UNIVERSIDAD TÉCNICA DEL NORTE

FACULTAD DE INGENIERÍA EN CIENCIAS APLICADAS

ESCUELA DE INGENIERÍA EN SISTEMAS COMPUTACIONALES

Tesis Previa a la Obtención del Título de Ingeniera en Sistemas Computacionales

TEMA

"Estudio de la Técnica ORM (Mapeo Objeto – Relacional)"

APLICATIVO

"Sistema de Gestión de Información Odontológica utilizando ORM para el Departamento de Bienestar Universitario de la UTN"

Autora: Tatiana Alexandra Freire Reyes

Director: Ing. Miguel Orquera.

Ibarra - Ecuador
Octubre 2008

Certificación

Certifico que la presente Tesis fue desarrollada en su totalidad por la egresada Srta. Tatiana Alexandra Freire Reyes, bajo mi dirección para lo cual firmo en constancia.

_____

Ing. Miguel Orquera

Director de Tesis

Agradecimiento

A Dios por el hermoso regalo de vivir y poder cosechar hoy estos triunfos y disfrutar junto a mi familia esta felicidad.

Al Ingeniero Miguel Orquera, quien supo orientarme en el presente trabajo investigativo a través de sus criterios técnicos y profesionales.

De manera especial al Ingeniero Iván Chiles e Ingeniero Fernando Garrido por su decida intervención en el desarrollo y culminación de este proyecto.

A todo el personal del Departamento de Bienestar Universitario por las facilidades, colaboración y confianza brindadas.

A mi amiga Evelin por su comprensión y apoyo constante.

Y a todas aquellas personas que con sus palabras y acciones estuvieron apoyándome en cada momento de mi vida.

Dedicatoria

Dedicado con amor a aquellas personas que con su ejemplo me enseñaron a trabajar cada día por alcanzar mis metas: mis padres, Gloria y Luis.

Taty

INDICE

CAPITULO I

Persistencia de Objetos

CAPITULO II

ORM (Mapeo Objeto Relacional)

CAPITULO III

Mapeo Objeto Relacional con JPA

CAPITULO IV

Persistencia Manual vs ORM

CAPITULO V

Análisis, Diseño e Implementación del "Sistema de Gestión de Información Odontológica para el Departamento de Bienestar Universitario de la UTN"

CAPITULO VI

Conclusiones y Recomendaciones

Figuras

Capítulo I

Capítulo II

Capítulo III

Capítulo V

Tablas

Capítulo V

INTRODUCCION

Los sistemas de computación están cada vez más integrados a las necesidades de la empresa moderna, ya no sólo para modelar procesos manuales sino también aprovechando la inmensa capacidad de análisis de información disponible. A medida que la complejidad de estos sistemas crece, es crucial que los componentes que modelan la lógica sean aislados de las distintas tecnologías utilizadas en la solución.

En sus comienzos, la Ingeniería de Software atacó esta problemática separando la naturaleza de los datos de sus procesos asociados. Una herencia de este principio son las bases de datos relacionales, verdaderos repositorios donde se mantienen esquemas modelando los datos y un lenguaje propio para manipularlos.

La programación orientada a objetos rompe con esta separación y fuerza el igual tratamiento de los datos y los procedimientos, basándose en los principios de encapsulación y ocultamiento de la información. Estos lenguajes proveen facilidades muy primitivas para que los objetos sobrevivan a una ejecución del programa, característica esencial de una aplicación empresarial.

En la actualidad la industria del software se enfrenta con los problemas inherentes causados por la integración de estas dos tecnologías: los modelos orientados a objetos y las bases de datos relacionales, distribuyendo así las responsabilidades de modelar la lógica y persistir los objetos. Esta integración impone verdaderos retos al momento de la construcción, ya que si bien comparten algunas características, existen disparidades en la representación y manejo de la información que imponen limitaciones e introducen costos de desarrollo y mantenimiento.

La presente tesis tiene como objetivo el estudio de la persistencia de objetos Java en una base de datos relacional, utilizando para ello una de las soluciones disponibles para esta integración denominada ORM (Mapeo Objeto – Relacional), en donde la persistencia se realiza de forma totalmente automática y transparente a la lógica de la aplicación. También se incluye el análisis, diseño e implementación del Sistema de Gestión de Información Odontológica para el Departamento de Bienestar Universitario de la UTN. Todo este desarrollo se enmarca en un ámbito de ejecución Web con requerimientos clásicos de una aplicación empresarial.

CAPITULO I

Persistencia de Objetos

"El mundo está en las manos de aquellos que tienen el coraje de soñar y correr el riesgo de vivir sus sueños".

Paulo Coelho

1.1 Introducción

Cuando abordamos el desarrollo de una aplicación específicamente una orientada a objetos, uno de los primeros requerimientos que debemos resolver es la integración con una base de datos relacional para guardar, actualizar y recuperar la información que utiliza nuestra aplicación.

En este capítulo se presenta los fundamentos, problema y la alternativa de solución común para la persistencia de Objetos en base de datos relacionales.

1.2 ¿Qué es Persistencia de Objetos?

Podemos encontrar diferentes definiciones del término persistencia, según distintos puntos de vista y autores. Veamos dos que con más claridad y sencillez, concretan el concepto de persistencia de objetos.

