```

4. Ejecuta la aplicación cliente

```
% java SavingsAccountClient
```

Vuelve a ejecutarla otra vez; esta segunda vez tiene que arrojar una excepción de tipo `InsufficientBalanceException`, porque se intenta hacer un cargo de una cantidad superior a la que tiene una cuenta de balance.

Ejercicio 4.2: Modificación del bean

1. Modifica el bean y el programa SQL para incluir las siguientes características

- Incluir el campo `string state`, de 2 caracteres, que puede tener el valor "OK" o "NU" (No Usada), así como los métodos de negocio `setUnused()` y `setUsed()`, que permiten cambiar a NU y OK respectivamente el valor de ese campo.
- Añadir el método Boolean `getUsed()` que nos devuelve true o false según la cuenta está usada o no.
- Incluir el método home `clearAccounts()`, que permita poner a 0.00 el balance todas las cuentas de aquellos clientes que tengan el campo `state="NU"`.

2. Escribe una pequeña aplicación cliente que pruebe la modificaciones del bean.

Ejercicio 5.1: Despliegue y prueba del bean CustomerEJB

En este primer ejercicio probaremos y desplegaremos el bean CustomerEJB. Se trata de un bean de entidad con persistencia gestionada por el contenedor.

Para el ejercicio usaremos el dominio weblogic "ejemplos" creado en el ejercicio anterior. El ejercicio consta de los siguientes pasos:

1. Creación de la tabla CUSTOMER de la base de datos Demo.
2. Compilación y despliegue del bean.
3. Compilación y prueba de la aplicación cliente.

Creación de la tabla customer

Vamos a crear la tabla que necesitamos en la base de datos demo:

1. Descargamos el fichero ejercicio5.zip, y lo descomprimos en el directorio de trabajo \$HOME/ejb

2. Se habrá creado el directorio \$HOME/ejb/customer. En ese directorio se encuentran los subdirectorios java, xml y sql. Dentro del directorio sql, se encuentra el script customer.sql con el que vamos a crear la base de datos.

3. Copiar ese script al directorio

```
$WL_HOME/weblogic700/samples/server/eval/pointbase/tools
```

```
% cp customer/sql/customer.sql
$WL_HOME/weblogic700/samples/server/eval/pointbase/tools
```

4. Ejecutamos la aplicación cliente del gestor de bases de datos con la instrucción

```
% cd $WL_HOME/weblogic700/samples/server/eval/pointbase/tools%
./startPointBaseCommander.sh
```

Introducimos los siguientes valores

```
Do you wish to create a "New/Overwrite" Database? [default: N]:
Select product to connect with: Embedded (E), or Server (S)? [default:
E]: S Please enter the driver to use: [default:
[com.pointbase.jdbc.jdbcUniversalDriver]: Please enter the
database URL to use: [default:
[jdbc:pointbase:server://localhost/sample]:
jdbc:pointbase:server://localhost/demo Username: [default: PBPUBLIC]:
examples Password: [default: PBPUBLIC]: examples PointBase Commander
4.2ECF build 183 SERVER Interactive SQL command language. Mac OS
X/10.1.5(C) Copyright 1998 - 2002 PointBase(R), Inc. All rights
reserved SQL>
```

6. Una vez en la consola SQL, ejecutaremos el comando

```
SQL > run "customer.sql";
```

(Recuerda el punto y coma!!). Ahora ya se habrá creado la tabla necesaria para el probar el bean de entidad.

Despliegue del bean

1. Vuelve al directorio `ejb`, donde se encuentran todos los ficheros necesarios para el bean y comprueba que se encuentra el fichero EJB JAR `customer-ejb.jar`.

2. Accede desde el navegador a la consola de administración y realiza el despliegue del bean `savingsaccount-ejb.jar`. Debes seleccionar la opción *EJB > Configure a new EJB* y pasar directamente al paso 2 para moverte por el árbol de directorios hasta el directorio `$HOME/ejb/customer` y seleccionar el fichero `customer-ejb.jar`. Selecciona el servidor `myserver` y pulsa *Configure and Deploy*.

Compilación y prueba de la aplicación cliente

1. Añade el directorio `WL_HOME/weblogic700/server/bin` al path:

```
% export PATH=$PATH:WL_HOME/weblogic700/server/bin
```

2. Actualiza las variables del entorno

```
% . setWLSEnv.sh% export CLASSPATH=$CLASSPATH:.
```

3. Vuelve al directorio donde se encuentra los fuentes y compila todos los ficheros

```
% cd $HOME/customer/src% javac *.java
```

4. Ejecuta la aplicación cliente para añadir registros de customers

```
% java AddCustomers
```

5. Ejecuta la aplicación cliente para buscar customers

```
% java FindCustomer
```

Ejercicio 5.2: Modificación del bean

1. Modifica la aplicación cliente `FindCustomer` para que actualice a `False` el valor del campo `hasGoodCredit`, con el método de negocio `setHasGoodCredit(False)`.

2. Modifica los ficheros del bean para añadir el campo `Integer edad`, así como los métodos de acceso `setEdad(Integer)` y `getEdad()`. Debes modificar también el fichero de creación de la base de datos. Recompila y reconstruye el fichero `customer-ejb.jar`.

3. Modifica las aplicaciones clientes `AddCustomers` y `FindCustomer` para que actualicen e impriman respectivamente la edad del `Customer`.

4. Añade un nuevo método de búsqueda

```
public Collection findByGoodCredit() throws javax.ejb.FinderException;
```

en la clase `CustomerHomeRemote.java`. Añade la cláusula EJB-QL correspondiente a esta búsqueda en el fichero `ejb-jar.xml`:

```
<query>
  <query-method>
    <method-name>findByGoodCredit</method-name>
    <method-params/>
  </query-method>
  <ejb-ql>
    SELECT OBJECT(c) FROM Customer c
    WHERE c.hasGoodCredit = TRUE  </ejb-ql>
</query>
```

5. Escribe una aplicación cliente que llame a `findByGoodCredit()` y muestre todos los customers que tienen ese campo a true.

Ejercicio 7: Descargar y probar el ejemplo7

Lanza el servidor de ejemplos creado en un ejercicio anterior. Descarga el fichero `ejemplo7.zip` y descomprímelo en el directorio de trabajo. Vamos a trabajar con los EJB definidos en ese fichero:

- Customer
- Address
- CreditCard
- Phone

Vamos a desplegarlo y a probarlo. Esta vez, en lugar de una aplicación cliente tenemos unas páginas JSP. Antes de desplegarlo hay que crear las tablas necesarias en la base de datos (el script se encuentra en el directorio `sql`).

Recordamos la forma de ejecutar el script `ejemplo7.sql`:

```
% cd sql
%
$WL_HOME/weblogic700/samples/server/eval/pointbase/tools/startPointBaseCommander.sh
Do you wish to create a "New/Overwrite" Database? [default: N]:
Select product to connect with: Embedded (E), or Server (S)?
[default: E]: S
Please enter the driver to use:
[default: [com.pointbase.jdbc.jdbcUniversalDriver]:
Please enter the database URL to use:
[default: [jdbc:pointbase:server://localhost/sample]:
    jdbc:pointbase:server://localhost/demo
Username: [default: PBPUBLIC]: examples
Password: [default: PBPUBLIC]: examples
PointBase Commander 4.2ECF build 183 SERVER
Interactive SQL command language. Mac OS X/10.1.5(C) Copyright
1998 - 2002 PointBase(R),
Inc. All rights reserved
SQL> run "ejemplo7.sql";
```

Una vez que hemos creado las tablas, ya podemos proceder al despliegue y a la prueba. Despliega el fichero `ejemplo7.ear` **como una aplicación**. Verás que tiene dos componentes, un fichero EJB JAR y un fichero WAR.

Para probar las páginas JSP debes abrir las URL:

- `http://localhost:7001/ejemplo7/client1.jsp`: se crea un ejb Customer y un ejb CreditCard y se relacionan
- `http://localhost:7001/ejemplo7/client2.jsp`: se crea un ejb Address y se relaciona con el Customer

- <http://localhost:7001/ejemplo7/client3.jsp>: se crean ejbs Phone y se hacen distintas operaciones

Ejercicio 7.1

1. Modifica la página JSP `client1.jsp` para:

- crear dos ejbs Customer con claves primarias 71 y 72
- comprobar si existen los ejb y no se creen unos nuevos si ya existen
- crear dos ejbs CreditCard distintos que se asignan a cada uno
- terminar asignando uno de los ejb CreditCard al otro Customer. Comprueba si se ha desconectado del primero.

Reconstruye el fichero ear para que incluya las modificaciones, vuelve a desplegarlo y comprueba si funciona bien.

2. Modifica la página JSP `client2.jsp` para:

- crear una dirección para el Customer 72
- intercambiar las direcciones del Customer 72 y del Customer 71

3. Modifica el ejb Customer para incluir el siguiente método de ayuda

