Magister en Ingeniería de Software

Universidad Nacional de La Plata

Facultad de Informática

Tesis

Migración hacia un modelo de persistencia orientado a objetos

Desarrollada por Nelson O. Di Grazia
Dirigida por Dr. Gustavo Rossi
Asesorada por Mg. Javier Bazzocco

Índice General

Capítulo 1: *Introducción y Objetivo*

1.1 Introducción 1
1.2 Objetivo 2
1.3 Contribuciones 3
1.4 Alcance 4
1.5 Organización del texto 4

Capítulo 2: *Reseña de la tecnología y de la empresa de telecomunicaciones*

2.1 Reseña de la tecnología J2EE 6
2.1.1 Capas de la tecnología J2EE 7
2.1.2 Componentes de la arquitectura J2EE 7
2.1.3 Arquitectura J2EE 8
2.2 Reseña de la tecnología Enterprise JavaBeans 9
2.2.1 El bean de sesión 10
2.2.2 El bean de entidad 10
2.2.3 El bean de mensajería 11
2.2.4 Ambiente Enterprise JavaBeans 11
2.3 Reseña de la tecnología JDO 12
2.3.1 ¿Qué es JDO? 12
2.3.2 Persistencia de objetos con JDO 13
2.3.3 Arquitectura de JDO 14
2.3.3.1 Instancia JDO 15
2.3.3.2 Ambientes en JDO 16
2.3.3.2.1 Ambiente no administrado 16
2.3.3.2.2 Ambiente administrado 16
2.3.3.3 El lenguaje JDOQL 17
2.3.4 Instancias JDO persistentes y transitorias 17
2.4 Reseña de la arquitectura de la empresa 18
2.5 Conclusión 19

Capítulo 3: *JDO en ambientes administrados*

3.1 Ambiente administrado 20
3.2 Administración de las transacciones 22
3.3 Uso de JDO en la arquitectura Enterprise JavaBeans 23
3.4 Integrando JDO con EJB 24

3.4.1 Integrando JDO con beans de sesión 24
3.4.1.1 Integrando JDO con beans de sesión sin estado 25
3.4.1.2 Integrando JDO con beans de sesión con estado 27
3.4.2 Integrando JDO con beans de entidad 29
3.4.3 Integrando JDO con beans de mensajería 32
3.5 Conclusión 33

Capítulo 4*: *Recopilación de la arquitectura de software de la empresa

4.1 Capas lógicas 35
4.2 Capas lógicas y los componentes de software 36
4.3 Mecanismos de persistencia 37
4.4 Objetos de negocio y su representación en la empresa 39
4.5 Diagrama de clases de objetos de negocio 41
4.6 Componentes de la arquitectura de software 42
4.7 Solicitudes de la capa del cliente 47
4.8 Interacción entre los componentes 50
4.9 Conclusión 50

Capítulo 5*: *Ejemplo de la arquitectura de software de la empresa

5.1 Dominio de ejemplo 53
5.2 Diseño 54
5.2.1 Diseño de objetos de negocio 54
5.2.2 Diseño de datos 56
5.3 Implementación 61
5.3.1 Implementación de objetos de negocio 61
5.3.1.1 Archivos xml de definición de objetos de negocio 61
5.3.2 Generación de código 65
5.3.3 Implementación de los servicios 66
5.3.3.1 Xml de servicios 66
5.3.3.2 Programación de los servicios 68
5.3.3.2.1 Clase de la lógica del negocio 68
5.3.3.2.1.1 Consulta de contratos 69
5.3.3.2.1.2 Inserción de contratos 70
5.4 Conclusión 73

Capítulo 6*: *Nueva arquitectura de software

6.1 Capas lógicas 76

6.2 Mecanismo de persistencia de las instancias 77
6.3 Objetos de negocio y su representación 78
6.4 Jerarquía de objetos de negocio 79
6.5 Componentes de la arquitectura de software 79
6.6 Solicitudes de la capa del cliente 81
6.7 Interacción entre los componentes 82
6.8 Conclusión 84

Capítulo 7*: *Ejemplo de la nueva arquitectura de software

7.1 Dominio de ejemplo 86
7.2 Diseño 87
7.2.1 Diseño de objetos de negocio 87
7.2.2 Diseño de datos 88
7.3 Implementación 88
7.3.1 Implementación de objetos de negocio 88
7.3.2 Generación de código 90
7.3.3 Implementación de los servicios 90
7.3.3.1 Programación de los servicios 91
7.3.3.1.1 Clase de la lógica del negocio 91
7.3.3.1.1.1 Consulta de contratos 91
7.3.3.1.1.2 Inserción de contratos 92
7.4 Conclusión 94

Capítulo 8*: *Migración entre las arquitecturas de software

8.1 Objetos de negocio 95
8.2 Acceso a los datos 97
8.3 Implementación de los servicios 97
8.4 Herramienta automática 98
8.5 Conclusión 98

Capítulo 9: *Conclusión y Trabajo futuro*

9.1 Conclusión 99
9.2 Trabajo futuro 102

Apéndice A*: *Mapeo relacional

A.1 Mapeo de clases 104
A.2 Mapeo de relaciones 107
A.2.1 Relaciones 1 a 1 107
A.2.1.1 Relaciones 1 a 1 (Unidireccional) 107

A.2.1.2 Relaciones 1 a 1 (Bidireccional) 109
A.2.2 Relaciones 1 a N 110
A.2.2.1 Relaciones 1 a N (Bidireccional) 111
A.2.3 Relaciones M a N 112
A.2.4 Relaciones de herencia 115

***Apéndice B**: Estructura del Descriptor de Persistencia*

B.1 Contenido del Descriptor de Persistencia 118
B.2 Definición de los tags 123
B.2.1 Elemento jdo 123
B.2.2 Elemento package 123
B.2.3 Elemento class 123
B.2.4 Elemento field 124
B.2.5 Elemento column 124
B.2.6 Elemento collection 124
B.2.7 Elemento extension 124
B.3 Ejemplo del archivo xml 124

Referencias Bibliográficas 126

CAPÍTULO 1

Introducción y Objetivo

1.1 Introducción

Actualmente, muchas empresas eligen la tecnología Java 2 Enterprise Edition (J2EE) para la construcción de sus productos de software. Esta tecnología, utiliza un modelo de desarrollo basado en componentes que permite implementar sistemas distribuidos usando el lenguaje de programación Java.

El modelo de componentes denominado Enterprise Java Bean (EJB)[1], agrega un soporte poderoso para la separación de la lógica de negocio de las restantes áreas de interés del desarrollo de software. Aunque esta división, o modularización, son fundamentos de la ingeniería del software, en los últimos años la construcción de aplicaciones usando este enfoque ha sufrido restricciones con respecto a los principios fundamentales de la orientación a objetos y a los principios del diseño.

Todo desarrollo basado en el modelo de componentes EJB obliga a que tanto las responsabilidades de la lógica de negocio como las responsabilidades de su administración, residan en un mismo elemento. Este conjunto de responsabilidades poco relacionadas genera una baja cohesión y un alto acoplamiento. Además, la utilización de este modelo para el manejo de la persistencia produce una diferencia importante entre nuestro modelo de diseño de objetos y su representación en el modelo de componentes EJB, lo que provoca que no se pueda aplicar algunos principios propios del paradigma orientado a objetos, como por ejemplo el *concepto de la herencia*.

[1] Enterprise Java Bean (EJB) es un modelo de componentes distribuidos del lado del servidor, cuyo objetivo es dotar al programador de un modelo que le permita abstraerse de los problemas generales de una aplicación empresarial.

De esta manera, los problemas antes descriptos plantean una serie de impedimentos a la hora de emplear el enfoque de componentes EJB para el almacenamiento de los datos en las aplicaciones empresariales.

Dentro de las alternativas para atacar los inconvenientes del modelo de componentes EJB, surge el *Framework Java Data Object (JDO)*[2] como un estándar de persistencia. El mismo permite que las responsabilidades de la administración de las instancias no residan en la propia instancia, y garantiza la aplicación de los principios fundamentales del paradigma orientado a objetos, tan relegado en el modelo de componentes EJB.

Puede decirse que el Framework JDO impacta en la forma de desarrollo de las aplicaciones, reduciendo el salto entre nuestro modelo de objetos de negocio a ser persistido y su representación en la tecnología subyacente. Sin embargo, la utilización de este modelo genera inquietudes en aquellas empresas que utilizan la arquitectura basada en componentes para la persistencia y que, a través de un cambio tecnológico, desean obtener todas las ventajas del enfoque orientado a objetos en el almacenamiento de sus datos.

Por último, la elección del Framework JDO no es excluyente de otras tecnologías similares que permitan la aplicación del paradigma orientado a objetos sobre la capa de persistencia.

1.2 Objetivo

El propósito de esta tesis es analizar el impacto generado al reemplazar el modelo de componentes EJB usado en la capa de persistencia por el modelo de desarrollo orientado a objetos. Para esto,

[2] Java Data Object (JDO) es una especificación abstracta de persistencia, cuya característica más destacable es la transparencia en la persistencia de los objetos de dominio.

se utilizará la arquitectura basada en componentes y los procesos de desarrollo empleados actualmente en la empresa de telecomunicaciones en donde desempeño mis actividades.

Por último, se espera obtener distintas experiencias del proceso aplicado, que puedan ser usadas a la hora de migrar una plataforma de manejo de la persistencia basada en componentes a otra orientada a objetos, independientemente de la implementación asociada a esta última.

1.3 Contribuciones

Las principales contribuciones que surgen de alcanzar el objetivo, se detallan a continuación:

- Enunciar un conjunto de *"lecciones aprendidas"* que puedan ser aplicadas para transformar la capa de manejo de la persistencia basada en componentes a otra orientada a objetos, y que respondan a preguntas tales como: ¿Que capas lógicas se ven afectadas?, ¿Que se migra primero?, ¿Quién tendrá la responsabilidad de la persistencia?.

- Proponer un método de migración, que permita una trasformación exitosa de los componentes en sus correspondientes representaciones en el paradigma orientado a objetos.

- Plantear una arquitectura de referencia para aquellas organizaciones que quieran construir aplicaciones usando el paradigma orientado a objetos en la capa de persistencia.

1.4 Alcance

El presente trabajo tiene como objetivo poner en práctica los conocimientos necesarios para vincular el esquema de desarrollo de componentes EJB empleado por la empresa de telecomunicaciones con la interfaz JDO, sin profundizar en los conceptos de las tecnologías en cuestión, ni cubrir los aspectos relevantes de cada implementación de la interfaz misma.

1.5 Organización del texto

Esta tesis se estructura en nueve capítulos y dos apéndices. A continuación se detalla brevemente el contenido del trabajo.

Capítulo 1: Breve introducción global y enunciado del objetivo.

Capítulo 2: Presenta una reseña de la arquitectura de desarrollo de software de la empresa de telecomunicaciones y de cada una de las tecnologías usadas en este trabajo.

Capítulo 3: Explica como los componentes del modelo Enterprise JavaBeans utilizan la tecnología JDO para llevar a cabo sus acciones.

Capítulo 4: Presenta una visión general de la arquitectura de software, utilizada por la empresa de telecomunicaciones en el desarrollo de sus productos de software.

Capítulo 5: Muestra un ejemplo de diseño y de implementación utilizando la arquitectura de desarrollo de la empresa de telecomunicaciones para la construcción de una aplicación de software.

Capítulo 6: Presenta la nueva arquitectura de software que, empleando el Framework JDO, permitirá a la empresa de telecomunicaciones desarrollar sus productos de software.

Capítulo 7: Muestra un ejemplo de diseño y de implementación empleando la nueva arquitectura de desarrollo para la creación de una aplicación de software.

Capítulo 8: Explica las actividades a realizar para la migración de la arquitectura actual de la empresa de telecomunicaciones hacia la nueva arquitectura que emplea JDO, como mecanismo de persistencia de los objetos del dominio.

Capítulo 9: Presenta las conclusiones generales de esta tesis y sugiere el eje para el futuro trabajo.

Apéndice A: Describe como la tecnología JDO se emplea en ambientes de base de datos relacionales y como se usan los mecanismos de mapeo para persistir los objetos del dominio, en el esquema relacional.

Apéndice B: Detalla la estructura del archivo xml utilizado por el modelo JDO, durante el proceso de modificación o de enriquecimiento de las clases.

CAPÍTULO 2

Reseña de la tecnología y de la empresa de telecomunicaciones

Este capítulo presenta una reseña de las tecnologías que se utilizarán en este trabajo, no se pretende tratarlas en profundidad sino brindar una introducción al lector de cada una de ellas para que adquiera el conocimiento necesario para comprender los temas tratados en esta tesis.

Finalmente, se describe la arquitectura de la empresa de telecomunicaciones que será tomada como medio de estudio del impacto del Framework JDO sobre sus procesos de desarrollo.

2.1 Reseña de la tecnología J2EE

La tecnología Java 2 Enterprise Edition (J2EE) es una arquitectura que permite implementar sistemas distribuidos usando el lenguaje de programación Java. La variedad de productos construidos con esta tecnología abarca desde pequeñas aplicaciones en redes empresariales hasta grandes aplicaciones para negocios específicos [18].

Esta arquitectura se centró en la definición de estándares y de contratos, para el desarrollo sencillo de aplicaciones orientadas a objetos y distribuidas, bajo el lenguaje de programación Java [18].

El objetivo principal de la tecnología J2EE es simplificar el desarrollo de las aplicaciones, a través de un modelo de componentes de software distribuidos. En este esquema, los componentes de

software contienen toda la lógica de negocio que, caso contrario, estaría diseminada en la toda la aplicación.

2.1.1 Capas de la tecnología J2EE

La tecnología J2EE separa el desarrollo del software en cuatro capas lógicas [18]: la *capa del cliente*, la *capa web*, la *capa lógica* y finalmente, la *capa de datos*.

La *capa del cliente* se encarga de mostrar la información y manejar la interacción con el usuario. La arquitectura J2EE soporta varias aplicaciones clientes.

La *capa web* maneja la interacción con los clientes web y genera las respuestas apropiadas a los usuarios.

La *capa lógica* es el elemento clave de la tecnología J2EE. Esta capa esta compuesta por componentes de software que contienen la lógica de negocio de la aplicación.

La *capa de datos*, formada por todos los recursos que almacenarán los datos de la aplicación.

2.1.2 Componentes de la arquitectura J2EE

El modelo de componentes de software distribuidos de la arquitectura J2EE esta formado por tres elementos [18]:

- *Servlet*. Componente que reside en la capa web que interactúa con los clientes mediante el esquema pedido/respuesta.

- *JavaServer Pages (JSP)*. Componente residente en la capa web que genera respuestas, con contenido dinámico, a los clientes.

- *Enterprise JavaBeans (EJB)*. Componente que reside en la capa lógica y contiene toda la lógica de negocio de la aplicación.

El último componente será profundizado más adelante considerando que, constituye un elemento clave para alcanzar los objetivos de este trabajo.

2.1.3 Arquitectura J2EE

En la arquitectura J2EE, todos los componentes viven en un *Servidor de Aplicaciones*. Un *Servidor de Aplicaciones* es un ambiente de ejecución el cuál, provee servicios de seguridad, de manejo de transacciones y de persistencia a los componentes agrupados en los contenedores residentes en él. También, puede ofrecer funcionalidades compatibles con la especificación J2EE que serán provistas por cada vendedor [18].

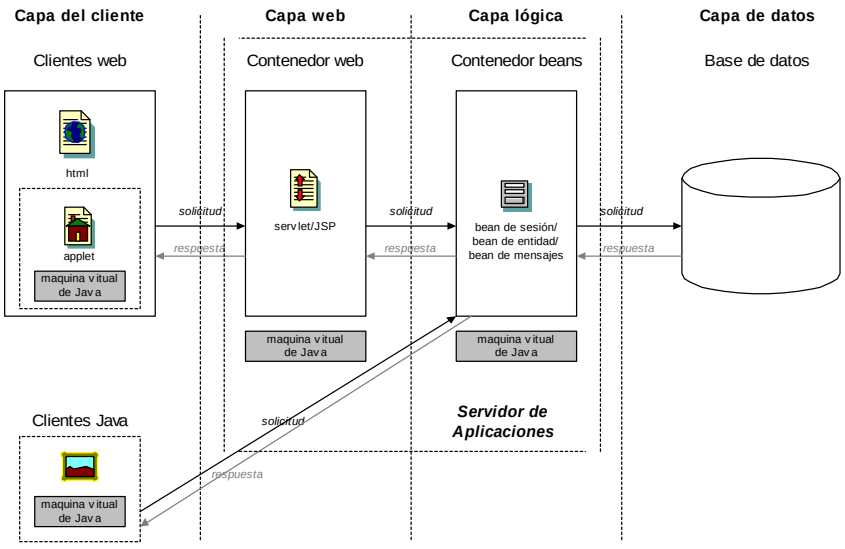

Figura 1 – *Arquitectura J2EE*

En un ambiente típico J2EE, los clientes realizan invocaciones a los servicios ofrecidos por los componentes de software que residen en el *Servidor de Aplicaciones*. Estos componentes, por medio de solicitudes intercambiadas entre las distintas capas de la tecnología J2EE retornan las respuestas a los pedidos realizados.

2.2 Reseña de la tecnología Enterprise JavaBeans

La tecnología *Enterprise JavaBeans (EJB)* es un modelo de componentes de software, comunmente conocidos como componentes EJB, para el desarrollo de sistemas distribuidos. Este Framework constituye la clave de la arquitectura J2EE [17].

Como mencionamos, estos componentes al igual que los otros componentes de la arquitectura J2EE, necesitan vivir en un *Servidor de Aplicaciones* para poder operar adecuadamente.

Este modelo define tres tipos de componentes [17]: el *bean de sesión*, el *bean de entidad* y el *bean de mensajería*.

Los *beans de sesión* son los componentes de sesión de la arquitectura Enterprise JavaBeans y representan un punto de entrada a la aplicación que está corriendo en el servidor. Estos agrupan toda la lógica del negocio relacionada con la solicitud realizada.

Los *beans de entidad* son los componentes de entidad de la arquitectura Enterprise JavaBeans, representan la vista de una instancia persistida en un medio de almacenamiento y sólo contienen lógica propia de la entidad.

Los *beans de mensajería* son los componentes de administración de mensajes de la arquitectura Enterprise JavaBeans.

2.2.1 El bean de sesión

Los beans de sesión pueden operar con los clientes de dos maneras distintas [17].

En la primera forma, los beans de sesión no mantienen un estado conversacional con el cliente. En esta situación el bean de sesión opera sin recordar el estado de las transacciones realizadas por el cliente. Este comportamiento recibe el nombre de *Stateless* o sin estado [17].

En el segundo caso, el bean de sesión es asignado sólo a un cliente y mantiene un estado conversacional con éste, entre cada llamada a sus métodos. En este escenario el bean de sesión opera recordando el estado de las transacciones realizadas por el cliente. Este comportamiento recibe el nombre de *Stateful* o con estado [17].

Por último, es importante conocer que los beans de sesión pueden administrar las transacciones vía el *Servidor de Aplicaciones* o en forma manual en el propio componente.

2.2.2 El bean de entidad

Los beans de entidad tienen dos mecanismos de persistencia. El primero, permite que el *Servidor de Aplicaciones* sea el responsable del manejo de la persistencia. El segundo, delega en el componente la lógica de persistirse [17].

Generalmente, cuando el bean maneja manualmente la persistencia, se emplean instrucciones de base de datos dentro del componente como mecanismo de almacenamiento.

Finalmente, este componente opera sobre contextos transaccionales administrados por el *Servidor de Aplicaciones*.

2.2.3 El bean de mensajería

El sistema de mensajería de la arquitectura J2EE, permite la comunicación asincrónica entre las distintas aplicaciones que conforman el sistema empresarial [17].

Este sistema permite el envío de mensajes entre cada elemento de la aplicación, manteniendo un anonimato entre los emisores de mensajes y los consumidores de mensajes.

La especificación J2EE utiliza el bean de mensajería para la administración de los mensajes enviados. Este componente actúa como un consumidor asincrónico de mensajes.

Los clientes que desean enviar un mensaje, no invocan directamente los servicios de este bean, sino que lo envían a través de un Framework diseñado para tal función y es el *Servidor de Aplicaciones*, el responsable de que el bean de mensajería lo procese [17].

2.2.4 Ambiente Enterprise JavaBeans

Una aplicación EJB consta de varios clientes accediendo a los componentes beans de sesión y de entidad. Si bien el cliente puede acceder a ambos componentes, la mayoría de las aplicaciones empresariales usan al bean de sesión como medio de acceso a la aplicación que esta corriendo en el servidor [17].

Cada bean de sesión resuelve la solicitud por medio propio o enviando pedidos al bean de entidad o al bean de mensajería.

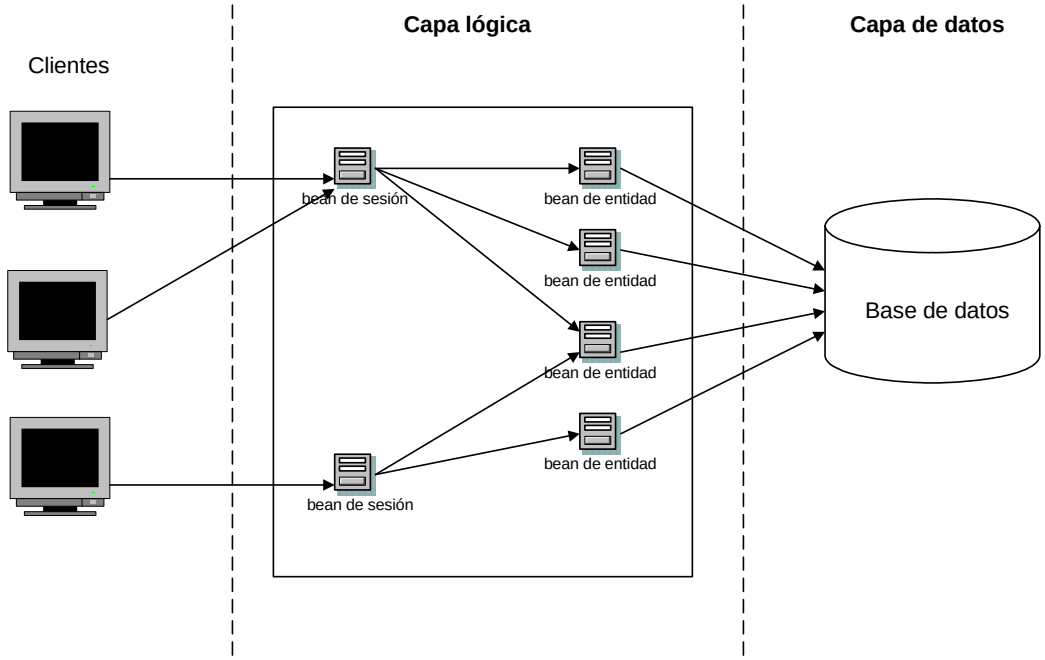

Figura 2 – *Aplicación EJB*

2.3 Reseña de la tecnología JDO

2.3.1 ¿Qué es JDO?

El *Framework Java Data Object* es un modelo abstracto de persistencia que permite acceder a los datos, sin la necesidad de escribir la lógica de almacenamiento relacionada con tal acceso [20]. Puede decirse que el foco principal de JDO es la persistencia de las instancias residentes en la memoria, sobre un medio de almacenamiento que permita preservarlas para necesidades futuras, sin requerir de la codificación de instrucciones para persistir y recibir los datos.

El impulso dado al Framework JDO por la comunidad de desarrolladores de software, se debe a la noción de persistencia transparente que este soporta, la cual puede ser resumida en los siguientes items:

- *Mapeo Objeto-Relacional transparente*. Cada atributo que conforma la clase es asociado a una o varias columnas de una o varias tablas del ambiente relacional en forma transparente.

- *Independencia del paradigma de almacenamiento*. El Framework JDO puede ser utilizado en ambientes con base de datos relacionales o con base de datos orientadas a objetos.

- *Clases más claras*. Considerando que el Framework JDO es el responsable de almacenar los objetos, no se necesita escribir el código de persistencia que dificulte la comprensión de la lógica de la clase.

- *Portabilidad*. La independencia de la plataforma de almacenamiento hace que la migración de las aplicaciones se logre con un mínimo esfuerzo.

Todos los servicios de persistencia ofrecidos por la plataforma JDO están disponibles solo a clases construidas bajo el lenguaje de programación Java y son accedidos por medio de un conjunto de interfaces que constituyen la API[3] de la arquitectura [20].

2.3.2 Persistencia de objetos con JDO

Como mencionamos, la principal característica de JDO es la persistencia transparente de los objetos es decir, la arquitectura elimina la necesidad de codificar instrucciones de almacenamiento para el acceso a los datos.

[3] Una API (del ingles Application Programming Interface - interfaz de programación de la aplicación) es un conjunto de especificaciones de comunicación entre componentes software.

La tecnología JDO requiere que los desarrolladores lleven a cabo dos grandes actividades para poder acceder a sus servicios [20].

En la primera actividad los desarrolladores definen las clases Java que serán persistidas, comunmente estas representan las entidades de dominio del problema. Cada una de las clases contendrá solo la lógica de negocio requerida por la aplicación.

Una vez definidas las clases a ser persistidas, el desarrollador construye un archivo xml[4] llamado *Descriptor de Persistencia*, el cual define la información que se usará para el almacenamiento de la clase.

Luego, el desarrollador ejecuta el proceso de ´enhanced´[5] con el objeto de agregar el comportamiento persistente a las clases definidas. Este proceso toma el modelo de clases Java más el *Descriptor de Persistencia* y como resultado, produce nuevas clases con todo lo necesario para que puedan ser tratadas por el ambiente JDO.

En la segunda actividad, los desarrolladores invocan los servicios de persistencia por medio de la API ofrecida por el Framework JDO.

2.3.3 Arquitectura de JDO

Anteriormente se comentó que el Framework JDO ofrece una serie de interfaces que constituyen la API de la arquitectura y que permiten a las aplicaciones acceder a las funciones de persistencia. Dentro de este conjunto, las más importantes son la interfaz llamada

[4] XML es el acrónimo de eXtensible Markup Language. Lenguaje de marcado extensible que puede usarse para almacenar datos en un formato estructurado, basado en texto y definido por el usuario.

[5] El proceso de 'enhanced' es aquel que modifica la implementación de una clase para agregarle comportamiento persistente.

Fábrica de Persistencia[6] y la interfaz llamada *Administrador de Persistencia*[7], ambas son claves para el acceso a los servicios del Framework JDO [20].

Todas las interfaces que conforman la tecnología JDO definen los servicios de persistencia sin brindar una implementación de los mismos. Para poder llevar a cabo el almacenamiento de las instancias, es necesario acceder a un conjunto de clases concretas que implementen las definiciones de estas interfaces. Este conjunto de clases recibe el nombre de *Implementación de JDO* y es proporcionado por el *vendedor de JDO* [20].