La primera definición dice así: "La persistencia de objetos significa que los objetos individuales pueden sobrevivir al proceso de la aplicación; pueden ser guardados a un almacén de datos y ser reconstruidos más tarde." [LIB01]

La otra definición dice: Se llama persistencia de objetos a su capacidad para guardarse y recuperarse desde un medio de almacenamiento. [WWW01]

En definitiva la persistencia de objetos es la capacidad que tienen los objetos de sobrevivir al proceso que los creó; permitiendo al programador almacenar, transferir, y recuperar el estado de los objetos.

1.3 Métodos de Persistencia de Objetos

En la actualidad podemos identificar tres formas usuales de persistir objetos: [LIB02]

- Serialización.
- Bases de Datos Orientadas a Objetos (ODBMS).
- Bases de Datos Relacionales.

1.3.1 Serialización

La serialización es el proceso de convertir un objeto en una secuencia de bytes para conservarlo en memoria, una base de datos o un archivo. Su propósito principal es guardar el estado de un objeto para poder crearlo de nuevo cuando se necesita. El proceso inverso se denomina deserialización. [WWW03]

El siguiente ejemplo muestra el proceso total de serialización.

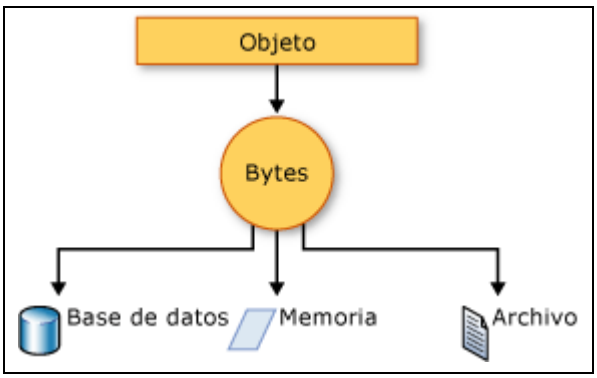


Figura 1.1 Funcionamiento de Serialización. Fuente: [WWW03]

El objeto se serializa en una secuencia que, además de los datos, contiene información sobre el tipo de objeto, como la versión, referencia cultural y nombre de ensamblado. Esa secuencia se puede almacenar en una base de datos, un archivo o en memoria.

La serialización permite al desarrollador guardar el estado de un objeto y volver a crearlo cuando es necesario, y proporcionar almacenamiento de objetos e intercambio de datos. A través de la serialización, un desarrollador puede realizar acciones como enviar un objeto a una aplicación remota por medio de un servicio Web, pasar un objeto de un dominio a otro, pasar un objeto a través de un firewall como una cadena XML[1] o mantener la seguridad o información específica del usuario entre aplicaciones.

Sin embargo no soporta transacciones, consultas o acceso compartido a los datos entre usuarios múltiples y se la utiliza sólo para proporcionar persistencia en aplicaciones simples o en entornos empotrados que no pueden gestionar una base de datos de forma eficiente.

[1] XML: Lenguaje de Marcado Extensible

Es evidente que, dada la tecnología actual, la serialización como método de persistencia es insuficiente para la alta concurrencia web y aplicaciones empresariales.

1.3.2 Base de Datos Orientadas a Objetos

Un ODBMS, o base de datos orientada a objetos, proporciona un método transparente para la persistencia. Permite consultar y trabajar con objetos directamente. [LIB04]

La persistencia transparente se refiere a la habilidad de manipular directamente, sin traducción o conversión, datos almacenados en la base utilizando un entorno de programación basado en objetos como Smalltalk, Java o C#. Esto contrasta con los RDBMSs donde se utiliza un sublenguaje como SQL o una interfaz de operación como ODBC o JDBC, luego, se utiliza código adicional para hacer la conversión a objetos.

En la siguiente figura se muestra el almacenamiento de objetos.

