UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

INVESTIGACIÓN DE LA PLATAFORMA J2EE Y SU APLICACIÓN PRÁCTICA

JUAN MANUEL BARRIOS NÚÑEZ

http://www.dcc.uchile.cl/~jbarrios/

jbarrios@dcc.uchile.cl

2003

UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

INVESTIGACIÓN DE LA PLATAFORMA J2EE Y SU APLICACIÓN PRÁCTICA

JUAN MANUEL BARRIOS NÚÑEZ

COMISIÓN EXAMINADORA		CALIFICACIONES		
		NOTA (n^o)	(Letras)	FIRMA
PROFESOR GUÍA SR. DIONISIO GONZÁLEZ	:	.....	.....	.....
PROFESOR CO-GUÍA SR. PATRICIO INOSTROZA	:	.....	.....	.....
PROFESOR INTEGRANTE SR. EDUARDO GODOY	:	.....	.....	.....
NOTA FINAL EXAMEN DE TÍTULO	:	.....	.....	.....

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

SANTIAGO DE CHILE
ENERO 2003

RESUMEN DE LA MEMORIA
PARA OPTAR AL TITULO DE
INGENIERO CIVIL EN COMPUTACION
POR: JUAN MANUEL BARRIOS NUÑEZ
FECHA: 30/05/2003
PROF. GUIA: SR. DIONISIO GONZALEZ

INVESTIGACIÓN DE LA PLATAFORMA J2EE Y SU APLICACIÓN PRÁCTICA

El presente trabajo tiene como objetivo adquirir conocimientos y experiencia teórica y práctica en el desarrollo de aplicaciones empresariales utilizando el modelo "Java 2 Platform, Enterprise Edition" (J2EE). Este nuevo modelo ha tomado importancia por proponer una arquitectura para desarrollar e integrar sistemas de una empresa, definiendo un servidor de aplicaciones que consta de múltiples componentes y servicios. Efectuar un estudio concreto sobre sus capacidades y elaborar metodologías de utilización es un paso necesario que permite su aplicación correcta en proyectos reales.

Para conseguir este objetivo, el trabajo fue dividido en una fase de investigación y en una fase de aplicación. En la fase de investigación se estudió la plataforma J2EE, sus tecnologías relacionadas y los patrones de diseño existentes para el desarrollo. En la fase de aplicación se utilizaron los conocimientos adquiridos para el desarrollo de un proyecto con el objetivo de encontrar una metodología de desarrollo para aplicaciones J2EE, obtener experiencia sobre las capacidades de esta plataforma y contar con un caso de estudio que permita apoyar el diseño y construcción de nuevos sistemas.

El resultado final es un informe que reúne los conocimientos necesarios para el entendimiento de la plataforma J2EE, su servidor de aplicaciones y sus componentes, junto con la implementación de un sistema de registro de actividades como proyecto práctico. Con este proyecto se obtuvo una metodología para abordar el desarrollo de un sistema J2EE, cuatro patrones de diseño para solucionar problemas concretos en la implementación de un sistema, y un conjunto de evaluaciones y conclusiones sobre el uso y las capacidades de esta tecnología.

J2EE es una arquitectura que ha evolucionado rápidamente, para transformarse en una opción a ser considerada para efectuar el desarrollo de aplicaciones empresariales, sin embargo su utilización se ha visto retrasada por la falta de conocimientos reales en su desarrollo e implementación. Por esta razón se necesita generar conocimientos concretos que permitan apoyar su uso correcto en aplicaciones empresariales reales, crear nuevos casos de estudio y desarrollar nuevos patrones de diseño que aporten con experiencia práctica en su utilización.

Índice general

1. Presentación **1**

1.1. Introducción 1

1.2. Motivación 2

1.3. Objetivos 3

2. Marco Teórico **5**

2.1. Plataforma Java 5

2.1.1. Lenguaje Java 5

2.1.2. Máquina virtual 6

2.1.3. Bibliotecas 6

2.1.4. Ediciones de Java 6

2.1.5. Java 2, Standard Edition 6

2.2. Java 2, Enterprise Edition 7

2.2.1. Arquitectura J2EE 8

2.2.2. Componentes de J2EE 9

2.2.3. Containers J2EE 10

2.2.4. Servicios J2EE 11

3. Componentes de J2EE **13**

3.1. Servlets 13

3.1.1. CGI 13

3.1.2. Container de Servlets 15

3.1.3. Ciclo de Vida 15

3.1.4. Cookies y Sesiones 16

3.1.5. Filtros 17

3.1.6. Clases principales 17

3.2. JavaServer Pages 18

3.2.1. Container JSP 19

3.2.2. Ciclo de vida 19

3.2.3. Sintaxis 20

3.2.4. Clases importantes 21

3.3. Enterprise JavaBeans 22

3.3.1. Objetivos 22

3.3.2. Tipos de Enterprise Beans 23

3.3.3. Container EJB 23

3.3.4. Session Beans 24

3.3.5. Entity Beans 32

3.3.6. Message-Driven Beans 46

4. Deploy **50**

4.1. Definiciones 50

4.2. Módulos 50
4.2.1. Descriptores 51
4.2.2. Aplicación J2EE 52
4.2.3. Módulos EJB 52
4.2.4. Módulos Web 52
4.2.5. Módulos de aplicaciones cliente 53
4.2.6. Módulos de adaptadores de recursos 53
4.3. Roles para la instalación 53
4.3.1. Proveedores de componentes 54
4.3.2. Ensambladores de aplicaciones 54
4.3.3. Instalador 54

5. Tecnologías de J2EE **55**

5.1. XML 55
5.2. Servicio de Nombres 56
5.3. CORBA 57
5.3.1. Java RMI 59
5.3.2. Comunicación entre RMI y CORBA 60
5.4. Conectores 61
5.5. Transacciones 62
5.5.1. Transacciones manejadas por el container 63
5.5.2. Transacciones manejadas por el bean 64
5.6. Mensajería 64

5.7. Seguridad 65

6. Patrones de Diseño **67**

6.1. Modelo-Vista-Controlador 67

6.2. Data Access Object 68

6.3. Session Façade 68

6.4. Service Locator 69

6.5. Value Object 69

6.6. Fast-Lane Reader 70

6.7. Consideraciones de diseño 70

6.7.1. Session beans 70

6.7.2. Entity beans 71

6.7.3. Message-driven beans 72

7. Proyecto: Sistema de Registro de Actividades **73**

7.1. Contexto y objetivos 73

7.2. Análisis 74

7.3. Diseño 76

7.3.1. Capa Negocio 77

7.3.2. Capa Web 79

7.4. Implementación 81

7.4.1. Herramientas de desarrollo 81

7.4.2. Implementación Capa Negocio 84

7.4.3. Implementación Capa Web 90

7.5. Deploy 95

8. Discusión y Conclusiones **96**

8.1. Conclusiones 96

8.2. Alternativas a J2EE 99

8.3. Trabajo futuro 100

A. Códigos fuentes **104**

A.1. EjbDAO.java 104

A.2. ActividadBean.java 106

A.3. ActionFormEntity.java 111

Capítulo 1

Presentación

1.1. Introducción

Internet y sus servicios, particularmente la Web, tienen una gran importancia en el desarrollo de las empresas en la actualidad, siendo factores esenciales para la llamada *Nueva Economía*.

En esta *Nueva Economía* la información toma tanta importancia como los productos, transformándose en parte vital para los mercados. El manejo de esta información crea nuevos desafíos para las empresas, que para mantenerse en el mercado o para obtener ventajas competitivas, tienen la necesidad de actualizar su forma de hacer negocios.

Inicialmente las empresas debieron realizar un correcto manejo de su información crítica, invirtiendo en la creación de uno o varios sistemas de información específicos a cada necesidad. En la actualidad el desafío es realizar una integración entre estos sistemas, aumentar la habilidad para correlacionar datos desde fuentes dispares y transformarlos en información útil para satisfacer distintas necesidades de la empresa.

Esta integración entre sistemas de información es crítica para el funcionamiento de las empresas y debe ser realizada en forma inteligente, es decir, con visión en el presente y, sobre todo, proyectada hacia el futuro. Por esto, la integración de sistemas debe cumplir ciertas características como son confiabilidad, disponibilidad, escalabilidad, seguridad y estandarización.

Confiabilidad, para garantizar que el manejo de la información será correcto. Disponibilidad, con el objetivo de mantener el sistema la mayor cantidad de tiempo posible en producción. Escalabilidad, para tener la facultad de satisfacer nuevos requerimientos que pueden aparecer en el tiempo y permitir la creación e integración de nuevos sistemas. Seguridad, para minimizar la posibilidad de ataques externos e internos debido a que se crearán

nuevos accesos a los depósitos de información crítica de la empresa. Estandarización, para no estar ligado a un proveedor en particular, ya que una dependencia de este tipo puede causar problemas en el cliente ya sea de tipo económico o técnico.

Con el objetivo de cumplir con los desafíos de la integración de una forma estándar, Sun Microsystems ha definido *Java 2 Platform, Enterprise Edition, J2EE*. J2EE es una especificación que señala un modelo de aplicación empresarial no ligada a un proveedor o producto en particular que satisface estas características y que, por tanto, permite incrementar la competitividad de las empresas en la *Nueva Economía* y proyectarlas hacia el futuro.

1.2. Motivación

En aplicaciones de integración para empresas es necesario utilizar un *servidor de aplicaciones*, es decir, un procesamiento intermedio que se encargue de unificar los accesos a las fuentes de información de la empresa, de unificar los servicios brindados por la empresa a distintos tipos de usuarios que pueden utilizar variadas formas de acceso, y de unificar el procesamiento de los datos en una única *lógica del negocio*, simplificando las aplicaciones al utilizar sólo partes de una lógica global.

Este servidor de aplicaciones se transforma rápidamente en parte esencial para el funcionamiento de la empresa, siendo muy deseable, por no decir necesario, que provea un conjunto de características avanzadas como manejo de transacciones, balanceo de carga, monitoreo, distribución, caché de resultados, administración de recursos, auditoría, seguridad y buen desempeño. Crear este servidor de aplicaciones es difícil y, en general, está fuera del ámbito de la lógica del negocio y de la misión de la empresa.

En un principio aparecieron en el mercado una gran cantidad de servidores de aplicación cada uno proporcionando sus propios servicios en un formato propietario no estándar. Esto tuvo como consecuencia que una vez elegido el servidor, el desarrollo de la solución queda ligada al producto y a su proveedor. Una gran dependencia del cliente con un proveedor en particular no es deseable en general para el primero, porque esta condición puede debilitar su capacidad negociadora en el tiempo y, probablemente, disminuir la cantidad o calidad de beneficios obtenidos.

La necesidad de independencia de proveedores por parte de los clientes hizo aparecer como primera solución una arquitectura basada en componentes y luego un comercio de componentes. En ese instante apareció como una buena opción el lenguaje Java al promocionar su independencia de plataforma, su estructura basada en componentes, una buena documentación y el respaldo de grandes compañías como Sun, IBM y Oracle.

Java 2 Platform, Enterprise Edition es una solución dada por Sun Microsystems para la creación de servidores de aplicación basado en el lenguaje Java. J2EE es una especifica-

ción, un concepto, que define una arquitectura para el desarrollo de aplicaciones basada en componentes que los servidores de aplicación J2EE-compatibles deben cumplir.

J2EE es una buena solución para el desarrollo de aplicaciones empresariales porque permite reducir el tiempo de desarrollo y mejorar la calidad del producto, sin embargo, J2EE no es simple porque consta de variadas componentes y unifica una gran cantidad de conceptos. Entre las nuevas componentes destacan servlets y JSP para generar contenidos dinámicos, y EJB por sus posibilidades de encapsular el acceso a una fuente de datos permitiendo el uso objetos de persistentes en el tiempo, centralizar la lógica del negocio y permitir utilizar programación distribuida. Sin embargo, J2EE y sus componentes son relativamente nuevos, con tres años desde la publicación de su primera versión, por lo que no son muy conocidas sus cualidades y objetivos lo que hace necesario un estudio profundo de este.

El último año han aparecido nuevos servidores de aplicación J2EE-compatibles de licencia libre que son una buena alternativa a los servidores de alto precio de proveedores tradicionales como IBM, Oracle y BEA. Además, existen en la actualidad una cantidad de patrones de diseño que son aplicables para el desarrollo de aplicaciones y cada vez aparecen nuevos patrones específicos para la plataforma J2EE. Estos dos elementos en conjunto hacen predecir un mayor uso de esta plataforma y la necesidad de experiencia en su utilización.

Para hacer un uso correcto de J2EE se necesita investigar sobre él, sobre sus posibilidades, sus ventajas y desventajas, y luego aplicarlo en situaciones concretas para generar experiencia que pueda ser utilizada por desarrolladores para crear aplicaciones.

El presente trabajo de título consistirá en el estudio de la especificación J2EE, en particular del servidor de aplicaciones, de sus componentes y sus servicios, para luego investigar los patrones de diseño existentes y posteriormente aplicar y evaluar los conocimientos en la realización de un proyecto específico definido y acotado.

Se recomienda al lector de este trabajo tener conocimientos sobre Internet y sus tecnologías relacionadas (principalmente Web, HTTP, HTML, etc.), la plataforma estándar de Java (J2SE), y para el tema de EJB se recomienda conocer de bases de datos relacionales y Java RMI o CORBA.

1.3. Objetivos

El principal objetivo del presente trabajo de título consiste en adquirir conocimientos y experiencia teórica y práctica en el desarrollo de aplicaciones empresariales utilizando el modelo J2EE. El elemento principal de J2EE es su servidor de aplicaciones, por tanto el estudio estará enfocado en las propiedades y características de este servidor y sus capacidades para un proyecto concreto.

Para lograr este objetivo, el trabajo ha sido dividido en dos fases: investigación y aplicación. Para la investigación se tiene como objetivo principal generar un documento que contenga información sobre:

- J2EE, en qué consiste, sus tecnologías, sus alcances y limitaciones.
- Servlets y JSP, sus objetivos, características y forma de utilización.
- EJB, su función en el modelo J2EE, sus propiedades y cómo utilizarlos.
- Patrones de diseño existentes, explicación y utilización.

Luego de la investigación se procederá a aplicar y evaluar los conocimientos adquiridos, desarrollando el proyecto de un sistema de registro de actividades de una empresa consultora, con el cual se desea obtener experiencia para:

- Conocer cuándo es posible utilizar el modelo J2EE para un proyecto.
- Obtener una metodología para desarrollar aplicaciones con J2EE.
- Obtener una evaluación sobre los patrones de diseño encontrados.
- Conocer detalles para desarrollar un proyecto J2EE (entorno de desarrollo, aplicaciones de apoyo útiles, etc.).

Capítulo 2

Marco Teórico

En este capítulo se presentará el contexto en el cual se enmarca J2EE, luego se presentará su arquitectura y finalmente se describirán los componentes y servicios que forman parte de él.

2.1. Plataforma Java

Una plataforma es el entorno de hardware o software en el cual se ejecuta un programa. A diferencia de las plataformas tradicionales como Linux, Windows y Solaris, la plataforma Java está basada sólo en software que se ejecuta sobre otras plataformas basadas en hardware. Por esto la plataforma Java y su software puede ser utilizado sobre variados sistemas operativos y hardware.

La plataforma Java está constituida de tres componentes: el lenguaje, la máquina virtual y las bibliotecas.

2.1.1. Lenguaje Java

El lenguaje Java es un lenguaje de propósito general, de alto nivel, que utiliza el paradigma de orientación a objetos. Su sintaxis y tipos están basados principalmente en C++, sin embargo, las diferencias principales con éste son la administración de memoria, siendo ejecutada por la máquina virtual automáticamente y no por el código de cada programa, y el soporte de procesos livianos o *threads* a nivel del lenguaje, que ayuda a controlar la sincronización de procesos paralelos. Estas características dan al lenguaje Java las propiedades de robustez y seguridad, evitando por ejemplo problemas de *buffer overflow* utilizados en ataques a sistemas.

2.1.2. Máquina virtual

Los programas escritos en Java son compilados como archivos ejecutables de una máquina virtual llamada *Java Virtual Machine (JVM)*. Existen implementaciones de esta máquina para múltiples plataformas, permitiendo ejecutar en diferentes arquitecturas el mismo programa ya compilado. La característica de independencia de la plataforma hace posible el libre intercambio de software desarrollado en Java sin necesidad de modificaciones, lo que ha sido llamado *"Write once, Run anywhere"*.[1]

Java es un lenguaje compilado e interpretado a la vez. Compilado ya que previo a su ejecución un programa debe ser transformado a un lenguaje intermedio, llamado *Java bytecodes*. Interpretado porque cada programa luego debe ser procesado y ejecutado por alguna implementación de la JVM específica a la plataforma.

2.1.3. Bibliotecas

El conjunto de bibliotecas del lenguaje es conocido como la *Java Application Programming Interface (Java API)* que es un gran conjunto de componentes que proporcionan diferentes herramientas para el desarrollo de programas Java. La API de Java está agrupada en conjuntos de bibliotecas relacionadas conocidas como *paquetes*, que contienen grupos de elementos básicos de Java, llamados *clases* e *interfaces*.

2.1.4. Ediciones de Java

La plataforma Java ha sido dividida en tres ediciones distintas según sus diferentes objetivos: *Java 2 Platform, Micro Edition*, orientado al desarrollo para artículos pequeños y móviles como PDAs, *Java 2 Platform, Standard Edition*, orientado al desarrollo para computadores personales y aplicaciones en general, y *Java 2 Platform, Enterprise Edition*, orientado al desarrollo de aplicaciones corporativas.

2.1.5. Java 2, Standard Edition

Java 2, Standard Edition (J2SE) es la edición principal de la plataforma Java sobre la cual se basan las demás ediciones. Provee las capacidades de desarrollo y ejecución de software escrito en lenguaje Java. Esta constituido de dos módulos principales (ver figura 2.1):

[1] Una traducción libre sería "Programe una vez, ejecute en cualquier lugar".

- *Software Development Kit (J2SE SDK)*, conocido inicialmente como *Java Development Kit (JDK)*, proporciona el software necesario para desarrollar programas en Java como es el compilador, el debugger y las bibliotecas con las funcionalidades del lenguaje.
- *Java Runtime Environment (JRE)*, contiene sólo el ambiente necesario y las bibliotecas principales para ejecutar software escrito en Java.

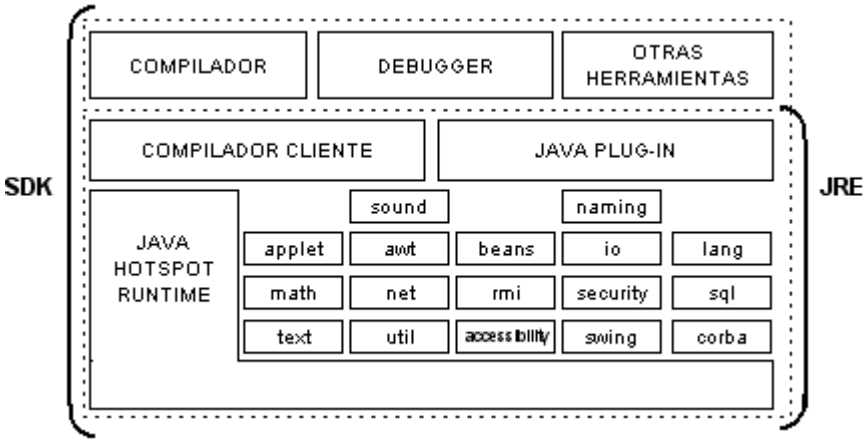

Figura 2.1: Java 2, Standard Edition

J2SE incluye herramientas y APIs para desarrollar aplicaciones con interfaz gráfica, acceso a base de datos, acceso a directorios, seguridad, entrada/salida, programación en red y varias otras funcionalidades.

Las bibliotecas principales son clases que se encuentran dentro de los paquetes *java.** y las bibliotecas con extensiones estándares se encuentran como clases dentro de los paquetes *javax.**. Para poder nombrar bibliotecas de aplicaciones desarrollados por terceros, se utiliza la convención de invertir el nombre de dominio de Internet del desarrollador y luego separar por funcionalidades, así existe por ejemplo *org.w3c.** y *org.apache.** entre otros.

2.2. Java 2, Enterprise Edition

Java 2, Enterprise Edition (J2EE) es una especificación que define una plataforma para crear aplicaciones empresariales utilizando un modelo de multicapas, dividiendo la aplicación en diferentes niveles, cada uno especializándose en una tarea en particular.

La figura 2.2 ilustra la composición de J2EE. Su estructura está basada en J2SE y un conjunto de sus APIs, a la cual J2EE aporta la especificación de *componentes*, *containers* y las APIs para los servicios de transacciones, mensajería, envío de correos y conectores de recursos externos.

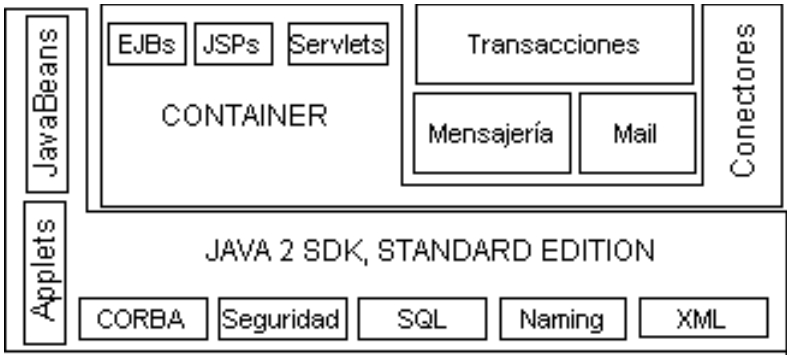

Figura 2.2: Estructura J2EE

La especificación de J2EE se encuentra en la versión 1.3, publicada en Agosto de 2001, y está compuesta de un conjunto de paquetes que, si bien algunos son opcionales para J2SE, son requeridos para la plataforma J2EE. Al momento de elaborar este informe aún se trabajaba en la elaboración del documento que detallará la versión final de J2EE 1.4.

2.2.1. Arquitectura J2EE

La especificación de J2EE define su arquitectura basándose en los conceptos de *capas*, *containers*, *componentes*, *servicios* y las características de cada uno de éstos. Las aplicaciones J2EE son divididas en cuatro capas: la capa cliente, la capa web, la capa negocio y la capa datos. La figura 2.3 representa estas capas y las componentes relacionadas.

Capa Cliente

Esta capa corresponde a lo que se encuentra en el computador del cliente. Es la interfaz gráfica del sistema y se encarga de interactuar con el usuario. J2EE tiene soporte para diferentes tipos de clientes incluyendo clientes HTML, applets Java y aplicaciones Java.

Capa Web

Se encuentra en el servidor web y contiene la lógica de presentación que se utiliza para generar una respuesta al cliente. Recibe los datos del usuario desde la capa cliente y basado en éstos genera una respuesta apropiada a la solicitud. J2EE utiliza en esta capa las componentes *Java Servlets* y *JavaServer Pages* para crear los datos que se enviarán al cliente.

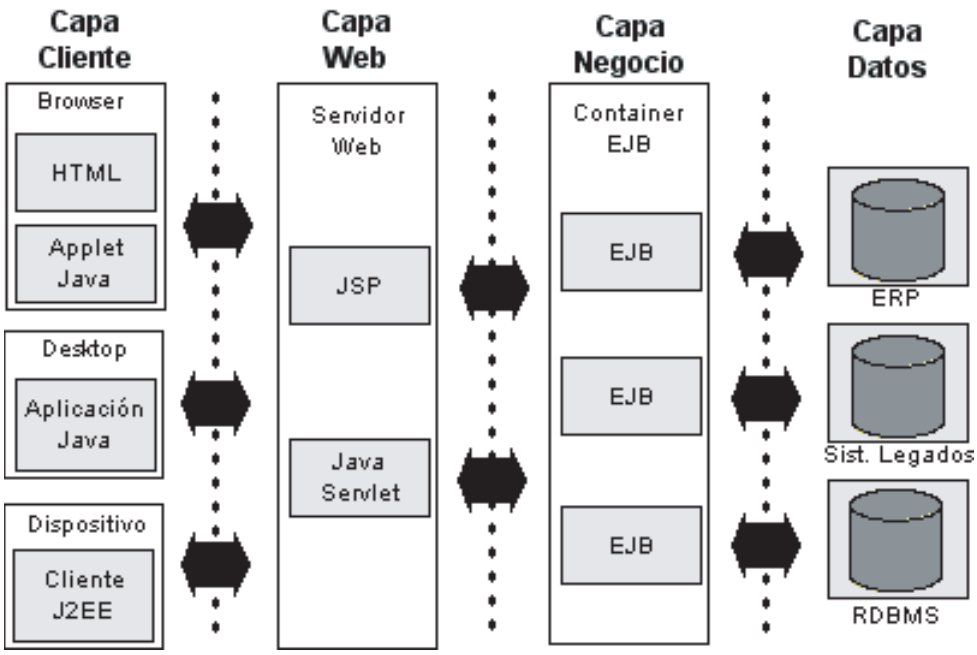

Figura 2.3: Arquitectura J2EE

Capa Negocio

Se encuentra en el servidor de aplicaciones y contiene el núcleo de la lógica del negocio de la aplicación. Provee las interfaces necesarias para utilizar el servicio de componentes del negocio. Las componentes del negocio interactúan con la capa de datos y son típicamente implementadas como componentes EJB.

Capa Datos

Esta capa es responsable del sistema de información de la empresa o *Enterprise Information System (EIS)* que incluye bases de datos, sistema de procesamiento datos, sistemas *legados*[2] y sistemas de planificación de recursos. Esta capa es el punto donde las aplicaciones J2EE se integran con otros sistemas no J2EE o con sistemas legados.

2.2.2. Componentes de J2EE

Cada componente de J2EE es una unidad de software independiente y funcional que cumple con las condiciones de interfaz definidas por la especificación de la componente y sólo

[2] *legacy systems*, sistemas que se utilizan desde largo tiempo en una empresa y que no son fáciles de actualizar. Por ejemplo en bancos se encuentran muchos sistemas antiguos escritos en COBOL.

tiene dependencias explícitas con su entorno de ejecución o *container*. Una componente puede estar compuesta por una única clase, o lo más común, por un conjunto de clases, interfaces y recursos.

Las componentes principales en la plataforma J2EE son cuatro:

1. Aplicaciones cliente, son programas nativos escritos en Java que en general poseen su propia interfaz gráfica y que se ejecutan en un proceso independiente en un computador personal. Son ejecutados dentro del container de aplicación dado por el JRE y tienen acceso a todas las capacidades de la capa media J2EE.
2. Applets, son componentes que se ejecutan típicamente en un browser web y proporcionan una interfaz web mejorada para aplicaciones J2EE. En general se ejecutan en un container de applets de un browser, pero pueden ejecutarse en una variedad de otras aplicaciones o dispositivos que proporcionen soporte para el container. Son utilizados como alternativa a interfaces más limitadas basadas en HTML.
3. Java Servlets y JavaServer Pages, son llamados colectivamente con el nombre de *componentes web*. Se ejecutan en un servidor web para responder a solicitudes HTTP desde clientes y pueden generar páginas HTML, que en general corresponde a la interfaz de usuario de una aplicación, o puede generar XML u otro formato de datos que será utilizado por otras componentes de la aplicación.
4. Enterprise JavaBeans, son componentes que contienen la lógica del negocio para una aplicación J2EE. Se ejecutan en un ambiente distribuido y que soporta transacciones. Encapsulan el acceso al EIS a través de la utilización de objetos que proveen la funcionalidad de manejo de transacciones y persistencia.

2.2.3. Containers J2EE

Un *container* es un servicio que proporciona la infraestructura necesaria a una componente para ser ejecutada, para proveer sus servicios a un cliente y para dar comunicación con otras componentes. Las componentes de una aplicación J2EE no interactúan directamente entre ellas, si no que deben utilizar los protocolos y métodos dados por el *container* para ese fin. Un *container* usualmente provee sus servicios a las componentes como un *Java Runtime Environment (JRE).*

Al existir un *container* entre las componentes y los servicios de J2EE se tiene la posibilidad de agregar transparentemente servicios como manejo de transacciones, chequeos de seguridad, administración de recursos y manejo de estados.

Un producto J2EE típico proveerá un *container* para cada tipo de componente de la aplicación: *container* de la aplicación cliente, *container* de applets, *container* de componentes web y *container* de EJB.

2.2.4. Servicios J2EE

J2EE especifica los siguientes servicios estándares, junto con las APIs necesarias para la utilización por parte de cada componente. Algunos de estos servicios actualmente son provistos por J2SE.

1. *HTTP* y *HTTPS*: Protocolos estándares utilizados para comunicaciones web y para comunicaciones seguras sobre *Secure Socket Layer (SSL)*, respectivamente. La API para clientes está definida por el paquete *java.net.** y la API para servidor está definida por las clases de servlets y JSP.

2. *JDBC*: Una API estándar para acceder a los recursos de una base de datos relacional de una forma independiente del proveedor. Esta API consta de dos partes, una interfaz para ser utilizada por las componentes y una interfaz de proveedores para definir drivers específicos. Oficialmente JDBC no es un acrónimo, aunque comúnmente se utiliza el nombre de *Java Database Connectivity*. [3, pág.461]

3. *JavaMail*: Una API que permite crear aplicaciones Java para mensajería y envío de correo electrónico en forma independiente de la plataforma y del protocolo a utilizar.

4. *JavaBeans Activation Framework (JAF)*: API que proporciona un framework de activación que es utilizado por otros paquetes, como JavaMail. Los desarrolladores pueden utilizar JAF para determinar el tipo de un trozo arbitrario de datos, accederlo, descubrir las operaciones disponibles en él e instanciar el bean apropiado para ejecutar esas operaciones. Por ejemplo, JavaMail usa JAF para determinar qué tipo de objeto instanciar dependiendo del *Mime-Type* del objeto.

5. *Remote Method Invocation-Internet Inter-ORB Protocol (RMI-IIOP)*: Está compuesto de APIs que permiten la programación distribuida a través de Java RMI. Los protocolos utilizados para la comunicación pueden ser JRMP (protocolo nativo de RMI), o IIOP (protocolo de CORBA). Ver sección 5.3.

6. *Java Interface Definition Language (JavaIDL)*: Permite a las aplicaciones actuar como clientes de servicios CORBA, invocando objetos externos utilizando el protocolo IIOP. Ver sección 5.3.

7. *Java Transaction API (JTA)*: Permite el manejo de transacciones. Las aplicaciones pueden utilizar JTA para iniciar, cerrar o abortar transacciones. Además permite al container comunicarse con monitores transaccionales y administradores de recursos. Ver sección 5.5.

8. *Java Message Service (JMS)*: Es una API que se utiliza para comunicarse con un *Message-Oriented Middleware (MOM)* en una forma independiente al proveedor para permitir mensajería del tipo punto a punto y del tipo publicar/subscribir entre sistemas. Ver sección 5.6.

9. *Java Naming and Directory Interface (JNDI)*: Una API estándar para el registro y acceso de servicios y objetos. Incluye soporte para *LDAP (Lightweight Directory Access Protocol)*, *COS (CORBA Object Services)*, Naming Service y Java RMI Registry. Ver sección 5.2.

10. *Java API for XML Parsing (JAXP)*: Permite a las aplicaciones la utilización de documentos XML a través de las APIs estándares SAX, DOM y XSLT. Ver sección 5.1.

11. *J2EE Connector Architecture (JCA)*: Una API de J2EE que permite agregar recursos nuevos a cualquier producto J2EE. La arquitectura Connector define un contrato entre un servidor J2EE y un adaptador de recursos para permitir esto. Ver sección 5.4.

12. *Java Authentication and Authorization Service (JAAS)*: Proporciona una forma para la identificación de usuarios y su autorización para acceder a recursos de la aplicación. Implementa una versión en Java del estándar *Plugable Authentication Module (PAM)*. Ver sección 5.7.

Capítulo 3

Componentes de J2EE

En este capítulo se describirán las componentes definidas por J2EE, como son servlets, JSP y EJB, sus características y propiedades. En esta explicación no se hará referencia a algún servidor en particular para dejar detalles específicos al capítulo referente al proyecto.