Por último, cada implementación del Framework JDO es dependiente del medio de almacenamiento que se emplee para persistir los datos.

2.3.3.1 Instancia JDO

Como mencionamos, los servicios ofrecidos por la plataforma JDO están disponibles para las clases construidas bajo el lenguaje de programación Java.

El término *Instancia JDO*, es usado para describir una clase construida en el lenguaje Java que puede ser tratada por el Framework JDO [20]. Es decir, una clase común Java que contiene el comportamiento necesario para poder persistirse sobre el ambiente JDO.

Para que una clase pueda tener la funcionalidad de persistirse, debe ser complementada con un código que sea comprendido por el

[6] La Fábrica de Persistencia en una interfaz usada para acceder a los servicios de persistencia brindados por el Administrador de persistencia.

[7] El Administrador de Persistencia es la interfaz primaria del Framework JDO. Contiene métodos para acceder a los servicios de persistencia.

Framework JDO. Este enriquecimiento de la clase se logra a través de la definición de la clase en el *Descriptor de Persistencia* y el proceso de ´enhanced´.

2.3.3.2 Ambientes en JDO

Los servicios que brinda el Framework JDO pueden ser usados de dos formas distintas [20].

En la primera manera, es la aplicación la que invoca los servicios de persistencia del Framework; este es llamado *ambiente no administrado*.

En el segundo caso, los servicios de persistencia son invocados por los componentes de software de sistemas distribuidos a través de una interfaz estándar. Este escenario recibe el nombre de *ambiente administrado*.

2.3.3.2.1 Ambiente no administrado

En el ambiente no administrado, es la aplicación la responsable de acceder a los servicios de persistencia del Framework. Esto, incluye obtener la interfaz llamada *Fábrica de Persistencia* y la interfaz llamada *Administrador de Persistencia*, administrar las transacciones e invocar a las operaciones de persistencia [20].

2.3.3.2.2 Ambiente administrado

En el ambiente administrado, el Framework JDO es integrado sobre una arquitectura distribuida basada en la especificación J2EE. Este ambiente, está compuesto por un conjunto de componentes de software que residen dentro de un *Servidor de Aplicaciones* e invocan los servicios de persistencia del Framework JDO, a través de una

interfaz estándar [20]. Es decir, los componentes de software son los encargados de obtener la interfaz llamada *Fábrica de Persistencia* y la interfaz llamada *Administrador de Persistencia* e invocar a las operaciones de persistencia.

En este esquema el manejo de las transacciones está coordinado entre ambas tecnologías.

2.3.3.3 El lenguaje JDOQL

El Framework JDO, además de brindar un conjunto de servicios para la persistencia de las instancias, ofrece un lenguaje de consulta para acceder a los objetos administrados por él. Este lenguaje de consulta recibe el nombre de lenguaje JDOQL[8] y permite que las aplicaciones puedan realizar consultas sobre los objetos persistentes administrados por la tecnología JDO [20].

2.3.4 Instancias JDO persistentes y transitorias

Anteriormente se comentó que una *Instancia JDO* es una clase común Java que contiene el comportamiento necesario para poder persistirse sobre el ambiente JDO.

Las *Instancias JDO* pueden ser de dos tipos, transitorias o persistentes [20].

Las *Instancias JDO transitorias* son aquellas que no han sido persistidas, es decir, que no son administradas por el Framework JDO. Estas instancias no tienen diferencia con otras instancias creadas con el lenguaje Java.

[8] JDOQL son las siglas de Java Data Object Query Language. Comprenden un conjunto de funcionalidades orientadas a la consulta de objetos persistentes.

Las *Instancias JDO transitorias* son transformadas en *Instancias JDO persistentes* cuando se almacenan utilizando los servicios del Framework JDO. En este momento la tecnología JDO es la responsable de administrar todos los procesos de almacenamiento y sincronización de las instancias.

2.4 Reseña de la arquitectura de la empresa

Aquí se presenta una reseña, que se profundizará en el *Capítulo 3,* de la arquitectura de desarrollo utilizada por una empresa de telecomunicaciones en la construcción de sus productos software. Esta arquitectura será tomada como base para analizar el impacto del Framework JDO sobre sus procesos de desarrollo.

La empresa de telecomunicaciones, de ahora en adelante llamada 'la empresa', desarrolla una familia de productos para la gestión integral de la atención de clientes y de la facturación para sí misma y para empresas de servicios especializadas en telecomunicaciones.

La arquitectura en cuestión utiliza un sistema distribuido basado en la especificación J2EE, maneja la persistencia manualmente por medio de la tecnología Enterprise JavaBeans, hace uso de varios patrones de diseño[9] recomendados para desarrollos en ambientes distribuidos con Enterprise JavaBeans, utiliza un Framework de construcción propia y divide al modelo de desarrollo de software en varias capas lógicas con responsabilidades claramente definidas.

Se decidió utilizar el modelo JDO como medio de persistencia de los objetos frente a la tecnología Enterprise JavaBeans que emplea la empresa de telecomunicaciones porque:

[9] Los patrones de diseño representan soluciones a problemas comunes en el desarrollo de software.

- Elimina la necesidad de codificar instrucciones de almacenamiento para el acceso a los datos.

- Es portable entre ambientes de almacenamiento.

- No impone limitaciones a los modelos de objetos.

- Enfoca el desarrollo sobre cuestiones del negocio y no sobre problemas de la infraestructura subyacente.

- Aumenta la productividad de los desarrolladores.

2.5 Conclusión

En este capítulo han sido expuestos una serie de conceptos y de términos tecnológicos empleados a lo largo de este trabajo. Estos elementos constituyen los requisitos necesarios para comprender el estudio realizado en esta tesis.

En los capítulos siguientes se analizará, en el plano de 'la empresa', como se trató la persistencia de las instancias. Y además, en los capítulos finales, se presentará una nueva estrategia de almacenamiento basada en JDO, junto a su aplicación en 'la empresa' en cuestión.

CAPÍTULO 3

JDO en ambientes administrados

Anteriormente se comentó que los servicios que brinda el Framework JDO pueden ser usados de dos maneras distintas. En el primer ambiente, es la aplicación la que invoca los servicios de persistencia del Framework, mientras que, en el segundo caso, los servicios de persistencia son invocados por los componentes de software que viven dentro del *Servidor de Aplicaciones* a través de una interfaz estándar. Este ambiente es más complicado que el anterior considerando que requiere la integración entre la implementación de JDO y la arquitectura de desarrollo distribuido.

Teniendo en cuenta que 'la empresa' emplea una arquitectura distribuida basada en la especificación J2EE, este capítulo muestra como los componentes de software que residen dentro del *Servidor de Aplicaciones* utilizan el Framework JDO para poder acceder a sus servicios.

3.1 Ambiente administrado

Para que la implementación de JDO pueda ser integrada en el ambiente distribuido los vendedores de JDO usan una arquitectura estándar para conectar la plataforma con los sistemas de base de datos empresariales.

Esta arquitectura de conexión actúa como puente entre ambas tecnologías [14], definiendo un conjunto de contratos entre el *Servidor de Aplicaciones* y la base de datos que son implementados en el *Adaptador de recurso* [20].

La interfaz llamada *Administrador de Persistencia*, perteneciente al Framework JDO, usa el *Adaptador de recurso* para conectarse con la base de datos empresarial. Este enlace pude realizarse en modo nativo o a través de otros vendedores [20].

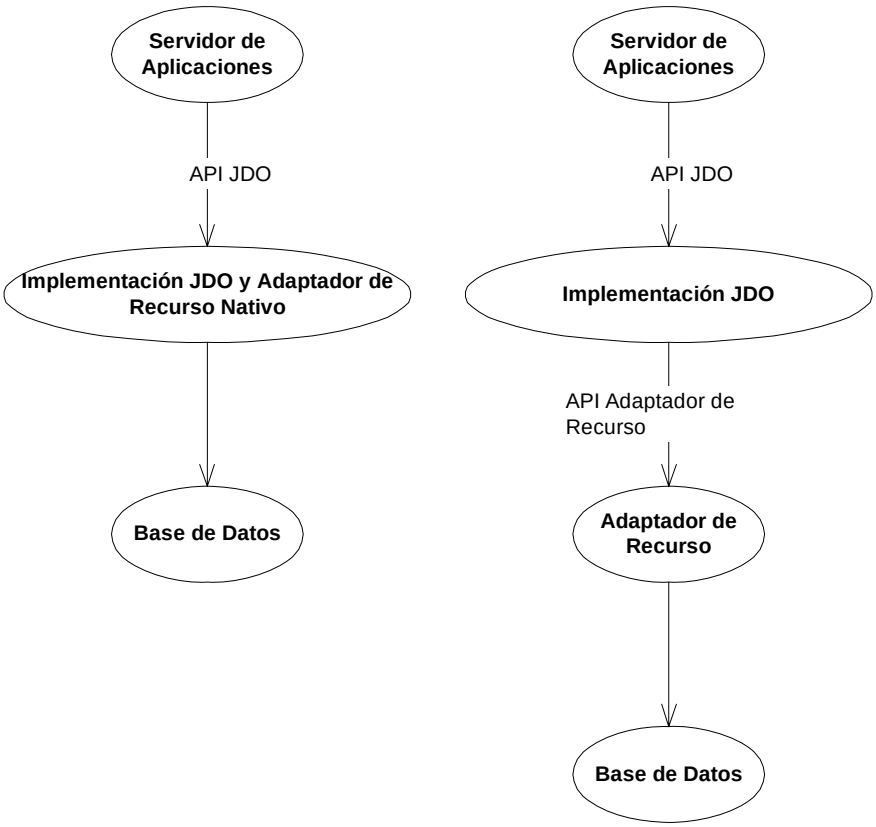

Figura 3 – *Implementación del Adaptador de recurso*

En este esquema se define una arquitectura de tres capas donde los componentes de software que viven dentro del *Servidor de Aplicaciones* y que atienden las solicitudes de los clientes, obtienen una referencia a la interfaz llamada *Fábrica de Persistencia* del Framework JDO. Esta referencia es recibida usando una interfaz estándar que ya fue configurada previamente [14].

Obtenida la referencia a la *Fábrica de Persistencia*, el componente recibe finalmente la instancia del *Administrador de*

Persistencia que le permitirá acceder a todos los servicios del Framework JDO [20].

Por último, el manejo de transacciones es combinado entre la implementación JDO y el *Servidor de Aplicaciones*. Esto, permite a los componentes de software, hacer uso del modelo de transacciones provisto por el *Servidor de Aplicaciones* o controlar las transacciones manualmente por medio de una interfaz designada para tal función [14].

3.2 Administración de las transacciones

Como se mencionó anteriormente, en el *ambiente administrado* se combina la tecnología distribuida y la tecnología JDO. En este esquema cada una provee su propio medio para manejar las transacciones.

En JDO se usa una clase retornada desde la interfaz *Administrador de Persistencia*. En cambio, en el ambiente distribuido los servicios están disponibles a través del *Servidor de Aplicaciones* para que los componentes de software puedan administrar las transacciones [14].

Bajo el ambiente administrado las transacciones de JDO pueden sincronizarse con las transacciones distribuidas suministradas por el *Servidor de Aplicaciones*. Esto permite que la implementación JDO pueda realizarse en forma totalmente transparente [14].

Los componentes de software residentes en el *Servidor de Aplicaciones* emplean dos técnicas para administrar las transacciones en la arquitectura distribuida.

El mecanismo más común es otorgar la responsabilidad al *Servidor de Aplicaciones* y no al componente del manejo de la ejecución de las transacciones. Sin embargo, si se quiere manejar la transacción manualmente la responsabilidad recae sobre el componente de software y no sobre el *Servidor de Aplicaciones* [17].

En el caso de la administración manual, se puede usar una clase proporcionada por JDO o una clase proporcionada por la arquitectura distribuida para controlar las transacciones [14].

En general se recomienda usar la clase provista por la arquitectura distribuida. Sin embargo, si no se necesita el servicio de transacciones distribuidas, pero si se requiere acceso transaccional, se debe usar las políticas de administración de transacciones del Framework JDO.

3.3 Uso de JDO en la arquitectura Enterprise JavaBeans

Como vimos, la arquitectura Enterprise JavaBeans es un modelo de componentes de software, comunmente llamados componentes EJB, para el desarrollo de aplicaciones de negocios distribuidas. Las aplicaciones construidas sobre esta arquitectura son instaladas sobre *Servidores de Aplicaciones* compatibles con la especificación J2EE [17].

Este esquema emplea tres tipos de componentes de software orientados a brindar servicios remotos a las aplicaciones [17]. El uso de JDO con cada uno de estos componentes será analizado luego por separado.

La integración entre JDO y la arquitectura distribuida permite a los componentes EJB hacer uso del Framework JDO como mecanismo de manejo de persistencia.

3.4 Integrando JDO con EJB

El primer paso a realizar en la integración de JDO con EJB es obtener la referencia de la interfaz llamada *Fábrica de Persistencia* perteneciente al Framework JDO. Anteriormente se comentó que la referencia se alcanza por medio de una interfaz estándar configurada previamente.

Comunmente, la instancia de la *Fábrica de Persistencia* se obtiene durante el proceso de configuración del contexto del EJB a través de diferentes métodos asignados para tal función [14]. Una vez recibida, ésta se almacena como un atributo dentro del componente EJB para futuras invocaciones.

Cuando el componente necesite acceder a los servicios de persistencia de JDO debe hacerlo a través de la interfaz llamada *Administrador de Persistencia*. Esta interfaz se obtiene como resultado de la invocación de un servicio publicado en la *Fábrica de Persistencia* [20].

Por último, se recomienda que cada *Administrador de Persistencia* se obtenga por cada método de negocio del EJB que use JDO y que, finalmente el recurso sea cerrado antes del retorno del método [14].

3.4.1 Integrando JDO con beans de sesión

Como mencionamos, los beans de sesión son los componentes de sesión de la arquitectura Enterprise JavaBeans y representan un punto de entrada a la aplicación que está corriendo en el servidor [17]. Estos pueden operar recordando el estado de las transacciones realizadas por el cliente (Stateful o con estado) u operar sin recordar el

estado de las transacciones realizadas por el cliente (Stateless o sin estado) [17].

Para concluir, los componentes beans de sesión pueden manejar las transacciones por medio del *Servidor de Aplicaciones* o en forma manual, en el propio componente.

3.4.1.1 Integrando JDO con beans de sesión sin estado

Anteriormente comentamos que los bean de sesión sin estado son aquellos que no mantienen un estado conversacional con el cliente [17]. Estos, generalmente, usan el manejo de transacciones ofrecido por el *Servidor de Aplicaciones*, salvo en el caso en que se requieran varias transacciones consecutivas durante la invocación de un método del bean [14].

Los bean de sesión sin estado, que utilizan los servicios de transacciones ofrecidos por el *Servidor de Aplicaciones*, no administran las transacciones, ya que es responsabilidad del *Servidor de Aplicaciones* hacerlo [17].

Una vez creado e inicializado el bean de sesión se coloca en un estado a partir del cual éste está disponible para atender solicitudes de los clientes [17].

Cada método del bean que responde a una solicitud del cliente, obtiene la interfaz *Administrador de Persistencia* para acceder a los servicios del Framework JDO [20]. Este proceso puede realizarse debido a que la transacción se inicia por el *Servidor de Aplicaciones* en forma previa a la invocación del método del EJB.

Por último, cada *Administrador de Persistencia* empleado en el método del bean debe ser cerrado antes del retorno del método [14].

A continuación se muestra la manera de utilizar los servicios de JDO en el componente bean de sesión sin estado.

```
import javax.ejb.*;
import javax.jdo.*;
import javax.naming.*;
import appl.entidades.*;

public class ProductoBean implements SessionBean {

  private SessionContext contexto = null;
  private PersistenceManagerFactory pmf = null;
  private static String pmfName = "java:comp/env/jdo/JPoxPMF";

  public void setSessionContext(SessionContext pContexto) {
     contexto = pContexto;

     try {
               Context ic = new InitialContext();
                pmf = (PersistenceManagerFactory)ic.lookup(pmfName);
     } catch (NamingException ex) {
               throw new EJBException("setSessionContext()", ex);
     }
  }

   public void unsetSessionContext()
   {
       pmf = null;
    }

  public void reducirStock(ProductoPK productoPK, int pCant)
  {
        PersistenceManager pm = null;

        try {
               pm = pmf.getPersistenceManager();
               Producto o = (Producto)pm.getObjectById(productoPK, true);
               o.reducir(pCant);
        }
        catch(Exception e){
               throw new EJBException("reducirStrock()", e);
        }
        finally {
               if(pm != null && !pm.isClosed())
                        pm.close();
        }

  }

  public void ejbActivate() {}


  public void ejbPassivate() {}


  public void ejbRemove() {}
```

```
  public void ejbCreate() {}

}
```

3.4.1.2 Integrando JDO con beans de sesión con estado

A diferencia del bean de sesión sin estado, el bean de sesión con estado recuerda las operaciones realizadas por el cliente [17].

Este componente puede administrar las transacciones manualmente en el propio bean a través de los servicios ofrecidos por la arquitectura distribuida o por la implementación JDO.

Si se emplean los servicios de transacción ofrecidos por la arquitectura distribuida dentro del bean, el *Administrador de Persistencia* debe ser obtenido luego de iniciar la transacción y debe ser cerrado antes de finalizarla [14].

En cambio, si se emplean los servicios de transacción ofrecidos por la implementación JDO, el *Administrador de Persistencia* debe obtenerse antes de la apertura de la transacción y debe ser cerrado luego que la transacción fue completada [14].

A continuación se muestra la manera de utilizar los servicios de JDO en el componente bean de sesión con estado.

```
import javax.ejb.*;
import javax.jdo.*;
import javax.naming.*;
import appl.entidades.*;

public class ProductoBean implements SessionBean {

    private SessionContext contexto = null;
    private PersistenceManagerFactory pmf = null;
    private static String pmfName = "java:comp/env/jdo/JPoxPMF";
    PersistenceManager pm = null;
```

```
private Producto p = null;

private UserTransaction ut = null;

public void setSessionContext(SessionContext pContexto) {
   contexto = pContexto;

   try {
            Context ic = new InitialContext();
            pmf = (PersistenceManagerFactory)ic.lookup(pmfName);
            ut = contexto.getUserTransaction();
   } catch (NamingException ex) {
            throw new EJBException("setSessionContext()", ex);
   }
}

public void obtenerProducto(ProductoPK productoPK){
     try {
            ut.begin();
            pm = pmf.getPersistenceManager();
            p = (Producto)pm.getObjectById(productoPK, true);
     }
     catch(Exception e){throw new EJBException("reducirStrock()", e);}
}

public void decrementar(int pCant){p.decrementar(pCant);}

public void incrementar(int pCant){p.incrementar(pCant);}

public void completar(){
            try
            {
                    pm.close();
                    ut.commit();
            }
            catch(Exception e){throw new EJBException(“completar()”, e);}
}

public void descartar(){
            try
            {
                    pm.close();
                    ut.rollback();
            }
            catch(Exception e){
                    throw new EJBException(“descartar()”, e);
            }
}


public void ejbActivate() {}


public void ejbPassivate() {}
```

```
    public void ejbRemove() {}

    public void ejbCreate() {}
}
```

3.4.2 Integrando JDO con beans de entidad

Los beans de entidad son los componentes que representan la vista de una instancia persistida en un medio de almacenamiento [13]. Este componente opera solo sobre contextos transaccionales administrados por el *Servidor de Aplicaciones*.

Los beans de entidad tienen dos mecanismos de persistencia. El primero, permite que el *Servidor de Aplicaciones* sea el responsable del manejo de la persistencia. El segundo, delega en el componente bean la lógica de persistirse [17].

Generalmente cuando el bean maneja la persistencia, se emplean instrucciones de base de datos dentro del componente como mecanismo de almacenamiento. Sin embargo, puede utilizarse el Framework JDO como alternativa de persistencia del componente bean.

Los beans de entidad que emplean JDO como mecanismo de persistencia hacen referencia a una o varias instancias JDO. Cada método de administración utilizado por el *Servidor de Aplicaciones* para sincronizar el bean con su instancia persistente en la base de datos, debe manipular la/las instancia/s JDO por medio de un *Administrador de Persistencia* [17].

A continuación se muestra la manera de utilizar los servicios de JDO en el componente bean de entidad.

```
import javax.ejb.*;
import javax.jdo.*;
import javax.naming.*;
import appl.entidades.*;

public class ProductoBean implements EntityBean {

   private EntityContext contexto = null;
   private PersistenceManagerFactory pmf = null;
   private static String pmfName = "java:comp/env/jdo/JPoxPMF";
   PersistenceManager pm = null;

   private Producto p = null;

   public void setEntityContext(EntityContext pContexto) {
      contexto = pContexto;

      try {
         Context ic = new InitialContext();
         pmf = (PersistenceManagerFactory)ic.lookup(pmfName);
      } catch (NamingException ex) {
         throw new EJBException("setSessionContext()", ex);
      }
   }

    public ProductoPK ejbCreate(int pId, String pDesc)
   {

        pm = pmf.getPersistenceManager();

        p = new Producto(pId, pDesc);
        pm.makePersistent(p);

        return (ProductoPK) JDOHelper.getObjectId(p);
    }

    public void ejbPostCreate() {}

    public void ejbLoad() {

        p = (Producto) pm.getObjectById(
                (ProductoPK) contexto.getPrimaryKey(), true);

    }

    public void ejbStored() {}

    public void ejbPassivate() {
        p = null;
        pm.close();
        pm = null;
    }

    public void ejbActivate() {
        if(pm == null || pm.isClosed())
                pm = pmf.getPersistenceManager();
    }

    public void ejbRemove() {
```

```
        pm.deletePersistent(p);
        pm.close();

    }

    public ProductoPK ejbFindByPrimaryKey(ProductoPK pk) throws FinderException
    {
        try
        {
                pm = pmf.getPersistenceManager();
                Producto p = (Producto) pm.getObjectById(pk, true);
        }
        catch(Exception e)
        {
                throw new FinderException("Producto no encontrado.", e);
        }
        finally
        {
                pm.close();
        }

        return  pk;
    }

    public void unsetEntityContext()
    {
        pmf = null;
    }

    public String getDescripcion()
    {
        return p.getDescripcion();
    }

    public void setDescripcion(String pDesc)
    {
        p.setDescripcion(pDesc);
    }

    public int getId(){return p.getId();}

    public void setPrecio(double pPrecio)
    {
        p.setPrecio(pPrecio);
    }

    public double getPrecio()
    {
        p.getPrecio();
    }

}
```

3.4.3 Integrando JDO con beans de mensajería

El bean de mensajería es el componente de administración de mensajes de la arquitectura Enterprise JavaBeans. Este es llamado por el *Servidor de Aplicaciones* cuando un mensaje asincrónico que llega a destino debe ser atendido [17].

Cuando se usa el Framework JDO con un bean de mensajería, la *Fábrica de Persistencia* debe ser obtenida durante el proceso de configuración del contexto del EJB a través de una interfaz estándar definida previamente. El método del componente, que es responsable de atender la solicitud, debe obtener una instancia del *Administrador de Persistencia*, usarla y finalmente cerrarla antes de su retorno [14].

A continuación se muestra la manera de utilizar los servicios de JDO en el componente bean de mensajería.

```
public class ProductoBean implements MessageDrivenBean {

   private MessageDrivenContext contexto = null;
   private PersistenceManagerFactory pmf = null;
   private static String pmfName = "java:comp/env/jdo/JPoxPMF";

   public void setMessageDrivenContext(MessageDrivenContext pContexto) {
      contexto = pContexto;
      try {
         Context ic = new InitialContext();
         pmf = (PersistenceManagerFactory)ic.lookup(pmfName);
      } catch (NamingException ex) {
         throw new EJBException("setMessageDrivenContext()", ex);
      }
   }

   public void onMessage(Message m) {
      PersistenceManager pm = null;

      try
      {
        if(m instanceof TextMessage)
        {
                String s = ((TextMessage) m).getText();
                pm = pmf.getPersistenceManager();

                Producto p = new Producto();
                p.setDescripcion(s);
```

```
                pm.makePersistent(p);
            }
        }
        catch (Exception ex)
        {
            throw new EJBException("onMessage()", ex);
        }
        finally
        {
            if(pm != null && !pm.isClosed())
                pm.close();
        }
    }

    public void ejbRemove() {}

    public void ejbCreate() {}

}
```

3.5 Conclusión

La arquitectura Enterprise JavaBeans es, un modelo basado en componentes de software para el desarrollo de aplicaciones distribuidas. El Framework JDO puede integrarse perfectamente a través de una arquitectura estándar, que actúa como puente entre la plataforma y los sistemas de base de datos empresariales.

Para la integración de ambas tecnologías, es necesario realizar una serie de configuraciones sobre el *Servidor de Aplicaciones* y determinar la forma en que los componentes beans de la arquitectura distribuida, utilizan los servicios de persistencia de JDO.

Las clases persistentes de JDO son usadas por los componentes beans de sesión para acceder a los objetos de negocio, por los beans de entidad para implementar su lógica de persistencia y por los componentes beans de mensajería para tratar mensajes asincrónicos.

CAPÍTULO 4

Recopilación de la arquitectura de software de la empresa

En este capítulo se detalla una visión general de la arquitectura de software utilizada por 'la empresa' en el desarrollo de productos que apuntan a satisfacer las necesidades de las empresas de telecomunicaciones.

Se presentan los distintos tipos de componentes que se emplean para modelar los objetos de negocio, comúnmente llamados entidades del dominio, y las asociaciones entre ellos. También, se explican los restantes elementos de la arquitectura que brindan los servicios de la capa de negocio[10] y que permiten la comunicación entre el cliente y el servidor.

Como se mencionó, 'la empresa' emplea un sistema distribuido basado en la especificación J2EE, maneja la persistencia manualmente por medio de la tecnología Enterprise JavaBeans, hace uso de varios patrones de diseño recomendados para desarrollos en ambientes distribuidos con EJB, utiliza un Framework de construcción propia y se apoya en la utilización de un *Servidor de Aplicaciones* que facilita la ejecución y el desarrollo de las aplicaciones de software.

Por último, es importante destacar que, si bien se abordan temas de la arquitectura que envuelven tanto a la capa del cliente como la capa del servidor, se pondrá mayor énfasis en esta última, considerando que la implementación del Framework JDO afecta a la capa del lado del servidor.

[10] La capa de negocio es la responsable de ofrecer toda la funcionalidad requerida por el sistema.