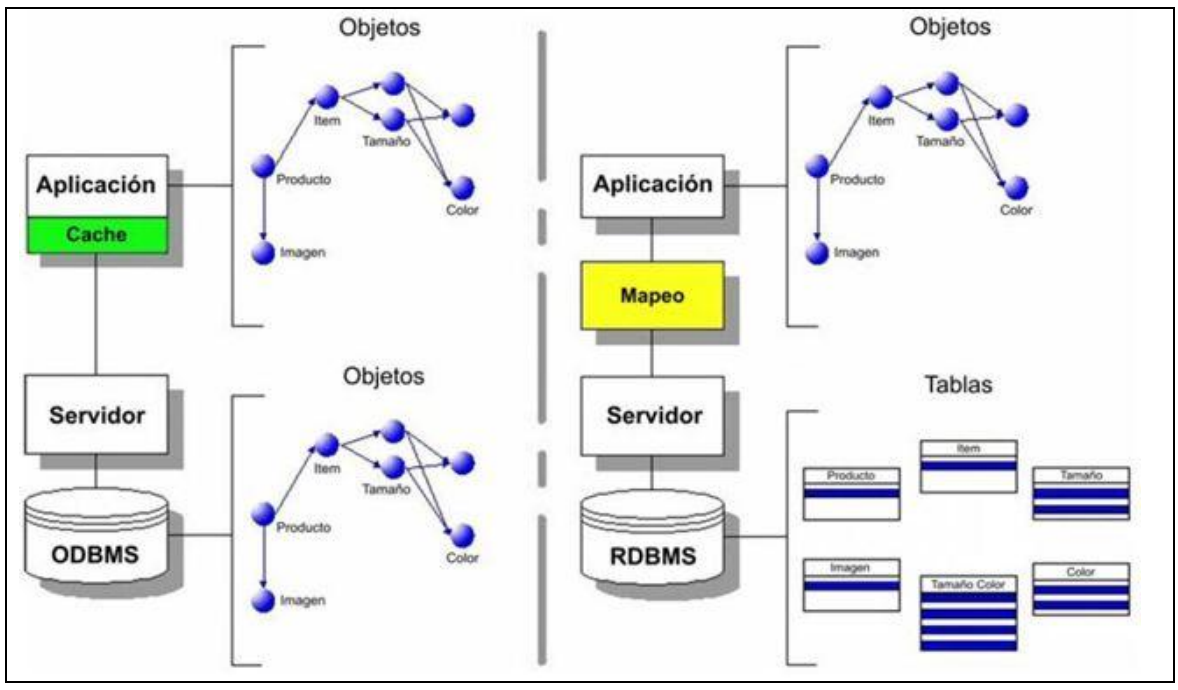


Figura 1.2 Almacenamiento de objetos vs Mapeo de objetos. Fuente: [WWW04]

La mayoría de los ODBMSs implementan un esquema de persistencia por capacidad de alcance. Ello significa que cualquier objeto referenciado por un objeto persistente también es persistido. Normalmente el programador puede especificar la profundidad de operación de este esquema en un árbol de objetos. De esta manera, conjuntos completos de objetos pueden ser almacenados y recuperados con una sola llamada; el ODBMS maneja

automáticamente los detalles de mantenimiento de las referencias cuando se guardan y recuperan objetos. [WWW04]

Las bases de datos orientadas a objetos son quizás la forma más sencilla de persistir un modelo de objetos, aunque el mercado de las tecnologías de bases de datos orientadas a objetos es aún pequeño e inestable comparado con el mercado de las bases de datos relacionales.

1.3.3 Bases de Datos Relacionales

1.3.3.1 Modelo Relacional

Una descripción muy sencilla y directa del modelo relacional, es aquella en donde se define que la responsabilidad de las bases de datos relacionales es modelar la información basándose en relaciones definidas en conjuntos finitos de valores llamados dominios. [WWW05]

Una relación es un conjunto de listas ordenadas de valores llamadas tuplas. Cada tupla es un elemento del producto cartesiano de dominios. Cada ocurrencia de un dominio en la definición de una relación se denomina atributo, y el mismo dominio puede llegar a repetirse dentro de ella. Los dominios y las relaciones son implementados como tablas con m filas y n columnas. Cada fila corresponde a una tupla y cada columna a un atributo relacional. Por eso es que a lo largo de esta tesis se usará el concepto de tabla, columna y fila en lugar de relación, atributo y tupla.

Cada columna de una tabla tiene un tipo y un nombre para poder ser referenciada sin importar su posición relativa. El tipo de una columna se limita a un conjunto pequeño de tipos predefinidos como integer, varchar o date. Las tablas se definen usualmente con restricciones impuestas a sus filas para evitar la duplicación de datos o dependencias cruzadas. Una restricción sobre una tabla es requerir que cada fila sea identificable unívocamente a partir de un subconjunto de sus columnas. A este subconjunto se lo denomina llave primaria. Cuando esta llave esta compuesta por una única columna se la denomina llave simple sino en caso de contar con más de una columna se dice que la llave es compuesta.

El acceso a los datos se realiza a través de un conjunto básico de operaciones: selección, proyección, producto, join, unión, intersección y diferencia. Estas operaciones se expresan

a través de un lenguaje denominado SQL (Lenguaje de Consultas Estructurado) usado para guardar, recibir y modificar información en la base de datos.

Para recibir información se debe realizar una selección (caracterizada por la consulta SELECT en SQL). La selección de las filas permite especificar ciertas condiciones para los atributos acompañada generalmente con una proyección que significa que sólo un subconjunto de las columnas son seleccionadas. Para consultas más complejas se utiliza el operador JOIN que crea una tabla temporal con conjuntos de columnas pertenecientes a dos o más tablas. El resultado de una selección es generalmente un conjunto de filas o RecordSet. Para recorrer este conjunto los lenguajes de programación brindan un cursor y operaciones de navegación encapsuladas en librerías.

De las tres formas de persistencia, sólo las bases de datos relacionales han demostrado ser escalables, robustas y lo suficientemente estándares para las aplicaciones empresariales. No obstante cuando se quiere persistir los objetos utilizando una de ellas, se puede observar que hay un problema de compatibilidad entre el paradigma de la Orientación a Objetos y el modelo relacional, la también llamada diferencia o impedancia objeto-relacional.

1.4 Impedancia Objeto-Relacional

"Impedancia Objeto-Relacional", se define como un conjunto de dificultades técnicas que surgen cuando una base de datos relacional se usa en conjunto con un programa escrito bajo el paradigma de la Orientación a Objetos. [LIB03]

La siguiente figura muestra algunas diferencias.