```
public void removePhoneNumberWithDelete(byte typeToRemove)
    throws javax.ejb.RemoveException {

    Collection phoneNumbers = this.getPhoneNumbers();
    Iterator iterator = phoneNumbers.iterator();

    while (iterator.hasNext()) {
        PhoneLocal phone = (PhoneLocal) iterator.next();
        if (phone.getType() == typeToRemove) {
            phoneNumbers.remove(phone);
            phone.remove();
        }
    }
}
```

Modifica el fichero `client3.jsp` para que se llame a este método y comprueba que realmente se han borrado los ejb Phone correspondientes.

Ejercicio 8: seguridad y control de acceso

En este ejercicio vamos a modificar el EJB `savingsaccount` para incorporar un control de acceso. En resumen, vamos a definir dos roles, Administrador y ReadOnly y vamos a probar la aplicación cliente

Descargar y probar el ejb `savingsaccount`

Lanza el servidor de ejemplos creado en un ejercicio anterior. Comprueba si se encuentra desplegado el EJB `savingsaccount-ejb.jar`. En el caso en que se encontrara, bórralo. Descarga el fichero `ejemplo8.zip` y descomprímelo en el directorio de trabajo. Vamos a trabajar con ese EJB. Debes desplegarlo y probar la aplicación cliente que se encuentra en el directorio `src`. Para ello hay que crear la tabla en la base de datos (el script se encuentra en el directorio `sql`), desplegar el bean, compilar la aplicación cliente y probarla.

Recordamos la forma de ejecutar el script `savingsaccount.sql`:

```
% cd sql
%
$WL_HOME/weblogic700/samples/server/eval/pointbase/tools/startPointBaseCommander.sh
Do you wish to create a "New/Overwrite" Database? [default: N]:
Select product to connect with: Embedded (E), or Server (S)?
[default: E]: S
Please enter the driver to use:
[default: [com.pointbase.jdbc.jdbcUniversalDriver]:
Please enter the database URL to use:
[default: [jdbc:pointbase:server://localhost/sample]:
    jdbc:pointbase:server://localhost/demo
Username: [default: PBPUBLIC]: examples
Password: [default: PBPUBLIC]: examples
PointBase Commander 4.2ECF build 183 SERVER
Interactive SQL command language. Mac OS X/10.1.5(C) Copyright
1998 - 2002 PointBase(R),
Inc. All rights reserved
SQL> run "savingsaccount.sql";
```

Crear de nuevos usuarios en el servidor weblogic

1. Abre la consola de administración. Pincha la opción *Realms* y después pincha en *myreal*. Aparecerá una pantalla de entrada de datos. Selecciona la pestaña superior *Contents* y, en la nueva pantalla que aparece, pincha el enlace *Manage Users within this Realm*. En la nueva pantalla, pincha en *Configure a new user....* Allí tendrás que definir el nombre y contraseña del nuevo usuario. Pon `admin` como usuario y `weblogic` como contraseña. Si escoges otra contraseña, debe tener 8 o más caracteres. Pulsa *Apply*.

2. Pincha en la pestaña *Groups*, selecciona el grupo `Administrators` y pulsa la flecha. De esta forma añadimos el usuario `admin` al grupo `Administrators`.
3. Crea ahora el usuario `operador`. Asígnalo al grupo `Operators`.

Modificación del bean `savingsaccount`

1. Desempaqueta el EJB JAR