4.1 Capas lógicas

El modelo de desarrollo de software de 'la empresa' hace uso de varias capas lógicas con responsabilidades claramente definidas. A continuación, se describe cada una de estas capas:

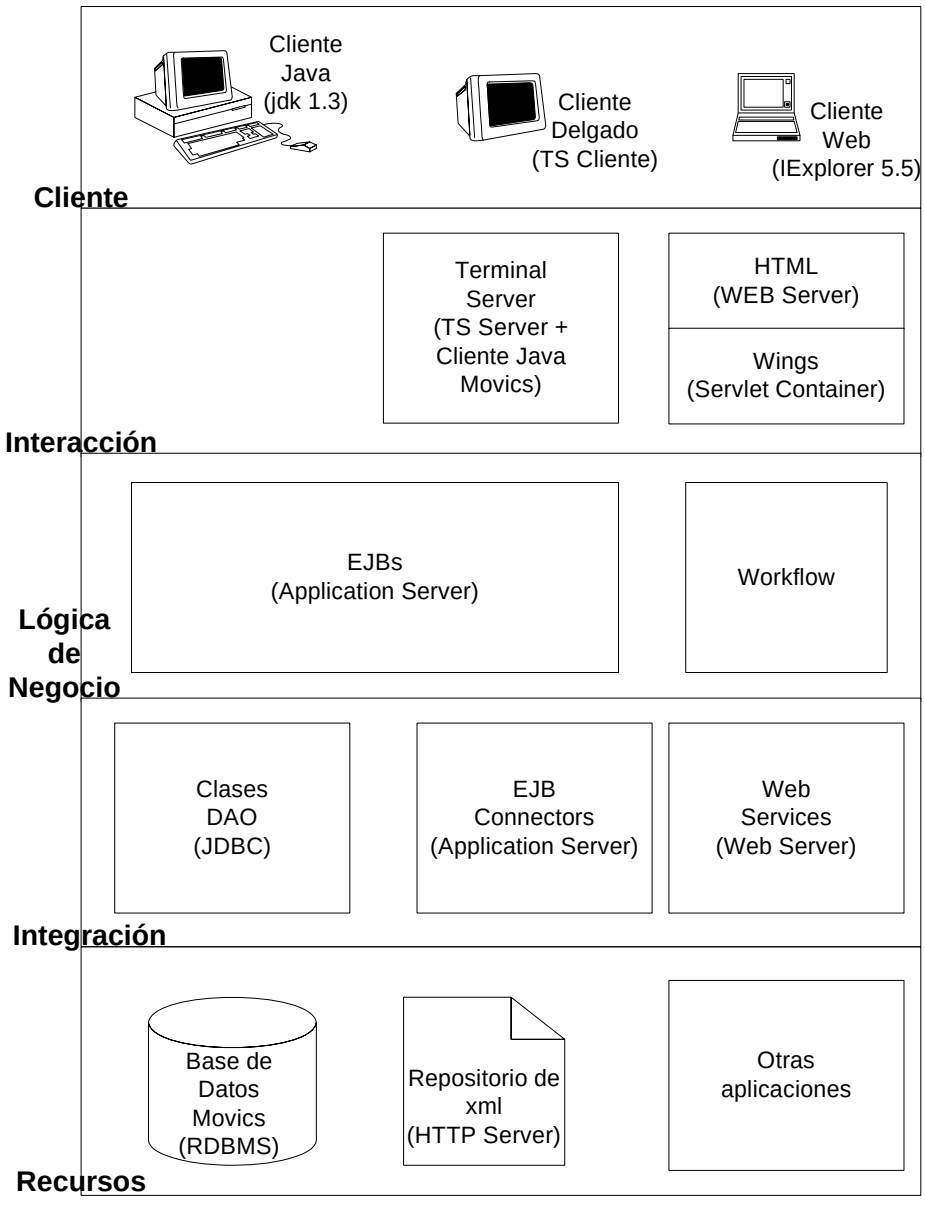

Figura 4 – *Capas lógicas*

- ***Cliente:*** Incluye los componentes de la aplicación que se ejecutan en el cliente y que son los encargados de permitir a los usuarios utilizar el sistema. Es importante

establecer que los clientes encargados de servir como punto de acceso al sistema pueden variar. Los clientes pueden ser aplicaciones java, cliente terminal y cliente web.

- ***Presentación*:** Contiene los componentes que se encargan de presentar la información y manejar la interacción con el usuario. Esta capa hace de puente a la capa de *Lógica de Negocio*.

- ***Lógica de Negocio*:** Compuesta por elementos que conforman la lógica del negocio brindada a los clientes. Esta capa contiene componentes que residen en un *Servidor de Aplicaciones* y utilizan la arquitectura Enterprise JavaBeans.

- ***Integración*:** Esta capa tiene componentes cuyo objetivo es conectar la capa de negocio con los recursos necesarios para lograr la funcionalidad deseada.

- ***Recursos*:** Esta capa abarca todos los elementos que componen los recursos de la aplicación. Estos pueden ser de características muy diferentes. Entre ellos se encuentran: base de datos, ERP, archivos de sistema, etc.

4.2 Capas lógicas y los componentes de software

La arquitectura de desarrollo de 'la empresa' utiliza diversos componentes de software para la programación de las opciones funcionales que componen a los productos de 'la empresa'. En la Figura 5 se muestran las principales clases que se deben implementar en cada capa.

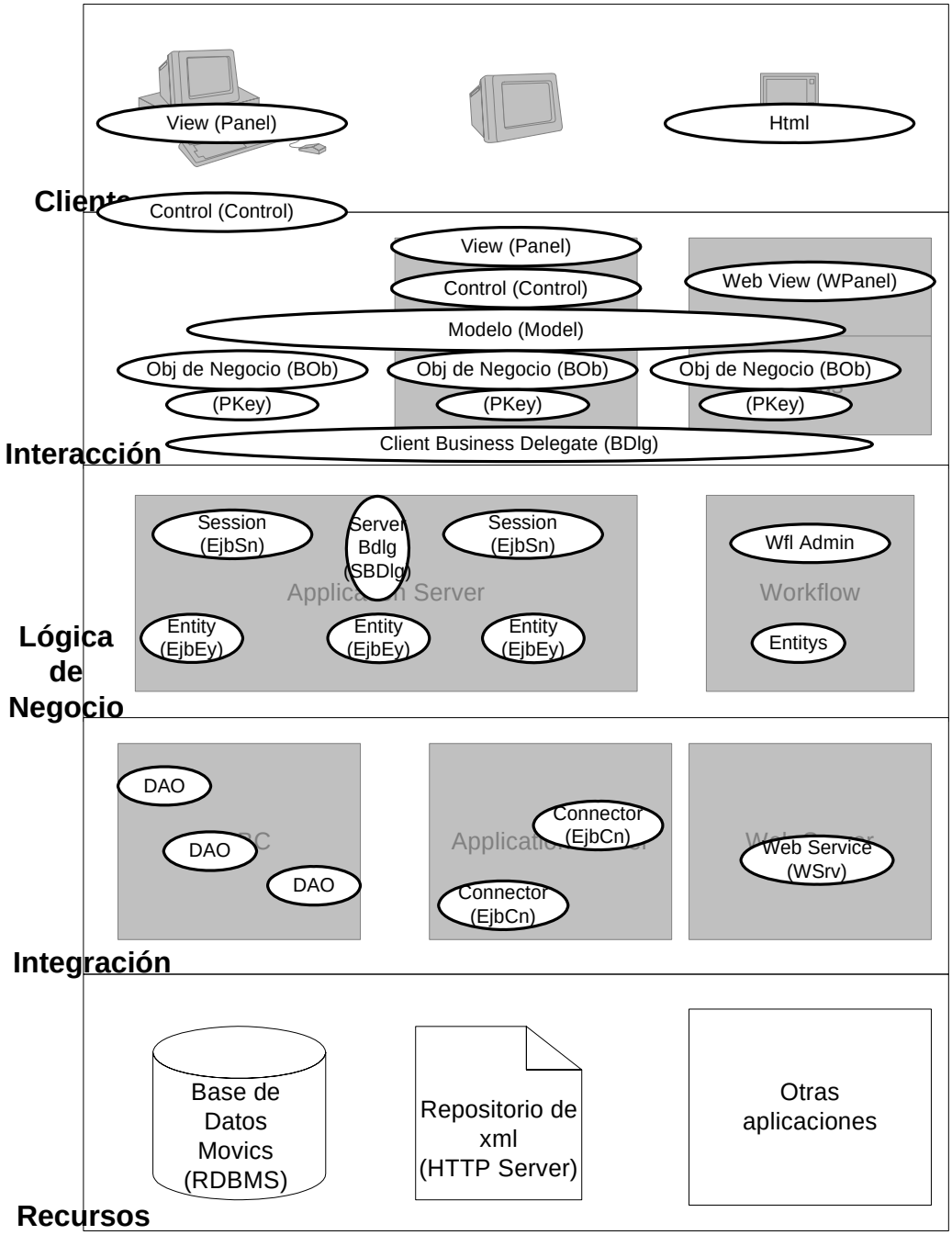

Figura 5 - *Componentes por capa lógica*

4.3 Mecanismos de persistencia

La implementación de los servicios que brindan los productos de software está basada en la especificación J2EE y en el Framework Enterprise JavaBeans como mecanismo de persistencia de los objetos

del dominio. Además, la empresa cuenta con un Framework de desarrollo propio que brinda una capa de funcionalidad adicional, por encima de EJB, que simplifica su programación y su utilización.

Como se mencionó, la arquitectura de software está basada en el uso de varios patrones de diseño que facilitan el mantenimiento, la escalabilidad y la adaptabilidad de las aplicaciones.

Junto a la utilización del patrón de diseño *Facade*, recomendado para el desarrollo de aplicaciones que emplean Enterprise JavaBeans para encapsular la complejidad de las interacciones y proporcionar un servicio de acceso uniforme a los clientes [11], se inserta un nuevo componente denominado *ApplicationLogic* responsable de implementar toda la lógica de los servicios que serán entregados por la capa del Servidor.

Los componentes beans de sesión que actúan como *Facade* y que representan un punto de entrada a la aplicación que está corriendo en el servidor [17], delegan su implementación al componente *ApplicationLogic* que tienen asociado.

Es importante destacar que la arquitectura de software hace uso del patrón de diseño *Business Delegate*. Este componente reside en la capa de presentación y en beneficio del cliente solicita los métodos remotos publicados en el componente beans de sesión que actúa como *Facade* [11].

También se emplean los patrones *Data Transfer Object* y *Data Access Object.*

El primero representa objetos serializables para la transferencia de datos sobre la red [11] y el segundo consiste en

entidades que abstraen y encapsulan todos los accesos a la fuente de datos [15].

En la arquitectura, el almacenamiento de los objetos del dominio, se logra delegando en los componentes beans de entidad la lógica de persistencia [17].

Por último, la administración de las relaciones entre los componentes persistentes se realiza automáticamente por el Framework de 'la empresa', permitiendo recuperar las entidades de negocio a medida que se solicitan.

4.4 Objetos de negocio y su representación en la empresa

Los objetos de negocio representan a las entidades del dominio de negocio que se quiere modelar. Estas entidades no están aisladas sino que establecen asociaciones unas con otras.

En estas asociaciones pueden existir relaciones especiales del tipo *todo-parte*. Este tipo de relación establece que la existencia de un objeto que representa la parte, pierde sentido al eliminarse el objeto que representa el todo.

La arquitectura permite modelar los objetos persistentes del dominio y sus relaciones diferenciando los objetos que representan al todo de los objetos que representan a las partes. De acuerdo con esto se definen los siguientes tipos de componentes:

- ***Entity:*** Representa una entidad del dominio que tiene sentido por si misma (todo).

- ***ODep(Objeto Dependiente):*** Representa un objeto que es parte de un Entity. Su vida esta limitada a la vida del Entity.

Dentro de los objetos dependientes, existe una discrepancia basada en su forma de persistencia. La persistencia del objeto dependiente, puede ser en la misma tabla donde se persiste el Entity asociado (denominado *InnerODep*) o en una tabla distinta.

Cuando las relaciones entre las entidades son de cardinalidad múltiple, la arquitectura define dos tipos de componentes: *EntityCollection* que representa una asociación con cero o n entidades y *ODepCollection* que representa la asociación con cero, uno o n objetos dependientes.

En la figura que se presenta a continuación, se ilustran los distintos tipos de relaciones que se pueden establecer, entre los componentes de la arquitectura, que modelan los objetos persistentes del dominio.

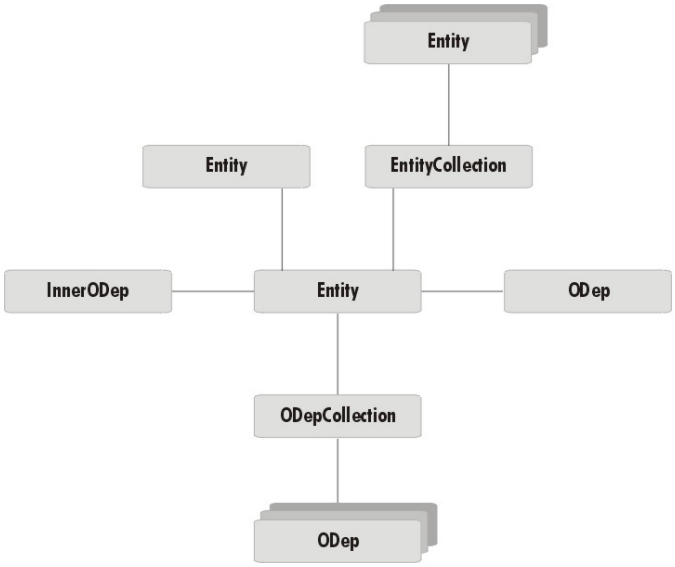

Figura 6 - *Relaciones entre entidades*

Aquí se observa a una entidad (Entity) asociada a otra entidad (Entity) o a un objeto dependiente (ODep). Estas asociaciones están implementadas por atributos en la entidad, que almacenan las referencias a los objetos asociados.

Cuando una entidad (Entity) se asocia con cero o muchas entidades (Entity), se emplea el objeto *EntityCollection* para modelar la relación. Sin embargo, cuando una entidad (Entity) se asocia con cero o muchos objetos dependientes (ODep), se utiliza el objeto *ODepCollection* para representar la relación.

Por último, cabe destacar que no puede existir una relación entre una entidad (Entity) y cero o muchos objetos dependientes *InnerODep*. Esto es debido a la propia definición del objeto dependiente interno, la cual establece que su información forma parte de la misma tabla que el Entity asociado.

4.5 Diagrama de clases de objetos de negocio

En la Figura 7, se presenta el diagrama de clases que modela las entidades, los objetos dependientes y las relaciones entre ellos.

Como se mencionó, el almacenamiento de los objetos del dominio, se logra delegando en los componentes beans de entidad la lógica de persistencia.

En la arquitectura, la mayor parte de la lógica relacionada con el almacenamiento de las entidades, se encuentra en la clase EntityBMP. Esta clase, a través del uso del patrón *Method Template*, invoca métodos implementados en las subclases, para obtener la funcionalidad deseada [4]. Cada una de estas subclases representa a los objetos de negocio que necesitan ser persistidos.

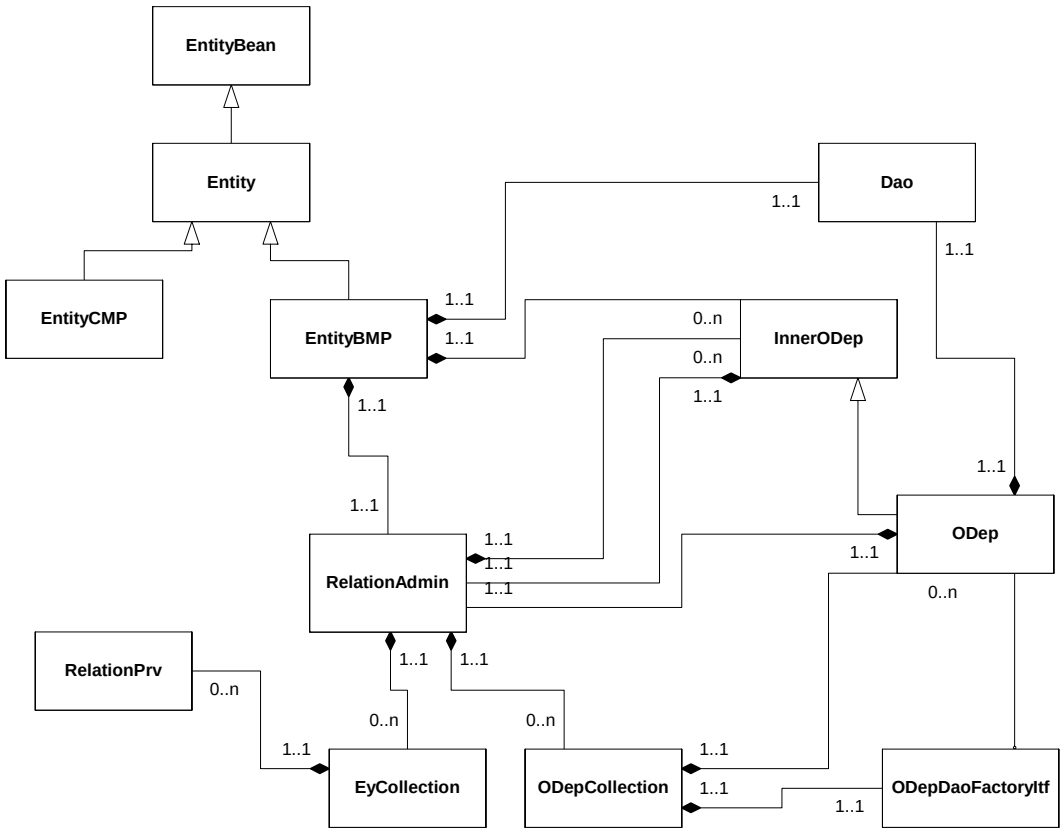

Figura 7 - *Diagrama de clases*

4.6 Componentes de la arquitectura de software

Varios de los componentes, que forman la arquitectura de software, son generados por herramientas automáticas que analizan archivos xml. Esto permite agilizar todo el proceso de implementación de los servicios, que componen la capa del negocio y de las entidades persistentes, que modelan los objetos del dominio.

A continuación, se enuncian los generadores con el fin de comprender cuales son los componentes generados. Estos elementos, constituyen un punto central en la arquitectura, para la implementación de los servicios, destinados a satisfacer las diversas solicitudes de los clientes remotos.

Los generadores de código se denominan Stubber y EntityBuilder.

- *Stubber:* Esta herramienta genera automáticamente los componentes que establecen la comunicación entre el cliente y el servidor. El generador, toma como dato de entrada un archivo xml, que especifica los servicios que se quieren implementar. A partir de este xml genera los siguientes componentes:

    - *ApplicationLogicInterface:* Este componente es una interfaz que contiene la declaración de todos los servicios, brindados por la capa del servidor. La interfaz es implementada por el componente *ApplicationLogic* y por el componente *BusinessDelegate*.

    - *Session:* Es un componente bean de sesión, que actúa como *Facade* y que contiene un conjunto de métodos brindados en la capa de servicios. Todos estos métodos son invocados por el componente *BusinessDelegate*, que reside en el cliente. El componente bean de sesión no posee la implementación de los servicios, sino que los publica. Su función es actuar como un proxy al *ApplicationLogic*. La implementación de los métodos del componente bean de sesión se delega en la invocación del método correspondiente del *ApplicationLogic*.

    - *BusinessDelegate*: Este componente hace transparente a los clientes la invocación de los servicios referentes a la lógica de negocio. Esta

clase actúa como un proxy a los servicios brindados por el bean de sesión. Es decir, cada método del componente, delega sus responsabilidades en el componente bean de sesión del lado del servidor. Este componente es del tipo Singleton[11].

- *Factory*: Este componente se utiliza tanto en la capa del cliente como en la capa del Servidor. Del lado del cliente retorna la clase *BusinessDelegate,* mientras que del lado del servidor, retorna la clase *ApplicationLogic*. Ambas clases implementan la interfaz *ApplicationLogicInterface*.

- *EntityBuilder:* Esta herramienta, genera automáticamente los componentes que representan los objetos de negocio. Como se definió anteriormente, los objetos de negocio son modelados a través de entidades (entity) y objetos dependientes (ODep). También, se genera un conjunto de clases, que administran la persistencia en la base de datos correspondiente. El generador toma como dato de entrada un archivo xml. En este archivo de xml se define toda la estructura de la entidad. Es decir, el mapeo de los atributos a sus correspondientes columnas de la tabla de la base de datos, las relaciones con otras entidades u objetos dependientes, tipo de entidad, etc. Las clases generadas se describen a continuación:

  - *Entity:* Este componente abstracto representa una entidad. Contiene los atributos de la entidad, los métodos para actuar sobre esos

[11] Componente que proporciona una única instancia de un objeto.

atributos, las referencias a los objetos relacionados, los métodos para agregar y eliminar objetos de las colecciones relacionadas. Cada entidad debe ser extendida para agregar su lógica de negocio. Este componente es un bean de entidad.

- *Dao:* Esta es una clase abstracta que contiene los métodos para guardar, recuperar, modificar y eliminar datos, como así también, para manejar la conexión con la base de datos. Este componente es responsable de todos los servicios que involucran el acceso al medio de almacenamiento de datos. Cada entidad (Entity) definida tiene un componente *Dao* asociado a ella. Es decir, cada entidad gestiona su persistencia a través de su clase *Dao*. Toda clase Dao debe ser extendida para agregar la lógica particular. También, el generador construye un *Dao* abstracto para cada objeto dependiente asociado a la entidad, sin embargo, su persistencia es manejada manualmente.

- *DaoSymbols:* Esta es una clase que contiene un conjunto de constantes que representan la estructura de la tabla, cuya persistencia es manejada por el *Dao*.

- *Rec:* Este componente representa un registro en la tabla. Su función es la comunicación entre los objetos Dao y las entidades u objetos dependientes. Se genera un componente por

cada entidad y por cada objeto dependiente relacionado.

- *Primary Key:* Esta clase representa la clave primaria de cada uno de los componentes que modelan los objetos de negocio. La clase contiene los atributos que componen la clave en la tabla de la base de datos donde se persiste el objeto. Se genera una clave primaria para cada entidad y para cada objeto dependiente asociado a la entidad.

- *Relation Provider*: Este componente se utiliza para gestionar las relaciones entre las entidades y los objetos dependientes.

- WrapperNaw*:* Los componentes gráficos esperan tratar con clases que implementen una determinada interfaz. Sin embargo, el pasaje de información entre el cliente y el servidor se realiza a través de objetos serializables. Para facilitar el pasaje de información y su posterior interpretación por la capa visual, se generan componente WrapperNaw, que encapsulan a los objetos serializables e implementan la interfaz requerida.

- Entity Home Abstracto*:* Es la interfaz del bean de entidad, que permite obtener las referencias a las entidades. Esta clase contiene los métodos para crear y obtener una entidad. También, es posible definir métodos que retornen colecciones

de entidades. Todo método declarado debe ser implementado en su correspondiente entidad.

- Entity Local Abstracto*:* Es la interfaz del bean de entidad, que permite interactuar con la entidad. Por defecto contiene los métodos que operan sobre los atributos de las entidad. El programador puede declarar métodos de negocio que deben ser implementados en la entidad.

Anteriormente, se mencionó el componente *ApplicationLogic*, sin realizar mayores explicaciones. Es importante mencionar que esta clase es relevante para la comunicación entre el cliente y el servidor. Su responsabilidad es implementar todos los servicios brindados por la capa de negocio.

El *ApplicationLogic* no es un componente generado automáticamente. El desarrollador debe implementar en esta clase cada servicio publicado en la interfaz *ApplicationLogicInterface.*

Por último, cada clase que necesitaba obtener las interfaces de los Enterprise JavaBeans utiliza el patrón S*erviceLocator*. Esta clase tiene métodos para pedir interfaces locales o remotas [16].

4.7 Solicitudes de la capa del cliente

Esta capa atiende los pedidos de los usuarios. Como respuesta a estos pedidos, realiza demandas a la capa de servicios a través de los componentes *ApplicationLogicInterface, Session, BusinessDelegate* y *Factory*. Todos estos componentes encapsulan la lógica de comunicación haciendo que los pedidos del cliente sean totalmente transparentes.

La comunicación con el servidor se realiza a través de la invocación remota de métodos. La arquitectura de software hace uso del patrón *Facade*, para encapsular la complejidad de estas interacciones.

La capa del cliente no realiza solicitudes directamente a las entidades, sino que invoca a los servicios del componente bean de sesión, que actúa como *Facade,* para gestionar sus necesidades [11].

El cliente puede enviar al servidor los argumentos que pueden necesitar sus servicios y el servidor puede devolver información como resultado del procesamiento de un pedido.

En la arquitectura existen dos componentes, que tienen la responsabilidad del intercambio de información entre la capa del cliente y la capa del servidor. Ambos componentes modelan el patrón *Data Transfer Object*, que representa objetos serializables para la transferencia de datos sobre la red [11].

Los componentes que permiten el pasaje de información son CObj y Par:

- *CObj:* Este componente contiene información que se envía desde el servidor al cliente. Es un componente que contiene métodos, que operan sobre los atributos que necesitan ser utilizados por el cliente. El *CObj* es armado y retornado desde la capa de servicios. Luego, el cliente encapsula el *CObj* en objetos *WrappersNaw*, para que la información recibida pueda ser tratada en los componentes gráficos.

- *Par:* Este componente, contiene información que se envía desde el cliente al servidor en las invocaciones de los

servicios. Al igual que el *CObj*, este componente contiene métodos, que operan sobre los atributos, que necesitan ser utilizados por el servidor, para procesar las solicitudes del cliente.

Dentro de las solicitudes, que puede realizar un cliente, tenemos aquellas que involucran a una entidad y aquellas que involucran a un conjunto de entidades. En la arquitectura se estableció, que cualquier pedido que afecte a una entidad debe ser resuelto por el *ApplicationLogic* y la propia entidad. Mientas que, si el pedido involucra a una lista de entidades, debe ser tratado directamente por el componente *Dao*. Es decir, que la solicitud sería atendida por el *ApplicationLogic* que delega la responsabilidad de resolver el pedido en el *Dao*.

Cuando una solicitud que afecta a varias entidades llega al *Dao*, se requiere una transformación en el componente *CObj* de los datos resultantes. Este problema fue resuelto por medio del uso del patrón *Strategy*, que permite definir distintas estrategias de transformación del conjunto de datos resultante [4].

Anteriormente, se mencionó a un conjunto de componentes correspondientes a la capa del servidor. Es importante conocer, que existen otros elementos que forman la capa del cliente. Estos componentes no serán explicados, ya que la aplicación de la infraestructura JDO afecta a la capa del lado del servidor.

Por último, la Figura 8 muestra la arquitectura de desarrollo, que emplea ‘la empresa’ para la construcción de todos sus productos de software, que apuntan al sector de telecomunicaciones.

4.8 Interacción entre los componentes

En la Figura 9, se ilustra un caso simple, que muestra la secuencia de mensajes intercambiados entre los componentes que conforman la arquitectura de software de 'la empresa'. Estos mensajes son el resultado de una solicitud iniciada por el usuario.

