4

UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

TECNOLOGIA J2EE APLICADA A UN SISTEMA DE COMERCIO ELECTRONICO

T E S I S

QUE PARA OBTENER EL TITULO DE:

LICENCIADA EN CIENCIAS DE LA COMPUTACION

P R E S E N T A:

GRISELDA GONZALEZ RODRIGUEZ

FACULTAD DE CIENCIAS
UNAM

DIRECTOR DE TESIS:
DRA. HANNA OKTABA

2002

FACULTAD DE CIENCIAS
SECCION ESCOLAR

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

M. EN C. ELENA DE OTEYZA DE OTEYZA
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunico a usted que hemos revisado el trabajo escrito:
TECNOLOGIA J2EE APLICADA A UN SISTEMA DE COMERCIO ELECTRONICO

realizado por GONZALEZ RODRIGUEZ GRISELDA

con número de cuenta 09561150-5 **, quién cubrió los créditos de la carrera de**

LIC. EN CIENCIAS DE LA COMPUTACION
Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis
Propietario
DRA. HANNA OKTABA
Propietario
M. EN C. MARIA GUADALUPE E. IBARGÜENGOITIA GONZALEZ
Propietario
M. EN I. MARIA DE LUZ GASCA SOTO
Suplente
M. EN C. GUSTAVO A. MARQUEZ FLORES
Suplente
ACT. ALEJANDRO TALAVERA ROSALES

Consejo Departamental de MATEMATICAS

DRA. AMPARO LOPEZ GAONA

FACULTAD DE CIENCIAS
CONSEJO DEPARTAMENTAL
DE
MATEMATICAS

A mis padres, familia,
profesores, compañeros y amigos.

Índice General

Introducción 1

1 Comercio Electrónico **4**

1.1 Historia. 5

1.2 Tipos de Comercio Electrónico. 6

1.3 Tipos de Pago para el Comercio Electrónico. 7

1.4 Tecnologías del Comercio Electrónico. 8

1.5 Seguridad en la Internet comercial. 9

1.6 Características de Sistema de Ventas Café. 10

2 Java 2 Enterprise Edition **11**

2.1 Orígenes de J2EE. 12

2.2 Plataforma J2EE. 13

2.2.1 Tecnologías J2EE. 13

2.3 Escenarios de Aplicación J2EE. 23

2.3.1 Escenario de Aplicación Multicapa. 25

2.3.2 Escenario Cliente *Stand-Alone*. 26

2.3.3 Escenario de Aplicación Centrado en el Web. 28

2.3.4 Escenario Negocio a Negocio. 29
2.4 Transacciones. 30
2.4.1 Propiedades de las Transacciones. 30
2.4.2 Transacciones en la plataforma J2EE. 31
2.4.3 Escenarios. 31
2.5 Seguridad. 33
2.6 Haciendo Modulos. 34

3 Enterprise JavaBeans **36**
3.1 Tecnología Enterprise JavaBeans. 36
3.1.1 Contenedor EJB. 37
3.1.2 Enterprise Beans. 39
3.1.3 *Enterprise Beans* como Objetos Distribuidos. 46
3.2 Enterprise bean de tipo Entity. 47
3.2.1 Container-Managed Persistence. 48
3.2.2 Bean-Managed Persistence. 54
3.3 Enterprise Beans Tipo Session. 59
3.3.1 Stateless Session Beans. 60
3.3.2 Stateful Session Beans. 64
3.4 Transacciones. 65
3.5 Instalando Enterprise JavaBeans 66
3.6 Clientes Enterprise JavaBeans 68

4 Uso de J2EE **69**

4.1 Necesidades del Cliente 69
4.2 Arquitectura Física del Sistema. 70
4.3 Clases de la capa de presentación. 72
4.3.1 Capa de presentación del subsistema de administración del Sitio Web. 72
4.3.2 Capa de presentación del subsistema de ventas vía Web. . . . 73
4.4 Clases de la capa de lógica de negocio. 74
4.4.1 Entity Beans. 74
4.4.2 Session Beans. 75
4.5 Ventajas y Desventajas. 76

5 Conclusiones **81**

Bibliografía 82
Glosario 85
Apéndice A: JBoss 94
Apéndice B: Ant 95
Apéndice C: Historia de Enterprise JavaBeans 97
Apéndice D: J2EE 1.3 98
Apéndice E: ProductoBean.java 102
Apéndice F: ControlWebBean.java 106

Introducción

En términos generales el comercio electrónico es la posibilidad de realizar transacciones comerciales empleando medios electrónicos, últimamente el medio más empleado es la Internet. La venta en el comercio electrónico se realiza de la misma forma en que se ha desarrollado la venta a través de los tiempos: hay un cliente que necesita un producto o servicio y un proveedor que lo proporciona; este último informa sobre todas las condiciones de su oferta y el cliente decide si la misma cubre sus necesidades. Si se llega a un acuerdo, la venta se realiza.

Cualquier forma de comercio electrónico pone a disposición del usuario, ya sea comprador o vendedor, lo más vanguardista de la tecnología para garantizarle ventajas competitivas.

El comercio electrónico tiene múltiples variantes, desde la simple presencia de un catálogo de productos hasta la entrega de la mercancía al consumidor final. Los modelos más conocidos son las tiendas virtuales y la plaza comercial.

En términos comunes, una tienda virtual tiene como finalidad ofrecer a los clientes un sistema para poder realizar los pedidos y los pagos de forma fácil, segura y acorde a sus necesidades.

Técnicamente es un conjunto de páginas Web normalmente generadas dinámicamente a partir de una base de datos, un grupo de plantillas y un conjunto de recursos.

La tienda tiene una estructura y diseño que permite al cliente tener una idea clara de lo que es la tienda y del tipo de productos que ofrece. Así mismo, cuenta con un buscador para localizar rápidamente los productos en los que está interesado y un carrito de compras, dispositivo que permite ir seleccionando productos, ponerlos en su carrito, y en cualquier momento puede modificar la cantidad de unidades del productos o eliminarlos. El sistema calcula automáticamente el total a pagar y permite seleccionar el tipo de pago a utilizar.

Existen dos formas en las que el cliente puede pagar sus productos, la primera es el

pago contra entrega, en la que el cliente paga sus productos en su domicilio; y pago con tarjeta de crédito, en la que el cliente a través del sitio Web proporciona los datos de la tarjeta y mediante una transacción bancaria se efectua el pago.

Una vez seleccionados los productos y la forma de pago el proveedor recibe, gracias al carrito de compras, el pedido al que dará el tratamiento pertinente para hacerlo llegar al cliente.

En este trabajo se desarrolló un sistema de comercio electrónico llamado Sistema de Ventas Café que pone a disposición de sus clientes la venta de CDs, DVDs y Libros, a través de una tienda virtual sobre la Internet.

El sistema cuenta con una herramienta de administración de los productos del sitio Web. Esta herramienta no tiene interacción con inventarios y sistemas contable-administrativos.

La seguridad en el sistema y las transacciones de pago con tarjeta de crédito no se desarrollaron en el proyecto y se contempla que sean proporcionados por empresas especializadas en los mismos.

Para el desarrollo del sistema se empleó el Proceso Unificado como metodología. Proceso Unificado es un método iterativo que apoya un entendimiento incremental del problema a través de refinamientos y crecimientos sucesivos del software. Proceso Unificado es el producto final de tres decadas de desarrollo y uso practico; su evolución como producto viene desde el *Objectory Process*, liberado en 1987, via *Rational Objectory Process*, liberado en 1997, hasta *Rational Unified Process*, liberado en 1998. Como tecnología de desarrollo se utilizó Java 2 Enterprise Edition (J2EE) que es una tecnología reciente orientada a soluciones empresariales. La tecnología J2EE es el esfuerzo realizado por Sun Microsystems y otras empresas para realizar una plataforma empresarial, la primera especificación fue J2EE 1.2, liberada en diciembre de 1999, y en agosto del 2001 aparece la especificación J2EE 1.3. Para lograr el producto final se dividió el trabajo en dos tesis complementarias "Tecnología J2EE aplicada a un Sistema de Comercio Electrónico" y "Proceso Unificado aplicado al desarrollo de un Sistema de Comercio Electrónico con J2EE" desarrolladas por Griselda González Rodríguez y Raymundo Ortega León respectivamente.

Sistema de Ventas Café es una aplicación distribuida multicapa con los clientes en el *front end*, recursos de datos en el *back end* y una capa intermedia donde se realiza la mayor parte del trabajo de desarrollo de la aplicación. La capa intemedia protege a la capa del cliente de la complejidad de la empresa y toma ventaja de la tecnología de Internet para minimizar la administración y capacitación del usuario.

Java 2 Enterprise Edition^{TM}(J2EE) reduce el costo y complejidad de la aplicación

multicapa al proporcionar una plataforma que provee de una infraestructura del lado del servidor. J2EE simplifica el desarrollo, instalación y mantenimiento de aplicaciones empresariales.

El objetivo de este trabajo es estudiar, comprender, aplicar la tecnología J2EE al sistema de comercio electrónico y darla a conocer.

La tesis está organizada de la siguiente forma:

El Capítulo 1 inicia dando un panorama de lo que es el comercio electrónico. Comercio electrónico es la posibilidad de realizar transacciones comerciales empleando medios electrónicos.

El Capítulo 2 describe a la tecnología Java 2 Enterprise Edition (J2EE) que es una plataforma para desarrollo e instalación de aplicaciones empresariales.

El Capítulo 3 describe a la tecnología Enterprise JavaBeans (EJB) que es una arquitectura de componentes del lado del servidor que permite y simplifica el proceso de construcción de aplicaciones de objetos distribuidos de tipo empresarial.

El Capítulo 4 muestra el uso de la tecnología J2EE en el sistema de comercio electrónico de venta de artículos. Describe el empleo de la tecnología J2EE aplicadas al Sistema de Ventas Café así mismo da las ventajas y desventajas al aplicar dicha tecnología.

El Capítulo 5 son las conclusiones del presente trabajo.

Contiene seis apéndices, el Apéndice A da las características del servidor de aplicaciones JBoss, el Apéndice B da una introducción a la herramienta *Ant*, el Apéndice C describe la historia y evolución de la tecnología EJB, el Apéndice D menciona los cambios más significativos en la especificación J2EE 1.3 y los Apéndices E y F contienen el código fuente de los *beans* Producto y ControlWeb respectivamente.

Capítulo 1

Comercio Electrónico

En términos generales, comercio electrónico es la posibilidad de realizar transacciones comerciales empleando medios electrónicos. La venta en comercio electrónico se realiza de la misma forma en que se ha desarrollado la venta a través de los tiempos: hay un cliente que necesita un producto o servicio y un proveedor que lo proporciona; este último informa sobre todas las condiciones de su oferta y el cliente decide si cubre sus necesidades. Si se llega a un acuerdo, la venta se realiza.

El comercio electrónico tiene múltiples variantes, desde la simple presencia de un catálogo de productos hasta la entrega de la mercancía al consumidor final; puede o no tener interacción con inventarios y sistemas contable-administrativos o bien, contar con la posibilidad de que el propio comprador adapte la información que recibe o el producto mismo. De esta manera, aparece un número creciente de formas de realizar las transacciones comerciales y se caracterizan por su evolución a una mayor complejidad de los modelos y a una mayor integración de tareas.

Los modelos recientes más conocidos son las tiendas virtuales (el comerciante lleva el control total de todas las operaciones y catálogos) y la plaza comercial (varias tiendas se localizan en un dominio común compartiendo infraestructuras y gastos). Sin embargo, existen modelos de mayor complejidad, como los dedicados a licitaciones, subastas y plataformas de colaboración; o bien, los que tienen contemplado un programa de atención al cliente y actualización automática de inventarios. Los requisitos del comercio electrónico son:

- El comercio es conducido vía comunicaciones electrónicas.
- El comercio incluye proveedores,
- consumidores,
- y un tipo de pago.

1.1 Historia.

En el siglo XIX con la introducción del telégrafo en 1845 se dió la primera oportunidad a proveedores y consumidores de conducir el comercio sin estar cerca físicamente. Así mismo, permitió que el intercambio de información fuera más rápido que cualquiera en su tiempo. La independencia en la localización de consumidores y proveedores y la tecnología para procesar más rápido la información fueron y siguen siendo conductores importantes del comercio electrónico. Los beneficios principales del comercio electrónico para el consumidor y proveedor son:

- Amplitud de elección.
- Amplitud de mercado.

El comercio electrónico sigue teniendo la característica del trato anónimo entre compañías y personas e incluso también para consumidores y proveedores de diferentes divisiones en la misma compañía. Se tienen cinco reglas que forman la base para los requerimientos y expectativas del usuario del comercio electrónico :

- El comercio electrónico permite la independencia geográfica del proveedor y del consumidor.
- El comercio electrónico fomenta que las actividades propias del comercio se realicen de forma más rápida.
- El comercio electrónico conduce aplicaciones inter-empresariales e intra-empresariales.
- El comercio electrónico debería soportar necesidades interativas.

  Con la llegada del teléfono, en 1875, el comercio electrónico se aceleró al permitir la realización de transacciones como parte de la misma sesión interactiva.
- El comercio electrónico permite la reingeniería del proceso de negocio para conseguir eficiencia.

  Hasta 1950 el telégrafo y el teléfono fueron los únicos métodos de comercio electrónico. Con la llegada del cómputo comercial, los mecanismos computarizados hicieron más eficientes los procesos administrativos de oficina. Estos sistemas no iban dirigidos al consumidor sino a los intermediarios en el mercado. Los beneficios de estos sistemas de comercio electrónico fueron la reducción en tiempo y costos administrativos. El uso de las computadoras fueron las primeras instancias de reingeniería del proceso de negocio, sustituyendo los procesos en papel con métodos más eficientes.

A finales de 1970 fueron implementadas otras formas de comercio electrónico: el *Electronic Data Interchange*[1] (EDI) formal,la transferencia automatizada de datos entre aplicaciones de computadora, facsímile y el correo electrónico. Durante este periodo surgen las *value-added networks*[2] (VANs) que soportan aplicaciones negocio-a-negocio. Al inicio las VANs fueron establecidas como resultado de las necesidades de un sector específico. Éstas rapidamente soportan EDI genérico y aplicaciones derivadas para comercio entre distintas compañías. Es así como las VANs desempeñaron un papel importante dentro del comercio electrónico al brindar aplicaciones especializadas y comunicaciones requeridas por la comunidad comercial. A fines de la década de los 80's las VANs fueron el mecanismo dominante para soportar el comercio electrónico entre grandes compañías, sus divisiones y socios a través de un amplio rango de tecnologías.

El comercio electrónico dependió principalmente de las VANs y redes de mensajería privadas, ambas se caracterizaron por los altos costos y la conectividad limitada. Los puntos de venta de la VAN tienen alta seguridad, confiabilidad y confirmación de recepción.

La Internet tiene conectividad mundial, crece en todos los segmentos de la sociedad, es interactiva y su uso es relativamente barato. El problema es que no hay autoridad central para el control de la Internet y la confiabilidad está en duda. La entrega no se garantiza y la seguridad se considera inexistente.

Ambas alternativas están creciendo y produciendo nuevos desarrollos técnicos.

1.2 Tipos de Comercio Electrónico.

El comercio electrónico, según los agentes implicados, puede subdividirse en las siguientes categorías:

- Comercio entre empresas (*Business to Business* o B2B):
  Es la posibilidad de intercambiar bienes o servicios, a través de la Internet o de otras redes de comunicación, para que sean integrados en la cadena de valor de otra empresa. Especialmente utilizados para el intercambio de propuestas, pedidos, facturas y otros.

[1]Intercambio Electrónico de Datos
[2]Red de Valor Agregado

- Comercio entre empresas y consumidores (*Business to Customer* o B2C):
  En este caso, el intercambio de bienes y servicios es a través de la Internet o de otras redes de comunicación, se produce entre empresas y clientes finales.

- Comercio entre consumidores y consumidores (*Consumer to Consumer* o C2C):
  Subastas en las que usuarios particulares venden productos.

- Comercio entre consumidores y empresas (*Consumer to Business* o C2B):
  Los consumidores particulares se agrupan para tener más fuerza y hacer pedidos a empresas.

- Comercio entre administración y empresas, consumidores o administraciones (*Administration to Business/Consumer or Administration* A2B/C/A):
  Relaciones con las administraciones públicas y los consumidores, empresas u otras administraciones.

- Comercio de amigo a amigo (*Peer to Peer* o P2P):
  Relaciones de amigo a amigo como el intercambio de música con programas como Napster, etc.

- Comercio entre empresa y trabajador (*Business to Employee* o B2E):
  Comunicación entre empresa y trabajador.

1.3 Tipos de Pago para el Comercio Electrónico.

Las principales formas de pago en el comercio electrónico son:

- **Pago contra entrega.**
  El cobro de los productos se efectua cuando un representante del proveedor entrega los mismos en el domicilio del consumidor. El pago es directo y sin transacciones electrónicas.

- **Efectivo electrónico.**
  El efectivo electrónico o dinero digital proporciona los medios para transferir dinero entre el proveedor y el consumidor sobre una red como la Internet. El efectivo electrónico emula las propiedades del dinero convencional y la forma en la que se realizan las transacciones con éste.

- **Tarjeta de Crédito.**

  Esta forma de pago tiene cuatro componentes típicos:

  1. El consumidor con un navegador Web.
  2. El servidor del proveedor que proporciona una página. El servidor toma las transacciones de la tarjeta.
  3. El banco del proveedor realiza las transacciones de la tarjeta.
  4. La institución que emitió la tarjeta al consumidor.

  La transacción electrónica de tarjeta de crédito se realiza en tres fases. La primera fase concluye con la compra de bienes por el consumidor. La segunda fase termina con la transferencia de dinero desde la cuenta del consumidor hacia el proveedor. La tercera fase informa al consumidor acerca de las deducciones a su cuenta.

1.4 Tecnologías del Comercio Electrónico.

Estas son las tecnologías más importantes dentro del comercio electrónico:

1. ***Electronic Data Interchange*** **(EDI).** El EDI es el intercambio computadora-computadora de información de negocio estructurada en un formato electrónico.

2. **Código de Barras**. Son usados para la identificación automática de productos por una computadora.

3. **Correo electrónico.** Mensajes escritos por un emisor y mandados en forma digital hacia los receptores.

4. **Internet**. La Internet es una red global descentralizada de millones de computadoras y redes de computadoras. La Internet es una herramienta para la comunicación entre personas, negocios y combinaciones de éstos. La red está creciendo rápidamente y cada vez tiene más usuarios.

5. ***World Wide Web*****(WWW)**. La WWW, comúnmente llamada Web, es una colección de documentos escritos y codificados con *HyperText Markup Language*[3] (HTML). Los documentos HTML son popularmente conocidos como páginas o portales. Con la ayuda de piezas de software llamadas navegadores el

[3] Lenguaje de Marcado de HiperTexto

usuario solicita estos documentos y son desplegados en su computadora local. Los documentos HTML contienen diferentes tipos de información como texto, imágenes, video, audio y ligas que puedan transportar al usuario a otras páginas Web. La Web es la aplicación más usada de la Internet.

6. **Formas electrónicas.** Las formas electrónicas son una tecnología que combina la familiaridad de las formas en papel con el poder del almacenamiento de información en forma digital para que esta información sea integrada a otras aplicaciones.

1.5 Seguridad en la Internet comercial.

La Internet original fue diseñada para la investigación y no para un ambiente comercial. Como tal, operaba en un dominio de confianza y la seguridad estaba relacionada con el respeto y honor entre los usuarios. De esta forma sólo se proporcionaba un nivel menor de seguridad.

Conforme la Internet creció, la comunidad se expandió y la seguridad existente era pobre. Un requisito indispensable en el comercio electrónico es garantizar la seguridad de las transacciones entre los compradores y los vendedores. El consumidor requiere que los datos personales que suministra en la transacción no sean capturados en la transmisión por alguien que posteriormente pudiera suplantar su identidad; por su parte el proveedor (o vendedor) debe asegurarse de la identidad de aquel que efectúa el pedido.

Para recibir los pagos seguros es preciso tener instalada una aplicación de comercio electrónico que genere identificadores únicos de pedido y cantidad total a pagar para los mismos, a ésta se le llama terminal punto de venta virtual (TPVV).

Toda TPVV debe funcionar sobre un servidor seguro utilizando mecanismos para codificar la información antes de viajar y un conjunto de reglas que determinan cómo se realizará el intercambio de información entre dos computadoras (protocolo) para garantizar que sólo emisor y receptor podrán entender la información.

Actualmente la mayoría de los pagos en línea es realizada con tarjetas de crédito, empleando alguno de estos protocolos:

- *Secure Socket Layer* (SSL): Estándar técnico abierto para la industria del comercio desarrollada por Visa y Mastercard como forma de facilitar transacciones seguras de pago con tarjeta sobre la Internet. Es un sistema que consiste en

codificar la información antes de enviarla, de forma que sólo el destinatario pueda conocer la información, evitando que accesos fraudulentos a ésta puedan hacer mal uso de la misma. Se puede suplantar la identidad del emisor.

- *Secure Electronic Transaction* (SET): Su objetivo es proteger los datos sensibles de los compradores respetando la confidencialidad de los datos y autentificando la identidad de todas las partes que intervienen. SET utiliza un sistema de firmas y certificados digitales que asegura que el emisor es quien dice ser y que sólo puede leer el mensaje el receptor autorizado.
- MOSET: Una combinación de los dos anteriores, pues la primera parte de la transacción (el envió de información al comerciante) se hace empleando SSL, en tanto, la comunicación entre el comerciante y el banco utiliza SET.

1.6 Características de Sistema de Ventas Café.

Sistemas de Ventas Café es una aplicación de tipo empresa-consumidor que utiliza tecnología Web, maneja el tipo de pago contra entrega y por tarjeta de crédito.

La seguridad en el sistema para el manejo de información sensible y las transacciones de pago con tarjeta de crédito están fuera del alcance del trabajo.

Capítulo 2

Java 2 Enterprise Edition

Las empresas de hoy necesitan extender su alcance, reducir sus costos y bajar sus tiempos de respuesta brindando servicios de fácil acceso a sus clientes, socios, empleados y proveedores.

Típicamente, las aplicaciones que proveen de estos servicios deben combinar *Enterprise Information System*[1] (EIS) existentes con nuevas funciones de negocio que entregan servicios a un amplio rango de usuarios. Estos servicios necesitan ser:

- *fácilmente disponibles*, para encontrar las necesidades de negocios globales de actualidad,
- *seguros*, para proteger la privacidad de usuarios y la integridad de los datos empresariales,
- *confiables y escalables*, para asegurar que las transacciones son exactas y rápidamente procesadas.

Por una variedad de razones, estos servicios son generalmente catalogados como aplicaciones distribuidas que consisten de muchas capas, incluyendo clientes en el *front end*, recursos de datos en el *back end* y una o más capas intermedias entre ellas donde se realiza la mayor parte del trabajo de desarrollo de la aplicación. La capa intermedia implementa los nuevos servicios que integran a los EIS existentes con las nuevas funciones del negocio y los datos de los nuevos servicios. La capa intemedia protege a la capa del cliente de la complejidad de la empresa y toma ventaja de tecnologías como Internet que maduraron rápidamente para minimizar la administración y capacitación del usuario.

[1]Sistemas de Información Empresarial

La plataforma *Java 2 Enterprise Edition™* (J2EE) reduce el costo y complejidad del desarrollo de estos servicios multicapa, resultando en servicios que pueden ser rápidamente instalados y fácilmente mejorados para que las empresas respondan a presiones competitivas.

2.1 Orígenes de J2EE.

Durante los primeros años de los noventas, los proveedores de EIS empiezan a responder a necesidades del cliente moviendose del modelo de aplicación cliente-servidor, de dos capas, a un modelo más flexible de tres capas y multicapas. Los modelos nuevos separan la lógica de negocio de los sistemas de servicio y de la interfaz de usuario, colocándola en la capa intermedia entre los dos. La evolución de los nuevos servicios de la capa intermedia - monitores de transacción, mensajes orientados a la capa intermedia, *object request brokers* y otros - dan impulsos adicionales a esta nueva arquitectura. Al mismo tiempo, el uso creciente de la Internet e *intranet* para aplicaciones empresariales contribuyeron a obtener un énfasis mayor por su fácil despliegue a los clientes.

El diseño multicapa simplifica el desarrollo, instalación y mantenimiento de aplicaciones empresariales. Permite a los desarrolladores enfocarse en lo esencial de la programación, sobre la lógica del negocio, contando con diversos servicios de *back-end* para proveer la infraestructura y aplicaciones del lado del cliente (*standalone* y navegadores Web) para proporcionar la interacción de usuario. Una vez desarrollada, la lógica del negocio puede ser instalada sobre servidores apropiados para las necesidades de la organización. Sin embargo, a pesar de estos beneficios, el modelo limita la habilidad del desarrollador para construir aplicaciones de componentes estandarizados, para instalar una aplicación sencilla sobre una gran variedad de plataformas o para escalar rápidamente aplicaciones para encontrar las condiciones cambiantes del negocio.

Dentro de Sun Microsystems, mucho esfuerzo de desarrollo apuntaba a lo que llegaría a ser la tecnología J2EE. Primero, la tecnología *servlet* mostró que los desarrolladores estaban ansiosos de crear *Common Gateway Interface* (CGI) como el comportamiento que podria correr sobre cualquier servidor Web que soportara la plataforma Java™. Segundo, la tecnología *Java Database Connectivity* (JDBC) proporciona un modelo con las características – **escribe una vez, ejecuta donde sea** – del lenguaje de programación Java hacia sistemas de bases de datos existentes. Finalmente, el éxito de la arquitectura de componentes *Enterprise JavaBeans* demostró la utilidad del encapsulamiento completo de conjuntos de comportamientos en componentes rápidamente reusables y fácilmente configurables. La convergencia de estos tres conceptos - con-

ducta del lado del servidor escrita en el lenguaje Java, conectores para permitir el acceso a sistemas empresariales existentes, modularización, fácil para la instalación de componentes - llevando a Sun Microsystems y a sus socios industriales al estándar J2EE.