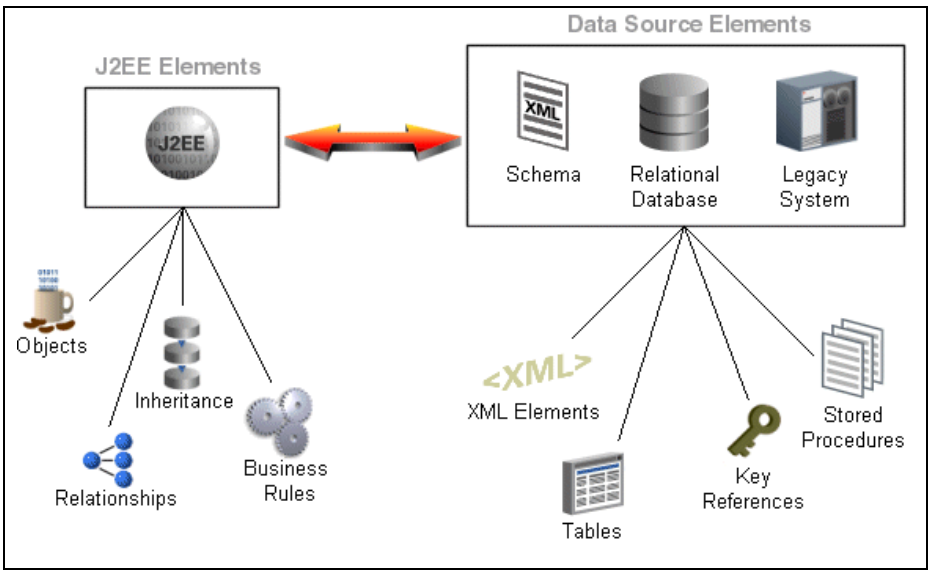


Figura 1.3 Impedancia Objeto-Relacional. Fuente: [WWW19]

Un ejemplo claro de esta impedancia se observa en el hecho que en el mundo de la programación orientada a objetos, se tiene un claro sentido de la pertenencia, a cada objeto le pertenecen sus correspondientes atributos; por ejemplo para el objeto Agenda Telefónica podríamos especificar como atributos a una colección de objetos llamados "persona", en la que a cada persona le corresponde su correspondiente atributo "teléfono", al transformar esto hacia el mundo relacional se ocuparía más de una tabla para almacenar la información, este simple hecho, hace notar que las tablas del modelo relacional son inconscientes de cómo están relacionadas con otras tablas a un nivel fundamental, puesto que aún cuando posean constraints para definir sus relaciones, para reconstruir el objeto originalmente persistido se debe construir un query, y dicho query debe especificar explícitamente como se relacionan las tablas entre sí, con esto se demuestra además que el lenguaje SQL a pesar de los constraints se mantiene inconsciente de las relaciones que a nivel de objeto poseen las tablas entre ellas.

Así como lo anteriormente expuesto se pueden enumerar distintos problemas que surgen entre los dos modelos:

Reglas de Acceso: En el modelo relacional los atributos pueden ser accesados y/o modificados a través de operadores relacionales predefinidos, mientras que en el modelo orientado a objetos, se permite que cada clase defina la forma en que serán alterados los atributos así como la interfase que ocupará para ello.

Ataduras del Esquema: Los objetos del modelo de la POO[2], no deben seguir ningún esquema en cuanto a que atributos deben o pueden tener, puesto que son definidos por el programador, mientras que las tablas deben seguir el esquema entidad-relación.

Identificador único: Las llaves primarias de una fila tienen generalmente una forma de poder representarse como texto visible, mientras que los objetos no requieren un identificador único externamente visible.

Estructura vs Comportamiento: La orientación a objetos se concentra primordialmente en asegurar que la estructura del programa sea razonable (entendible, extensible, reusable, segura, etc), mientras que los sistemas relacionales ponen el énfasis en tipo de comportamiento que el sistema tendrá una vez en producción (eficiencia, adaptabilidad, rapidez, etc.). Los métodos de la POO asumen que el principal usuario del código orientado a objetos y sus beneficios es el desarrollador de aplicaciones, mientras que el modelo relacional enfatiza que la forma en que los usuarios finales perciben el comportamiento del sistema es mucho más importante.

De todo esto surge la necesidad de utilizar algún mecanismo para integrar la información contenida en nuestros objetos con los datos almacenados en la base de datos relacional. Esto típicamente se logra a través de una capa de traducción objeto – relacional.

A continuación se explica la arquitectura de una aplicación y una de las alternativas más comunes para la transformación de objetos a bases de datos relacionales.

1.5 Arquitectura basada en capas

Para organizar una aplicación empresarial, la industria del software ha convergido en una Arquitectura basada en Capas, dividiendo el sistema en tres capas básicas: la capa de presentación, la capa de lógica de dominio y la capa de persistencia, como se muestra en la siguiente figura. [WWW05]

[2] POO: Programación Orientada a Objetos.

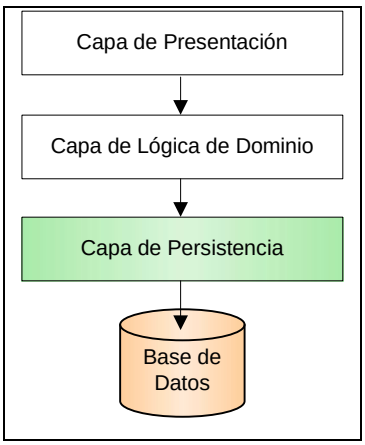


Figura 1.4 La Capa de Persistencia es la base de una Arquitectura en Capas.

El principio detrás de esta arquitectura es que cada capa dependa sólo de los elementos contenidos en ella o en las capas situadas por debajo, teniendo responsabilidades bien definidas y evitando cualquier tipo de acoplamiento con las capas superiores.