El usuario inicia una solicitud que se atiende en la capa de presentación. Esta capa, encapsula el pedido en el componente *Par* y lo envía a la capa de negocio, por medio del componente *BusinessDelegate*.

El componente *BusinessDelegate*, al recibir una solicitud, realiza llamadas a métodos remotos, publicados en la interfaz del componente bean de sesión, que actúa como *Facade*. Al llegar el pedido a la capa de negocio, es tratado por el *ApplicationLogic*, que determina si se resuelve como una solicitud a una entidad de negocio (modelada con un elemento del Framework Enterprise JavaBeans), o como una solicitud directa a la base de datos, por medio del componente *Dao*. El resultado se encapsula y se retorna al cliente.

Por último, al llegar la respuesta del pedido a la capa de presentación se procesa y finalmente se muestra al usuario.

4.9 Conclusión

La arquitectura de software representa la estructura jerárquica de los componentes del programa, la manera de actuar de estos componentes y la estructura de datos usados por estos componentes [12].

La arquitectura de software de 'la empresa', emplea las tecnologías J2EE y Enterprise JavaBeans para el desarrollo de sus

productos de software, que apuntan a satisfacer las operaciones de una empresa de telecomunicaciones. El modelo de desarrollo, utiliza patrones de diseño recomendados para definir un conjunto de componentes, de relaciones y de interacciones con el fin de satisfacer los requerimientos generados por el usuario.

Los componentes de software se dividen en capas lógicas. Cada una de estas define las responsabilidades en el proceso de resolución de un pedido del usuario. Las capas lógicas son: Cliente, Presentación, Lógica de Negocio, Integración y Recursos.

Por último, es importante conocer que la tecnología J2EE facilita el mantenimiento, la escalabilidad y la adaptabilidad de las aplicaciones. Sin embargo, el uso del Framework Enterprise JavaBeans como alternativa de persistencia de objetos, provoca la existencia de gran cantidad de códigos destinados a gestionar el ciclo de vida de una instancia EJB y hace que el *Servidor de Aplicaciones* invierta tiempo en operaciones de sincronización, entre las instancias en memoria y, las instancias en la base de datos.

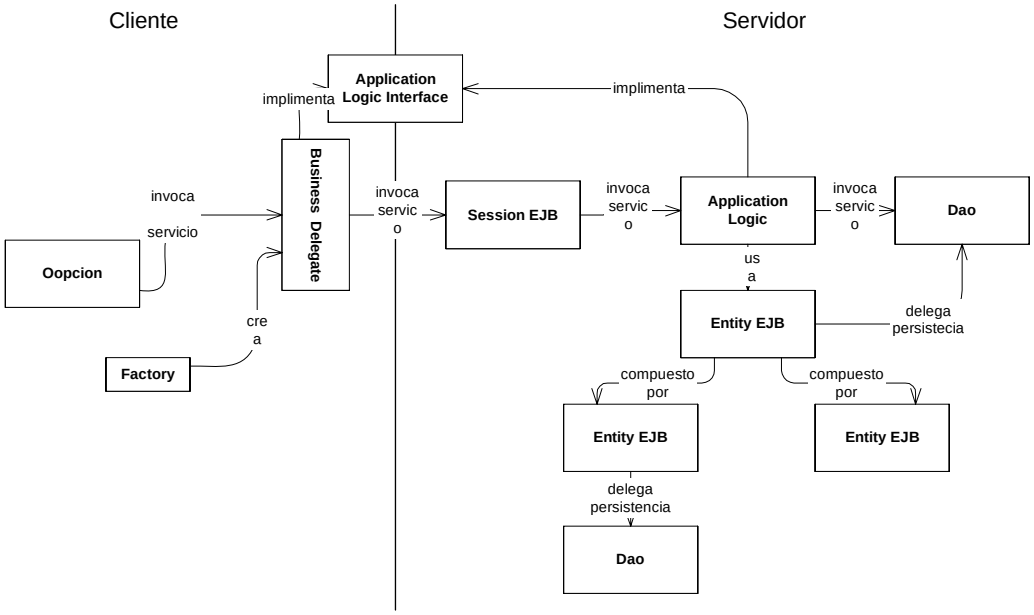

Figura 8 – *Arquitectura de software completa*

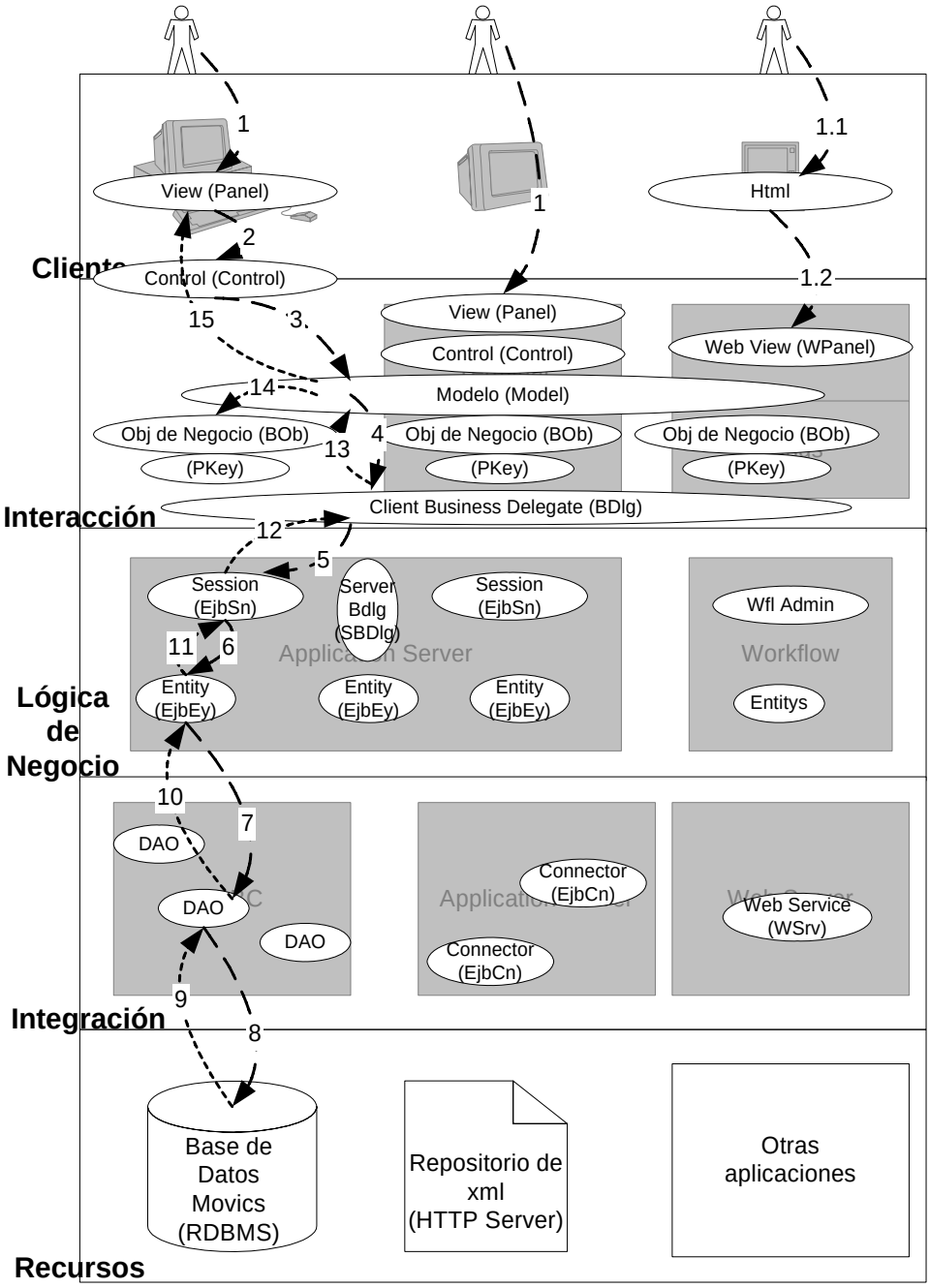

Figura 9 – *Interacción entre los componentes*

CAPÍTULO 5

Ejemplo de la arquitectura de software de la empresa

El presente capítulo muestra un ejemplo de diseño y de implementación, de una aplicación utilizando la arquitectura de software presentada anteriormente.

El objetivo es, que el lector obtenga un conocimiento práctico de los conceptos de diseño que la organización emplea, para resolver las cuestiones que afectan al desarrollo del software.

El dominio tomado como ejemplo, es una visión muy simplificada de un caso real, para facilitar la comprensión del lector sobre el uso de las metodologías de desarrollo de 'la empresa'.

Por último, es importante destacar, que sólo se detalla el diseño de la arquitectura que envuelve a la capa del lado del servidor. Esto se debe a que la aplicación del Framework JDO afecta a esta capa.

5.1 Dominio de ejemplo

La empresa en cuestión, desea automatizar las tareas requeridas para gestionar el seguimiento de los contratos que la compañía dispone, con las operadoras de servicios del mercado de telecomunicaciones.

Un contrato es un acuerdo comercial entre dos o más compañías de telecomunicaciones. Cada contrato tiene fecha de vigencia, información de los puntos de interconexión entre las

operadoras (pois), información de los prestadores, información de los servicios, información de los cargos e información de los planes.

La aplicación deberá guardar los datos referentes a los contratos. También, deberá registrar las relaciones entre estos acuerdos con los puntos de interconexión (pois), con los prestadores, con los servicios y con los planes asociados.

La implementación del ejemplo, corresponde a una parte de la funcionalidad entregada por el módulo Intercarriers de la compañía. Este módulo dispone de un conjunto de servicios destinados a manejar las relaciones entre las operadoras comerciales.

Las funciones a implementar son:

- *Generación del Contrato*: Este proceso genera un nuevo acuerdo entre la compañía y otras operadoras de telecomunicaciones.

- *Consulta del Contrato*: Esta función permite mostrar los datos referentes a los contratos vigentes entre la compañía y otras prestadoras de servicio.

5.2 Diseño

5.2.1 Diseño de objetos de negocio

Aquí se presenta el modelo de los objetos de negocio que corresponden al dominio problema planteado anteriormente.

Por convención de 'la empresa', todos los elementos que sean generados tendrán el prefijo Itc (por pertenecer al módulo Intercarriers). También, es importante saber que se establecen sufijos

de acuerdo al tipo de componente. Aquellos componentes que sean una Entidad tendrán el sufijo EjbEy, los que sean objetos dependientes tendrán el sufijo ODep y la capa de servicios estará representada por un componente con sufijo Appl.

El modelo de objetos de negocio contiene las siguientes entidades (Entitys): *Contrato*, *Pois*, *Prestadores*, *Servicos*, *Cargo* y *Plan*. Estas son nombradas como: *ItcContratoEjbEy*, *ItcPoisEjbEy*, *ItcPrestadorEjbEy*, *ItcServicioEjbEy*, *ItcCargoEjbEy* e *ItcPlanEjbEy*.

También, en el modelo se consideran las relaciones entre ellas. La entidad *Contrato* mantiene relaciones con cardinalidades múltiples a las entidades *Pois*, *Servicio*, *Cargo*, *Prestador* y *Plan*. Estas asociaciones son modeladas con clases *EntityCollection*.

Luego, se identifican los objetos dependientes y su tipo de relación. En el ejemplo, la entidad *Contrato* establece una asociación con sus datos a través del objeto dependiente *DatosContratos*. Esta asociación se modela como una relación con cardinalidad múltiple usando la clase *ODepCollection*.

Por último, los servicios que se le brindan al cliente serán publicados mediante un archivo xml llamado ItcIntercarriersAppl.xml. A partir de éste, se generan los componentes necesarios, para establecer la comunicación entre el cliente y la capa de servicios. La implementación de estos servicios se realizará en la clase ItcIntercarrierAppl, que representará al único *Application Logic* de la aplicación.

La siguiente figura representa las relaciones entre los objetos de negocio comentadas anteriormente.

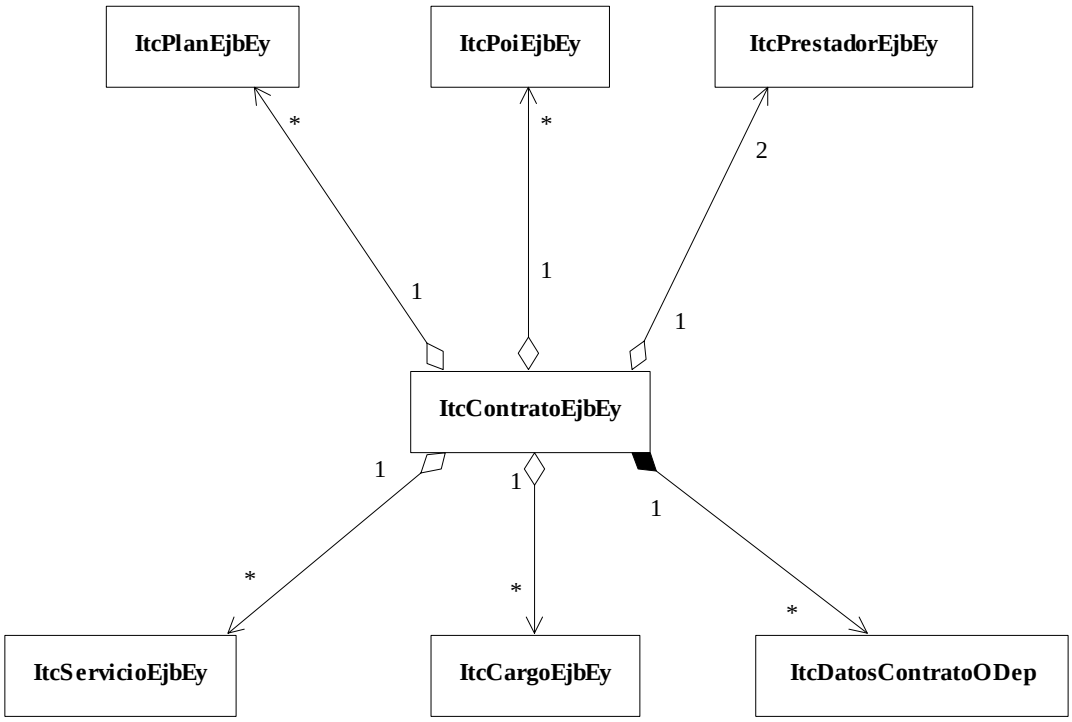

Figura 10 - *Relaciones entre los objetos de negocio*

5.2.2 Diseño de datos

Aquí se presenta el diseño de las tablas de la base de datos, que servirá de medio para persistir los objetos de negocio, que se modelaron anteriormente.

La entidad Plan será persistida en:

Columna	Tipo	Comentario
PLN_COD_PLAN	CHAR (6)	Clave
PLN_DESCRIPCION	VARCHAR2 (30)	
PLN_COD_TARIFA	NUMBER (9)	
PLN_COMENTARIO	VARCHAR2 (100)	
CRMTIMESTAMP	DATE	

La entidad Cargo será persistida en:

Columna	Tipo	Comentario
CAR_CARGO_ID	NUMBER (12)	Clave
CAR_CARGO_DESC_ID	NUMBER (12)	
CAR_MONTO	NUMBER (16,6)	
CAR_MONEDA	CHAR (6)	
CAR_PERIODICIDAD	CHAR (10)	
CAR_OBSERVACION	VARCHAR2 (100)	
CAR_FECHA_INICIO	DATE	
CAR_FECHA_VENCIMIENTO	DATE	
CAR_FECHA_MOD	DATE	
CAR_USR_MOD	VARCHAR2 (30)	
CAR_FECHA_ALTA	DATE	
CAR_USR_ALTA	VARCHAR2 (30)	
CRMTIMESTAMP	DATE	

La entidad Servicio será persistida en:

Columna	Tipo	Comentario
CSR_COM_SERVICIO_ID	NUMBER (12)	Clave
CSR_CODIGO	VARCHAR2 (6)	
CSR_NOMBRE	VARCHAR2 (50)	
CSR_TIPO	CHAR (1)	
CSR_DESCRIPCION	VARCHAR2 (1000)	
CSR_ESTADO	CHAR (1)	
CSR_FECHA_MOD	DATE	
CSR_USR_MOD	VARCHAR2 (30)	
CSR_FECHA_ALTA	DATE	
CSR_USR_ALTA	VARCHAR2 (30)	
CRMTIMESTAMP	DATE	

La entidad Prestador será persistida en:

Columna	Tipo	Comentario
PRE_PRESTADOR_ID	NUMBER	Clave
PRE_TIPO	VARCHAR2 (10)	
PRE_NOMBRE	VARCHAR2 (100)	
PRE_CODIGO	VARCHAR2 (50)	
PRE_ALIAS	VARCHAR2 (50)	
PRE_NEGOCIO	VARCHAR2 (50)	
PRE_REFERENTE	CHAR (1)	
PRE_FRANQUICIA	CHAR (1)	
PRE_PADRE	NUMBER	
PRE_OBSERVACION	VARCHAR2 (100)	
PRE_FECHA_MOD	DATE	
PRE_USR_MOD	VARCHAR2 (30)	

PRE_FECHA_ALTA	DATE	
PRE_USR_ALTA	VARCHAR2 (30)	
CRMTIMESTAMP	DATE	
PRE_CICLO	NUMBER (2)	
PRE_INX	CHAR (1)	
PRE_RMG	CHAR (1)	
PRE_CYO	CHAR (1)	
PRE_COD_ENTE_REGULADOR	NUMBER (2)	
PRE_COD_RMG	VARCHAR2 (6)	
PRE_CYO_SEQUENCE	NUMBER (10)	
PRE_PERSONA_ID	NUMBER (12)	
PRE_CICLO_CYO	NUMBER (2)	

La entidad Poi será persistida en:

Columna	Tipo	Comentario
POI_POI_ID	NUMBER (12)	Clave
POI_PRESTADOR_ID	NUMBER (12)	
POI_CODIGO	VARCHAR2 (12)	
POI_DESCRIPCION	VARCHAR2 (100)	
POI_FECHA_MOD	DATE	
POI_USR_MOD	VARCHAR2 (30)	
POI_FECHA_ALTA	DATE	
POI_USR_ALTA	VARCHAR2 (30)	
CRMTIMESTAMP	DATE	
POI_UBICACION_ID	NUMBER (12)	
POI_POI_ID_ANT	NUMBER (12)	
POI_PRESTADOR_ID_INX	NUMBER (12)	
POI_NEGOCIO	VARCHAR2 (100)	

La entidad Contrato será persistida en:

Columna	Tipo	Comentario
CON_CONTRATO_ID	NUMBER (12)	Clave
CON_CODIGO	VARCHAR2 (6)	
CON_FECHA_MOD	DATE	
CON_USR_MOD	VARCHAR2 (30)	
CON_FECHA_ALTA	DATE	
CON_USR_ALTA	VARCHAR2 (30)	
CRMTIMESTAMP	DATE	
CON_DESCRIPCION	VARCHAR2 (100)	
CON_LINEA_NEGOCIO	CHAR (1)	

El objeto dependiente DatosContratos será persistido en:

Columna	Tipo	Comentario
DAT_DATOS_CONTRATO_ID	NUMBER (12)	Clave

DAT_CONTRATO_ID	NUMBER (12)	Clave
DAT_FECHA_INICIO	DATE	
DAT_FECHA_VENCIMIENTO	DATE	
DAT_TIPO_LIQ	CHAR (10)	
DAT_PLAZO_LIQ	NUMBER (12)	
DAT_PLAN	CHAR (10)	
DAT_FECHA_MOD	DATE	
DAT_USR_MOD	VARCHAR2 (30)	
DAT_FECHA_ALTA	DATE	
DAT_USR_ALTA	VARCHAR2 (30)	
DAT_FECHA_DESDE	DATE	
DAT_FECHA_HASTA	DATE	
CRMTIMESTAMP	DATE	
DAT_OBSERVACION	VARCHAR2 (100)	
DAT_ESTADO	CHAR (1)	
DAT_VERSION	NUMBER (12)	
DAT_OPERACION	CHAR (1)	
DAT_CON_CODIGO	VARCHAR2 (6)	

Como se mencionó anteriormente, las entidades pueden mantener relaciones con otras entidades y con objetos dependientes. Estas asociaciones son modeladas en tablas.

La asociación entre Contrato y Plan será persistida en:

Columna	Tipo	Comentario
RCPL_CONTRATO_ID	NUMBER (12)	Clave
RCPL_PLAN_ID	CHAR (6)	Clave
RCPL_REL_ID	NUMBER (12)	
RCPL_DESCRIPCION	VARCHAR2 (100)	
RCPL_FECHA_DESDE	DATE	
RCPL_FECHA_HASTA	DATE	
RCPL_FECHA_MOD	DATE	
RCPL_USR_MOD	VARCHAR2 (30)	
RCPL_FECHA_ALTA	DATE	
RCPL_USR_ALTA	VARCHAR2 (30)	
RCPL_OPERACION	CHAR (1)	
CRMTIMESTAMP	DATE	

La asociación entre Contrato y Cargo será persistida en:

Columna	Tipo	Comentario
RCC_CARGO_ID	NUMBER (12)	Clave
RCC_CONTRATO_ID	NUMBER (12)	Clave
RCC_FECHA_DESDE	DATE	
RCC_FECHA_HASTA	DATE	

RCC_FECHA_MOD	DATE	
RCC_USR_MOD	VARCHAR2 (30)	
RCC_FECHA_ALTA	DATE	
RCC_USR_ALTA	VARCHAR2 (30)	
CRMTIMESTAMP	DATE	
RCC_REL_ID	NUMBER (12)	
RCC_DESCRIPCION	VARCHAR2 (100)	
RCC_OPERACION	CHAR (1)	

La asociación entre Contrato y Servicio será persistida en:

Columna	Tipo	Comentario
RCSC_REL_ID	NUMBER (12)	
RCSC_COM_SERVICIO_ID	NUMBER (12)	Clave
RCSC_CONTRATO_ID	NUMBER (12)	Clave
RCSC_EXP_SERVICIO_ID	NUMBER (12)	
RCSC_DESCRIPCION	VARCHAR2 (100)	
RCSC_FECHA_DESDE	DATE	
RCSC_FECHA_HASTA	DATE	
RCSC_FECHA_MOD	DATE	
RCSC_USR_MOD	VARCHAR2 (30)	
RCSC_FECHA_ALTA	DATE	
RCSC_USR_ALTA	VARCHAR2 (30)	
CRMTIMESTAMP	DATE	
RCSC_OPERACION	CHAR (1)	

La asociación entre Contrato y Prestador será persistida en:

Columna	Tipo	Comentario
RCP_CONTRATO_ID	NUMBER (12)	Clave
RCP_PRESTADOR_ID	NUMBER (12)	Clave
RCP_FECHA_DESDE	DATE	
RCP_FECHA_HASTA	DATE	
RCP_FECHA_MOD	DATE	
RCP_USR_MOD	VARCHAR2 (30)	
RCP_FECHA_ALTA	DATE	
RCP_USR_ALTA	VARCHAR2 (30)	
RCP_REL_ID	NUMBER (12)	
RCP_DESCRIPCION	VARCHAR2 (100)	
RCP_REFERENTE	CHAR (1)	
RCP_OPERACION	CHAR (1)	
RCP_CON_CODIGO	CHAR (6)	
CRMTIMESTAMP	DATE	

La asociación entre Contrato y Poi será persistida en:

Columna	Tipo	Comentario
RPC_POI_ID	NUMBER (12)	Clave
RPC_CONTRATO_ID	NUMBER (12)	Clave
RPC_FECHA_DESDE	DATE	
RPC_FECHA_HASTA	DATE	
RPC_FECHA_MOD	DATE	
RPC_USR_MOD	VARCHAR2 (30)	
RPC_FECHA_ALTA	DATE	
RPC_USR_ALTA	VARCHAR2 (30)	
RPC_REL_ID	NUMBER (12)	
RPC_DESCRIPCION	VARCHAR2 (100)	
RPC_OPERACION	CHAR (1)	
CRMTIMESTAMP	DATE	

5.3 Implementación

5.3.1 Implementación de objetos de negocio

Para implementar una entidad de negocio bajo la arquitectura de software de 'la empresa', es necesario escribir un archivo xml que la describa. Es decir, que detalle su estructura y sus relaciones con otras entidades.

Una vez generado el archivo xml, se ejecuta la herramienta de generación de código llamada *EntityBuilder*. Como resultado, el proceso construirá una serie de clases concretas y abstractas. El desarrollador debe extender e implementar las clases abstractas.

5.3.1.1 Archivos xml de definición de objetos de negocio

Los archivos xml que describen los objetos de negocio comienzan con el tag[12] <*Entity*>. Este tag posee los siguientes atributos que definen al objeto:

[12] En Html, los tags son códigos que determinan la estructura y la presentación de la información en un documento

- *Name*: Nombre de la entidad.
- *Module*: Nombre del módulo.
- *TableName*: Nombre de la tabla donde se persiste.
- *Tipo*: Tipo de Entidad. Los tipos son Entity u ODep.
- *Package*: paquete donde se almacenan las clases generadas.

Dentro del tag <*Entity*> aparecen varios tags <*Attribute*>. Estos representan los atributos que describen a la entidad. El tag <*Attribute*> posee una serie de atributos:

- *Name*: Nombre del atributo.
- *DbName*: Nombre del campo de la tabla al que se asocia.
- *JavaType*: Tipo de dato java del atributo.
- *SqlType*: Tipo de dato del campo relacionado con el atributo.
- *AccessType*: Tipo de acceso a sus valores.
- *Length*: Longitud del campo de la base de datos.
- *AcceptNull*: Indica si acepta valores null.

La representación de las relaciones también se define por medio de tags. La asociación entre una entidad y un conjunto de entidades se representa por medio del tag <*EntityCollection*>. Este tag está compuesto por una serie de atributos y de tags que lo describen.

Los atributos son:

- *Target*: Nombre de la entidad con la que se establece la relación.
- *Name*: Nombre del atributo de la entidad origen que hace referencia a la colección de entidades. Se considera entidad origen a la que se describe en el archivo xml.
- *RelationName*: Nombre de la relación.

- *TableName*: Nombre de la tabla donde se persiste la relación.

Los tag son:

- <*SourcePK*> Define la entidad usada como origen en la asociación.
- <*TargetPK*> Define la entidad usada como destino en la asociación.
- <*Attribute*> Atributos que describen la relación. Los elementos que lo componen ya fueron enunciados anteriormente.

Tanto el tag <*SourcePK*> como el tag <*TargetPK*> están compuestos por el tag <*JoinAttribute*> que modela la asociación. Este tag contiene los siguiente atributos:

  - *Name*: Nombre del atributo de la entidad que modela la relación.
  - *DbName*: Nombre del campo de la tabla al que se asocia el atributo que modela la relación.
  - *JavaType*: Tipo de dato Java asociado al atributo.
  - *SqlType*: Tipo de dato del campo relacionado con el atributo.
  - *Length*: Longitud del campo de la base de datos.
  - *JoinedAttr*: Nombre del atributo de la entidad origen o destino que se toma como base para la relación.