2.2 Plataforma J2EE.

La plataforma J2EE usa un modelo de aplicación multicapa distribuido. Esto significa que la lógica de aplicación está divida de acuerdo a la función y a los diversos componentes, que constituyen a la aplicación J2EE, para que sean instalados sobre diferentes máquinas dependiendo sobre que capa del ambiente J2EE multicapa pertenece el componente de la aplicación.

Mientras que una aplicación J2EE puede consistir de tres o cuatro capas, las aplicaciones multicapa J2EE son generalmente consideradas de tres capas porque son distribuidas sobre diferentes lugares: máquina cliente, máquina servidor y máquinas de base de datos o sistemas legados en el *back end*. Las aplicaciones de tres capas que se ejecutan de esta forma extienden el modelo cliente servidor estándar de dos capas poniendo un servidor de aplicación multicapa entre la aplicación cliente y el medio de almacenaje.

2.2.1 Tecnologías J2EE.

La plataforma J2EE especifica las tecnologías para soportar aplicaciones empresariales multicapa. Estas tecnologías se dividen principalmente en tres categorías: componentes, servicios y comunicación.

La tecnología de componentes es usada por los desarrolladores para crear las partes esenciales de la aplicación empresarial, la interfaz de usuario y la lógica de negocio. La tecnología de componentes permite que los desarrollos de módulos puedan ser reusados por multiples aplicaciones empresariales. La tecnología de componentes es soportada por servicios a nivel de sistema de la plataforma J2EE que simplifican la programación de la aplicación y permiten a los componentes ser adecuados para usar los recursos disponibles en el ambiente sobre el cual son instalados.

Muchas aplicaciones requieren tener acceso a EIS existentes, la plataforma J2EE provee de los *Application Program Interfaces* (APIs) para acceso de base de datos, manejo de transacción, servicios de nombrado y directorio, y mensajes. Finalmente, J2EE provee de tecnología que permiten la comunicación entre clientes y servidores

y entre objetos colaborativos instalados en distintos servidores.

Tecnología de Componentes.

Un componente es una unidad de software a nivel de aplicación. La plataforma J2EE soporta los siguientes tipos de componentes: componentes *JavaBean™*, *applet*, clientes de aplicación, componentes *Enterpriese JavaBean* (EJB) y componentes de Web. Los clientes de aplicación y *applets* se ejecutan sobre la plataforma cliente, los componentes de Web y EJB se ejecutan sobre la plataforma del servidor.

Todos los componentes J2EE dependen del soporte en tiempo de ejecución de una entidad a nivel de sistema llamada contenedor. Los contenedores proporcionan a los componentes de servicios como manejo del ciclo de vida, seguridad, instalación y manejo de hilos de ejecución[2]. Como los contenedores manejan estos servicios, el comportamiento de muchos componentes puede ser declarativamente adecuado a través del *Deployment Descriptor* cuando el componente es instalado en el contenendor.

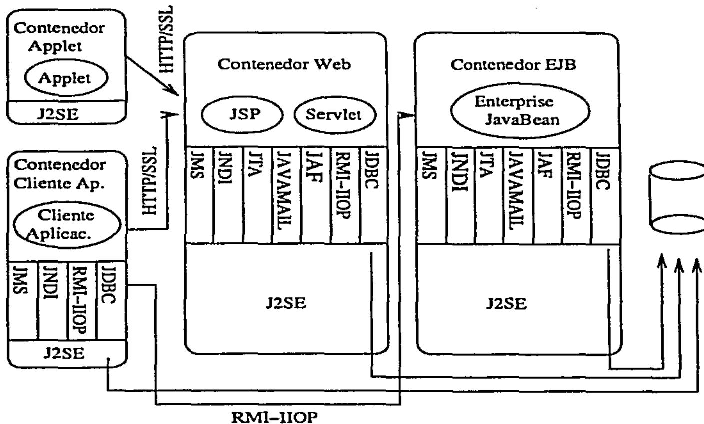

Figura 2.1: Componentes y Contenedores.

[2]Traduccion literal de threading

Applets y Clientes de Aplicación. Los *applets* y clientes de aplicación son componentes cliente que se ejecutan sobre su propia máquina virtual Java. Un contenedor de *applet* incluye soporte para el modelo de programación del *applet*. Un cliente J2EE puede hacer uso del Java *Plug-in* para proveer el ambiente de ejecución que requiere el *applet*. El contenendor para el cliente de aplicación proporciona acceso a los APIs de servicios y comunicación J2EE.

Componentes de Web. Un componente de Web es una entidad de software que proporciona una respuesta a una requisición. Un componente de Web típicamente genera la interfaz de usuario para una aplicación basada en componentes de Web.

Servlet. Un *servlet* es un programa que extiende la funcionalidad del servidor de Web. Los *servlets* reciben una requisición de un cliente, dinámicamente generan la respuesta (posiblemente consultando una base de datos para satisfacer la requisición) y manda la respuesta en un documento HTML o *eXtensible Markup Language* (XML) al cliente.

El desarrollador de *servlet* usa el API para:

- Inicializar y finalizar al *servlet*.
- Tener acceso al ambiente del *servlet*.
- Recibir requisiciones y mandar respuestas.
- Mantener información de la sesión del cliente.
- Interactuar con otros *servlets* y componentes.

JavaServer Pages. La tecnología *JavaServer Pages™* (JSP™) proporciona una forma extendible para generar contenido dinámico hacia un cliente Web. Una página JSP es un documento basado en texto que describe como procesar una requisición para crear una respuesta. Una página JSP contiene:

- Una plantilla de datos para dar formato al documento Web. Típicamente la plantilla usa elementos HTML o XML. Los diseñadores pueden editar y trabajar con estos elementos sobre la página JSP sin afectar el contenido dinámico. Este enfoque simplifica el desarrollo porque separa la presentación de la generación del contenido dinámico.

- Elementos JSP y *scriptlets* generan el contenido dinámico en el documento Web. Muchas páginas JSP usan componentes *JavaBeans* o EJB para realizar procesamiento más complejo que requiere la aplicación. Acciones JSP estándar

pueden tener acceso e instanciar al *bean*, establecer o recuperar atributos del *bean* y descargar *applets*. JSP es extendible a través del desarrollo de acciones adecuadas, las cuales son encapsuladas en bibliotecas de etiquetas.

Contenedores de Componentes de Web. Los componentes de Web son instalados por contenedores de *servlet*, JSP y Web. Además de los servicios estándar del contenedor, un contenedor *servlet* proporciona servicios de red (al que requisiciones y respuestas son mandadas), requisiciones decodificadas y respuestas de formatos. Todos los contenedores *servlets* deben soportar el protocolo HTTP para requisiciones y respuestas, pero ademas pueden soportar otros protocolos requisición-respuesta como el HTTPS. Un contenedor JSP provee del mismo servicio que el contenedor de *servlet* y una máquina que interpreta y procesa la página JSP a un *servlet*. Un contenedor de Web provee de los mismos servicios que el contenedor JSP y tiene acceso a los servicios J2EE y los APIs de comunicación.

Componentes Enterprise JavaBean. La arquitectura *Enterprise JavaBean*™ (EJB™) es una tecnología del lado del servidor para desarrollar e instalar componentes que contienen la lógica del negocio de una aplicación empresarial. Los componentes EJB, llamados *enterprise bean*, son escalables, transaccionales y seguros. Hay dos tipos de *enterprise bean*: *session* y *entity beans*.

Session Bean. Un *session bean* es creado para proporcionar algún servicio a nombre del cliente y usualmente existe sólo para la sesión cliente-servidor. Un *session bean* realiza operaciones como cálculos o acceso a base de datos para el cliente.

Entity Bean. Un *entity bean* es un objeto persistente que representa datos mantenidos en un medio de almacenaje, su foco está centrado en el dato. Un *entity bean* puede manejar su propia persistencia o puede delegar esta función al contenedor. El *bean* es identificado por una llave primaria.

Contenedores de Componentes EJB. Los *enterprise bean* son instalados por un contenedor EJB. Además de los servicios estándares del contenedor, un contenedor EJB proporciona un rango de servicios de transacción y persistencia y tiene acceso a los servicios J2EE y a los APIs de comunicación.

Tecnología de Servicio.

La tecnología de Servicio de la plataforma J2EE permite a las aplicaciones tener acceso a un amplio rango de servicios de manera uniforme.

Java Database Connectivity. El API de *Java Database Connectivity^{TM}* (JDBC^{TM}) proporciona una conectividad independiente de la base de datos entre la plataforma J2EE y un amplio rango de recursos de datos tabulares. La tecnología JDBC permite al *Application Component Provider*[3] a:

- Realizar la conexión y autenticación a un servidor de base de datos.
- Manejar transacciones.
- Mover sentencias SQL a la máquina de base de datos para preprocesarlas y ejecutarlas.
- Ejecutar procedimientos almacenados.
- Inspeccionar y modificar los resultados de sentencias *select.*

La plataforma J2EE require de *JDBC 2.0 Core API* (incluido en la plataforma J2SE) y el *JDBC 2.0 Extension API*, que provee *row sets*, conexión de nombrado vía *Java Naming and Directory Interface* (JNDI), *connection pooling* y soporte de transacciones distribuidas. Las características de *connection pooling* y transacciones distribuidas son destinadas para uso de *drivers* JDBC para la coordinación con el servidor J2EE.

Java Transaction API y Service. El *Java^{TM}Transaction API* (JTA) permite a las aplicaciones tener acceso a transacciones de manera que sean independientes a implementaciones específicas. JTA especifica interfaces estándares de Java entre el manejador de transacción y las partes involucradas en un sistema de transacción: la aplicación transaccional, el servidor J2EE y el manejador que controla el acceso a recursos compartidos afectados por la transacción.

El *Java^{TM}Transaction Service* (JTS) especifica la implementación de un manejador de transacción que soporta JTA e implementa el mapeo de Java de la especificación *Object Management Group Object Transaction Service 1.1.* Un manejador de transacciones JTS proporciona los servicios y manejo de funciones requeridas para soportar la

[3]Provedor de Componentes de Aplicación

demarcación de la transacción, el manejo de recursos transaccionales, sincronización y propagación de información que es específica a una instancia particular de transacción.

Java Naming and Directory Interface. El API *Java Naming and Directory Interface*^{TM} (JNDI) proporciona la funcionalidad de nombrado y directorio. Provee a las aplicaciones con métodos para realizar operaciones estándares de directorio, como asociación de atributos con objetos y búsqueda de objetos usando sus atributos. Utilizando JNDI, una aplicación puede almacenar y recuperar cualquier tipo de objeto Java.

Como JNDI es independiente de cualquier implementación específica, las aplicaciones pueden usar JNDI para tener acceso a multiples servicios de nombrado y directorio, incluyendo los servicios existentes como LDAP, NDS, DNS y NIS. Esto permite a las aplicaciones coexistir con aplicaciones y sistemas legados.

Tecnología J2EE Connector. La tecnología J2EE *Connector* permite a componentes J2EE como *enterprise bean* interactuar con EIS. El software de EIS incluye varios tipos de sistemas: planeación de recursos empresariales, procesamiento de transacción en *mainframe* y bases de datos no relacionales entre otros. La tecnología J2EE *Connector* simplifica la integración de diversos sistemas EIS. Cada EIS requiere sólo de una implementación de la tecnología J2EE *Connector*. La implementación agrega a la especificación del J2EE *Connector*, la portabilidad a través de todos los servidores J2EE.

Para usar la tecnología J2EE *Connector*, un vendedor de EIS provee de un *connector* estándar para su EIS. El *connector* tiene la capacidad para adaptarse a cualquier servidor EJB que soporte la tecnología J2EE *Connector*. Similarmente, un vendedor de servidor EJB extiende su sistema para soportar esta tecnología J2EE *Connector* y entonces se asegura de la conectividad completa de multiples EIS.

Tecnología de Comunicación.

La tecnología de comunicación proporciona de un mecanismo de comunicación entre clientes y servidores y entre los objetos colaborativos instalados en diferentes servidores. La especificación J2EE requiere del soporte para los siguientes tipos de tecnología de comunicación:

- Protocolos de Internet.
- Protocolos de *Remote Method Invocation.*
- Protocolos del *Object Management Group.*
- Tecnología de mensajes.
- Formatos de datos.

Protocolos de Internet. Los protocolos de Internet definen los estándares sobre los cuales las diferentes piezas de la plataforma J2EE se comunican unas con otras y con entidades remotas. La plataforma J2EE soporta los siguientes protocolos de Internet:

- TCP/IP – *Transport Control Protocol over Internet Protocol.* Estos dos protocolos proveen de la entrega confiable de flujos de datos de un *host* a otro. *Internet Protocol* (IP), el protocolo básico de Internet, habilita la entrega no confiable de paquetes individuales de un *host* a otro. *Transport Control Protocol* (TCP) agrega las nociones de conexión y confiabilidad.

- HTTP 1.0 – *Hypertext Transfer Protocol.* El protocolo Internet es usado para recuperar objetos de hipertexto de *host* remotos. El mensaje HTTP consiste de requisiciones del cliente hacia el servidor y la respuesta del servidor al cliente.

- SSL 3.0 – *Secure Socket Layer.* Un protocolo de seguridad que proporciona la privacidad sobre la Internet. El protocolo permite a las aplicaciones cliente-servidor comunicarse en una forma que no pueda ser *eavesdropped* o *tampered.* Los servidores siempre son autenticados y los clientes opcionalmente son autenticados.

Protocolos de *Remote Method Invocation.* *Remote Method Invocation* (RMI) es un conjunto de APIs que permite a los desarrolladores construir aplicaciones distribuidas en el lenguaje de programación Java. RMI usa interfaces del lenguaje Java para definir objetos remotos y una combinación de la tecnología de serialización Java y el *Java Remote Method Protocol* (JRMP) para regresar las invocaciones de métodos locales en invocaciones de métodos remotos. La plataforma J2EE soporta el protocolo JRMP, el mecanismo de transporte para comunicación entre objetos en el lenguaje Java en diferentes espacios de direcciones.

Protocolos del *Object Management Group.* Los protocolos del *Object Management Group* (OMG) permiten a los objetos instalados por la plataforma J2EE tener acceso a objetos remotos desarrollados usando la tecnología *Common Object Request Broker Architecture* (CORBA) de OMG y viceversa. Los objetos CORBA

son definidos usando *Interface Definition Language* (IDL). Un *Application Component Provider* define la interfaz del objeto remoto en IDL y entonces usa un compilador IDL para generar los *stub* cliente y servidor que conectan a las implementaciones de los objetos a un *Object Request Broker* (ORB), una biblioteca que permite a objetos CORBA localizar y comunicarse con otros. Los ORBs se comunican con otros usando *Internet Inter-ORB Protocol* (IIOP). La tecnología OMG requerida por la plataforma J2EE es Java IDL y RMI-IIOP

Java IDL. Java IDL permite a clientes Java invocar operaciones de objetos CORBA que han sido definidos usando IDL y han sido implementados en cualquier lenguaje con un mapeo CORBA. Java IDL es parte de la plataforma J2SE. Ésta consiste de un API CORBA y ORB. El *Application Component Provider* usa el compilador IDL llamado *idlj* para generar el *stub* cliente Java para un objeto CORBA definido en el IDL. El cliente Java es ligado con el *stub* y usa el API CORBA para tener acceso al objeto CORBA.

RMI-IIOP. RMI-IIOP es una implementación del API RMI sobre IIOP. RMI-IIOP permite a *Application Component Providers* escribir interfaces remotas en el lenguaje de programación Java. La interfaz remota puede ser convertida a un IDL e implementada en cualquier otro lenguaje que sea soportado por un mapeo OMG y ORB para el lenguaje. Clientes y servidores pueden escribirse en cualquier lenguaje usando IDL derivado de las interfaces RMI. Cuando las interfaces remotas son definidas como interfaces Java RMI, RMI sobre IIOP provee de interoperbilidad con objetos CORBA implementados en cualquier lenguaje. RMI-IIOP contiene:

- El compilador rmic genera:
  - El *stub* cliente y servidor que trabajan con cualquier ORB.
  - Un archivo IDL compatible con la interfaz RMI. Para crear un objeto servidor C++, un *Application Component Provider* usaría un compilador para producir el *stub* servidor y el *skeleton* para el objeto servidor.
- Un API CORBA y ORB

Los clientes de aplicación deben usar RMI-IIOP para comunicarse con *enterprise beans*.

Tecnología de Mensajes. La tecnología de mensajes provee de una forma de mandar y recibir mensajes asincrónicamente. El *Java Message Service* (JMS) provee de una interfaz para tomar requisiciones, reportes o eventos asincrónicamente que son consumidos por aplicaciones empresariales. Mensajes JMS son usados para coordinar estas aplicaciones. El API *JavaMail* proporciona de una interfaz para mandar y recibir mensajes destinados a usuarios. Aunque el API pueda usar notificaciones asíncronas, se prefiere JMS cuando la velocidad y confiabilidad son los requerimientos primarios.

Java Message Service. El *Java Message Service* (JMS) es un API para usar sistemas de mensajería empresarial como *IBM MQ Series* y *TIBCO Rendezvous*. Mensajes JMS contienen información bien definida que describe acciones de negocio específicas. Aún con el intercambio de estos mensajes, las aplicaciones siguen el progreso de la empresa. JMS soporta estilos de mensajería punto a punto y publica-subscribe.

En mensajería punto a punto, un cliente manda un mensaje a la cola de mensajes de otro cliente. Frecuentemente un cliente tendrá todos sus mensajes entregados en una cola. Muchas colas son creadas administrativamente y son tratadas como recursos estáticos por sus clientes.

En mensajería publica-subscribe, los clientes publican mensajes *to* y subscriben el mensaje *from*, hay nodos bien conocidos en una jerarquía basada en contenido llamada tópico. Un tópico puede ser pensado como un agente de mensajes que junta y distribuye los mensajes direccionandolos a éste. Por contar con el tópico como un intermediario, los publicadores de mensajes son independientes de los subcriptores y viceversa. El tópico automáticamente se adapta como los publicadores y subscriptores vienen y van. Los publicadores y subscriptores están activos cuando los objetos que representan existen. JMS también soporta la durabilidad opcional de subscriptores para recordar la existencia de los subscriptores mientras están inactivos.

Las definiciones del API JMS deben estar incluidas en un producto J2EE, pero al producto no se le requiere que incluya la implementación de los objetos *Connection-Factory* y *Destination* de JMS. Éstos son los objetos usados por una aplicación para tener acceso al proveedor de servicios JMS.

JavaMail. El API *JavaMail*^{TM} proporciona de un conjunto de clases abstractas e interfaces que comprende un sistema de correo electrónico. Las clases abstractas e interfaces soportan varias implementaciones diferentes de almacén de mensajes, formatos y transportes. Muchas aplicaciones simples solo necesitarán interactuar con el sistema de mensajería a través de estas clases e interfaces base.

Las clases abstractas en *JavaMail* pueden ser heredadas para proveer de nuevos protocolos y agregar funcionalidad cuando sea necesario. Además, *JavaMail* incluye subclases concretas que implementan protocolos de correo de Internet usados ampliamente y conforme a las especificaciones RFC822 y RFC2045. Están listas para usarse en el desarrollo de la aplicación. Los desarrolladores pueden heredar clases de *JavaMail* para proporcionar las implementaciones de un sistema de mensajes particular como *IMAP4*, *POP3*, y *SMTP*.

JavaBeans Activation Framework.

El *JavaBeans Activation Framework* (JAF) integra el soporte para tipos de datos *Multipurpose Internet Mail Extensions* (MIME) en la plataforma Java. Componentes *JavaBeans* pueden ser especificados para operar sobre datos *MIME*, como visualización o edición de datos. El JAF también provee de un mecanismo para mapear extensiones de nombres de archivos a tipos MIME.

El JAF es usado por *JavaMail* para tomar el dato incluido en mensajes de correo electrónico; aplicaciones típicas no necesitarán usar el JAF directamente, aunque las aplicaciones que hacen uso sofisticado del correo electrónico pueden necesitarlo.

Formatos de Datos. Los formatos de datos definen el tipo de datos que pueden ser intercambiados entre componentes. La plataforma J2EE requiere del soporte para los siguientes formatos de datos:

- HTML 3.2. El lenguaje de marcado usado para definir documentos de hipertexto disponibles sobre la Internet. HTML permite que imagenes, sonido, video, campos de formas, referencias a otros documentos HTML y formatos de texto básico sean embebidos. Los documentos HTML tienen una dirección global única y pueden tener una liga a otra dirección.
- Archivos de imagen. La plataforma J2EE soporta dos formatos para archivos de imágenes: *Graphics Interchange Format* (GIF), un protocolo para la transmición en línea e intercambio de tramos de datos gráficos y *Joint Photographic Experts Group* (JPEG), un estándar para compresión de escalas de grises o imagenes a color.
- Archivo JAR. Un formato de archivo independiente de la plataforma para ser integrado en un sólo archivo.
- Archivo *class*. El formato de un archivo Java compilado tal como está establecido en la especificación de la máquina virtual de Java. Cada archivo *class* contiene una clase o interfaz del lenguaje Java y consiste de un flujo de bytes.

- XML. Un lenguaje de marcado basado en texto que permite definir las marcas necesarias para identificar los datos y texto en documentos XML. Como con HTML, la identificación de los datos se logra a través de las etiquetas. Pero a diferencia de HTML, las etiquetas XML describen datos, más que dar un formato de despliegue. De la misma forma en que se definen los nombres de los campos para una estructura de datos, se está en libertad de usar cualquier etiqueta XML que tenga sentido para una aplicación dada. Cuando multiples aplicaciones usan el mismo dato XML, tienen que acordar sobre el nombre de las etiquetas que quieren usar.

2.3 Escenarios de Aplicación J2EE.

La especificación y tecnología J2EE pueden, por definición, hacer pocas suposiciones acerca de como los APIs serán usados para dar la funcionalidad a nivel aplicación. Las decisiones y elecciones a este nivel son acuerdos, entre la funcionalidad y la complejidad.

El modelo de programación J2EE necesita adoptar escenarios de aplicación que tratan con el contenedor de Web y EJB como entidades lógicas opcionales. La Figura 2.2 refleja algunos escenarios clave incluyendo aquellos donde el contendor de Web o EJB y potencialmente ambos son omitidos.

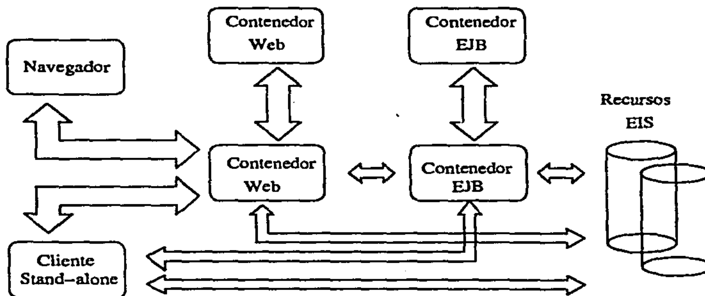

Figura 2.2: Escenarios de Aplicación J2EE.

La aplicación simple refleja un modelo de aplicación multicapa. Esta decisión asume la presencia tando de un contenedor Web como un contenedor EJB. Los siguientes requerimientos empresariales influyen mucho en las elecciones hechas:

- La necesidad de hacer cambios rápidos y frecuentes en la aplicación.
- La necesidad para particionar la aplicación por líneas de presentación y lógica del negocio así como para incrementar la modularidad.
- La necesidad para simplificar el proceso de asignar recursos humanos capacitados para terminar la tarea de desarrollo tal que el trabajo pueda proseguir por vías cooperativas pero relativamente independientes.
- La necesidad de tener desarolladores familiarizados con los procesos de negocio sin agobiarse del trabajo de diseño gráfico e interfaz de usuario, para lo cual pueden no estar idealmente calificados.
- La necesidad de tener el vocabulario necesario para comunicar la lógica del negocio a los equipos involucrados con factores humanos y a los que realizan el trabajo de estética de la aplicación.
- La habilidad para ensamblar aplicaciones usando componentes de una gran variedad de fuentes, incluyendo componentes lógicos de negocios externos.
- La habilidad de instalar componentes transaccionales a través de multiples plataformas de sotfware y hardware independiente de la tecnología de base de datos.
- La habilidad para exteriorizar datos internos sin tener que hacer muchas suposiciones acerca del consumidor de los datos.

Claramente la omisión de cualquier o todos estos requerimientos podría influir en algunas desiciones y elecciones a nivel aplicación que un diseñador podría realizar. El modelo de programación J2EE toma el enfoque que es muy deseable para diseñar una aplicación de 3 capas tal que la migración de una arquitectura multicapa futura es simplificada a través de la reusabilidad de componentes. Aunque es razonable hablar de una lógica de presentación deshechable (esto es, aplicaciones con una presentación que cambian rápidamente), esto es todavía inercia significante asociada con la lógica del negocio. Esto es aún más frecuente en el caso de esquemas de bases de datos y datos en general.

En resumen, el modelo de programación J2EE promueve un modelo que se anticipa al crecimiento, alienta la reusabilidad de código orientado a componentes y fortalece la

comunicación entre capas. Esto es la integración de capas que quedan en el corazón del modelo de programación J2EE.