```
% jar xvf savingsaccount-ejb.jar
```

2. Modifica el fichero `META-INF/ejb-jar.xml` para incorporar el control de acceso al EJB. Para ello borra las siguientes líneas

```
<method-permission>
  <unchecked />
  <method>
    <ejb-name>SavingsAccountEJB</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

y sustitúyelas por estas otras

```
<security-role>
  <role-name>Administrador</role-name>
</security-role>
<security-role>
  <role-name>ReadOnly</role-name>
</security-role>

<method-permission>
  <role-name>Administrador</role-name>
  <method>
    <ejb-name>SavingsAccountEJB</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>

<method-permission>
  <role-name>ReadOnly</role-name>
  <method>
    <ejb-name>SavingsAccountEJB</ejb-name>
    <method-name>getFirstName</method-name>
  </method>
  <method>
    <ejb-name>SavingsAccountEJB</ejb-name>
    <method-name>getLastName</method-name>
  </method>
  <method>
    <ejb-name>SavingsAccountEJB</ejb-name>
    <method-name>findByPrimaryKey</method-name>
  </method>
  <method>
    <ejb-name>SavingsAccountEJB</ejb-name>
    <method-name>getBalance</method-name>
```



```

</method>
<method>
  <ejb-name>SavingsAccountEJB</ejb-name>
  <method-name>findByLastName</method-name>
</method>
<method>
  <ejb-name>SavingsAccountEJB</ejb-name>
  <method-name>findInRange</method-name>
</method>
</method-permission>

```

3. Por último, debes hacer corresponder los roles con los usuarios antes creados. Para ello debes añadir las siguientes líneas al final del fichero `META-INF/weblogic-ejb-jar.xml`, justo antes del elemento `</weblogic-ejb-jar>`

```

<security-role-assignment>
  <role-name>Administrador</role-name>
  <principal-name>admin</principal-name>
</security-role-assignment>
<security-role-assignment>
  <role-name>ReadOnly</role-name>
  <principal-name>operador</principal-name>
</security-role-assignment>

```

4. Construye de nuevo el bean

```
% jar cvf savingsaccount-ejb.jar *.class META-INF/*
```

y vuelve a desplegarlo usando la consola del servidor de aplicaciones.

5. Una vez desplegado el bean, prueba la aplicación cliente. Deberá arrojar un error producido porque no se tiene suficientes privilegios para trabajar con el bean.

```

Intentando encontrar la cuenta con identificador: i0
Caught an exception.
java.rmi.AccessException: Security Violation: User: '<anonymous>'
has
insufficient permission to access EJB: type=<ejb>,
application=savingsaccount-ejb,
module=savingsaccount-ejb, ejb=SavingsAccountEJB,
method=findByPrimaryKey,
methodInterface=Home, signature={java.lang.String}.

```

6. Para autenticarte, introduce el siguiente código en la aplicación cliente:

```

public static Context getInitialContext()
throws javax.naming.NamingException {
  Properties p = new Properties();
  p.put(Context.INITIAL_CONTEXT_FACTORY,
"weblogic.jndi.WLInitialContextFactory");
  p.put(Context.SECURITY_PRINCIPAL, "admin");
  p.put(Context.SECURITY_CREDENTIALS, "weblogic");
}

```

```
p.put(Context.PROVIDER_URL, "t3://localhost:7001");  
return new javax.naming.InitialContext(p);  
}  
}
```