Dentro del tag <*EntityCollection*> se utilizan otros tags para definir los métodos utilizados para el manejo de colecciones.

Las relaciones entre una entidad y sus objetos dependientes se representan por medio del tag <*ODepCollection*>. Este tag tiene una serie de atributos que lo describen.

- *ODepName*: Nombre del objeto dependiente con el que se establece la relación.
- *Name*: Nombre del atributo de la entidad origen que hace referencia a la colección de objetos dependientes.

Dentro del tag <*ODepCollection*> se utilizan otros tags para definir los métodos utilizados para el manejo de colecciones.

Por último, todo objeto dependiente tiene un tag <*Owner*> que identifica a su dueño. Este tag se utiliza para administrar la relación. El tag tiene atributos y otros tags que lo describen:

Los atributos son:

- *Name*: Nombre de la entidad dueño del objeto dependiente.

Los tag son:

- <PKAttribute> Define los atributos para modelar la relación. Este tag está compuesto por:
    - *Name*: Nombre del atributo de la entidad dueño del objeto dependiente que es usado para la relación.

- *OdepAttrName*: Nombre del atributo del objeto dependiente usado para la relación.

Como ya se mencionó, este archivo se procesa con una herramienta que genera automáticamente un conjunto de clases que representan a los objetos de negocio y que administran la persistencia en la base de datos correspondiente.

Finalmente, a modo de ejemplo, el archivo xml de definición del objeto de negocio *Plan* queda conformado de la siguiente manera:

```
- <Entity name="Plan" module="ITC" tableName="CLL_PLANES"
    dbTimestampName="CRMTIMESTAMP" type="EyBMP"
    package="crm.appl.itc.contratos" generateDAO="T"
    generateEY="T">
   <Attribute name="codigo" dbName="PLN_COD_PLAN"
      javaType="String" sqlType="CHAR" accessType="rw"
      acceptNull="F" length="6" isPk="T" />
   <Attribute name="descripcion"
      dbName="PLN_DESCRIPCION" javaType="String"
      sqlType="VARCHAR" accessType="rw" acceptNull="T"
      length="30" isPk="F" />
   <Attribute name="codTarifa" dbName="PLN_COD_TARIFA"
      javaType="long" sqlType="NUMERIC" accessType="rw"
      acceptNull="T" length="9" isPk="F" />
   <Attribute name="comentario" dbName="PLN_COMENTARIO"
      javaType="String" sqlType="VARCHAR" accessType="rw"
      acceptNull="T" length="100" isPk="F" />
</Entity>
```

5.3.2 Generación de código

Anteriormente, se mencionó que muchos de los componentes que conforman la arquitectura de software se generan con herramientas automáticas.

Cada uno de estos generadores, toma como entrada archivos xml y a partir de estos genera un conjunto de clases concretas y abstractas. Estas últimas son implementadas por los desarrolladores para agregar las funciones necesarias del dominio.

Las clases generadas son: *ApplicationLogicInterface*, *Session*, *BusinessDelegate*, *Factory*, *ApplicationLogic*, *Entity*, *Dao*, *DaoSymbols*, *Rec*, *Primary Key*, *Relation Provider*, *WrapperNaw*, *Entity Home Abstracto* y *Entity Local Abstracto*.

5.3.3 Implementación de los servicios

La capa del servidor publica un conjunto de servicios que estarán disponibles para los clientes. Estos servicios se definen a través de un archivo xml, el cual sirve de base para la generación de componentes que establecen la comunicación entre el cliente y el servidor.

Luego de publicar los servicios se continúa con su implementación. La clase *ApplicationLogic* es el lugar donde los desarrolladores implementan la lógica necesaria para resolver los pedidos de los clientes.

5.3.3.1 Xml de servicios

El archivo xml de servicio se comienza a definir utilizando el tag <*appl*>. Este tag especifica las propiedades generales del componente *ApplicationLogic.*

Los atributos del tag <appl> son:

- *Name*: Nombre del componente *ApplicationLogic*.
- *Package*: Indica el paquete que contendrá las clases.
- *Module*: Nombre del módulo de la aplicación.
- *SessionType*: Indica si el componente Session Bean usado para acceder a los servicios es *Stateless* o *Stateful*.

Dentro del tag <*appl*> se utiliza el tag <*service*> para especificar cada servicio. En este tag se especifica el nombre y los parámetros del método que implementa el servicio.

Los atributos del tag <*service*> son:

- *Name*: Nombre del servicio.
- *Type*: Indica el tipo de servicio. Los valores son: *Query*, *Insert*, *Update*, *Delete* y *Process*.
- *Return*: Clase de retorno del método.
- *Transaction*: Define si el servicio se ejecuta en una transacción o no.

Dentro del tag <*service*> se utiliza el tag <*parameter type*> para especificar el parámetro de cada servicio. Los atributos del tag son:

- *Name*: Nombre del parámetro.
- *Type*: Indica el tipo de parámetro.

En el ejemplo se definen dos servicios: uno de consulta (*Query*) y el otro de inserción (*Insert*). Estos se publican en el xml de servicios y se implementan en la clase *ApplicationLogic.*

El servicio de consulta tiene el nombre de getContratoCObj (el prefijo get se agrega por la herramienta cuando se generan los componentes), recibe como parámetro un objeto ItcContratoPKey y retorna un objeto ItcContratoCObj.

El servicio de inserción tiene el nombre de addContrato (el prefijo add se agrega por la herramienta cuando se generan los componentes), recibe como parámetro un objeto ItcContratoCreatePar y no retorna nada.

Por último, el archivo xml de servicios queda conformado de la siguiente manera:

```
<appl name="Interconexion" package="crm.appl.itc.contratos"
     module="ITC" sessionType="Stateless">
  <service name="ContratoCObj" type="Query"
       return="ItcContratoCObj" transaction="Required">
    <parameter type="ItcContratoPKey" name="pPKey" />
  </service>
  <service name="Contrato" type="Insert" return="void"
       transaction="Required">
    <parameter type="ItcContratoCreatePar" name="pPar" />
  </service>
</appl>
```

5.3.3.2 Programación de los servicios

Aquí se presenta el código que implementa los servicios publicados anteriormente. También, se realiza una explicación de la implementación del método asociado a cada servicio y de las clases utilizadas.

5.3.3.2.1 Clase de la lógica del negocio (ApplicationLogic)

El código de la clase *ApplicationLogic* declara e inicializa las variables que almacenarán las interfaces de los componentes Enterprise JavaBeans usados.

La clase *ApplicationLogic* contendrá la implementación de los métodos que brindan las respuestas a las solicitudes realizadas por los clientes. Estos métodos fueron previamente publicados en el xml de servicios.

5.3.3.2.1.1 Consulta de contratos

Este servicio retorna los datos de un contrato dado. Esta función es implementada en el método getContratoCObj(), el cual recibe como parámetro una clave primaria que identifica al contrato.

El servicio utiliza el método ejbFindxxx() propuesto por el Framework Enterprise JavaBeans para la búsqueda de entidades en la base de datos.

A partir de una clave primaria del contrato se realiza una búsqueda para obtener la entidad del contrato buscado. A continuación, se encapsulan los datos de la entidad en un objeto serializable *CObj* que es enviado al cliente como respuesta a su pedido.

El código del servicio que implementa la consulta de los datos de un contrato en el *ApplicationLogic* es:

```
public ItcContratoCObj getContratoCObj(ItcContratoPKey pPKey)
   throws CrmException
 {

      ItcContratoCObj cobj = new ItcContratoCObj();

      ItcContratoEjbEyLc lc = null;

      if (pPKey == null)  return null;
      else
      {
           try
           {
             lc = _contratoHome.findByPrimaryKey(pPKey);
             cobj.setCodigo(lc.getCodigo());
             cobj.setLineaNegocio(lc.getLineaNegocio());
             cobj.setDescripcion(lc.getDescripcion());
             cobj.setFechaMod(lc.getFechaModificacion());
             cobj.setUsrMod(lc.getUsuarioModificacion());
             cobj.setFechaAlta(lc.getFechaAlta());
             cobj.setUsrAlta(lc.getUsuarioAlta());
           }
           catch (Exception e)
           {
                throw new CrmEJBNamingException(CLASS_NAME,
                    " getContratoCObj(ItcContratoPKey)", e);
           }
      }

      return cobj;
 }
```

5.3.3.2.1.2 Inserción de contratos

Este servicio permite agregar un nuevo contrato. Esta funcionalidad se implementa a través del método addContrato(), el cual recibe un objeto *Par* como parámetro, el cual contiene todos los datos del nuevo contrato.

El servicio utiliza el método ejbCreatexxx() propuesto por el Framework Enterprise JavaBeans para la creación de entidades en la base de datos.

A partir de los datos encapsulados en el objeto serializable *Par*, se crea la nueva entidad, se cargan los datos de las entidades asociadas al contrato, y finalmente, se realiza la persistencia del nuevo contrato en la base de datos.

El código del servicio que implementa la inserción de los datos de un contrato en el *ApplicationLogic* es:

```
public void addContrato (ItcContratoCreatePar pPar) throws CrmException
{

  try
  {

    /*- Cabecera del Contrato ------------------------------------------*/
    // Genero una nueva pk
    ItcContratoPKey pkContrato = new ItcContratoPKey();

    // Creo la cabecera del contrato
    ItcContratoEjbEyLc lcContrato = _contratoHome.create(
                                    pkContrato,
                                    pPar.getLineaNegocio(),
                                    pPar.getCodigo(),
                                    pPar.getFechaModificacion(),
                                    pPar.getUsrModificacion(),
                                    pPar.getFechaAlta(),
                                    pPar.getUsrAlta());

     if (pPar.getDescripcionCon() != null)
     lcContrato.setDescripcion(pPar.getDescripcionCon());

    /*- Genero el ID de Version comun a todas las relaciones ----------*/
    ItcNextValueSequencePar verSeq = new ItcNextValueSequencePar(
                            ItcSymbols.ITC_CONTRATO_VERSION_SEQ);
    long versionId =
            _validacionDao.getNextValueSequence(verSeq).getSecuencia().
```

```
            longValue();

        /*- Datos del Contrato ------------------------------------------------*/
        // Agrego los datos al ODep de Datos de Contrato
        ItcDatosContratoODep _datosContratoOdep = new
                                        ItcDatosContratoODep();
        ItcDatosContratoPKey pkDatos = new ItcDatosContratoPKey();

        _datosContratoOdep.setDatosId(pkDatos.getDatosId());
        _datosContratoOdep.setConId(pkContrato.getContratoId());
        _datosContratoOdep.setFechaInicio(pPar.getFechaInicio());
        _datosContratoOdep.setFechaVto(pPar.getFechaVencimiento());
        _datosContratoOdep.setTipoLiq(pPar.getTipoLiquidacion());
        _datosContratoOdep.setPlazoLiq(pPar.getPlazoLiquidacion());
        _datosContratoOdep.setPlan(pPar.getPlan());
        _datosContratoOdep.setFechaDesde(pPar.getFechaDesde());
        _datosContratoOdep.setFechaHasta(pPar.getFechaHasta());
        _datosContratoOdep.setObservacion(pPar.getObservacion());
        _datosContratoOdep.setVersion(versionId); // Version Id
        _datosContratoOdep.setOperacion(pPar.getOperacion());
        _datosContratoOdep.setFechaModificacion(pPar.getFechaModificacion());
        _datosContratoOdep.setUsuarioModificacion(pPar.getUsrModificacion());
        _datosContratoOdep.setFechaAlta(pPar.getFechaAlta());
        _datosContratoOdep.setUsuarioAlta(pPar.getUsrAlta());

        lcContrato.addDatosContrato(_datosContratoOdep);

        // Doy de alta la relacion de los prestadores
        /*- Referente del Contrato --------------------------------------------*/
        ItcPrestadorEjbEyLc lcReferente =
                    _prestadorHome.findByPrimaryKey(pPar.getPkReferente());

        lcContrato.addPrestador(lcReferente,
                        versionId,
                        pPar.getDescripcionRel(),
                        pPar.getOperacion(),
                        pPar.getFechaDesde(),
                        pPar.getFechaHasta(),
                        String.valueOf(ItcSymbols.ITC_INT_TRUE),
                        pPar.getFechaModificacion(),
                        pPar.getUsrModificacion(),
                        pPar.getFechaAlta(),
                        pPar.getUsrAlta());

        /*- Prestador del Contrato --------------------------------------------*/
        ItcPrestadorEjbEyLc lcPrestador =
                    _prestadorHome.findByPrimaryKey(pPar.getPkPrestador());

        lcContrato.addPrestador(lcPrestador,
                        versionId,
                        pPar.getDescripcionRel(),
                        pPar.getOperacion(),
                        pPar.getFechaDesde(),
                        pPar.getFechaHasta(),
                        String.valueOf(ItcSymbols.ITC_INT_FALSE),
                        pPar.getFechaModificacion(),
                        pPar.getUsrModificacion(),
                        pPar.getFechaAlta(),
                        pPar.getUsrAlta());

        /*- Pois del Contrato -------------------------------------------------*/
        // Obtengo los pois asociados y doy de alta la relacion
        // Obtengo los pois asociados y doy de alta/actualizo la cuenta asociada
        Iterator colPois = pPar.getPois().iterator();
        while (colPois.hasNext())
        {

           ItcPoiCObj cobjPoi = (ItcPoiCObj)colPois.next();
```

```
    ItcPoiEjbEyLc lcPoi =
          _poiHome.findByPrimaryKey(new
                      ItcPoiPKey(cobjPoi.getPoiId()));

    lcContrato.addPoi(lcPoi,
                versionId,
                pPar.getDescripcionRel(),
                pPar.getOperacion(),
                cobjPoi.getFechaDesde(),
                cobjPoi.getFechaHasta(),
                pPar.getFechaModificacion(),
                pPar.getUsrModificacion(),
                pPar.getFechaAlta(),
                pPar.getUsrAlta());

     try
     {

       ItcCuentaEjbEyLc lcCta = _cuentaHome.findByPrimaryKey(
                          new ItcCuentaPKey(cobjPoi.getPoiId()));
       // Actualizo
       lcCta.setNroCliente(lcReferente.getPrestadorID());
       lcCta.setCiclo(lcReferente.getCiclo());
       lcCta.setCicloAnterior(lcReferente.getCiclo());
       lcCta.setCicloProximo(lcReferente.getCiclo());
       lcCta.setTipoCuenta(ItcSymbols.ITC_TIPO_CUENTA);
       lcCta.setEstado(ItcSymbols.ITC_ESTADO_CUENTA);

     }
     catch(FinderException ex)
     {

       // Doy de alta
       ItcCuentaEjbEyLc lcCta =
                _cuentaHome.create(new ItcCuentaPKey(
                                        cobjPoi.getPoiId()),
                             lcReferente.getCiclo(),
                             lcReferente.getCiclo(),
                             lcReferente.getCiclo(),
                             ItcSymbols.ITC_TIPO_CUENTA,
                             ItcSymbols.ITC_ESTADO_CUENTA);

       lcCta.setNroCliente(lcReferente.getPrestadorID());
     }

}


/*- Servicios del Contrato ------------------------------------------*/
// Obtengo los servicios asociados y doy de alta la relacion
Iterator colSrvs = pPar.getServicios().iterator();
while (colSrvs.hasNext())
{

    ItcComServicioCObj cobjSrv = (ItcComServicioCObj)colSrvs.next();
    ItcComServicioPKey pkSrv =
                      new ItcComServicioPKey(cobjSrv.getId());
    ItcComServicioEjbEyLc
              lcComServicio = _conservicioHome.findByPrimaryKey(pkSrv);

    // Obtengo y doy de alta las expresiones asociadas al servicio
    Iterator colExps =
              _conservicioDao.getExpresionesContrato(pkSrv).iterator();
    while (colExps.hasNext())
    {

       ItcExpresionesContratoCObj cobjExps =
                       (ItcExpresionesContratoCObj)colExps.next();
```

```
                lcContrato.addComServicio(lcComServicio,
                                versionId,
                                cobjExps.getExpresionId(),
                                pPar.getDescripcionRel(),
                                pPar.getOperacion(),
                                cobjSrv.getFechaDesde(),
                                cobjSrv.getFechaHasta(),
                                pPar.getFechaModificacion(),
                                pPar.getUsrModificacion(),
                                pPar.getFechaAlta(),
                                pPar.getUsrAlta());
                    }

                }

                /*- Planes del Contrato ------------------------------------------------*/
                // Obtengo los planes asociados y doy de alta la relacion
                Iterator colPlanes = pPar.getPlanes().iterator();
                while (colPlanes.hasNext())
                {
                   ItcPlanCObj cobjPlan = (ItcPlanCObj)colPlanes.next();
                   ItcPlanEjbEyLc lcPlan = _planHome.findByPrimaryKey(
                                    new ItcPlanPKey(cobjPlan.getCodigo()));
                   lcContrato.addPlan(lcPlan,
                                versionId,
                                pPar.getDescripcionRel(),
                                pPar.getOperacion(),
                                cobjPlan.getFechaDesde(),
                                cobjPlan.getFechaHasta(),
                                pPar.getFechaModificacion(),
                                pPar.getUsrModificacion(),
                                pPar.getFechaAlta(),
                                pPar.getUsrAlta());

                }

                CrmLog.printVerbose("=>ItcInterconexionAppl.addContrato(pPar) - " +
                            "Contrato creado");

            }
            catch(Exception e)
            {
                throw new CrmEJBNamingException( CLASS_NAME,
                    " addContrato(ItcContratoCreatePar),
                    e);
            }
        }
```

5.4 Conclusión

En este capítulo se presentó toda la información necesaria para que el lector comprenda, por medio de un ejemplo sencillo, la metodología utilizada por 'la empresa' para el desarrollo de sus productos de software.

La definición de los objetos de negocio se plasma a través de un conjunto de archivos xml de diseño propio, los cuales son utilizados

por las herramientas de generación automática de código. Estas herramientas a partir de clases previamente construidas y junto a los archivos xml, generan los componentes que permiten modelar las entidades del dominio.

Los servicios brindados por el servidor se publican en archivos xml propios, los cuales sirven de entrada a los procesos que generan los componentes que brindan la comunicación entre el cliente y el servidor.

La fase final de la construcción permite agrupar el conjunto de clases generadas y construidas por los desarrolladores en archivos para su fácil instalación.

CAPÍTULO 6

Nueva arquitectura de software

En este capítulo se detalla la nueva arquitectura de software que, empleando el Framework JDO, permitirá la aplicación del modelo orientado a objetos sobre la capa de persistencia.

Se presentan los diversos componentes de software que se utilizan para modelar las entidades de negocio y las relaciones entre ellas. También, se explican los elementos de la arquitectura que brindan los servicios que satisfacen las necesidades funcionales de los clientes.

Recordando que 'la empresa' emplea un sistema distribuido basado en la especificación J2EE, la nueva arquitectura de software hace uso de patrones de diseño recomendados para desarrollos de aplicaciones distribuidas. Además, se apoya en el uso conjunto de un *Servidor de Aplicaciones* y de una implementación JDO, logrando la integración de los servicios del back-end[13] con la persistencia transparente de los objetos de negocio.

Por último, es importante destacar que, solo se abordan temas de la arquitectura que envuelven a la capa del servidor. Esto se debe a que la implementación del Framework JDO afecta a la lógica funcional brindada por el servidor, haciendo a la capa del cliente totalmente transparente a la nueva arquitectura de desarrollo.

[13] El back-end es la parte de un sistema de software que procesa las solicitudes de los clientes.

6.1 Capas lógicas

Como se mencionó anteriormente, el modelo de desarrollo de software hace uso de varias capas lógicas con responsabilidades claramente definidas. Estas capas, como también sus responsabilidades, se continuarán manteniendo en la nueva arquitectura de desarrollo de software.

En la siguiente figura se presentan los principales componentes empleados por cada capa lógica con objeto de implementar sus responsabilidades.

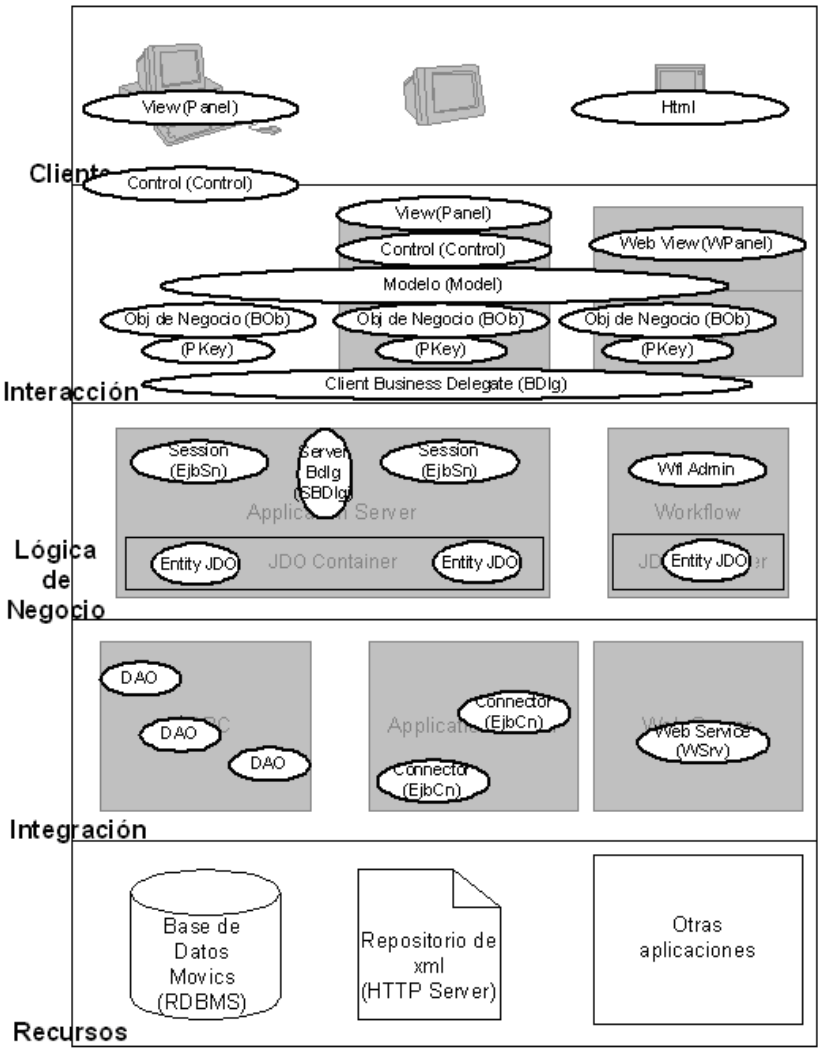

Figura 11 – *Componentes por capa lógica*

Si bien se mantienen las responsabilidades de las capas lógicas en la nueva arquitectura, es importante observar que aparece un nuevo componente en la capa de Negocio, la *entidad JDO*. Este nuevo componente de software tiene como objetivo, reemplazar la funcionalidad del bean de entidad, de la arquitectura anterior.

6.2 Mecanismo de persistencia de las instancias

La implementación de los servicios brindados por los productos de software, está basada en la utilización conjunta de la arquitectura distribuida J2EE con el Framework JDO como mecanismo de persistencia de los objetos del dominio.

El uso del Framework JDO, brinda la persistencia directa del modelo de objetos del dominio, haciendo totalmente transparente el proceso de almacenamiento de las instancias sobre una base de datos [2]. Así, JDO, evita que los desarrolladores escriban códigos de persistencia para acceder a los datos.

Las clases Java, que representan tanto el modelo de datos como sus atributos, pueden ser persistidas directamente por el Framework JDO [2]. Es decir, no es necesario contar con lógica relacionada para el almacenamiento de las instancias sobre la base de datos.

Las relaciones entre las instancias son representadas usando una referencia, cuando la asociación tiene cardinalidad uno; o una colección de objetos, cuando la asociación tiene cardinalidad múltiple [2].

Anteriormente, se expresó que el modelo de datos a ser persistido se especifica por medio de un conjunto de clases Java. Adicionalmente a esto, es necesario brindar al Framework JDO

información relacionada con el almacenamiento de los objetos. Esta información no se expresa directamente en la clase Java sino a través de un archivo xml, llamado *Descriptor de Persistencia*, que describe todo lo necesario para la persistencia de las instancias [14].

Por último, el Framework JDO utiliza el proceso de ´enhanced´ con el objeto de agregar comportamiento persistente a una clase. Este proceso toma el modelo de clases Java más el *Descriptor de Persistencia* y, como resultado, produce nuevas clases con todo lo necesario para que puedan ser tratadas por el ambiente JDO [3].

6.3 Objetos de negocio y su representación

Como se mencionó, los objetos de negocio representan a las entidades del dominio del negocio, que se quieren modelar. Estas entidades establecen asociaciones entre sí.

En la nueva arquitectura, las entidades de negocio que serán persistidas no requieren lógica adicional, es decir, son clases Java comunes que hacen referencia a otras clases o a colecciones de clases.

Las asociaciones del tipo *todo-parte*, donde la existencia de un objeto que representa la parte pierde sentido al eliminarse el objeto que representa el todo, no requieren de un código especial que discrimine los objetos involucrados en la relación. En este caso, solamente se debe especificar en el *Descriptor de Persistencia*, que el objeto que corresponde a la parte, está incluido en el objeto que corresponde al todo.

Cuando existen relaciones entre las entidades con cardinalidad múltiple, se utiliza el Framework de colecciones de Java e información adicional, en el *Descriptor de Persistencia,* para representar tales asociaciones. En cambio, si la relación con cardinalidad múltiple tiene

un conjunto de atributos que la describe, hay que sumar a lo anterior la creación de una entidad falsa que agrupe tales características.

Por último, las relaciones de una entidad con otra, se modelan usando referencias simples entre las clases e información adicional en el *Descriptor de Persistencia*.

6.4 Jerarquía de objetos de negocio

Anteriormente, se expresó que los objetos que modelan las entidades de negocio, no necesitan de ninguna lógica especial para lograr persistirse en un medio de almacenamiento. Estos son representados por clases comunes de Java.

En este esquema, no existe ninguna jerarquía de clases creada con el objeto de alcanzar la persistencia de las instancias, como ocurre en la arquitectura de 'la empresa'. Aquellas jerarquías concebidas, están íntegramente relacionadas con la lógica propia del dominio del problema.

Por último, cabe destacar que el Framework JDO, es el responsable de implementar toda la lógica necesaria, para que las instancias que desean persistirse puedan ser almacenadas. Esta característica hace que el desarrollador de la aplicación se enfoque en el problema del dominio y no en como persistir las instancias.

6.5 Componentes de la arquitectura de software

La nueva arquitectura, utiliza patrones de diseño, para generar varios componentes de software, que se localizan en las capas del modelo de desarrollo, con el objetivo de resolver las solicitudes de los clientes.