La Figura 2.2 ilustra un número de escenarios de aplicación que un producto J2EE debería ser capaz de soportar. Desde una perspectiva J2EE, aquí no hay ninguna inclinación implícita a favor de un escenario de aplicación sobre otro. Sin embargo, un producto J2EE debería soportar cualquiera de estos escenarios.

2.3.1 Escenario de Aplicación Multicapa.

La Figura 2.3 ilustra un escenario de aplicación en el cual el contenedor de Web instala componentes de Web que son casi exclusivamente dedicados a tomar la lógica de presentación de la aplicación. La entrega de contenido Web dinámico a un cliente es responsabilidad de la página JSP. El contenedor EJB instala componentes de aplicación que responden a requisiciones de la capa Web y tiene acceso a recursos EIS. La habilidad de desasociar el acceso de datos de interacciones de usuario final es lo que da fuerza a este escenario particular. La aplicación es implicitamente escalable. Pero lo más importante, la funcionalidad de la aplicación está relativamente aislada de la interfaz gráfica del usuario final.

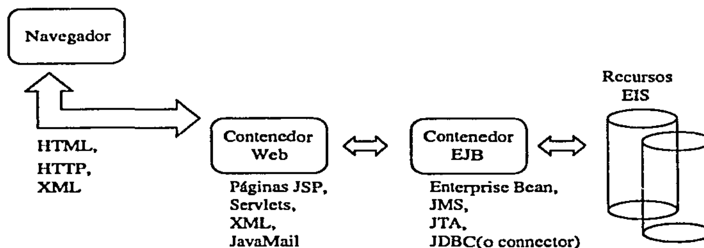

Figura 2.3: Aplicación Multicapa.

XML es incluido como parte esencial de este escenario. El rol de mensajería de datos XML de producir y consumir mensajes de datos XML en el contenedor Web es visto como una forma extremadamente flexible de adoptar un conjunto diverso de plataformas cliente. Estas plataformas pueden variar en el rango desde navegadores de propósito general habilitados con XML hasta máquinas de entrega de XML especializado dirigido a soluciones verticales. Independiente de las áreas de aplicación

específica, se asume que los mensajes de datos XML utilizarán HTTP como su transporte de comunicación. El término mensajería de datos XML es usado para denotar el modelo de programación donde XML es usado para intercambiar información como lo opuesto a la promoción del modelo de objetos ortogonal al modelo de objetos Java. La relación de XML con Java es vista como complementaria.

En la capa Web, la pregunta de sí usar páginas JSP o *servlets* surge repentinamente. El modelo de programación J2EE promociona la tecnología JSP como el servicio de programación preferido dentro del contenedor Web. Páginas JSP cuentan con la funcionalidad del *servlet* pero el modelo de programación J2EE toma la posición de que las páginas JSP son formas más naturales para diseños Web. El contenedor Web es optimizado para la creación del contenido dinámico destinado a clientes Web y que el uso de tecnología JSP debería ser visto como la norma mientras el uso de *servlets* probablemente será la excepción.

2.3.2 Escenario Cliente *Stand-Alone*.

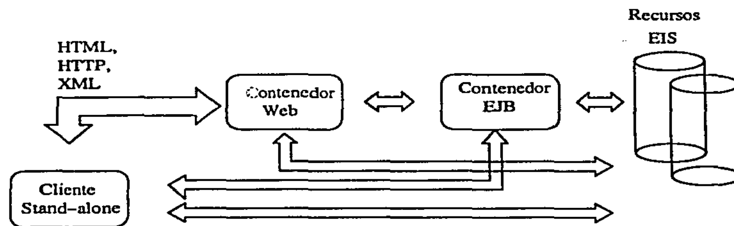

Figura 2.4: Clientes *Stand-Alone*.

La Figura 2.4 ilustra un escenario cliente *stand-alone*.

Desde la perspectiva del modelo de programación J2EE, consideramos tres tipos de clientes *stand-alone*:

- Clientes EJB interactuando directamente con un servidor EJB, esto es, *enterprise beans* instalados sobre un contenedor EJB. Tal escenario es ilustrado en la Figura 2.5. Se asume que RMI-IIOP será usado en este escenario y que el servidor EJB tendrá acceso a los recursos EIS usando JDBC.

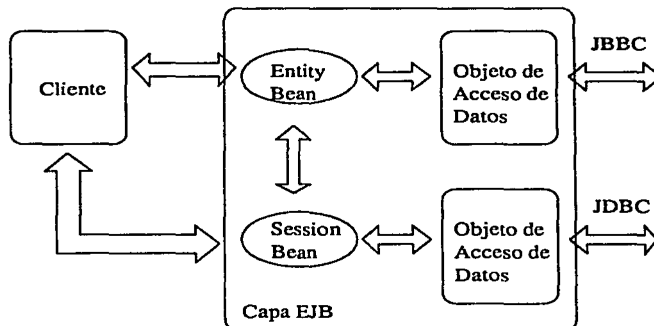

Figura 2.5: Cliente Java Centrado en EJB.

- Clientes de aplicación Java *stand-alone* tienen acceso a los recursos EIS directamente usando JDBC. En este escenario, la presentación y lógica del negocio están ambos por definición localizados sobre la plataforma cliente y pueden de hecho ser firmemente integrados en una aplicación simple. Este escenario colapsa la capa intermedia en la plataforma cliente y es esencialmente un escenario de aplicación cliente servidor con distribución de aplicación asociada, mantenible y con problemas de escalabilidad.

- Los clientes *Visual Basic* consumen contenido Web dinámico, el uso más frecuente es en forma de mensajes de datos XML. En este escenario, el contenedor Web está esencialmente tomando transformaciones XML y provee de conectividad Web a los clientes. La lógica de presentación se asume que será tomada por la capa del cliente. La capa Web puede ser diseñada para tomar la lógica del negocio y tener acceso directo a los recursos EIS. Idealmente, la lógica del negocio es puesta en el servidor EJB, donde el modelo de componentes puede ser totalmente influenciado.

2.3.3 Escenario de Aplicación Centrado en el Web.

La Figura 2.6 illustra un escenario de aplicación centrado en el Web.

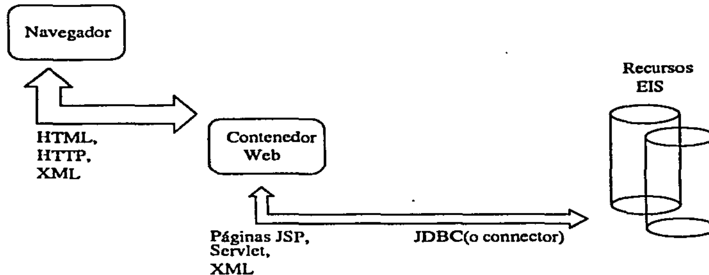

Figura 2.6: Escenario de Aplicación Centrado en el Web.

Hay numerosos ejemplos donde un servidor EJB (al menos inicialmente) podría ser considerado un exceso dado el problema a ser abordado. En esencia, la especificación J2EE no obliga a un modelo de aplicación de 2, 3 capas o multicapa, ni en realidad podría hacerlo. El punto es que es importante usar las herramientas apropiadas para el espacio del problema dado.

El escenario de aplicación centrado en el Web de 3 capas es de uso general. El contenedor de Web es esencialmente donde reside la lógica de negocio y de presentación, y se asume que JDBC será usado para tener acceso a recursos EIS.

La Figura 2.7 proporciona una vista más cercana del contenedor Web en un escenario de aplicación Web.

Es importante tener en mente que el término contenedor de Web es usado aqui en una forma muy precisa. Por ejemplo, si un producto J2EE elige implementar, un servidor J2EE, tal que el contenedor Web y EJB están localizados en el mismo servidor (esto asume que la comunicación entre contenedores está optimizada de alguna forma y que los detalles de implementación son privados), entonces el modelo de programación J2EE trata a la aplicación instalada sobre una plataforma esencialmente como un escenario multicapa.

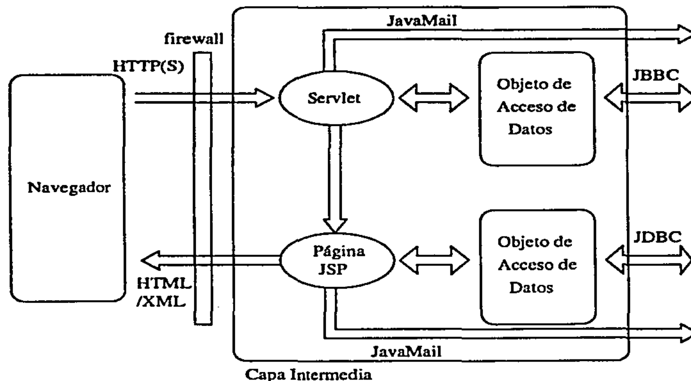

Figura 2.7: Contenedor Web en un escenario de tres capas.

2.3.4 Escenario Negocio a Negocio.

La Figura 2.8 ilustra el escenario negocio a negocio.

Este escenario se enfoca sobre interacciones al mismo nivel entre el Web y los contenedores EJB. El modelo de programación J2EE promueve el uso de mensajería de datos XML sobre HTTP como el medio primario del establecimiento perdido de la comunicación entre los contenedores de Web. Ésta es una forma natural para el desarrollo e instalación de soluciones de comercio basadas en Web.

Las comunicaciones al mismo nivel entre contenedores EJB actualmente son más acopladas a soluciones apropiadas para ambientes de Internet. Con la inminente integración de JMS en la plataforma J2EE, el desarrollo de soluciones de *intranet* llegarán a ser cada vez más prácticas.

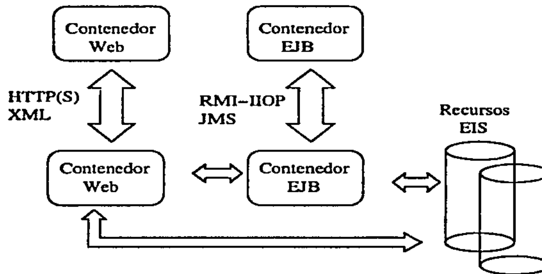

Figura 2.8: Escenario Negocio a Negocio.

2.4 Transacciones.

Las transacciones son un mecanismo para simplificar el desarrollo de aplicaciones empresariales multiusuarios distribuidas. Para hacer cumplir las reglas estrictas sobre la habilidad de la aplicación para tener acceso y actualizar datos, las transacciones aseguran la integridad de los datos. Un sistema transaccional asegura que una unidad de trabajo es ejecutada completamente o todo el trabajo realizado es deshecho. Las transacciones libran al programador de la aplicación de tratar con problemas complejos de recuperación de fallas y programación multiusuario.

2.4.1 Propiedades de las Transacciones.

Todas las transacciones comparten las propiedades de atomicidad, consistencia, aislamiento y durabilidad. Estas propiedades son denotadas por la sigla *ACID*.

La atomicidad requiere que todas las operaciones de la transacción sean realizadas exitosamente para que la transacción sea considerada completa. Si alguna de las operaciones de la transacción no puede ser realizada, entonces nada puede ser aceptado.

La consistencia se refiere a la consistencia de los datos. Una transacción debe asegurar que los datos permanecerán consistentes de un estado a otro. La transacción debe preservar la semántica de éstos y la integridad física.

El aislamiento requiere que cada transacción parezca ser la única transacción actual en la manipulación de los datos. Otras transacciones pueden ejecutarse simultaneamente. Sin embargo, una transacción debería no ver la manipulación de los datos de las otras transacciones inmediatamente sino hasta que éstas sean terminadas exitosamente y hayan realizado su trabajo. Por las interdependencias entre las actualizaciones, una transacción podría obtener una vista consistente de la base de datos donde ésta sólo verá un subconjunto de las otras actualizaciones de las transacciones. El aislamiento protege a la transacción de este tipo de inconsistencia de los datos.

Durabilidad significa que las actualizaciones hechas por transacciones persisten en la base de datos debido a fallas que puedan ocurrir después de que la operación se realizó y ésta también asegura que la base de datos pueda ser recuperada después de que un sistema o medio falló.

2.4.2 Transacciones en la plataforma J2EE.

El soporte para transacciones es un elemento esencial de la arquitectura J2EE. La plataforma J2EE soporta la demarcarción de la transacción de manera declarativa o en programación. El proveedor de componentes puede usar el *Java Transaction API* para delimitar en programación los límites de las transacciones en el código del componente. La delimitación de la transacción declarativa es apoyada por el *enterprise bean*, donde las transacciones son iniciadas y terminadas automáticamente por el contenedor del *bean*. En ambos casos, el peso de la implementación del manejador de transacción está sobre la plataforma J2EE. El servidor J2EE implementa los protocolos de transacción de bajo nivel necesario, las interacciones entre el manejador de transacciones y los sistemas de base de datos JDBC, propagación del contexto de la transacción y opcionalmente *commit* de dos faces distribuido.

La plataforma J2EE soporta una aplicación transaccional que consta de una combinación de *servlets* o páginas JSP que tienen acceso a multiples *enterprise beans* en una sola transacción. Cada componente puede adquirir una o más conexiones para tener acceso a uno o más manejadores de recursos compartidos.

2.4.3 Escenarios.

Acceso a Bases de Datos Multiples.

En la Figura 2.9 un cliente invoca a un *enterprise bean* X. El *bean* X tiene acceso a la base de datos A usando una conexión JDBC. Entonces el *bean* X llama a otro

enterprise bean Y. Y tiene acceso a la base de datos B. El servidor J2EE y los adaptadores de recursos de ambos sistemas de bases de datos aseguran que ambas bases de datos realicen su trabajo completamente o todo el trabajo hasta el momento realizado es deshechado y ningún cambio es registrado en las bases de datos.

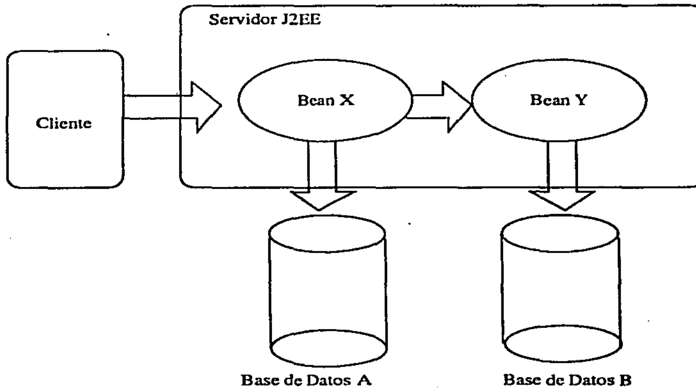

Figura 2.9: Teniendo acceso a Bases de Datos Multiples en la misma Transacción.

Un *Application Component Provider* no tiene que escribir código extra para asegurar la semántica de la transacción. El *enterprise bean* X y Y tiene acceso al sistema de base de datos usando el API de acceso de clientes JDBC. Detrás de la escena, el servidor J2EE alista las conexiones para ambos sistemas como parte de la transacción. Cuando la transacción es aceptada, el servidor J2EE y los manejadores de recursos realizan un protocolo *commit* de dos fases para asegurar la atomicidad de la actualización en los dos sistemas.

Teniendo acceso a EIS desde Multiples Servidores EJB.

En la Figura 2.10 un cliente invoca el *enterprise bean* X, él cual actualiza datos en el EIS A y entonces llama a otro *enterprise bean* Y que está instalado en otro servidor J2EE. El *bean* Y realiza accesos de lectura y escritura sobre el EIS B.

Cuando X invoca a Y, los dos servidores J2EE cooperan para propagar el contexto de transacción de X a Y. Esta propagación de contexto de la transacción es transaparente

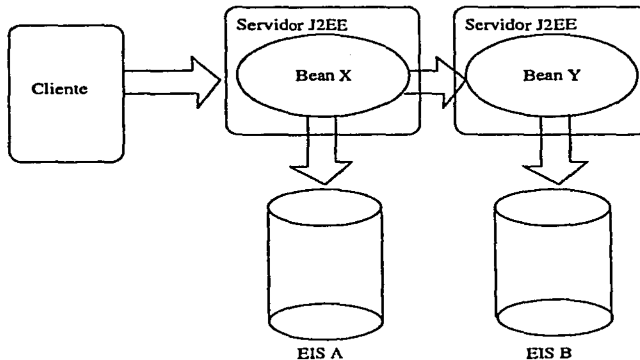

Figura 2.10: Teniendo acceso a Multiples EIS en la misma Transacción.

para el código de la aplicación. En tiempo de *commit* de la transacción, los dos servidores J2EE usan un protocolo *commit* de dos fases que asegura que dos EIS son actualizados bajo una sola transacción.

2.5 Seguridad.

En un ambiente de cómputo empresarial, las fallas, carencias o aquello que compromete la disponiblidad de recursos de cómputo pueden poner en peligro la viabilidad de la empresa. Una organización debe tomar medidas para identificar amenazas de seguridad. Una vez que son identificadas, las medidas deberían ser puestas en práctica para reducirlas.

El modelo de programación de aplicación J2EE aisla a los desarrolladores de detalles de implementación específica sobre la seguridad de la aplicación. J2EE provee este aislamiento de manera que tiene efecto complementario de mejorar la portabilidad de las aplicaciones en una forma que las aplicaciones puedan ser instaladas en diversos ambientes de seguridad.

La plataforma J2EE define contratos declarativos entre los que desarrollan y los que configuran aplicaciones en ambientes operacionales. En el contexto de seguridad de

aplicación, los proveedores de aplicación definen en forma declarativa los requerimientos de seguridad de sus aplicaciones de una forma que puedan ser satisfechas. Los requerimientos de seguridad de una aplicación son expresados en sintaxis declarativa en un documento llamado *deployment descriptor*. Un desarrollador de aplicación emplea herramientas específicas del contenedor para mapear los requerimientos capturados en el *descriptor* sobre los mecanismos de seguridad que son implementados por los contenedores J2EE.

En muchos casos, contenedores J2EE pueden proveer de la funcionalidad de seguridad de aplicaciones completamente fuera de la implementación de la aplicación. En otros casos, la funcionalidad de la seguridad está implementada en la programación de la aplicación. El contrato declarativo que acompaña a una aplicación debe comunicar los requerimientos de seguridad de la aplicación. incluyendo la identificación donde es necesario ligar la funcionalidad de seguridad a mecanismos o valores de ambiente específicos.

Los mecanismos de seguridad J2EE combinan los conceptos de *hosting* de contenedor y la especificación declarativa de los requerimientos de seguridad con la disponibilidad de mecanismos embebidos de aplicación. Ésto provee de un modelo poderoso para cómputo de componentes seguros, portables y distribuidos.

2.6 Haciendo Modulos.

Los componentes son empaquetados por separado y envueltos en una aplicación J2EE para su instalación. Cada componente, sus archivos relacionados como archivos HTML, GIF o clases auxiliares del lado del servidor y un *deployment descriptor* (DD), son ensamblados en un módulo y agregado a la aplicación J2EE. Una aplicación J2EE está compuesta de uno o más módulos de componentes *enterprise bean*, Web o cliente de aplicación. La solución empresarial final puede usar una aplicación J2EE o estar compuesta de dos o más aplicaciones J2EE dependiendo de los requerimientos de diseño.

Una aplicación J2EE y cada uno de sus módulos tiene su propio DD. Un DD es un archivo basado en texto XML con una extensión "xml" que describe la configuración de instalación del componente. El DD de un módulo *enterprise bean*, por ejemplo declara atributos de transacción y autorizaciones de seguridad para el *bean*. La información del DD es declarativa y puede cambiar sin modificar el código fuente del *bean*. En tiempo de ejecución, el servidor J2EE lee el DD y actua en el componente de acuerdo a éste.

Una aplicación J2EE, con todos sus módulos, es entregada en un *Enterprise ARchive*[4] (EAR). Un archivo EAR es un archivo JAR con extensión "ear".

- Cada archivo EJB JAR contiene su DD, archivos relacionados y los archivos con extensión "class" para el *enterprise bean*.
- Cada archivo JAR del cliente de la aplicación contiene su DD, archivos relacionados y los archivos con extensión "class" para el cliente de la aplicación.
- Cada archivo WAR contiene su DD, archivos relacionados y los archivos con extensión "class" para el servlet o con extensión "jsp" para una página JSP.

El uso de módulos y archivos EAR hace posible ensamblar diferentes aplicaciones J2EE usando algunos de los mismos componentes. No se necesita código extra, es solo cuestión de ensamblar varios módulos J2EE en archivos EAR.

[4]Archivo Empresarial

Capítulo 3

Enterprise JavaBeans

Enterprise JavaBeans™ (EJB) y J2EE incorporan conceptos importantes como cómputo distribuido, bases de datos, seguridad, software dirigido a componentes y más. Combinando estas tecnologías se consiguió un gran avance para la comunidad Java. La tecnología EJB es una parte integral de J2EE.

EJB es una arquitectura de componentes del lado del servidor que permite y simplifica el proceso de construcción de aplicaciones de objetos distribuidos de tipo empresarial.

La especificación EJB 1.1 define una arquitectura para el desarrollo e instalación de aplicaciones basadas en objetos distribuidos y transaccionales, componentes de software del lado del servidor. Las organizaciones pueden construir sus propios componentes o comprarlos a vendedores. Estos componentes del lado del servidor, llamados *enterprise beans*, son objetos distribuidos que son instalados en contendores EJB y proveen de servicios remotos para clientes distribuidos a través de la red.

3.1 Tecnología Enterprise JavaBeans.

La especificación EJB define una arquitecura para sistemas de objetos distribuidos y transaccionales basada en componentes. La especificación obliga a un modelo de programación, esto es, convenciones o protocolos y un conjunto de clases e interfaces que forman el API EJB. El modelo de programación EJB provee a los desarrolladores de *beans* y a los vendedores de servidores EJB con un conjunto de contratos que define una plataforma común para desarrollo. La meta de estos contratos es asegurar la portabilidad a través de vendedores mientras soporta un conjunto vasto de funcionalidad.

3.1.1 Contenedor EJB.

Enterprise beans son componentes de software que se ejecutan en un ambiente especial llamado contenedor EJB. El contenedor los hospeda y maneja. Un *bean* no puede funcionar fuera del contenedor EJB. El contenedor EJB maneja cada aspecto del *bean* en tiempo de ejecución incluyendo el acceso remoto al *bean*, seguridad, persistencia, transacción, concurrencia, acceso y *pool* de recursos.

El contenedor aisla al *enterprise bean* del acceso directo de aplicaciones cliente. Cuando una aplicación invoca un método remoto del *bean*, el contenedor primero intercepta la invocación para asegurar que la persistencia, transacción y seguridad son aplicadas de forma correcta a cada operación que el cliente realiza sobre el *bean*. El contenedor maneja seguridad, transacciones y persistencia automáticamente para el *bean*, así no se tenga que escribir este tipo de lógica en el código del *bean*. El desarrollador del *bean* se enfoca sobre la encapsulación de las reglas de negocio, mientras el contenedor cuida todo lo demás.

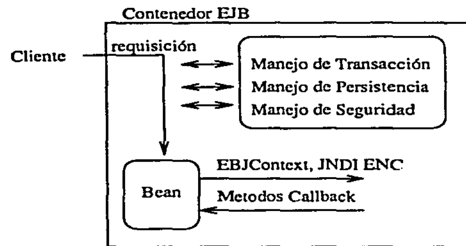

Figura 3.1: Contenedores EJB manejan a los *enterprise beans* en tiempo de ejecución.

Los contenedores manejan muchos *beans* simultáneamente. Para reducir el consumo de memoria y procesamiento, los contendores mandan a los *beans* al *pool* de recursos y manejan el ciclo de vida de todos los *beans* muy cuidadosamente. Cuando un *bean* no sea usado, el contendor lo pondrá en un *pool* para ser reusado por otro cliente o posiblemente lo expulsa de su memoria y solo lo traerá de regreso cuando lo necesite. Como las aplicaciones cliente no tienen acceso directo al *bean* – el contenedor queda entre el cliente y el *bean* – la aplicación cliente no se ocupa de las actividades de manejo de recursos del contenedor. El *bean* que no está en uso, por ejemplo, podría ser expulsado de la memoria del servidor, mientras su referencia remota sobre el cliente permanece intacta. Cuando el cliente invoca un método sobre la referencia remota, el contenedor simplemente reestablece al *bean* para servir la requisición. La

aplicación no se ocupa del proceso completo.

Un *enterprise bean* depende del contenedor para todas sus necesidades. Si él necesita tener acceso a una conexión JDBC u otro *enterprise bean*, lo hace a través del contenedor; si un *bean* necesita tener acceso a la identidad del que lo llamo, obtiene una referencia a sí mismo o tiene acceso a las propiedades a través del contenedor. El *bean* interactúa con su contenedor a través de uno de los mecanismos: métodos *callback*, la interfaz EJBContext o JNDI.

- Métodos *Callback*.
  Cada *bean* implementa un subtipo de la interfaz *EnterpriseBean* que define a los métodos, llamados métodos *callback*. Cada método alerta al *bean* de diferentes eventos en su ciclo de vida y el contenedor invocará estos métodos para notificar al *bean* de cuando activarse, guardar su estado en la base de datos, eliminar al *bean* de memoria, etcétera. Los métodos *callback* permiten al *bean* realizar trabajo inmediatamente antes o después de algún evento.

- *EJBContext*.
  Cada *bean* contiene un objeto *EJBContext* el cual es una referencia directa del contenedor. La interfaz *EJBContext* provee de métodos para interactuar con el contenedor para que el *bean* pueda requerir información acerca de su ambiente como la identidad de su cliente, el estatus de la transacción o para obtener referencias remotas de sí mismo.