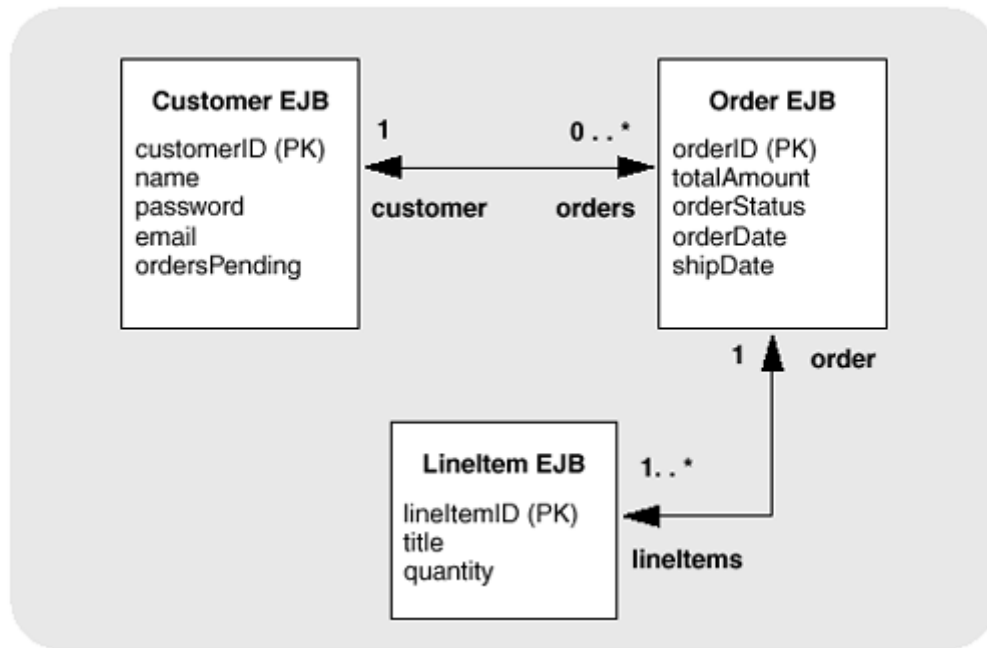
7. Prueba a autenticarte como usuario `operador`. Verás que se obtiene el siguiente error en el control de acceso.

```
Intentando encontrar la cuenta con identificador: i0  
...  
Intentando encontrar la cuenta con identificador: i16  
Intentando encontrar la cuenta con identificador: i17  
Intentando encontrar la cuenta con identificador: i18  
Intentando encontrar la cuenta con identificador: i19  
Caught an exception.  
java.rmi.AccessException: Security Violation: User: 'operador'  
has insufficient permission to access EJB: type=,  
application=savingsaccount-ejb, module=savingsaccount-ejb,  
ejb=SavingsAccountEJB, method=getId, methodInterface=Remote,  
signature={}
```

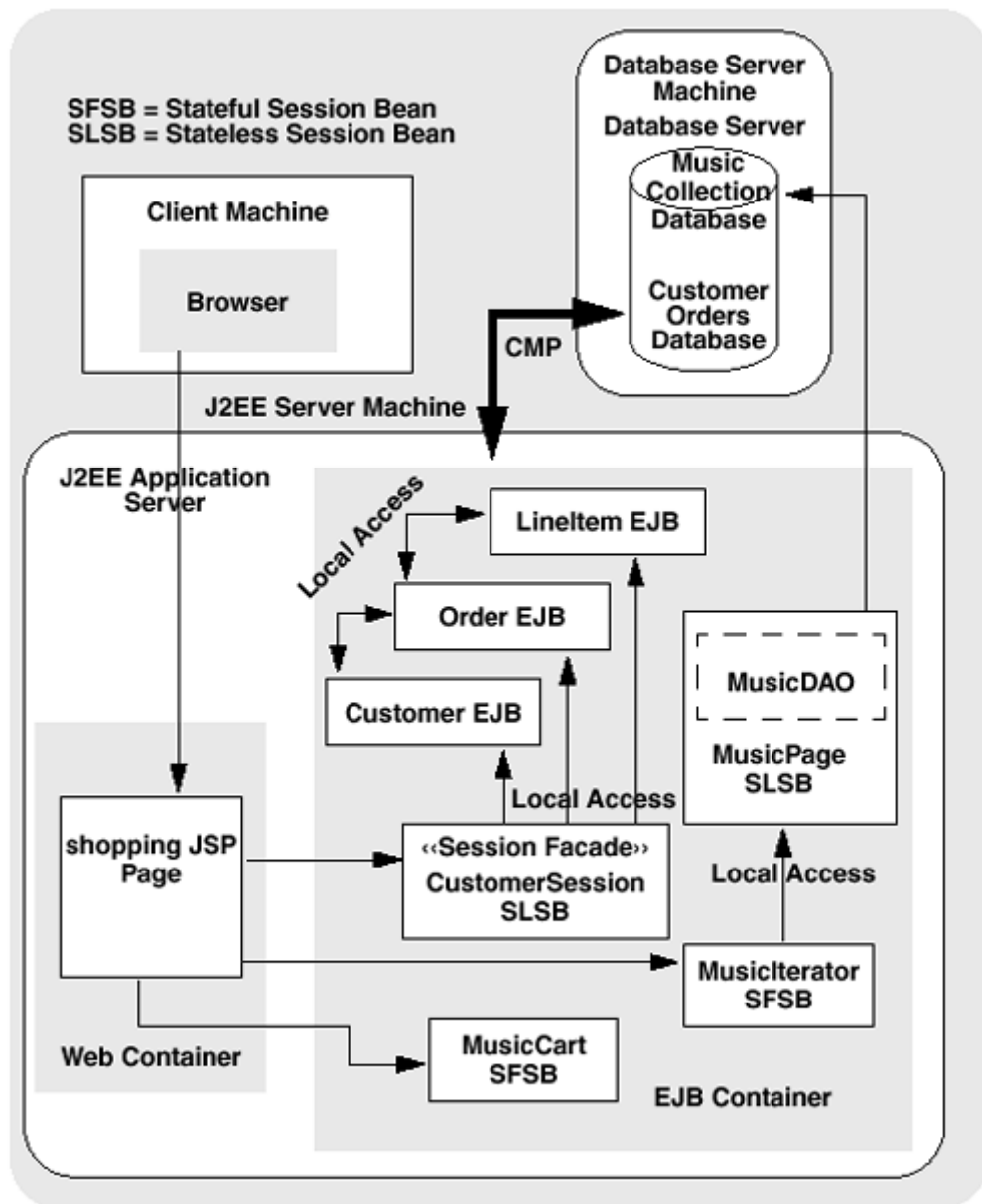
8. Modifica por último el descriptor de despliegue para que el rol `ReadOnly` pueda acceder al método `getId`. Despliega otra vez el bean y vuelve a probar la aplicación cliente. Comprueba qué sucede.