3.1. Servlets

Es una componente web de J2EE desarrollada con el objetivo de procesar requerimientos de un cliente o *requests* y generar respuestas con contenidos web dinámicos. Para ser ejecutados es necesaria la utilización de un servidor que de soporte a servlets y su container.

Por ejemplo, un servlet puede procesar los datos desde un formulario en HTML, mantener un registro de la transacción, actualizar una base de datos, contactar algún sistema remoto y retornar un documento dinámico o redirigir a otro servlet.

Los servlets son una buena opción como reemplazo a CGI, ya que proporciona una forma para generar documentos dinámicos que tiene un mejor desempeño al procesar cada *request* en un proceso liviano, tener acceso a las características de Java y sus extensiones, y poseer un manejo simple de parámetros, cookies y sesiones.

Para entender qué son los servlets es necesario conocer su antecesor, los CGIs.

3.1.1. CGI

Internet fue diseñada inicialmente para ser un depósito de datos estáticos guardados en páginas HTML, sin embargo, rápidamente se vio la necesidad de poder proveer datos

generados dinámicamente.

La primera respuesta a esta necesidad fue *Common Gateway Interface (CGI)*. Esta interfaz permite a un servidor Web ejecutar programas en variados lenguajes, de preferencia Perl, Delphi o C, al que se le entregan datos desde un formulario HTML a través de la entrada estándar o de variables de ambiente. Luego los procesa con las capacidades que provea el lenguaje elegido, como por ejemplo acceso a bases de datos, utilizar archivos, ejecutar programas externos, etc. Finalmente, los resultados de la ejecución deben ser escritos en un formato adecuado hacia la salida estándar que es direccionada al cliente.

Aún cuando CGI es una solución práctica tiene algunas limitaciones, que con el tiempo se han tratado de minimizar, pero aún se mantienen:

- Debido a la necesidad de crear un nuevo proceso para ejecutar un programa CGI cada vez que es invocado, CGI se torna intensivo en recursos y por tanto relativamente lento, en particular en los lenguajes interpretados, por lo que no proporciona una buena escalabilidad. Existen algunas soluciones a este problema como es el caso *mod_perl* para Perl sobre Apache, el cual consiste en mantener un sólo intérprete en ejecución que es compartido por todos los CGIs. Si bien soluciona el problema, obliga a ser más cuidadoso en la utilización de variables debido a que los CGIs comparten el mismo ambiente.
- El código utilizado para acceder a recursos del sistema, como archivos o bases de datos, son específicos a la plataforma del servidor. Por tanto muchas aplicaciones no funcionarán al trasladarse de servidor a menos que se realicen modificaciones en los códigos fuentes. Es particularmente cierto si se utilizan lenguajes de más bajo nivel, como C, con el fin de optimizar el tiempo de respuesta.
- Las aplicaciones CGI se tornan difíciles de mantener debido a que combinan contenidos y formas de presentación en un solo código fuente, lo que hace necesario ser diestro en ambos aspectos para crear y modificar un CGI. Se han hecho mejoras en este ámbito, en particular en Perl, como la creación de módulos que permiten la utilización de plantillas o templates con el objeto de separar el manejo de datos de su presentación.
- No existe un método simple para mantener y compartir recursos entre distintos CGIs, lo cual complica características como mantener sesiones de usuario o variables compartidas. Como posibles soluciones se pueden utilizar archivos en el servidor que son modificados concurrentemente, o guardar toda la información a compartir entre CGIs como datos en el cliente, por ejemplo a través de *cookies*, sin embargo estas soluciones o son complejas, para el primer caso, o son inseguras y restringidas en tamaño, para el segundo.

3.1.2. Container de Servlets

El container de servlet, o motor de servlets, es una extensión del servidor web que proporciona los servicios necesarios para la utilización de servlets, como son proveer acceso a la red, decodificar *requests* MIME, formatear respuestas MIME y administrar el ciclo de vida de cada servlet. Un container web debe soportar al menos el protocolo HTTP/1.0, pero puede utilizar otros protocolos como HTTP/1.1 o HTTPS.

Un container de servlets puede utilizar restricciones de seguridad en el entorno en el cual se ejecuta un servlet. En J2SE o J2EE estas restricciones pueden ser agregadas utilizando la arquitectura de permisos definida por la plataforma Java 2. Por ejemplo, un servidor de aplicaciones puede limitar la creación de *threads* para asegurar que otros componentes del container no serán afectados negativamente por la posible reducción en el desempeño del sistema.

3.1.3. Ciclo de Vida

Un servlet tiene un ciclo de vida bien definido, es decir, está especificado cómo y cuándo un servlet debe ser cargado e instanciado en la memoria, inicializado, ejecutado y finalmente destruido.

Instanciación

El container de servlets es el encargado de cargar e instanciar cada servlet. Esto puede suceder en la inicialización del container o postergado hasta que el container determine que el servlet es necesario para procesar un *request*. Los archivos necesarios para ejecutar un servlet deben ser localizados por el container para poder cargarlos utilizando el *Java Class Loader* que proporciona el lenguaje.

Inicialización

Luego de ser instanciado, el container debe inicializar el servlet. La inicialización permite a un servlet realizar tareas como leer datos persistentes de configuración, crear recursos costosos como conexiones a bases de datos, o cualquier tarea que deba ser ejecutada sólo una vez previamente a interactuar con clientes.

Aunque los servlets se ejecutan en servidores multithreads, no existen problemas de concurrencia durante su inicialización. El container debe ejecutar sólo una vez el método de inicialización al crear la instancia del servlet y no debe volver a hacerlo sin primero destruirlo.

Interacción con Clientes

Luego de ser inicializado, el servlet es utilizado para procesar *requests* de diferentes clientes. Todos los *requests* que corresponden al mismo servlet son procesados por la misma instancia en forma concurrente por lo cual hay que tomar las medidas necesarias para evitar problemas en el acceso a variables compartidas por posibles problemas de sincronización como *data-races*[1] o *deadlocks*[2]. El container garantiza que si un servlet implementa la interfaz *SingleThreadModel* sólo un thread a la vez ejecutará el servlet, evitando de forma simple, pero menos óptima, estos problemas de sincronización.

Destrucción

No es obligatorio que un container mantenga un servlet cargado por un período de tiempo específico. Cuando el container determina que un servlet debe ser destruido, que puede ser entre la inicialización del servlet y el cierre del container, se ejecuta un método particular con el objetivo de guardar configuraciones, cerrar conexiones, archivos, etc.

Cuando se encuentra en ejecución el método de destrucción, el container no debe aceptar nuevos clientes para esa instancia, y una vez finalizado debe eliminar la referencia a la instancia con el fin de ser eliminado por el recolector de basura de Java.

3.1.4. Cookies y Sesiones

Los servlets permiten la utilización de cookies y sesiones en forma limpia y prácticamente transparente para el desarrollador. Las cookies son pequeños datos en texto plano, identificados por un nombre, que son guardados en el computador del cliente con el objetivo de mantener configuraciones especiales para cada uno o realizar seguimiento de ellos. Cada cookie está asociada a un sitio web en particular y tiene un tiempo de expiración y debe ser enviada por el navegador en cada *request* para que el container las proporcione a cada servlet.

Una sesión permite identificar a un usuario a través de más de un *request*. Esto no es trivial dada la naturaleza *stateless* del protocolo HTTP[3] y se logra a través de varias estrategias. La más utilizada es a través de una cookie en el cliente, de nombre *jsessionid*,

[1] Corresponden a inconsistencias en el valor de una variable al ser leída y sobreescrita por dos threads en forma simultánea. La solución típica es crear una llave única de exclusión que puede ser obtenida por sólo un thread a la vez.

[2] Cuando se utilizan varias llaves para la exclusión de threads puede suceder que dos o más threads queden bloqueados en forma indefinida, cada uno en espera de una llave que posee otro.

[3] *Stateless*, que no guarda estados. Una solicitud HTTP es totalmente independiente de otra solicitud HTTP aunque sean consecutivas o del mismo cliente.

que guarda un identificador único de sesión que es recuperado en cada *request*. Otra opción es utilizar el mecanismo propio implementado en el protocolo HTTPS para identificar sesiones. Un último recurso para reconocer sesiones si no se permiten cookies y no se utiliza HTTPS, consiste en reescribir cada URL presentada al cliente agregando el parámetro *jsessionid* y el identificador de sesión, que debe ser mantenido a través de cada página.

La sesión permite manipular información como la fecha de creación, la de último acceso y el tiempo de expiración. Además permite compartir objetos entre múltiples conexiones de un usuario incluso en aplicaciones distribuidas en múltiples containers. Aunque para el caso de sesiones en aplicaciones distribuidas, cada objeto agregado a la sesión debe implementar la interfaz *java.io.Serializable* para poder ser transmitida a los restantes containers.

3.1.5. Filtros

Los filtros son características nuevas aparecidas en la versión 2.3 de la especificación de servlets, y permiten implementar un objeto para que ejecute un proceso previo o posterior a la ejecución de uno o más recursos web. Como recurso web se entiende un contenido estático como un archivo HTML, XML, JPG o GIF, o un contenido dinámico como un servlet o JSP.

Al declarar cada filtro se debe definir los recursos web con los que debe ejecutar, siendo posible utilizar caracteres globales para asociarlo todos o a un grupo de ellos. Incluso se puede crear una cadena de ejecución de filtros asociando más de uno al mismo recurso web.

Un filtro se puede utilizar, por ejemplo, para asegurar que un usuario ha sido identificado. Para esto, basta con implementar un filtro que compruebe la existencia de la sesión validada y asociarlo con todos los recursos del sitio en el descriptor de la aplicación con el patrón "*/**".

Otras posibilidades de uso de los filtros son por ejemplo conversión de imágenes, compresión de datos, triggers que actúen bajo alguna condición, cambios de tipos MIME, modificación de un documento XML, encriptación de datos y auditoría, entre otras.

3.1.6. Clases principales

Los objetos para la definición de servlets se encuentran divididos en dos paquetes: *javax.servlet.**, que proporciona clases necesarias para crear servlets genéricos, independientes del protocolo utilizado, y *javax.servlet.http.**, que proporciona las clases que definen un servlet específico para el protocolo HTTP.

- *Servlet* define la funcionalidad básica que tiene un servlet como es su ciclo de vida

(métodos *init*, *destroy*) y procesar *requests* (método *service*). Es implementado por la clase *GenericServlet*.

- *SingleThreadModel* es una interfaz utilizada para marcar los servlets que se desea que se ejecuten en forma secuencial por *requests* simultáneos, evitando posibles problemas de procesamiento paralelo.

- *HttpServlet* agrega la funcionalidad para procesar los variados tipos de *request* HTTP (principalmente los métodos *doGet*, *doPost* para procesar formularios GET y POST, respectivamente).

- *HttpServletRequest* proporciona información del *request* del cliente al servlet a través del protocolo HTTP. Se pueden obtener datos como encabezados, cookies y características genéricas como dirección del cliente y parámetros recibidos desde él (*getParameter*). Permite también asociar objetos (*setAttribute*) y acceder al objeto *HttpSession*.

- *HttpServletResponse* asiste a un servlet para enviar una respuesta al cliente a través de un canal de comunicación binario o de texto (*getWriter*). Provee también funcionalidad específica para respuestas HTTP como enviar encabezados, cookies (*addCookie*) y errores con códigos numéricos como 403 (*sendError*).

- *HttpSession* permite identificar un usuario a través de más de una página, para esto se le asigna un identificador único que se mantiene mientras el usuario navega en el sitio (*getId*). Permite asignar un tiempo máximo de inactividad y se le pueden asociar objetos para compartir entre servlets (*setAttribute* y *getAttribute*).

3.2. JavaServer Pages

JavaServer Pages (JSP) es una componente de J2EE para construir fácilmente aplicaciones con contenido web como HTML, DHTML, XHTML y XML, en forma dinámica, con gran poder y flexibilidad. JSP se basa en los siguientes conceptos:

- Plantillas o templates. Una parte importante del contenido esta compuesto por una plantilla. Típicamente en esta plantilla se encuentran fragmentos HTML o de texto.

- Contenido dinámico. JSP provee una forma simple de agregar datos dinámicos a cada plantilla al permitir incrustar instrucciones de programación en éste. El lenguaje es generalmente Java, aunque se puede utilizar otro lenguaje que sea soportado por el container JSP.

- Encapsulación de funcionalidad. JSP provee dos formas distintas para encapsular funcionalidad: componentes JavaBeans y bibliotecas de etiquetas o *taglibs*.

JavaServer Pages fue creado con el objetivo de proveer un método declarativo y centrado en la presentación para crear servlets. Además de los beneficios que proveen los servlets, JSP ofrece la posibilidad de desarrollar rápidamente servlets donde los contenidos y la presentación se encuentran separados, y reutilizar código con el uso de una arquitectura basada en componentes.

Una página JSP es un documento de texto que describe como procesar un *request* para crear una respuesta, mezclando datos estáticos de plantillas con acciones dinámicas. JSP ofrece además los siguientes beneficios principales:

- Separación de roles. JSP permite la separación de los roles de diseñador y desarrollador. El diseñador crea presentaciones para cada página ubicando contenidos estáticos y dinámicos pensando en los usuarios. El desarrollador escribe componentes que interactúen con los objetos del servidor para proveer la información y lógica del negocio.
- Reutilización de componentes. JSP permite la utilización de componentes reutilizables como JavaBeans, EJB y taglibs, lo cual se traduce en mayor eficiencia en el desarrollo y menor probabilidad de inconsistencias en una aplicación.

3.2.1. Container JSP

El container JSP es una entidad de nivel de sistema que proporciona la administración de páginas JSP y su ciclo de vida. Cada *request* enviado a una página JSP es capturado por el container JSP y entregado al objeto apropiado que la implementa.

Dada la relación existente entre JSP y servlets -cada JSP es traducido a un servlet-, el container de servlets es el mismo container JSP, el cual también es conocido como *container web*.

3.2.2. Ciclo de vida

El ciclo de vida de una página JSP está compuesto por dos fases: traducción y ejecución.

Traducción

En la fase de traducción, el container determina la clase servlet que implementa a la página JSP. La traducción desde una página JSP fuente al servlet que le corresponde puede suceder en cualquier instante entre la instalación de la página JSP en el container y la recepción de un *request* dirigido a ella. La forma exacta en que es realizada la traducción de

página JSP a servlet que la implementa es dependiente del container, aunque debe cumplir con los requisitos de clases dadas en la especificación [18, cap.8], como es crear un servlet que extienda de *HttpJspPage* e implementar el método *_jspService* con la funcionalidad de la página JSP.

Ejecución

En la fase de ejecución, el container utiliza la clase servlet creada para responder los requerimientos invocando el método *_jspService*. El container es responsable de instanciar todos los objetos necesarios para procesar el *request*, invocar el objeto que implementa la página JSP y luego debe enviar al cliente el objeto con la respuesta generada.

3.2.3. Sintaxis

Una página JSP contiene elementos y datos de la plantilla. Un elemento es una instancia de un tipo de elemento conocido para el container. Los datos de la plantilla es todo lo restante, es decir, cualquier cosa que el traductor de JSP no conozca, que son enviados al usuario sin modificaciones dentro de invocaciones a *out.write()* en el método *_jspService*. Existen tres tipos de elementos: Directivas, Acciones y Programación.

Directivas

Proveen información global para la fase de traducción de la página JSP. Tienen la sintaxis: `<%@ directiva %>`

Existen tres directivas: *page* que permite definir propiedades de la página como el juego de caracteres, bibliotecas a incluir y página de error, *taglib* que se utiliza para declarar cada biblioteca de tags a usar en la página, e *include* que permite agregar texto a la página en forma estática desde un archivo externo en la fase de traducción.

Acciones

Proveen información para la fase de procesamiento. La interpretación de una acción puede depender del *request* específico recibido. Una acción puede ser estándar, es decir definida en la especificación, o puede ser personalizada, es decir definida a través del mecanismo de extensión de tags. La sintaxis es similar a un documento XML: existe un tag de inicio que incluye su nombre más atributos, un cuerpo opcional y un tag de cierre. Por ejemplo:

```
<prefijo:nombretag atributo="valor">cuerpo</nombretag>
```

Programación

Permiten manipular objetos y ejecutar código que afecte el contenido generado. Existen tres tipos de elementos de programación: Declaración, Scriptlets y Expresiones.

1. Declaración. Su sintaxis es `<%! ... %>`. Son utilizados para declarar variables y métodos para la página. El cuerpo de este elemento es traducido como instrucciones globales a la clase *HttpJspPage*. Por ejemplo:

   `<%! int n; %>`, declara una variable global a la página.

   `<%! public int abs(int i){ if(n<0) return -n; else return n; }%>`, declara la función global *abs* como global para la página.

2. Scriptlets. Su sintaxis es `<% ... %>`. Pueden contener cualquier fragmento de código del lenguaje de programación de la página que haya sido declarado en la directiva *page*. Los scriptlets se ejecutan en cada proceso de *request* y pueden modificar objetos, declarar variables, llamar funciones o cualquier otra acción que proporcione el lenguaje. El cuerpo de este elemento es copiado al método *_jspService*.

3. Expresiones. Su sintaxis es `<%= ... %>`. Corresponde a una expresión del lenguaje de programación cuyo resultado es evaluado como un string y agregado a la respuesta. El cuerpo de este elemento es copiado al método *_jspService* dentro de una instrucción *out.write()*. Por ejemplo:

   `N es un número <%= N>=0?"positivo":"negativo" %>`, escribe si N es positivo o negativo.

Es necesario señalar que si bien las directivas de programación permiten una gran flexibilidad para la creación de una página JSP, tienden a aumentar rápidamente la complejidad de éstas y de la lógica de presentación, dificultando la mantención de la aplicación y la separación de roles. Como regla general, se debe minimizar la utilización de estas directivas en favor de las componentes más especializadas como JavaBeans o taglibs donde el código puede ser centralizado y reutilizado.

3.2.4. Clases importantes

Las clases específicas a la creación de páginas JSP se encuentran en el paquete *javax.servlet.jsp.**, donde se detallan las propiedades que debe cumplir un container para traducir una página JSP.

- *JspPage* extiende la interfaz *javax.servlet.Servlet* y define la interacción genérica entre la implementación de una página JSP y el container. Se definen los métodos *jspInit* y *jspDestroy* que son invocados para inicializar o eliminar los valores específicos a la página y pueden ser sobreescritos a través de elementos de declaración en el JSP.
- *HttpJspPage* extiende el objeto *JspPage* y es la clase que contiene la implementación en servlet de la página JSP en el método *_jspService*.
- *PageContext* permite acceder a todos los objetos asociados con la página JSP creando una capa sobre los detalles de la implementación del container. Da acceso a los objetos de *request* (*getRequest*), de la sesión (*getSession*) y obtener un canal de salida al cliente (*getOut*), entre muchos otros.

3.3. Enterprise JavaBeans

Enterprise JavaBeans (EJB) es una arquitectura que permite la creación de componentes de aplicaciones distribuidas y orientadas a transacciones. Las aplicaciones escritas utilizando EJB son escalables, transaccionales y multiusuarios.

Las características esenciales de EJB son:

- Contiene la lógica del negocio que opera con el *Enterprise Information System (EIS)*.
- Las instancias son creadas y manejadas por el container EJB.
- Puede ser configurado editando sus parámetros de entorno vía archivos XML.
- Las características de seguridad y transacciones se encuentran separadas de las clases EJB, lo que permite la operación de aplicaciones externas y middlewares.

3.3.1. Objetivos

La arquitectura EJB tiene variados objetivos en su especificación:

- Ser el estándar para arquitectura de componentes en la construcción de aplicaciones distribuidas orientadas a objetos escritas en Java.
- Facilitar la creación de aplicaciones, ya que los desarrolladores no deberán preocuparse de conceptos de bajo nivel como manejo de estados y transacciones, administración de recursos, multi-threading y otros.

- Cumplir la filosofía de Java de desarrollar una vez y luego instalar en múltiples plataformas sin necesidad de efectuar modificaciones o recompilaciones.
- Definir un contrato que permita crear herramientas de distintos proveedores para el desarrollo e instalación de componentes.
- Ser compatible con el protocolo CORBA, y así tener posibilidad de integración con múltiples sistemas.

3.3.2. Tipos de Enterprise Beans

La arquitectura de EJB define tres tipos diferentes de objetos enterprise beans:

- *Session beans*. Modelan la lógica de los procesos de negocio, es decir, modelan acciones como por ejemplo la lógica de calcular precios, transferir fondos entre cuentas, ejecutar una orden de compra, etc. Se ejecutan en representación de un único cliente.
- *Entity beans*. Contienen el modelo de datos del negocio y la lógica interna de los datos como por ejemplo un producto, una orden, un empleado, la lógica del cambio de nombre de un cliente, reducir la cantidad de dinero de una cuenta, etc. Su tiempo de vida es tan largo como los datos en el sistema de almacenamiento que representan.
- *Message-driven beans*. Modelan acciones, pero sólo se ejecutan luego de recibir un mensaje. Contienen la lógica de procesar un mensaje en forma asíncrona como puede ser recibir un mensaje con la necesidad de actualizar el stock de cierto producto e invocar el session bean que se encargan de solucionarlo.

3.3.3. Container EJB

El cliente nunca invoca directamente los métodos que poseen las instancias de los enterprise beans, si no que la invocación es recibida por el container EJB y luego delegada a la instancia. Al recibir las ejecuciones, el container EJB simplifica el trabajo de un desarrollador -también conocido como *Bean Provider*- al tener la posibilidad de ejecutar automáticamente tareas de middleware en forma implícita como:

- Administración de transacciones: El container EJB provee de un servicio de transacciones, que es expuesto a través de una API de alto nivel conocida como *Java Transaction API (JTA)* (ver sección 5.5).
- Seguridad: Puede autenticar y autorizar cada usuario, a través de roles definidos en XML y/o utilizando JAAS (ver sección 5.7).

- Persistencia: Automáticamente guarda cualquier objeto persistente al sistema de almacenamiento, de donde puede recuperarlos cuando sea necesario.
- Accesibilidad remota: El container EJB permite a los objetos acceso a servicios de red, sin tener que ser programado completamente por el desarrollador, si no que sólo definiendo interfaces de acceso remoto.
- Acceso concurrente: El container EJB automáticamente maneja invocaciones concurrentes desde los clientes, asegurando que un cliente ejecutará un bean a la vez, creando una cola de espera para cada bean o instanciando múltiples objetos, evitando problemas de sincronización de *threads*.
- Monitoreo: El container EJB puede realizar un seguimiento de los métodos que son invocados y mostrar información de desempeño en tiempo real que apoye la administración del sistema.

La capa intermedia que proporciona el container EJB, es representada por el objeto *EJBObject* o por *EJBLocalObject* que debe extender toda interfaz que defina métodos a ser utilizados por el cliente. Un ejemplo de su uso se ilustra en la figura 3.1.

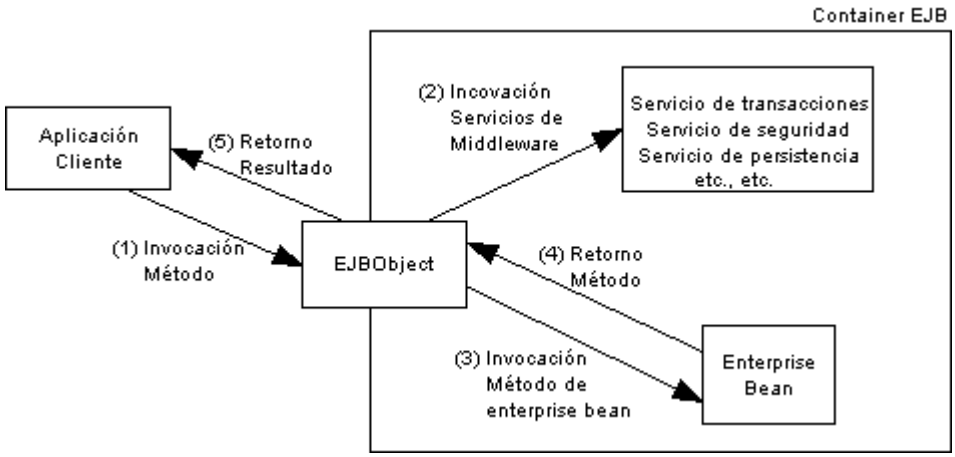

Figura 3.1: EJBObject

Además de estas tareas, el container EJB realiza la administración de las instancias existentes para los objetos, es decir, administra los diferentes ciclos de vida.

3.3.4. Session Beans

Los session beans son objetos no persistentes que implementan la lógica del negocio que se ejecuta en el servidor. Es posible pensar que un session bean es una extensión lógica del programa cliente que se ejecuta en el servidor y contiene información específica al cliente.

El tiempo de vida es limitado por la duración de la sesión del cliente, por lo cual no son persistentes en el tiempo.

Cada session bean mantiene una interacción con un cliente que se desarrolla a través de la ejecución de los distintos métodos que provee. Existen dos tipos de session beans que varían en la forma de modelar esta interacción: *stateless* y *stateful*.

Debido a su tiempo de vida reducido y a la no persistencia de sus datos, un session bean -ya sea *stateless* o *stateful*- no sobrevive a fallas en el container o en el servidor. En este caso el bean es eliminado de la memoria, siendo necesario que el cliente vuelva a iniciar una conexión para poder continuar con su uso.

Stateless Session Beans

Están pensados para modelar los procesos de negocios que tienden naturalmente a una única interacción, por tanto no requieren de mantener un estado entre múltiples invocaciones. Después de la ejecución de cada método, el container puede decidir mantenerlo, destruirlo, limpiar toda la información resultante de ejecuciones previas, o reutilizarlo en otros clientes. La acción a tomar depende de la implementación del container.

Un stateless session bean sólo debe contener información que no es específica a un cliente como una referencia a una fuente de recursos, que es guardada en una variable privada y que puede ser eliminada en cualquier instante por el container. Por tanto sólo puede definir un método sin parámetros para su creación, llamado *ejbCreate()*, ya que no es necesario que reciba un valor del cliente para ser inicializado.

Los métodos independientes y que están determinados sólo por los parámetros entregados por el cliente son candidatos a ser representados en este tipo de bean. Por ejemplo un método que realice un cálculo matemático con los valores entregados y retorne el resultado, o un método que verifique la validez de un número de tarjeta de crédito son posibles métodos implementables por stateless session beans.

Ciclo de Vida

El ciclo de vida de un stateless session bean se inicia cuando el container decide crear una nueva instancia, que depende de la implementación del container y puede variar desde instanciar al recibir la primera solicitud al bean o instanciar N beans al iniciar la aplicación.

Una vez instanciado, el container ejecuta el método *setSessionContext* con el objetivo de entregar al bean una referencia al container y sus configuraciones y luego ejecuta el método *ejbCreate* para ser inicializado, por ejemplo obtener una referencia a una base de datos.

En este instante el bean se encuentra en estado *ready* en el cual recibe y ejecuta las invocaciones a sus métodos, donde en cada ejecución puede ser asignado a clientes diferentes según la implementación del container.

Una vez que el container decide que la instancia debe ser eliminada, se ejecuta el método *ejbRemove* para permitir que libere los recursos externos que utilice y termine en forma correcta su ejecución. Ver figura 3.2.

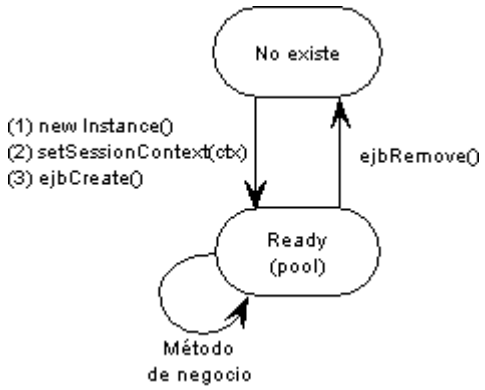

Figura 3.2: Ciclo de vida de stateless session beans

Debido a que no mantienen estado de interacciones, todas las instancias de la misma clase son equivalentes e indistinguibles para un cliente, sin importar quien utilizó un bean en el pasado. Por tanto pueden ser intercambiados entre un cliente y otro en cada invocación de un método, creando un *pool* de stateless session beans. Ver figura 3.3.

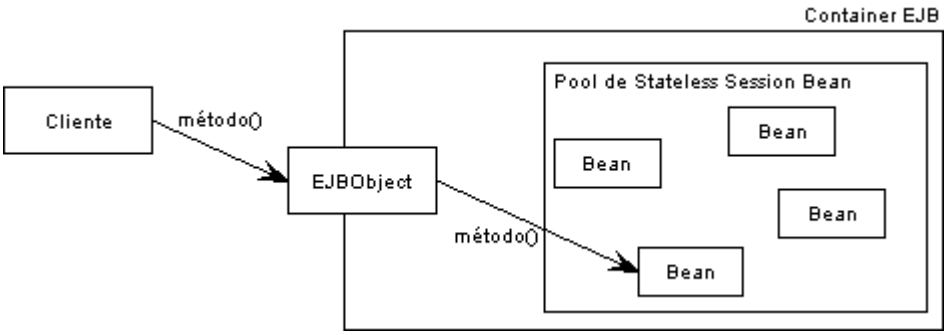

Figura 3.3: Pool de stateless session beans

Sin embargo, es necesario señalar que sólo un cliente puede invocar una instancia a la vez, esto es, el container no debe permitir la ejecución de múltiples *threads* sobre una misma instancia con el objeto de evitar posibles problemas derivados del procesamiento paralelo.

Stateful Session Beans

Están diseñados para servir procesos de negocio que abarcan múltiples llamados a funciones o transacciones. Para lograr esto es necesario guardar el estado en que se encuentra el bean luego de cada ejecución del cliente, el cual se mantiene y actualiza para cada nueva invocación del mismo cliente.

Un ejemplo de stateful session bean es implementar un carro de compras donde el cliente selecciona variados ítemes que deben ser agregados a él. Para realizar este proceso es necesario que una componente deba mantenerse ligada a un cliente en particular entre distintos *requests*, actualizando su estado cuando corresponda.

Ciclo de vida

Al no poder ser intercambiados entre clientes, los stateful session beans deben mantenerse en espera del *request* del cliente al cual corresponden, sin poder ser reutilizados en este período. Para disminuir este derroche de recursos sin utilización se crean los conceptos de estado *activo* y estado *pasivo*.

Para limitar el número de instancias de stateful session beans en la memoria, el container tiene la capacidad de utilizar el protocolo de serialización de Java para convertirlo en un conjunto de datos almacenables en una unidad externa. Luego, para que el bean pueda eliminar recursos que luego puede volver a adquirir como referencias a bases de datos o sockets, el container invoca el método *ejbPassivate*. Una vez terminado este método, el bean entra al estado *pasivo* y la instancia puede ser reinicializada o eliminada.

Cuando un bean se encuentra *pasivo* y el container desea transformarlo en *activo*, debe leer la información guardada para restaurar su estado original y luego invocar el método *ejbActivate* para recuperar los recursos eliminados previamente.

El algoritmo para decidir cuando un bean debe cambiar de estado es dependiente del container. Normalmente se utiliza una estrategia *Least Recently Used (LRU)* que simplemente elige el bean con más tiempo inactivo para cambiar su estado a *pasivo*. La única excepción a esta regla es cuando el bean se encuentra realizando una transacción ya que no puede cambiar de estado mientras ésta no finalice. Después, el algoritmo utilizado normalmente para activar un bean pasivo es esperar que el cliente correspondiente invoque uno de sus métodos [6, pág.86]. El ciclo de vida de un stateful session bean se puede ver en la figura 3.4.

Al mantener un estado interno que representa a un cliente, un stateful session bean no puede ser intercambiado entre clientes, lo que imposibilita un *pool* de stateful session beans.