- JNDI.
  Java Naming and Directory Interface (JNDI) es una extensión estándar de la plataforma Java para tener acceso a sistemas de nombres como LDAP, *NetWare*, sistemas de archivos, etcétera. Cada *bean* automáticamente tiene acceso a un sistema de nombres especial llamado *Environment Naming Context* (ENC). El ENC es manejado por el contenedor y los *beans* tienen acceso al ENC usando JNDI. El JNDI ENC permite al *bean* tener acceso a recursos como conexiones JDBC, otros *enterprise beans* y propiedades específicas del *bean*.

La especificación EJB define un contrato *bean*-contenedor que incluye los mecanismos descritos anteriormente así como un conjunto estricto de reglas que describen como los *enterprise beans* y sus contenedores se comportarán en tiempo de ejecución, como el acceso de seguridad es verificado, como las transacciones son manejadas, como la persistencia es aplicada, etc. El contrato está diseñado para hacer a los *beans* portables entre los contenedores EJB. Vendedores como BEA, IBM y *GemStone* venden servidores de aplicación que incluyen contenedores EJB. Idealmente, cualquier *bean* que se apegue a la especificación debería ejecutarse en cualquier contenedor EJB.

La portabilidad es central para el valor que produce el EJB. Ésta asegura que un *bean* desarrollado para un contenedor puede ser migrado a otro si este otro ofrece mejor desempeño o características. La portabilidad permite a las organizaciones y desarrolladores mejores oportunidades de elección.

Además de la portabilidad, la simplicidad del modelo de programación EJB le da más valor. Que el modelo de programación sea simple significa que el *bean* pueda ser desarrollado más rapido sin requerir de mucha experiencia en objetos distribuidos, transacciones y otros sistemas empresariales. EJB brinda procesamiento de la transacción y desarrollo de objetos distribuidos en la línea central.

3.1.2 Enterprise Beans.

Para crear un componente EJB del lado del servidor, se necesita proporcionar dos interfaces que definen los métodos de negocio del *bean*, más la clase de implementación del *bean*. El cliente usa una de las interfaces pública del *bean* para crear, manipular y eliminar *beans* del servidor EJB. La clase implementación, a ser llamada la clase *bean*, es instanciada en tiempo de ejecución y llega a ser un objeto distribuido.

Los *enterprise beans* viven en un contenedor EJB y las aplicaciones cliente tienen acceso a los *beans* sobre la red a través de sus interfaces *remote* y *home*. Un *bean* es un componente que representa conceptos de negocio como un Producto o ControlWeb.

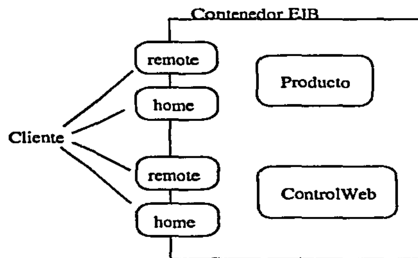

Figura 3.2: Vista Conceptual de la Arquitectura EJB.

Interfaces *Remote* y *Home*.

Las interfaces *home* y *remote* representan al *bean*, pero el contenedor lo aisla del acceso directo de aplicaciones cliente. Cada vez que el *bean* es requerido, creado o eliminado, el contenedor maneja el proceso completo.

La interfaz *home* representa los métodos del ciclo de vida del componente mientras que la interfaz *remote* representa los métodos de negocio del *bean*. Las interfaces *remote* y *home* heredan de las interfaces *javax.ejb.EJBObject* y *javax.ejb.EJBHome* respectivamente. Estos tipos de interfaces EJB definen un conjunto estándar de métodos útiles y proporcionan tipos base para todas las interfaces *remote* y *home*.

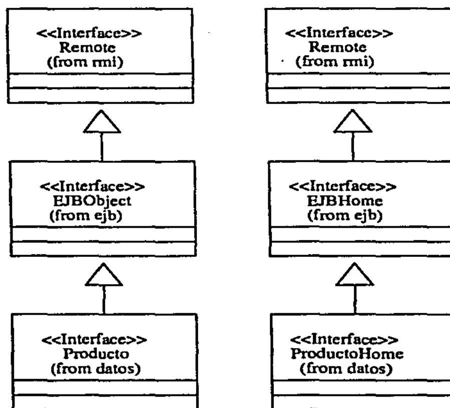

Figura 3.3: Diagrama de Clases de las interfaces *Remote* y *Home*.

Los clientes usan la interfaz *home* del *bean* para obtener referencia a la interfaz *remote* del *bean*. La interfaz *remote* define métodos de negocio como los métodos de acceso y cambio para modificar el título de un Producto o métodos de negocio que realicen tareas como usar al *bean* ControlWeb para registrar la compra de un cliente desde el sitio Web. A continuación hay un ejemplo de como una aplicación cliente puede tener acceso al *bean* Producto. En este caso la interfaz *home* es ProductoHome y la interfaz *remote* es el tipo Producto.

```
ProductoHome home =  // ... obtiene una referencia que
                     // implementa la interfaz home

// Usa la interfaz home para crear una
// instancia nueva del bean Producto.
Producto producto = home.create(codigoDeBarras, titulo, genero,
                                portada, precio);

// usando un metodo de negocio sobre el Producto.
producto.setPrecio(precio);
```

La interfaz *remote* define los métodos de negocio del *bean*; los métodos que son específicos al concepto de negocio que representa. Las interfaces *remote* heredan de la interfaz *javax.ejb.EJBObject* que a su vez hereda de la interfaz *java.rmi.Remote.* A continuación se muestra la definición de la interfaz *remote* para el *bean* Producto.

```
...
public interfaz Producto extends EJBObject {
    public String getCodigoDeBarras()throws RemoteException;
    public String getTitulo()throws RemoteException;
    public void setTitulo(String titulo)throws RemoteException;
    public String getGenero()throws RemoteException;
    public void setGenero(String genero)throws RemoteException;
    public String getPortada()throws RemoteException;
    public void setPortada(String portada)throws RemoteException;
    public float getPrecio()throws RemoteException;
    public void setPrecio(float precio)throws RemoteException;
    public void actualiza(String[] campos)throws RemoteException;
    public Vector daDatos()throws RemoteException;
}
```

La interfaz *remote* define métodos de acceso y cambio para lectura y actualizacion de información acerca del concepto del negocio. Esto es típico del tipo de *bean* llamado *entity bean* que representa un objeto de negocio persistente; objetos de negocio que son almacenados en una base de datos. El *entity bean* representa datos de negocio en la base de datos y agregan conducta específica a los datos.

Métodos de Negocio.

Los métodos de negocio pueden ser representados como tareas que realiza un *bean*. Aunque los *entity beans* a menudo tienen métodos orientados a tareas, las tareas

son más típicas de un tipo llamado *session bean*; éstos no representan datos como los *entity beans*, más bien representan procesos de negocio o agentes que realizan un servicio, como realizar una venta en Web. A continuación se muestra la definición de la interfaz *remote* para ControlWeb de tipo *session bean*.

```
...
public interfaz ControlWeb extends EJBObject {
    public Object[][] daEstrenos(int tipoDeProducto)
        throws Exception,RemoteException;
    public Object[] daGenerosDelProducto(int tipoDeProducto)
        throws Exception,RemoteException;
    public Object[][] daEstrenosDelGenero(int tipoDeProducto,String genero)
        throws Exception,RemoteException;
    public String[] daDetalle(String codigoDeBarras)
        throws Exception,RemoteException;
    public Object[][] buscaProductos(int tipoDeProducto, int indice,
                                String criterio)
        throws Exception,RemoteException;
    public Object[][] daTipoDeProducto()
        throws Exception,RemoteException;
    public int registraCompra(String nombre, String apellido,
        String direccion, String colonia, String codigoPostal,
        String ciudad, String estado, String telefono,
        boolean credito, String tarjetahabiente, int numeroTarjeta,
        String tipo, String fecha, String [][] productos,
        float total)
        throws Exception,RemoteException;
}
```

Los métodos de negocio definidos en la interfaz *remote* ControlWeb representan procesos más que métodos de acceso simple. El bean ControlWeb actua como agente en el sentido de que realiza tareas en nombre del usuario, pero no es persistente por sí mismo en la base de datos.

Hay dos tipos básicos de *enterprise bean*: *entity bean* que representa datos en la base de datos y *session bean* que representa procesos o actua como agente para la realización de tareas.

Entity Beans.

Para cada interfaz *remote* hay una clase de implementación; un objeto de negocio que implementa los métodos de negocio definidos en la interfaz. Ésta es la clase *bean*; el elemento principal del *bean*. A continuación está la definición parcial de la clase *bean* Producto.

```
...
public class ProductoBean implements EntityBean {
    public String codigoDeBarras;
    public String titulo;
    public String genero;
    public String portada;
    public float precio;

    protected EntityContext contexto;

    public String getCodigoDeBarras(){  //Regresa el codigo
        return codigoDeBarras;  //de barras del producto.
    }

    public void setCodigoDeBarras(String codigoDeBarras){
        //Establece el codigo de barras del producto.
        this.codigoDeBarras = codigoDeBarras;
    }

    ...
}
```

ProductoBean es la clase implementación. Toma el dato y provee de métodos de acceso y otros métodos de negocio. Como un *entity bean*, ProductoBean proporciona una vista de objeto del dato Producto. En vez de escribir lógica de acceso a la base de datos, la aplicación puede simplemente usar la interfaz *remote* del *bean* Producto para tener acceso al dato del producto. *Entity bean* implementa el tipo *javax.ejb.EntityBean* que define un conjunto de métodos de notificacion que usa para interactuar con su contenedor.

Session Beans.

El *bean* ControlWeb es un *session bean* que es similar en muchos aspectos al *entity bean*. Los *session beans* representan un conjunto de procesos o tareas que son rea-

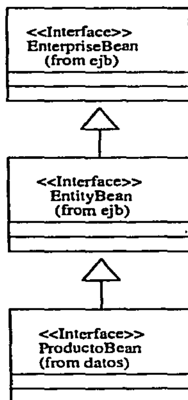

Figura 3.4: Diagrama de clases para la clase *bean*.

lizadas en nombre de la aplicación cliente. Los *session beans* pueden usar otros *beans* para realizar una tarea o tener acceso a la base de datos directamente.

La clase *bean* no implementa las interfaces *remote* o *home*. EJB no requiere que la clase *bean* implemente estas interfaces; de hecho esto no es aconsejable porque los tipos base de las interfaces *remote* y *home* (*EJBObject* y *EJBHome*) definen otros métodos que son implementados por el contenedor automáticamente. La clase *bean*, sin embargo, proporciona las implementaciones para todos los métodos de negocio definidos en la interfaz *remote* además de otros métodos.

Métodos del Ciclo de Vida.

Además de la interfaz *remote*, todos los *beans* tienen una interfaz *home*. Esta interfaz provee de métodos de ciclo de vida para la creación, eliminación y localización de *beans*. Esta conducta de ciclo de vida son separadas fuera de la interfaz *remote* que representa conducta que no es específica a una instancia de *bean*. A continuación está la definición de la interfaz para el *bean* Producto. Éste hereda de la *interfaz javax.ejb.EJBHome* que a su vez hereda de la interfaz *java.rmi.Remote*.

```
...
public interfaz ProductoHome extends EJBHome{
    public Producto create (String codigoDeBarras,
        String titulo, String genero,
        String portada, float precio)
        throws CreateException,RemoteException;

    public Producto findByPrimaryKey (String codigoDeBarras)
        throws FinderException,RemoteException;
}
```

El método *create()* se usa para crear una nueva entidad. Esto resultará en un nuevo registro en la base de datos. El *home* puede tener muchos métodos *create()* y el tipo de regreso de todos debe ser del tipo de dato de la interfaz *remote*. En este caso, la invocación *create()* sobre la interfaz ProductoHome regresará una instancia de tipo Producto. El método *findByPrimaryKey()* es usado para localizar una instancia específica del *bean* Producto. Se pueden definir tantos métodos *finds* como se necesite.

Resumiendo, las interfaces *remote* y *home* son usadas por aplicaciones para tener acceso a *enterprise beans* en tiempo de ejecución. La interfaz *home* permite a las aplicaciones crear o localizar *beans*, mientras la interfaz *remote* permite invocar métodos de negocio. A continuación un fragmento de código que lo ilustra:

```
ProductoHome home =
//Obtiene la referencia hacia el objeto ProductoHome

Producto producto =
  home.create("1234", "Latex for Linux",
              "Computacion", "latex.jpg",
              125.50);
producto.setPrecio(132.00);

Producto producto2 =
  home.findByPrimaryKey("1234");

String titulo = producto2. getTitulo();
//La salida es "Latex for Linux"
System.out.println(titulo);
```

La interfaz *javax.ejb.EJBHome* también define otros métodos que ProductoBean automáticamente hereda, incluyendo el conjunto de métodos *remove()* que permite a las aplicaciones destruir instancias *bean*.

3.1.3 *Enterprise Beans* como Objetos Distribuidos.

Las interfaces *remote* y *home* son interfaces del tipo *Java RMI Remote*. La interfaz java.rmi.Remote es usada por objetos distribuidos para representar al *bean* en un espacio de direcciones diferentes (proceso o máquina). Un *enterprise bean* es un objeto distribuido. Lo que significa que la clase *bean* es instanciada y vive en el contenedor pero aplicaciones que viven en otros espacios de direcciones pueden tener acceso al *bean*.

Para hacer una instancia de objeto en un espacio de dirección disponible a otra requiere un pequeño engaño que involucra *sockets* de red. Para hacer el trabajo, se envuelve la instancia en un objeto especial llamado *skeleton* que tiene una conexión de red a otro objeto especial llamado *stub*. El *stub* implementa la interfaz *remote* así ésta se ve como un objeto de negocio. Como el *stub* no contiene lógica de negocio; toma una conexión *socket* de red al *skeleton*. Cada vez que un método de negocio es invocado sobre la interfaz remota del *stub*, éste manda un mensaje de red hacia el *skeleton* diciendole que método fue invocado. Cuando el *skeleton* recibe un mensaje de red del *stub*, identifica el método invocado y los argumentos, entonces invoca el método correspondiente sobre la instancia actual. La instancia ejecuta el método de negocio y regresa el resultado al *skeleton* que lo manda al *stub*.

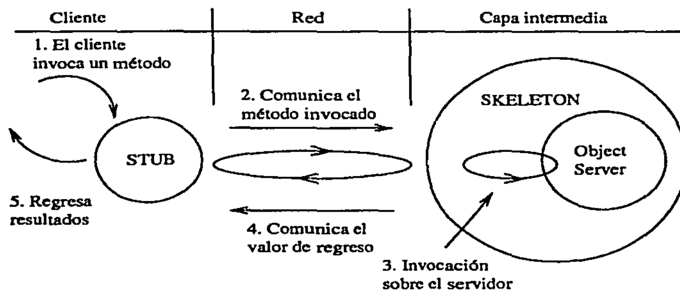

Figura 3.5: Pasos en la invocación de métodos.

El *stub* regresa el resultado a la aplicación que invocó su método de interfaz *remote*. Desde la perspectiva de aplicación usando *stub*, esto se ve como si el *stub* hiciera el trabajo localmente. Actualmente, el *stub* es sólo un objeto de red vago que manda la requisición a través de la red hacia el *skeleton* que invoca el método sobre la instacia actual. La instancia hace todo el trabajo, el *stub* y *skeleton* sólo pasan la identidad

del método y argumentos de ida y regreso a través de la red.

En EJB, el *skeleton* para las interfaces *remote* y *home* son implementadas por el contenedor, no por la clase *bean*. Esto es para asegurar que cada método invocado sobre éstos referencian tipos por una aplicación cliente son primero tomados por el contenedor y entonces delegados a la instancia *bean*. El contenedor debe interceptar estas requisiciones destinadas al *bean* con el fin de que este pueda aplicar persistencia (*entity beans*), transacción y control de acceso automático.

Los protocolos de objetos distribuidos definen el formato de mensajes de red mandados entre los espacios de direcciones. Estos protocolos son tomados automáticamente. Muchos servidores EJB soportan *Java Remote Method Protocol* (JRMP) o el Protocolo de CORBA *Internet Inter-ORB Protocol* (IIOP).

La especificación EJB requiere que se use una versión especializada de Java RMI API, cuando trabaja con un *bean* remotamente. Java RMI es un API para tener acceso a objetos distribuidos y es un tanto un protocolo agnóstico. Así, un servidor EJB puede soportar JRMP o IIOP. El servidor EJB tiene la opción de soportar IIOP, una versión especializada de Java RMI, llamada Java RMI-IIOP. Java RMI-IIOP usa IIOP como el protocolo y el Java RMI API. Los servidores EJB no tienen que usar IIOP, pero tienen restricciones respecto a Java RMI-IIOP, así EJB usa las convenciones y tipos especializados de Java RMI-IIOP, pero el protocolo puede ser cualquiera.

3.2 Enterprise bean de tipo Entity.

El *entity bean* es uno de los dos tipos de *bean* primario. El *entity bean* es usado para representar datos en la base de datos. Proporciona una interfaz orientada a objetos hacia los datos que deberían normalmente tener acceso vía JDBC o algún otro API en el *back-end*. Los *entity beans* proveen de un modelo de componentes que permite enfocarse en la lógica de negocio del *bean*, mientras el contenedor cuida el manejo de persistencia, transacción y control de acceso.

Hay dos tipos de persistencia para el *entity bean*: *Container-Managed Persistence*[1] (CMP) y *Bean-Managed Persistence*[2] (BMP). Con CMP, el contenedor maneja la persistencia del *bean*. Se usan herramientas de vendedores para asociar los campos del *bean* hacia la base de datos y no hay código escrito en la clase *bean* que haga un acceso a la base de datos. Con BMP, el *entity bean* contiene código de acceso a la base de datos (usualmente JDBC) y es responsable para leer y escribir su estado

[1] Persistencia Manejada por el Contenedor
[2] Persistencia Manejada por el Bean

a la base de datos. BMP *entities* tienen ayuda del contenedor pues éste alertará al *bean* cuando es necesario hacer una actualización o leer su estado de la base de datos. El contenedor puede tomar algún bloqueo o transacción, para que la base de datos mantenga la integridad.

3.2.1 Container-Managed Persistence.

CMP *beans* son más simples de crear y más difíciles de soportar para el servidor EJB. Esto es porque toda la lógica de sincronización del estado del *bean* con la base de datos es tomada automáticamente por el contenedor. Esto significa que no se necesita escribir alguna lógica de acceso, mientras el servidor EJB es supuesto para cuidar toda las necesidades de persistencia automáticamente. Muchos vendedores EJB soportan persistencia automática para la base de datos relacional, pero el nivel de soporte varía. Algunos proveen de mapeo sofisticado Objeto-Relacional, mientras otros son muy limitados.

Se explicará el desarrollo de ProductoBean que maneja CMP.

Clase Bean.

Un *enterprise bean* es un componente completo que está formado de al menos dos interfaces y una clase implementacion *bean*. A continuación se muestra un bosquejo de la clase *bean* ProductoBean, el cual puede consultarse en el Apéndice E.

```
...
public class ProductoBean implements EntityBean {
    public String codigoDeBarras;
    public float precio;
    // Los demas atributos son titulo, genero y portada de tipo String
    protected EntityContext contexto;

    public String getCodigoDeBarras(){ // Regresa el codigo
        return codigoDeBarras; // de barras del producto.
    }

    public void setCodigoDeBarras(String codigoDeBarras){
        this.codigoDeBarras = codigoDeBarras;
    }
    // Similarmente se tienen metodos para los atributos titulo,
    // genero, portada y precio
```

```
    ...

    public String ejbCreate (String codigoDeBarras, String titulo,
        String genero, String portada, float precio)
        throws CreateException {
        ...
    }

    public void ejbPostCreate(String codigoDeBarras, String titulo,
        String genero, String portada, float precio){}

    public void ejbRemove() {}
    // Otros metodos callback
    ...

    public void actualiza(String[] campos){
        //Actualiza los campos del producto.
        ...
    }

    public Vector daDatos(){
        //Regresa los datos del producto.
        ...
    }
}
```

No hay lógica de acceso de base de datos en el *bean*. La clase ProductoBean, por ejemplo, podría ser asociada fácilmenta a cualquier base de datos que contenga datos similares a los campos del *bean*. En este caso los campos de instancia del bean son *String* y *float*.

Estos campos son llamados *container-managed fields* (CMFs)[3] porque el contenedor es responsable de la sincronización de su estado con la base de datos; el contenedor maneja los campos. Los CMFs pueden ser tipos de datos primitivos o cualquier tipo serializable. En este caso utilizamos ambos un tipo primitivo *float* y objetos serializables de tipo *String*. El orden para asociar los objetos dependientes a la base de datos podrían necesitar herramientas de asociación bastante sofisticadas. No todos los campos en un *bean* son CMFs; algunos sólo pueden ser campos de instancia planos para uso transitorio del *bean*. Se distinguen los CMFs de campos de instancia planos indicando que campos son manejados por el contenedor en el *deployment descriptor* (DD).

[3]Campos Manejados por el Contenedor

Los CMFs deben tener tipos correspondientes en la base de datos (columnas en RDBMS) o directamente a través de la asociación Objeto-Relacional. ProductoBean, por ejemplo, asocia a la tabla Producto en la base de datos que podría tener la siguiente definición:

```
CREATE TABLE PRODUCTO {
codigoDeBarras    CHAR(20) PRIMARY KEY,
titulo            CHAR(50),
genero            CHAR(50),
portada           CHAR(30),
precioFLOAT
}
```

Con CMP, el vendedor debe tener algún tipo de herramienta propietaria que asocie los CMFs del *bean* a las columnas correspondientes en una tabla específica, PRODUCTO en este caso.

Una vez que los campos de *bean* son asociados a una tabla en la base de datos y el *bean* Producto es instalado, el contenedor manejará la creación, carga, actualización y eliminación de registros de la tabla PRODUCTO en respuesta a métodos invocados en las interfaces *remote* y *home* del *bean* Producto.

Un subconjunto (uno o más) de los CMFs serán identificados como la llave primaria del *bean*. La llave primaria es el índice o apuntador a un registro único en la base de datos que forma el estado del *bean*. En el caso de ProductoBean, el campo codigoDeBarras es la llave primaria y sera usada para localizar datos *beans* en la base de datos. Llaves primarias de tipo primitivo son representados por sus objetos *wrappers* correspondientes. Las llaves primarias que están formadas de muchos campos, llamadas llaves primarias compuestas, serán representadas por una clase especial. Las llaves primarias son similares en concepto a llaves primarias en una base de datos relacional.

Interfaz *Home*.

Para crear una nueva instancia de CMP *entity bean* e insertar datos en la base de datos, se debe invocar el método *create()* de la interfaz *home* del *bean*. La interfaz *home* del *bean* Producto está definida por la interfaz ProductoHome. A continuación se muestra la definición de ProductoHome.

```
...
public interfaz ProductoHome extends EJBHome{
    public Producto create (String codigoDeBarras,
        String titulo, String genero,
        String portada, float precio)
        throws CreateException,RemoteException;

    public Producto findByPrimaryKey (String codigoDeBarras)
        throws FinderException,RemoteException;
}
```

Éste es un ejemplo de como una interfaz *home* podría ser usada por una aplicación cliente para crear un nuevo Producto.

```
ProductoHome home =
//Obtiene la referencia hacia el objeto ProductoHome

Producto producto =
  home.create("1234","Latex for Linux",
              "Computacion","latex.jpg",
              125.50);
```

La interfaz *home* del *bean* puede declarar cero o más métodos *create()*, cada uno de ellos debe tener el método *ejbCreate()* correspondiente en la clase *bean*. Estos métodos de creación son ligados en tiempo de ejecución.

Cuando el método *create()* sobre una interfaz *home* es invocado, el contenedor delega la llamada del método hacia la instancia de *bean* que concuerda con el método *ejbCreate()*. Los métodos *ejbCreate()* son usados para inicializar el estado de la instancia antes que un registro sea insertado en la base de datos. Cuando el método *ejbCreate()* es terminado el contenedor lee los CMFs e inserta un nuevo registro en la tabla PRODUCTO indexado por la llave primaria, en este caso codigoDeBarras asocia a la columna PRODUCTO.CODIGODEBARRAS.

En EJB, un *entity bean* técnicamente no existe hasta después que sus datos han sido insertados en la base de datos que ocurre durante la llamada del método *ejbCreate()*. Una vez que el dato ha sido insertado, el *entity bean* existe y puede tener acceso a su propia llave primaria y referencias remotas, lo cual es posible hasta después de que el método *ejbCreate()* termina y los datos están en la base de datos. El método *ejbPostCreate()* permite al *bean* hacer cualquier procesamiento después de la creación antes que inicie a servir requisiciones del cliente. Para cada *ejbCreate()* debe haber un método *ejbPostCreate()*.

Los métodos en la interfaz *home* que inicia con "*find*" son llamados métodos *find*. Éstos son usados para consultar el servidor EJB para *entity beans* específicos, basados sobre el nombre de los métodos y argumentos. No hay un lenguaje de consulta estándar definido para estos métodos, así cada vendedor los implementará de forma diferente. En CMP *entity beans*, los métodos *find* no son implementados con métodos en la clase *bean*; los contenedores los implementán cuando el *bean* es instalado de manera específica por el vendedor. Algunos vendedores usan herramientas de mapeo Objeto-Relacional para definir la conducta del método mientras otros simplemente requieren del comando SQL apropiado.