El patrón *Facade*, componente bean de sesión, encapsula la complejidad de las interacciones y proporciona un servicio de acceso uniforme a los clientes [11]. Este componente delega en el componente *ApplicationLogic* la implementación de sus servicios.

El elemento *ApplicationLogic* implementa los servicios solicitados por el cliente. Este componente se construye por el desarrollador de la aplicación del negocio.

Otro patrón utilizado en la arquitectura es el *Business Delegate*, que reside en la capa de presentación y, en beneficio de los componentes del cliente, llama a los métodos remotos, publicados en el bean de sesión, que actúa como medio de acceso a la aplicación, que esta corriendo en el servidor [11].

También se emplean los patrones *Data Transfer Object* y *Data Access Object (Dao).* El primero representa objetos serializables para la transferencia de datos sobre la red [11] y el segundo consiste en un grupo de entidades, que encapsulan todos los accesos a la fuente de datos [15].

La capa de diseño DAO, contiene las responsabilidades que permiten el acceso a los datos. La implementación de los servicios brindados por esta capa, está basada en el uso de componentes propios del Framework JDO, junto al lenguaje de consulta, definido para tal función dentro de su arquitectura. También, es posible el uso de sentencias SQL para el acceso a los datos como alternativa a las capacidades brindadas por la arquitectura JDO, siendo esta opción no recomendable.

Aquellos componentes que forman parte del proceso de comunicación entre el cliente y el servidor continúan siendo generados por procesos automáticos. Es decir, el componente

ApplicationLogicInterface, el componente *Session*, el componente *BusinessDelegate* y el componente *Factory* son generados por la herramienta de generación de código llamada *Stubber*.

Sin embargo, aquellos componentes que representan los objetos de negocio que serán persistidos, no se generarán a partir de la herramienta llamada *EntityBuilder* como ocurre en la arquitectura de 'la empresa'.

En este esquema, aparece una nueva herramienta cuyo objetivo consiste en crear las instancias que permiten el acceso a los datos (DAO), construir las claves primarias correspondiente a las entidades y realizar el proceso de 'enhanced' a las clases persistentes.

Por último, se utiliza el patrón S*erviceLocator* para obtener el acceso a las capacidades del Framework JDO.

6.6 Solicitudes de la capa del cliente

El cliente recibe los pedidos de los usuarios y, en respuesta, realiza demandas a la capa de servicios a través de los componentes *ApplicationLogicInterface, Session, BusinessDelegate* y *Factory*. Todos estos componentes encapsulan la lógica de comunicación haciendo que las solicitudes del cliente sean totalmente transparentes.

La comunicación con el servidor, se realiza a través de la invocación remota de los métodos publicados en el bean de sesión, que actúa como *Facade*, encapsulando la complejidad de las interacciones [11].

El cliente puede enviar al servidor parámetros que eventualmente necesitan sus servicios y el servidor puede devolver información como resultado del procesamiento de un pedido.

En la arquitectura, existen dos componentes, cuya responsabilidad es el intercambio de información entre la capa del cliente y la capa del servidor. Ambos componentes modelan el patrón *Data Transfer Object* para lograr la transferencia de datos sobre la red [11]. Los componentes que permiten el pasaje de información son el *CObj* y el *Par*.

Dentro de las solicitudes, que puede realizar un cliente, tenemos aquellas que involucran a una entidad y aquellas que involucran a un conjunto de entidades. En la arquitectura, se estableció, que cualquier pedido que afecta a una entidad debe ser resuelto por el *ApplicationLogic* y por la propia entidad la cual, es representada por una *instancia JDO*, mientras que, si el pedido involucra a una lista de entidades, debe ser tratado directamente por el componente *Dao*. Es decir, la solicitud es atendida por el *ApplicationLogic* que delega la responsabilidad de resolver el pedido en el componente *Dao*, el cual, por medio del lenguaje ofrecido por la arquitectura JDO accede a los datos solicitados haciéndolos serializables para que puedan viajar hasta el cliente.

Por último, la Figura 12 muestra la arquitectura de desarrollo que se emplea para la construcción de todos los productos de software.

6.7 Interacción entre los componentes

En la Figura 13 se muestra la secuencia de mensajes intercambiados entre los componentes de las capas lógicas de la nueva arquitectura de software. Estos mensajes son el resultado de una solicitud iniciada por el cliente.

El cliente inicia un pedido que es atendido por la capa de presentación. Esta, encapsula la solicitud (componente Par) y la envía a la capa de negocio por medio del componente *Busines Delegate*.

El *Busines Delegate* al recibir una solicitud realiza llamadas a métodos remotos, publicados en la interfaz del componente bean de sesión que actúa como *Facade*. Al llegar el pedido a la capa de negocio, es tratado por el *ApplicationLogic,* que determina si se resuelve como una solicitud a una entidad de negocio modelada con una *instancia JDO*, o como una solicitud directa al componente *Dao*. El resultado es encapsulado y retornado hacia el cliente.

Por último, al llegar la respuesta del pedido a la capa de presentación es procesada y finalmente mostrada al usuario.

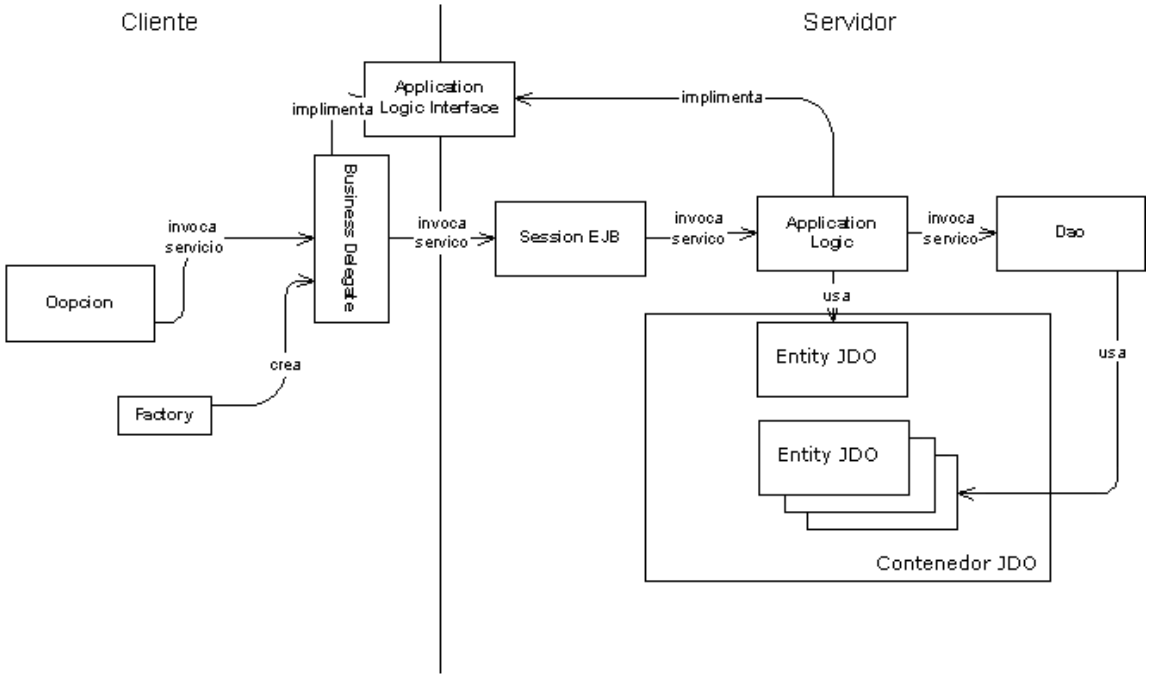

Figura 12 – *Nueva arquitectura de software*

6.8 Conclusión

La nueva arquitectura de software, utiliza la tecnología distribuida J2EE y el Framework JDO, para el desarrollo de los productos de software. El empleo de los servicios ofrecidos por JDO, permite que las entidades de negocio sean modeladas con clases comunes Java, sin necesidad de contar con lógica relacionada con el almacenamiento de las instancias sobre la base de datos. Como resultado, las aplicaciones recorren el modelo de objetos en memoria y es el Framework JDO quien brinda a las instancias la capacidad de persistirse.

En este esquema, el modelo de desarrollo utiliza patrones de diseño recomendados, para definir un conjunto de componentes, de relaciones y de interacciones con el fin de satisfacer los requerimientos generados por el usuario.

Si bien se mantienen las responsabilidades de las capas lógicas en la nueva arquitectura, es importante observar que aparece un nuevo componente en la capa de Negocio, la *entidad JDO*. Este nuevo componente de software tiene como objetivo reemplazar la funcionalidad del bean de entidad de la arquitectura anterior.

Por último, es importante destacar que si bien el uso de la tecnología distribuida facilita el mantenimiento, la escalabilidad y la adaptabilidad de las aplicaciones, es el Framework JDO quien permite una mejora en la distribución de los recursos, haciendo que los desarrolladores concentren sus esfuerzos en los problemas del dominio del negocio.

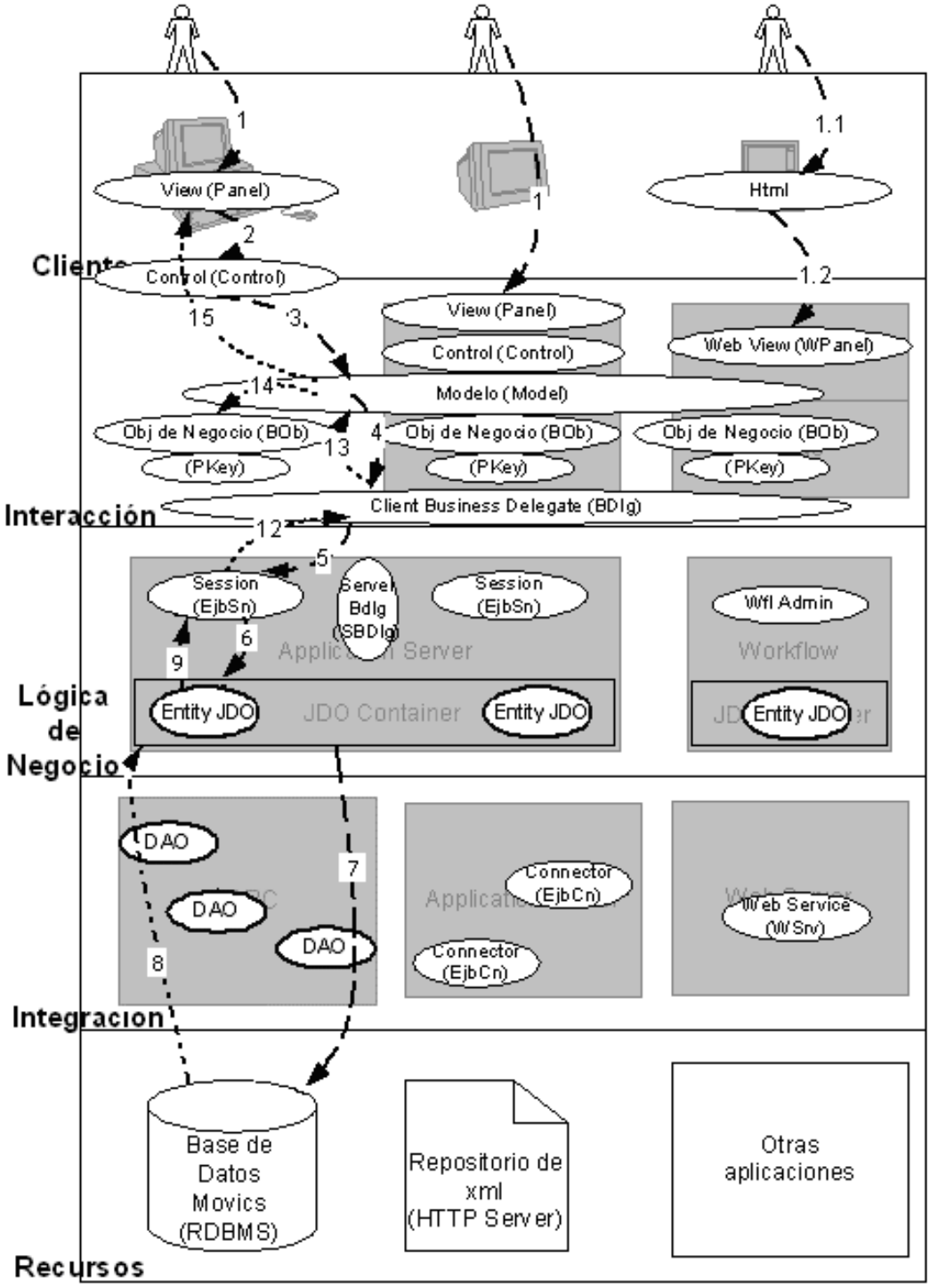

Figura 13 – *Secuencia de mensajes entre componentes*

CAPÍTULO 7

Ejemplo de la nueva arquitectura de software

El presente capítulo brinda un ejemplo del empleo de la arquitectura de software presentada anteriormente.

A continuación se describen los conocimientos necesarios, para que el lector pueda utilizar la nueva arquitectura de desarrollo, en las actividades de diseño y de implementación propias, del proceso de construcción de la aplicación de software.

El dominio que se emplea para la presentación del ejemplo es semejante al utilizado en el *Capítulo 5 - Ejemplo de la arquitectura de software de la empresa*. Esta semejanza, permite identificar de manera práctica, donde impacta la tecnología JDO, en la arquitectura de desarrollo utilizada en la empresa de telecomunicaciones.

Por último, los temas aquí tratados, solo afectan a la lógica correspondiente a la capa del servidor.

7.1 Dominio de ejemplo

Como fue mencionado, el dominio del negocio es similar al presentado en el ejemplo del *Capítulo 5 - Ejemplo de la arquitectura de software de la empresa*.

La concordancia de los dominios, tiene como finalidad, permitir al lector apreciar las diferencias, entre el uso de la arquitectura de desarrollo de 'la empresa' y el empleo de la nueva arquitectura de desarrollo que integra JDO, como mecanismo de persistencia de los objetos dentro de la plataforma distribuida J2EE.

7.2 Diseño

7.2.1 Diseño de objetos de negocio

El diseño de las entidades de negocio, así como las relaciones que permiten modelar el dominio del problema, se presentan en el apartado 5.2.1 del *Capítulo 5 - Ejemplo de la arquitectura de software de la empresa*.

Como fue expresado, por convención de 'la empresa', todos los elementos que sean generados tendrán el prefijo Itc (por pertenecer al módulo Intercarriers). También se conservan, salvo algunos casos, los sufijos de acuerdo al tipo de componente.

El modelo de objetos de negocio mantiene las siguientes entidades: *Contrato*, *Pois*, *Prestadores*, *Servicos*, *Cargo* y *Plan*, las cuales son nombradas como: *ItcContratoEy*, *ItcPoiEy*, *ItcPrestadorEy*, *ItcServicioEy*, *ItcCargoEy* e *ItcPlanEy*.

Las relaciones entre las entidades de negocio serán las mismas que las establecidas previamente, cambiando la manera en que son modeladas. La entidad *Contrato* mantiene relaciones con cardinalidades múltiples a las entidades *Pois*, *Servicio*, *Cargo*, *Prestador* y *Plan*. Estas asociaciones están representadas por el Framework de colecciones de Java y a través de información adicional en el *Descriptor de Persistencia* de JDO.

Las entidades que fueron modeladas como objetos dependientes en la arquitectura anterior serán, en la nueva arquitectura, modeladas con objetos comunes Java y representadas en el *Descriptor de Persistencia* como objetos embebidos, estableciendo una relación con la instancia que la contiene. Si esta asociación es una relación simple, se modela usando referencias. En cambio, si es una

relación múltiple, se representa con el Framework de colecciones de Java y se describe tal unión en el *Descriptor de Persistencia* de JDO.

Por último, si la relación entre las entidades está descripta por un conjunto de atributos, hay que crear una entidad falsa que agrupe tales características. Esto ocurre en las relaciones entre la entidad *Contrato* con *Plan*, *Cargo*, *Poi*, *Prestador* y *Servicio*. Estas asociaciones son representadas con las siguientes entidades falsas: *ItcContratoCargoEy*, *ItcContratoPoiEy*, *ItcContratoPrestadorEy* y *ItcContratoServicioEy*.

7.2.2 Diseño de datos

El diseño de las tablas de la base de datos que servirá de medio para persistir los objetos de negocio, será el mismo que el establecido en el apartado 5.2.2 del *Capítulo 5 - Ejemplo de la arquitectura de software de la empresa*, sin realizar cambios en la estructura de almacenamiento.

7.3 Implementación

7.3.1 Implementación de objetos de negocio

Para implementar una entidad de negocio bajo la nueva arquitectura de software, es necesario codificarla como una clase común Java y crear el *Descriptor de Persistencia* que la defina, es decir, que detalle su estructura y sus relaciones con otras entidades.

Una vez generado el *Descriptor de Persistencia,* se ejecuta la herramienta de enhanced provista por el vendedor JDO, con el objeto de agregar el comportamiento necesario para el almacenamiento de las instancias en el medio de persistencia. Este proceso utiliza el

modelo de clases Java y el *Descriptor de Persistencia* para producir un conjunto de objetos capaces de ser empleados por el Framework JDO.

Este trabajo se acompaña de dos apéndices, con el propósito de permitir al desarrollador, definir un esquema de clases persistentes que pueda ser tratado por el ambiente JDO. En el *Apéndice A - Mapeo relacional* se describen los mecanismos de mapeo para persistir los objetos del dominio en el esquema relacional y, en el *Apéndice B – Estructura del Descriptor de Persistencia* se detalla el contenido del archivo xml utilizado por JDO para el enriquecimiento de las clases.

Finalmente, a modo de ejemplo, el *Descriptor de Persistencia* que define el objeto de negocio *Plan* queda conformado de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo
   PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata
       2.0//EN" "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
   <package name="jdo">

     <class name="ItcPlanEy" table="PLAN" identity-type="application"
       objectid-class="jdo.pk.ItcPlanPK">
       <field name="codPlan" primary-key="true"
              persistence-modifier="persistent">
                     <column name="PLN_COD_PLAN"
                            jdbc-type="VARCHAR" length="6"
                                   allows-null="false"/>
       </field>
       <field name="descripcion" persistence-modifier="persistent">
                     <column name="PLN_DESCRIPCION"
                            jdbc-type="VARCHAR" length="30"
                                   allows-null="false"/>
       </field>
       <field name="codTarifa" persistence-modifier="persistent">
                     <column name="PLN_COD_TARIFA"
                            jdbc-type="NUMERIC"
                                   allows-null="true"/>
       </field>
       <field name="comentario" persistence-modifier="persistent">
                     <column name="PLN_COMENTARIO"
                            jdbc-type="VARCHAR" length="100"
                                   allows-null="true"/>
```

```
        </field>
    </class>
  </package>
</jdo>
```

7.3.2 Generación de código

Como se mencionó, muchos de los componentes que forman la arquitectura de software son generados a partir de herramientas de generación automática de código.

En la nueva arquitectura cada uno de los elementos que participa de los procesos de comunicación entre el cliente y el servidor, continuarán siendo generados por la herramienta de generación automática *Stubber,* propia de 'la empresa' en cuentión.

Las clases generadas por la herramienta *Stubber* son: *ApplicationLogicInterface*, *Session*, *BusinessDelegate*, *Factory*, *ApplicationLogic* y *WrapperNaw*.

Por último, es necesario contar con una nueva herramienta cuyo objetivo es, construir las clases que corresponden a las claves primarias de las entidades, crear los componentes que permiten el acceso a los datos (DAO) y realizar el proceso de 'enhanced' a las clases persistentes.

7.3.3 Implementación de los servicios

Las actividades que permiten formalizar los servicios que serán brindados a los clientes, continuarán siendo las mismas que las utilizadas por 'la empresa' en sus procesos de desarrollo. Es decir, los servicios son publicados en un archivo xml de definición de servicios, realizando la implementación de los mismos sobre el componente *ApplicationLogic*, el cual es construido por el desarrollador.

Esta fase, presenta diferencias al momento de realizar las implementaciones de las operaciones, que utilizan las instancias persistentes. Estas discrepancias están dadas por el uso de la arquitectura JDO, en contraposición al Framework Enterprise JavaBeans, como medio de acceso a los datos almacenados.

7.3.3.1 Programación de los servicios

Aquí, se presenta el código que implementa los servicios definidos en el apartado 5.3.3.2 del *Capítulo 5 - Ejemplo de la arquitectura de software de la empresa*. Este código, utiliza los recursos entregados por el Framework JDO para acceder a los objetos persistentes.

7.3.3.1.1 Clase de la lógica del negocio (ApplicationLogic)

La clase *ApplicationLogic*, obtiene la referencia de la *Fábrica de Persistencia* para acceder a los servicios brindados por el Framework JDO. También, contiene la implementación de los servicios solicitados por los clientes.

7.3.3.1.1.1 Consulta de contratos

A partir de una clave primaria del contrato, se realiza una consulta para obtener una instancia del contrato. El proceso de búsqueda, emplea los mecanismos ofrecidos por la arquitectura JDO para hallar el objeto correspondiente. Como resultado, se encapsulan los datos en un componente serializable que es enviado luego al cliente.

El código que obtiene el contrato desde la base de datos es:

```
public ItcContratoCObj getContratoCObj(ItcContratoPK pPk) {
```

```
        ItcContratoCObj cobj = null;

        if(pmf != null)
        {
                PersistenceManager pm =
                        pmf.getPersistenceManager();

                ItcContratoEy con = (ItcContratoEy)
                                pm.getObjectById(pPk);

                cobj = new ItcContratoCObj();

                cobj.setPk(pPk);
                cobj.setCodigo(con.getCodigo());
                cobj.setLineaNegocio(con.getLineaNegocio());
                cobj.setDescripcion(con.getDescripcion());
                cobj.setFechaMod(con.getFechaMod());
                cobj.setFechaAlta(con.getFechaAlta());
                cobj.setUsrAlta(con.getUsrAlta());
                cobj.setUsrMod(con.getUsrMod());

                pm.close();
        }

        return cobj;
}
```

7.3.3.1.1.2 Inserción de contratos

A partir de los datos del contrato encapsulados en el objeto serializable se crea la nueva entidad y se cargan los datos de las entidades asociadas al contrato. Finalmente, se utilizan los servicios del Framework JDO para persistir los datos de la nueva instancia en el medio de almacenamiento.

El código que crea el nuevo contrato en la base de datos es:

```
public void addContrato(ItcContratoCreatePar par) {

        if(pmf != null)
        {
              PersistenceManager pm = pmf.getPersistenceManager();

//Cabecera del contrato ----------------------------------------------------
-------
                ItcContratoEy con = new
                ItcContratoEy(par.getCodigo(),
                par.getLineaNegocio(),
                par.getFechaAlta(), par.getFechaMod(),
                par.getUsrAlta(), par.getUsrMod());
                con.setDescripcion(par.getDescripcion());


//Datos del contrato -------------------------------------------------------
------
```

```
                    ItcDatosContratoEy conODep = new
                    ItcDatosContratoEy(par.getFechaInicio(),
                    par.getFechaVencimiento());

                    conODep.setTipoLiq(par.getTipoLiq());
                    conODep.setPlazoLiq(par.getPlazoLiq());
                    conODep.setPlan(par.getPlan());

                    conODep.setObservacion(par.getObservacion());
                    conODep.setOperacion(par.getOperacion());
                    conODep.setVersion(par.getVersion());

                    conODep.setFechaDesde(par.getFechaDesde());

                    conODep.setFechaHasta(par.getFechaHasta());
                    conODep.setFechaAlta(par.getFechaAlta());
                    conODep.setFechaMod(par.getFechaMod());
                    conODep.setUsrAlta(par.getUsrAlta());
                    conODep.setUsrMod(par.getUsrMod());


                    con.addDatos(conODep);


//Referente del contrato ----------------------------------------------------
----------

                    con.addPrestador((ItcPrestadorEy)pm.getObjectById(
                    par.getReferente()),
                    par.getFechaDesde(), par.getFechaHasta(),
                    ItcPrestadorEy.REFERENTE);


//Prestador del contrato ----------------------------------------------------
----------

                    con.addPrestador((ItcPrestadorEy)pm.getObjectById(
                    par.getReferente()),
                    par.getFechaDesde(), par.getFechaHasta(),
                    ItcPrestadorEy.PRESTADOR);


//Pois del contrato ---------------------------------------------------------
----

Iterator itePoi = par.getPois();
                    while(itePoi.hasNext())

                            con.addPoi((ItcPoiEy)pm.getObjectById((Itc
                            PoiPK) itePoi.next()),
                            par.getFechaDesde(),
                            |par.getFechaHasta());


//Servicios del contrato ----------------------------------------------------
----------
                    Iterator iteServ = par.getServicios();
                    while(iteServ.hasNext())
                            con.addServicio((ItcServicioEy)pm.getObject
                            ById((ItcServicioPK) iteServ.next()),
                            par.getFechaDesde(), par.getFechaHasta());

//Plan del contrato ---------------------------------------------------------
----

                    Iterator itePlan = par.getPlanes();
                    while(itePlan.hasNext())
```

```
                                con.addPlan((ItcPlanEy)pm.getObjectById((I
                                tcPlanPK) itePlan.next()),
                                par.getFechaDesde(), par.getFechaHasta());

                        pm.makePersistent(con);