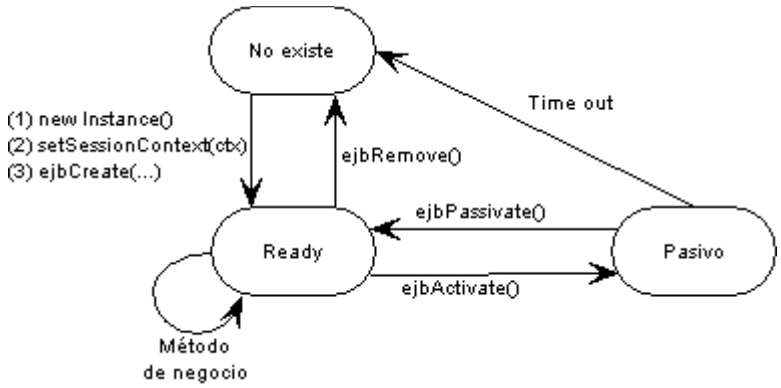

Figura 3.4: Ciclo de vida de stateful session beans

Implementación de session beans

La implementación de un session bean stateless o stateful está compuesta de al menos tres archivos diferentes: dos interfaces y una clase.

La primera interfaz debe extender de *java.ejb.EJBObject* que define la funcionalidad que será utilizada por el cliente. Este será el objeto que el cliente obtendrá para ejecutar los métodos del session bean aquí definidos. *EJBObject* es una interfaz remota de RMI al extender de *java.rmi.Remote*, para así proveer al session bean la capacidad de ejecución distribuida en múltiples máquinas. Por ejemplo:

```
public interface Saludo extends javax.ejb.EJBObject {
  public String hola() throws java.rmi.RemoteException;
}
```

La segunda interfaz debe extender de *java.ejb.EJBHome* que define los métodos "administrativos" que puede utilizar el cliente como son los métodos de creación *create()* y de eliminación de beans *remove()*. *EJBHome* extiende de *java.rmi.Remote* por lo cual también es una interfaz remota con posibilidades de ejecución distribuida. El cliente obtendrá el objeto *EJBHome* desde el container a través de JNDI y luego de la ejecución de uno de los métodos *create* definidos obtendrá una instancia de la interfaz creada con *EJBObject*. Por ejemplo:

```
public interface SaludoHome extends javax.ejb.EJBHome {
  Saludo create(String nombre) throws java.rmi.RemoteException,
                                      javax.ejb.CreateException;
}
```

La clase a crear debe implementar la interfaz *javax.ejb.SessionBean* y debe contener la implementación de los métodos definidos en las dos interfaces previas. Por ejemplo:

```
public class SaludoBean implements javax.ejb.SessionBean {
  private SessionContext ctx;
  public String nombre;

  //Implementación de los métodos de SessionBean
  public void ejbRemove() {
  }
  public void ejbActivate() {
  }
  public void ejbPassivate() {
  }
  public void setSessionContext(javax.ejb.SessionContext ctx) {
    this.ctx = ctx;
  }

  //Implementación de los métodos create declarados en la interfaz SaludoHome (EJBHome)
  public void ejbCreate(String nombre) {
    this.nombre=nombre;
  }

  //Implementación de los métodos declarados en la interfaz Saludo (EJBObject)
  public String hola() {
    return "Hola " + nombre + "!";
  }
}
```

El nombre del método *void ejbCreate(String nombre)* no corresponde directamente con el método *Saludo create(String nombre)*. Esto sucede porque el cliente al llamar a *create* no ejecuta directamente *ejbCreate* si no que ejecuta una implementación del container donde agrega los servicios de middleware mencionados en la sección 3.3.3 y luego delega su ejecución en el método *ejbCreate* correspondiente. Así mismo sucede con los métodos definidos en la interfaz que extiende de *EJBObject*.

Como las dos interfaces necesarias extienden de *java.rmi.Remote*, toda llamada al bean es potencialmente una invocación remota. Si bien, tiene el beneficio de que una aplicación puede ser instalada en un servidor o en varios servidores en forma distribuida sin modificar el código fuente, tiene una gran desventaja en desempeño si la aplicación se encuentra instalada en un sólo servidor y en la misma JVM. En la versión 2.0 de la especificación de EJB se han agregado dos nuevas interfaces llamadas *locales*, que no extienden de *java.rmi.Remote* si no que entregan sus parámetros por referencia dentro de la misma JVM, proporcionando la misma funcionalidad pero eliminando el costo de serializar y des-serializar los datos. Por ejemplo, la interfaz local sería:

```
public interface SaludoLocal extends javax.ejb.EJBLocalObject {
  public String hola();
}
```

Y la interfaz local de *EJBHome*:

```
public interface SaludoLocalHome extends javax.ejb.EJBLocalHome {
  SaludoLocal create(String nombre) throws javax.ejb.CreateException;
}
```

La implementación del session bean no debe ser modificada, si no sólo la aplicación cliente que ahora puede utilizar las interfaces locales si desea obtener los beneficios de encontrarse en la misma JVM.

El diagrama de las clases e interfaces involucradas en el ejemplo se puede ver en la figura 3.5. Las clases e interfaces han sido divididas según quien las proporciona, en primer lugar se encuentran las interfaces de J2SE, luego las de J2EE y utilizando éstas se encuentran las creadas en el ejemplo. Las interfaces ahí definidas son implementadas por clases generadas automáticamente por el container que serán los objetos que obtendrá y ejecutará el cliente.

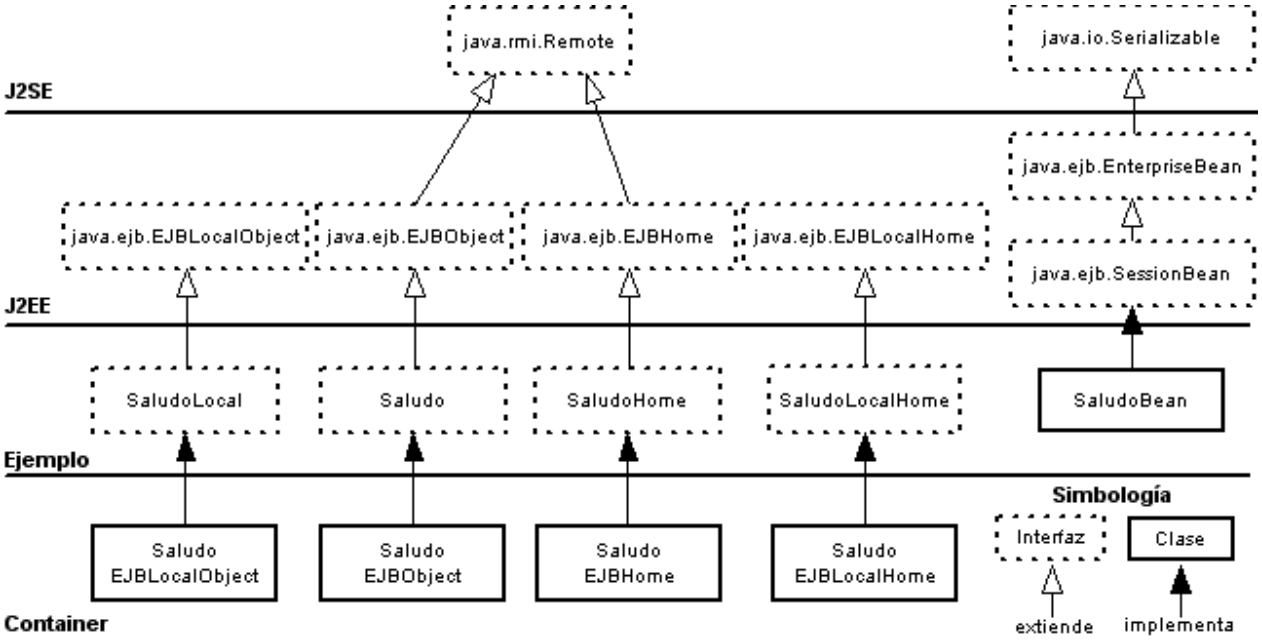

Figura 3.5: Diagrama de clases para session beans

Cabe señalar que el session bean implementado en el ejemplo debe ser declarado como tipo *stateful*, ya que una vez creado la invocación de sus siguientes métodos deben estar ligados al mismo cliente, si no el método *hola()* retornaría un nombre equivocado correspondiente a otro cliente.

Descriptor

Luego para instalar el bean en el container es necesario declararlo en el descriptor de nombre *ejb-jar.xml*. En este archivo se describen todos los enterprise beans que utilizará el container. Dentro de ese archivo debe detallarse el elemento `<session>`:

```
<ejb-jar>
  <enterprise-beans>
   ...
    <session>
      <description>Bean de prueba</description>
      <ejb-name>Saludo</ejb-name>
      <home>SaludoHome</home>
      <remote>Saludo</remote>
      <local-home>SaludoLocalHome</local-home>
      <local>SaludoLocal</local>
      <ejb-class>SaludoBean</ejb-class>
      <session-type>Stateful</session-type>
      <transaction-type>Container</transaction-type>
    </session>
   ...
  </enterprise-beans>
</ejb-jar>
```

En el código se declaran los datos principales del bean como el nombre que lo identificará, los nombres de las cuatro interfaces y de la clase que lo implementa, el tipo de session bean -stateless o stateful- y la forma en que manejará las transacciones -Container o Bean- (ver sección 5.5).

Ejecución

Un extracto de un programa cliente debe contener las siguientes líneas:

```
Context ctx = new javax.naming.InitialContext();
SaludoHome home = (SaludoHome)javax.rmi.PortableRemoteObject.narrow(
                                ctx.lookup("jndi/Saludo"), SaludoHome.class);
Saludo bean=home.create("Juan");
System.out.println(bean.hola());
```

La llamada a la función *narrow* es necesaria para la operabilidad con CORBA (ver sección 5.3). Si se decide utilizar las interfaces locales, el código sería el siguiente:

```
Context ctx = new javax.naming.InitialContext();
SaludoLocalHome home = (SaludoLocaHome)ctx.lookup("jndi/Saludo");
SaludoLocal bean=home.create("Juan");
System.out.println(bean.hola());
```

Se puede ver un diagrama de la ejecución es este código en la figura 3.6.

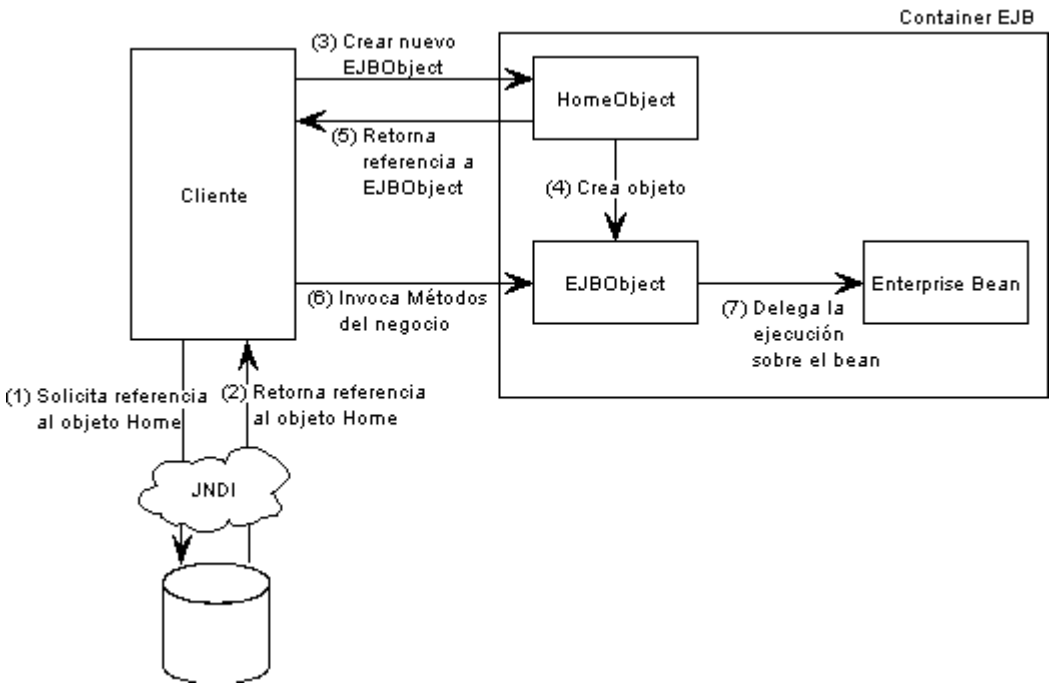

Figura 3.6: Ejecución del programa cliente

3.3.5. Entity Beans

Los entity beans otorgan una vista de objeto Java a los datos del negocio guardados en una unidad de almacenamiento. Permiten un acceso compartido de múltiples usuarios y tienen un tiempo de vida independiente de la duración de las sesiones de los clientes.

Los entity beans son componentes de datos que conocen como utilizar una unidad de almacenamiento para mantener la información que guardan en forma persistente. Por tanto proporcionan una capa que envuelve el almacenamiento de los datos simplificando la tarea de su acceso y manipulación.

Dependiendo del enfoque utilizado para ejecutar la persistencia de datos, los entity beans son divididos en dos grupos: *Bean-Managed Persistence (BMP)* y *Container-Managed Persistence (CMP)*. Si bien utilizan métodos de persistencia diferentes sus propiedades son similares.

Los entity beans se utilizan para dar una visión y un acceso orientado a objetos de una base de datos relacional. Para esto debe decidirse la *granularidad* del modelo, es decir el nivel de detalle de los objetos entity beans a crear. Esto es discutido en la sección 6.7.2.

Estructura

La estructura utilizada para su creación es similar a los session bean, es decir, cuenta con una interfaz remota o local y con una interfaz *home* que también puede ser remota o local. Las interfaces de acceso proporcionarán funcionalidades específicas a cada instancia como leer y asignar valores a los campos, llamados *getter* y *setter*, y obtener y modificar las relaciones con otros entity beans. Los métodos de una clase que son independientes de una instancia son declarados en la interfaz *home* como métodos para crear instancias, métodos para buscar instancias y otros independientes de ellas como por ejemplo el cálculo del promedio del valor de un campo en particular de las instancias.

Se define una nueva clase a implementar llamada *Primary Key (PK)*, que tiene la misión de guardar la información necesaria para identificar en forma única una instancia de un entity bean, que puede contener el número de campos necesarios y del tipo que desee, siempre que sea serializable ya que es enviada a través de la red en el caso de aplicaciones distribuidas.

Propiedades

Los entity beans son objetos duraderos en el tiempo, es decir sobreviven a fallas en el container y el servidor porque son sólo representaciones de datos existentes en una unidad de almacenamiento permanente y tolerante a fallas. Si falla la máquina donde se encuentra el servidor, los entity beans podrán ser reconstruidos en la memoria leyendo la información que existe en la base de datos, que corresponde a la última transacción finalizada con éxito.

Los entity beans representan una forma de visualizar la información en la base de datos, por lo que la modificación de una instancia debe significar una modificación en la base de datos y viceversa. El container está encargado de efectuar la sincronización de datos invocando los métodos *ejbLoad* y *ejbStore* cuando corresponda. La implementación de estos métodos está a cargo del desarrollador para beans tipo BMP, o a cargo del container para beans tipo CMP. Una forma de modificar los entity beans es modificar directamente la base de datos y esperar que el container actualice la información a las instancias, sin embargo se pueden producir problemas de inconsistencia si el container utiliza un *caché* de datos.

Al igual que los session beans, el container no debe permitir la ejecución de *threads* sobre una misma instancia de un entity bean debido a la alta probabilidad de generar problemas de concurrencia y la dificultad para manejar una transacción. Sin embargo, si el container mantiene una cola de espera para utilizar un bean en forma secuencial puede causar grandes problemas de desempeño. Una solución para el acceso secuencial es crear varias instancias de la misma clase que representen la misma información, así cada cliente puede acceder en forma independiente en paralelo. Naturalmente eso tiene un problema, ya que se puede generar corrupción de datos o inconsistencias al duplicar la información, sin embargo esto es solucionado al hacer uso de transacciones para aislar las acciones de los clientes. Las transacciones dan la ilusión a un cliente que tienen acceso exclusivo a los datos cuando en

realidad los datos son compartidos por muchos usuarios (ver sección 5.5).

Al no permitir que dos *threads* ejecuten en una misma instancia se genera un error en casos con llamadas circulares. Por ejemplo, un cliente ejecuta un entity bean A, A llama un método en el entity bean B, y luego B llama un método en A, todo dentro de la misma transacción (ver figura 3.7). En este caso el container ve que ha ingresado un nuevo *thread* sobre A y lanza una excepción terminando con la ejecución. Sin embargo, en algunas situaciones este comportamiento no es el deseado ya que no son dos threads los que ejecutan simultáneamente, si no que es el mismo thread el que ingresa dos veces. Para permitir estas ejecuciones circulares, un entity bean puede ser declarado como *re-entrante*. Dado que el container no puede, en general, diferenciar un ciclo legal de un mismo cliente sobre un entity bean reentrante con una llamada concurrente ilegal de otro cliente sobre el mismo bean, hay que tener la precaución de crear código que permita manejar multiples threads y en lo posible no utilizar entity beans *re-entrantes*.

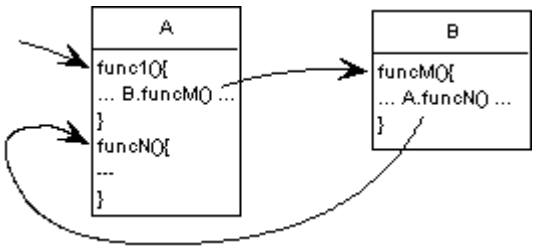

Figura 3.7: El objeto A es reentrante

Ciclo de Vida

Cuando el cliente desea crear un nuevo entity bean significa que desea crear un nuevo registro en la base de datos. Para esto invoca el método *create* que lo toma el container y lo delega al método *ejbCreate*. El método *ejbCreate* toma los parámetros recibidos, los utiliza para asignarlos a sus atributos y debe retornar al container el objeto *PK* que lo identifica en forma única. Con la llave primaria el container puede crear el entity bean que es el que retorna al cliente.

Cuando el cliente desea eliminar un registro debe invocar el método *remove* que es recibido por el container, el que delega la ejecución sobre *ejbRemove* que debe eliminar el registro de la base de datos y preparase para ser borrado por el recolector de basura. Sin embargo el container no debe necesariamente permitir eliminar la instancia existente, si no que puede agregarla al *pool* de instancias para ser reutilizada.

Para llevar a cabo esta reutilización es necesario crear o eliminar cualquier tipo de recursos externos que utilice la implementación del entity bean que no esté ligada a la base de datos como archivos o sockets. Por esto cuando el container decide utilizar una instancia

del *pool* debe ejecutar del método *ejbActivate* para que el bean obtenga los recursos que necesite y luego el método *ejbLoad* para obtener los datos. Cuando el container deposita una instancia en el *pool* debe invocar el método *ejbStore* para guardar su estado y luego *ejbPassivate* para que liberar sus recursos.

Como los entity beans están identificados únicamente por su llave primaria, se pueden implementar búsquedas de datos llamadas *finders* que son análogas a ejecutar sentencias *select* en SQL. Debido a que las búsquedas no están asociadas a ninguna instancia en particular son proporcionadas al cliente a través de la interfaz *home* del entity bean. Similar a los métodos de creación, los *finders* retornan un objeto PK o un conjunto de ellos (*java.util.Collection*) que utiliza el container para crear la o las instancias que son retornados al cliente. Para el caso de beans CMP, se permite crear métodos *ejbSelect* que son similares a los *finders* salvo que son internos a una clase. El ciclo de vida que es administrado por el container para entity beans se muestra en la figura 3.8.

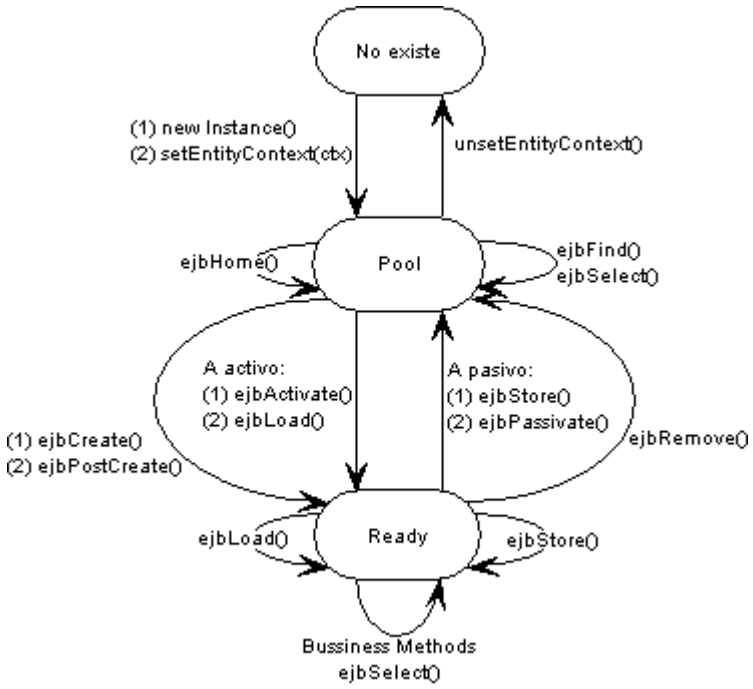

Figura 3.8: Ciclo de vida de un entity bean

Entity Beans BMP

Son los entity beans en los cuales el desarrollador se encarga de implementar la comunicación con la base de datos. Para esto es necesario tener acceso a ella a través de JDBC, y probablemente sea necesario instalar drivers específicos al proveedor y estar acostumbrado a usar esta API.

La estructura es similar a los session beans: una interfaz de usuario local o remota que extiende de *javax.ejb.EJBObject* o *javax.ejb.EJBLocalObject*, una interfaz home local o remota que extiende de *javax.ejb.EJBHome* o *javax.ejb.EJBLocalHome*, y una clase que implementa los métodos definidos que extiende de *javax.ejb.EntityBean*. Además es necesario implementar la llave primaria de la clase que debe extender de *java.io.Serializable*.

La interfaz de usuario debe contener los métodos getter y setter de sus campos y relaciones, más los métodos específicos a una instancia.

```
public interface Persona extends javax.ejb.EJBObject {
  //método especifico de la instancia
  public int getLargoNombre() throws java.rmi.RemoteException;
  //getter/setters
  public String getNombre() throws java.rmi.RemoteException;
  public void setNombre(String nombre) throws java.rmi.RemoteException;
  public int getEdad() throws java.rmi.RemoteException;
  public void setEdad(int edad) throws java.rmi.RemoteException;
}
```

La interfaz home contiene los create, los finders y los métodos independientes de cada instancia:

```
public interface PersonaHome extends javax.ejb.EJBHome {
  //creates
  Persona create(String personaId, int edad) throws
           javax.ejb.CreateException, java.rmi.RemoteException;
  //finders
  public Persona findByPrimaryKey(PersonaPK key) throws
           javax.ejb.FinderException, java.rmi.RemoteException;
  public Collection findByNombre(String nombre) throws
           javax.ejb.FinderException, java.rmi.RemoteException;
  //independiente de las instancias
  public int numeroPersonas() throws java.rmi.RemoteException;
}
```

Las clases locales son análogas, sólo hay que modificar la clase de la cual extienden y eliminar la excepción *java.rmi.RemoteException*.

Luego es necesario definir la llave primaria, que debe implementar la interfaz *java.io.Serializable* e implementar los métodos *toString*, *equals* y *hashCode*. En este caso definiremos como llave primaria un identificador correlativo que debe ser entregado como parámetro.

```
public class PersonaPK implements java.io.Serializable {
  //campo
```

```
  public String personaId;

  //constructor
  public PersonaPK(String id){
    this.personaId=id;
  }
  public PersonaPK(){
  }
  public String toString(){
    return this.personaId;
  }
  public int hashCode(){
    return this.personaId.hashCode();
  }
  public boolean equals(Object persona){
    return ((PersonaPK)persona).personaId.equals(this.personaId);
  }
}
```

Si bien en este caso es opcional crear una clase para hacer de identificador -ya que puede declararse directamente *java.lang.String* como clase PK- es recomendable crear siempre una clase especial, porque separa la utilización del identificador único de su implementación simplificando la actualización del sistema en el caso que ocurran cambios en la PK.

Finalmente, es necesario crear la clase que implementa los métodos. Hay que recordar que la invocación de un método por parte del cliente es capturada por el container el cual ejecuta tareas de middleware para luego delegarla sobre el bean, por lo que el nombre de los métodos declarados en las interfaces de cliente no corresponden a los implementados en el bean si no que estos últimos deben contener el prefijo *ejb*. Se utilizará *pseudo-código* con el objeto de no extender la implementación del bean, en especial en el manejo de JDBC.

```
public class PersonaBean implements javax.ejb.EntityBean {
  private EntityContext ctx;

  //campos de estado
  private String personaId;
  private String nombre;
  private int edad;

  //constructor
  public AccountBean() {
  }

  //metodos del negocio
  public int getLargoNombre(){
    //calculo largo del nombre
    //retorno
  }

  //getter/setter
```

```
public String getNombre(){
  return this.nombre;
}
public void setNombre(String nombre){
  this.nombre=nombre;
}
public int getEdad(){
  return this.edad;
}
public void setEdad(int edad){
  this.edad=edad;
}

//independiente de una instancia
public int ejbHomeNumeroPersonas(){
  //obtención conexión BD
  //ejecución query SELECT correspondiente en la BD
  //devolver conexión BD
  //retornar resultado
}

//requerido por el Container-EJB
public void ejbActivate() {
}
public void ejbPassivate() {
}
public void ejbRemove() throws RemoveException {
  //obtención identificador único de la instancia
   PersonaPK pk = (PersonaPK) ctx.getPrimaryKey();
   String id=pk.personaId;
  //obtención conexión BD
  //ejecución query DELETE con el identificador id
  //devolver conexión BD
}
public void ejbLoad() {
  //obtención identificador único de la instancia
  //obtención conexión BD
  //ejecución query SELECT con el identificador id
  //asignar datos obtenidos a campos del bean
  //devolver conexión BD
}
public void ejbStore() {
  //obtención conexión BD
  //ejecución query UPDATE con el identificador personaId
  //devolver conexión BD
}
public void setEntityContext(EntityContext ctx) {
  this.ctx=ctx;
}
public void unsetEntityContext() {
  this.ctx=null;
}

//create
```

```
  public PersonaPK ejbCreate(String personaId,String nombre) throws CreateException{
    //asignar a los campos los parámetros y valores por defecto
     this.personaId=personaId;
     this.nombre=nombre;
     this.edad=0;
    //obtención conexión BD
    //ejecución query INSERT con el identificador personaId
    //devolver conexión BD
    //retornar llave primaria
     return new PersonaPK(personaId);
  }

  //finders
  //el método ejbFindByPrimaryKey es obligatorio para todo entity bean
  public PersonaPK ejbFindByPrimaryKey(PersonaPK key) throws FinderException {
    //obtención conexión BD
    //ejecución query SELECT con el identificador id
    //si no hay datos retornar una excepción
    //devolver conexión BD
    //retornar key
  }
  public Collection ejbFindByNombre(String nombre) throws FinderException {
    //obtención conexión BD
    //ejecución query SELECT con el parámetro nombre
    //con cada registro crear un PersonaPK y agregarlo a un Vector
    //devolver conexión BD
    //retornar el Vector
  }
}
```

Finalmente es necesario declarar el entity bean en el container. Para esto se utiliza la etiqueta `<entity>` en el archivo *ejb-jar.xml*. El descriptor a definir es el siguiente:

```
<ejb-jar>
  <enterprise-beans>
   ...
    <entity>
      <ejb-name>Persona</ejb-name>
      <home>PersonaHome</home>
      <remote>Persona</remote>
      <local-home>PersonaLocalHome</local-home>
      <local>PersonaLocal</local>
      <ejb-class>PersonaBean</ejb-class>
      <persistence-type>Bean</persistence-type>
      <prim-key-class>PersonaPK</prim-key-class>
      <reentrant>False</reentrant>
      <resource-ref>
        <res-ref-name>jdbc/ejbPool</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
    </entity>
```

```
   ...
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>Persona</ejb-name>
        <method-intf>Local</method-intf>
        <method-name>*</method-name>
      </method>
      <method>
        <ejb-name>Persona</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Se comienza dando los datos del bean como su nombre, las interfaces y la clase que lo implementa. Después se declara el tipo de persistencia a utilizar (en este caso *Bean* porque es BMP), la clase que será la llave primaria y si es reentrante o no. Luego se declara una referencia a una fuente de datos externa que será un pool de conexiones, el nombre JNDI para poder utilizarlo, el tipo de clases que contiene y quién realiza la autorización para poder utilizarla (el container o la aplicación).

El elemento `<assembly-descriptor>` permite definir características como roles de seguridad, permisos de ejecución y las características para transacciones manejadas por el container. En este caso se define que todos los métodos de la interfaz local y de la interfaz remota deben soportar el tipo de transacción *Required* (ver sección 5.5).

Entity Beans CMP

Son los entity beans en los cuales el container se encarga de implementar la persistencia en una base de datos. Permiten separar lo que es la clase entity bean con su lógica, de lo que es su "representación persistente". Con esto se elimina de la implementación del bean una gran cantidad de código -principalmente JDBC- a costo de definir en diferentes descriptores XML la forma en que el container debe llevar a cabo la persistencia. La definición de CMP tuvo grandes cambios en la especificación de EJB 2.0, agregando y mejorando funcionalidades como manejo de relaciones y EJB QL.

Un entity bean CMP está compuesto de cinco partes:

- La interfaz de cliente (local o remota) que contiene la definición de los métodos específicos a una instancia.

- La interfaz home (local o remota) con la definición de *finders* y métodos no específicos a una instancia que utilizará el cliente.
- La clase que será utilizada para guardar el identificador único del bean, que puede ser una clase nueva o una ya existente en el lenguaje.
- La clase entity bean con las implementaciones de los métodos declarados en las interfaces previas, salvo para getters, setters y finders que serán implementados por el container.
- El *esquema de persistencia abstracto* que contiene el detalle la persistencia del bean en la base de datos a través de descriptores XML.

Para separar la clase entity bean de su representación persistente, el desarrollador debe definir la clase entity bean como una clase abstracta que implementa la interfaz *javax.ejb.EntityBean*. El container se encargará de crear una clase que herede de ella que implemente la persistencia a través de JDBC.

Esta clase abstracta no debe contener ningún campo persistente ni ninguna relación con otro entity bean, ya que serán creados por el container. Sin embargo, los campos pueden ser accedidos creando los *getter* y *setter* correspondientes y declarándolos como métodos abstractos. Luego estos campos son declarados en el descriptor a través de los elementos `<cmp-field>` para campos propios y `<cmr-field>` para relaciones.

Dado que un *finder* es utilizado para buscar y retornar entidades al cliente, no existiría forma para que el desarrollador realice una búsqueda interna de ciertos datos necesarios para la implementación de ciertos métodos de negocio. Para solucionar este problema es que en CMP se han creado los *ejbSelect* que son métodos análogos a los *finders*, con la principal diferencia que no son declarados en la interfaz home.

Para especificar al container cómo debe implementar un *finder* o un *ejbSelect*, es decir qué sentencia SQL debe ejecutar, es que se ha definido el *EJB Query Language (EJB QL)*. Una sentencia EJB QL debe ser transformada por el container a un lenguaje destino de una fuente de datos, como SQL para una base de datos relacional, para poder ser ejecutado. La ventaja principal es que EJB QL utiliza el esquema de persistencia abstracto de los entity beans para definir operaciones y expresiones, así la sentencia se escribe en relación al objeto y no a la representación en la base de datos.

En general, EJB QL es bastante similar a SQL, por lo cual no es difícil de ser entendido ya que utiliza una sintaxis similar -aunque reducida- para seleccionar objetos o valores. Debe contener la instrucción SELECT y la instrucción FROM y opcionalmente una instrucción WHERE que puede estar compuesta por parámetros a través de instrucciones `?N` que es reemplazado por el N-ésimo parámetro recibido. Un ejemplo de una query EJB QL que se utilizaría en un método para buscar personas con edad mayor a cierto parámetro, es la siguiente:

```
SELECT OBJECT(a)
FROM Persona AS a
WHERE a.edad > ?1
```

En esta sentencia, *Persona* se refiere al esquema de persistencia abstracto, y *edad* corresponde a uno de sus campos, por tanto no se hacen referencias directas a tablas o columnas. Cuando se declare como se efectuará la persistencia del esquema abstracto, el container tendrá la información necesaria para transformar la sentencia EJB QL en una sentencia SQL para ser ejecutada.

Para implementar un entity bean CMP, debemos definir la interfaz local y/o remota, que contendrá los getter y setter. A continuación se mostrará la interfaz remota de cliente:

```
public interface Producto extends EJBObject {
  public String getNombre() throws RemoteException;
  public void setNombre(String nombre) throws RemoteException;
  public double getPrecio() throws RemoteException;
  public void setPrecio(double precio) throws RemoteException;
  public String getProductoID() throws RemoteException;
}
```

La interfaz home sería la siguiente:

```
public interface ProductoHome extends EJBHome {
  //create
  Producto create(String productoID, String nombre, double precio)
    throws CreateException, RemoteException;

  //finders
  public Producto findByPrimaryKey(ProductoPK key)
      throws FinderException, RemoteException;
  public Collection findByNombre(String nombre)
      throws FinderException, RemoteException;
  public Collection findByPrecio(double precio)
      throws FinderException, RemoteException;
  public Collection findBaratos(double precioMaximo)
      throws FinderException, RemoteException;
  public Collection findAll()
      throws FinderException, RemoteException;
}
```

La llave primaria a implementar debe ser serializable. Una restricción para los campos que la componen es que deben pertenecer a los campos de CMP definidos en el descriptor, para el caso de ejemplo bastará con un campo tipo String que guarde el identificador único.