Hay dos tipos básicos de métodos *find*: de una y multiples entidades. Los métodos de una entidad regresan una referencia remota a un *entity bean* específico que concuerda con la requisición del *find*. Si ningún *entity bean* es hallado, el método lanza un *ObjectNotFoundException*. Cada *bean* debe definir el método *findByPrimaryKey()* de una entidad que toma al argumento como el tipo de la llave primaria del *bean*. Los métodos *find* de multiples entidades regresan una colección (de tipo *Enumeration* o *Collection*) de entidades que concuerdan con la requisición *find*. Si ningún *entity bean* es hallado, el método regresa una colección vacía.

Interfaz Remote.

Cada *entity bean* debe definir una interfaz *remote* además de la interfaz *home*. La interfaz *remote* define los métodos de negocio del *entity bean*. Ésta es la definición de la interfaz *remote* para el *bean* Producto:

```
...
public interfaz Producto extends EJBObject {
    public String getCodigoDeBarras()throws RemoteException;
    public String getTitulo()throws RemoteException;
    public void setTitulo(String titulo)throws RemoteException;
    public String getGenero()throws RemoteException;
    public void setGenero(String genero)throws RemoteException;
    public String getPortada()throws RemoteException;
    public void setPortada(String portada)throws RemoteException;
    public float getPrecio()throws RemoteException;
    public void setPrecio(float precio)throws RemoteException;
    public void actualiza(String[] campos)throws RemoteException;
    public Vector daDatos()throws RemoteException;
}
```

A continuación se da un ejemplo de como una aplicación cliente podría usar la interfaz *remote* para interactuar con el *bean*:

```
Producto producto =
// obtiene una referencia remota al bean

// actualiza el precio del producto
producto.setPrecio(254.20);
```

Los métodos de negocio en la interfaz *remote* son delegados hacia los métodos de negocio que concuerden en la instancia *bean*. En el *bean* Producto, los métodos de negocio son simples métodos de acceso y cambio, pero podrían ser más complicados. En otras palabras, los métodos de negocio del *entity* no están limitados a lectura y escritura de datos, pueden realizar tareas y cálculos.

Métodos Callback.

La clase bean define métodos *create* que concuerdan con métodos en la interfaz *home* y métodos de negocio que concuerdan con métodos en la interfaz *remote*. La clase *bean* también implementa un conjunto de métodos *callback* que permiten al contenedor notificar al *bean* de eventos en su ciclo de vida. Los métodos callback son definidos en la interfaz *javax.ejb.EntityBean* que es implementada por todos los *entity beans*. La interfaz EntityBean tiene la siguiente definición. La clase *bean* implementa estos métodos.

```
public interfaz javax.ejb.EntityBean {
    public void setEntityContext();
    public void unsetEntityContext();
    public void ejbLoad();
    public void ejbStore();
    public void ejbActivate();
    public void ejbPassivate();
    public void ejbRemove();
}
```

El método *setEntityContext()* proporciona al *bean* una interfaz hacia el contenedor llamado *EntityContext*. La interfaz *EntityContext* contiene métodos para obtener información del contexto bajo el cual el *bean* es operado en algún momento particular. *EntityContext* es usada para tener acceso a información de seguridad acerca del que hace la llamada; para determinar el estatus de la transacción actual o para forzar a hacer un *rollback* a la transacción; o para obtener una referencia al mismo *bean*, su

home o su llave primaria. *EntityContext* sólo se establece una vez en la vida de la instancia del *entity bean*, así su referencia debería ser puesta en uno de los campos de instancia del *bean* por si se necesitará después.

El método *unsetEntityContext()* es usado al final del ciclo de vida del *bean* – antes de que la instancia sea sacada de memoria – para ya no hacer referencia al *EntityContext* y realizar limpieza de último minuto.

Los métodos *ejbLoad()* y *ejbStore()* en CMP *entities* son invocados cuando el estado del *entity bean* es un ser sincronizado con la base de datos. *ejbLoad()* es invocado justo después de que el contenedor ha actualizado los CMFs con su estado de la base de datos. El método *ejbStore()* es invocado justo antes de que el contenedor escriba los CMFs a la base de datos. Estos métodos son usados para modificar datos como un ser sincronizado. Esto es común cuando el dato almacenado en la base de datos es diferente al datos usado en los campos del bean. Los métodos podrían ser usados, por ejemplo, para comprimir datos antes de que sean almacenados y descomprimirlos cuando sean recuperados de la base de datos.

Los métodos *ejbPassivate()* y *ejbActivate()* son invocados sobre el *bean* por el contenedor justo antes de que el *bean* pase a un estado pasivo y justo después de que el *bean* es activado, respectivamente. Que el *entity bean* este en estado pasivo significa que la instancia de *bean* es desasociada con su referencia remota ası el contenedor pueda sacarlo de memoria o reusarlo. Es una medida para conservar recursos que el contenedor emplea para reducir el número de instancias en memoria. Un *bean* podría estar en estado pasivo si no se ha usado durante algún tiempo o como una operación normal realizada por el contenedor para maximizar el reuso de recursos. Algunos contenedores eliminarán *beans* de memoria, mientras otros reusaran instancias para otras referencias remotas más activas. Los métodos *ejbPassivate()* y *ejbActivate()* notifican al *bean* cuanto está en estado pasivo (desasociado con la refencia remota) o es activado (asociado con una referencia remota).

3.2.2 Bean-Managed Persistence.

BMP *enterprise bean* maneja la sincronización de su estado con la base de datos como el dirigido por el contendor. El *bean* usa un API de base de datos (usualmente JDBC) para leer y escribir sus campos a la base de datos, pero el contenedor dice cuando hacer cada operación de sincronización y maneja las transacciones para el *bean* automáticamente. BMP permite realizar operaciones de persistencia que son bastantes complicadas para el contenedor o usar recursos de datos que no son soportados por el contenedor – bases de datos legadas por ejemplo.

Se modificará la clase ProductoBean a un BMP *bean.* Esta modificación no impacta las interfaces *remote* y *home.* De hecho, no se modificará ProductoBean directamente. En vez de eso, se cambiará a BMP extendiendo el *bean* y sobreescribiendo los métodos apropiados. A continuación está la definición de la clase que hereda del *bean* Producto para hacerlo BMP *entity.* En muchos casos, no se extiende un *bean* para hacerlo BMP, sólo se implementará al *bean* como BMP directamente. Esta estrategia (de extender al CMP *bean*) es hecha por dos razones: permite al *bean* ser CMP o BMP y consume menos cantidad de código a desplegar.

```
public class ProductoBean_BMP extends ProductoBean {
    public void ejbLoad() {
        // sobreescribe la implementacin
    }
    public void ejbStore() {
        // sobreescribe la implementacion
    }
    public void ejbCreate() {
        // sobreescribe la implementacion
    }
    public void ejbRemove() {
        // sobreescribe la implementacion
    }
    private Connection getConnection() {
        // nuevo metodo auxiliar
    }
}
```

Con BMP *beans,* los métodos *ejbLoad()* y *ejbStore()* son usados en forma diferente por el contendor y *bean* que con CMP. En BMP, estos métodos contienen código para leer los datos del *bean* de la base de datos y escribir los cambios a la base de datos. Los métodos son llamados sobre el *bean* cuando el servidor EJB decide que es tiempo de leer o escribir datos.

El *bean* ProductoBean_BMP maneja su propia persistencia. En otras palabras, los métodos *ejbLoad()* y *ejbStore()* deben incluir lógica de acceso a la base de datos así el *bean* pueda cargar y almacenar sus datos cuando el contenedor EJB decide hacerlo. El contenedor ejecutará los métodos automáticamente cuando sea apropiado.

El método *ejbLoad()* es usualmente invocado por el contenedor al inicio de la transacción, justo antes de que el contenedor lo delegue a un método de negocio del *bean.* A continuación el código muestra como implementar el método usando JDBC.

```
public class ProductoBean_BMP extends ProductoBean {
    public void ejbLoad() {
        Connection con;
        try {
            String primaryKey =
                (String)ejbContext.getPrimaryKey();
            con = this.getConnection();
            Statement stmt =
                con.createStatement("SELECT * FROM Producto " +
                                    " WHERE codigoDeBarras = '" +
                                    primaryKey + "'");
            ResultSet rs = stmt.executeQuery();
            if (rs.next()) {
                codigoDeBarras = primaryKey;
                titulo = rs.getString("TITULO");
                genero = rs.getString("GENERO");
                portada = rs.getString("PORTADA");
                precio = rs.getFloat("PRECIO");
            }
        } catch (SQLException sqle) {
            throw new EJBException(sqle);
        } finally {
            if (con!=null)
                con.close();
        }
    }
}
```

El método *ejbLoad()* usa la referencia *EntityContext* del *bean* para obtener la llave primaria de la instancia. Esto asegura que usa el índice correcto de la base de datos. Obviamente, ProductoBean_BMP necesitará usar los métodos heredados *setEntityContext()* y *unsetEntityContext()*.

El método *ejbStore()* es invocado por el contenedor sobre el *bean* y al final de la transacción, sólo antes de que el contenedor intente hacer permanentes todos los cambios a la base de datos.

```
public class ProductoBean_BMP extends ProductoBean {
    public void ejbLoad() {
        ... lee datos de la base de datos
    }

    public void ejbStore() {
        Connection con;
```

```
        try {
            String primaryKey =
                (String)ejbContext.getPrimaryKey();
            con = this.getConnection();
            PreparedStatement sqlPrep =
                con.prepareStatement(
                    "UPDATE producto set " +
                    "titulo = ?, genero = ?, portada = ?, " +
                    "precio = ?" +
                    "WHERE codigoDeBarras = ?"
                );
            sqlPrep.setString(1,titulo);
            sqlPrep.setString(2,genero);
            sqlPrep.setString(3,portada);
            sqlPrep.setFloat(4,precio);
            sqlPrep.setString(5,codigoDeBarras);
            sqlPrep.executeUpdate();
        } ...
    }
}
```

En ambos métodos el *bean* sincroniza su propio estado con la base de datos usando JDBC. El método *getConnection()* no es un método estándar EJB, éste es un método privado auxiliar implementado para ocultar el mecanismo de obtención de la conexión de base de datos. Ésta es la definición del método.

```
public class ProductoBean_BMP extends ProductoBean {
    public void ejbLoad() {
        ... lee datos de la base de datos
    }

    public void ejbStore() {
        ... escribe datos a  la base de datos
    }

    private Connection getConnection()
        throws SQLException {
        InitialContext jndiContext =
            new InitialContext();
        DataSource source = (DataSource)
            jndiContext.lookup("java:comp/env/jdbc/OracleDB");
        return source.getConnection();
    }
}
```

Conexiones a la base de datos son obtenidas desde el contenedor usando un contexto default JNDI ENC. ENC provee de acceso a conexiones JDBC transaccionales.

EN BMP, los métodos *ejbLoad()* y *ejbStore()* son invocados por el contenedor para sincronizar la instancia *bean* con los datos de la base de datos. Para insertar y eliminar registros de la base de datos, los métodos *ejbCreate()* y *ejbRemove()* son implementados con lógica de acceso a la base de datos similar. Los métodos *ejbCreate()* son implementados para que un nuevo registro sea insertado en la base de datos y los métodos *ejbRemove()* son implementados para que los datos de la entidad sean eliminados de la base de datos. Los métodos *ejbCreate()* y *ejbRemove()* de un BMP *entity* son invocados por el contenedor en respuesta a invocaciones sobre sus métodos correspondientes. Estos métodos se muestran a continuación.

```
public String ejbCreate( String codigoDeBarras,
    String titulo, String genero,
    String portada, float precio){
    Connection con;
    try {
        con = this.getConnection();
        Statement stmt =
            con.createStatement("INSERT INTO producto VALUES ('" +
                                codigoDeBarras + "','" + titulo +
                                "','" + genero + "','" + portada +
                                "'," + precio +")");
        stmt.executeUpdate();
        return codigoDeBarras;
    } ...
}

public void ejbRemove() {
    String primaryKey =
        (String)ejbContext.getPrimaryKey();
    Connection con;
    try {
        con = this.getConnection();
        Statement stmt =
            con.createStatement("DELETE FROM producto" +
                                "WHERE codigoDeBarras = '" +
                                primaryKey + "'");
        stmt.executeUpdate();
    } ...
}
```

En BMP, la clase *bean* es responsable de implementar los métodos *finds* definidos en la interfaz *home*. Para cada método *find* definido en la interfaz *home* debe haber un método *ejbFind()* en la clase *bean*. Los métodos *ejbFind()* localizan los registros apropiados en la base de datos y regresan su llave primaria al contenedor. El contenedor convierte las llaves primarias en referencias del *bean* y las regresa al cliente. A continuación se presenta el ejemplo de implementación del método *ejbFindByPrimaryKey()* en la clase ProductoBean_BMP que corresponde al método *findByPrimaryKey()* definido en la interfaz ProductoHome.

```
public String ejbFindByPrimaryKey(String primaryKey)
    throws ObjectNotFoundException {
    Connection con;
    try {
        con = this.getConnection();
        Statement stmt =
            con.createStatement("SELECT * FROM Producto " +
                                " WHERE codigoDeBarras = '" +
                                primaryKey + "'");
        ResultSet rs = stmt.executeQuery();
        if (rs.next())
            return primaryKey;
        else
            throw ObjectNotFoundException();
    } ...
}
```

Los métodos *find* de una entidad regresan una llave primaria simple o lanza un *ObjectNotFoundException* si ningún registro es localizado. Métodos *finds* multientidad regresan una colección de llaves primarias. El contenedor convierte esta colección en una colección de referencias remotas que son regresadas al cliente.

Si ningún registro es encontrado, se regresa una colección vacía al contenedor y éste a su vez regresa una colección vacía al cliente.

Con la implementación de todos estos métodos y un poco de cambios menores al DD del *bean*, el ProductoBean_BMP está listo para ser instalado como BMP *entity*.

3.3 Enterprise Beans Tipo Session.

Los *session beans* son usados para manejar las interacciones de un *entity bean* y otro de *session* y generalmente realizan tareas a nombre del cliente. Los *session beans* no

son objetos de negocio persistentes como los *entity beans.*

Hay dos tipos básicos de *session bean*: *Stateless* y *Stateful*. *Stateless session beans* están formados de métodos de negocio que se comportan como procedimientos; operan sólo sobre los argumentos pasados cuando son invocados. *Stateless beans* son llamados "*stateless*" porque son transitorios; no mantienen estado de negocio entre invocaciones de métodos. *Stateful session beans* encapsulan la lógica del negocio y estado específico a un cliente. Son llamados "*stateful*" porque mantienen el estado de negocio entre invocaciones, lo sostienen en memoria y no son persistentes.

3.3.1 Stateless Session Beans.

Stateless session beans, como todos los *session beans*, no son objetos de negocio persistentes. Representan procesos de negocio o tareas que son realizadas a nombre del cliente. Cada invocación de métodos de negocio *stateless* es independiente de las invocaciones anteriores. Los *stateless session bean* son más fáciles de manejar por el contenedor, tienden a procesar requicisiones más rápido y usan menos recursos. Estos *beans* no recuerdan nada de una invocación de método a la próxima invocación.

Un ejemplo de *stateless session bean* es el *bean* ControlWeb, representa los servicios para recuperar productos y realizar compras en el sitio Web. A continuación se muestra las interfaces *remote* y *home* del *bean* ControlWeb.

```
public interfaz ControlWeb extends EJBObject { //interfaz remote
    public Object[][] daEstrenos(int tipoDeProducto)
        throws Exception,RemoteException;
    public Object[] daGenerosDelProducto(int tipoDeProducto)
        throws Exception,RemoteException;
    public Object[][] daEstrenosDelGenero(int tipoDeProducto, String genero)
        throws Exception,RemoteException;
    public String[] daDetalle(String codigoDeBarras)
        throws Exception,RemoteException;
    public Object[][] buscaProductos(int tipoDeProducto, int indice,
        String criterio) throws Exception,RemoteException;
    public Object[][] daTipoDeProducto() throws Exception,RemoteException;
    public int registraCompra(String nombre, String apellido,
        String direccion, String colonia, String codigoPostal,
        String ciudad, String estado, String telefono,boolean credito,
        String tarjetahabiente, int numeroTarjeta,String tipo,
        String fecha, String [][] productos, float total)
        throws Exception,RemoteException;
}
```

```
//interfaz home
public interfaz ControlWebHome extends EJBHome {
    ControlWeb create() throws RemoteException, CreateException;
}
```

La interfaz *remote*, define todos los métodos de negocio. La interfaz *home*, ControlWebHome provee de un método *create()* sin argumentos. Todas las interfaces *home* para *stateless session beans* definirán un sólo método *create()* sin argumentos, los *session beans* no tienen métodos *find* y no pueden ser incializados con algún argumentos cuando son creados. Los *stateless session beans* no tienen métodos *find* porque los *stateless beans* son todos equivalentes y no son persistentes. En otras palabras, no hay *stateless beans* únicos que puedan ser cargados de la base de datos. Como los *stateless beans* no son persistentes, éstos sólo son servicios transitorios. Cada cliente que usa el mismo tipo de *session bean* obtiene el mismo servicio.

A continuación se encuentra un bosquejo para el *bean* ControlWebBean, el cual puede consultarse en el Apéndice F.

```
public class ControlWebBean implements SessionBean{
    private Connection con; //Conexion de la base de datos.

    public void ejbCreate() { //Establece la conexion
        ...
    }

    public void ejbRemove() { //Cierra la conexion
        ....
    }

    public void ejbActivate() {}

    public void ejbPassivate() {}

    public void setSessionContext(SessionContext sc) {}

    private void estableceConexion() //Establece la conexion
        throws NamingException, SQLException { //con la base de datos.
        ...
    }

    public Object[][] daEstrenos(int tipoDeProducto)
    throws Exception {
        //Se realiza una busqueda de estrenos de acuerdo al tipo de producto
```

```
    //para obtener las referencias remotas de estos productos y
    //procesar la informacion
    ...
}

public Object[] daGenerosDelProducto(int tipoDeProducto)
    throws Exception{
    //Se arma una consulta de acuerdo al tipo de producto para
    //obtener los generos
    ...
}

public Object[][] daEstrenosDelGenero(int tipoDeProducto,
    String genero)
    throws Exception{
    //Se realiza una busqueda sobre el tipo de producto y
    //el genero para obtener las referencias remotas de
    //estos productos y procesar la informacion
    ...
}

public String[] daDetalle(String codigoDeBarras)
    throws Exception{
    //Regresa la toda la informacion de un producto de
    //acuerdo al codigo de barras
    ...
}

public Object[][] buscaProductos(int tipoDeProducto, int indice,
    String criterio) throws Exception{
    //Se realiza una busqueda por criterio para obtener las
    //referencias remotas de estos productos y procesar
    //la informacion
    ...
}

public Object[][] daTipoDeProducto() throws Exception{
    //Regresa los tipos de producto que se manejan en el Sitio.
    ...
}

public int registraCompra(String nombre, String apellido,
    String direccion, String colonia, String codigoPostal,
    String ciudad, String estado, String telefono,
    boolean credito, String tarjetahabiente,
```

```
        int numeroTarjeta,String tipo, String fecha,
        String [][] productos,float total) throws Exception{
        //Se guardan los datos del cliente y si es una tarjeta de
        //credito se guardan los datos de la tarjeta
        //Se guardan los elementos del carrito y se genera el No. de pedido
        ...
    }
}
```

Stateless beans podrían ser usados para tener acceso a recursos inusuales de base de datos o realizar cómputos complejos. El *bean* ControlWeb es usado por clientes en vez de conexiones directas así los servicios pueden ser mejor manejados por el contenedor EJB que hará *pool* con las conexiones y manejo de transacciones automáticamente para el cliente EJB.

El método *ejbCreate()* es invocado al inicio de su ciclo de vida y sólo una vez. En este método es conveniente inicializar conexiones de recursos y variables que serán usadas por el *stateless bean* para su ciclo de vida.

El método *ejbRemove()* es invocado sólo una vez en el su ciclo de vida por el contenedor; cuando el bean sea destruido.

Invocaciones de los métodos *create()* y *remove()* sobre sus interfaces *home* y *remote* por el cliete no resultan en invocaciones sobre métodos *ejbCreate()* y *ejbRemove()* sobre la instancia *bean*. En vez de esto, una invocación del método *create()* provee al cliente con una referencia al tipo *stateless bean* y el método *remove()* invalida la referencia. Los contenedores decidirán cuando las instancias de *bean* son creadas y destruidas e invocarán los métodos *create()* y *remove()*. Esto permite a instancias *stateless* ser compartidas entre muchos clientes sin impactar las referencias de los clientes.

El método *setSessionContext()* provee a la instancia *bean* con una referencia hacia el *SessionContext* que tiene el mismo propósito que EntityContext para ProductoBean.

Los métodos *ejbActivate()* y *ejbPassivate()* no son implementados porque no son usados en *stateless session beans* ya que no pueden estar en un estado pasivo. Estos métodos son definidos en la interfaz *javax.ejb.SessionBean* para los stateful *session beans* y se provee una implementación vacía para *stateless session beans*.

Stateless session beans tambien se usan para tener acceso a bases de datos así como para coordinar la interacción con otro *bean* para realizar una tarea.

En la especificación EJB, RMI sobre IIOP es el modelo de programación especificado, así tipos referenciados por CORBA deben ser soportados. Referencias CORBA

no pueden ser cambiadas usando *Java native casting*. En vez de eso, el método *PortableRemoteObject.narrow()* debe ser usado para cambiar una referencia de un tipo a un subtipo. JNDI siempre regresa un tipo Object, todas las referencias de *bean* deberían explicitamente cambiar al subtipo para soportar la portabilidad entre contenedores.

3.3.2 Stateful Session Beans.

Stateful session beans son dedicados a un cliente para mantener un estado conversacional entre invocaciones de métodos. Cuando un cliente crea un *stateful bean*, la instancia *bean* es dedicada a servir solo a ese cliente. Esto hace posible mantener el estado conversacional que es el estado de negocio que puede ser compartido por métodos en el mismo *stateful bean*.

Para conservar recursos, *stateful session beans* pueden estar en un estado pasivo cuando no son usados por el cliente. El estado pasivo en un *stateful session beans* es diferente que para los *entity beans*. En *stateful beans*, el estado pasivo significa que el estado conversacional del *bean* es escrito a un medio de almacenaje secundario (frecuentemente disco) y la instancia es eliminada de memoria. La referencia del cliente hacia el *bean* no es afectada por el estado pasivo, permanece viva y usable mientras el *bean* está en estado pasivo. Cuando el cliente invoca un método sobre un *bean* que se encuentra en estado pasivo, el contenedor activará al *bean* por instanciar una nueva instancia y restaura su estado conversacional con el estado escrito del medio de almacenaje. Este proceso pasivo/activo es frecuentemente terminado usando serialización de Java pero puede ser implementado en otras formas propietarias con tal de que el mecanismo se comporte igual que la serialización normal (una excepción de esto es que los campos transitorios no necesitan ser puesto a sus valores iniciales default cuando el *bean* es activado).

Stateful session beans usan los métodos *ejbActivate()* y *ejbPassivate()*. El contenedor invocará estos métodos para notificar al *bean* cuando vaya a estar en estado pasivo (*ejbPassivate()*) e inmediatamente siguiendo la activación (*ejbActivate()*). Estos métodos se deberían usar para cerrar recursos abiertos y para hacer otro tipo de limpieza antes de que el estado de la instancia sea escrito al medio de almacenaje y sacado de memoria.

El método *ejbRemove()* es invocado sobre la instancia *stateful* cuando el cliente invoca el método *remove* sobre la interfaz *home* o *remote*. El *bean* debería usar el método *ejbRemove()* para hacer algún tipo de limpieza.

3.4 Transacciones.

En el Capítulo 2 se vieron las propiedades de las transacciones. Los contenedores proveen a los EJB *beans* de estas propiedades.

EJB proporciona un manejo de transacción de forma declarativa a través del *deployment descriptor*. El desarrollador del *bean* especifica el tipo de transacción. Cuando el *bean* es instalado, el contenedor lee el *deployment descriptor* asociado a éste y automáticamente provee el soporte de transacción necesario. EJB proporciona de seis tipos transacciones:

- TX_NOT_SUPPORTED
- TX_BEAN_MANAGED
- TX_REQUIRED
- TX_SUPPORT
- TX_REQUIRES_NEW
- TX_MANDATORY

TX_NOT_SUPPORTED significa que el *bean* no soporta transacciones y no puede ser usado dentro de una transacción. Si el cliente intenta usar el *bean* dentro de una transacción, la transacción del cliente permanecerá suspendida hasta que el *bean* termine su operación y entonces la transacción del cliente continua.

TX_BEAN_MANAGED, el *bean* maneja su propia transacción. Esta opción es el único tipo de transacción en la cual el *bean* puede directamente manipular la transacción.

Para el modo TX_REQUIRED, si el cliente tiene una transacción abierta cuando invoca el método del *bean*, el *bean* se ejecuta dentro del contexto de la transacción del cliente. De otro modo el contenedor inicia una nueva transacción.

TX_SUPPORT significa que si el cliente tiene una transacción en progreso cuando invoca al método del *bean*, la transacción del cliente es usada. De otro modo, no se crea una nueva transacción.

En TX_REQUIRES_NEW el *bean* requiere que el cliente siempre inicie una nueva transacción, incluso si una transacción está en progreso. Si una transacción esta en camino, el contenedor la suspende temporalmente hasta que la transacción del *bean* finalice.

Para TX_MANDATORY, el *bean* requiere que el cliente tenga una transacción abierta antes de que el cliente intente usar al *bean*. Si ninguna transacción está disponible ocurre un error.