                        pm.close();
                }
        }
```

7.4 Conclusión

En este capítulo, se presentó toda la información, necesaria para que el lector comprenda, la utilización de la nueva metodología de desarrollo.

La definición de los objetos de negocio, se realiza a través de clases comunes Java junto a la elaboración del *Descriptor de Persistencia* que permite describirlos. Ambos elementos son utilizados por una herramienta que realiza el proceso de 'enhanced', crea las instancias que representan el acceso a los datos y construye las claves pertenecientes a las entidades persistidas.

Por último, los mecanismos de construcción de los servicios, así como los componentes que brindan la comunicación con el cliente, continuarán siendo los mismos que los utilizados por 'la empresa' en sus procesos de desarrollo.

CAPÍTULO 8

Migración entre las arquitecturas de software

En los capítulos anteriores, se realizó el análisis de la arquitectura empleada por 'la empresa', para sus procesos de desarrollo y, también se explicó la nueva arquitectura de software, que utiliza el Framework JDO, como mecanismo de persistencia de las instancias del negocio. En el presente capítulo se muestran los conocimientos necesarios para lograr una migración entre ambas arquitecturas.

Aquí se presentan los aspectos requeridos para transformar, los componentes que modelan los objetos de negocio en la arquitectura de 'la empresa', en su equivalente, en el nuevo ambiente de desarrollo. También, se explican los elementos necesarios para convertir los servicios brindados por el anterior entorno en su semejante, en el nuevo escenario.

8.1 Objetos de negocio

Anteriormente, se mencionó que las entidades de negocio representan a las abstracciones del dominio, las cuales pueden ser tangibles o intangibles. Estos elementos, en la nueva arquitectura, son modelados por clases comunes de Java y no necesitan de ninguna lógica especial para lograr persistirse en el medio de almacenamiento.

Como conocemos, las entidades se relacionan unas con otras. La representación de estas relaciones en el nuevo ambiente tiene las siguientes características:

- Son modeladas haciendo referencia a otras clases o a colecciones de clases. En ambos casos se requiere representar estas asociaciones en el *Descriptor de Persistencia*, propio del Framework JDO.

- Si tienen cardinalidad múltiple, además de definirlas en el *Descriptor de Persistencia*, es necesario usar el Framework de colecciones de Java para crearlas.

- Si son relaciones especiales del tipo *todo-parte* solamente se debe especificar en el *Descriptor de Persistencia*, que el objeto que corresponde a la parte, está incluído en el objeto que corresponde al todo.

- En caso de relaciones compuestas por atributos que la describen, hay que crear una entidad falsa que agrupe tales características.

Las entidades del dominio definidas en la arquitectura de 'la empresa' requieren:

- Ser detalladas en el *Descriptor de Persistencia* perteneciente al nuevo ambiente.

- Trasformar las relaciones entre las clases para que representen la lógica del dominio.

- Eliminar la lógica asociada con el almacenamiento de las instancias transitorias[14].

[14] Una instancia transitoria representa a una instancia de una clase que no ha sido persistida.

8.2 Acceso a los datos

En apartados precedentes, se expresó que la capa DAO contiene las responsabilidades que permiten el acceso a los datos. En la nueva arquitectura, la implementación de estos servicios está basada en el uso de componentes propios del Framework JDO, junto al lenguaje de consulta definido para tal función dentro de su arquitectura. También, es posible el uso de sentencias SQL para el acceso a los datos como alternativa a las capacidades brindadas por la arquitectura JDO.

Lo mencionado requiere que cada uno de los métodos definidos en los componentes DAO de la arquitectura de 'la empresa', sean recodificados usando las herramientas anteriormente definidas.

8.3 Implementación de los servicios

En la nueva arquitectura, las actividades que permiten concretar los servicios, que serán entregados a los clientes del sistema, son similares a las empleadas por 'la empresa' en sus procesos de desarrollo, presentando discrepancias en las implementaciones de los servicios del componente *ApplicationLogic*.

La migración de los servicios, entre ambas arquitecturas, requiere que la codificación de los métodos publicados en la capa lógica, sea realizada usando los mecanismos ofrecidos, por la arquitectura JDO, en lugar de emplear el Framework Enterprise JavaBeans como medio de acceso a las instancias persistidas.

Por último, es necesario que desde el componente *ApplicationLogic* se pueda obtener acceso a las capacidades brindadas por el Framework JDO.

8.4 Herramienta automática

Como se detalló, para lograr la funcionalidad en el nuevo ambiente, es necesaria la creación de una herramienta cuyo objetivo es, crear las instancias que permiten el acceso a los datos, construir las claves primarias correspondientes a las entidades y realizar el proceso de 'enhanced' a las clases persistentes.

8.5 Conclusión

En este capítulo se presentó la información necesaria para llevar a cabo la migración entre ambos esquemas de desarrollo.

Para lograr el "salto" de la arquitectura de 'la empresa' a la nueva plataforma de desarrollo es preciso, definir las entidades del dominio y sus relaciones basado en la lógica del negocio, usar los servicios de JDO como medio de acceso a los datos almacenados y crear un proceso orientado a realizar las tareas de 'enhanced' sobre las clases persistentes.

CAPÍTULO 9

Conclusión y Trabajo Futuro

9.1 Conclusión

Durante el desarrollo de esta tesis se arribó a un conjunto de resultados producto de las experiencias adquiridas de la migración de un modelo basado en componentes a otro orientado a objetos, sobre la capa de persistencia. Para obtenerlos, se tomó como referencia la arquitectura de software y los procesos de desarrollo que se emplean actualmente en la empresa de telecomunicaciones en donde desempeño mis actividades, y se migraron los mismos a una nueva arquitectura que soporta el paradigma orientado a objetos para el almacenamiento de los datos.

El uso del modelo orientado a objetos sobre un sistema basado en componentes que manejan la persistencia manualmente por medio de la tecnología Enterprise JavaBeans, provoca cambios en la funcionalidad brindada en la capa del servidor. Estas modificaciones, ocasionan una variación en la lógica presente en los componentes que actúan como *Facade* y que, además, contienen la funcionalidad relacionada con el acceso a los datos almacenados en la base de datos.

De esta manera, los servicios responsables del almacenamiento requieren que tanto las entidades del dominio que deben ser persistidas como sus relaciones, se modelen usando las capacidades ofrecidas por el lenguaje de programación subyacente.

Además, se necesita que la codificación de los métodos publicados en la capa lógica, sea realizada con una tecnología que permita la aplicación del paradigma orientado a objetos sobre la capa

de persistencia en reemplazo del Framework Enterprise JavaBeans, como medio de acceso a las instancias almacenadas.

En esta tesis, además de analizarse el impacto de la migración entre ambos modelos, se enuncia un conjunto de *"lecciones aprendidas"* que puedan ser aplicadas para lograr dicha transformación. De tal modo, cambiar un modelo de componentes por otro orientado a objetos en la capa de persistencia implica que:

- Los componentes persistentes se deben transformar en objetos planos (clases comunes Java) que representen las entidades del modelo de dominio a ser persistidas. Para ello, es importante apoyarse en el modelo de diseño de objetos e identificar cuales de ellos conforman el componente en cuestión.

- Las relaciones entre los componentes persistentes se deben modelar usando referencias a clases o a colecciones de clases que representen los componentes en cuestión.

- Las relaciones especiales del tipo *todo-parte,* donde el componente persistente contiene la lógica de almacenamiento de la parte, se deben reemplazar con un objeto de dominio que represente a la parte con su relación correspondiente.

- Las relaciones compuestas por atributos que la describen se deben modelar con una entidad que agrupe tales características.

- La lógica relacionada con el almacenamiento de las instancias propias, incluida en los componentes

persistentes, se debe eliminar ya que en la tecnología subyacente se ofrecen estos servicios.

- La jerarquía de objetos para en el nuevo modelo se debe crear usando los objetos de software que representan los componentes persistentes y lógica existente en el modelo de diseño.

- La lógica presente en los servicios de los componentes que actúan como *Facade* se debe modificar para acceder a los objetos persistentes, tal que la tecnología utilizada permita la aplicación del paradigma orientado a objetos sobre la capa de persistencia. Además, es necesario agregar dentro de esta capa un acceso a los servicios de persistencia brindados por dicha tecnología.

- Los servicios brindados en la capa DAO se deben implementar usando los objetos de software, que representan a las entidades del dominio y que reemplazan a los componentes persistentes, y el lenguaje de consulta definido por la tecnología subyacente para tal función.

- Las sentencias SQL para el acceso a los datos no son recomendadas. Si se requiere su utilización, es preferible definirlas en forma declarativa dentro de la propia tecnología.

- Los componentes que modelan el patrón *Data Transfer Object,* que contienen información de las instancias persistentes, pueden ser reemplazados por la propia instancia.

A partir de lo explicado en las *"lecciones aprendidas",* puede inferirse que el conjunto de actividades requeridas al momento de migrar un modelo para la capa de persistencia basado en componentes a otro orientado a objetos implica: analizar los componentes y su implementación más el modelo de diseño de objetos, para poder determinar qué objetos de software representaran a los componentes y cómo los mismos se relacionaran para modelar la lógica del dominio. Asimismo, se debe modificar los servicios de los componentes que tienen la implementación de las responsabilidades brindadas por la capa del servidor y los componentes que modelan el patrón Dao, para acceder a los datos bajo la nueva plataforma.

En definitiva, las capacidades ofrecidas por la utilización del modelo orientado a objetos para la persistencia pueden ser tentadoras. Sin embargo, en las empresas que cuentan con una arquitectura de desarrollo ya definida el impacto no es trivial y se requiere de un análisis "costo-beneficio" para determinar si es factible o no la utilización de esta tecnología.

Por último, todas las experiencias recabadas son aplicables a otras tecnologías similares que permitan la aplicación del paradigma orientado a objeto sobre la capa de persistencia.

9.2 Trabajo futuro

En esta tesis se analizó el impacto producido por el uso del modelo orientado a objetos sobre un sistema distribuido basado en componentes, tal como propone la especificación J2EE, que persiste los objetos de negocio en plataformas de base de datos relacionales. Si bien la plataforma relacional es la tecnología más utilizada para el almacenamiento de la información, esta no es la única opción del mercado.

De esta manera, dentro de las alternativas de almacenamiento existentes, podemos mencionar el agrupamiento de datos sobre archivos en formato XML, bases de datos jerárquicas, en red u orientadas a objetos, sistemas legacy o incluso en archivos denominados 'planos'.

Actualmente, la plataforma estándar que soporta las alternativas de almacenamiento antes mencionadas, es JDO, la cual es empleada con éxito en ambientes con base de datos relacionales o con base de datos orientadas a objetos.

Por lo tanto, dada la amplia utilización del modelo de componentes para el desarrollo de aplicaciones distribuidas, en trabajos futuros resultaría interesante analizar el impacto acaecido en la práctica en empresas que emplean EJB de Entidad y bases de datos relacionales como medio de persistencia, al cambiar la infraestructura de almacenamiento, y en particular, cuando se utilizan bases de datos orientadas a objetos como soporte.

Asimismo, sería enriquecedor obtener un nuevo conjunto de *"lecciones aprendidas"* que extiendan las presentes en este trabajo y que aporten experiencias sobre el proceso aplicado, como así también que describan metodológicamente y de manera estándar los pasos necesarios para realizar migraciones entre las plataformas estudiadas.

APÉNDICE A

Mapeo relacional

JDO es independiente de la plataforma de almacenamiento que se utilice como medio de persistencia [20]. Es decir, el Framework puede ser empleado en ambientes con base de datos relacionales o con base de datos orientadas a objetos.

El empleo del Framework en almacenamientos relacionales requiere, que el modelo de objetos del dominio sea mapeado a tablas, que constituyen el esquema relacional [20].

Las estrategias de mapeos están estandarizadas y toman a los objetos del dominio como punto de partida, para construir el mapeo con las tablas del esquema relacional [20].

Debido a que 'la empresa' cuenta con bases de datos relacionales para el almacenamiento de sus datos, este apéndice muestra como el Framework JDO, es empleado en ambientes relacionales y como son usados los mecanismos de mapeo para persistir los objetos del dominio en el esquema relacional.

A.1 Mapeo de clases

Cuando se persiste una clase que modela al objeto del dominio se necesita definir como es representada en el esquema relacional [7].

Generalmente, toda clase se representa con una tabla. Sin embargo, existen situaciones que requieren que la clase sea representada en varias tablas de la base de datos [20].

Los mecanismos de mapeo, son los encargados de enlazar los objetos de la aplicación con las tablas de la base de datos relacional.

En el proceso de mapeo, cada atributo que conforma la clase es asociado a una o varias columnas, de una o varias tablas, del ambiente relacional [20].

El mapeo es especificado a través de un archivo xml llamado *Descriptor de Persistencia* [20]. Este archivo contiene un conjunto de tags que definen la información de persistencia de la clase [14].

A continuación, se muestra un ejemplo de mapeo de una clase con una tabla de la base de datos.

```
package jdo;

import java.util.Date;

import jdo.pk.NotePK;
import jdo.pk.PrimaryKey;

public class Note extends Entity {

	public int id;
	public String name = null;
	public Date date = null;

	protected Note(){}

	public Note(int id, String name, Date date)
	{
		this.id = id;
		this.name = name;
		this.date = date;
	}

	public Date getDate() {
		return date;
	}

	public void setDate(Date date) {
		this.date = date;
	}

	public int getId() {
		return id;
	}
```

```
	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	/* (non-Javadoc)
	 * @see java.lang.Object#toString()
	 */
	public String toString() {
		return "(Note) id = " + id + " name = " + name + " date = " + date;
	}

	/* (non-Javadoc)
	 * @see jdo.Entity#getPK()
	 */
	public PrimaryKey getPK() {
		return new NotePK(this.id);
	}

}
```

En este caso queremos mapear la clase *Note* con las columnas *ID*, *NAME* y *DATE* de la tabla *NOTE*. El *Descriptor de Persistencia* es el siguiente:

```
<jdo>
   <package name="jdo">
      <class name="Note" detachable="true" table="NOTE" identity-
type="application"
          objectid-class="jdo.pk.NotePK">
         <field name="id" primary-key="true" persistence-modifier="persistent">
           <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
         </field>
         <field name="name" persistence-modifier="persistent">
           <column name="NAME" jdbc-type="VARCHAR" length="45" allows-
null="false"/>
         </field>
         <field name="date" persistence-modifier="persistent">
           <column name="DATE" jdbc-type="DATE" allows-null="false"/>
         </field>
      </class>
   </package>
</jdo>
```

En el *Descriptor de Persistencia* anterior, se definió la tabla y las columnas de la base de datos que serán utilizadas para persistir la entidad del dominio. Además, se usó un conjunto de tags para definir información adicional, que será empleada por la implementación JDO para la persistencia de los objetos.

Una visión de los tags que pueden ser usados en el *Descriptor de Persistencia*, puede ser obtenida del *Apéndice B – Estructura del Descriptor de Persistencia* o de la especificación de JDO [19].

A.2 Mapeo de relaciones

Un sistema de software orientado a objetos está compuesto por componentes que encapsulan datos y funciones, los cuales pueden comunicarse a través de mensajes, y heredar los atributos y los comportamientos de otros componentes [21].

El Framework JDO define un conjunto de tags para materializar las asociaciones estructurales y las relaciones de herencia, que modelan las políticas del usuario.

A.2.1 Relaciones 1 a 1

Una relación de 1 a 1 se establece cuando un objeto de una clase, es asociado con otro objeto de la misma o distinta clase. Estas asociaciones pueden ser bidireccionales (ambos objetos se conocen) o unidireccionales (solamente uno conoce al otro) [5].

A.2.1.1 Relaciones 1 a 1 (Unidireccional)

Aquí se presentan dos clases: *Product* y *Note*, donde solamente la clase *Product* conoce a la clase *Note*.

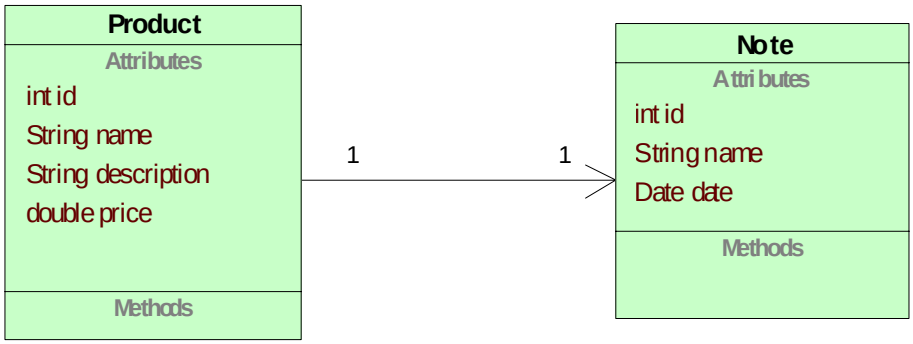

Figura 14 – *Relación de 1 a 1(Unidireccional)*

El *Descriptor de Persistencia* que permite modelar esta relación es:

```
<jdo>
   <package name="jdo">
       <class name="Product" detachable="true" table="PRODUCT"
          identity-type="application" objectid-class="jdo.pk.ProductPK">
         <field name="id" primary-key="true" persistence-modifier="persistent">
           <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
         </field>
         <field name="name" persistence-modifier="persistent">
           <column name="NAME" jdbc-type="VARCHAR" length="255" allows-
null="false"/>
         </field>
         <field name="description" persistence-modifier="persistent">
           <column name="DESCRIPTION" jdbc-type="VARCHAR" length="255"
           allows-null="true"/>
         </field>
         <field name="price" persistence-modifier="persistent">
           <column name="PRICE" jdbc-type="DECIMAL" length="17" scale="2"
           allows-null="false"/>
         </field>
         <field name="note" persistence-modifier="persistent">
            <column name="NOTE" jdbc-type="NUMERIC" allows-null="false"/>
         </field>
       </class>
       <class name="Note" detachable="true" table="NOTE" identity-
type="application"
          objectid-class="jdo.pk.NotePK">
         <field name="id" primary-key="true" persistence-modifier="persistent">
           <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
         </field>
         <field name="name" persistence-modifier="persistent">
           <column name="NAME" jdbc-type="VARCHAR" length="45" allows-
null="false"/>
         </field>
         <field name="date" persistence-modifier="persistent">
           <column name="DATE" jdbc-type="DATE" allows-null="false"/>
         </field>
       </class>
```

```
   </package>
</jdo>
```

En este entorno, se crean dos tablas para representar la relación unidireccional. Una tabla llamada *PRODUCT* con los campos *ID*, *NAME*, *DESCRIPTION*, *PRICE* y *NOTE*, y otra tabla llamada *NOTE* con los campos *ID*, *NAME* y *DATE*.

El campo *NOTE* de la tabla *PRODUCT* es usado para mantener la asociación entre ambas entidades del dominio.

A.2.1.2 Relaciones 1 a 1 (Bidireccional)

Aquí se presentan dos clases, *Client* y *Account*, que se conocen mutuamente.

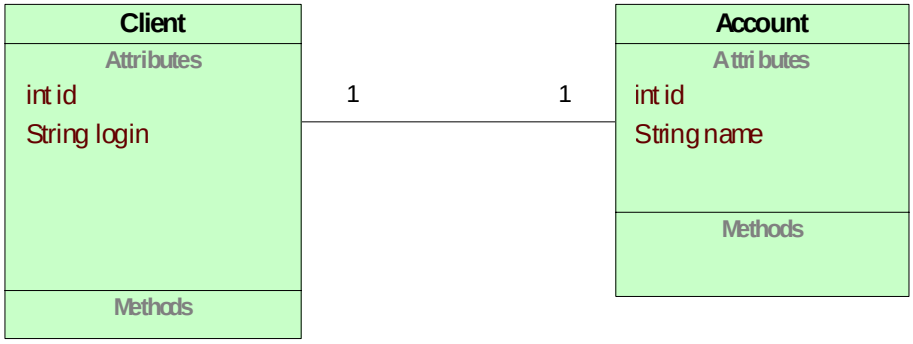

Figura 15 – *Relación de 1 a 1(Bidireccional)*

El *Descriptor de Persistencia* que permite describir esta relación es:

```
<jdo>
   <package name="jdo">
       <class name="Client" detachable="true" table="CLIENT" identity-
type="application"
         objectid-class="jdo.pk.ClientPK">
         <field name="id" primary-key="true" persistence-modifier="persistent">
           <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
         </field>
         <field name="login" persistence-modifier="persistent">
```

```
          <column name="LOGIN" jdbc-type="VARCHAR" length="45" allows-
null="false"/>
        </field>
        <field name="account" persistence-modifier="persistent">
          <column name="ACCOUNT_ID" jdbc-type="NUMERIC" allows-
null="true"/>
        </field>
      </class>
      <class name="Account" detachable="true" table="ACCOUNT"
          identity-type="application" objectid-class="jdo.pk.AccountPK">
        <field name="id" primary-key="true" persistence-modifier="persistent">
          <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
        </field>
        <field name="name" persistence-modifier="persistent">
          <column name="NAME" jdbc-type="VARCHAR" length="45" allows-
null="false"/>
        </field>
        <field name="client" persistence-modifier="persistent">
          <column name="CLIENT_ID" jdbc-type="NUMERIC" allows-null="false"/>
        </field>
      </class>
   </package>
</jdo>
```

En esta situación se crean dos tablas para representar la relación bidirecional. Una tabla llamada *CLIENT* con los campos *ID*, *LOGIN* y *ACCOUNT_ID*, y otra tabla llamada *ACCOUNT* con los campos *ID*, *NAME* y *CLIENT_ID*.

Los campos *ACCOUNT_ID* y *CLIENT_ID* son usados para mantener la asociación bidireccional entre ambos objetos del dominio.

A.2.2 Relaciones 1 a N

Una relación de 1 a N se establece cuando un objeto de una clase tiene asociado un conjunto de objetos de otra clase [6].

JDO emplea diversos mecanismos para representar estas asociaciones [9]. Aquí modelaremos las relaciones de 1 a N de tipo bidireccional por medio de la interfaz Set[15] de Java.

[15] Interfaz del lenguaje de programación Java que brinda un conjunto de servicios para el manejo de colecciones de objetos.

Es importante distinguir dos cosas: la primera, conocer que implementaciones de la interfaz Set soporta el vendedor JDO; y la segunda, notar que la interfaz Set no permite elementos duplicados.

A.2.2.1 Relaciones 1 a N (Bidireccional)

Aquí la clase *Product*, tiene asociado un conjunto de objetos de la clase *Client*, y cada objeto *Client* tiene una referencia al objeto *Product* que lo contiene.

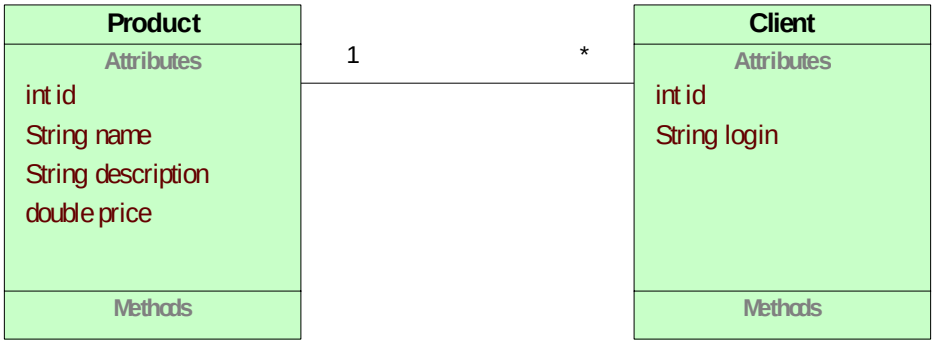

Figura 16 – *Relación de 1 a N(Bidireccional)*

El *Descriptor de Persistencia* que permite representar esta relación es:

```
<jdo>
   <package name="jdo">
      <class name="Product" detachable="true" table="PRODUCT" identity-
type="application"
          objectid-class="jdo.pk.ProductPK">
          <field name="id" primary-key="true" persistence-modifier="persistent">
            <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
          </field>
          <field name="name" persistence-modifier="persistent">
            <column name="NAME" jdbc-type="VARCHAR" length="255" allows-
null="false"/>
          </field>
          <field name="description" persistence-modifier="persistent">
            <column name="DESCRIPTION" jdbc-type="VARCHAR" length="255"
              allows-null="true"/>
          </field>
          <field name="price" persistence-modifier="persistent">
            <column name="PRICE" jdbc-type="DECIMAL" length="17" scale="2"
              allows-null="false"/>
```

```
        </field>
        <field name="clients" persistence-modifier="persistent" mapped-by="prod">
          <collection element-type="jdo.Client"/>
        </field>
      <class name="Client" detachable="true" table="CLIENT" identity-
type="application"
        objectid-class="jdo.pk.ClientPK">
        <field name="id" primary-key="true" persistence-modifier="persistent">
          <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
        </field>
        <field name="login" persistence-modifier="persistent">
          <column name="LOGIN" jdbc-type="VARCHAR" length="45" allows-
null="false"/>
        </field>
        <field name="prod" persistence-modifier="persistent">
          <column name="PRODUCT_ID" jdbc-type="NUMERIC" allows-
null="true"/>
        </field>
      </class>
   </package>
</jdo>
```

En este ámbito se crean dos tablas para describir la relación.

Una tabla llamada *PRODUCT* con los campos *ID*, *NAME*, *DESCRIPTION* y *PRICE*, y otra tabla llamada *CLIENT* con los campos *ID*, *LOGIN* y *PRODUCT_ID*.

El campo *PRODUCT_ID* de la tabla *CLIENT* es usado para mantener la asociación entre ambas entidades del dominio.

En el *Descriptor de Persistencia,* se utiliza el tag llamado *mapped-by* para indicar a la implementación JDO, que el campo *prod* de la clase *Client*, es utilizado para obtener la colección de objetos asociados con el objeto *Product* [6].

A.2.3 Relaciones M a N

Una relación de M a N, se establece cuando un objeto de la clase A, tiene asociado un conjunto de objetos de la clase B, y cada objeto de la clase B, tiene asociado un conjunto de objetos de la clase A [10].

Estas asociaciones pueden representarse de diferentes maneras [9]. A continuación, modelaremos las relaciones de M a N por medio de la interfaz Set de Java.

Aquí la clase *Product* tiene asociado un conjunto de objetos de la clase *Provider*, y cada objeto *Provider*, tiene una referencia a un conjunto de objetos de la clase *Product*.

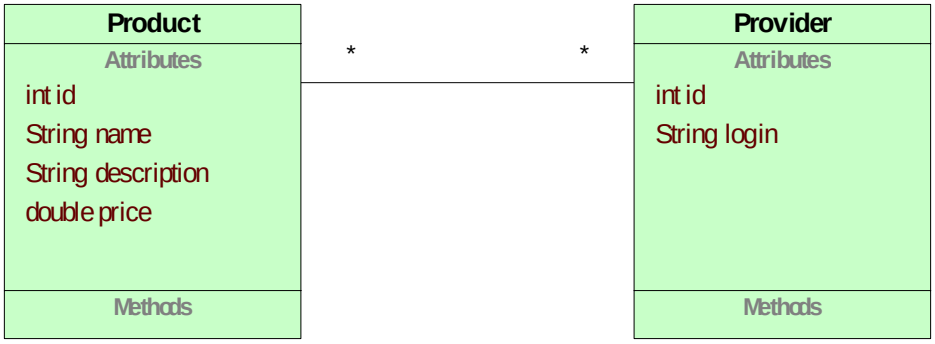

Figura 17 – *Relación de M a N*

El *Descriptor de Persistencia* que permite definir esta relación es:

```
<jdo>
   <package name="jdo">
      <class name="Product" detachable="true" table="PRODUCT" identity-
type="application"
          objectid-class="jdo.pk.ProductPK">
         <field name="id" primary-key="true" persistence-modifier="persistent">
           <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
         </field>
         <field name="name" persistence-modifier="persistent">
           <column name="NAME" jdbc-type="VARCHAR" length="255" allows-
null="false"/>
         </field>
         <field name="description" persistence-modifier="persistent">
           <column name="DESCRIPTION" jdbc-type="VARCHAR" length="255"
             allows-null="true"/>
         </field>
         <field name="price" persistence-modifier="persistent">
           <column name="PRICE" jdbc-type="DECIMAL" length="17" scale="2"
             allows-null="false"/>
         </field>
         <field name="providers" table="PRODUCTS_PROVIDERS"
            persistence-modifier="persistent">
            <collection element-type="jdo.Provider" embedded-element="false"/>
```