Capa de Presentación

Maneja la interacción entre el usuario y la aplicación. A veces el nombre presentación presta a pensar en sólo salida de la aplicación, pero en realidad maneja la interacción en ambas direcciones. En algunas ocasiones el usuario puede ser otro sistema comunicándose por ejemplo a través de un servicio remoto.

Esta comunicación se hace especialmente notoria en ambientes Web, donde la capa de presentación no sólo tiene que crear documentos entendibles por los usuarios sino manejar los mensajes enviados por el browser como consultas o datos de formularios. Para esto se suele utilizar un esquema de Controladores y Vistas, donde los controladores son el eje entre las capas inferiores y las vistas, el patrón de diseño Model View Controller es un ejemplo popular de esta forma de estructurar la aplicación. Algunos autores dividen los controladores en una capa separada llamada Capa de Aplicación.

Capa de Lógica de Dominio

Representa conceptos de negocio como reglas o estados. Lo que distingue a esta lógica por sobre el resto de la aplicación es que mantiene los conceptos centrales del proceso, siendo muchas veces la ventaja competitiva por sobre el resto de los productos. Es por ello que, a pesar de que por lo general sólo constituye un conjunto de módulos pequeños comparado el resto de la capas, suele tener mayores requerimientos como tests

automáticos o revisiones. Esta importancia se realza en las aplicaciones empresariales, la lógica irregular y propensa a cambios requiere un tratamiento especial.

Capa de Persistencia

Esta capa brinda servicios para sincronizar la capa de lógica con un medio de almacenamiento. Para esto se deben identificar los objetos en memoria que deben sobrevivir a la ejecución del programa teniendo así una persistencia de largo plazo.

Dependiendo del tipo de aplicación existen distintos requisitos para esta capa. Por ejemplo en una aplicación de diseño CAD generalmente cuando se edita un documento se trae un gran conjunto de información a memoria desde el medio de almacenamiento ya que traer subconjuntos de información haría que la aplicación tenga un tiempo de respuesta pobre. En cambio en una aplicación empresarial generalmente las operaciones se concentran sólo en un grafo limitado de objetos, por lo que traer toda la información disponible es un gasto inaceptable. Para esto debe existir un especial interés en la optimización entre la cantidad de información que se trae a memoria y la realmente utilizada.

1.6 Persistencia de Objetos Manual

Java Database Connectivity (JDBC)

Java DataBase Connectivity es el API[3] de Java que define cómo una aplicación accederá a una base de datos, independientemente del motor de base de datos. Se lo realiza mediante ejecuciones de sentencias SQL. [WWW01]

[3] API: Interfaz de Programación de Aplicaciones. Representa una interfaz de comunicación entre componentes software. http://es.wikipedia.org/wiki/Application_Programming_Interface

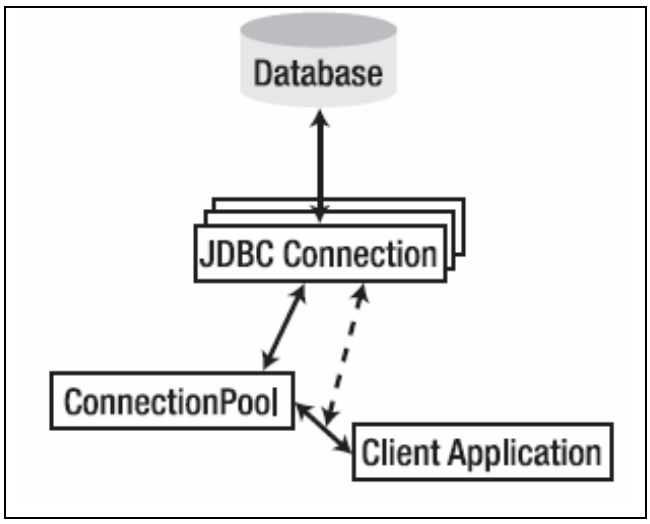


Figura 1.5 Acceso a datos mediante JDBC. [LIB06]

No obstante, es necesario gestionar explícitamente los valores de los campos y su proyección en tablas de una base de datos relacional.

Por tanto:

- Hay que tratar con dos modelos de datos, lenguajes y paradigmas de acceso a los datos, muy diferentes (Java y SQL).

- El esfuerzo necesario para implementar el mapping entre el modelo relacional y el modelo de objetos es demasiado grande:

  - Muchos desarrolladores nunca llegan a definir un modelo de objetos para sus datos.

  - Se limitan a escribir código Java procedural para manipular las tablas de la base de datos relacional subyacente.

  - Se pierden los beneficios y ventajas del desarrollo orientado a objetos.

Esta forma de trabajo es cada vez menos frecuente por el alto costo de desarrollo y mantenimiento.

CAPITULO II

ORM (Mapeo Objeto Relacional)

"Para empezar un gran proyecto, hace falta valentía. Para terminar un gran proyecto, hace falta perseverancia".

2.1 Introducción

La mayor parte de las aplicaciones orientadas a objetos precisan de la implementación de mecanismos que permitan a los objetos mantenerse vivos tras la finalización del proceso que les dio vida. El problema de la persistencia de objetos Java ha dado lugar a utilizar alternativas que proporcionan al programador una manera sencilla, automática y transparente basada en bases de datos relacionales, de incluir esa funcionalidad en sus desarrollos denominada, ORM (Mapeo Objeto Relacional).