Teniendo en mente que las transacciones EJB no son anidadas. Esto es, si un *bean* tiene TX_REQUIRES_NEW como un atributo de transacción, recibe una llamada con un contexto de transacción abierto y suspende la transacción, la transacción suspendida será completamente olvidada para la nueva transacción. Además, despues que la nueva transacción ha sido aceptada, esta permanecera en ese estado incluso si la transacción anterior falla y es deshecha.

3.5 Instalando Enterprise JavaBeans

El contenedor toma la persistencia, transacciones, concurrencia y control de acceso automático para los *enterprise beans*. La especificación EJB describe un mecanismo declarativo de como estas cosas serán tomadas, a través del uso de un archivo de tipo XML llamado *Deployment Descriptor* (DD). Cuando un *bean* es instalado en un contenedor, el contenedor lee el DD para averiguar como la transacción, persistencia (*entity beans*), control de acceso deberían ser tomados.

El DD tiene un formato predefinido que todos los *beans* EJB deben usar y todos los servidores EJB deben saber como leer. Este formato es especificado en un XML *Document Type Definition* (DTD). El DD describe el tipo del *bean* (*session* o *entity*) y las clases usadas para las interfaces *remote* y *home* y la clase *bean*. También especifica los atributos transaccionales de cada método en el *bean*, que roles de seguridad puede tener acceso a cada método (control de acceso), y si la persistencia en el *entity bean* es tomada automáticamente o es realizada por el *bean*. A continuación se presenta un ejemplo de un DD usado para describir al *bean* Producto:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<ejb-jar>
  <display-name>Productos</display-name>
  <enterprise-beans>
    <entity>
      <ejb-name>ProductoBean</ejb-name>
      <home>datos.ProductoHome</home>
      <remote>datos.Producto</remote>
      <ejb-class>datos.ProductoBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-field>
        <field-name>titulo</field-name>
      </cmp-field>
```

```
      <cmp-field>
        <field-name>codigoDeBarras</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>precio</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>genero</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>portada</field-name>
      </cmp-field>
        <primkey-field>codigoDeBarras</primkey-field>
    </entity>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>ProductoBean</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Los servidores de aplicación EJB usualmente proveen de herramientas que pueden ser usadas para construir los DDs, esto simplifica el proceso.

Para que un *bean* sea instalado se necesita empaquetar sus archivos *remote*, *home*, la clase *bean* y el DD asociado en un archivo JAR. El archivo DD debe ser almacenado en el JAR bajo el nombre especial META-INF/ejb-jar.xml. Este archivo JAR, llamado ejb-jar, es neutral al vendedor; puede ser instalado en cualquier contenedor EJB que soporte la especificación EJB. Cuando un *bean* es instalado en un contenedor EJB su DD es leido del JAR para determinar como manejar el *bean* en tiempo de ejecución. Se deben asociar los atributos del DD al ambiente del contenedor. Esto incluía la asociación de seguridad hacia el sistema de seguridad del ambiente, agregando al *bean* hacia el sistema de nombrado del contenedor EJB. Una vez que se ha terminado de instalar al *bean* es disponible para aplicaciones cliente y otros *beans*.

3.6 Clientes Enterprise JavaBeans

Clientes EJB pueden ser aplicaciones *standalone*, *servlets*, *applets* u otros *enterprise beans.* Todos los clientes usan la interfaz *home* del *server bean* para obtener referencias al *server bean.* Esta referencia tiene el tipo de dato de la interfaz *remote* del *server bean*, además el cliente interactua con el *server bean* a través de los métodos definidos en la interfaz *remote.*

```
InitialContext ic = new InitialContext();
// Obtaine una referencia a ControlWebHome
Object objRef = ic.lookup("ejb/ControlWeb");
// Cambia el objeto regresado por el JNDI hacia el tipo
// adecuado
ControlWebHome home =
    (ControlWebHome)PortableRemoteObject.narrow(objRef,
     ControlWebHome.class);
// Usa la interfaz home para crear una nueva
// instancia del bean ControlWeb
controlWeb = home.create();
// Usa un metodo de negocio en ControlWeb.
controlWeb.setMetodo(algo);
```

Un cliente obtiene primero una referencia a la interfaz *home* usando JNDI ENC para buscar al *server bean.*

Después de que la interfaz *home* es obtenida, el cliente usa los métodos definidos sobre la interfaz *home* para crear, hallar o eliminar al *server bean.* Invocando uno de los métodos *create()* de la interfaz *home* regresa la referencia remota del *bean* que el cliente usa para realizar su trabajo.

Capítulo 4

Uso de J2EE

J2EE es una especificación para crear aplicaciones a nivel empresarial, multiusuarios, portables, seguras, transaccionales y escalables del lado del servidor. Provee de una arquitectura basada en componentes para crear aplicaciones de n-capas distribuidas del lado del servidor.

A continuación se describe una aplicación de comercio electrónico y como se le da solución a través de la tecnología J2EE.

4.1 Descripción de las Necesidades del Sistema de Comercio Electrónico.

Ventas Café es una compañia ficticia que ofrece al público una gran variedad en CDs de música, DVD y Libros. La compañía decidió vender sus productos usando catálogos en línea.

Ventas Café quiere bajar los costos de negocio haciendo ventas directamente al cliente final, a través de un Sitio de comercio electrónico de ventas vía Web. Estas son las características del sistema:

Administración del Sitio. El administrador del Sitio puede agregar, eliminar y modificar los productos que se despliegan en el mismo. El administrador puede marcar los productos que se consideran estrenos y eliminar la marca de los productos que ya no son considerados así. También puede realizar búsquedas sobre los productos.

Catálogo en línea. Los usuarios ven la línea de productos sobre el Web y el detalle de cada producto.

Búsquedas en línea. Los usuarios pueden realizar búsquedas sobre los productos del Sitio.

Generación del carrito de compras en línea. El usuario puede agregar productos en su carrito mientras observa el catálogo de productos. El usuario puede ver el contenido de su carrito y eliminar productos, modificar la cantidad de productos mientras ve su carrito.

Generación de orden. Una vez que el usuario ha terminado la selección de productos y ha decidido, ordenarlos se genera un pedido.

De acuerdo a estas necesidades se detectan dos subsistemas: el subsistema de administración del Sitio Web y el subsistema de ventas vía Web. El subsistema de ventas vía Web utiliza el escenario de aplicación multicapa[1], mientras el subsistema de administración del Sitio Web utiliza el escenario cliente *stand-alone* centrado en EJB.

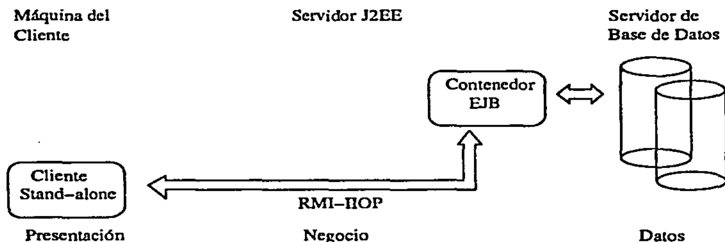

Figura 4.1: Escenario del subsistema de administración del Sitio Web.

4.2 Arquitectura Física del Sistema.

El sistema tiene una arquitectura física de tres capas: la capa de presentación, la capa de lógica de negocio y la capa de datos. La capa de presentación está instalada en la

[1]Material revisado en la Sección 2.3

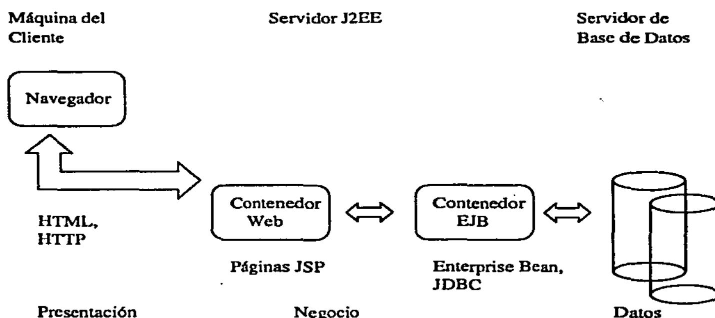

Figura 4.2: Escenario del subsistema de ventas vía Web.

máquina cliente, la capa de lógica de negocio está instalada en la máquina servidor y la capa de datos está instalada en la máquina de base de datos. A continuación se describe cada una de estas capas:

Capa de presentación. Se tiene una capa de presentación distinta para cada subsistema.

La capa de presentación para el subsistema de administración del Sitio Web consiste de una interfaz gráfica de usuario desarrollada con *Swing*. Permite administrar los productos que se muestran en el Sitio Web.

La capa de presentación para el subsistema de ventas vía Web involucra al servidor de *servlets* y páginas JSP *Tomcat*, para la interacción con el usuario final. La capa de presentación despliega la información requerida en HTML para el usuario final; también lee e interpreta las selecciones de usuarios y realiza invocaciones a la capa de negocio. La implementacion de la capa de presentación usa JSPs.

Capa de lógica de negocio. Consiste de multiples EJBs, corriendo en el servidor de aplicaciones JBoss. Los componentes son reusables e independientes de cualquier lógica de interfaz de usuario. La capa de negocio está formada de objetos *entity beans* que representan los datos y *session beans* que proporcionan las reglas del negocio. Los dos subsistemas de la capa de presentación comparten esta capa.

Capa de datos. Es donde se almacena los datos en forma permanente. La base de datos es Oracle 8i , ésta agrega toda la información persistente relacionada con el Sitio de comercio electrónico.

4.3 Clases de la capa de presentación.

La capa de presentación despliega la interfaz gráfica de usuario para los dos clientes finales. Para la capa de presentación del subsistema de ventas vía Web, usa JSP para interactuar con el cliente sobre HTTP. Para la capa de presentación del subsistema de administración del Sitio Web, usa *Swing* sobre RMI-IIOP.

4.3.1 Capa de presentación del subsistema de administración del Sitio Web.

Estos son los componentes principales para la capa de presentación del subsistema de administración del Sitio Web.

Ventana IEntradaAlmacen. IEntradaAlmacen es la ventana principal al sistema. Mantienen la sesión del administrador con el sistema. Permite al administrador manejar los productos del Sitio.

Ventana IAgregarProducto. IAgregarProducto agrega al sistema los productos que se muestran en el Sitio.

Ventana IEliminarProducto. IEliminarProducto elimina del sistema los productos que por alguna razón ya no se exhiben en el Sitio.

Ventana IModificarProducto. IModificarProducto modifica la información de los productos que se muestran en el Sitio.

Ventana IBuscarProducto. IBuscarProducto realiza búsquedas sobre los productos. Así mismo, permite marcar productos que se muestran en el Sitio como estrenos.

Ventana IDesmarcarProducto. IDesmarcarProducto eliminar la marca de estrenos de los productos que están marcados como estrenos.

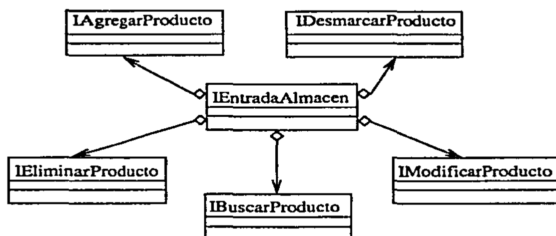

Figura 4.3: Modelo de clases de la capa de presentación del subsistema de administración del Sitio Web.

4.3.2 Capa de presentación del subsistema de ventas vía Web.

Estos son los componentes principales para la capa de presentación del subsistema de ventas vía Web.

JSP PaginaPrincipal. PaginaPrincipal es el primer JSP que el cliente usa en el Sitio Web. Despliega los estrenos de todos los productos. Muestra un conjunto de carpetas con los distintos productos que se manejan en el Sitio.

JSP Carpeta. Carpeta representa las carpetas de los productos que se muestran en el Sitio. Despliega los estrenos por producto. Desde la Carpeta se pueden mandar a realizar búsquedas.

JSP DetalleProducto. DetalleProducto muestra la información detallada de un producto seleccionado por el cliente. La información desplegada es de acuerdo al producto que se seleccionó.

JSP BusquedaDeProductos. BusquedaDeProductos presenta el resultado de la búsqueda que previamente solicitó el cliente.

JSP MuestraCarrito. MuestraCarrito representa el carrito del Cliente. Puede consultar, agregar, eliminar y modificar la cantidad de unidades por producto.

JSP DatosCliente. DatosCliente pide al cliente sus datos generales una vez que el cliente solicita realizar la compra de sus productos.

JSP DatosCompra. DatosCompra despliega los datos del cliente y el contenido del carrito. Pide al cliente su confirmación para continuar la compra.

JSP Compra. Compra genera el pedido y le proporciona al cliente su número de pedido.

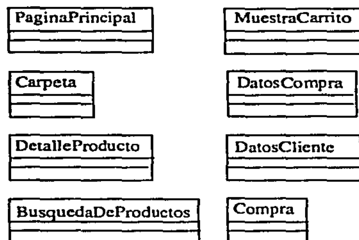

Figura 4.4: Modelo de clases de la capa de presentación del subsistema de ventas vía Web.

4.4 Clases de la capa de lógica de negocio.

La capa de lógica de negocio representa las reglas y los datos de negocio. La capa está compuesta de EJBs del tipo *entity beans* y *session beans.* La persistencia y transaccion es manejada por el contenedor.

4.4.1 Entity Beans.

Estos son los *entity beans* que representan los datos.

Producto. Es una abstracción general de los productos que se despliegan en el Sitio Web, con los siguientes datos:

- Código de barras único del producto.
- Título del producto.
- Género al que pertenece el producto.

- Portada del producto.
- Precio del producto.

CD. Es una especialización de Producto y representa un CD de música con los siguientes datos particulares:

- Disquera a la que pertenece el CD.
- Interprete del CD.

DVD. Es una especialización de Producto y representa un DVD con los siguientes datos particulares:

- Elenco que participa en el DVD.
- Region del DVD.
- clasificacion del DVD.

Libro. Es una especialización de Producto y representa un Libro con los siguientes datos particulares:

- Autor del libro.
- Editorial del libro.

4.4.2 Session Beans.

Estos son los *session beans* que representan las reglas del negocio.

ControlDeProductosSinEstado. Este *bean* se encarga de agregar, marcar y desmarcar productos.

ControlDeProductos. Este *bean* se encarga de eliminar y modificar los productos.

BusquedaDeProductos. Este *bean* se encarga de realizar los diferentes tipos de búsquedas.

ControlWeb. Este *bean* es el encargado de manejar todo lo relacionado con el Sitio Web.

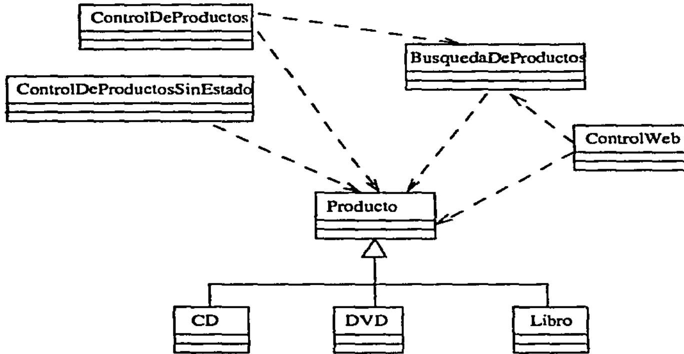

Figura 4.5: Modelo de clases de la capa de lógica de negocio.

4.5 Ventajas y Desventajas.

Estas son las ventajas que se obtuvieron al aplicar la tecnología J2EE al sistema de comercio electrónico:

- **Arquitectura que facilita el mantenimiento.**

  La mantenibilidad se entiende como la capacidad de un producto de software de ser modificado. Las modificaciones pueden incluir correcciones, mejoras o adaptaciones de software a cambios en el ambiente, requerimientos y especificaciones funcionales. Como la aplicación tiene una arquitectura de capas, darle mantenenimiento al sistema es una tarea relativamente sencilla por tener la lógica de aplicación separada e independiente.

  - El sistema tiene una arquitectura física de tres capas. En la capa cliente se despliega la interfaz de usuario. En la capa servidor se ejecuta la lógica de negocio y parte de la lógica de presentación (JSPs), en esta capa reside el servidor de aplicaciones; y la última capa es de datos es ahí donde reside el manejador de base de datos. Esta arquitectura beneficia al sistema porque al ocurrir un problema solo es en una capa específica y este no se propaga a nivel aplicación.

  - El sistema tiene una arquitectura lógica de tres y cuatro capas. En el caso del subsistema de administración del Sitio Web la arquitectura es de tres capas (Swing, EJBs y datos), mientras que el subsistema de ventas vía Web tiene una arquitectura de cuatro capas (Navegador, JSPs, EJBs y datos). Esta arquitectura separar la lógica de negocio de la lógica de presentación. En el contenedor de Web radica la lógica de presentación que se encuentra en el servidor (JSPs) y en el contenedor EJB radica la lógica de negocio. Tener este tipo de arquitectura quita complejidad a la aplicación haciendo el desarrollo más sencillo.

- **Portabilidad con respecto al servidor de aplicaciones.**

  La portabilidad se entiende como la capacidad de un producto de software de ejecutarse sin cambios bajo diferentes servidores de aplicación.

  - El sistema es portable. En la capa de lógica de negocio se siguió la especificación EJB 1.1, la lógica de presentación utiliza Java, JSP y HTML, toda la tecnología empleada en el sistema se apegó a la especificación J2EE que garantiza la portabilidad entre servidores de aplicaciones. La aplicación no está ligada a un servidor en particular y por tanto es independiente del servidor y se puede elegir el que más se adapte a las necesidades del dueño del sistema de comercio electrónico.

- **Distribuido de manera transparente.**

  En un sistema distribuido los componentes de hardware y software están situados en máquinas autónomas y unidos por una red, se comunican y coordinan sus acciones por paso de mensajes.

  Como J2EE proporciona servicios de comunicación, el desarrollador ya no se tiene que preocupar por implementar estos servicios y para él son transparentes, es decir no tiene que implementar *sockets*, servicios de nombrado etcétera. para lograr que el sistema sea distribuido.

  - El sistema es distribuido. El sistema cuenta con una máquina servidor, una máquina de base de datos y diversas máquinas clientes, las máquinas se encuentran en diferentes espacios de dirección. La lógica de negocio se ejecuta en la máquina servidor y la interfaz de usuario se ejecuta en las máquinas clientes, una parte de la lógica de presentación se ejecuta en la máquina servidor (JSPs), es así como se logra que la lógica de aplicación se ejecute en diferentes espacios de direcciones como si todo se ejecutará en una sola máquina. La comunicación se logra con los servicios de comunicación que proporciona J2EE. Al ser la lógica de aplicación distribuida se tiene un mejor desempeño. La mayor parte de la lógica de presentación se ejecuta en la máquina servidor, permitiendo así que en las máquinas

cliente la ejecución de lógica sea mínima y tenga un buen servico en el sitio.

- **Escalable.**

  La escalabilidad se entiende como la capacidad del sistema para adaptarse a un incremento de la carga del servicio, adaptarse a la comunidad de usuarios y permitir la integración de recursos adicionales.

  - El sistema es fácilmente escalable. Al agregar un recurso adicional al sistema para lograr un mejor desempeño, la lógica de aplicación no se ve impactada y los cambios son menores para que siga ejecutándose en forma normal. Si el recurso es otra máquina servidor, los componentes de la lógica de negocio sólo se tienen que distribuir en las máquinas servidor. Si el recurso es otra máquina de base de datos solo se tendrían que hacer cambios mínimos en la lógica de negocio para que conozca al nuevo recurso. En ambos casos la lógica de presentación puede quedar sin cambios y sí llegará a ocurrir sería porque necesita conocer a la otra máquina servidor y sólo basta con indicarle el espacio de dirección del recurso. Los sistemas de comercio electrónico tienden a crecer en recursos por tener un amplio mercado. Estos sistemas deben tener buen desempeño para satisfacer las necesidades del cliente porque al ser sistemas sobre Internet, si al cliente no se le da un buen servicio, éste abandona el sitio sin realizar la compra que es el objetivo principal del sistema.

- **Robusto.**

  Se dice que un sistema es robusto si ha demostrado la habilidad de recuperarse airosamente del rango completo de entradas y situaciones excepcionales en un ambiente dado.

  - El sistema es robusto. Al ser J2EE una especificación, un producto J2EE provee de un conjunto de excepciones bien definidas que se deben manejar durante el proceso de desarrollo permitiendo que la aplicación pueda mandar mensajes generales de los problemas que puedan surgir sin tener que deterner por completo la ejecución de la aplicacion. Esta es una característica fundamental que deben tener los sistemas de comercio electrónico ya que como son dirigidos al cliente final es presindible para que el sistema brinde un buen servicio.

- **Reusabilidad.**

  La reusabilidad es el grado para el cual un módulo de software u otro producto de trabajo puede ser usado en más de un programa de cómputo o sistema de software.

  - La lógica de negocio usa componente reusables independientes de la interfaz de usuario, la capa de presentación de ambos subsistemas utiliza algunos componentes comunes de la lógica de negocio.

- **Facilidad de configuración.**

  La configuración de componentes se realiza a través del *Deployment Descriptor* que permite establecer las características del componente.

  - Los componentes de software del lado del servidor son fácilmente configurables al hacer uso del *Deployment Descriptor* (DD), éste sigue un formato estándar a los servidores de aplicaciones, en el DD se configuran en forma declarativa los elementos de seguridad, manejo de transacción, manejo de persistencia y control de acceso.

- **Disponibibilidad de recursos recurrentes.**

  La plataforma J2EE cuenta con un conjunto de recursos que se usan a menudo en las aplicaciones empresariales. Contar con estos recursos reduce el tiempo de desarrollo.

  - La tecnología J2EE reduce el tiempo de desarrollo ya que proporciona servicio de acceso de base de datos, manejo de transacción, servicios de nombrado y directorio, seguridad, manejo del ciclo de vida de los componentes y servicios de comunicaciones. El desarrollador sólo debe enfocarse en la lógica de negocio y delega los otros aspectos al servidor.

Las desventajas que se identificaron al aplicar la tecnología J2EE al sistema de comercio son:

- **Poca facilidad de manejo de errores por parte del desarrollador.**

  - Al ser J2EE una especificación se tiene que hacer uso de las excepciones que maneja como *RemoteException*, *CreateException*, *RemoveException*, etcétera, pero la excepción no proporciona la información necesaria para mandar un mensaje al usuario indicándole el origen del problema.

- **Alto costo de aprendizaje.**

  - Se requiere personas con cierto grado de conocimiento de J2EE para la instalación, configuración y desarrollo de componentes, administración del servidor de aplicaciones y de base de datos. Se tiene que hacer una inversión en la capacitacion del personal, esta inversión es de tiempo o costo.

ESTA TESIS NO SALE
DE LA BIBLIOTECA

- **Alto costo de infraestructura.**
  - Se requiere de una gran infraestructura de software y hardware para una aplicación desarrollada en J2EE. Los requisitos promedio de equipo de hardware para el servidor de aplicaciones es: un servidor pentium III a 900Mgz con 256Mb en RAM y para tener un buen desempeño se tendría que utilizar otro servidor para la base de datos. Además se requiere de una licencia para el servidor de aplicaciones, para la base de datos, contar con servicios de Internet para el sitio y contratar la empresa que provee el servicio de servidor seguro.

Capítulo 5

Conclusiones

J2EE es una tecnología reciente que permite desarrollar soluciones empresariales rápidamente y con un costo menor. La especificación J2EE 1.2 es liberada a finales de 1999 y los primeros servidores de aplicacion surgen en el año 2000, la especificación de J2EE 1.3 se liberó en agosto del 2001 y todavía hay empresas que trabajan con sus servidores para lograr cubrir esta especificación. Como se puede apreciar J2EE ha estado evolucionando por los buenos resultados que ha obtenido en la empresa.

J2EE define un ambiente que soporta requerimientos de tipo empresarial. Cualquier servidor J2EE provee de soporte para los servicios empresariales críticos. La consistencia de la plataforma proporciona un nivel de confianza a las organizaciones que quieren implementar aplicaciones empresariales usando la tecnología Java.

Como J2EE es independiente del vendedor del servidor de aplicaciones, las organizaciones que requieren desarrollar servicios de comercio electrónico pueden elegir la implementación de la plataforma de una gran variedad de vendedores, éstos pueden diferenciar sus productos de los otros al soportar requerimientos específicos como seguridad rigurosa, escalabilidad extrema, fiabilidad a prueba de fallas, estructuras de datos complejas, transacciones heterogéneas, integración legada y fácil uso. Las organizaciones pueden seleccionar el servidor que mejor se adapte a sus necesidades.

La Internet está obligando a muchas empresas a adoptar un nuevo enfoque al realizar sistemas de comercio electrónico porque ésta tiene cada vez más usuarios que pueden ser compradores potenciales para las empresas. Estos nuevos sistemas requieren de una arquitectura de aplicación distribuida, multicapa basada sobre una infraestructura robusta y escalable para dar un mejor servicio a sus clientes. J2EE provee de una plataforma integrada y sólida que las organizaciones necesitan para abordar al comercio.

J2EE requiere que las empresas cuenten con una gran infraestructura tanto en software como en hardware y otro tipo de recursos, es así como J2EE es una solución dirigida a las grandes y medianas empresas.

Hacer el sistema de comercio electrónico Ventas Café fue difícil en el sentido de que al inicio hubo que obtener los recursos de hardware necesarios para desarrollar la aplicación y encontrar el servidor de aplicaciones y de base de datos que cumpliera con la especificación J2EE.