```
            <join>
              <column name="PRODUCT_ID" jdbc-type="NUMERIC" allows-
null="false"/>
            </join>
            <element>
              <column name="PROVIDER_ID" jdbc-type="NUMERIC" allows-
null="false"/>
            </element>
          </field>
      </class>
      <class name="Provider" detachable="true" table="PROVIDER"
         identity-type="application" objectid-class="jdo.pk.ProviderPK">
          <field name="id" primary-key="true" persistence-modifier="persistent">
           <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
          </field>
          <field name="name" persistence-modifier="persistent">
           <column name="NAME" jdbc-type="VARCHAR" length="255" allows-
null="false"/>
          </field>
          <field name="products" table="PRODUCTS_PROVIDERS"
            persistence-modifier="persistent">
            <collection element-type="jdo.Product" embedded-element="false"/>
            <join>
              <column name="PROVIDER_ID" jdbc-type="NUMERIC" allows-
null="false"/>
            </join>
            <element>
              <column name="PRODUCT_ID" jdbc-type="NUMERIC" allows-
null="false"/>
            </element>
          </field>
      </class>
   </package>
</jdo>
```

En este ambiente, se crean tres tablas para describir la relación. Una tabla llamada *PRODUCT* con los campos *ID*, *NAME*, *DESCRIPTION* y *PRICE*. Otra tabla llamada *PROVIDER* con los campos *ID* y *NAME*. Y una tabla de unión llamada *PRODUCTS_PROVIDERS* con los campos *PROVIDER_ID* y *PRODUCT_ID*.

Los campos *PROVIDER_ID* y *PRODUCT_ID* de la tabla *PRODUCTS_PROVIDERS* son usados para mantener la relación entre ambos objetos del dominio.

La unión, es representada por medio del tag llamado *join*, que asocia una o varias columnas de la tabla de unión, con una o varias columnas de la tabla origen de la relación [20].

A.2.4 Relaciones de herencia

Es normal que en todo desarrollo de software orientado a objetos, existan relaciones donde una clase, comparte la estructura y el comportamiento definido en una o más clases [1]. A continuación, se empleará solamente una de las tres maneras que define el Framework JDO para modelar este tipo de relación.

Aquí la clase *Product* es especializada por la clase *Book* y la clase *Disk*:

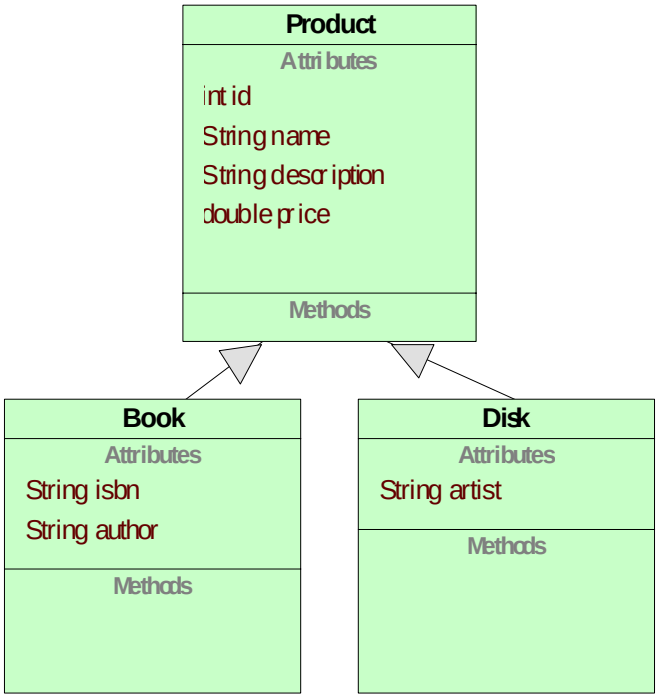

Figura 18 – *Relación de herencia*

El *Descriptor de Persistencia* que permite mapear esta relación es:

```
<jdo>
   <package name="jdo">
<class name="Product" detachable="true" table="PRODUCT" identity-
type="application"
          objectid-class="jdo.pk.ProductPK">
         <field name="id" primary-key="true" persistence-modifier="persistent">
```

```
            <column name="ID" jdbc-type="NUMERIC" allows-null="false"/>
          </field>
          <field name="name" persistence-modifier="persistent">
            <column name="NAME" jdbc-type="VARCHAR" length="255" allows-
null="false"/>
          </field>
          <field name="description" persistence-modifier="persistent">
            <column name="DESCRIPTION" jdbc-type="VARCHAR" length="255"
              allows-null="true"/>
          </field>
          <field name="price" persistence-modifier="persistent">
            <column name="PRICE" jdbc-type="DECIMAL" length="17" scale="2"
              allows-null="false"/>
          </field>
        </class>
        <class name="Book" detachable="true" table="BOOK"
          persistence-capable-superclass="jdo.Product" identity-type="application">
          <inheritance strategy="new-table"/>
          <field name="isbn" persistence-modifier="persistent">
            <column name="ISBN" jdbc-type="VARCHAR" length="22" allows-
null="false"/>
          </field>
          <field name="author" persistence-modifier="persistent">
            <column name="AUTHOR" jdbc-type="VARCHAR" length="45" allows-
null="true"/>
          </field>
        </class>
       <class name="Disk" detachable="true" table="DISK"
          persistence-capable-superclass="jdo.Product" identity-type="application">
          <inheritance strategy="new-table"/>
          <field name="artist" persistence-modifier="persistent">
            <column name="ARTIST" jdbc-type="VARCHAR" length="45" allows-
null="false"/>
          </field>
        </class>
    </package>
</jdo>
```

En este entorno se crean tres tablas para representar la relación. Una tabla llamada *PRODUCT* con los campos *ID*, *NAME*, *DESCRIPTION* y *PRICE*. Otra tabla llamada *BOOK* con los campos *ID*, *ISBN* y *AUTHOR*. Y una tercera tabla llamada *DISK* con los campos *ID* y *ARTIST*.

Los campos *ID* de la tablas *PRODUCT*, *BOOK* y *DISK* son usados para mantener la relación de herencia entre los objetos del dominio.

En el *Descriptor de Persistencia*, se utiliza el tag llamado *persistence-capable-superclass* para relacionar las clases, y el tag llamado *inheritance strategy*, para definir la estrategia de herencia que debe aplicarse [8].

APÉNDICE B

Estructura del Descriptor de Persistencia

En este apéndice, se describe la estructura del archivo xml utilizada por el Framework JDO, durante el proceso de modificación o de enriquecimiento de las clases. Aquí, no se pretende tratar el tema en profundidad, sino brindar una introducción al lector de los tags, que componen al *Descriptor de Persistencia* para que éste, adquiera el conocimiento necesario para comprender los temas tratados en este trabajo.

B.1 Contenido del Descriptor de Persistencia

El *Descriptor de Persistencia* esta compuesto por los siguientes tags [20]:

```
<!ELEMENT jdo (extension*, (package|query)+, extension*)>
<!ATTLIST jdo catalog CDATA #IMPLIED>
<!ATTLIST jdo schema CDATA #IMPLIED>

<!ELEMENT package (extension*, (interface|class|sequence)+,
extension*)>
<!ATTLIST package name CDATA ''>
<!ATTLIST package catalog CDATA #IMPLIED>
<!ATTLIST package schema CDATA #IMPLIED>

<!ELEMENT interface (extension*, datastore-identity?, primary-
key?, inheritance?, version?, join*, foreign-key*, index*,
unique*, property*, query*, fetch-group*, extension*)>
<!ATTLIST interface name CDATA #REQUIRED>
<!ATTLIST interface table CDATA #IMPLIED>
<!ATTLIST interface identity-type
(datastore|application|nondurable) #IMPLIED>
<!ATTLIST interface objectid-class CDATA #IMPLIED>
<!ATTLIST interface requires-extent (true|false) 'true'>
<!ATTLIST interface detachable (true|false) 'false'>
<!ATTLIST interface embedded-only (true|false) #IMPLIED>
<!ATTLIST interface catalog CDATA #IMPLIED>
<!ATTLIST interface schema CDATA #IMPLIED>

<!ELEMENT property (extension*, (array|collection|map)?, join?,
embedded?, element?, key?, value?, order?, column*, foreign-
key?, index?, unique?, extension*)>
```

```
<!ATTLIST property name CDATA #REQUIRED>
<!ATTLIST property persistence-modifier
(persistent|transactional|none) #IMPLIED>
<!ATTLIST property default-fetch-group (true|false) #IMPLIED>
<!ATTLIST property load-fetch-group CDATA #IMPLIED>
<!ATTLIST property null-value (default|exception|none) 'none'>
<!ATTLIST property dependent (true|false) #IMPLIED>
<!ATTLIST property embedded (true|false) #IMPLIED>
<!ATTLIST property primary-key (true|false) 'false'>
<!ATTLIST property value-strategy CDATA #IMPLIED>
<!ATTLIST property sequence CDATA #IMPLIED>
<!ATTLIST property serialized (true|false) #IMPLIED>
<!ATTLIST property field-type CDATA #IMPLIED>
<!ATTLIST property table CDATA #IMPLIED>
<!ATTLIST property column CDATA #IMPLIED>
<!ATTLIST property delete-action
(restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST property indexed (true|false|unique) #IMPLIED>
<!ATTLIST property unique (true|false) #IMPLIED>
<!ATTLIST property mapped-by CDATA #IMPLIED>
<!ATTLIST property recursion-depth CDATA #IMPLIED>
<!ATTLIST property field-name CDATA #IMPLIED>

<!ELEMENT class (extension*, implements*, datastore-identity?,
primary-key?, inheritance?, version?, join*, foreign-key*,
index*, unique*, column*, field*, property*, query*, fetch-
group*, extension*)>
<!ATTLIST class name CDATA #REQUIRED>
<!ATTLIST class identity-type
(application|datastore|nondurable) #IMPLIED>
<!ATTLIST class objectid-class CDATA #IMPLIED>
<!ATTLIST class table CDATA #IMPLIED>
<!ATTLIST class requires-extent (true|false) 'true'>
<!ATTLIST class persistence-capable-superclass CDATA #IMPLIED>
<!ATTLIST class detachable (true|false) 'false'>
<!ATTLIST class embedded-only (true|false) #IMPLIED>
<!ATTLIST class persistence-modifier (persistence-
capable|persistence-aware|non-persistent) #IMPLIED>
<!ATTLIST class catalog CDATA #IMPLIED>
<!ATTLIST class schema CDATA #IMPLIED>

<!ELEMENT primary-key (extension*, column*, extension*)>
<!ATTLIST primary-key name CDATA #IMPLIED>
<!ATTLIST primary-key column CDATA #IMPLIED>

<!ELEMENT join (extension*, primary-key?, column*, foreign-
key?, index?, unique?, extension*)>
<!ATTLIST join table CDATA #IMPLIED>
<!ATTLIST join column CDATA #IMPLIED>
<!ATTLIST join outer (true|false) 'false'>
<!ATTLIST join delete-action
(restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST join indexed (true|false|unique) #IMPLIED>
<!ATTLIST join unique (true|false) #IMPLIED>

<!ELEMENT version (extension*, column*, index?, extension*)>
<!ATTLIST version strategy CDATA #IMPLIED>
<!ATTLIST version column CDATA #IMPLIED>
```

```
<!ATTLIST version indexed (true|false|unique) #IMPLIED>

<!ELEMENT datastore-identity (extension*, column*, extension*)>
<!ATTLIST datastore-identity column CDATA #IMPLIED>
<!ATTLIST datastore-identity strategy CDATA 'native'>
<!ATTLIST datastore-identity sequence CDATA #IMPLIED>

<!ELEMENT implements (extension*, property*, extension*)>
<!ATTLIST implements name CDATA #REQUIRED>

<!ELEMENT inheritance (extension*, join?, discriminator?,
extension*)>
<!ATTLIST inheritance strategy CDATA #IMPLIED>

<!ELEMENT discriminator (extension*, column*, index?,
extension*)>
<!ATTLIST discriminator column CDATA #IMPLIED>
<!ATTLIST discriminator value CDATA #IMPLIED>
<!ATTLIST discriminator strategy CDATA #IMPLIED>
<!ATTLIST discriminator indexed (true|false|unique) #IMPLIED>

<!ELEMENT column (extension*)>
<!ATTLIST column name CDATA #IMPLIED>
<!ATTLIST column target CDATA #IMPLIED>
<!ATTLIST column target-field CDATA #IMPLIED>
<!ATTLIST column jdbc-type CDATA #IMPLIED>
<!ATTLIST column sql-type CDATA #IMPLIED>
<!ATTLIST column length CDATA #IMPLIED>
<!ATTLIST column scale CDATA #IMPLIED>
<!ATTLIST column allows-null (true|false) #IMPLIED>
<!ATTLIST column default-value CDATA #IMPLIED>
<!ATTLIST column insert-value CDATA #IMPLIED>

<!ELEMENT field (extension*, (array|collection|map)?, join?,
embedded?, element?, key?, value?, order?, column*, foreign-
key?, index?, unique?, extension*)>
<!ATTLIST field name CDATA #REQUIRED>
<!ATTLIST field persistence-modifier
(persistent|transactional|none) #IMPLIED>
<!ATTLIST field field-type CDATA #IMPLIED>
<!ATTLIST field table CDATA #IMPLIED>
<!ATTLIST field column CDATA #IMPLIED>
<!ATTLIST field primary-key (true|false) 'false'>
<!ATTLIST field null-value (exception|default|none) 'none'>
<!ATTLIST field default-fetch-group (true|false) #IMPLIED>
<!ATTLIST field embedded (true|false) #IMPLIED>
<!ATTLIST field serialized (true|false) #IMPLIED>
<!ATTLIST field dependent (true|false) #IMPLIED>
<!ATTLIST field value-strategy CDATA #IMPLIED>
<!ATTLIST field delete-action
(restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST field indexed (true|false|unique) #IMPLIED>
<!ATTLIST field unique (true|false) #IMPLIED>
<!ATTLIST field sequence CDATA #IMPLIED>
<!ATTLIST field load-fetch-group CDATA #IMPLIED>
<!ATTLIST field recursion-depth CDATA #IMPLIED>
<!ATTLIST field mapped-by CDATA #IMPLIED>
```

```
<!ELEMENT foreign-key (extension*, (column* | field* |
property*), extension*)>
<!ATTLIST foreign-key table CDATA #IMPLIED>
<!ATTLIST foreign-key deferred (true|false) #IMPLIED>
<!ATTLIST foreign-key delete-action
(restrict|cascade|null|default|none) 'restrict'>
<!ATTLIST foreign-key update-action
(restrict|cascade|null|default|none) 'restrict'>
<!ATTLIST foreign-key unique (true|false) #IMPLIED>
<!ATTLIST foreign-key name CDATA #IMPLIED>

<!ELEMENT collection (extension*)>
<!ATTLIST collection element-type CDATA #IMPLIED>
<!ATTLIST collection embedded-element (true|false) #IMPLIED>
<!ATTLIST collection dependent-element (true|false) #IMPLIED>
<!ATTLIST collection serialized-element (true|false) #IMPLIED>

<!ELEMENT map (extension)*>
<!ATTLIST map key-type CDATA #IMPLIED>
<!ATTLIST map embedded-key (true|false) #IMPLIED>
<!ATTLIST map dependent-key (true|false) #IMPLIED>
<!ATTLIST map serialized-key (true|false) #IMPLIED>
<!ATTLIST map value-type CDATA #IMPLIED>
<!ATTLIST map embedded-value (true|false) #IMPLIED>
<!ATTLIST map dependent-value (true|false) #IMPLIED>
<!ATTLIST map serialized-value (true|false) #IMPLIED>

<!ELEMENT key (extension*, embedded?, column*, foreign-key?,
index?, unique?, extension*)>
<!ATTLIST key column CDATA #IMPLIED>
<!ATTLIST key table CDATA #IMPLIED>
<!ATTLIST key delete-action
(restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST key update-action
(restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST key indexed (true|false|unique) #IMPLIED>
<!ATTLIST key unique (true|false) #IMPLIED>
<!ATTLIST key mapped-by CDATA #IMPLIED>

<!ELEMENT value (extension*, embedded?, column*, foreign-key?,
index?, unique?, extension*)>
<!ATTLIST value column CDATA #IMPLIED>
<!ATTLIST value table CDATA #IMPLIED>
<!ATTLIST value delete-action
(restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST value update-action
(restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST value indexed (true|false|unique) #IMPLIED>
<!ATTLIST value unique (true|false) #IMPLIED>
<!ATTLIST value mapped-by CDATA #IMPLIED>

<!ELEMENT array (extension*)>
<!ATTLIST array element-type CDATA #IMPLIED>
<!ATTLIST array embedded-element (true|false) #IMPLIED>
<!ATTLIST array dependent-element (true|false) #IMPLIED>
<!ATTLIST array serialized-element (true|false) #IMPLIED>
```

```
<!ELEMENT element (extension*, embedded?, column*, foreign-
key?, index?, unique?, extension*)>
<!ATTLIST element column CDATA #IMPLIED>
<!ATTLIST element table CDATA #IMPLIED>
<!ATTLIST element delete-action
(restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST element update-action
(restrict|cascade|null|default|none) #IMPLIED>
<!ATTLIST element indexed (true|false|unique) #IMPLIED>
<!ATTLIST element unique (true|false) #IMPLIED>
<!ATTLIST element mapped-by CDATA #IMPLIED>

<!ELEMENT order (extension*, column*, index?, extension*)>
<!ATTLIST order column CDATA #IMPLIED>
<!ATTLIST order mapped-by CDATA #IMPLIED>

<!ELEMENT fetch-group (extension*, (fetch-
group|field|property)*, extension*)>
<!ATTLIST fetch-group name CDATA #REQUIRED>
<!ATTLIST fetch-group post-load (true|false) #IMPLIED>

<!ELEMENT embedded (extension*, (field|property)*, extension*)>
<!ATTLIST embedded owner-field CDATA #IMPLIED>
<!ATTLIST embedded null-indicator-column CDATA #IMPLIED>
<!ATTLIST embedded null-indicator-value CDATA #IMPLIED>

<!ELEMENT sequence (extension*)>
<!ATTLIST sequence name CDATA #REQUIRED>
<!ATTLIST sequence datastore-sequence CDATA #IMPLIED>
<!ATTLIST sequence factory-class CDATA #IMPLIED>
<!ATTLIST sequence strategy
(nontransactional|contiguous|noncontiguous) #REQUIRED>

<!ELEMENT index (extension*, (column* | field* | property*),
extension*)>
<!ATTLIST index name CDATA #IMPLIED>
<!ATTLIST index table CDATA #IMPLIED>
<!ATTLIST index unique (true|false) 'false'>

<!ELEMENT query (#PCDATA|extension)*>
<!ATTLIST query name CDATA #REQUIRED>
<!ATTLIST query language CDATA #IMPLIED>
<!ATTLIST query unmodifiable (true|false) 'false'>
<!ATTLIST query unique (true|false) #IMPLIED>
<!ATTLIST query result-class CDATA #IMPLIED>

<!ELEMENT unique (extension*, (column* | field* | property*),
extension*)>
<!ATTLIST unique name CDATA #IMPLIED>
<!ATTLIST unique table CDATA #IMPLIED>
<!ATTLIST unique deferred (true|false) 'false'>

<!ELEMENT extension ANY>
<!ATTLIST extension vendor-name CDATA #REQUIRED>
<!ATTLIST extension key CDATA #IMPLIED>
<!ATTLIST extension value CDATA #IMPLIED>
```

B.2 Definición de los tags

B.2.1 Elemento jdo

Este elemento, representa el tag de mayor nivel en el *Descriptor de Persistencia* [20]. Es usado para permitir declarar múltiples tags <package>. Es un tag requerido.

B.2.2 Elemento package

Este elemento, agrupa las clases persistentes en un determinado paquete [20]. Es un tag requerido.

B.2.3 Elemento class

Este elemento, declara los atributos de una clase en particular [20]. Solamente las clases a ser persistidas deben ser declaradas, es decir, las clases que no son persitidas no deben ser incluídas en el archivo xml.

También, en este tag se define la identidad de la clase, la cual permite a las aplicaciones recibir instancias específicas desde la base de datos [20]. Existen tres posibles alternativas: *dataStore* | *application* | *nondurable*.

La identidad *dataStore* indica al Framework JDO, que es el responsable de administrar la identidad de la clase. La identidad *application* permite al desarrollador administrar la identidad. Y finalmente, la identidad *nondurable* establece que la clase no tendrá soporte de identidad.

Por último, cuando la clase utiliza la identidad *application* es preciso definir, que clase actuará como identificadora de la clase en cuestión.

B.2.4 Elemento field

Este elemento, es opcional y detalla el nombre de los atributos declarados en la clase [20].

También, en este tag se especifica si el atributo en cuestión es persistente, transitorio o ninguno de ambos.

B.2.5 Elemento column

Este elemento, es utilizado para establecer el mapeo entre los atributos declarados en la clase y los campos declarados en la tabla de la base de datos [20].

También, en este tag se define el tipo de dato, su longitud y su escala.

B.2.6 Elemento collection

Cuando una clase se asocia con cero o muchas clases, se emplea este tag para modelar la relación [20]. En este elemento, se especifica el tipo de clase que corresponde a la colección.

B.2.7 Elemento extension

Este elemento especifica las extensión del vendedor JDO [20].

B.3 Ejemplo del archivo xml

A continuación, se presenta un ejemplo de un archivo xml. El lector puede notar, que se describen dos clases con sus correspondientes atributos y que, el campo *planes* de la clase *Contrato* contiene una colección de elementos de tipo Plan.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo  PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects
                Metadata 2.0//EN"  "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
   <package name="jdo">
        <class name="ItcContratoEy" table="CONTRATO" identity-type="application"
                objectid-class="jdo.pk.ItcContratoPK">
                <field name="id" primary-key="true" persistence-
                        modifier="persistent" value-strategy="autoassign">
                        <column name="CON_CONTRATO_ID" jdbc-
                                type="NUMERIC" allows-null="false"/>
                </field>
                <field name="codigo" persistence-modifier="persistent">
                        <column name="CON_CODIGO" jdbc-type="VARCHAR"
                                length="6" allows-null="false"/>
                </field>
                <field name="fechaMod" persistence-modifier="persistent">
                        <column name="CON_FECHA_MOD" jdbc-type="DATE"
                                allows-null="false"/>
                </field>
                <field name="fechaAlta" persistence-modifier="persistent">
                        <column name="CON_FECHA_ALTA" jdbc-type="DATE"
                                allows-null="false"/>
                </field>
                <field name="descripcion" persistence-modifier="persistent">
                        <column name="CON_DESCRIPCION" jdbc-type="VARCHAR"
                                length="100" allows-null="true"/>
                </field>
                <field name="planes" persistence-modifier="persistent"
                                 mapped-by="contrato">
                        <collection element-type="jdo.ItcContratoPlanEy"/>
                </field>
        </class>
        <class name="ItcPlanEy" table="PLAN" identity-type="application"
                objectid-class="jdo.pk.ItcPlanPK">
                <field name="codPlan" primary-key="true"
                        persistence-modifier="persistent">
                        <column name="PLN_COD_PLAN" jdbc-type="VARCHAR"
                                length="6" allows-null="false"/>
                </field>
                <field name="descripcion" persistence-modifier="persistent">
                        <column name="PLN_DESCRIPCION" jdbc-type="VARCHAR"
                                length="30" allows-null="false"/>
                </field>
                <field name="codTarifa" persistence-modifier="persistent">
                        <column name="PLN_COD_TARIFA" jdbc-type="NUMERIC"
                                allows-null="true"/>
                 </field>
        </class>
   </package>
</jdo>
```

REFERENCIAS BIBLIOGRÁFICAS

[1] Booch, G. (1996). Análisis y diseño orientado a objetos. Addison Wesley.

[2] Jordan, David. (2001). A comparison between Java Data Object (JDO), Serialization and JDBC for Java Persistence. Recuperado Marzo 23, 2006, desde la World Wide Web: http://www.jdocentral.com/pdf/ DavidJordan_JDOversion_12Mar02.pdf.

[3] Jordan, David. (2003, Abril). Java Data Object. USA:O'Relly.

[4] Gamma, Erich. (1995). Design Patterns: Elements of reusable Object-Oriented Software. Addison-Wesley.

[5] JPox. (2006). 1-1 Relationships. JPox. Recuperado Marzo 8, 2006, desde la World Wide Web: http://www.jpox.org/docs/1_1/relationships_1_1.html.

[6] JPox. (2006). 1-N/N-1 Relationships with Sets. JPox. Recuperado Marzo 8, 2006, desde la World Wide Web: http://www.jpox.org/docs/1_1/relationships_1_N_set.html.

[7] JPox. (2006). Class Mapping. JPox. Recuperado Marzo 8, 2006, desde la World Wide Web: http://www.jpox.org/docs/1_1/class_mapping.htm.

[8] JPox. (2006). Inheritance Strategies. JPox. Recuperado Marzo 8, 2006, desde la World Wide Web: http://www.jpox.org/docs/1_1/inheritance.html.

[9] JPox. (2006). JDO Containers. JPox. Recuperado Marzo 8, 2006, desde la World Wide Web: http://www.jpox.org/docs/1_1/collections.html.

[10] JPox. (2006). M-N Relationships. JPox. Recuperado Marzo 8, 2006, desde la World Wide Web: http://www.jpox.org/docs/1_1/relationships_M_N.html.

[11] Marinescu, Floyd. (2002). EJB Design Patterns. Canada: John Wiley & Sons.

[12] Pressman, Roger S. (1998) Ingeniería del Software: Un enfoque práctico. Addison-Wesley.

[13] Roman, Ed. (2002). Mastering Enterprise JavaBeans. USA: John Wiley & Sons.

[14] Roos, Robin M. (2003). Java Data Object. London: Addison Wesley.

[15] Sun Microsystems. (2006). Core J2EE Patterns- Data Access Object. Sun Microsystems. Recuperado Marzo 9, 2006, desde la World Wide Web: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html.

[16] Sun Microsystems. (2006). Core J2EE Patterns-Service Locator. Sun Microsystems. Recuperado Marzo 9, 2006, desde la World Wide Web: http://java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html

[17] Sun Microsystems. (2003, Noviembre 24). Enterprise JavaBeans Specifications. Version 2.1. Sun Microsystems.

[18] Sun Microsystems. (2003, Noviembre 24). Java 2 Platform Enterprise Edition Specifications. Version 1.4. Sun Microsystems. Final release.

[19] Sun Microsystems. (2002, Mayo 5). J2EE DTDs. Recuperado Marzo 8, 2006, desde la World Wide Web: http://java.sun.com/dtd/jdo_1_0.dtd.

[20] Sun Microsystems. (2005, Agosto 10). Java Data Object 2.0 Specification. Proposed Final Draft. Sun Microsystems.

[21] Yourdon, E. (1994). Object-Oriented Systems Design. USA: Prentice Hall.