```
public class ProductoPK implements java.io.Serializable {
  public String productoID;
  public ProductoPK(String productoID) {
    this.productoID = productoID;
  }
  public ProductoPK() {
  }
  public String toString() {
    return productoID.toString();
  }
  public int hashCode() {
    return productoID.hashCode();
  }
  public boolean equals(Object prod) {
    return ((ProductoPK)prod).productoID.equals(productoID);
  }
}
```

Luego es necesario crear la implementación del entity bean CMP:

```
public abstract class ProductoBean implements EntityBean {
  protected EntityContext ctx;
  public ProductoBean() {
  }

  //getter/setters abstractos
  public abstract String getNombre();
  public abstract void setNombre(String nombre);
  public abstract double getPrecio();
  public abstract void setPrecio(double price);
  public abstract String getProductoID();
  public abstract void setProductoID(String productoID);

  //métodos requeridos por el container
  public void ejbActivate() {
  }
  public void ejbRemove() {
  }
  public void ejbPassivate() {
  }
  public void ejbLoad() {
  }
  public void ejbStore() {
  }
  public void setEntityContext(EntityContext ctx) {
    this.ctx = ctx;
  }
  public void unsetEntityContext() {
    this.ctx = null;
  }
  //create
  public ProductoPK ejbCreate(String productoID, String nombre, double precio)
```

```
      throws CreateException {
    setProductoID(productoID);
    setNombre(nombre);
    setPrecio(precio);
    return new ProductoPK(productoID);
  }
  public void ejbPostCreate(String productoID, String nombre, double precio) {
  }

  //no se implementan finders
}
```

En este código destacan los getter y setters abstractos, ya que el container es quien los implementará. Debe existir un método *ejbPostCreate* por cada *ejbCreate* con los mismos argumentos, que es ejecutado por el container luego de *ejbCreate*. Su existencia se basa en la separación en dos pasos que existe en la creación de un entity bean, el primero para crear la entidad con sus campos propios y el segundo para asignar de relaciones con otros beans. Por esto, en la implementación de *ejbPostCreate* se puede obtener la llave primaria y completar la inicialización de campos de relaciones con otros beans.[14, pág.172]

Finalmente es necesario declarar el entity bean creado en el archivo descriptor *ejb-jar.xml*. La declaración es la siguiente:

```
<ejb-jar>
  <enterprise-beans>
   ...
    <entity>
      <ejb-name>Producto</ejb-name>
      <home>ProductoHome</home>
      <remote>Producto</remote>
      <local-home>ProductoLocalHome</local-home>
      <local>ProductoLocal</local>
      <ejb-class>ProductoBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>ProductoPK</prim-key-class>
      <reentrant>False</reentrant>

      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>Producto</abstract-schema-name>
      <cmp-field>
        <field-name>productoID</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>nombre</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>precio</field-name>
      </cmp-field>
      <query>
        <query-method>
```

```
        <method-name>findByNombre</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
        <![CDATA[SELECT OBJECT(a) FROM Producto AS a WHERE nombre = ?1]]>
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findByPrecio</method-name>
        <method-params>
          <method-param>double</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
        <![CDATA[SELECT OBJECT(a) FROM Producto AS a WHERE precio = ?1]]>
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findBaratos</method-name>
        <method-params>
          <method-param>double</method-param>
        </method-params>
      </query-method>
      <ejb-ql>
        <![CDATA[SELECT OBJECT(a) FROM Producto AS a WHERE precio < ?1]]>
      </ejb-ql>
    </query>
    <query>
      <query-method>
        <method-name>findAll</method-name>
        <method-params>
        </method-params>
      </query-method>
      <ejb-ql>
        <![CDATA[SELECT OBJECT(a) FROM Producto AS a WHERE productoID IS NOT NULL]]>
      </ejb-ql>
    </query>
  </entity>
 ...
</enterprise-beans>
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>Producto</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>*</method-name>
    </method>
    <method>
      <ejb-name>Producto</ejb-name>
      <method-intf>Local</method-intf>
```

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

En el código se declara el nombre del bean y las clases que la implementan, luego se señala el tipo de persistencia a utilizar, que será la versión de CMP descrita en la especificación 2.0. Después es necesario declarar el nombre del esquema abstracto y los campos que la componen. Luego se declara la implementación de los finders definidos por su nombre y sus parámetros, donde las consultas EJB QL son escritas dentro de los elementos `<![CDATA[` y `]]>` para que el *parser* XML no confunda los caracteres `<` y `>` con los delimitadores de tags. El elemento `<assembly-descriptor>` es similar a un entity bean BMP, es decir, define roles de seguridad, permisos de ejecución y tipos de transacciones.

Si bien en el esquema abstracto se definen los nombres de los campos involucrados, no se especifica cual es la relación exacta con la base de datos. La definición de la persistencia de un entity bean CMP y su relación con tablas y columnas es específica al container y debe ser detallada en descriptores XML especializados. Para más detalles ver la implementación del proyecto 7 y el apéndice A.2.

3.3.6. Message-Driven Beans

Los message-driven beans son componentes detalladas en EJB 2.0 que pueden recibir y consumir mensajes asíncronamente. Un message-driven bean es invocado por el container como resultado de la recepción de un mensaje enviado por un cliente utilizando *Java Message Service (JMS)*.

Un cliente no ejecuta directamente un message-driven bean, si no que sólo debe utilizar la API de JMS para enviar mensajes. Por esto, un message-driven bean no tiene una clase *home* ni interfaz local o remota ni retorna valores o excepciones al cliente. El cliente no espera que su mensaje sea respondido si no que continúan su ejecución una vez enviado.

Los message-driven beans sólo reciben mensajes JMS, sin conocer de antemano la información sobre contenido del mensaje recibido. Por esta razón sólo tienen un método con lógica de negocio llamado *onMessage()*, que recibe un *Message* JMS que puede representar todos los tipos de mensajes existentes en JMS como mensajes de bytes, de texto y de objetos serializables. Luego hay que discriminar el tipo de mensaje recibido utilizando el operador *instanceOf*.

Los message-driven beans son *stateless* ya que no mantienen estados de conversación entre cada procesamiento de mensajes recibidos, por lo cual las instancias de la misma clase

son equivalentes entre sí y deben implementar solo un método *ejbCreate()* sin parámetros.

Ciclo de Vida

Para evitar errores de programación, un message-driven bean no puede ser ejecutado por más de un thread a la vez. Por esta razón y además por no guardar estados, el container puede crear un pool de este tipo de beans similar al existente para stateless session beans.

Debido a su naturaleza *stateless* el ciclo de vida de un message-driven es simple. Cuando el container desea agregar un nuevo message-driven bean al *pool* debe crear una nueva instancia, asignar el contexto para que obtenga parámetros de sistema y finalmente ejecutar el método *ejbCreate()*. Después el bean se encuentra en capacidad para recibir y procesar mensajes. Cuando el container decide reducir el número de instancias existentes invoca el método *ejbRemove()* terminando su ciclo de vida. Ver figura 3.9.

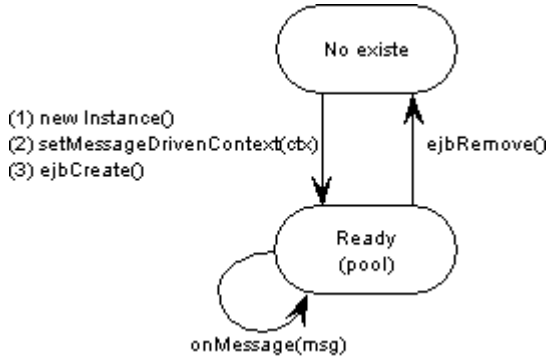

Figura 3.9: Ciclo de vida de un message-driven bean

Debido a su reducido tiempo de vida y a la no persistencia de sus datos, un message-driven bean no sobrevive a fallas en el container o en el servidor, debiendo ser reinstanciado para continuar recibiendo mensajes luego de ésta.

Implementación

Debido a que los message-driven beans sólo tienen como misión procesar los mensajes recibidos sin tener una relación directa con el cliente, no es necesario crear interfaces home ni locales ni remotas, por lo que basta crear una sola clase que implemente las interfaces *java.ejb.MessageDrivenBean* y *javax.jms.MessageListener*, para satisfacer el ciclo de vida del bean y el proceso de mensajes, respectivamente. Por ejemplo, un message-driven bean que escriba a la salida estándar lo recibido es el siguiente:

```
public class LogBean implements MessageDrivenBean, MessageListener {
```

```
  MessageDrivenContext ctx;
  public void setMessageDrivenContext(MessageDrivenContext ctx) {
    this.ctx = ctx;
  }
  public void ejbCreate() {
  }
  public void ejbRemove() {
  }
  public void onMessage(Message msg) {
    if (msg instanceOf TextMessage) {
      TextMessage tm = (TextMessage) msg;
      try {
        String text = tm.getText();
        System.out.println("Mensaje: " + text);
      } catch(JMSException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Un objeto *Message* puede ser del tipo *BytesMessage* para un grupo de bytes cualquiera, *MapMessage* para grupos de variables y su valor, *ObjectMessage* para objetos, *StreamMessage* para mensajes con tipos primitivos en bytes y *TextMessage* para mensajes de texto plano.

Descriptor

Para declarar el message-driven bean creado anteriormente es necesario agregar el elemento `<message-driven>` en el archivo *ejb-jar.xml*. En ese elemento hay que definir el nombre del bean, la clase que lo implementa, el manejo de la transacción y el tipo de mensajes a procesar (*javax.jms.Topic* o *javax.jms.Queue*) (ver sección 5.6).

```
<ejb-jar>
  <enterprise-beans>
   ...
    <message-driven>
      <ejb-name>Log</ejb-name>
      <ejb-class>LogBean</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <destination-type>javax.jms.Topic</destination-type>
      </message-driven-destination>
    </message-driven>
   ...
  </enterprise-beans>
</ejb-jar>
```

Se define que las transacciones serán el tipo de mensajes a procesar será *Topic*, sin embargo, no se señala el nombre del tópico específico a utilizar para recibir los mensajes porque se debe utilizar un descriptor dependiente del servidor.

Ejecución

Un programa que ejecute el message-driven bean sería el siguiente:

```
Context ctx = new InitialContext();
TopicConnectionFactory factory = (TopicConnectionFactory)
    ctx.lookup("javax.jms.TopicConnectionFactory");
TopicConnection connection = factory.createTopicConnection();
TopicSession session =
    connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
Topic topic = (Topic) ctx.lookup("prueba");
TopicPublisher publisher = session.createPublisher(topic);
TextMessage msg = session.createTextMessage();
msg.setText("Cuerpo del mensaje");
publisher.publish(msg);
```

El código contiene lo siguientes pasos: inicializa JNDI, luego a través de JNDI obtiene el objeto para iniciar la conexión al servicio de mensajería. Con este objeto crea una conexión y con este último crea una sesión. Después busca el canal de comunicación de nombre *prueba* y se señala que se desea enviar un mensajes en él. Finalmente crea el texto del mensaje y lo envía por el canal de comunicación, donde será recibido por el message-driven bean que lo procesará.

Capítulo 4

Deploy

En este capítulo se describirá el proceso de transformar una aplicación desde la etapa de desarrollo a una aplicación en etapa de producción en un servidor de aplicaciones.

4.1. Definiciones

J2EE proporciona facilidades para simplificar el empaquetamiento e instalación de una aplicación en su ambiente operativo. Utiliza archivos JAR para empaquetar módulos y aplicaciones, y archivos XML para la descripción y personalización de componentes y aplicaciones.

El proceso de instalar y personalizar una aplicación en un ambiente operacional es conocido como *deploy*. El proceso de unir componentes en módulos y los módulos en aplicaciones es conocido como empaquetamiento o *packaging*.

Una aplicación J2EE está compuesta por una o más componentes J2EE y por un *descriptor de deploy* de la aplicación, que enumera las componentes de la aplicación como módulos.

4.2. Módulos

Un módulo J2EE es una colección de uno o más componentes J2EE del mismo tipo más un descriptor de deploy. Un módulo representa la unidad básica de composición de una aplicación J2EE y puede ser instalada directamente en un container J2EE, o junto con otros módulos pueden ser combinados para formar una aplicación J2EE.

Los módulos y aplicaciones son empaquetados en *unidades de deploy*, que son archivos comprimidos similares a un archivo JAR con una estructura interna específica definida para cada tipo de módulo por la especificación de cada componente. Existen cuatro tipos de módulos J2EE:

- Módulos EJB, contienen clases EJB y relacionados.
- Módulos Web, contiene componentes web y sus recursos.
- Módulos de aplicaciones cliente, contiene las clases e interfaces necesarias para que una aplicación cliente ejecute los EJB disponibles.
- Módulos de adaptadores de recursos, contiene conectores Java, adaptadores de recursos y bibliotecas de soporte para ser agregados al container.

4.2.1. Descriptores

Además de componentes y recursos, cada unidad de deploy contiene un *descriptor de deploy*, que es un archivo XML que especifica las dependencias existentes entre cada componente y su entorno.

Los descriptores de deploy describen los contenidos de las unidades de deploy y configuran componentes y aplicaciones a su entorno. Además externalizan las relaciones entre componentes por lo que pueden ser modificadas sin alterar el código del programa.

Los descriptores especifican dos tipos de información:

- Información estructural. Datos que describen los componentes contenidos en la unidad de deploy, sus relaciones internas y sus dependencias externas. Esta información corresponde a características que no son configurables en la instalación como nombres de clases e interfaces de EJB o mecanismos de persistencia. Un container utiliza la información estructural para administrar las instancias de las componentes en tiempo de ejecución.
- Información de ensamble. Esta información opcional describe la forma en que los contenidos de una unidad de deploy son unidas con otras unidades de deploy para producir una nueva componente. Incluye nombres de relaciones de enterprise beans, roles de seguridad y variables de entorno.

Existen cinco tipos de descriptores de deploy, cada uno corresponde a un tipo de unidad de deploy. Cada tipo de descriptor está definido por su correspondiente especificación como un archivo DTD (ver sección 5.1). Las herramientas de desarrollo usualmente generan automáticamente los descriptores por lo que en general no es necesario crearlos o modificarlos directamente.

4.2.2. Aplicación J2EE

Una aplicación J2EE se empaqueta como unidades de deploy en archivos JAR con la extensión *.ear* (Enterprise ARchive) que contienen uno o más módulos J2EE y un descriptor de deploy.

La estructura de un archivo EAR está definida por la unión de cada módulo, más el descriptor de deploy de la aplicación llamado *application.xml* que se encuentra en el directorio de nombre *META-INF*.

4.2.3. Módulos EJB

Un módulo EJB es empaquetado como un archivo JAR con la extensión *.jar* (EJB-JAR) que contiene las clases Java que implementan los enterprise beans y sus interfaces locales y remotas, clases e interfaces Java necesarias que no estén incluidas en J2EE, y un descriptor de deploy que provee información estructural y de aplicación para el módulo EJB. El descriptor debe tener el nombre *META-INF/ejb-jar.xml*

4.2.4. Módulos Web

Un módulo Web es empaquetado como un archivo JAR con la extensión *.war* (Web ARchive) que contiene clases Java como servlets y clases auxiliares -que pueden ser empaquetadas en archivos JAR propios-, páginas JSP y sus clases auxiliares, documentos estáticos como HTML e imágenes, applets y sus clases, y un descriptor de deploy. Como otros módulos, un archivo WAR puede ser instalado como una aplicación web independiente o empaquetado dentro de un archivo EAR como parte de una aplicación J2EE.

La estructura interna de un archivo WAR está representada en la figura 4.1. El directorio raíz, llamado *context root*, contiene las páginas JSP, gráficos, applets y otros archivos que la aplicación proporcione al cliente ordenados en directorios según se desee. En este lugar se encuentra también el directorio *WEB-INF* que contiene archivos que no son proporcionados al cliente directamente si no que son necesarios para el módulo como el descriptor de deploy *web.xml*, el directorio *lib* que contiene las bibliotecas JAR que serán agregadas automáticamente al *classpath* en tiempo de ejecución y el directorio *classes* que contiene las clases necesarias para la aplicación que no se encuentren en un archivo JAR, que usualmente están estructuradas en directorios por paquetes.

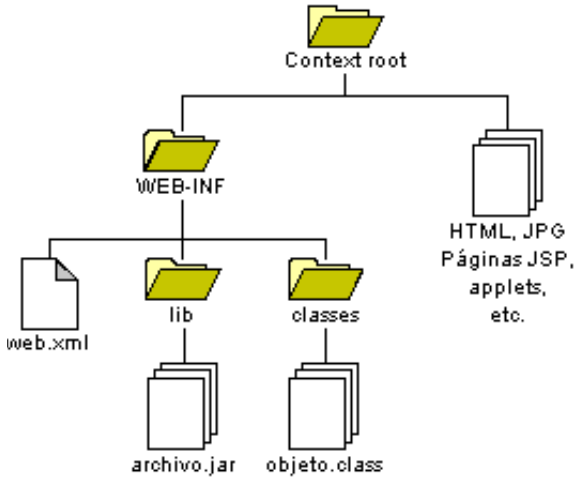

Figura 4.1: Estructura de un archivo WAR

4.2.5. Módulos de aplicaciones cliente

Los módulos de aplicaciones cliente son empaquetados en un archivo JAR con la extensión *.jar* (EJB-Client JAR) que contiene todos los archivos necesarios para que un programa cliente pueda utilizar los EJB de un módulo EJB-JAR como son las interfaces de cliente y las llaves primarias. Este módulo es necesario si el cliente utiliza una aplicación Java stand-alone que interactúa directamente con los EJB. Si este módulo no es creado, será necesario proveer al cliente el archivo EJB-JAR que contiene todas las clases e implementaciones.

4.2.6. Módulos de adaptadores de recursos

Un módulo Java Connector es empaquetado en un archivo JAR con la extensión *.rar* (Resource Adapter Archive) que contiene clases e interfaces java que implementan el contrato de la arquitectura connector y la funcionalidad del adaptador de recursos, empaquetados en uno o más archivos JAR, que contienen clases auxiliares del adaptador de recursos, bibliotecas dependientes de la plataforma, archivos de ayuda y documentación, y el descriptor de deploy. El descriptor de deploy debe tener el nombre *META-INF/ra.xml*.

4.3. Roles para la instalación

Toman parte tres roles distintos en el proceso de empaquetamiento e instalación: proveedores de componentes, ensambladores de la aplicación e instaladores.

4.3.1. Proveedores de componentes

Son los desarrolladores que crean las componentes EJB, servlets, JSP, HTML, applets, aplicaciones clientes y clases auxiliares. Además crean el descriptor de deploy para cada componente y pueden asignar valores por defecto para variables de ensamble. Ensambladores e instaladores pueden cambiar o definir su propia información de ensamble, pero usualmente no modifican la información estructural.

4.3.2. Ensambladores de aplicaciones

Están encargados de combinar las diferentes componentes desarrolladas en una sola aplicación y proveer información de ensamble para la aplicación definidas en el elemento `<assembly-descriptor>`. Define páginas de error, restricciones de seguridad y roles para servlets y EJB. Ensamblan módulos en una unidad de deploy más grande.

4.3.3. Instalador

Instalan componentes y aplicaciones J2EE en un ambiente operacional. Utilizan las herramientas creadas por el proveedor del servidor de aplicaciones para instalar los módulos y configurarlos en su entorno de ejecución, personalizando los elementos de los descriptores de deploy para el entorno. Típicamente la instalación involucra dos tareas.

- Instalación: Mover la aplicación al servidor, generar clases adicionales específicas al container y las interfaces que permiten al container manejar las componentes en tiempo de ejecución e instalar las componentes y clases e interfaces adicionales en el servidor J2EE.
- Configuración: El instalador resuelve todas las dependencias externas declaradas por el proveedor de componentes y sigue las instrucciones de ensamble definidas por el ensamblador de aplicaciones.

Capítulo 5

Tecnologías de J2EE

En este capítulo se describen diferentes tecnologías que son utilizada para un desarrollo en J2EE como son XML, JNDI, CORBA, conectores, transacciones, mensajería y seguridad. Para cada tema se pretende dar una explicación introductoria que permita comprender sus fundamentos y principales características.

5.1. XML

Extensible Markup Language (XML) es un método para representar y describir datos en un archivo de texto. XML es una versión simplificada de *Standard Generalized Markup Language (SGML)* que es el estándar internacional para definir la estructura y contenido de cualquier tipo de documento. Cabe señalar que HTML es una estructura de documento definida a través de SGML y que a la vez puede ser definida a través de XML, denominándose XHTML.

XML se estructura en base a etiquetas y atributos que pueden ser personalizables, las que se anidan para formar una estructura lógica de árbol. Un ejemplo simple es el siguiente:

```
<productos>
  <producto>
    <nombre>Artículo 1</nombre>
    <precio moneda="pesos">5000</precio>
  </producto>
</productos>
```

Cada archivo XML posee una estructura interna donde existen varios tipos de nodos diferentes como son Elementos (representados por las etiquetas), Atributos (las propiedades de las etiquetas) y Textos (texto dentro de cada etiqueta).

La forma de este árbol estructural o las propiedades que posea (binario, recursivo, balanceado, etc.) depende de la definición que se haga de él a través del llamado *Document Type Definition (DTD)* el cual es un archivo que contiene las reglas formales para la definición de los elementos, atributos y entidades que tendrá el archivo XML, sus frecuencias y sus jerarquías internas. Al contener la definición de la estructura en forma independiente se tiene la capacidad de efectuar validación de archivos, análisis de los datos a través de su estructura y un procesamiento más rápido al conocer ésta con anticipación. Un posible archivo DTD para el ejemplo es el siguiente:

```
<!ELEMENT productos (producto*)>
<!ELEMENT producto (nombre, precio?)>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT precio (#PCDATA)>
<!ATTLIST precio moneda CDATA #REQUIRED>
```

Las instrucciones de definición deben estar entre los delimitadores `<!` y `>` y las palabras claves para creación de elementos y atributos son *ELEMENT* y *ATTLIST*, respectivamente. Para definir un elemento se escribe su nombre y entre paréntesis los nodos hijos que contendrá junto con sus frecuencias (* para 0 o más veces, + para 1 o más veces, ? para 0 ó 1 vez). Para elementos que contendrán texto, es decir, sin elementos hijos, se utiliza la palabra clave *PCDATA*. Para declarar atributos se nombra el elemento al cual pertenece, el nombre del atributo a crear, el tipo de atributo (por ejemplo *CDATA* o *character data* para datos cualquiera) y si es un atributo obligatorio se utiliza la palabra clave *REQUIRED*.

J2EE utiliza los archivos DTD para especificar la estructura que deben contener los diferentes descriptores para el empaquetamiento de cada tipo de módulo y los distintos archivos de configuración. Así, en cada especificación de componente o servicio que utilice archivos XML de configuración, contiene un capítulo con la definición del DTD de cada archivo, describiendo la utilización de cada elemento.

Java API for XML Processing (JAXP) es una API que permite procesar archivos XML en forma independiente del algoritmo utilizado, así se pueden utilizar diferentes métodos utilizando la misma API estándar. Por ejemplo, JAXP incluye soporte para los métodos *Simple API for XML (SAX)*, API de dominio público, *Document Object Model (DOM)*, API desarrollada por la *WWW Consortium (W3C)*, y *eXtensible Stylesheet Language Transformations (XSLT)* API utilizada para efectuar transformaciones sobre un archivo XML (por ejemplo obtener una página HTML a partir de un XML).

5.2. Servicio de Nombres

Java Naming and Directory Interface (JNDI) es una API de J2EE, distribuida bajo el paquete *javax.naming.**, que proporciona una interfaz estándar para encontrar usuarios,

máquinas, redes, servicios, etc. Se puede utilizar JNDI por ejemplo para encontrar un objeto en una red o conectar a una base de datos.

El servicio de nombres provee dos tareas: asociar nombres con objetos (*bind*) y buscar un objeto dado un nombre (*look up*). Cada nombre a asociar está compuesto de su nombre atómico y su nombre compuesto. En el string */ejb/bean*, los strings *ejb* y *bean* son nombres atómicos y */ejb/bean* es un nombre compuesto. Los nombres pueden ser organizados en una forma similar a las direcciones web: un protocolo, una máquina, un puerto, un conjunto de directorios (denominados *contextos*) y un nombre de objeto. Para iniciar una búsqueda se requiere de un contexto base, que puede ser obtenido instanciando el objeto *javax.naming.InitialContext*, a través del cual se pueden obtener los recursos.

JNDI esta compuesto de dos elementos: una API y una *Service Provider Interface (SPI)*. La API es utilizada por el cliente para adquirir los recursos existentes en los directorios. La SPI es implementada por cada proveedor en forma particular a cada uno de sus productos en sus protocolos propietarios, permitiendo acceso a múltiples recursos manteniendo un código portable. Ver figura 5.1.

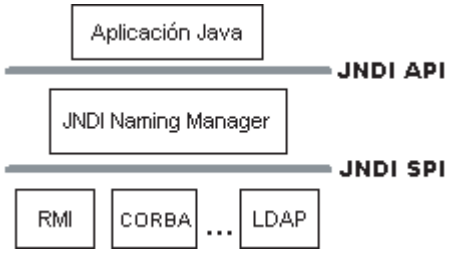

Figura 5.1: Arquitectura JNDI

5.3. CORBA

Common Object Request Broker Architecture (CORBA) es un estándar para escribir sistemas de objetos distribuidos en forma completamente neutral con respecto a la plataforma, lenguaje y proveedores. CORBA fue definido por el *Object Management Group (OMG)* que es un grupo fundado en 1989 por grandes compañías como 3Com Corporation, Canon, Hewlett-Packard, Sun Microsystems y Unisys Corporation con el objetivo de crear mercados de componentes de softwares creando estándares como CORBA y UML.

Los conceptos de EJB y RMI provienen de CORBA, aunque ha sido simplificado. EJB y J2EE tienen un enfoque centrado en Java y componentes. CORBA, por otro lado, ofrece una gama más amplia de capacidades que incluye servicios de tiempo, *locking* distribuido, relaciones y más. Además CORBA posee características utilizables para la integración de sistemas escritos en distintos lenguajes.

Un *Object Request Broker (ORB)* es una componente existente en cada máquina que se encarga de realizar la comunicación de datos con los ORB de otras máquinas. Esta comunicación es efectuada a través de la red utilizando el protocolo estándar de CORBA conocido como *Internet Inter-ORB Protocol (IIOP)* (ver figura 5.2). Existen variadas implementaciones de los ORB como por ejemplo OrbixWeb de Iona, VisiBroker de Inprise y ComponentBroker de IBM.

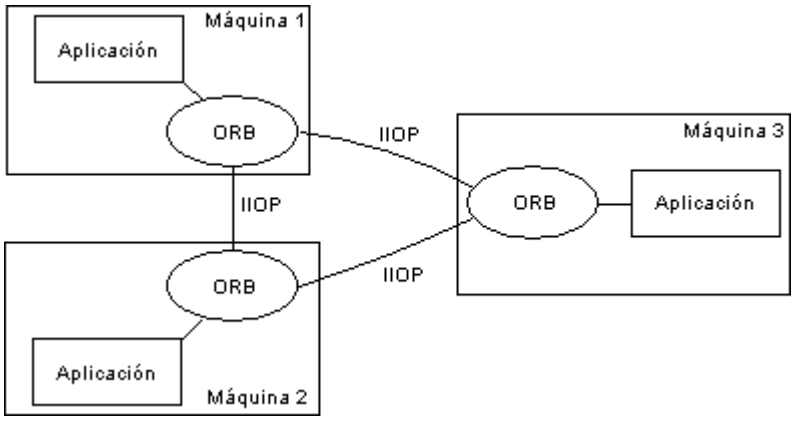

Figura 5.2: CORBA ORB

La piedra angular de la portabilidad de CORBA es el *Interface Definition Language (OMG IDL)* que es el lenguaje que se utiliza para definir las interfaces entre clientes y los objetos que utilizan. IDL crea una interfaz en un lenguaje neutral e independiente de la plataforma que puede ser implementada en cualquier lenguaje que soporte CORBA. Luego para que un objeto sea utilizado a través de la red, su implementación debe cumplir con la definición de su interfaz correspondiente. IDL es un lenguaje solo declarativo y es bastante similar a Java y C++. Un pequeño ejemplo es el siguiente:

```
module examples {
  interface Saludo {
    string hola(in string nombre);
  }
}
```

Para ser utilizada una interfaz debe ejecutarse una transformación de su declaración IDL a código perteneciente al lenguaje de programación que debe ser soportado por CORBA, con el cual luego puede ser programado y utilizado.

Un programa cliente que desee invocar un método remotamente debe utilizar un *stub*, el cual es un objeto local que cumple con la interfaz IDL y que representa al objeto remoto en la máquina local. Cada invocación a un método del *stub* es enviada a través del ORB al servidor donde es recibido por el ORB remoto. Al recibir la invocación, el ORB remoto ejecuta el *skeleton* del objeto, el cual es un objeto que también cumple con la interfaz IDL pero que

delega la invocación de un método sobre la implementación real del objeto (ver figura 5.3). Cabe señalar que los *stub* y *skeleton* utilizados en la comunicación son pregenerados a partir de la definición de la interfaz IDL.

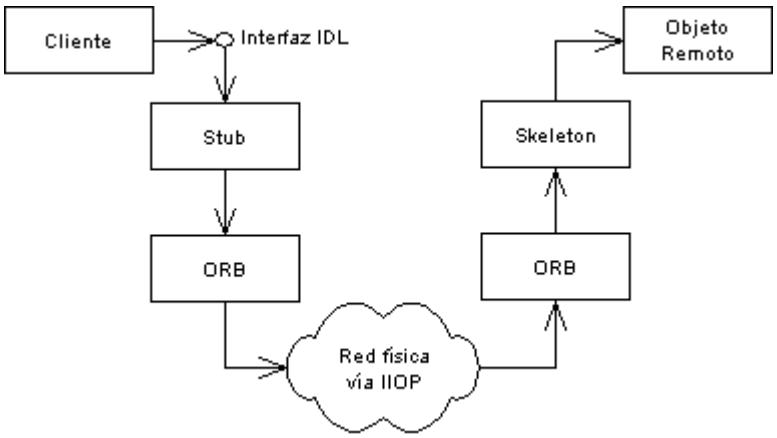

Figura 5.3: Stubs y Skeletons

Además de permitir la comunicación de objetos sobre una red, la OMG ha publicado un conjunto de servicios conocidos como *CORBA Object Services (COS)* que dan capacidades adicionales a los objetos distribuidos. Entre estos servicios se encuentra el *CORBA Naming Service (COS Naming)* que permite buscar objetos CORBA a través de un nombre, de una forma similar a JNDI y *CORBA Object Transaction Service (OTS)* que permite el uso de objetos CORBA para ejecutar transacciones.