Por otro lado, tomó mucho tiempo el conocer la tecnología J2EE porque no se contó con ninguna capacitación. El tiempo de aprendizaje fue mucho mayor del previsto porque era una tecnología casi completamente nueva y estos temas no fueron vistos durante mi formacion académica porque fue una tecnología que surgió a finales de 1999 y no es muy conocida en el ambito universitario. El aprendizaje fue autodidacta. Una vez que se asimiló, el desarrollo de la aplicación fue en un tiempo corto. Tomó más tiempo aprender que desarrollar.

Al realizar este sistema de pronto se encontró de que no sólo eran necesarios conocimiento de la tecnología J2EE, sino que se requería saber sobre la administración del servidor de aplicaciones para configurar e instalar los componentes de software. En este caso particular, también hubo que administrar la base de datos, todo este proceso requirió invertir tiempo.

No fue suficiente conocer J2EE sino que se utilizaron otros lenguajes y herramientas en el desarrollo de la aplicación, por ejemplo se utilizó *javascript* en la lógica de presentación del lado del cliente para la validación de campos y se utilizó *Ant*, herramienta de *The Apache Software Foundation*, para hacer el proceso de compilación automático. Podemos concluir que al realizar una aplicación se requiere el conocimiento de más tecnologías de las que se había planeado.

Para trabajos futuros se podrían revisar puntos que no se tomaron en cuenta en este documento:

- Cambiar al sistema de servidor de aplicaciones a un servidor que soporte la especificación J2EE 1.3.
- Migrar los componentes de la capa de lógica de negocio que se encuentran en la especificación EJB 1.1 a la 1.2 haciendo uso de los nuevos conceptos como interfaces locales y persistencia manejada por el contendor 1.2.
- Hacer pruebas de desempeño de la aplicación y, si se requiere, optimizarla.
- Ver y aplicar conceptos relacionados con seguridad en la aplicación.

Bibliografía

[1] Gerhard Schmied
High Quality Messaging and Electronic Commerce: technical fundation, standards, and protocols
Ed. Springer-Verlag, 1999

[2] Raymond Pyle
Electronic Commerce and the Internet
Communications of the ACM, June 1996, Vol. 39, No. 6, p. 23

[3] http://www.careergun.com/WHAT.htm
What is E-commerce?

[4] Anis Bhimani
Securing the Commercial Internet
Communications of the ACM, June 1996, Vol. 39, No. 6, p. 23

[5] Vijay Ahuja
Secure Commerce on the Internet
Ed. AP PROFESSIONAL, 1997

[6] Tom valesky
ENTERPRISE JAVABEANS™, Developing Component-Based Distributed Applications
ADDISON-WESLEY, 1999

[7] Ed Roman
Mastering Enterprise JavaBeans™ and the Java™ 2 Platform, Enterprise Edition
1a. Edición
WYLEY COMPUTER PUBLISHING, 1999

[8] Ed Roman, Scott Ambler, Tyler Jewell
Mastering Enterprise JavaBeans™
2a. Edición
WYLEY COMPUTER PUBLISHING, 2002

[9] Martin Fowler
ANALYSIS PATTERNS
Reusable Objects Models
ADDISON-WESLEY, 1997

[10] http://java.sun.com/

[11] http://www.oracle.com/

[12] http://www.rational.com/

[13] http://www.jboss.org/

[14] http://www.theserverside.com/

[15] http://www.javaworld.com/

[16] http://ute.edu.ec/~mjativa/ce/main.html

[17] http://ciberconta.unizar.es/leccion/eCONTA/100.HTM

[18] http://www.extela.com/comercio/

[19] http://www.spain-lions.net/internet/comercio/comercio2.html

[20] http://www.finmall.com.mx/comelec.htm

[21] http://www.inf.utfsm.cl/ rmonge/sd/apuntes.html

[22] http://di002.edv.uniovi.es/ lourdes/publicaciones/bt99.pdf

[23] http://www.irt.org/

[24] http://home.netscape.com/eng/mozilla/30/handbook/javascript/

Glosario

ACID. Siglas para las cuatro propiedades que debe garantizar una transacción: Atomicidad, Consistencia, aislamiento (del inglés *Isolation*) y Durabilidad.

Aplicación distribuida. Aplicación formada de distintos componentes ejecutándose en ambientes de ejecución separados, usualmente sobre diferentes plataformas conectadas en red. Típicamente las aplicaciones distribuidad son de dos capas (cliente-servidor), tres capas (cliente-capa intermedia-servidor) y multicapa (cliente-multiples capas intermedias-multiples servidores).

Aplicación J2EE. Cualquier unidad instalable con funcionalidad J2EE. Ésta puede ser un módulo simple o un grupo de módulos empaquetados en un archivo .ear con un DD de aplicación J2EE. Las aplicaciones J2EE son típicamentes diseñadas para ser distribuidas a través de multiples capas de computo.

Aplicación JSP. Aplicación de Web *stand-alone* escrita usando tecnología *JavaServer Pages* que puede contener páginas JSP, *servlet*, archivos HTML, imágenes, applets y componentes *JavaBean*.

Aplicación Web. Aplicación escrita para la Internet, incluyendo aquellas construidas con tecnología Java como *JavaServer Pages* y *servlet*, así como CGI y Perl.

API. Ver *Application Program Interface*.

Applet. Componente que típicamente se ejecuta en un navegador Web, pero puede ejecutarse en otras aplicaciones o dispositivos que soporten el modelo de programación del *applet*.

Application Program Interface. Es un conjunto de rutinas, protocolos y herramientas para la construcción de aplicaciones de software.

Application Component Provider. Fabricante que provee de clases Java que implementan métodos de componentes, definiciones de páginas JSP y cualquier DD requerido.

Archivo EAR. Archivo JAR que contiene una aplicación J2EE.

Archivo JAR EJB. Archivo JAR que tiene un módulo EJB.

Archivo JSP. Archivo que contiene una página JSP. El archivo debe tener extensión jsp.

Archivo WAR. Archivo JAR que contiene un módulo de Web.

Ciclo de vida. Son los eventos en el marco de trabajo de la existencia del componente. Cada tipo de componente tiene eventos definidos que marcan su transición en estados donde tienen disponibilidad variable de uso.

Cliente de la aplicación. Componente cliente de la primer capa que se ejecuta con su propia máquina virtual. Un cliente de aplicación tiene acceso a algunos API de la plataforma J2EE (JNDI, JDBC, RMI-IIOP, JMS).

Commit. El punto en una transacción cuando todas las actualizaciones hacia cualquier recurso incluido en la transacción son hechas permanentes.

Componente. Unidad de software a nivel de aplicación apoyada por el contenedor. Los componentes son configurables en tiempo de instalación. La plataforma J2EE define cuatro tipos de componentes: *enterprise bean*, Web, *applet* y clientes de aplicación.

Componente JavaBean. Clase Java que puede ser manipulada en una herramienta visual y ser parte de aplicaciones. Un componente *JavaBean* debe tener ciertas propiedades y convenciones de interfaz de eventos.

Componente Web. Componente que provee servicios en respuesta a una requisición; *servlet* o página JSP.

Connector. Mecanismo de extensión estándar a contenedores para proveer de conectividad a EIS. Un contenedor es específico a un EIS y consiste de un adaptador de recursos y herramientas de desarrollo de aplicación para la conectividad del EIS. El adaptador de recursos es puesto en el contenedor por medio de su soporte para contratos a nivel sistema definidos en la arquitectura del *connector*.

Contenedor. Entidad que provee del manejo del ciclo de vida, seguridad, instalación y servicios en tiempo de ejecución a componentes. Cada tipo de contenedor también provee de servicios específicos del componente.

Contenedor de applet. Contenedor que incluye soporte para el modelo de programación del *applet*.

Contenedor cliente de la aplicación. Contenedor que soporta componentes cliente de la aplicación.

Contenedor EJB. Contenedor que implementa el contrato del componente EJB de la arquitectura J2EE. Este contrato especifica el ambiente de ejecución para el *enterprise bean* que incluye seguridad, concurrencia, manejo del ciclo de vida, transacción, instalación, nombrado y otros servicios. Un contenedor EJB es provisto por un servidor J2EE o EJB.

Contenedor JSP. Contenedor que provee de los mismos servicios que el contenedor *servlet* y una máquina que interpreta y procesa páginas JSP en *servlet.*

Contenedor servlet. Contenedor que provee de servicios de red sobre requisiciones y respuestas que son mandadas, requisiciones decodificadas y respuestas de formatos. Todos los contenedores *servlet* deben soportar HTTP como el protocolo para requisiciones y respuestas, pero también puede soportar otros protocolos adicionales como HTTPS.

Contenedor Web. Contenedor que implementa el contrato de componente Web de la arquitectura J2EE. Este contrato especifica el ambiente en tiempo de ejecución para los componentes Web que incluye seguridad, concurrencia, manejo del ciclo de vida, transacción, instalación y otros servicios. Un contenedor de Web provee de los mismos servicios que el contenedor JSP y una vista de los APIs de la plataforma J2EE. Un contendor de Web es provisto por un servidor de Web o J2EE.

Contexto EJB. Objeto que permite al *entity bean* invocar servicios que provee el contenedor y obtener información del cliente que invocó al método.

CORBA. *Common Object Request Broker Architecture.* Lenguaje independiente, el modelo de objetos distribuido especificado por OMG.

DD. *Deployment Descriptor.* Archivo XML que describe como debería ser instalado cada módulo y aplicación.

DTD. *Document Typc Definition.* Descripción de la estructura y propiedades de una clase de archivos XML.

EIS. Ver *Enterprise Information System.*

EJB. Ver *Enterprise JavaBean.*

Enterprise bean. Componente que implementa una tarea o entidad de negocio y reside en un contenedor EJB.

Enterprise Information System.. Las aplicaciones que comprende un sistema empresarial existente para tomar información extensa de la compañia. Estas aplicaciones proveen de infraestructura de información para una empresa. Un *enterprise information system* ofrece un conjunto bien definido de servicios a

sus clientes. Estos servicios son expuestos a sus clientes como interfaces locales o remotas. Los ejemplos de *enterprise information system* incluye: sistemas de planeación de recursos empresariales, sistemas de procesamiento de transacción *mainframe* y sistemas de bases de datos legadas.

Enterprise JavaBean. Arquitectura de componentes para el desarrollo e instalación de aplicaciones orientadas a objetos, distribuidas y a nivel empresarial. Las aplicaciones que son escritas usando la arquitectura *Enterprise JavaBean* son escalables, transaccionales y seguras.

Entity bean. *Enterprise bean* que representa datos persistentes mentenidos en la base de datos. Un *entity bean* puede manejar su propia persistencia o puede delegar esta función a su contenedor. Un *entity bean* es identificado por una llave primaria. Si el contenedor donde el *entity bean* está instalado llega a fallar, el *entity bean*, su llave primaria, y cualquier referencia remota sobrevive a la falla.

HTML. *Hypertext Markup Language.* Lenguaje de marcado para documentos de hipertexto sobre la Internet. HTML permite que sean embebidos imagenes, sonidos, video, formas, referencias hacia otros objetos con URL y formateo de texto básico.

HTTP. *Hipertext Transfer Protocol.* El protocolo de Internet usado para traer objetos de hipertexto de *host* remotos. Mensajes HTTP consisten de requisiciones de clientes a servidores y respuestas de servidores a clientes.

HTTPS. HTTP en capas sobre el protocolo SSL.

Interface *home*. Una de dos interfaces para el *enterprise bean.* La interfaz *home* define cero o más métodos para manejar al *bean.* La interface *home* de un *session bean* define métodos *create* y *remove*, mientras que para un *entity bean* define métodos *create, finder* y *remove.*

Interface *remote*. Una de dos interfaces para el *enterprise bean.* La interface *remote* define los métodos de negocio que puede invocar un cliente.

IDL. *Interface Definition Language.* Lenguaje usado para definir interfaces hacia objetos CORBA remotos. Las interfaces son independientes del sistema operativo y lenguaje de programación.

IIOP. *Internet Inter-ORB Protocol.* Protocolo usado para la comunicación entre CORBA *object request brokers.*

J2EE. Ver *Java 2 Enterprise Edition.*

J2SE. Ver *Java 2 Standard Edition.*

JAR. *Java ARchive.* Archivo independiente de la plataforma que permite a muchos archivos ser agregados en uno sólo.

Java 2 Enterprise Edition. Ambiente para desarrollo e instalación de aplicaciones empresariales. La plataforma J2EE consiste de un conjunto de servicios, APIs y protocolos que proveen de funcionalidad para el desarrollo multicapa de aplicaciones basadas en Web.

Java 2 Standard Edition. La plataforma de tecnología esencial de Java.

Java Message Service. API para el uso de sistemas de mensajería empresarial como *IBM MQ Series, TIBCO Rendevous*, etcétera.

Java Naming and Directory Interface. API que provee de la funcionalidad de los servicios de nombrado y directorio.

Java Transaction API. API que permite a las aplicaciones y servidores J2EE a acceso de transacciones.

Java Transaction Service. Especifica la implementación de un manejador de transacción que soporte JTA e implemente la asociación Java hacia la especificación OMG *Object Transaction Service* (OTS) 1.1 a un nivel más bajo del API.

Java IDL. Tecnología que provee de interoperabilidad con CORBA y capacidades de conectividad para la plataforma J2EE. Estas capacidades permiten a las aplicaciones J2EE invocar operaciones sobre servicios de red remotos usando OMG IDL e IIOP.

JavaMail. API para mandar y recibir correo electrónico.

JavaServer Pages. Tecnología de Web extendible que usa plantillas de datos, elementos personalizados, lenguaje de *scripting* y objetos de Java del lado del servidor para regresar contenido dinámico a un cliente. Típicamente la plantilla de datos son elementos HTML o XML y en muchos casos el cliente es un navegador de Web.

JDBC. API para la conectividad independiente de la base de datos entre la plataforma J2EE y un amplio rango de recursos de datos.

JMS. Ver *Java Message Service.*

JNDI. Ver *Java Naming and Directory Interface.*

JSP. Ver *JavaServer Pages.*

JTA. Ver *Java Transaction API.*

JTS. Ver *Java Transaction Service.*

Lógica del negocio. El código que implementa la funcionalidad de una aplicación. En el modelo *Enterprise JavaBean*, esta lógica es implementada por los métodos del *enterprise bean.*

Llave primaria. Objeto que identifica de forma unica al *entity bean.*

Manejador de recursos. Provee de acceso a un conjunto de recursos compartidos. El manejador de recursos participa en transacciones que son externamente controladas y coordinadas por un manejador de transacción. Un manejador de recursos está típicamente en diferentes espacios de direcciones o sobre una máquina diferente de los clientes que las accesan.

Manejador de transacción. Provee de servicios y manejo de funciones requeridas para el soporte de demarcacion de transacciones, manejo de recursos transaccionales, sincronizacion y propagación del contexto de la transacción.

Mensaje. En *Java Message Service*, una requisición asíncrona, reporte o evento que es creado, mandado y consumido por una aplicación empresarial, no por un humano. Contiene información vital necesaria para coordinar aplicaciones empresariales, en la forma de dato formateado que describe acciones de negocio específicas.

Métodos *callback*. Métodos del componente llamados por el contenedor para notificarlo de eventos importantes en su ciclo de vida.

Método *create*. Método definido en la interface *home* e invocado por el cliente para crear un *enterprise bean.*

Método *finder*. Método definido en la interface *home* e invocado por un cliente para localizar un *entity bean.*

Método de negocio. Método del *enterprise bean* que implementa la lógica o reglas del negocio de una aplicación.

Método *remove*. Método definido en la interface *home* e invocado por el cliente para destruir un *enterprise bean.*

MIME. Ver *Multipurpose Internet Mail Extensions.*

Módulo. Unidad de software que consiste de uno o más componentes J2EE del mismo tipo de contenedor y un DD del tipo. Hay tres tipos de módulos: EJB, Web y cliente de aplicación. Los módulos pueden ser instalados como unidades *stand-alone* o ensambladas en una aplicación.

Módulo cliente de la aplicación. Unidad de software que consiste de uno o más clases y un DD del cliente de la aplicación.

Módulo EJB. Unidad de software que consiste de uno o más *enterprise bean* y un DD del EJB.

Módulo Web. Unidad que consiste de uno o más componentes de Web y un DD.

Multipurpose Internet Mail Extensions. Es un estándar de Internet oficial que especifica como los mensajes deben ser formateados y así puedán ser intercambiados entre diferentes sistemas de correo electrónico. MIME es un formato muy flexible, que permite incluir virtualmente cualquier tipo de archivo o documento en un mensaje de correo electrónico. Específicamente, mensajes MIME pueden contener texto, imágenes, audio, video y otros datos específicos de aplicación.

ORB. *Object Request Broker.* Biblioteca que permite a objetos CORBA localizar y comunicarse con otros.

OTS. *Object Transaction Service.* Definición de las interfaces que permite a objetos CORBA participar en transacciones.

Página JSP. Documento basado en texto usando plantillas fijas de datos y elementos JSP que describen como procesar una requisición para crear una respuesta.

Persistencia. Protocolo para transferir el estado de un *entity bean* entre sus variables de instancia y la base de datos.

Persistencia manejada por el bean. Transferencia de datos entre las variables del *entity bean* y el manejador de recursos, es manejada por el *entity bean*.

Persistencia manejada por el contenedor. Transferencia de datos entre las variables del *entity bean* y el manejador de recursos, es manejada por el contenedor de *bean*.

Producto J2EE. Implementación conforme a la especificación de la plataforma J2EE.

Recurso EIS. Entidad que provee de la funcionalidad específica del EIS para sus clientes. Ejemplo son: registro en sistemas de bases de datos, un objeto de negocios en un sistema de planeación de recursos empresariales y un programa de transacción en un sistemas de procesamiento de transacción.

RMI. *Remote Method Invocation.* Tecnología que permite a un objeto ejecutándose en una máquina virtual invocar métodos sobre un objeto ejecutándose en una máquina virtual diferente.

RMI-IIOP. Version de RMI implementada para usar el protocolo CORBA IIOP. RMI sobre IIOP provee de interoperabilidad con objetos CORBA implementados en cualquier lenguaje si todas las interfaces remotas son originalmente definidas como interfaces RMI.

Rollback. El punto en una transacción cuando todas las actualizaciones hacia cualquier recurso involucrado en la transaccion son revertidas.

Servlet. Un programa Java que extiende la funcionalidad de un servidor Web, generando contenido dinámico e interactuando con clientes Web usando el paradigma requisición-respuesta.

Servidor J2EE. La porción en tiempo de ejecución de un producto J2EE. El servidor J2EE provee contenedores EJB y/o Web.

Servidor EJB. Software que provee de servicios al contenedor EJB.

Servidor Web. Software que provee de servicios para acceso de Internet, intranet o extranet. Un servidor de Web hospeda sitios Web, provee de soporte para el protocolo HTTP y otros y ejecuta programas del lado del servidor (como CGI o servlet) que realizan ciertas funciones. En la arquitectura J2EE, un servidor de Web provee de servicios a un contenedor Web. Por ejemplo, un contenedor de Web típicamente depende del servidor de Web para proveer la toma de mensajes HTTP. La arquitectura J2EE asume que un contenedor de Web está instalado en un servidor de Web del mismo vendedor, así no se especifica contrato entre las dos entidades. Un servidor de Web puede instalar uno o más contenedores Web.

Sesión. Objeto usado por un *servlet* para rastrear la interacción del usuario con la aplicación Web a través de multiples requisiciones HTTP.

Session bean. *Enterprise bean* que es creado por un cliente y usualmente existe sólo para la sesión cliente-servidor. Un *session bean* realiza operaciones, como cálculos o acceso a base de datos, para el cliente. Mientras que un *session bean* puede ser transaccional, no es recuperable si una falla en el sistema ocurre. Objetos *session bean* pueden ser sin estado o pueden mantener un estado conversacional a través de métodos y transacciones. Si un *session bean* mantiene el estado, entonces el contenedor EJB maneja su estado. Sin embargo, el objeto *session bean* debe manejar su propia persistencia de datos.

Swing. Swing es un conjunto de componentes de interface de usuario gráfica. Simplifica la instalación de aplicaciones proporcionando un conjunto completo de elementos de interface de usuario escritos completamente en el lenguaje de programación Java. Los componentes de Swing dan una apariencia sin depender de un sistema de ventaneo específico.

SSL. *Secure Socket Layer.* Protocolo de seguridad que provee de privacidad sobre la Internet. El protocolo permite a las aplicaciones cliente-servidor comunicarse en forma que no puedan ser *eavesdropped* o *tampered.* Los servidores son siempre auteticados y los clientes opcionalmente lo son.

SQL. *Structured Query Language.* El lenguaje de base de datos relacionales estandarizado para definir objetos de la base de datos y manipulación de datos.

Tomcat. Es un contenedor servlet para las tecnologías Java *Servlet* y *JavaServer Pages.* Tomcat es desarrollado en un ambiente abierto y colaborativo y liberado bajo la *Apache Software License.*

Transacción. Unidad atómica de trabajo que modifica datos. Una transacción incluye una o más sentencias de programa, todo sera completado o revertido. Las transacciones permiten a usuarios múltiples acceder múltiples datos concurrentemente.

Transacción manejada por el bean. Transacción cuyos límites son definidos por el *enterprise bean.*

Transacción manejada por el contenedor. Transacción cuyos límites son definidos por el contenedor EJB. Un *entity bean* debería usar transacciones manejadas por el contenedor.

URL. *Uniform Resource Locator.* Estándar para escribir una referencia textual a un pieza arbitraria de datos en el Web. El URL se ve como protocolo://host/recurso donde el protocolo especifica el protocolo para recuperar el objeto (como HTTP o FTP), *host* especifica el nombre en Internet del *host* designado y recurso es una cadena (a menudo un archivo) que se le pasa al gestor del protocolo sobre el *host* remoto.

XML. *eXtensible Markup Language.* Lenguaje de marcado que permite definir las etiquetas necesarias para identificar el contenido, datos y texto en documentos XML.

Apéndice A: JBoss.

J2EE es un conjunto de estándares, cuando son usados en conjunto, que proporcionan una plataforma de desarrollo e instalación de aplicaciones de Web. J2EE incluye estándares para la capa intermedia (EJB y JMS), conectividad con base de datos (JDBC), transacciones (JTA/JTS), presentación (servlets y Java Server Pages) y servicios de directorio (JNDI).

JBoss es un servidor de aplicaciones basado en las especificaciones J2EE, desarrollado bajo el *open source GNU Lesser General Public License.*

Jboss esta escrito completamente en Java y requiere un sistema Java compatible con el JDK 1.3.

JBoss es uno de los integrantes principales de los grupos de *Java Open Source*, integra y desarrolla los servicios para una implementación completa basada en J2EE.

JBoss proporciona JBossServer, el contenedor básico EJB e infraestructura JMX. JBossMQ, para mensajeo JMS, JBossMX, para correo electrónico, JBossTX, para transacciones JTA/JTS, JBossSX, para JAAS basado en seguridad, JBossCX, para conectividad JCA y JBossCMP, para persistencia CMP. JBoss permite mezclar y adaptar estos componentes a través de JMX por reemplazar cualquier componente que se desee con una implementación conforme a JMX para los mismos APIs. JBoss incluso no impone componentes de JBoss.

Su meta es proporcionar el stack J2EE en el mundo *free/open*. La razón de su éxito proviene de JMX. JMX (Java Management eXtension) es la mejor arma que encontraron para la integración del software. Proporciona una espina dorsal común en la cual adopta sobre módulos, contenedores y *plugins*.

JBoss proporciona las implementaciones de muchos de estos servicios y se está en libertad de incluir otra implementación sobre la base JMX.

Apéndice B: Ant.

Antes se tenían que reescribir aplicaciones para que corrieran sobre diferentes sistemas operativos y recompilar las aplicaciones para que se ejecutaran sobre diferentes procesadores, la tarea completa de "manejar la construcción" fue una operacion compleja que para proyectos grandes se tenía gente trabajando sobre esa tarea todo el tiempo. La herramienta principal fue *make*, una herramienta poderosa que permitia mezclar las reglas de inferencia (que comandos se requerian para producir un archivo object ".o" de un archivo fuente ".c") con dependencias explicitas:

```
#makefile example

main: app.exe

app.exe: core.obj extras.obj
    #linking commands
    link /o app.exe core.obj extras.obj

core_headers: core.h platform.h

core.c: core_headers

extras.c: core_headers extras.h

.c.obj:
    #rule to compile a c file using a magic variable
    cc -c $@
```

Para cualquier persona experimentada en *makefiles* este un proyecto simple de C con dos archivos fuentes (core.c y extras.c), cada uno depende de algún archivo *header*, una regla de compilación establecida al final y un *linker* para ligar a los archivos.

Este es un archivo *make* simple pero uno más elaborado se vuelve muy complejo y uno que sea independiente de la plataforma es todavía más complejo ya que los comandos para realizar varias tareas varían de un sistema a otro. Esto es lo que *imake* viene a ser, un programa que genera los *makefile* apropiados para una plataforma probando el software instalado en el sistema para ver que trabaje y genere un *makefile* específico a la plataforma. El desarrollo a través de la plataforma nunca fue fácil.

Java cambio la noción completa de software a través de la plataforma. Ahora se puede ejecutar un programa sobre diferentes sistemas operativos y microprocesadores.

Ant cambio la noción completa de desarrollo de software a través de la plataforma. *Ant* es una herramienta de construcción basada en Java de *The Apache Software Foundation.*

Ant usa y extiende las clases de Java. Los archivos de configuracion son XML basados en llamadas a un árbol de etiquetas donde varias tareas se ejecutan. Cada tarea es ejecutada por un objeto que implementa la interface *Task* particular.