En este capítulo discutiremos el concepto de ORM, componentes, principales requerimientos para un buen framework ORM y herramientas.

2.2 ¿Qué es ORM?

El mapeo Objeto/Relacional es "la persistencia automatizada y transparente de las tablas en una Base de Datos relacional, usando metadatos[4] que definen el mapeo entre los objetos y la Base de Datos". [LIB01]

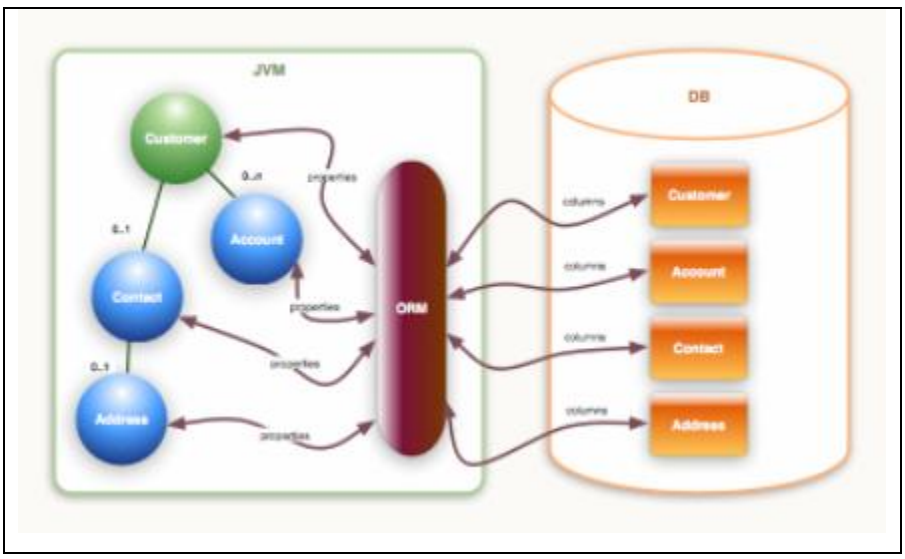


Figura 2.1 Mapeo Objeto Relacional

En esencia, ORM transforma datos de una representación en otra.

[4] Metadatos: Datos que describen otros datos. http://es.wikipedia.org/wiki/Metadatos

Por tanto, ORM es una técnica que se utiliza para poder ligar las bases de datos y los conceptos de orientación a objetos creando ”bases de datos virtuales”, es decir la aplicación desde dentro utiliza frameworks[5], los mismos que son intermediarios entre la base de datos relacional y la aplicación totalmente orientada a objetos. [WWW06]

Se considera ORM a cualquier capa de persistencia que proporcione autogeneración de SQL a través de metadatos (generalmente XML). No se considera ORM a una capa de persistencia creada por el propio desarrollador.

2.3 Componentes

Una solución ORM está formada por cuatro partes:

1. Una API que posibilite la realización de operaciones de creación, actualización y borrado sobre objetos de clases persistentes.
2. Un lenguaje o API para poder especificar consultas sobre dichas clases.
3. Una opción para especificar mapeo de metadatos.
4. Una técnica para que la implementación del ORM pueda llevar a cabo búsquedas, asociaciones u otras funciones de optimización. [LIB01]

2.3.1 JDBC: Componente indispensable para los ORM

Sin tener en cuenta la solución de mapeo O/R[6] que se vaya a utilizar para comunicarse con la base de datos relacional, todos ellos dependen de JDBC. Teniendo en cuenta que la mayor parte de las aplicaciones se comunican con bases de datos relacionales, es fundamental considerar cada uno de los niveles del software (desde el código del programa hasta la fuente de datos) para asegurar que el diseño de persistencia O/R sea óptimo.

Tal y como se verá más adelante, cada una de las soluciones de mapeo O/R tiene una dependencia particular en el driver JDBC para poder comunicarse con la base de datos de una forma eficiente. Si el driver JDBC que va a participar en la comunicación no es óptimo, la posible gran eficiencia de cualquier framework quedará debilitada. Por tanto,

[5] Framework: Estructura de soporte definida, en la cual otro proyecto de software puede ser organizado y desarrollado.
[6] O/R: Objeto Relacional.

elegir el driver JDBC que mejor se adapte a la aplicación es esencial a la hora de construir un sistema eficiente en el que participe una solución de mapeo O/R.

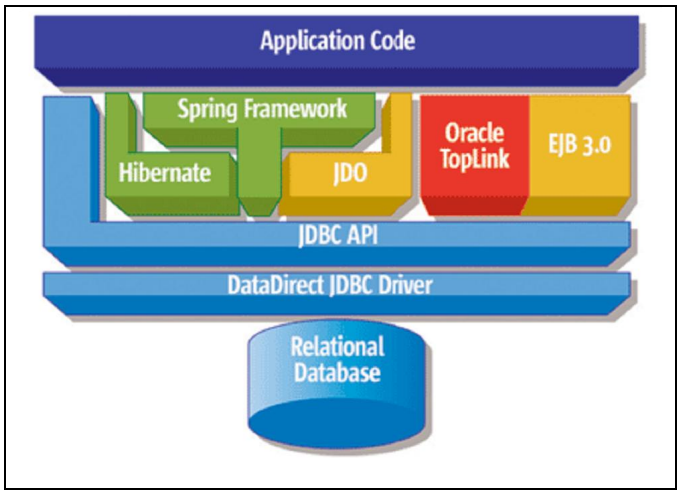


Figura 2.2 Interacción de componentes. Fuente: [WWW07]

La figura muestra una representación de los diferentes mecanismos o soluciones de mapeo O/R y cómo se relacionan con el código de la aplicación y con los recursos de datos relacionados. Esto muestra claramente la función crítica que desempeña el driver JDBC puesto que está situado en la base de cada uno de los frameworks.