5.3.1. Java RMI

Java Remote Method Invocation (Java RMI) es similar en estructura a CORBA, contiene stubs y skeletons para la comunicación y utiliza interfaces comunes para acceder a objetos remotos, sus diferencias radican en que RMI es centrado en Java, por lo que utiliza un protocolo de comunicación nativo conocido como JRMP y no contiene ORBs ni lenguaje IDL.

Para invocar un método de un objeto remotamente se debe definir una interfaz Java que debe extender de *java.rmi.Remote*. Esta interfaz es la que debe implementar el objeto remoto y que también implementará el *stub* en la máquina local. La generación de stubs y skeletons se realiza ejecutando el compilador de RMI (*rmic*) sobre la clase del objeto a compartir. Para poder unir cliente con servidor se define el *RMI registry* el cual es un servicio (ejecutado por defecto en el puerto 1099) que recibe referencias de objetos desde servidores y entrega referencias de objetos a clientes a través de stubs.

Para realizar el envío de parámetros a un método remoto o el retorno de datos desde él, se debe garantizar que el objeto a ser transmitido puede ser transformado a un conjunto

de bytes para ser enviado y poder ser reconstruible luego de su recepción, esto es conocido como *serialización*. Para señalar que un objeto es serializable debe implementar la interfaz *java.io.Serializable* la que no contiene métodos si no que sólo se utiliza para marcar clases. Todos los tipos de datos primitivos son serializables y su unión también lo es, sin embargo, no todo objeto es serializable, por ejemplo no tendría sentido serializar una referencia a un archivo local abierto para ser enviado a otra máquina.

5.3.2. Comunicación entre RMI y CORBA

CORBA es un estándar robusto que permite la operación entre distintos lenguajes. RMI, por otro lado, fue diseñado para una comunicación distribuida simple entre objetos Java. Aunque RMI y CORBA son muy similares, históricamente han sido tecnologías incompatibles. Al programar con RMI se debe crear código utilizando su API. Si luego se desea usar CORBA como protocolo de comunicación el código debe ser reescrito utilizando la API de CORBA.

Idealmente se espera que un cliente RMI pueda ejecutar objetos remotos de un servidor CORBA y que un cliente CORBA pueda ejecutar objetos RMI. La dificultad entre su comunicación radica esencialmente en los protocolos utilizados: IIOP para CORBA y JRMP para RMI.

Para permitir la intercomunicación entre RMI y CORBA se decidió utilizar como protocolo común IIOP, debido a que es más maduro y robusto, y existen aplicaciones de distintos proveedores diseñadas para la interoperabilidad de objetos distribuidos utilizando este protocolo.

La combinación entre RMI y CORBA no es una tarea simple dadas ciertas incompatibilidades entre ellos, por lo cual fue necesario que la OMG creara una nueva versión de su especificación de CORBA para hacer posible la utilización del protocolo llamado RMI-IIOP. Las principales dificultades en la elaboración de este protocolo correspondieron a la posibilidad de efectuar *cast* de objetos remotos, que no existe en CORBA, y definir interfaces en lenguaje IDL, que no existe en RMI:

- Al utilizar RMI-IIOP, no se puede simplemente hacer un *cast* de un objeto recibido por la red, porque el *stub* puede no existir en la máquina cliente. Esto se permite en RMI ya que es posible obtener una versión del *stub* que se necesita desde el servidor de RMI, sin embargo, esto no está definido en CORBA. Por lo cual se debe efectuar un *cast seguro* ejecutando explícitamente el método estático *javax.rmi.PortableRemoteObject.narrow()* que convierte un objeto en la interfaz remota solicitada. Este método recibe como parámetro un objeto y la clase a la cual se desea hacer el *cast*, retornando el objeto resultante o emitiendo una excepción si la clase no existe.
- RMI-IIOP define una correspondencia entre los tipos de RMI y los tipos de IDL, que

proporciona una forma definida para transformar automáticamente los tipos de Java utilizados en RMI-IIOP a los utilizados por IDL. Con esto se puede escribir un *compilador Java-to-IDL* que automáticamente realice esta transformación y obtenga el IDL que puede ser utilizado por clientes CORBA al llamar objetos RMI. Con este compilador se pueden generar aplicaciones Java y luego agregar la interoperabilidad de CORBA generando el IDL automáticamente.

Otra compatibilidad posible entre J2EE y CORBA se da en la relación directa entre COS Naming y JNDI que pueden ser unidos utilizando RMI-IIOP. Para esto es necesario contar con un driver específico para el servidor JNDI para permitir la capacidad de compatibilidad con CORBA. Con esto un cliente RMI-IIOP puede acceder a servidores RMI-IIOP y CORBA sin modificar el código si no sólo instalando el driver correcto.

5.4. Conectores

J2EE Connector Architecture (JCA) define una arquitectura estándar para conectar la plataforma J2EE con recursos externos y EIS heterogéneos como por ejemplo RDBMS o ERP. Define un mecanismo escalable y seguro para dar soporte a la integración de sistemas de datos con servidores J2EE y sus aplicaciones.

JCA permite que un proveedor proporcione un adaptador de recursos estándar para su producto el cual es conectado en un servidor de aplicaciones J2EE para agregar la infraestructura necesaria para utilizarlo. Por tanto, un adaptador de recursos es un *driver* de nivel de sistema que es utilizado por una aplicación Java para conectar con un EIS.

Así, un servidor que da soporte a JCA asegura la conectividad con múltiples sistemas y productos de diferentes vendedores, y un producto que provee de un adaptador de recursos tiene la capacidad de ser utilizado por cualquier servidor que de soporte a JCA.

El servidor J2EE y el sistema externo colaboran a través de un conector para mantener transparentes a los componentes de la aplicación los mecanismos de nivel del sistema como transacciones, conexiones y seguridad. Para esto se define una *Common Client Interface (CCI)* que es una API estándar (definida en el paquete *javax.resource.cci.**) que es utilizada por un cliente para acceder a un EIS y que es implementada por cada proveedor a través de una SPI. CCI no es un reemplazo para JDBC si no un complemento, ya que define una API paralela para utilizar sistemas de información distintos a una base de datos relacional. Ver figura 5.4.

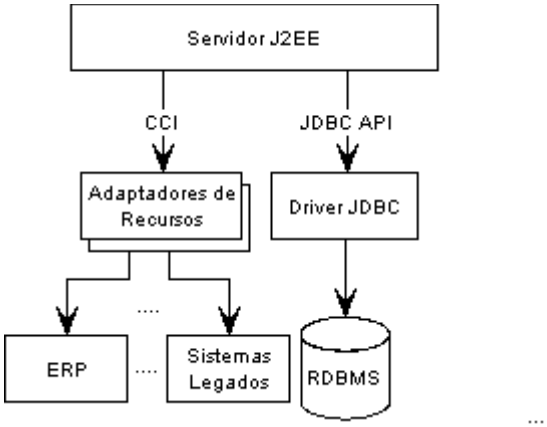

Figura 5.4: Relación CCI y JDBC

5.5. Transacciones

Un servicio clave para el desarrollo de aplicaciones robustas, en especial al realizar operaciones con una base de datos, son las *transacciones*. Una transacción en una serie de operaciones que simulan ser ejecutadas como una sola gran instrucción, así las transacciones garantizan que todas las operaciones del conjunto serán exitosas o ninguna de ellas lo será.

Las transacciones permiten a múltiples usuarios compartir los mismos datos y garantizar que cualquier modificación a un conjunto de datos será ejecutada completamente sin la intervención de otros clientes. Permiten las interacciones de un mismo conjunto de datos en forma concurrente por múltiples usuarios, simulando que cada cliente es el único usuario en la base de datos.

Al utilizar transacciones las operaciones siempre se ejecutarán con un conjunto de cuatro propiedades, conocidas como propiedades ácidas por sus siglas en inglés. Atomicidad garantiza que las operaciones se ejecutarán como una única unidad. Consistencia garantiza que la transacción no romperá las reglas de datos por una ejecución incompleta. Aislamiento garantiza que se pueden ejecutar múltiples transacciones concurrentemente sin interferir entre ellas. Durabilidad garantiza que una transacción finalizada exitosamente actualizará los datos en forma persistente.

La especificación J2EE señala que un container debe soportar al menos el modelo de transacciones planas (*flat transactions*), es decir, todas las instrucciones de la transacción forman una única unidad. Sin embargo, existen containers que dan soporte para el modelo de transacciones anidadas (*nested transactions*), donde cada transacción pueden contener subtransacciones y si una subtransacción falla existe la posibilidad de tomar la decisión de continuar la transacción mayor, efectuar otra acción, o abortar la transacción completa. La utilización de transacciones anidadas permite mayor control sobre ellas, al costo de crear un código no necesariamente portable entre containers.

Cada transacción debe contener una instrucción para iniciarla y una instrucción para señalar si ha sido exitosa o no. Para iniciar una transacción se utiliza la instrucción *begin*, para aceptarla y hacer los cambios permanentes se utiliza *commit* y para abortarla cancelando todos los cambios hechos se utiliza *abort* o *roll-back*.

En un enterprise bean se puede diferenciar dos tipos de transacciones. Las transacciones manejadas por el bean, en las cuales es el bean el responsable de iniciar una transacción y luego aceptarla o abortarla y las transacciones manejadas por el container, donde el container es el que se encarga de iniciarla y de aceptarla o abortarla. La decisión de cual tipo de transacción utilizar debe ser declarada en el elemento `<transaction-type>` del archivo *ejb-jar.xml*.

5.5.1. Transacciones manejadas por el container

En una transacción manejada por el container es necesario definir ciertos atributos para especificar como debe controlarla. Existen seis tipos de atributos del cual debe ser elegido uno de ellos y declarado en el elemento `<trans-attribute>` del descriptor:

- *Required*. Se utiliza este modo si se desea que el bean se ejecute siempre dentro de una transacción. Si una transacción se encuentra en curso, el bean participa en ella. Si no existe ninguna transacción en curso, el container creará una especial para la ejecución del método del bean y al finalizar el container ejecuta un *commit* si es que no ha sido abortada por el bean. Este modo es utilizado cuando se necesita que el bean sea transaccional.

- *RequiresNew*. En este modo el método del bean siempre ejecutará en una nueva transacción. Si no existen transacciones, se crea una especial igual al modo Required. Si existe una transacción en curso, es suspendida mientras el bean ejecuta en una transacción independiente y una vez terminado el método del bean, la transacción inicial continúa su curso. Se puede utilizar este modo cuando el bean es transaccional pero no se desea que comparta instrucciones con otros beans o con el cliente.

- *NotSupported*. En este modo el bean no puede ser ejecutado dentro de una transacción. Por tanto si se está desarrollado una transacción, es suspendida su ejecución mientras se ejecuta el bean y luego es reanudada al finalizar el bean. Este modo puede ser utilizado cuando el bean no requiere de sistemas transaccionales o no se desean las propiedades ácidas para su ejecución. Por ejemplo se puede utilizar para ciertos beans con reportes de baja prioridad y no se desea bajar el performance del sistema con ellos.

- *Supports*. En este modo el comportamiento depende de si existe una transacción en curso. Si es así, es utilizada dentro del bean. Si no es así, el bean no se ejecuta dentro de una transacción. Dada la dualidad de comportamiento, este modo debe ser utilizado con cuidado. Puede utilizarse cuando se desea delegar la decisión de usar una transacción sobre otro bean.

- *Mandatory*. En este modo es obligatoria la existencia de una transacción en curso para su ejecución. De no ser así el container debe emitir la excepción *javax.ejb.TransactionRequiredException* si se está utilizando interfaces remotas o *javax.ejb.TransactionRequiredLocalException* si se utiliza una interfaz local. Este modo garantiza que la ejecución será realizada bajo una transacción que fue iniciada por el cliente. Es útil para beans que componen un workflow y que son parte de un sistema mayor.

- *Never*. En este modo es obligatoria la no existencia de una transacción. De existir es emitida la excepción *java.rmi.RemoteException* si el cliente es remoto, o *javax.ejb.EJBException* si es un cliente local. Este modo permite asegurar al cliente que este bean no necesita de utilizar transacciones y puede ser utilizado en el desarrollo de sistemas no transaccionales por naturaleza y asegurar que así será utilizado.

5.5.2. Transacciones manejadas por el bean

Las transacciones manejadas por el bean permiten un control avanzado sobre el comportamiento de la transacción. Para poder hacer uso de la transacción es necesario utilizar una API especializada conocida como *Java Transaction API (JTA)*.

JTA es un conjunto de interfaces de alto nivel divididas en tres conjuntos: un conjunto de interfaces para administradores de recursos que utilizan el estándar *X/Open XA*, un conjunto de interfaces para transacciones manejadas por servidores de aplicación, y -la que nos interesa- un conjunto de interfaces para el uso de transacciones por aplicaciones transaccionales.

Al ser una interfaz de alto nivel, JTA necesita de una implementación que debe ser dada por cada proveedor. Esta implementación es realizada a través de la API *Java Transaction Service (JTS)*. Por otra parte, JTS es una implementación en Java del *OMG Object Transaction Service (OTS)* el cual es un servicio de CORBA para la utilización de transacciones.

JTA contiene la interfaz *javax.transaction.UserTransacion* que permite controlar las transacciones en un bean. Esta interfaz posee métodos como *begin()*, *commit()* y *rollback()*, que permiten manejar el flujo de la transacción. Este objeto puede ser obtenido por un enterprise bean a través del método *getUserTransaction* del objeto *EJBContext* que le entrega el container para su inicialización, o a través de JNDI haciendo un *lookup* a una dirección definida en el container como por ejemplo *java:comp/UserTransaction*.

5.6. Mensajería

La mensajería es una alternativa a la invocación remota de métodos. La idea principal es crear una entidad intermedia entre el servidor y el cliente, que se encargue de recibir

mensajes desde uno o más *productores* y comunicarlos a uno más *consumidores* existentes.

Un *Message-Oriented Middleware (MOM)* es un término usado para referirse a cualquier infraestructura que soporte mensajería. Algunos productos comerciales que tienen una arquitectura basada en MOM son BEA Tuxedo, Microsoft MSMQ, IBM MQSeries, entre otros.

Java Message Service (JMS) es un estándar diseñado con el objetivo de eliminar los problemas de portabilidad entre los distintos proveedores. JMS está constituido de una API para escribir código para enviar y recibir mensajes, y de una SPI para agregar drivers específicos a cada proveedor.

Para iniciar una mensajería es necesario escoger un *dominio*, es decir, es necesario decidir un estilo de mensajería a utilizar. Existen dos tipos de mensajería: publicar/subscribir y punto-a-punto.

En publicar/subscribir, existen muchos publicadores (productores) y muchos subscriptores (consumidores). Los subscriptores registran su interés en cierto tema particular, llamado *tópico*. Los publicadores crean mensajes, llamados *eventos*, que son distribuidos a todos los subscriptores correspondientes. Es decir, es un sistema basado en grupos en los cuales los mensajes son emitidas en forma de *broadcast*, similar a la televisión donde existen diferentes canales donde cada uno emite señales que son recepcionadas por todos sus subscriptores.

En punto-a-punto, existe un único consumidor para cada mensaje. Múltiples productores pueden enviar mensajes al consumidor, los cuales son agregados a una cola (*queue*) en espera de ser consumidos. Luego el consumidor recibe los mensajes secuencialmente efectuando para cada uno las acciones correspondientes, es decir, es un sistema similar a realizar un llamado telefónico a una grabadora y luego el consumidor lee los mensajes uno a uno para procesarlos.

5.7. Seguridad

En J2EE, la seguridad de cada componente es proporcionada por su correspondiente container, el cual define dos tipos de seguridad: declarativa y programática.

La seguridad declarativa se refiere a expresar la seguridad de la aplicación, incluyendo roles, controles de acceso y requisitos de autenticación, a través de archivos XML externos a la aplicación.

La seguridad programática consiste en realizar las decisiones de seguridad sobre aplicaciones conscientes de ella. Esta forma es útil cuando la seguridad declarativa no es suficiente para expresar el modelo de seguridad utilizado por la aplicación.

Existen dos medidas de seguridad que un cliente debe cumplir para acceder a una aplicación: autenticación y autorización.

Autenticación verifica que el cliente sea quien dice ser. Para esto el cliente debe identificarse, ya sea dando nombres de usuario y passwords, o a través de credenciales que son verificadas contra los datos de usuarios existentes. Una vez autenticado, al cliente se le asocia una identidad que se mantiene para la sesión.

Autorización corresponde a verificar la validez de la ejecución de cierta funcionalidad de la aplicación por parte del cliente, dada su identidad obtenida en la autenticación.

Para apoyar estas medidas de seguridad de forma estándar, se ha creado el *Java Authentication and Authorization Service (JAAS)* el cual es una interfaz portable que permite autenticar y autorizar usuarios en Java.

Capítulo 6

Patrones de Diseño

Un patrón de diseño es *la mejor solución práctica a un problema común y recurrente* [5, pág.xiv]. Un patrón documenta y explica un problema que puede suceder al diseñar o implementar una aplicación, y luego aborda una solución práctica a ese problema. En este capítulo se describirán algunos patrones de diseño importantes que existen para el desarrollo de aplicaciones J2EE.

Este listado dista de ser detallado y estricto ya que sería extenderse demasiado en un tema para el cual existen libros exclusivamente dedicados, si no que pretende ser una pequeña guía de algunos patrones relevantes

6.1. Modelo-Vista-Controlador

En general, una misma aplicación deben permitir acceso a múltiples usuarios utilizando diferentes tipos de interfaces. Por ejemplo, una vista HTML para clientes, una vista WML para clientes, una interfaz para aplicaciones stand-alone de administración y una vista en XML para servicios web a proveedores, donde cada una de estas interfaces de acceso deben proporcionar los mismos datos a los clientes y ser parte de la misma lógica del negocio.

Aplicando el patrón Modelo-Vista-Controlador (MVC) se logra separar el modelo de negocios, de la presentación usada para ser visualizados, de la lógica utilizada para procesar acciones externas. Esta separación permite múltiples vistas sobre el mismo modelo de datos, lo que simplifica la implementación (permitiendo separar roles de desarrollo) y la mantención del sistema.

El patrón MVC está compuesto de tres elementos:

- Modelo: representa los datos empresariales y las reglas para acceder a ellos y modificarlos.
- Vista: especifica cómo deben ser desplegados los datos contenidos por el Modelo.
- Controlador: transforma interacciones efectuadas en la Vista en acciones a ser ejecutadas por el Modelo.

En el desarrollo del proyecto se aplicó este patrón de diseño por medio del framework para aplicaciones web llamado *Jakarta Struts*.

6.2. Data Access Object

Las aplicaciones pueden utilizar JDBC para acceder datos existentes en una base de datos relacional, sin embargo, al hacerlo se está ligando el modelo de datos existente y el tipo de base de datos con la lógica de la aplicación, lo cual trae consigo principalmente problemas de dependencia y mantención.

Usar un *Data Access Object (DAO)* permite abstraer y encapsular todos los accesos a una base de datos en un objeto especial que utiliza las conexiones para ejecutar las consultas necesarias para obtener y almacenar datos.[1, pág.390]

Un DAO implementa los mecanismos de acceso requeridos para trabajar con la fuente de datos, que puede ser un RDBMS u otro servicio externo. El DAO debe esconder completamente los detalles de la implementación a sus clientes. Así, cualquier cambio en la base de datos no requiere cambio en los clientes si no sólo en el DAO.

Un objeto DAO sólo debe contener operaciones de lectura y modificación de datos. No debe contener operaciones con lógica de negocio.

6.3. Session Façade

Dado que la información es guardada en entity beans, la primera opción al desarrollar de una aplicación cliente puede ser obtener cada entity bean e invocar directamente lo métodos necesarios para satisfacer la lógica de la aplicación. Sin embargo esto se traduce en alto tráfico en la red, poca mantenibilidad y alto acoplamiento. La solución es crear una capa por sobre los entity beans, llamada *Session Façade*, que provea la implementación de las funcionalidades requeridas por el cliente disminuyendo la dependencia con el modelo, minimizando el número de invocaciones remotas y proporcionando una capa de servicios uniforme y de más alto nivel a los clientes.[1, pág.293] [5, pág.5]

La solución típica para encontrar la funcionalidad a exponer a través del Session Façade es estudiar los casos de uso de la aplicación y transformarlos en los session beans que componen esta capa. Sin embargo, en algunos casos crear un session bean para cada caso de uso puede significar la existencia de una gran cantidad beans complicando esta capa. En esos casos es recomendable integrar los beans relacionados con los mismos tipos de datos y así mantenerlos en un número reducido.

6.4. Service Locator

En más de una ocasión un cliente deberá hacer uso de JNDI ya sea para obtener una conexión a la base de datos, una referencia a la clase home de un enterprise bean, o una referencia a los canales de mensajería. Al ser algo común para muchas componentes, tiende a aparecer código similar múltiples veces y la creación repetida del objeto *InitialContext* que puede tomar cierto tiempo.

Utilizar un *Service Locator* permite abstraer todo los usos de JNDI simplificando el código del cliente, creando un único punto de control y mejorando el performance de la aplicación [1, pág.367]. Para el caso particular, y más utilizado, de obtener objetos home existe el patrón de diseño *EJBHomeFactory* que consiste en buscar el nombre de la referencia de cierto objeto en los archivos XML de configuración de la aplicación para luego hacer utilizar JNDI para obtenerlo.[5, pág.92]

6.5. Value Object

Al modelar entity beans, para cada campo contenido generalmente se crea un método para obtener el valor y uno para modificarlo. Cuando se utilizan entity beans con interfaces remotas, y en especial cuando contiene muchos campos, se puede crear un gran tráfico en la red al ser necesario invocar cada uno de los métodos en forma sucesiva, provocando problemas de performance y escalabilidad. En estos casos es recomendable utilizar una forma para transportar el conjunto de valores desde y hacia el entity bean en una sola unidad, evitando múltiples invocaciones remotas.

Este objeto de transporte es conocido como *Value Object* o *Data Transfer Object*, y debe ser una clase serializable en la cual los valores son asignados en forma local, son enviados por la red y luego son recuperados en forma local en la otra máquina. [1, pág.261] [5, pág.47]

6.6. Fast-Lane Reader

Típicamente un usuario utilizará los resultados de una consulta sólo para lectura como despliegue y navegación. La implementación de un *finder* está dividida en dos etapas: consulta y obtención de un conjunto de N llaves primarias, y ejecución de N consultas para instanciar cada entity bean según su identificador, lo que es conocido como el problema de las $N + 1$ consultas. Al utilizar CMP los container permiten realizar configuraciones para cargar datos en forma masiva reduciendo el a dos el número de consultas necesarias, sin embargo para BMP no existe una solución de similar característica.

En este caso es recomendable utilizar una implementación del *finder* utilizando JDBC para obtener y desplegar los datos. Para esto existen varios enfoques, ya sea utilizando *Fast-Lane Reader*, *JDBC for Reading* [5, pág.76], o un objeto más elaborado como *Value List Handler* [1, pág.353] que controle las búsquedas y mantenga caché de resultados.

Estas patrones significan romper las capas en la que está estructurada la aplicación, por lo que deben ser implementados con cuidado de no aumentar las dependencias entre componentes.

6.7. Consideraciones de diseño

Para el diseño del modelo de negocio basado en EJB, hay que tener en cuenta algunas consideraciones generales, que en algunos casos no son abordadas por un patrón de diseño en particular, las que serán resumidas en esta sección.

6.7.1. Session beans

Un stateless session bean no mantiene ningún estado sobre las consecutivas llamadas a sus métodos, por tanto el container puede crear un pool para reutilizarlo en múltiples clientes. Un stateful session bean, por mantener un estado conversacional con un cliente en particular, debe mantenerse siempre ligado al mismo cliente sin posibilidades de ser compartido y por tanto siendo potencialmente más costoso en recursos.

Un enfoque erróneamente abordado en algunos diseños con intención de mejorar el performance del sistema, es modelar cada stateful bean como un stateless bean, entregando todo el estado conversacional como parámetros de sus métodos o leyéndolo desde la base de datos. Este enfoque crea problemas de escalabilidad al ser necesario reconstruir el estado en cada invocación con utilización extra de tráfico en la red, carga en la base de datos, costo de acceso, problemas de consistencia de la lógica del negocio al ser necesario que cada aplicación

cliente se encargue de mantener el estado, y en algunos casos incluso problemas de seguridad al delegar sobre una aplicación externa decisiones sobre la acción a tomar en cada invocación.

Por tanto la regla general es utilizar un stateful session bean cuando la lógica de negocio contenida en él sea mejor representada por este tipo que por un stateless, evitando los problemas antes descritos, y dejar la mantención de estados en la aplicación cliente sólo para las funcionalidades propias de ella.

6.7.2. Entity beans

Un entity bean puede contener lógica del negocio, sin embargo esta funcionalidad debe ser usada con moderación. La lógica a implementar dentro de un entity bean debe corresponder a procedimientos propios y exclusivos a los datos contenidos. Se debe evitar hacer manipulación de relaciones e interacciones con otros objetos o el usuario, ya que estas características deben ser implementadas por session beans.

Existen dos posibles formas de persistencia de un entity bean: BMP y CMP. Utilizar BMP tiene la ventaja principal de permitir manipular la persistencia en forma avanzada como utilizar procedimientos almacenados, reunir información de múltiples tablas y bases de datos, permitir formas complejas de optimización y poder acceder a sistemas legados. CMP en cambio necesita de un relación simple con la persistencia (en el momento de realizar esta investigación la mayoría de los servidores permitían sólo una tabla por bean), pero contiene ventajas de portabilidad, simplificación para implementar relaciones, optimizaciones globales para la carga de datos y menor tiempo de desarrollo. Por estas características, la regla general para decidir qué tipo de persistencia utilizar es: preferir siempre CMP a menos que sea obligatorio utilizar BMP.

Al efectuar el modelo de negocio contenido por los entity beans se debe tomar una decisión importante sobre la *granularidad* de los datos, es decir, el nivel de detalle de los objetos que se utilizará para el modelamiento.

Una granularidad gruesa consiste en modelar entity beans como objetos mayores que contienen como campos propios un conjunto de objetos de menor tamaño. La granularidad gruesa permite que en las invocaciones remotas sean pocas y en cada una viaje por la red un grupo de datos, sin embargo, requiere en general de entity beans BMP para definir la persistencia. Esta fue la solución preferida hasta EJB 1.1.

Una granularidad fina corresponde a crear entity beans por cada elemento mínimo existente en la lógica del negocio, lo que potencialmente se traduce en tener muchas invocaciones remotas con las posibles saturaciones de la red y baja escalabilidad. Sin embargo desde EJB 2.0 con la mejora de CMP y la creación de interfaces locales, se ha permitido que este enfoque haya tomado fuerza porque significa tener objetos minimales fácilmente persistentes con CMP (por ejemplo crear un bean para una tabla definiendo un campo para cada co-

lumna y cada instancia corresponderá a cada fila de la tabla), que no tendrían problemas de performance si se utilizan interfaces locales.

Finalmente, la decisión del enfoque a utilizar dependerá en ocasiones de particularidades de cada proyecto, por lo cual hay que tener presentes los dos enfoques y decidir entre ellos para lograr la mejor opción.

6.7.3. Message-driven beans

Message-driven bean son utilizados para modelar operaciones que no es necesario ejecutar en el mismo momento y que pueden ser procesadas en un tiempo posterior a su invocación. Con esto se permite al cliente enviar el mensaje con la solicitud y continuar su proceso sin esperar una respuesta. Algunos casos modelables por este tipo de bean son ciertas validaciones, chequeos y solicitudes como por ejemplo solicitudes de aumento de stock o seguimiento del despacho de productos.

Si la ejecución de un proceso session bean toma una cantidad significativa de tiempo o puede quedar bloqueada en forma indefinida, es probable que tenga un impacto negativo en el desempeño de la aplicación si se encuentra dentro de una transacción. En estos casos se puede modelar el proceso como un message-driven bean, o lo más utilizado, crear un message-driven bean que invoque el método de un session bean en forma asíncrona dependiendo del mensaje recibido.

Capítulo 7

Proyecto: Sistema de Registro de Actividades

En el presente capítulo se describirá el proyecto realizado para el presente trabajo, su contexto, sus objetivos, la metodología utilizada para enfrentar el desarrollo y un conjunto de patrones de diseño creados para solucionar problemas comunes a este tipo de proyecto.

7.1. Contexto y objetivos

El proyecto se enmarca dentro de la empresa consultora de computación Pragma Ltda. Como la mayoría de las empresas consultoras, Pragma basa su principal actividad de producción en función a proyectos.

Los consultores de la empresa deben mantener un registro de las actividades realizadas diariamente, existendo dos tipos de actividades definidas: actividades de proyecto y actividades especiales.

Una actividad de proyecto es un periodo de tiempo del día que se utiliza para trabajar para un proyecto en una etapa específica. Una actividad especial es un periodo de tiempo laboral utilizado en actividades productivas o no productivas que no están ligadas a un proyecto en particular. Así, una actividad especial puede ser una reunión interna de la empresa, una licencia médica, o una reunión con un cliente por motivos externos a un proyecto en particular.

El objetivo del proyecto es crear un pequeño sistema que provea una forma de ingreso de actividades a través de Internet, efectuando una prueba de las capacidades de J2EE -particularmente de EJB- para el desarrollo de un proyecto. Por tanto, el objetivo principal es crear una experiencia en el desarrollo de aplicaciones en cuatro capas que permita definir

una metodología a utilizar para enfrentar un desarrollo J2EE, por sobre desarrollar un nuevo sistema para ser puesto en producción. Como objetivo secundario se tiene definir un entorno de desarrollo, aplicaciones útiles y otras recomendaciones para llevar a cabo el desarrollo de una aplicación J2EE.

7.2. Análisis

En la presente sección se describirá lo que fue la etapa de análisis del sistema. Esta comenzó por el estudio del proceso de registro de actividades en la empresa, para luego continuar con la investigación de sistemas existentes. Con esta información se definieron los actores y los casos de uso involucrados en el proceso para finalizar con restricciones generales que debe cumplir el sistema a desarrollar.

Inicialmente cada consultor debía obtener una planilla electrónica a través de la red local en la cual debía ingresar los datos de cada una de sus actividades en el formato ahí especificado. Por cada proyecto en que participaba el consultor era necesario crear una nueva planilla cada mes, la que debía ser guardada en un directorio especial compartido en la red local. El último día de cada mes las planillas eran obtenidas para efectuar los procesos administrativos necesarios como por ejemplo calcular el salario de los consultores con contrato por hora y obtener la cantidad a facturar a clientes con cobro por actividad.

Luego se creó una sistema para registrar las horas a través de un software especial. Para esto se utilizó la base de datos de la empresa (Oracle 8.1.7) donde se creó un esquema con el modelo de datos que permitiera satisfacer los requerimientos del sistema. Así se crearon las tablas, columnas y relaciones ilustradas en la figura 7.1.

El modelo señala que cada proyecto pertenece a un cliente en particular y puede tener una o más etapas de desarrollo definidas (por ejemplo análisis, diseño, etc.). Una persona de la empresa registra sus actividades, ya sean de proyecto o especiales. Si es actividad de proyecto debe especificar la etapa del proyecto a la que corresponde. Si es actividad especial debe seleccionar el tipo de actividad especial (licencia, vacaciones, reunión urgente, etc.) y, si corresponde al tipo, debe definir el cliente relacionado.

Para desarrollar las pantallas de acceso se utilizó Oracle Forms 6i y Oracle Reports 6i, que luego los usuarios accedía a través de la red local debiendo tener instalado este software en el cliente. Un consultor que necesitaba utilizar el sistema debía identificarse y luego ingresar las actividades en la pantalla mostrada en la figura 7.2.