Ejercicio 9: Un ejemplo completo con EJBs

Vamos a probar una aplicación completa que usa EJBs. Se trata de una tienda on-line de discos. El esquema de las relaciones entre los beans de entidad de la aplicación es el siguiente:



La arquitectura de la aplicación es la siguiente:



1. Descarga el fichero `ejercicio9.zip` y descomprímelo en el directorio de trabajo. Verás que tiene un fichero EAR en el que está empaquetada toda la aplicación. En los distintos directorios se encuentran los ficheros JAVA y JSP para implementar la aplicación. En el directorio `sql` se encuentra el script `sql` para crear las bases de datos.
2. Lanza el servidor de ejemplos y carga el script `sql`.
3. Despliega la aplicación en el servidor de ejemplos.
4. Prueba la aplicación, accediendo a la página `http://localhost:7001/shopping/login`. Verás que, cuando intentas introducir un nuevo cliente aparece un error. Este error es debido a que se ha incluido en la aplicación código Java propietario de una base de datos (CloudScape) que no es la que usamos.

Ejercicio (optativo, y sólo si te hacen tilín los EJBs): modifica la aplicación para que no use código Java propietario de la base de datos y pueda funcionar correctamente en nuestro entorno habitual de trabajo.

Bibliografía

- Richard Monson-Haefel: *Enterprise JavaBeans, 3rd Edition*. Ed. O'Reilly, September 2001
- Gail Anderson, Paul Anderson: *Enterprise JavaBeans™ Component Architecture: Designing and Coding Enterprise Applications*. Ed. Prentice Hall, March 2002
- The O'Reilly Java Authors: *Java™ Enterprise Best Practices*. Ed. O'Reilly, December 2002
- Vlada Matena, Beth Stearns: *Applying Enterprise JavaBeans™: Component-Based Development for the J2EE™ Platform*, Ed. Addison Wesley, December 2000
- The J2EE Tutorial for J2EE 1.3 (http://java.sun.com/j2ee/tutorial/1_3-fcs/index.html)
- Enterprise JavaBeans Specification 2.0 (<http://java.sun.com/products/ejb/docs.html#specs>)