La eficiencia del driver JDBC tiene importantes consecuencias en el comportamiento de las aplicaciones. Cada mecanismo de mapeo O/R es completamente dependiente del driver, sin tener en cuenta el diseño de la API del framework que esté expuesta al código fuente de la aplicación.

Como los mecanismos de mapeo O/R generan llamadas eficientes para acceder a la base de datos, mucha gente defiende que la importancia del driver JDBC se ha visto reducida. Sin embargo, como en cualquier arquitectura, la totalidad de eficiencia en una aplicación siempre estará afectada por el nivel más débil del sistema.

Independientemente del código JDBC generado, los mecanismos de mapeo O/R son incapaces de controlar cómo los drivers interactúan con la base de datos. Entonces la eficiencia de la aplicación depende en gran parte de la habilidad que tenga el driver del nivel JDBC para mover todos los datos manejados entre la aplicación y la base de datos.

Aunque hay múltiples factores que considerar a la hora de elegir un driver JDBC, seleccionar el mejor driver JDBC posible basándose en comportamiento, escalabilidad y fiabilidad es la clave para obtener el máximo beneficio de cualquier aplicación basada en un framework O/R. [WWW07]

2.4 Mapeando objetos a RDBMS

Es muy común encontrar aplicaciones orientadas a objetos manipulando datos en bases de datos relacionales, mediante el uso de técnicas de mapeo por metadatos. [WWW09].

Dentro de las consideraciones a tener en cuenta encontramos:

- Un atributo o propiedad podría mapearse a cero o más columnas en una tabla de un RDBMS.
- Algunos atributos o propiedades de los objetos no son persistentes (calculados por la aplicación).
- Algunos atributos de un objeto son también objetos (Cliente --> Dirección) y esto refleja una asociación entre dos clases que deben tener sus propios atributos mapeados.

En principio identificaremos dos tipos de metadatos de mapping[7]: property mapping y relationship mapping.

- Property mapping: Mapping que describe la forma de persistir una propiedad de un objeto.

- Relationship mapping: Mapping que describe la forma de persistir una relación (asociación, agregación, o composición) entre dos o mas objetos.

El mapping más simple es un property mapping de un atributo simple a una columna de su mismo tipo.

[7] Mapping: Proceso de conectar objetos/atributos a tablas/columnas.

Pero, excepto para casos triviales, no siempre es tan sencillo como mapear una clase a una tabla en una relación uno a uno donde cada propiedad de la clase corresponde a un campo de una tabla en la base de datos.

Por otra parte para soportar la conversión entre los dos modelos es necesario incorporar al modelo OO (orientado a objetos) lo que se conoce como información shadow. Esto es, cualquier dato extra (agregado al modelo OO original y sin significado para el negocio) que necesiten mantener los objetos para poder persistirse, como por ejemplo identificación de las propiedades que corresponden a la clave primaria en la base de datos, marcas que permitan el control de intentos de modificación concurrente de los datos en la base (timestamps o números de versión de los datos), atributos especiales que indiquen si el objeto ya fue persistido (para determinar si se usa INSERT o UPDATE), etc.

Los metadatos que configuran el mapeo pueden llegar a ser muy complejos y es necesario comprender muy bien que se está configurando y las implicaciones de usar diferentes alternativas.

2.4.1 Mapeando relaciones de herencia

Como ya comentamos, los RDBMS[8] no soportan herencia en forma nativa y por lo tanto es necesario mapear.

La decisión fundamental está en la respuesta a la pregunta ¿cómo organizo los atributos heredados en el modelo de datos?

Para ilustrar la problemática, y las distintas estrategias, utilizaremos el ejemplo presentado por Ambler para mostrar de forma sencilla la evolución de un modelo de información Orientado a Objetos. [LIB03]

Consideremos el siguiente modelo orientado a objetos y una posible evolución del mismo al incorporar la clase Ejecutivo como especialización de la clase Empleado en una etapa posterior.

[8] RDBMS: Base de Datos Relacional.