A continuación se presenta un ejemplo de un archivo llamado build.xml:

```
<?xml version="1.0"?>
<!-- build file for lesson 1 -->

<project name="tutorial" default="build" basedir=".">
    <target name="build">
        <javac srcdir="." />
    </target>
</project>
```

Ant se puede bajar del sitio apache http://jakarta.apache.org en código binario o fuente.

Apéndice C: Historia de Enterprise JavaBeans,

La especificación EJB 1.0 fue liberada en la primavera de 1998. Este es el primer enfoque que define un modelo de componentes del lado del servidor para la plataforma Java. La especificación EJB logró un alto grado de aceptación y obtuvo un papel importante en el desarrollo de aplicaciones distribuidas. No obstante, EJB tuvo varios problemas y uno de ellos es el manejo de seguridad. Sin embargo, la naturaleza distribuidad de EJB hizo un acercamiento sofisticado con respecto a la seguridad indispensable.

EJB provee de servicios de bajo nivel, tales como manejo de la transacción o persistencia de objetos. Al proveer de tales servicios, llega a ser posible implementar componentes que están libres de código relacionado con los servicios. Además, las aplicaciones distribuidas llegan a ser más fáciles de desarrollar, son más flexibles y escalables. Se logra una separación entre la lógica de negocio y la interfaz de usuario. La nueva filosofía introdujó una alternativa importante para la arquitectura cliente-servidor tradicional. Desde que la especificación fue liberada, despertó gran interés en este nuevo concepto de programación distribuida, pero también de aplicaciones del servidor y proveedores de aplicaciones cliente.

La especificación EJB 1.1 fue liberada en mayo de 1999.

Las características nuevas en EJB 1.1 son:

- Soporte obligatorio de *entity bean.*
- La adopción del *deployment descriptor* XML.
- La creación de un contexto JNDI default.
- Cambios a seguridad.

La especificación EJB 1.2 fue liberada en agosto del 2001.

Las características nuevas en EJB 1.2 son:

- Interfaces locales que proveen la eficiencia de relaciones *entity* dentro del contenedor.
- Relaciones manejadas por el contenedor, el contenedor maneja las relaciones de objeto entre *entity beans.*
- Cambios a EJB QL, un lenguaje de consulta para métodos *finder* EJB portables.
- Mejoras a las interfaces de *Message-Driven Beans* así como para las interfaces *remote entity bean.*

Apéndice D: J2EE 1.3.

La especificación J2EE 1.3 fue liberada en agosto del 2001 por Sun Microsystems bajo el programa Java Community Process. La especificación ofrece a los desarrolladores J2EE de la *Connector Architecture* e incrementa la integración para XML. Además, la especificación incrementa el nivel de soporte para el API *Java Message Service* (JMS) de opcional a requerido y provee de soporte para mensajes manejados por la arquitectura EJB e interoperabilidad de componentes EJB a través de IIOP.

Soporte XML.

XML – una parte esencial de la plataforma J2EE para habilitar tecnología negocio a negocio – se refuerza en la versión J2EE 1.3. Algunas de estas mejoras incluyen: el nuevo *Java API for XML Parsing* (JAXP) 1.1, soporte de servlet para filtros y la habilidad para escribir y manipular páginas *Java Server Pages*™(JSP™) en XML, incluyendo la validación de estos contra los esquemas.

JAXP permite la lectura, manipulación y generación de documentos XML a través de APIs de Java. JAXP 1.1 soporta los últimos estádares XML, incluyendo DOM SAX y XSLT. Ademas, cualquier *parser* conforme a XML puede ser transparentemente integrado con una aplicación Java y los desarrolladores tienen la habilidad para intercambiar *parsers* dependiendo de las necesidades de la aplicación, sin cambiar el código.

Los filtros *servlet* de Java son un nuevo tipo de componentes de web en la capa web para la liberación 1.3. Los filtros soportan la aplicación de multiples transformaciones basadas en contexto para requisiciones de *servlet* y respuesta de datos. Permiten a los desarrolladores separar mejor el código que genera el contenido del código que los tranforma, facilitando el mantenimiento y reuso. Uno de los usos de los filtros en una aplicación de la plataforma J2EE es transformar la salida para satisfacer las características de presentación de los diferentes dispositivos de clientes mandando la requisición. Otros cambios hacia las tecnologías JSP y *servlet* en la plataforma J2EE incluye capacidades mejoradas para procesamiento XML en arquitecturas de aplicación de n-capas.

El uso de XML para los *deployment descriptors* de objetos de negocio tales como transacción y seguridad tienen al *bean* como parte de la especificación desde la especificación EJB 1.2. Como la información del *deployment descriptor* es declarativa, puede ser cambiada sin modificar el código fuente del *bean*. En tiempo de ejecución, la información del *descriptor* es leida por el servidor y actua de acuerdo a éste.

J2EE Connector Architecture.

La J2EE *Connector Architecture* provee de un estándar para aplicaciones integradas en el *back-end*, tales como sistemas *Enterprise Resource Planning* (ERP) y *Customer Resource Management* (CRM) para "*plug and play*" con cualquier servidor de aplicación compatible con la plataforma J2EE 1.3. Con el soporte para la *Connector Architecture*, los vendedores de servidores de aplicación son hábiles para influenciar adaptadores de recursos para la conectividad entre varios *enterprise information systems* (EISs)[1] sin trabajo adicional.

Un adaptador de recursos es un componente de la plataforma J2EE que implementa la tecnología J2EE Connector Architecture para un EIS específico. Por medio de este adaptador que comunica a una aplicación con un EIS. Almacenado en un archivo *Resource Adapter archive* (RAR), un adaptador de recursos puede ser instalado sobre cualquier servidor, similar a un archivo EAR (un archivo JAR que contiene una aplicación).

Mientras que los *drivers* de la tecnología *Java DataBase Connectivity* (JDBC™) permiten a las aplicaciones J2EE acceder e integrar los datos desde sistemas manejadores de base de datos relacionales, la J2EE *Connector Architecture* basada en adaptadores de recursos permiten el acceso y la integración con EISs hetereogeneos que pueden no ser relacionales.

Así, la J2EE *Connector Architecture* soluciona muchos de los cambios enfrentados por los desarrolladores de software y vendedores de servidores de aplicación actuales quienes deben proveer de integración a sistemas hetereogeneos existentes.

Java Message Service API.

El *Java Message Service* (JMS) API es un API de Java que permite a las aplicaciones crear, mandar, recibir y leer mensajes.

El mensajeo permite la comunicación distribuida que es débilmente acoplada: el que envia y el receptor no tienen que estar disponibles al mismo tiempo para comunicarse. En tecnologías fuertemente acopladas como llamadas a procedimientos remotos, una aplicación debe estar enterada de los métodos de la aplicación remota.

Cuando el JMS API fue introducido en 1998, su propósito más importante fue permitir a aplicaciones Java interoperar con los sistemas existentes de la capa intermedia orientados a objetos tales como MQSeries de IBM y TIB/Rendezvous de TIBCO software.

[1] Sistemas de Información Empresarial

Desde ese momento, muchos vendedores han adoptado e implementado el JMS API, así un producto JMS puede ahora proveer de una capacidad de mensajeo completa para una empresa. En la liberación de J2EE 1.2, el vendedor sólo se le requería proveer de interfaces JMS API, no implementarlas.

Con la agregacion de JMS API hacia la plataforma J2EE 1.3, el desarrollo empresarial es más simplificado: acoplado débilmente, confiable, interacciones asíncronas entre componentes de la plataforma J2EE y sistemas legados pueden ocurrir sin atar la red u otros recursos.

Message-Driven Beans e Interoperabilidad IIOP.

El JMS API ha mejorado la plataforma al soportar un nuevo tipo de enterprise bean, el *message-driven bean*, el cual permite a las aplicaciones procesar JMS API mensajes asincronicamente.

El contenedor recibe el mensaje y activa al *bean*. Entonces, el *bean* realiza su trabajo y va a distancia: la transaccion es enterada y no hay interfaces *home* y *remota*. Ordinariamente los *enterprise beans* permiten mandar mensajes y recibirlos sincrónicamente, pero no asincrónicamente.

El mensaje puede ser mandado por cualquier componente J2EE – desde un cliente de aplicación, otro *enterprise bean* o componente de web – o desde una aplicación o sistema que no use la tecnología J2EE. Además, un desarrollador puede agregar conducta nueva a una aplicación existente basada en tecnología J2EE por añadir un *message-driven bean* para operar sobre eventos específicos. Así, la aplicación puede ejecutar si o no todos los componentes que están ejecutandose simultaneamente, porque no hay dependencias sobre los componenetes y sus interfaces.

Así para la *Connector Architecture*, la especificación también provee de interoperabilidad entre componentes EJB a través de IIOP. Este protocolo permite invocaciones de *session* y *entity beans* de los componentes que están instalados en productos de diferentes vendedores. De esta manera, un servidor de aplicaciones puede llamar a una interface *home* o *remote* de un componente EJB sobre otro servidor de aplicaciones desde otros lenguajes, usando CORBA 2.3.

Impulso de la Industria.

Desde la liberación de la especificación de la plataforma J2EE 1.2 en Diciembre de 1999, 28 compañias han firmado como concesionarios y 13 han pasado el conjunto de prueba de compatibilidad. La mayoría de los concesionarios actualmente envían los productos con una marca de compatibilidad con la plataforma J2EE, pasando a sus cliente los beneficios probados de la propuesta de valor de la tecnología J2EE: tiempo

de entrega de soluciones más rápidas al mercado, libertad de elección y conectividad simplificada.

Desde su incepción en 1995 como abierto, el proceso inclusivo para desarrollar y revisar la tecnología Java, el programa *Java Community Process* ha sostenido la evolución de la plataforma Java en cooperación con la comunidad internacional de desarrolladores usando la tecnología Java. Más de 90 especificaciones de la tecnología Java están en desarrollo en el programa de *Java Community Process*, el cual tiene más de 350 compañías y participantes individuales.

Apéndice E: ProductoBean.java.

```
package datos;

import java.util.Vector;
import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;

/**
 * @author Griselda Gonzalez Rodriguez
 * @version 2.0
 */

/**
 * Implementacion de los metodos del EJB Producto.
 * @version 2.0
 */

public class ProductoBean implements EntityBean {
    public String codigoDeBarras;
    public String titulo;
    public String genero;
    public String portada;
    public float precio;

    protected EntityContext contexto;

    /**
     * Regresa el codigo de barras del producto.
     */
    public String getCodigoDeBarras(){
        return codigoDeBarras;
    }

    /**
     * Establece el codigo de barras del producto.
     */
    public void setCodigoDeBarras(String codigoDeBarras){
       this.codigoDeBarras = codigoDeBarras;
    }

    /**
     * Regresa el titulo del producto.
```

```
 */
public String getTitulo(){
    return titulo;
}

/**
 * Establece el titulo del producto.
 */
public void setTitulo(String titulo){
   this.titulo = titulo;
}

/**
 * Regresa el genero del producto.
 */
public String getGenero(){
   return genero;
}

/**
 * Establece el genero del producto.
 */
public void setGenero(String genero){
    this.genero = genero;
}

/**
 * Regresa el nombre del archivo de la portada del producto.
 */
public String getPortada(){
    return portada;
}

/**
 * Establece el nombre del archivo para la portada del producto.
 */
public void setPortada(String portada){
    this.portada = portada;
}

/**
 * Regresa el precio del producto.
 */
public float getPrecio(){
    return precio;
```

```
}

/**
 * Establece el precio del producto.
 */
public void setPrecio(float precio){
    this.precio = precio;
}

public String ejbCreate (String codigoDeBarras,
    String titulo, String genero,
    String portada, float precio)
    throws CreateException {
    setCodigoDeBarras(codigoDeBarras);
    setTitulo(titulo);
    setGenero(genero);
    setPortada(portada);
    setPrecio(precio);
    return codigoDeBarras;
}

public void ejbPostCreate(String codigoDeBarras,
    String titulo, String genero,
    String portada, float precio) {
}

public void ejbRemove() {}

public void ejbLoad() {}

public void ejbStore() {}

public void ejbPassivate() {}

public void ejbActivate() {}

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

public void unsetEntityContext() {
    contexto = null;
}

/**
```

```
     * Actualiza los campos del producto.
     */
    public void actualiza(String[] campos){
        setTitulo(campos[1]);
        setGenero(campos[2]);
        setPortada(campos[3]);
        setPrecio(Float.parseFloat(campos[4]));
    }

    /**
     * Regresa los datos del producto.
     */
    public Vector daDatos(){
        Vector v = new Vector();
        v.add(codigoDeBarras);
        v.add(titulo);
        v.add(genero);
        v.add(portada);
        v.add(Float.toString(precio));
        return v;
    }
}
```

Apéndice F: ControlWebBean.java.

```
package controlador.web;

import javax.rmi.PortableRemoteObject;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.RemoveException;
import javax.ejb.FinderException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;

import datos.CDHome;
import datos.DVDHome;
import datos.LibroHome;
import datos.Producto;
import utileria.ConstantesDeProductos;

import controlador.BusquedaDeProductos;
import controlador.BusquedaDeProductosHome;

import java.util.Vector;

/**
 * Implementacion de los metodos del EJB ControlWeb.
 * @version 2.0
 */

public class ControlWebBean implements SessionBean{
    /** Conexin de la base de datos. */
    private Connection con;

    public void ejbCreate() {
        try {
```

```
            estableceConexion();
        } catch (Exception ex) {
            throw new EJBException("Unable to connect to database. " +
                                   ex.getMessage());
        }
    }

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

    public void ejbActivate() {}

    public void ejbPassivate() {}

    public void setSessionContext(SessionContext sc) {
    }

    /**
     * Establece la conexin con la base de datos.
     */
    private void estableceConexion()throws NamingException, SQLException {
        InitialContext ic = new InitialContext();
        String nombreBD = "java:comp/env/jdbc/OracleDB";
        DataSource ds = (DataSource) ic.lookup(nombreBD);
        con =  ds.getConnection();
    }

    /**
     * Regresa los productos marcados como estreno.
     * @parameter tipoDeProducto El tipo del producto sobre el cual se buscan
     * los estrenos.
     * @return Los estrenos de acuerdo al tipo de producto.
     */
    public Object[][] daEstrenos(int tipoDeProducto)
        throws Exception {
        try {

            Context initial = new InitialContext();
            Object objRef = initial.lookup("ejb/BusquedaDeProductos");
```

```
        BusquedaDeProductosHome homeBusqueda =
            (BusquedaDeProductosHome)PortableRemoteObject.narrow(objRef,
                BusquedaDeProductosHome.class);
        BusquedaDeProductos busqueda = homeBusqueda.create();

        String[] codigos = busqueda.daEstrenos(tipoDeProducto);

        Object[][] o = new Object[codigos.length][];
        Producto[] productos = busqueda.daProductos(tipoDeProducto,codigos);

        busqueda.remove();

        for(int i=0; i < codigos.length; i++){
            Vector v = productos[i].daDatos();
            Object datos [] =
               new Object[tipoDeProducto == ConstantesDeProductos.LIBRO ?
               6 : 5];
            datos[0] = v.elementAt(0);//codigo de barras
            datos[1] = v.elementAt(1);//ttulo
            datos[2] = v.elementAt(3);//portada
            datos[3] = v.elementAt(4);//precio
            switch(tipoDeProducto){
                case ConstantesDeProductos.CD:
                    datos[4] = v.elementAt(5);//interprete
                    break;
                case ConstantesDeProductos.LIBRO:
                    datos[5] = v.elementAt(6);//editorial
                case ConstantesDeProductos.DVD:
                    datos[4] = v.elementAt(5);//autor o region
                    break;
            }
            o[i] = datos;
        }
        return o;
    }catch(NamingException ne){
        System.err.println("Excepcion de nombrado " + ne.getMessage());
    }catch(CreateException ce){
        throw new Exception();
    }catch(RemoveException re){
        throw new Exception();
    }
    return new Object[][]{};
}

/**
```

```
 * Regresa los generos de acuerdo al tipo del producto.
 * @parameter tipoDeProducto El tipo del producto sobre el cual se
 * obtiene la jerarquia.
 */
public Object[] daGenerosDelProducto(int tipoDeProducto) throws Exception{
    try{
        Statement stmt =
            con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY);

        String consulta =
            "Select descripcion from jerarquia where tipo = " +
            tipoDeProducto;
        ResultSet rs = stmt.executeQuery(consulta);
        if(rs.last()){
            String[] generos = new String[rs.getRow()];
            rs.first();
            int i=0;
            do{
               generos[i++] = rs.getString("descripcion");
            }while(rs.next());
           return generos;
        }else
        return new String[]{};
    }catch(SQLException e){
        throw new Exception();
    }
}

/**
 * Regresa los productos marcados como estreno.
 * @parameter tipoDeProducto El tipo del producto.
 * @parameter genero El genero del producto sobre el cual se buscan
 * los estrenos.
 * @return Los estrenos de acuerdo al tipo de producto y genero.
 */
public Object[][] daEstrenosDelGenero(int tipoDeProducto,String genero)
    throws Exception{
    try{

        Context initial = new InitialContext();
        Object objRef = initial.lookup("ejb/BusquedaDeProductos");
        BusquedaDeProductosHome homeBusqueda =
        (BusquedaDeProductosHome)PortableRemoteObject.narrow(objRef,
                                    BusquedaDeProductosHome.class);
```

```
        BusquedaDeProductos busqueda = homeBusqueda.create();

        String[] codigos =
            busqueda.daEstrenosDelGenero(tipoDeProducto,genero);
        Object[][] o = new Object[codigos.length][];
        Producto[] productos =
            busqueda.daProductos(tipoDeProducto,codigos);
        busqueda.remove();

        for(int i=0; i < codigos.length; i++){
            Vector v = productos[i].daDatos();
            Object datos [] =
             new Object[tipoDeProducto ==
                ConstantesDeProductos.LIBRO ? 6 : 5];
            datos[0] = v.elementAt(0);//codigo
            datos[1] = v.elementAt(1);//ttulo
            datos[2] = v.elementAt(3);//portada
            datos[3] = v.elementAt(4);//precio
            switch(tipoDeProducto){
                case ConstantesDeProductos.CD:
                    datos[4] = v.elementAt(5);//interprete
                    break;
                case ConstantesDeProductos.LIBRO:
                    datos[5] = v.elementAt(6);//editorial
                case ConstantesDeProductos.DVD:
                    datos[4] = v.elementAt(5);//autor o region
                    break;
            }
            o[i] = datos;
        }
        return o;
    }catch(NamingException ne){
        System.err.println("Excepcion de nombrado " + ne.getMessage());
    }catch(CreateException ce){
        throw new Exception();
    }catch(RemoveException re){
        throw new Exception();
    }
    return new Object[][]{};
}

public String[] daDetalle(String codigoDeBarras) throws Exception{
    String campos[] = new String[]{};
    try {
        Statement stmt = con.createStatement();
```

```
    String consulta = "select clave from tipo_codigo_de_barras " +
        "where codigo_de_barras = '" + codigoDeBarras + "'" ;
    ResultSet rs = stmt.executeQuery(consulta);
    if(!rs.next())
        throw new Exception();
    else{
    Producto producto = null;
    int tipoDeProducto = rs.getInt(1);
    Context initial = new InitialContext();
    switch(tipoDeProducto) {
            case ConstantesDeProductos.CD: {
            Object objref = initial.lookup("ejb/CD");
            CDHome home =
                (CDHome)PortableRemoteObject.narrow(objref,
                        CDHome.class);
            producto = home.findByPrimaryKey(codigoDeBarras);
                }
            break;
            case ConstantesDeProductos.DVD:{
            Object objref = initial.lookup("ejb/DVD");
            DVDHome home =
                (DVDHome)PortableRemoteObject.narrow(objref,
                        DVDHome.class);
            producto = home.findByPrimaryKey(codigoDeBarras);
            }
            break;
            case ConstantesDeProductos.LIBRO:{
            Object objref = initial.lookup("ejb/Libro");
            LibroHome home =
            (LibroHome)PortableRemoteObject.narrow(objref,
                        LibroHome.class);
            producto = home.findByPrimaryKey(codigoDeBarras);
            }
            break;
    }

    campos = new String[producto.daDatos().size()];
    producto.daDatos().copyInto(campos);
    }
    stmt.close();
}catch(FinderException fe){
    throw new Exception();
}catch(NamingException ne){
    System.err.println("Excepcion de nombrado " + ne.getMessage());
}catch(SQLException sqle) {
```

```
            throw new Exception();
        }

        return campos;
    }

    public Object[][] buscaProductos(int tipoDeProducto, int indice,
        String criterio) throws Exception{
        try{

            String criterios[] = new String[]{"","",""};
            criterios[indice] = criterio;
            Context initial = new InitialContext();
            Object objRef = initial.lookup("ejb/BusquedaDeProductos");
            BusquedaDeProductosHome homeBusqueda =
                (BusquedaDeProductosHome)PortableRemoteObject.narrow(objRef,
                                         BusquedaDeProductosHome.class);
            BusquedaDeProductos busqueda = homeBusqueda.create();

            String[] codigos = busqueda.busca(criterios,tipoDeProducto,true);

            Object[][] o = new Object[codigos.length][];
            Producto[] productos = busqueda.daProductos(tipoDeProducto,codigos);

            busqueda.remove();

            for(int i=0; i < codigos.length; i++){
                Vector v = productos[i].daDatos();
                Object datos [] =
                    new Object[tipoDeProducto ==
                        ConstantesDeProductos.DVD ? 4 : 5];
                datos[0] = v.elementAt(0);//codigo de barras
                datos[1] = v.elementAt(1);//ttulo
                datos[2] = v.elementAt(4);//precio
                switch(tipoDeProducto){
                    case ConstantesDeProductos.CD:
                        datos[3] = v.elementAt(5);//interprete
                        datos[4] = v.elementAt(6);//disquera
                        break;
                    case ConstantesDeProductos.LIBRO:
                        datos[3] = v.elementAt(5);//autor
                        datos[4] = v.elementAt(6);//editorial
                        break;
                    case ConstantesDeProductos.DVD:
                        datos[3] = v.elementAt(6);//region
```

```
                    break;
                }
                o[i] = datos;
            }
            return o;
        }catch(NamingException ne){
            System.err.println("Excepcion de nombrado " + ne.getMessage());
        }catch(CreateException ce){
            throw new Exception();
        }catch(RemoveException re){
            throw new Exception();
        }
        return new Object[][]{};
    }

    /**
     * Regresa los generos de acuerdo al tipo del producto.
     * @parameter tipoDeProducto El tipo del producto sobre el cual se
     * obtiene la jerarquia.
     */
    public Object[][] daTipoDeProducto() throws Exception{
        try{
            Statement stmt =
               con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                         ResultSet.CONCUR_READ_ONLY);
            String consulta = "Select * from tipo_de_productos";
            ResultSet rs = stmt.executeQuery(consulta);
            if(rs.last()){
                String[][] tipos = new String[rs.getRow()][];
                rs.first();
                int i=0;
                do{
                    tipos[i++] =
                         new String[]{Integer.toString(rs.getInt("clave")),
                    rs.getString("nombre"),rs.getString("imagen")};
                }while(rs.next());
                return tipos;
            }else
            return new String[][]{};
        }catch(SQLException e){
            throw new Exception();
        }

    }
```

```
/**
 * Registra la compra del cliente y regresa el no. de pedido.
 * @parameter nombre El nombre del cliente.
 * @parameter apellido El apellido del cliente.
 * @parameter direccion La direccion del cliente.
 * @parameter colonia La colonia del cliente.
 * @parameter codigoPostal El codigo postal del cliente.
 * @parameter ciudad La ciudad del cliente.
 * @parameter estado El estado de la Republica donde recibe el cliente.
 * @parameter telefono El telefono del cliente.
 * @parameter credito Indica si el pago es con tarjeta de credito.
 * @parameter tarjetahabiente El nombre del tarjeta habiente de la tarjeta.
 * @parameter numeroTarjeta El numero de cuenta de la tarjeta.
 * @parameter tipo El tipo de tarjeta(Visa o MasterCard).
 * @parameter fecha La fecha de expiracion de la tarjeta.
 * @parameter productos Los productos de la compra.
 * @parameter total El total de la compra.
 */
public int registraCompra(String nombre, String apellido,String direccion,
    String colonia, String codigoPostal,String ciudad,
    String estado, String telefono,boolean credito,
    String tarjetahabiente, int numeroTarjeta,String tipo,
    String fecha, String [][] productos,float total) throws Exception{
    try{
        Statement stmt = con.createStatement();

        String consulta =
            "select secuencia.nextval numero_pedido from dual";
        ResultSet rs = stmt.executeQuery(consulta);
        int numeroDePedido = rs.getInt(1);
        consulta =
            "insert into cliente values (" + numeroDePedido + ",'"
            + nombre + "','" + apellido + "','" + direccion +
            "','" + colonia + "','" + codigoPostal + "','" +
            ciudad + "','" + estado + "','" + telefono + "'," +
            (credito ? "1" : "0") + "," + total + ")";
        if(stmt.executeUpdate(consulta) < 0)
            throw new Exception();
        if(credito){
            consulta = "insert into tarjeta values (" + numeroDePedido +
                ",'" + tarjetahabiente + "'," + numeroTarjeta + ",'" +
                tipo + "','" + fecha + "')";
            if(stmt.executeUpdate(consulta) < 0)
                throw new Exception();
```

```
            }
            for(int i = 0; i < productos.length; i++) {
                consulta = "insert into carrito values (" + numeroDePedido +
                    ",'" + productos[i][1] + "'," + productos[i][0] + ")";
                if(stmt.executeUpdate(consulta) < 0)
                        throw new Exception();
            }
            con.commit();
            return numeroDePedido;
        }catch(SQLException e){
            throw new Exception();
        }
    }
}
```