Para desarrollar el actual sistema de registro de actividades se comenzó estudiando los casos de uso del sistema. En este proceso se identificaron dos actores, llamados *usuario* y *administrador*. El primero representa a los consultores de la empresa que ingresarán al sitio a registrar sus actividades, el segundo representa a la persona que debe mantener la

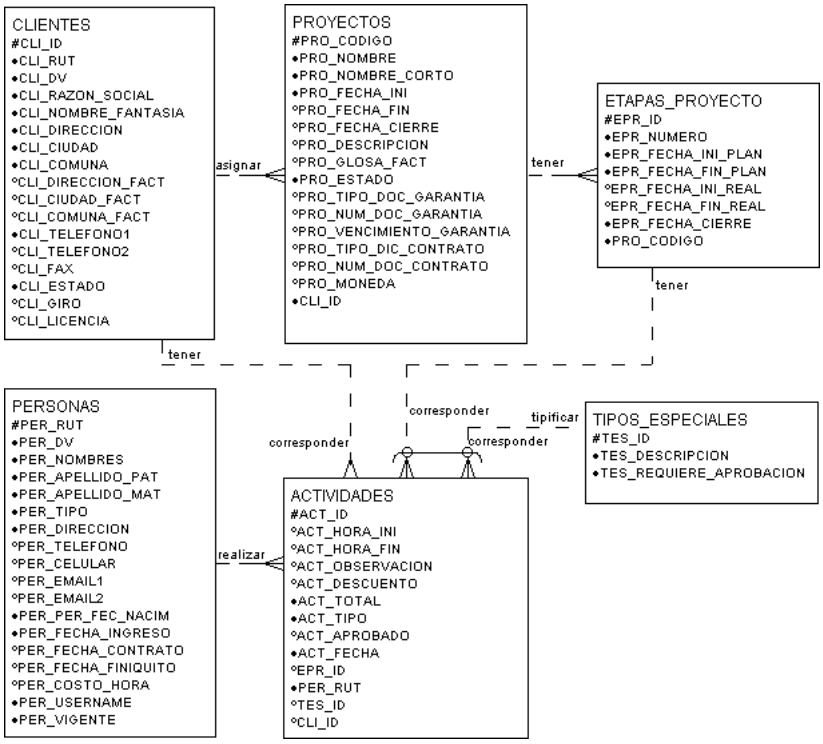

Figura 7.1: Modelo Relacional

información del sistema agregando nuevos proyectos, modificando clientes, creando nuevos usuarios, etc.

Para que un usuario pueda acceder al sitio deberá identificarse previamente, una vez autenticado podrá ver la información de sus actividades ingresadas y tendrá la posibilidad de ingresar nuevas actividades. Un administrador deberá identificarse y luego podrá visualizar la toda la información existente en el sitio ya sean clientes, proyectos, etapas de proyecto, personas, actividades y tipos de actividades especiales, y para cada uno de los seis tipos de datos tendrá la posibilidad de agregar uno nuevo, modificar la información de uno existente o borrar cada registro.

Con estas funcionalidades definidas, el sistema estará compuesto de cuatro casos de uso (ver figura 7.3):

- Validación: Utilizado por el usuario y el administrador. Consiste en validar el ingreso correcto de un actor al sistema a través de su nombre de usuario y su password.
- Búsqueda de información: Utilizado por el usuario y el administrador. Consiste en desplegar la información existente en la base de datos. Para el caso de un usuario debe permitir ver sólo los registros de actividades creados por él, y para el caso de un administrador debe permitir obtener todos los datos.

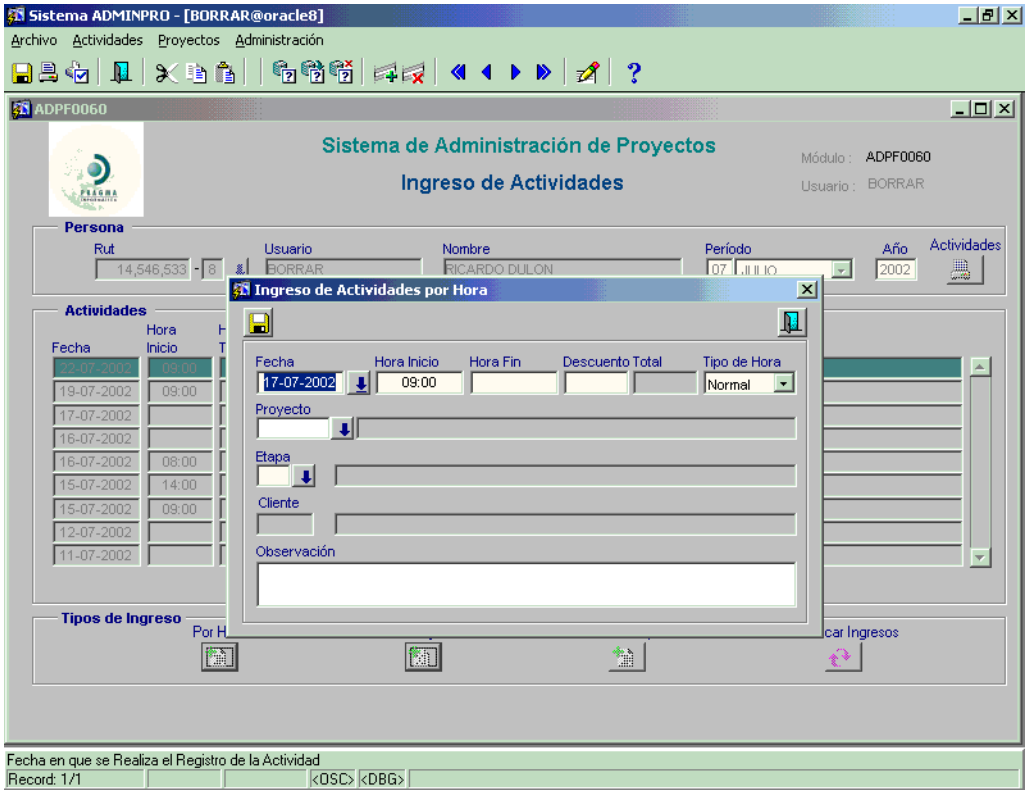

Figura 7.2: Ingreso de actividades con del sistema desarrollado en Oracle Forms 6i

- Registro de actividades: Utilizado por el usuario. Permite efectuar el registro de una nueva actividad, modificar una actividad existente o eliminarla, siempre y cuando haya sido creada por él.
- Administración de datos: Utilizado por el usuario. Permite crear, modificar o borrar registros de cada uno de los diferentes tipos de datos existentes en el sistema.

El proyecto a desarrollar debe cumplir con la restricción de ser compatible con el sistema ya existente para el registro de actividades, por lo cual no se permite modificar el modelo en la base de datos. El sistema debe otorgar acceso Web para el registro de actividades, con lo cual se permite acceso a los consultores que se encuentren realizando labores fuera de las oficinas de la empresa.

7.3. Diseño

La etapa de diseño del sistema define la estructura bajo la cual se creará el sistema. Para utilizar la arquitectura J2EE el sistema se compondrá de las cuatro capas correspondientes:

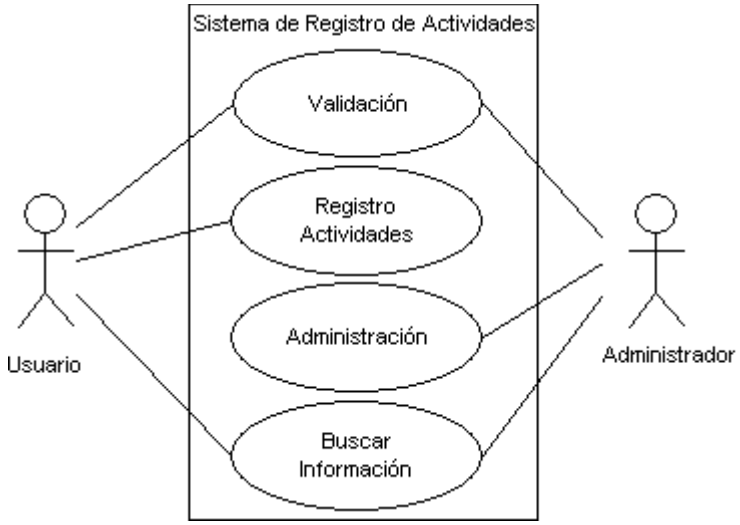

Figura 7.3: Diagrama de casos de uso

- Capa Datos: fue definida en la etapa de análisis y corresponde al modelo presentado en la figura 7.1 además de un conjunto de *sequence* utilizadas para asignar valores a las llaves primarias de las tablas *Clientes*, *Etapas_Proyecto*, *Tipos_Especiales* y *Actividades*.
- Capa Negocio: define el modelo a utilizar para EJB, el cual será descrito en la sección 7.3.1.
- Capa Web: corresponde a la estructura utilizada por servlets y JSP para procesar un requerimiento de un cliente web utilizando la capa EJB. Su estructura estará marcada por la utilización del patrón MVC y será detallada en la sección 7.3.2.
- Capa Cliente: un cliente para acceder al sistema deberá utilizar un navegador el cual desplegará el código HTML que retorne cada JSP. Se utilizará JavaScript para automatizar algunas características de confirmaciones y validaciones.

7.3.1. Capa Negocio

Como se explicó en la sección 3.3, EJB es utilizado como una capa de abstracción que proporciona una visión de objetos Java al modelo relacional existente en la base de datos. Dado el modelo de datos existente se prefirió por una granularidad fina para efectuar el modelo de objetos (ver sección 6.7.2). Así el diagrama de clases utilizado para la persistencia de datos del proyecto es el mostrado en la figura 7.4.

Cada clase mostrada corresponde a un entity bean. Cada entity bean contiene campos que componen la llave primaria y campos que son obligatorios, que son los representados por una *P* para el caso de llave primaria y por una *R* para el caso de campos requeridos. En resumen, se utilizaron seis entity beans: *Cliente*, *Proyecto*, *EtapaProyecto*, *Persona*, *Actividad*

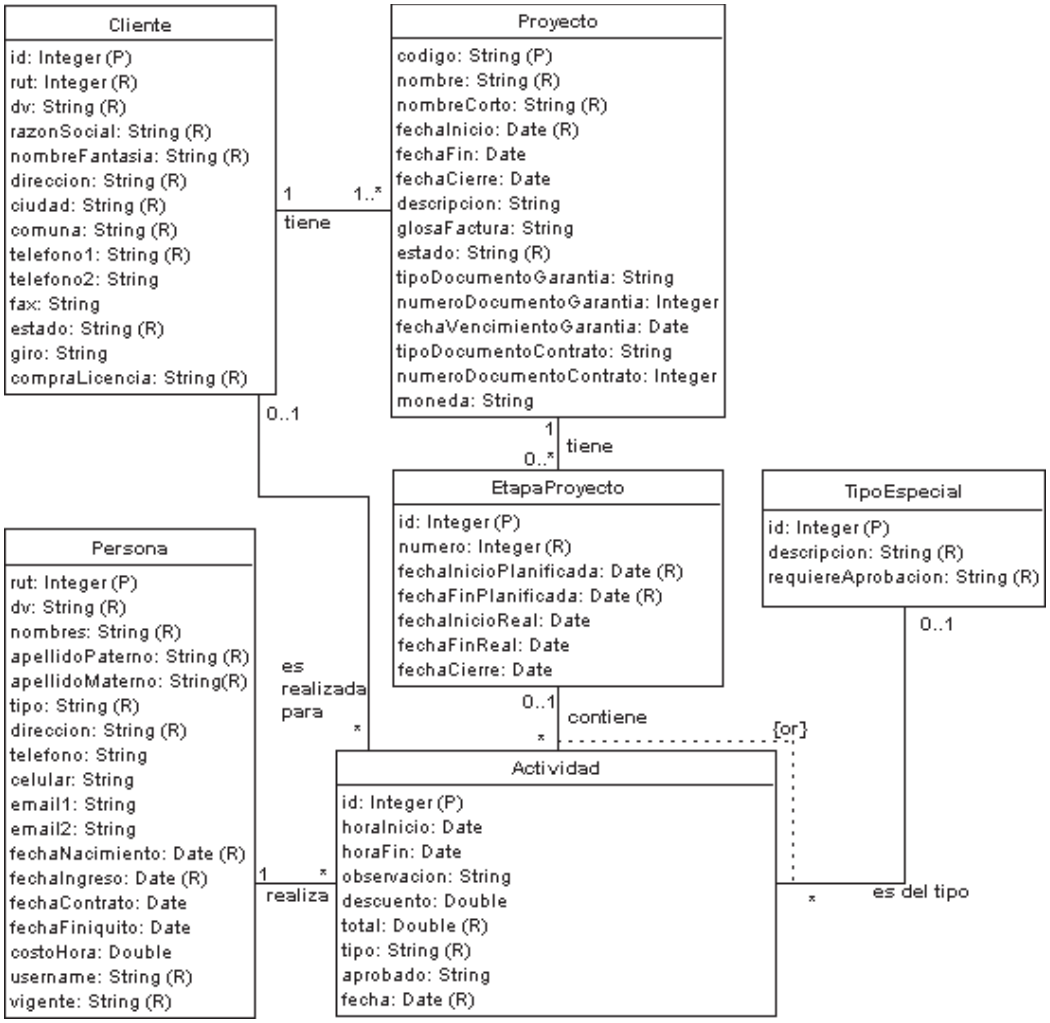

Figura 7.4: Diagrama de clases

y *TipoEspecial*, cada una con una correspondencia directa a las tablas existentes en la base de datos.

Utilizando el patrón de diseño DAO (ver sección 6.2), se definió la existencia de una clase especial que contiene todo la lógica necesaria para acceder a la base de datos, centralizando todo el código dependiente de ésta.

Aplicando el patrón de diseño Session Façade (ver sección 6.3), para cada caso de uso corresponde un session bean, el cual provee las funcionalidades que el cliente necesita. Así se crea una capa de servicios proporcionados a las aplicaciones cliente, donde cada session bean deberá utilizar uno o más entity beans para poder cumplir sus funcionalidades, creando una capa de separación entre cliente y datos. Así, cuando un cliente accede al sistema ya sea a través de web, a través de un applet, o a través de una aplicación stand-alone, deberá utilizar los mismos session bean que contienen la lógica del sistema, evitando duplicaciones de lógica

que pueden significar inconsistencias y dificultad de mantención.

Se crearon los siguientes session beans para modelar los casos de uso:

- ValidacionBean: Es un stateless session bean que contiene los métodos correspondientes a validar un usuario o un administrador dado un username y password.
- RegistroActividadesBean: Contiene los métodos necesarios para que un usuario registre sus actividades. Es un stateful session bean, ya que para ser creado necesita la persona a la que estará asignado y luego todas los métodos involucrados se efectuarán bajo ese contexto. Contiene métodos para buscar actividades, crear actividad especial, crear actividad de proyecto, modificar actividades y borrar actividades. En cada método se verifica que la actividad corresponda al usuario registrado.
- AdministracionBean: Contiene los métodos para crear, modificar y borrar datos de los seis tipos de entity bean. Es un stateless session bean.
- VerInformacionBean: Es un stateless session bean que contiene métodos para buscar información general de las cuales no hay restricción como ver los proyectos existentes y sus etapas, los tipos de actividad especial, los clientes y las personas existentes.

La capa se session beans es la encargada de crear, modificar y borrar entity beans, evitando que un usuario modifique directamente los entity beans. La principal razón para esto reside en que de esta forma se garantiza que las modificaciones son ejecutadas bajo transacciones, permitiendo mantener la consistencia de los datos. Por ejemplo, si una aplicación cliente comienza a modificar los campos de una entidad y una de las modificaciones arroja un error, el bean quedará en un estado intermedio con campos actualizados y campos sin actualizar. Otra razón para delegar sobre session beans la modificación de entity beans corresponde a tener un mayor control del nivel de seguridad del sistema, garantizando que los datos serán modificados por quien este autorizado independiente del tipo de acceso que utilice por medio de elementos como `<method-permission>` y `<security-identity>` del archivo *ejb-jar.xml*.

Se estudió el sistema existente y no se encontró una funcionalidad que debiera ser solucionada utilizando message-driven bean, es decir, una característica del sistema que fuera modelable correctamente por envío de mensajes para ser procesados en forma asíncrona.

7.3.2. Capa Web

Dadas las diferencias de funcionalidades que debe proveer el sistema para cada actor, se decidió por dividirlo en dos módulos web:

- Un módulo de administración, a través del cual los administradores tendrán acceso a ver y modificar toda la información relacionada con el sistema.
- Un módulo de registro de actividades, donde cada usuario involucrado en el desarrollo de un proyecto registrará las actividades realizadas.

Luego de decidir los módulos a crear, es necesario detallarlos a través de mapas de navegación. Un mapa de navegación permite determinar un módulo web a desarrollar a través de las páginas web que contiene y de las relaciones entre ellas.

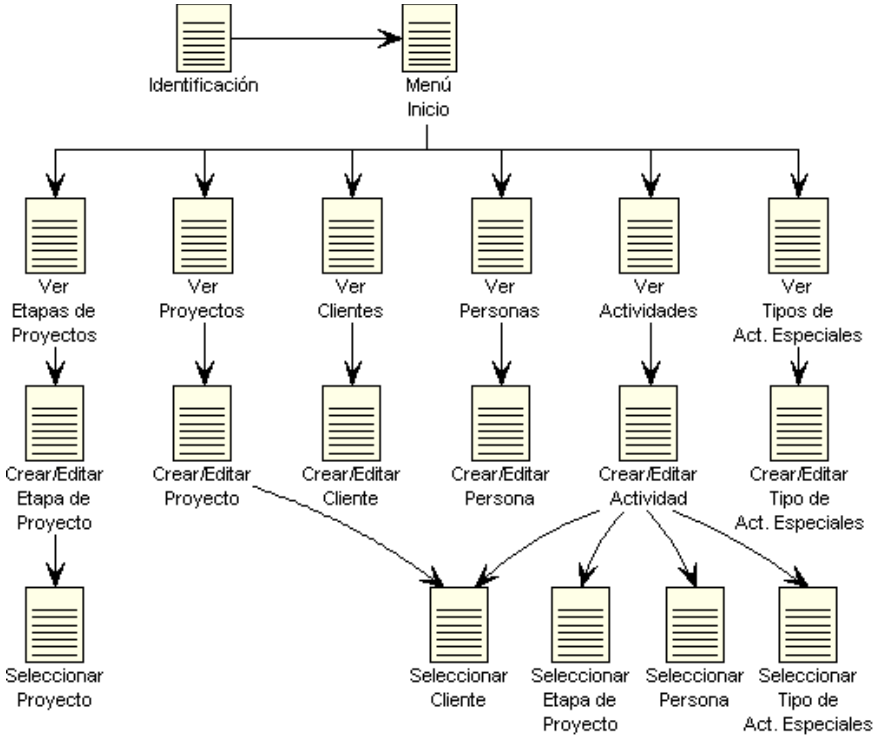

Figura 7.5: Mapa de Navegación para Administrador

El mapa de navegación para el módulo del administrador es mostrado en la figura 7.5. En ella se ve primero una identificación del administrador, luego se ingresa a un menú que enumera las posibilidades del administrador, que será administrar -ver, crear, modificar y borrar- los seis tipos de objetos existentes en el sistema. Para el caso de objetos que mantienen relaciones 1 a N, se tiene la posibilidad de ir a una página especial para seleccionar el objeto relacionado. Por esto se crearon páginas para poder definir la relación de *Proyecto* con *Cliente*, la relación de *EtapaProyecto* con *Proyecto*, y la relación de *Actividad* con *Persona*, *TipoEspecial*, *Cliente* y *EtapaProyecto*.

El mapa de navegación para el módulo de registro de actividades para un usuario se encuentra en la figura 7.6.

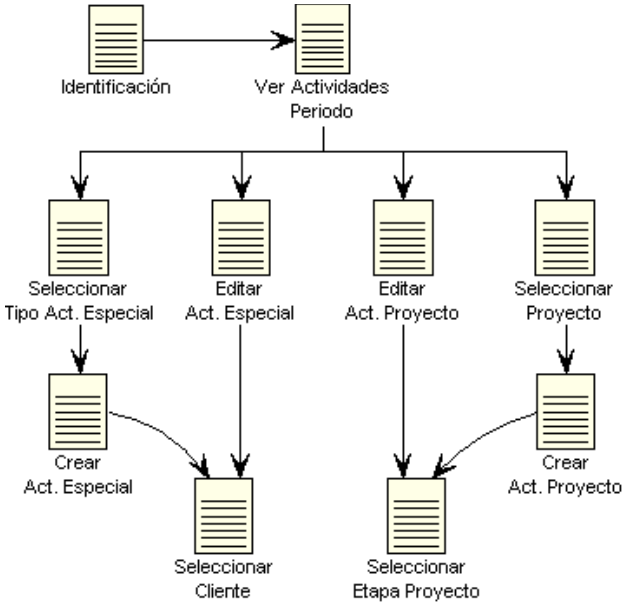

Figura 7.6: Mapa de Navegación para Usuario

Luego de identificarse el usuario, puede acceder a ver sus actividades ingresadas para algún mes en particular. En esa página tendrá la posibilidad de editar una actividad ingresada, ya sea de proyecto o especial, o crear una nueva actividad. Para crear una actividad de proyecto es necesario previamente elegir el proyecto para luego elegir alguna de sus etapas. Para crear una actividad especial es necesario previamente elegir el tipo de actividad especial para luego ingresar los datos correspondientes.

7.4. Implementación

Una vez concluida la etapa de diseño con los datos necesarios ya definidos se procede a implementar el sistema. Para iniciar la implementación fue necesario decidir cuál sería el entorno de desarrollo y las herramientas a utilizar. Luego de esa decisión, la implementación fue dividida en Capa Negocio y Capa Web.

7.4.1. Herramientas de desarrollo

Para comenzar el desarrollo fue necesario contar con J2SE SDK, el cual es la base de J2EE. Se utilizó la versión 1.3.1 de Sun Microsystems la que se utilizó para compilar y ejecutar el software desarrollado.

Como entorno de programación se eligió *Eclipse* 2.0.1, por ser un IDE de distribución

gratuita, código abierto y con buenas características para el programación como por ejemplo soporte para *refactoring* (permite modificar nombres de clases, métodos y variables), compilación de código fuente mientras se escribe (señala errores sin necesidad de compilar una clase), apoyo para la generación automática de código (por ejemplo, generar getters y setters), corrección de errores (por ejemplo, crear bloques try/catch para errores de excepción no capturada o agregar imports para errores de clase no encontrada), y permitir la creación y utilización de plug-ins que otorgan acceso a múltiples funcionalidades.

Elipse es un proyecto de la comunidad *eclipse.org* conformado por empresas proveedoras de software como IBM, Borland, Rational, RedHat y SuSE, que tiene como objetivo crear una plataforma de desarrollo de nivel comercial con código abierto y altamente integrable. Eclipse está estructurado en base a plug-ins que pueden ser creados utilizando una API especial para ello y que pueden ser agregados fácilmente (sólo ubicarlos dentro de un directorio). Así, existe una gama amplia y creciente de plug-ins desarrollados para Eclipse, ya sea por empresas o por usuarios comunes.

De entre ellos el principal plug-in de terceros que se utilizó para el desarrollo del proyecto fue *Lomboz* 0.9 que da apoyo para el desarrollo J2EE, permitiendo definir contenedores Web y contenedores EJB, efectuar debug en un servidor local y realizar el deploy de la aplicación para su instalación en su servidor de producción.

Para apoyar el desarrollo de EJB se utilizó la herramienta *XDoclet* 1.1.2, que permite incrustar información dentro de comentarios de javadoc a través de tags especiales que luego son procesados para efectuar ciertas acciones o configuraciones deseadas. Por ejemplo, en la sección 3.3 se explicó que la creación de un enterprise bean consta de al menos cuatro partes:

1. Crear las interfaces que serán utilizadas por el cliente, que pueden ser hasta cuatro: local, remota, home local y home remota. Para el caso de entity beans además se debe crear la clase que será la llave primaria.

2. Implementar la clase que contendrá la funcionalidad de los métodos declarados en las interfaces, junto con los métodos utilizados por el container para administrar su ciclo de vida.

3. Declarar el bean en el archivo *ejb-jar.xml* detallando el nombre de la clase e interfaces, tipo de transacción a utilizar, permisos de ejecución sobre los métodos y para el caso entity beans CMP las sentencias EJB QL y las relaciones con otros beans.

4. Detallar en archivos XML especiales la información específica al servidor como nombre JNDI para ser publicado, nombre de tablas, columnas y llaves foráneas de las relaciones para entity beans CMP.

XDoclet soluciona esto centralizando toda la información en la clase que implementa el bean, donde se agregan instrucciones especiales al inicio de la clase y en la declaración de cada

método, que son leídas por XDoclet al ejecutar el comando javadoc generando automáticamente las cuatro interfaces necesarias, la clase con la llave primaria, el archivo *ejb-jar.xml* y los archivos de configuración especializados para los servidores soportados, transformándose en una herramienta muy necesaria para el desarrollo de EJB. XDoclet requiere de la utilización de *javadoc* para procesar los tags (que es distribuido con J2SE), y de la herramienta *Ant* para ser ejecutado (que viene incluida en la instalación de Eclipse).

Como servidor de aplicaciones se decidió por *JBoss* 3.0.3 por ser un servidor de aplicaciones de código abierto, licencia gratis, soporte de funciones avanzadas y elaborado completamente en Java, por tanto disponible para múltiples plataformas. La versión utilizada de JBoss contenía a *Jakarta Tomcat* 4.1.12 como servidor web. Existen otras opciones de servidor de aplicación como por ejemplo BEA Weblogic, IBM Websphere, Iona Orbix y Oracle IAS, sin embargo, se prefirió JBoss por ser el servidor de código abierto más utilizado y en general la puerta de entrada a J2EE.

Dado que el proyecto debía ser compatible sobre el sistema ya existente, fue necesario elegir como software de base de datos el utilizado por la empresa, que es *Oracle* 8.1.7. Para acceder a la base de datos de forma directa se utilizó la versión gratuita del software *TOAD* 6.3.7 el cual permite modificar fácilmente los datos existentes en ella, facilitando la creación y modificación de datos como usuarios, esquemas, tablas, columnas y restricciones.

Para el desarrollo de las aplicaciones web se utilizó *Jakarta Struts* 1.0.2, el cual es un framework de código abierto escrito en Java que apoya la creación de sitios web utilizando el patrón de diseño MVC (ver sección 6.1). En Struts el Modelo es otorgado a objetos JavaBeans para datos generales y a objetos *ActionForm* para datos ingresados desde formularios web por el usuario, simplificando su recuperación y validación. La Vista está compuesta por páginas JSP que se encargan de desplegar la información del Modelo utilizando un conjunto de taglibs que facilitan este despliegue. El Controlador es implementado por el servlet *ActionServlet* el cual es configurado a través de un archivo XML para definir sus *ActionMapping*, es decir las relaciones entre dirección web que accede un usuario y objeto *Action* a ejecutar. Un *Action* encapsula la lógica del negocio al procesar datos, interpretar los resultados y entregar el control a la componente Vista para crear la respuesta.

Para acceder a los módulos web del sistema se utilizó Internet Explorer 5.5, el navegador predeterminado de Windows, y *Mozilla* 1.1 el cual es un navegador de código abierto con versiones para múltiples plataformas. Mozilla es el proyecto continuador de Netscape Communicator y se utiliza como base para múltiples navegadores como Netscape y Galeon (browser para Gnome en Linux).

El proyecto fue desarrollado íntegramente una máquina Pentium III de 750 MHz con 320 MB en RAM, utilizando el sistema operativo Windows 2000. En ella se instaló la versión para Windows de Oracle 8.1.7, y luego TOAD para crear el usuario y el esquema que contendría la información del proyecto. Luego instaló la versión J2SE SDK junto con sus bibliotecas y documentación. Después de esto, se instaló JBoss siendo necesario modificar el nombre del directorio donde se instala por defecto Tomcat desde *tomcat-4.1.x* a *catalina* (modificar

archivo *server/default/deploy/tomcat41-service.xml*) para tener compatibilidad con el deploy del plug-in Lomboz de Eclipse, luego se configuró para utilizar Oracle como fuente de datos (crear archivo *server/default/deploy/oracle-service.xml*) y el puerto 8080 como servidor web (modificar archivo catalina/conf/server.xml).

Una vez instalado y configurado el servidor, se procedió a instalar Eclipse junto con sus plug-ins. Se configuró Lomboz para utilizar JBoss, y XDoclet para utilizar javadoc de J2SE y las bibliotecas de Ant desde la instalación de Eclipse. Para finalizar, se instaló Mozilla para ser usado junto con Internet Explorer para acceder al sistema y probar los módulos web. La figura 7.7 muestra un esquema que ilustra las herramientas utilizadas y sus relaciones.

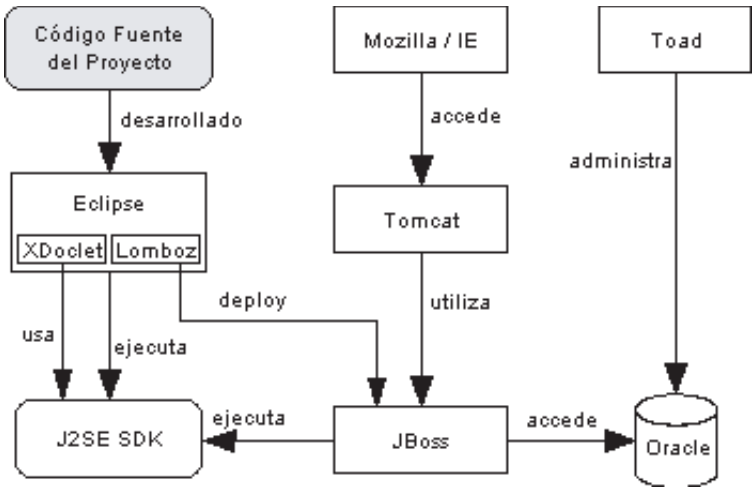

Figura 7.7: Herramientas de desarrollo

Si bien el sistema fue desarrollado en Windows 2000, no existen grandes problemas para ser desarrollado en Linux o Unix, ya que el software principal (J2SE, JBoss, Oracle, Eclipse y Mozilla) tienen versiones para estas plataformas.

7.4.2. Implementación Capa Negocio

La implementación de los EJB está dividida en la implementación de los entity beans y de los session beans.

Entity Beans

Para probar la diferencia de implementación entre BMP y CMP, se realizó una prueba creando un entity bean utilizando BMP (objeto Cliente) y un entity bean utilizando CMP (objeto Proyecto). Para la implementación del bean BMP fue necesario externalizar todo el

código JDBC y SQL dentro de clases especiales de acceso (patrón de diseño DAO). Esas clases crecieron rápidamente en tamaño y complejidad. Para implementar el bean CMP fue de gran utilidad la herramienta XDoclet, facilitando la creación de beans y disminuyendo en forma considerable el tiempo de desarrollo. Sin embargo, hubo problemas para definir una relación entre estos dos beans por ser de distinto tipo, por lo cual se decidió por implementar todos los beans como CMP para probar la tecnología.

Para crear los beans se utilizó el modelo de clases de la figura 7.4 donde se crearon los métodos getter y setter abstractos para cada campo.

Para programar los beans se utilizaron diferentes instrucciones de XDoclet: las de prefijo `@ejb:` corresponden a configuraciones en el archivo *ejb-jar.xml* y las de prefijo `@jboss:` corresponden a configuraciones en los archivos *jboss.xml*, *jbosscmp-jdbc.xml* y *jaws.xml*. En la declaración de cada clase se utilizaron las siguientes instrucciones:

- `@ejb:bean` para definir datos generales como nombre del bean, tipo de persistencia e interfaces a generar (locales y/o remotas).
- `@ejb:transaction` para definir el tipo de transacción a utilizar en el bean. Se utilizó Required (ver sección 5.5).
- `@ejb:finder` para definir un finder a ser agregado a la interfaz home con el nombre de método dado por el parámetro *signature* y la sentencia EJB-QL declarada en el parámetro *query*. Por ejemplo: `@ejb:finder signature="java.util.Collection findAll()" query="SELECT Object(a) from Persona a"` crea un finder que retorna todas las personas existentes.
- `@ejb:select`, análogo a un finder salvo que es para definir el método sólo en la misma clase y luego se debe crear un método abstracto con el mismo nombre que el declarado en el parámetro *signature* para poder utilizarlo.
- `@jboss:table-name` para declarar el nombre de la tabla en la cual el container de JBoss debe efectuar la persistencia del entity bean.
- `@jboss:create-table` y `@jboss:remove-table` para declarar si el container debe intentar crear la tabla en su inicialización o eliminarla al finalizar su servicios.

Luego del encabezado, en cada getter de campo se utilizaron las siguientes instrucciones para definir características sobre el campo relacionado:

- `@ejb:persistent-field` para que el campo sea agregado al archivo *ejb-jar.xml* como perteneciente al esquema de persistencia abstracto del bean.
- `@ejb:pk-field` utilizado si el campo pertenece a la llave primaria del bean agregándose al objeto PK que será generado automáticamente.

- `@jboss:column-name` para agregar la relación entre este campo y la columna de la tabla en la base de datos.

- `@ejb:relation` y `@jboss:relation` para declarar datos generales de la relación como el nombre y el tipo de relación (se utilizaron relaciones 1 a N unidireccionales), y datos específicos al container como el nombre de la columna que mantiene la relación.

Los métodos que pertenecían a la interfaz de cliente (local o remota) se les marcaba con la instrucción `@ejb:interface-method`, los métodos de la clase home con `@ejb:home-method` y los de creación con `@ejb:create-method`. Para más detalles en la utilización de las instrucciones XDoclet se puede ver el código fuente listado en el apéndice A.2.

Además se utilizó XDoclet para generar una clase especial para cada bean que contenía las instrucciones necesarias para obtener una instancia a través de JNDI evitando contener este tipo de código en archivos de implementaciones, lo que es una aplicación del patrón *Service Locator* (ver sección 6.4).

En el desarrollo de los entity beans fue necesario lidiar con tres problemas principales: valores nulos para tipos primitivos, generación de llaves primarias e implementación de relaciones obligatorias.

El primer problema consistió en que las columnas tipo *number* de la base de datos fueron modeladas como campos tipo *int* o *double* en los entity beans y al estar desarrollando el proyecto no se encontró forma para dar un valor *null* a estos campos. La solución implementada fue utilizar los objetos Java correspondientes a los tipos primitivos, así las columnas tipo *number* fueron modeladas como campos tipo *Integer* o *Double* dependiendo si la columna aceptaba decimales o no.[1] Así para dar un valor null a una columna, basta con asignar la referencia al objeto correspondiente como null.

Las soluciones del segundo y tercer problema fueron más complejas, ameritando un estudio más profundo de cada problema y su solución, las que son presentadas como patrones de diseño por ser genéricas y reutilizables en situaciones similares.

Patrón de diseño para generación de llaves primarias

La generación de llaves primarias es un problema frecuente para entity beans, y se discuten ciertas soluciones en [5], sin embargo estas soluciones no son adecuadas para el proyecto, debido a la restricción dada por una base de datos ya definida donde debe utilizarse un *sequence* como generador de llaves, e integrándose con otros sistemas independientes que utilicen las mismas tablas.

[1] Inicialmente se pensó en utilizar campos tipo *Float* para columnas con decimales, sin embargo debido a que las operaciones básicas en Java son implementadas para int o double se prefirió por utilizar *Double* para evitar transformaciones innecesarias.

Una solución posible -las más directa- es consultar primero por el valor de la secuencia y luego crear el entity bean con ella, lo que se traduce en ejecutar una sentencia extra para cada creación de un entity bean. La solución implementada es una mejora de ésta, solicitando un conjunto de valores de cada secuencia y manteniéndolos en memoria, así cuando se necesita un valor de secuencia se obtienen de inmediato K valores en una sola consulta los que son asignados a las próximas solicitudes. Para lograr la implementación fue necesario crear un procedimiento almacenado que recibe un nombre de secuencia, toma K valores y los retorna en un sólo valor con separadores definidos, que luego son procesados para obtener los valores originales.

Debido a que el objeto que otorgará las llaves primarias, llamado EjbDAO, deberá ser compartido por todos los entity beans, hay que adecuarlo para la ejecución paralela de múltiples threads sobre él. Para esto se ejecuta *synchronized* sobre el nombre de la secuencia a ejecutar[2] logrando que sólo las llamadas sobre una misma secuencia sean exclusivas. Para ver el código de la implementación ver el apéndice A.1.

La solución dada debe funcionar bien incluso para sistemas distribuidos donde existen múltiples máquinas virtuales en ejecución, aunque quizás menos eficiente al mantener un conjunto de llaves primarias para cada JVM y la inserciones en la base de datos no ocurrirán según orden correlativo de identificador.

Patrón de diseño para relaciones obligatorias en JBoss

El problema con la implementación de relaciones obligatorias reside en la separación en dos pasos para la creación de un entity bean, dados por los métodos *ejbCreate* y *ejbPostCreate* (ver sección 3.3.5). La implementación efectuada por JBoss consiste en ejecutar el método *ejbCreate* para asignar los campos, insertar la fila en la base de datos, ejecutar el método *ejbPostCreate* para asignar relaciones y luego actualizar la fila en la base de datos agregando los nuevos datos. Sin embargo, cuando existe una relación obligatoria, la columna correspondiente en la base de datos se encuentra con la restricción *not null*, lo que hace fallar la implementación de JBoss en la inserción inicial[3]. En los foros de JBoss y en su documentación se señala que esta limitación será eliminada en una versión futura [19], pero no se dan soluciones posibles más allá de modificar la tabla.

La solución implementada consiste en simular la relación obligatoria agregando un nuevo campo en el entity bean con la llave primaria del objeto relacionado, que es asignado en el *ejbCreate*, y luego implementar el método get de la relación como un *ejbSelect* interno de la clase, y el método set como una asignación de la llave primaria del objeto al campo

[2]La solución exige que el parámetro recibido sea siempre el mismo objeto, por tanto basta con que el String que contiene el nombre de la secuencia que es entregada como parámetro sea una constante, por ejemplo declarándola como *static final*.

[3]Una solución existente para este problema en algunos servidores de aplicación es efectuar un preproceso a las sentencias previo a ejecutarlas en la base de datos.

que mantiene la relación. Por ejemplo para simular la relación obligatoria de Proyecto con Cliente es necesario utilizar un código similar a este:

```
//getter del campo que mantendrá la relación
public abstract Integer getClienteId();

//setter del campo que mantendrá la relación
public abstract void setClienteId(Integer clienteId);

//getter de la relación, deja de ser abstract
//es necesario crear un select para obtener el objeto dado el identificador
public Cliente getCliente() {
  return (Cliente) ejbSelectCliente(getClienteId()).iterator().next();
}
//setter de la relación, deja de ser abstract
//obtiene el campo que corresponde a la llave primaria del objeto y lo asigna al bean
public void setCliente(Cliente c) {
  setClienteId(((ClientePK) c.getPrimaryKey()).getId());
}

//select para obtener el objeto Cliente dado su identificador
//el EJB QL que ejecutará será este: SELECT OBJECT(c) FROM Cliente c WHERE c.id= ?1
public abstract java.util.Collection ejbSelectCliente(Integer clienteId);
```

Cabe señalar que se ha eliminando el manejo de excepciones para simplificar el código fuente aquí presentado. Con la modificaciones hechas al bean, el campo *cliente* deja de pertenecer al esquema de persistencia abstracto de Proyecto y es agregado el campo *clienteId*. Luego, la instrucción *setCliente* debe ser transferida de *ejbPostCreate* a *ejbCreate* para solucionar el problema de la relación obligatoria. En el apéndice A.2 se puede ver la implementación dada para el entity bean Actividad que contiene una relación obligatoria con Persona y relaciones opcionales con EtapaProyecto, TipoEspecial y Cliente.

Session Beans

Una vez terminada la implementación de entity beans, se procedió a implementar los session beans definidos en la etapa de diseño. De la herramienta XDoclet se utilizaron principalmente las instrucciones `@ejb:bean` y `@ejb:transaction` en el encabezado de cada bean, y para los métodos se utilizaron `@ejb:interface-method` o `@ejb:create-method`. Los beans implementados son cuatro:

- ValidacionBean.java: Stateless bean que contiene métodos de validación como *validarPersona(username, password)* retorna el correspondiente entity bean Persona si fue validado correctamente o null si no.

- RegistroActividadesBean.java: Stateful bean inicializado con el entity Persona. Contiene métodos como *crearActividadProyecto* y *crearActividadEspecial* que reciben los parámetros necesarios para crear cada tipo de actividad, *borrarActividad* que recibe el identificador o el objeto a borrar, *modificarActividadProyecto* y *modificarActividadEspecial* que recibe la actividad original y los nuevos datos a ingresar; y *buscarActividades* que retorna todas las actividades de la persona. En cada modificación solicitada se verifica que la Actividad a borrar o actualizar corresponda a la Persona que inicializó el bean.

- AdministracionBean.java: Stateless bean que contiene métodos para crear, borrar y modificar cada uno de los seis entity beans existentes. Además contiene el buscador de Actividades ya que solo el administrador puede ver todas las actividades existentes.

- VerInformacionBean.java: Stateless bean con métodos para buscar entity beans según distintos parámetros. Por ejemplo, *buscarPersonas* retorna el conjunto de Personas existentes, *buscarPersonaByUsername* retorna el bean Persona correspondiente al username entregado, y *buscaPersona* retorna el entity bean Persona dado su llave primaria. No contiene buscadores de Actividades ya que un usuario no puede visualizar las actividades de otras personas.

Estructura Capa Negocio

La figura 7.8 resume la estructura utilizada para la implementación la capa de negocio en el proyecto.

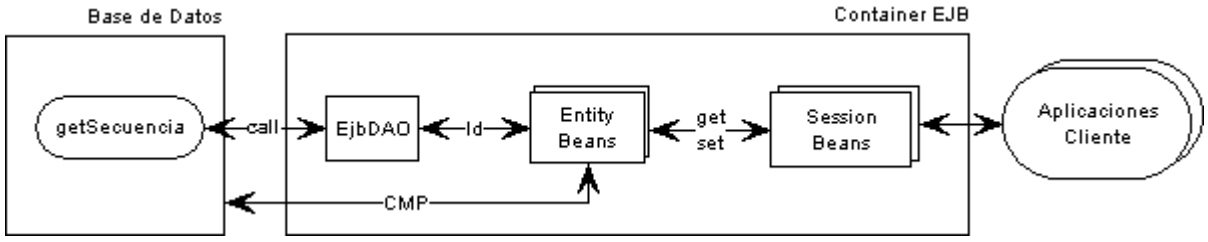

Figura 7.8: Diagrama Capa Negocio

La estructura está compuesta de tres subcapas: la primera está compuesta por el objeto DAO que es el encargado de obtener los números correlativos correspondientes a las llaves primarias de cada tipo de entity bean. Para lograrlo debe ejecutar un procedimiento existente en la base de datos que retorna un conjunto de llaves, las cuales distribuye. La segunda subcapa está compuesta por entity beans cuya persistencia es administrada por el container, y que al momento de ser creadas deben solicitar un identificador único al DAO. La tercera subcapa está compuesta por los session beans que forman la Session Façade y que utilizan y modifican entity beans para proporcionar las funcionalidades a las aplicaciones clientes.

7.4.3. Implementación Capa Web

Para efectuar la implementación de un sitio utilizando el framework Struts es necesario seguir dos pasos principales:

- Definir cada pantalla que pertenecerá al sistema, en particular con sus requerimientos en términos de datos desplegados e ingresados. Corresponde a refinar las páginas existentes en el mapa de navegación y determinar los ingresos de datos por parte del usuario.
- Definir todas las trayectorias que existirán entre las páginas, detallando las posibles formas de entrada y salida de cada una. Corresponde a refinar las relaciones entre páginas existentes en el mapa de navegación.

Con el primer paso se consigue definir lo que corresponderá a las clases *ActionForm* del sistema, es decir, los formularios en los cuales se guardará información y que en general corresponderá a datos ingresados por el usuario. El segundo paso permite determinar el *ActionMapping* básico del sitio, es decir, el conjunto de clases *Action* mínimo que será necesario implementar para el sistema.

Luego de haber sido refinado el mapa de navegación se constató que esencialmente existían dos tipos de páginas: navegadores de registros y de modificación de datos. Estos tipos de páginas fueron abordados en forma estándar creando soluciones reutilizables detalladas en a continuación.

Cabe señalar que para la creación de las páginas JSP fue necesario realizar un pequeño diseño gráfico de la página decidiendo imágenes a desplegar, colores a utilizar y tipos de letras y tamaños. Esta tarea puede haber sido encargada a una persona especializada y el resultado ser utilizado en el sistema sin grandes problemas gracias a la separación de la visualización otorgada por el patrón MVC.

Patrón de Diseño para Navegación de Registros

Para la navegación de registros es deseable tener la posibilidad de filtrar datos, ordenarlos según el valor de algún campo y dividirlos por páginas. Para satisfacer estas funcionalidades en forma estándar se creó el objeto *ActionFormFiltro*.

ActionFormFiltro recibe como parámetro un conjunto de objetos, los cuales recorre uno a uno invocando el método filtrar abstracto que debe ser implementado por las clases que lo utilicen. Los objetos resultantes los ordena según haya decidido el usuario y una vez finalizado ve cual es la sección de datos que está viendo el usuario para retornar el subconjunto de datos correspondiente a la página visible.

ActionFormFiltro extiende de *ActionForm* debido a que los datos de los filtros, el orden y las páginas son ingresadas por el usuario. Además implementa un conjunto de funciones para simplificar la implementación de los filtros como igualdades case-insensitive o textos similares tipo *like*.

Para la creación de un filtro basta heredar este objeto y definir los campos a filtrar e implementar los métodos abstractos de comparación y filtrado de objetos específicos a los campos definidos. Luego, dada una lista de entity beans obtenida de una consulta a un session bean, *ActionFormFiltro* se encarga filtrarlos, ordenarlos y permitir la navegación de los resultados en páginas de largo definido en la clase que lo extiende.

Para el despliegue en la pantalla de los datos, fue necesario crear un conjunto de *taglibs* para JSP, que se encargaran de mostrar los links para efectuar el ordenamiento, mostrar una imagen para señalar el tipo de ordenamiento aplicado y desplegar la numeración de páginas inferior.

En la figura 7.9 se muestra el navegador de proyectos, donde se puede filtrar por código, nombre del proyecto y nombre del cliente, se puede ordenar en forma ascendente y descendente por cada una de las columnas desplegadas y se puede visualizar los resultados a través de la barra de navegación inferior.

Figura 7.9: Navegador de registros de Proyectos

Patrón de Diseño para edición de Entity Beans

Para el caso de las páginas de modificación de datos, se tiene un entity bean con el registro seleccionado del navegador y se quiere ocupar un formulario para que el usuario ingrese o modifique sus datos. Para esto es necesario crear un *ActionForm* y copiar los datos del entity bean a él[4]. El ActionForm es utilizado para capturar los datos desde el usuario, los que deben ser validados y luego ingresados al entity bean correspondiente.

Como solución reutilizable para esto, se creó un objeto llamado *ActionFormEntity* que extiende de ActionForm que se encarga de dar esta funcionalidad de forma genérica, un objeto *TipoCampo* que se encarga de validar los formatos y transformar a objetos los campos ingresados por el usuario, y un objeto *RefManager* que se encarga de mantener referencias a objetos como el entity bean del cual provienen los datos y los objetos a ser asignados a las relaciones del bean.

Así, para crear el formulario que editará los datos contenidos por un entity bean basta con crear una clase que herede de *ActionFormEntity*, definir los campos que serán actualizados asignándole el mismo nombre que en el entity bean pero declarándolo de tipo String -a través de formularios web sólo se reciben Strings-, crear los getter y setter de cada campo y definir el tipo de cada campo a través de constantes del objeto *TipoCampo*. Las validaciones y el copiado de campos desde y hacia el entity bean son realizadas automáticamente utilizando el paquete de reflexión de Java para buscar los métodos de cada objeto e invocarlos. El código fuente de la clase *ActionFormEntity* se encuentra en el apéndice A.3.

El objeto *TipoCampo* está encargado de cumplir dos funciones: dado un objeto y un tipo, lo transforma a su versión en String para poder ser presentado al usuario; y dado un String y un tipo, verifica que el formato sea el correcto y crea el objeto correspondiente que será asignado al entity bean. En caso que los datos recibidos sean inválidos según el tipo especificado, es emitida una excepción que es traducida en un mensaje al usuario en el campo con problemas (ver figura 7.10).

El objeto *RefManager* necesita como parámetro la sesión del cliente para guardar en ella referencias a objetos dentro del formulario. Por ejemplo, para la edición de un Proyecto, el usuario ingresa datos y debe elegir un Cliente. Al hacerlo el formulario que estaba editando es guardado en la sesión para ser dirigido a la página de selección de Clientes. Cuando el usuario elige un Cliente, vuelve a la edición de Proyectos donde es recuperado el formulario inicial y es agregada la referencia del objeto Cliente a la sesión para luego ser ingresada al entity bean de Proyecto al presionar el botón Actualizar.

[4]No es buena idea utilizar el propio entity bean para llenar el formulario, ya que cualquier ingreso del usuario será enviado al entity bean en forma directa, modificándolo inmediatamente sin dar la posibilidad de efectuar validaciones previas.

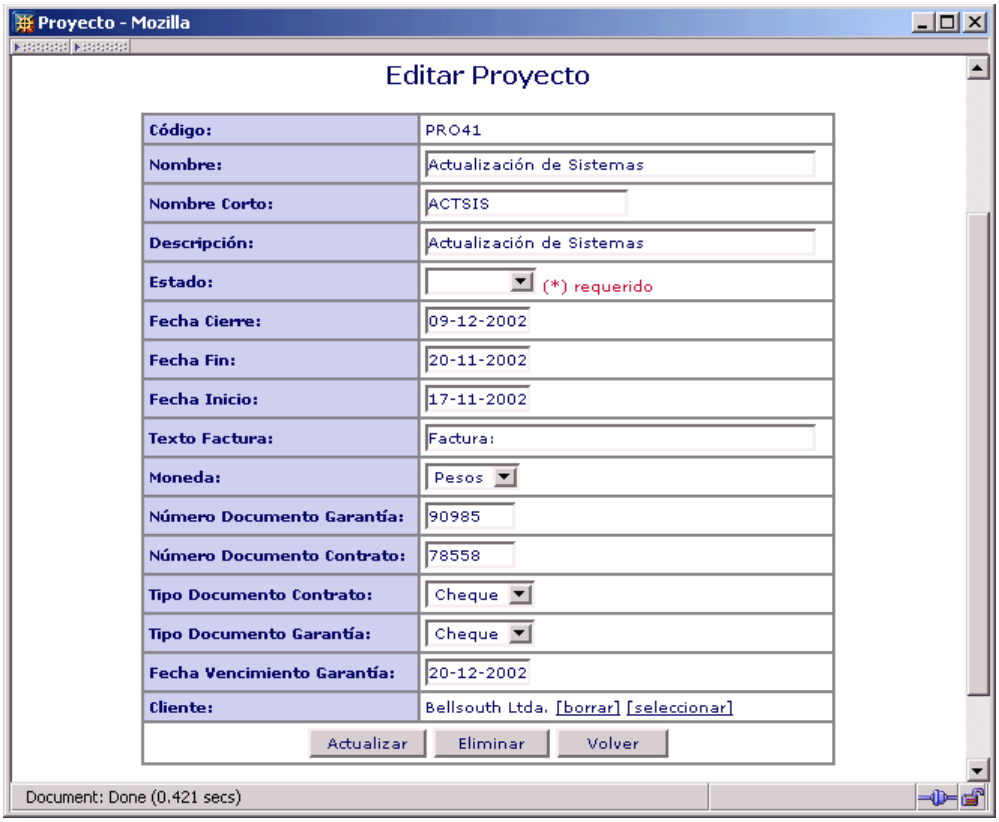

Figura 7.10: Edición de un registro de Proyecto

Estructura Capa Web

En la figura 7.11 se ilustra la estructura utilizada para desarrollar un módulo Web. Cada uno de los dos módulos web desarrollados contienen la misma estructura.

El proceso de una solicitud de un cliente se inicia cuando se recibe un requerimiento web del cliente que es tomado por el controlador de Struts y delegado sobre el objeto Action correspondiente según la dirección web solicitada.

Este objeto Action es el encargado de reunir la información necesaria y ejecutar la acción correspondiente. Para esto, probablemente necesitará una instancia de uno o más session beans que puede conseguir a través del objeto *EJBServices*. *EJBServices* se encarga de mantener instancias locales de session beans y asignarlas según se solicite, y para el caso de la primera solicitud utiliza JNDI para conseguir una referencias en el depósito del container y la mantiene como local. Una vez conseguida la referencia a el o los session beans, el objeto Action ejecuta el proceso para llevar a cabo la solicitud del cliente, luego obtiene los datos necesarios para la visualización creando el objetos ActionFormFiltro para el caso de navegadores o ActionFormEntity para editores. Con los datos para la visualización ya definidos delega la ejecución sobre la página JSP adecuada.

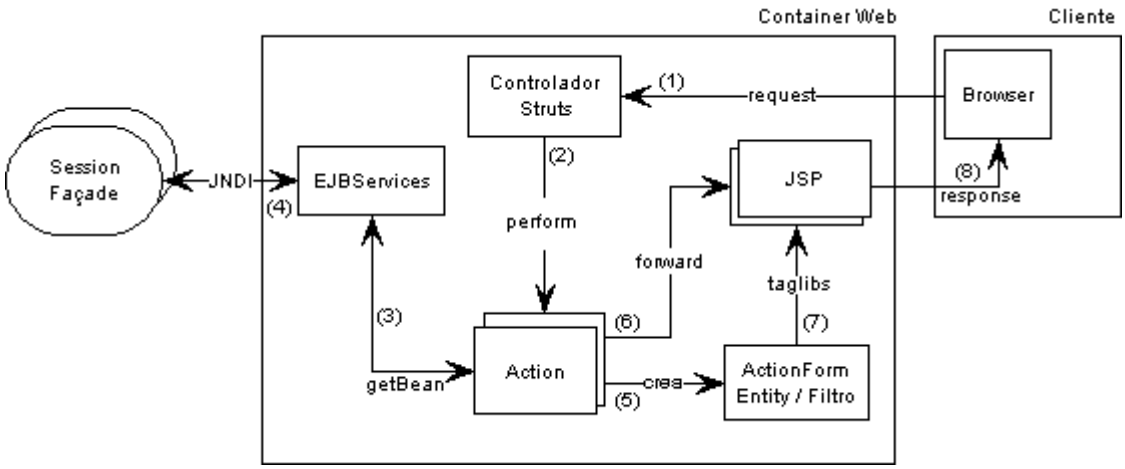

Figura 7.11: Diagrama módulo Web

La página JSP es la encargada de reunir la información ya obtenida y generar la visualización que será la retornada al cliente. Para esto deberá obtener los objetos otorgados por el Action utilizando los taglibs provistos por Struts y los creados en el desarrollo del proyecto. Una vez reunida la información y ejecutado el JSP, la respuesta es enviada al cliente para que tome su próxima acción.

Módulo para el administrador

Una vez creados los elementos reutilizables para apoyar la creación de páginas, se procedió a utilizarlos sucesivamente para desarrollar el sitio junto con un conjunto de taglibs para apoyar la visualización.

Así se crearon páginas de identificación y menú principal, las páginas de navegación de información de Clientes, Personas, Proyectos, Etapas de Proyectos, Actividades y Tipos de Actividades (ver figura 7.9) con sus correspondientes objetos *ActionFormFiltro* y *Action* encargados de obtener los datos, ejecutar el filtro y direccionar al JSP, las páginas de edición de cada uno de estos seis tipos de datos (ver figura 7.10) con sus objetos *ActionFormEntity* y *Action* encargados de invocar los session beans y dirigir al JSP correspondiente según la acción del usuario, y la página de salida del sistema.

Módulo para el usuario

Nuevamente se utilizaron los objetos para la navegación y edición registros. El módulo estaba compuesto de dos editores de registros -uno para actividades especiales y otro para actividades de proyecto-, cinco navegadores de registros -de actividades, de proyectos, de

etapas de proyectos, de tipos especiales y de clientes-, y la página de ingreso y de validación. Se utilizó también los taglibs creados para el desarrollo del módulo administrador.

7.5. Deploy

La etapa de deploy consiste en ubicar el sistema en su etapa de producción. Dado el entorno de desarrollo utilizado para llevar a cabo el proyecto, el deploy se efectuaba constantemente sobre JBoss, por lo cual para ubicar el sistema en su lugar final bastó con modificar el tipo de compilación para eliminar datos de debug de los byte-codes. El plug-in Lomboz de Eclipse permite efectuar el deploy sobre JBoss sin problemas presionando sobre un botón.

JBoss tiene la característica conocida como *hot deploy* con la cual, al instalar una nueva aplicación en su directorio de deploy, la reconoce inmediatamente y la instala sin necesidad de reiniciar el servidor. Esta característica funciona bien para páginas JSP de módulos web, sin embargo, no funciona bien para módulos ejb, porque no finaliza los servicios antiguos para reemplazarlos por los nuevos, debiendo ser reiniciado JBoss en este caso, lo que tomaba un tiempo aproximado de 30 segundos.

Capítulo 8

Discusión y Conclusiones

El trabajo realizado ha sido amplio: se ha adquirido conocimientos sobre J2EE, sus componentes y sus servicios, se han estudiado patrones de diseño existentes para el desarrollo de aplicaciones, se ha desarrollado un sistema funcional aplicando lo estudiado y utilizando los patrones existentes, y se han creado cuatro patrones de diseño nuevos con soluciones a los problemas encontrados en la implementación, por tanto las conclusiones son múltiples y de variados aspectos. Después de éstas, se realiza una discusión sobre las posibles alternativas existentes a J2EE, para finalizar con un conjunto de trabajos posibles para continuar y complementar esta investigación.

8.1. Conclusiones

Si bien J2EE es relativamente nuevo, ha evolucionado rápidamente para transformarse en una buena solución para desarrollar aplicaciones empresariales por cumplir necesidades avanzadas como escalabilidad, disponibilidad, estandarización, integración, consistencia y seguridad. Su creciente éxito puede verificarse en el progresivo aumento de la cantidad y calidad de software ligado a J2EE, el aumento en el número de empresas y sitios utilizándolo y los cada vez mayores requerimientos del mercado por desarrollo en J2EE.

J2EE utiliza una arquitectura basada en cuatro capas: un cliente, un servidor web, un servidor de aplicaciones y una base de datos. Si bien la arquitectura basada en tres capas es el enfoque más utilizado hasta ahora, que también puede ser implementado con J2EE, una arquitectura de cuatro capas puede proveer mayor disponibilidad y escalabilidad, aunque estos beneficios tienen el costo de aumentar la complejidad para el diseño y el desarrollo del sistema.

J2EE, al ser una especificación, no está ligado a ningún proveedor en particular, por lo cual una aplicación desarrollada apegada a los estándares puede ser instalada en distin-

tos servidores de aplicación sin mayores dificultades. Sin embargo, esto hace desarrollar una aplicación bajo el mínimo común denominador de las características de los servidores, eliminando las capacidades propias provistas por cada software. Tomar la decisión entre usar cierta funcionalidad o evitarla, generalmente significa ligarse a un proveedor en particular o preferir la estandarización y la libertad de elección. Esta es una elección no menor que debe ser analizada previamente.

Elegir cual será el software que centralizará toda la lógica del negocio en el servidor de aplicaciones es una decisión difícil. En el mercado actualmente existe una cantidad creciente de servidores de aplicación, que pueden ser divididos entre los que hay que cancelar una suma de dinero para utilizarlos y los de distribución gratuita. Para una empresa que se está iniciando en J2EE, los servidores gratuitos son una buena opción por reducir notablemente los costos de desarrollo de los sistemas iniciales, sin embargo, cuando el servidor de aplicaciones tome mayor importancia dentro de la empresa, la disponibilidad y escalabilidad sea fundamental, y no se desee asumir el riesgo de posibles fallas prolongadas, será necesario o bien comprar un servidor de aplicaciones de algún proveedor establecido y migrar los sistemas desarrollados para garantizar que existirá soporte técnico al cual recurrir en caso de falla, o bien contratar alguna de las empresas conocidas como *Application Service Providers (ASP)* que proporcionan estos y otros servicios en forma externa.

EJB es parte fundamental para la arquitectura de J2EE al ser la componente que mantiene la lógica del negocio de la empresa. Su diseño es complejo y se requiere de conocimientos y experiencia para realizar un modelamiento correcto y eficiente, lo que hace aumentar su costo de desarrollo. EJB debe ser visto como una inversión a futuro, como una decisión a nivel gerencial de la forma de desarrollar el área informática de la empresa. La decisión de utilizarlo debe ser consistente en el tiempo para justificar la inversión que se realizará en su desarrollo.

No se debe diseñar EJB como uno de los módulos para desarrollar un proyecto web, ya que es subdimensionar EJB tanto en capacidades como en tiempo de desarrollo y complejidad, lo que puede ser causal directa del fracaso del proyecto. EJB ha sido diseñado para ser utilizado por muchos sistemas que necesiten datos de la empresa, siendo el punto que contiene la lógica del negocio que es ejecutada por diferentes software de acceso, ya sean aplicaciones web, programas stand-alone o applets, para dar acceso a los clientes del sistema manteniendo el mismo el nivel transaccional y de seguridad. Por tanto, el desarrollo de EJB es un proyecto en sí mismo, que luego de finalizado permite crear diferentes formas de acceso a él para cada uno de los diferentes clientes.

Al centralizar la lógica del negocio se debe tomar particular atención en los niveles de seguridad que existirán en el sistema desarrollado. Si no se toman las precauciones del caso se puede exponer gran cantidad de información a terceros, sin embargo, si se realiza un desarrollo seguro se puede mejorar el nivel de seguridad del sistema al estar todo el manejo de la información en un solo lugar, evitando mantener múltiples sistemas de menor tamaño. Para facilitar esto, EJB permite utilizar diferentes roles de acceso para restringir la utilización de cada bean e incluso restringir a nivel de cada método.

Al modelar el negocio a través de EJB se debe tener en cuenta el modelo relacional existente en la base de datos. Un buen modelo de negocio debe ser un punto de equilibrio entre la granularidad utilizada para modelar entity beans y el desempeño obtenido en las consultas a la base de datos. Esta técnica requiere de gran experiencia y conocimiento de EJB y de base de datos, lo que puede significar un alto costo para una empresa.

El desarrollo de aplicaciones web con J2EE, compuesto por servlets y JSP, es una buen enfoque la creación de sitios si es utilizado en forma correcta. Para desarrollar un sitio web en forma modular, los servlets sólo deben ser utilizados para procesamiento de información y lógica de la aplicación, para delegar sobre JSP la lógica de la presentación. Por esto, un servlet no debe responder al cliente con instrucciones de presentación, ni un JSP contener referencias al lenguaje si no que cualquier procesamiento necesario debe ser incluido en un servlet o en tags personalizados y reutilizables entre páginas. Struts, una vez comprendido, facilita el desarrollo de buenas aplicaciones web al prácticamente obligar al desarrollador a crear el sitio en pequeños módulos diferenciando entre el modelo y la visualización. Struts puede mejorar la calidad del desarrollo de un sitio facilitando la asignación de tareas específicas a personas especializadas, sin embargo, es un framework cuyo aprendizaje no es fácil ya que modifica la metodología de construir un sitio, lo cual es necesario tomar en cuenta previo a decidir su uso en una aplicación web.

Existe una cantidad creciente de patrones de diseño para el desarrollo en J2EE, sin embargo, hay que diferenciar entre los patrones de diseño generales y los más específicos. Para el proyecto se utilizaron dos patrones de alto nivel que definen la arquitectura global del sistema: capas y MVC. El primer patrón es la base de J2EE y fue utilizado para la implementación de cada componente creando componentes especializadas, con pocas dependencias entre sí, implicando la utilización de otros patrones de diseño más específicos como Session Façade y DAO. El segundo patrón separa los datos, la visualización y relación entre ellos, permitiendo la división del trabajo en roles específicos y facilitando la utilización de múltiples formas de acceso.

El proyecto realizado para el presente trabajo fue relativamente simple, sin embargo, no era el objetivo desarrollar un gran proyecto si no que uno relativamente menor en el cual hacer una prueba de las características de J2EE y buscar una forma concreta para llevar a cabo un proyecto J2EE. Así se encontró un buen conjunto de herramientas para el desarrollo dado por Eclipse, JBoss, Struts y XDoclet, una metodología para afrontar un proyecto J2EE, dada por la definición de casos de uso, diagrama de clases y mapas de navegación, y se crearon un conjunto de patrones de diseño útiles para la implementación de un sistema.

Para el desarrollo del proyecto tuvo una gran incidencia el requerimiento de integración con el sistema existente, lo que impidió aplicar alguna de las soluciones existentes a los problemas de generación de llaves primarias y relaciones obligatorias en JBoss, siendo necesaria la investigación y creación de soluciones nuevas. En caso contrario se hubiera utilizado una solución ya existente para la generación de llaves primarias y se habría modificado la base de datos para declarar todas las relaciones como opcionales.

8.2. Alternativas a J2EE

Existen pocas alternativas que compitan directamente con J2EE, debido a que muchos proveedores en vez de crear productos que similares a J2EE, están escogiendo implementar una plataforma J2EE-compatible. En particular J2EE tiene dos posibles competidores: Microsoft y OMG.

OMG ha definido una infraestructura distribuida basada en ORBs (ver sección 5.3), ha creado un conjunto de servicios distribuidos llamados CORBAservices y ha definido una componente similar a EJB llamada *CORBA Component Model (CCM)* que es independiente del lenguaje. Si bien CORBA puede ser considerado un competidor de J2EE, en la actualidad es más bien complementario ya que todas las plataformas J2EE-compatibles deben implementar su protocolo de comunicación a través del cual se asegura operabilidad con él, y debido a que CCM será una versión de EJB independiente del lenguaje, una aplicación CCM-compatible podrá utilizar objetos EJB sin mayores modificaciones.

Microsoft es el único proveedor que ofrece una plataforma para desarrollo de aplicaciones empresariales que compite directamente con J2EE. Su producto es conocido como *.NET*, anteriormente llamado DNA, que es un modelo de desarrollo para aplicaciones empresariales para la plataforma Windows. Existen analogías directas entre .NET y J2EE como ADO, ActiveX, ASP, COM+ y MTS, para JDBC, applets, JSP, EJB y JTA, respectivamente. Sin embargo, existen diferencias entre .NET y J2EE, siendo la diferencia principal la independencia del lenguaje de .NET v/s independencia del proveedor de J2EE.

J2EE es una plataforma independiente de un proveedor, así una aplicación que cumple con los estándares de J2EE puede ser ejecutada en cualquier servidor de aplicaciones J2EE-compatible, sin embargo, tiene una limitación, las aplicaciones J2EE deben ser implementadas en Java.

Microsoft .NET es una plataforma independiente del lenguaje, así los desarrolladores pueden elegir el lenguaje y las herramientas a utilizar para implementar aplicaciones .NET. Entre los lenguajes a elegir se encuentran Visual Basic, Visual C++, Visual J++ y Visual C#, esto se logra transformándolos a un lenguaje intermedio independiente del lenguaje de programación (IL). Sin embargo, para ser ejecutado se debe utilizar la plataforma Windows y su software como IIS, DTC, OLE DB e Internet Explorer si se utilizan ActiveX.

J2EE es una plataforma más madura, se creó primero y su avance es muy rápido dados los aportes de muchas personas a través del mundo que apoyan el software Java. Por otra parte Microsoft .NET ofrece una solución integrada en la plataforma Windows, donde Microsoft hace el esfuerzo en asegurar que sus productos pueden trabajar juntos sin mayores problemas de configuración.

Como un competidor menor de J2EE puede considerarse la solución típica utilizada en el mundo GNU: Linux, Apache, PHP o Perl, y MySql o PosgreSQL. En ella, el desarrollo es más

simple por necesitar menores conocimientos técnicos que para J2EE (distribución, mensajería, servicio de nombres, transacciones, etc.), sin embargo, apunta a aplicaciones de menor tamaño que no necesitan de grandes necesidades transaccionales ni alta disponibilidad. Sin embargo, para aplicaciones no empresariales que no necesitan de cuatro capas, esta solución es una buena opción para tomar en cuenta.

Dada las distintas tecnologías para desarrollar aplicaciones, es que se han creado métodos para poder relacionarlas. Éstas son conocidas como tecnologías de integración de aplicaciones, y están compuesta principalmente por CORBA y Servicios Web. CORBA permite la comunicación de aplicaciones independiente de la arquitectura, el sistema operativo, el lenguaje de programación y el tipo de red utilizada (ver sección 5.3), si bien su ofrecimiento es bueno, no ha logrado posicionarse como la mejor solución existente ya sea por su complejidad o por las pocas implementaciones existentes. Los Servicios Web son un conjunto de tecnologías que utilizan XML para comunicarse e intercambiar información, usando un protocolo de comunicación estándar basado en XML conocido como SOAP. Los Servicios Web son muy recientes, pero han logrado una buen éxito inicial por comunicarse a través de HTTP.

8.3. Trabajo futuro

Entre los aspectos futuros que pueden continuar esta investigación se encuentran:

- Realizar una prueba de comparación entre la variedad de servidores de aplicación existentes. Así se podrán conocer las capacidades y debilidades de cada uno, lo que servirá de apoyo para tomar la importante decisión de elegir un servidor de aplicación para una empresa. Por ejemplo, en la actualidad se nombra a BEA Weblogic como el mejor servidor dado su conjunto de características, lo cual puede ser cierto o no.
- Realización de una aplicación empresarial para ser puesta en producción. En ella se podrán estudiar las capacidades de J2EE para un proyecto real, los problemas que aparecen y posibles formas de solucionarlos. Con esto, se tendrá la experiencia de una aplicación real en uso, donde se puedan probar características como distribución, escalabilidad, concurrencia, seguridad y mensajería en un medio ambiente real. Por ejemplo, para el proyecto realizado en este trabajo no se encontró una funcionalidad que pudiera ser modelada por message-driven beans, sin embargo, en una aplicación empresarial es probable que exista una tarea que deba ser modelable por este tipo de enterprise bean. Este trabajo es complementable con el punto anterior, ya que para realizar una aplicación empresarial en algún momento se deberá tomar la decisión del servidor de aplicaciones a utilizar.
- Elaborar un estudio comparativo sobre las plataformas para desarrollo de aplicaciones empresariales y particularmente comparar y evaluar J2EE y Microsoft .NET. Estos son los competidores que en la actualidad están ofreciendo soluciones para empresas, y que

para tomar una buena decisión debe ser mejor conocidos en sus similitudes y diferencias para elegir no sólo por la publicidad de cada uno.

- Estudiar la integración de sistemas otorgado por CORBA y Servicios Web. CORBA ha promovido la integración desde sus inicios, sin embargo, los servicios web han tenido un aumento en su éxito, y probablemente mayor éxito que CORBA, por su utilización de tecnologías de moda como XML y Web. Conocer sus capacidades y limitaciones es un conocimiento necesario para desarrollar aplicaciones empresariales que se relacionen a través de Internet.

- Estudiar las ventajas y desventajas de JDO en comparación con EJB. *Java Data Objects (JDO)* es una API muy reciente para acceder a la base de datos a través de objetos en forma más liviana que EJB, lo que lo transformaría en más simple y más eficiente, sin embargo, no proporcionaría lógica interna ni objetos distribuidos. JDO y EJB pueden ser incluso complementarios para implementaciones de entity beans BMP o session beans que acceden a la base de datos. Conocer la relación existente entre ellos, los casos en que son alternativas distintas y los casos en que son complementarios significaría un aporte para un mejor diseño y desarrollo de proyectos J2EE.

Bibliografía

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns, Best Practices and Design Strategies*. Pretince Hall, 2001.

[2] S. Bodoff, D. Green, K. Haase, E. Jendrock, and B. Stearns. *The J2EE Tutorial*. Addison-Wesley, 2002.

[3] Marty Hall. *Core Servlets and JavaServer Pages*. Pretince Hall, 2001.

[4] Nicholas Kassem. *Designing Enterprise Applications with J2EE*. Addison-Wesley, 2000.

[5] Floyd Marinescu. *EJB Design Patterns*. John Wiley & Sons Inc., 2002.

[6] Ed Roman. *Mastering Enterprise JavaBeans*. John Wiley & Sons Inc., second edition, 2002.

[7] Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[8] Scott Stark. *JBoss Administration and Development*. Second edition, 2002.

[9] Sun Microsystems, Inc. *J2EE Connector Architecture Specification, Version 1.0*, Julio 1999.

[10] Sun Microsystems, Inc. *Java Naming and Directory Interface Application Programming Interface, Version 1.2*, Julio 1999.

[11] Sun Microsystems, Inc. *Java Transaction API (JTA), Version 1.0.1*, Agosto 1999.

[12] Sun Microsystems, Inc. *Java Transaction Service (JTS), Version 1.0*, Diciembre 1999.

[13] Sun Microsystems, Inc. *JDBC 2.1 API*, Octubre 1999.

[14] Sun Microsystems, Inc. *Enterprise JavaBeans Specification, Version 2.0*, Agosto 2001.

[15] Sun Microsystems, Inc. *Java 2 Platform, Enterprise Edition Specification, Version 1.3*, Agosto 2001.

[16] Sun Microsystems, Inc. *Java API for XML Processing, Version 1.1*, Febrero 2001.

[17] Sun Microsystems, Inc. *Java Servlet Specification, Version 2.3*, Agosto 2001.

[18] Sun Microsystems, Inc. *JavaServer Pages Specification, Version 1.2*, Agosto 2001.

[19] Dain Sundstrom. *JBossCMP*. Second edition, 2002.

[20] Anne Thomas. *J2EE, Ensuring Consistency, Portability, and Interoperability*. Patricia Seybold Group, 1999.

Apéndice A

Códigos fuentes

En este apéndice se presentarán códigos fuentes mostrando la implementación de los patrones de diseño creados para la capa de EJB, explicados la sección 7.4.2.

A.1. EjbDAO.java

Este es el código fuente de la clase DAO utilizada para el módulo de EJB, contiene la funcionalidad para la obtención de llaves primarias.

```
import java.sql.CallableStatement;
import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Types;
import java.util.Hashtable;
import java.util.StringTokenizer;
import java.util.Vector;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

public class EjbDAO {

  private static String DATA_SOURCE= "java:/OracleDS";
  private static String SEQUENCES_PROCEDURE= "{ ?= call getSecuencia(?,?) }";
  private static int GRUPO_SECUENCIAS= 5;
  private static int NUMERO_INTENTOS= 5;
  private static String STRING_SEPARADOR_SECUENCIAS= "-";

  private static EjbDAO dao= new EjbDAO();
```

```
public static EjbDAO getDAO() {
  return dao;
}
//Será el objeto que contendrá los valores para las llaves primarias
private Hashtable valoresSequences;;
private Context ctx;

public EjbDAO() {
  valoresSequences =new Hashtable();
  try {
    ctx= new InitialContext();
  } catch (NamingException e) {
  }
}
/**
 * Retorna un objeto Connection desde el pool de conexiones
 */
public Connection getConnection() {
  try {
    if(ctx==null)
      ctx= new InitialContext();
    javax.sql.DataSource ds= (javax.sql.DataSource) ctx.lookup(DATA_SOURCE);
    return ds.getConnection();
  } catch (Exception e) {
    System.out.println("EjbDAO.getConnection: " + e.toString());
    throw new RuntimeException("EjbDAO.getConnection: " + e.toString());
  }
}
/**
 * Retorna un valor correlativo con el próximo número que retornará la secuencia
 * con el nombre nombreSequence.
 * El String nombreSequence debe ser un campo static final
 * y será único, así que puedo hacer un synchronized sobre él.
 */
public Integer getSequenceNextVal(String nombreSequence)
  throws SQLException {
  //obtengo los numeros de secuencia
  //la obtencion de numero debe ser dentro de un synchronized
  synchronized (nombreSequence) {
    StringTokenizer st= null;
    Object o= valoresSequences.get(nombreSequence);
    if (o == null) {
      //si no existen los debo crear, hago un numero determinado de intentos
      for(int i=0; i<NUMERO_INTENTOS && st==null; ++i){
        st= cargarValores(nombreSequence);
      }
      //si no se pudo obtener resultados
      if (st == null)
        throw new RuntimeException("No se carga sequence " + nombreSequence);
      //guardo los resultados en el hash
      valoresSequences.put(nombreSequence, st);
    } else
      st= (StringTokenizer) o;
    //tomo el numero correspondiente
```

```
      int n= Integer.parseInt(st.nextToken());
      if (!st.hasMoreTokens())
        valoresSequences.remove(nombreSequence);
      return new Integer(n);
    }
  }
  /**
   * Dado un nombre de secuencia retorna el StringTokenizer que contiene los valores
   * retornados por la base de datos para la secuencia sequence.
  private StringTokenizer cargarValores(String sequence) {
    try {
      Connection con= getConnection();
      CallableStatement cstmt= con.prepareCall(SEQUENCES_PROCEDURE);
      cstmt.setString(2, sequence);
      cstmt.setInt(3, GRUPO_SECUENCIAS);
      cstmt.registerOutParameter(1, Types.VARCHAR);
      cstmt.execute();
      String values= cstmt.getString(1);
      con.close();
      return new StringTokenizer(values, STRING_SEPARADOR_SECUENCIAS);
    } catch (Exception e) {
        System.out.println("EjbDAO.cargarValores:" + e.toString());
        return null;
    }
  }
}
```

A.2. ActividadBean.java

Aquí se presenta el código fuente del entity bean Actividad donde se muestra una implementación del patrón para relaciones obligatorias y la utilización del patrón para generación de llaves primarias. Se pueden ver en los comentarios javadoc las instrucciones de XDoclet: el prefijo `@ejb:` señala las utilizadas para crear las interfaces relacionadas y construir el archivo *ejb-jar.xml*, y el prefijo `@jboss:` para configuraciones específicas al servidor JBoss creando los archivos *jboss.xml*, *jbosscmp-jdbc.xml* y *jaws.xml*.

```
import javax.ejb.EntityBean;

/**
 * @ejb:bean name="Actividad"
 *  jndi-name="jndi/ActividadBean"
 *  type="CMP"
 * cmp-version="2.x"
 * view-type="local"
 *
 * @ejb:transaction type="Required"
 *
 * @jboss:table-name ACTIVIDADES
```

```
 * @jboss:create-table false
 * @jboss:remove-table false

 @ejb:finder signature="java.util.Collection
                    findByPersonaProyecto(cl.entity.Persona per, cl.entity.Proyecto pro)"
 query="SELECT OBJECT(a)
          FROM Actividad a, Persona p, Proyecto pro
         WHERE a.personaId= p.rut AND p=?1
           AND a.etapaProyecto.proyectoId=pro.codigo AND pro= ?2"
 *
 * @ejb:finder
 *  signature="java.util.Collection findByPersona(cl.entity.Persona per)"
 *  query="SELECT OBJECT(a) FROM Actividad a, Persona p WHERE a.personaId= p.rut AND p=?1"
 *
 * @ejb:finder
 *  signature="java.util.Collection findAll()"
 *  query="Select Object(a) from Actividad a"
 *
 * @ejb:select
 *  signature="java.util.Collection ejbSelectPersona(java.lang.Integer personaId)"
 *  query="SELECT OBJECT(a) FROM Persona a WHERE a.rut= ?1"
 *
 **/
public abstract class ActividadBean implements EntityBean {

  //nombre de la secuencia que otorga el Id en la base de datos para este bean
  private static final String SEQUENCE_NAME="ACT_SEQ";

  //GETTERS
  /**
   * @ejb:interface-method
   * @ejb:persistent-field
   * @ejb:pk-field
   * @jboss:column-name ACT_ID
   **/
  public abstract Integer getId();
  /**
   * @ejb:interface-method
   * @ejb:persistent-field
   * @jboss:column-name ACT_APROBADO
   **/
  public abstract String getAprobado();
  /**
   * @ejb:interface-method
   * @ejb:persistent-field
   * @jboss:column-name ACT_DESCUENTO
   **/
  public abstract Double getDescuento();
  /**
  * @ejb:interface-method
  * @ejb:persistent-field
  * @jboss:column-name ACT_FECHA
  **/
  public abstract java.util.Date getFecha();
```

```
/**
 * @ejb:interface-method
 * @ejb:persistent-field
 * @jboss:column-name ACT_HORA_FIN
 **/
public abstract java.util.Date getHoraFin();
/**
 * @ejb:interface-method
 * @ejb:persistent-field
 * @jboss:column-name ACT_HORA_INI
 **/
public abstract java.util.Date getHoraInicio();
/**
 * @ejb:interface-method
 * @ejb:persistent-field
 * @jboss:column-name ACT_OBSERVACION
 **/
public abstract String getObservacion();
/**
 * @ejb:interface-method
 * @ejb:persistent-field
 * @jboss:column-name ACT_TIPO
 **/
public abstract String getTipo();
/**
 * @ejb:interface-method
 * @ejb:persistent-field
 * @jboss:column-name ACT_TOTAL
 **/
public abstract Double getTotal();

//SETTERS
/**
 * @ejb:interface-method
 */
  public abstract void setId(Integer id);
/**
 * @ejb:interface-method
 */
  public abstract void setAprobado(String aprobado);
/**
 * @ejb:interface-method
 */
  public abstract void setDescuento(Double descuento);
/**
 * @ejb:interface-method
 */
  public abstract void setFecha(java.util.Date fecha);
/**
 * @ejb:interface-method
 */
  public abstract void setHoraFin(java.util.Date horaFin);
/**
 * @ejb:interface-method
```

```
 */
  public abstract void setHoraInicio(java.util.Date horaInicio);
/**
 * @ejb:interface-method
 */
  public abstract void setObservacion(String observacion);
/**
 * @ejb:interface-method
 */
  public abstract void setTipo(String tipo);
/**
 * @ejb:interface-method
 */
  public abstract void setTotal(Double total);

//RELACIONES OPCIONALES
/**
 * @ejb:interface-method
 * @ejb:relation name="Actividad-EtapaEsp-REF" role-name="activ-tipoesp"
 *   target-ejb="TipoEspecial" target-role-name="activ-tipoesp-target"
 *   target-multiple="yes"
 * @jboss:relation related-pk-field="id" fk-column="TES_ID" fk-constraint="false"
 **/
public abstract TipoEspecial getTipoEspecial();

/**
 * @ejb:interface-method
 * @ejb:relation name="Actividad-EtapaProy-REF" role-name="activ-etapaproy"
 *   target-ejb="EtapaProyecto" target-role-name="target-activ-etapaproy"
 *   target-multiple="yes"
 * @jboss:relation related-pk-field="id" fk-column="EPR_ID" fk-constraint="false"
 * @jboss:column-name EPR_ID
 **/
public abstract EtapaProyecto getEtapaProyecto();
/**
 * @ejb:interface-method
 * @ejb:relation name="Actividad-Cliente-REF" role-name="activ-cliente"
 *   target-ejb="Cliente" target-role-name="target-activ-cliente"
 *   target-multiple="yes"
 * @jboss:relation related-pk-field="id" fk-column="CLI_ID" fk-constraint="false"
 * @jboss:column-name CLI_ID
 **/
public abstract Cliente getCliente();
/**
 * @ejb:interface-method
 */
public abstract void setTipoEspecial(TipoEspecial tipoEspecial);
/**
 * @ejb:interface-method
 */
public abstract void setEtapaProyecto(EtapaProyecto etapaProyecto);
/**
 * @ejb:interface-method
 */
```

```
public abstract void setCliente(Cliente cliente);


//RELACION OBLIGATORIA CON ENTITY BEAN PERSONA
/**
 * @ejb:interface-method
 **/
//getter originalmente de la relación
public Persona getPersona(){
  try {
    return (Persona) ejbSelectPersona(getPersonaId()).iterator().next();
  } catch (javax.ejb.FinderException e) {
    throw new RuntimeException(e.toString());
  }
}
/**
* @ejb:interface-method
*/
//setter de la relación
public void setPersona(Persona persona){
  setPersonaId(((PersonaPK) persona.getPrimaryKey()).getRut());
}

/**
 * @ejb:persistent-field
 * @jboss:column-name PER_RUT
 **/
//getter y setter del campo que no serán escritos en las interfaces para cliente
public abstract Integer getPersonaId();
public abstract void setPersonaId(Integer personaId);

//metodo de búsqueda cuyo EJB QL se encuentra al inicio de la clase
public abstract java.util.Collection ejbSelectPersona(Integer personaId)
  throws javax.ejb.FinderException;

//CREATE
/**
* @ejb:create-method
*/
  public ActividadPK ejbCreate(Date horaInicio, Date horaFin, String observacion,
        Double descuento, Double total, String tipo, String aprobado, Date fecha,
        Persona persona, EtapaProyecto etapaProyecto, TipoEspecial tipoEspecial,
        Cliente cliente)  throws javax.ejb.CreateException {
    try {
      setHoraInicio(horaInicio);
      setHoraFin(horaFin);
      setObservacion(observacion);
      setDescuento(descuento);
      setTotal(total);
      setTipo(tipo);
      setAprobado(aprobado);
      setFecha(fecha);
      setPersona(persona);  //se asigna relación obligatoria
      //generar llave primaria
```

```
        Integer n= EjbDAO.getDAO().getSequenceNextVal(SEQUENCE_NAME);
        setId(n);
        return new ActividadPK(n);
      } catch (Exception e) {
        throw new javax.ejb.CreateException(+e.toString());
      }
    }

    public void ejbPostCreate(Date horaInicio, Date horaFin, String observacion,
          Double descuento, Double total, String tipo, String aprobado, Date fecha,
          Persona persona, EtapaProyecto etapaProyecto, TipoEspecial tipoEspecial,
          Cliente cliente){
      //se asignan relaciones opcionales modeladas por CMP
      setEtapaProyecto(etapaProyecto);
      setTipoEspecial(tipoEspecial);
      setCliente(cliente);
    }
}
```

A.3. ActionFormEntity.java

En este apéndice se lista el código fuente del archivo *ActionFormEntity.java* el cual corresponde a la implementación del patrón de diseño para edición de entity beans. Este código utiliza reflexión para copiar los datos desde el entity bean hacia la clase que heredará de ActionFormEntity. Para crear la clase que herede, se debe crear los getter y setter de los campos a mostrar en la web (con el mismo nombre que contiene el entity bean) más un método que retorne el tipo del campo (si es fecha, número, decimal, string o referencia a un objeto) como una constante de la clase *TipoCampo*.

```
import java.lang.reflect.*;
import java.util.Date;
import javax.ejb.*;
import javax.servlet.http.*;
import org.apache.struts.action.*;

public abstract class ActionFormEntity extends ActionForm {

    protected Object entityBean;      //guarda el entity
    protected RefManager refManager;  //mantiene referencias

    private Integer entityRef;        //referencia al entity neam
    private String action;            //para los forms

    /**
     * Constructor a ser usado por struts (despues ejecuta reset)
     */
    public ActionFormEntity() {
    }
```

```
  /**
   * Constructor a ser usado por un programa (que no ejecute reset)
   */
  public ActionFormEntity(HttpServletRequest request) {
      this.refManager= new RefManager(request.getSession());
  }

  //getters (estos campos se mantendrán entre forms)
  public final Integer getEntityRef() {
      return entityRef;
  }

  public final String getAction() {
      return action;
  }

  //setters
  public final void setEntityRef(Integer entityRef) {
      this.entityRef= entityRef;
  }

  public final void setAction(String action) {
      this.action= action;
  }

  /**
   * Esto retorna la PK del entity bean asociado,
   * con este metodo luego se puede hacer el
   * setEntityBean para hacer un update del entity bean
   */
  public Object obtenerEntityBean() {
      try {
          if (entityRef == null)
              return null;
          return refManager.get(this.getClass().getName(), entityRef);
      } catch (Exception e) {
          System.out.println("getEntityPK: " + e.toString());
          return null;
      }
  }
  /**
   * debe retornar un String[] con el nombre de los campos que hacen de
   * primary key. Es necesario sobreescribiro para cada subclase.
   * No le puse getNombreCamposPK para no confundir al mapear.
   */
  public abstract String[] nombreCamposPK();

//para asociar un entity a este ActionForm
  public void asociarEntityBean(EJBLocalObject entityBean) {
      this.entityBean= entityBean;
      crearCampoEntity(entityBean);
  }
//para asociar un entity a este ActionForm
  public void asociarEntityBean(EJBObject entityBean) {
```

```
        this.entityBean= entityBean;
        crearCampoEntity(entityBean);
    }
  //crea y guarda la referencia al entity
    private void crearCampoEntity(Object object) {
        this.entityRef= refManager.add(this.getClass().getName(), object);
    }

    /**
     * Copio todos los campos que se cumpla la condicion:
     *  (obj) entity.getXXX()
     *  (void) this.setXXX(String)
     * ademas debe existir un metodo:
     *  (int) this.tipoXXX()
     * Se debe haber seteado el entityBean previamente.
     */
    public void mapearEntityAForm() {
        Class[] sinArgumentos= new Class[0];
        Object[] sinParametros= new Object[0];
        Method[] metodosForm= this.getClass().getMethods();
        for (int k= 0; k < metodosForm.length; ++k) {
            String nombreSetter= metodosForm[k].getName();
            Method setter= metodosForm[k];
            if (nombreSetter.startsWith("set")
                && setter.getParameterTypes().length == 1) {
                String nombreCampo= nombreCampo(nombreSetter);
                try {
                    //obtengo el getter
                    String nombreGetter= nombreGetter(nombreCampo);
                    Method getter=
                        this.entityBean.getClass().getMethod(nombreGetter, sinArgumentos);
                    //obtengo el valor del get
                    Object valorObject= getter.invoke(this.entityBean, sinParametros);
                    //obtengo el tipo en el form
                    String nombreTipo= nombreTipo(nombreCampo);
                    Method tipo= this.getClass().getMethod(nombreTipo, sinArgumentos);
                    int tipoCampo=
                        ((Integer) tipo.invoke(this, sinParametros)).intValue();
                    //lo transformo a string (los forms son solo de strings)
                    Object[] val= new Object[1];
                    val[0]= TipoCampo.formateador.ObjectToString(valorObject, tipoCampo);
                    //invoco en el setter
                    setter.invoke(this, val);
                } catch (Exception e) {
                }
            }
        }
    }
    /**
     * Sobreescrito de ActionForm.
     * Es similar a mapearFormAEntity lo unico es que no ejecuta el setter del entity,
     * las excepciones de formato las guarda al ActionErrors,
     * y no utiliza el entityBean (que en el instante en que se
     * ejecuta no ha sido asociado)
```

```
 * Si hay un problema tira un error del tipo
 * error.formato asociado al campo
 * No se ejecuta si en el request hay un atributo nombre notValidate
 * con valor notValidate
 */
public ActionErrors validate(
    ActionMapping mapping,
    HttpServletRequest request) {
    ActionErrors errors= new cl.mistruts.MiActionErrors();
    if ("notValidate".equals(request.getAttribute("notValidate")))
        return errors;
    if ("notValidate".equals(request.getParameter("notValidate")))
        return errors;
    Class[] sinArgumentos= new Class[0];
    Object[] sinParametros= new Object[0];
    Method[] metodosForm= this.getClass().getMethods();
    for (int k= 0; k < metodosForm.length; ++k) {
        String nombreGetter= metodosForm[k].getName();
        Method getter= metodosForm[k];
        if (nombreGetter.startsWith("get")
            && getter.getParameterTypes().length == 0) {
            String nombreCampo= nombreCampo(nombreGetter);
            try {
                //obtengo el valor del get (es un string)
                Object obj= getter.invoke(this, sinParametros);
                //obtengo el tipo en el form
                String nombreTipo= nombreTipo(nombreCampo);
                Method tipo= this.getClass().getMethod(nombreTipo, sinArgumentos);
                int tipoCampo=
                    ((Integer) tipo.invoke(this, sinParametros)).intValue();
                //lo valido con el metodo de TipoCampo
                Object valorObject=
                    TipoCampo.formateador.StringToObject(obj, tipoCampo);
            } catch (FormatoException e) {
                errors.add(
                    nombreCampo,
                    new ActionError(e.getErrorCampo(), e.getParametro()));
            } catch (Exception e) {
            }
        }
    }
    return errors;
}
/**
 * Retorna el nombre del campo con el formato especificado
 * en el metodo tipoXX
 */
public Object getCampoObject(String nombreCampo) throws Exception {
    Class[] sinArgumentos= new Class[0];
    Object[] sinParametros= new Object[0];
    //obtengo el getter
    String nombreGetter= nombreGetter(nombreCampo);
    Method getter= this.getClass().getMethod(nombreGetter, sinArgumentos);
    //obtengo el tipo en el form
```

```
    String nombreTipo= nombreTipo(nombreCampo);
    Method tipo= this.getClass().getMethod(nombreTipo, sinArgumentos);
    int tipoCampo= ((Integer) tipo.invoke(this, sinParametros)).intValue();
    //obtengo el valor del get
    Object obj= getter.invoke(this, sinParametros);
    //si es objeto lo retorno
    if (tipoCampo == TipoCampo.OBJETO)
        return obj;
    //lo transformo a objeto si es string
    String valor= obj == null ? null : obj.toString();
    Object valorObject= TipoCampo.formateador.StringToObject(valor, tipoCampo);
    return valorObject;
}
/**
 * Retorna el nombre del campo con el formato especificado
 * en el metodo tipoXX
 */
public Integer getCampoInteger(String nombreCampo) throws Exception {
    Object o= getCampoObject(nombreCampo);
    if (o == null)
        return null;
    return (Integer) o;
}
/**
 * Retorna el nombre del campo con el formato especificado
 * en el metodo tipoXX
 */
public java.util.Date getCampoDate(String nombreCampo) throws Exception {
    Object o= getCampoObject(nombreCampo);
    if (o == null)
        return null;
    return (java.util.Date) o;
}
/**
 * Retorna el nombre del campo con el formato especificado
 * en el metodo tipoXX
 */
public Double getCampoDouble(String nombreCampo) throws Exception {
    Object o= getCampoObject(nombreCampo);
    if (o == null)
        return null;
    return (Double) o;
}
/**
 * Retorna el nombre del campo con el formato especificado
 * en el metodo tipoXX
 */
public String getCampoString(String nombreCampo) throws Exception {
    Object o= getCampoObject(nombreCampo);
    if (o == null)
        return null;
    return (String) o;
}
/**
```

```
     * hace un getXX y retorna el string sin formatearlo
     * tal como esta en el form
     */
    public String getCampo(String nombreCampo) {
        try {
            Class[] sinArgumentos= new Class[0];
            Object[] sinParametros= new Object[0];
            //obtengo el getter
            String nombreGetter= nombreGetter(nombreCampo);
            Method getter= this.getClass().getMethod(nombreGetter, sinArgumentos);
            //obtengo el valor del get
            Object obj= getter.invoke(this, sinParametros);
            String valor= obj == null ? null : obj.toString();
            return valor;
        } catch (Exception e) {
            return null;
        }
    }

    private String nombreGetter(String campo) {
        return "get" + campo.substring(0, 1).toUpperCase() + campo.substring(1);
    }
    private String nombreSetter(String campo) {
        return "set" + campo.substring(0, 1).toUpperCase() + campo.substring(1);
    }
    private String nombreTipo(String campo) {
        return "tipo" + campo.substring(0, 1).toUpperCase() + campo.substring(1);
    }
    private String nombreCampo(String nombreGetSet) {
        return nombreGetSet
            .substring("get".length(), "get".length() + 1)
            .toLowerCase()
            + nombreGetSet.substring("get".length() + 1);
    }
    /**
     * Sobreescrito de ActionForm.
     * En el reset tomo la session para poder hacer referencias a los objetos
     */
    public void reset(ActionMapping mapping, HttpServletRequest request) {
        this.refManager= new RefManager(request.getSession());
    }
}
```

Juan Manuel Barrios Núñez
http://www.dcc.uchile.cl/~jbarrios/
jbarrios@dcc.uchile.cl
Santiago, Chile.