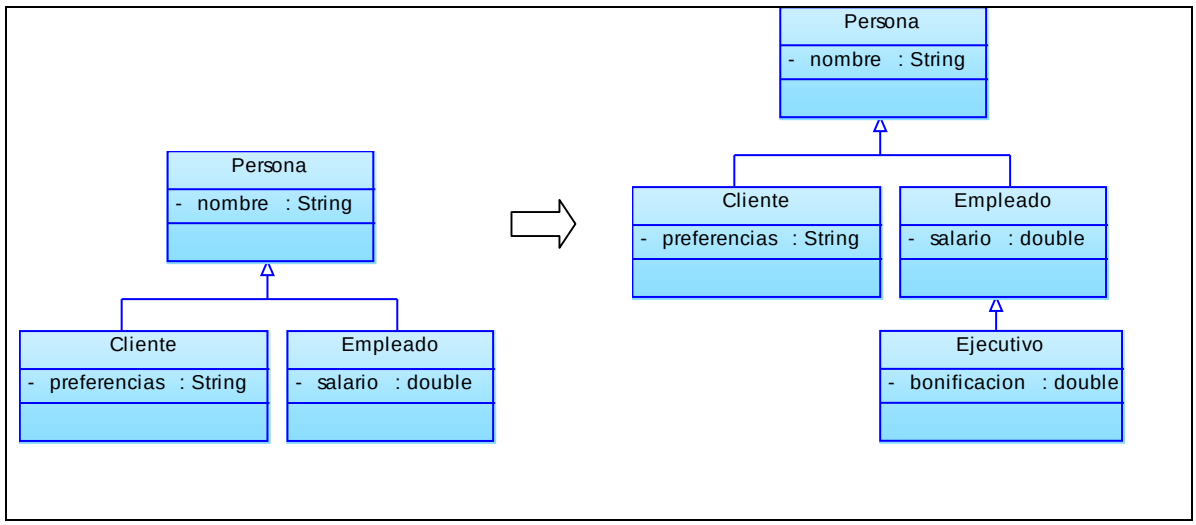


Figura 2.3 Ejemplo Modelo Orientado a Objetos

Entre las estrategias posibles para mapear las relaciones de herencia encontramos:

- Mapear la jerarquía de clases a una sola tabla.
- Mapear cada clase concreta a su propia tabla.
- Mapear cada clase a su propia tabla.

Veremos las básicas de cada una de ellas y usaremos el ejemplo para presentar sus fortalezas y debilidades.

2.4.1.1 Estrategia 1: Mapear la jerarquía de clases a una sola tabla

Se mapean todos los atributos de todas las clases de la jerarquía a una sola tabla.

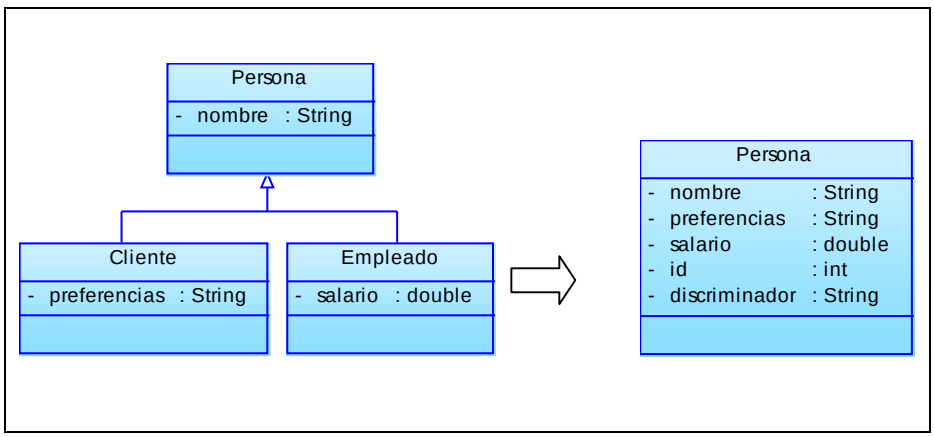


Figura 2.4 Mapeo de todas las clases de herencia a una tabla

En el ejemplo para el modelo original se crea una tabla conteniendo los campos de todas las clases:

```
tabla PERSONA [
    ID : Clave primaria.
    DISCRIMINADOR: VARCHAR
    NOMBRE: VARCHAR
    PREFERENCIAS: VARCHAR
    SALARIO: DECIMAL
]
```

Vemos que además es necesario agregar en la clase Persona un campo para la clave primaria (información shadow) y un campo discriminador en la tabla que permita determinar de qué clase es la instancia persistida. El discriminador puede contener simplemente el nombre completo de la clase concreta del objeto que se ha grabado.
Por ejemplo, si nuestras clases están en el paquete test y grabó una instancia de Empleado, el campo discriminador contendrá el valor "test.Empleado", lo cual me permite reconstruir de forma eficiente el objeto desde la base de datos.

Ahora, que sucede con la evolución del modelo. Al agregar la clase Ejecutivo es necesario agregar el campo "bonificacion: decimal" a la tabla. Y definirlo como nullable!! porque los registros correspondientes a Cliente o Empleado no definen un valor para la bonificación (no tiene sentido).

Ventajas

- Es un enfoque simple que permite agregar nuevas clases de forma sencilla (solo agregar columnas).
- Soporta polimorfismo mediante inspección del valor del campo discriminador.
- Provee acceso muy eficiente a los datos (Sin JOINS).

Desventajas

- Alto acoplamiento con la jerarquía de clases ya que todas se persisten en la misma tabla. Un cambio en la jerarquía afecta a todas.
- Potencial desperdicio de espacio en la base de datos, por campos irrelevantes a la clase. Por ejemplo al tener que grabar un valor por defecto para la bonificación al grabar un Cliente.

- Impide la definición de restricciones dado que hay campos que deben quedar en null o en un valor por defecto si no corresponden a la clase de la instancia que se está grabando.
- Si la jerarquía es muy grande obtengo tablas muy grandes (con muchos campos).

¿Cuándo usar?

Buena estrategia para clases simples o jerarquías poco profundas, dónde los tipos de la jerarquía no se solapan.

2.4.1.2 Estrategia 2: Mapear cada clase concreta a su propia tabla

Se crea una tabla por cada clase concreta, redundando todos los atributos y no se mapean clases abstractas.

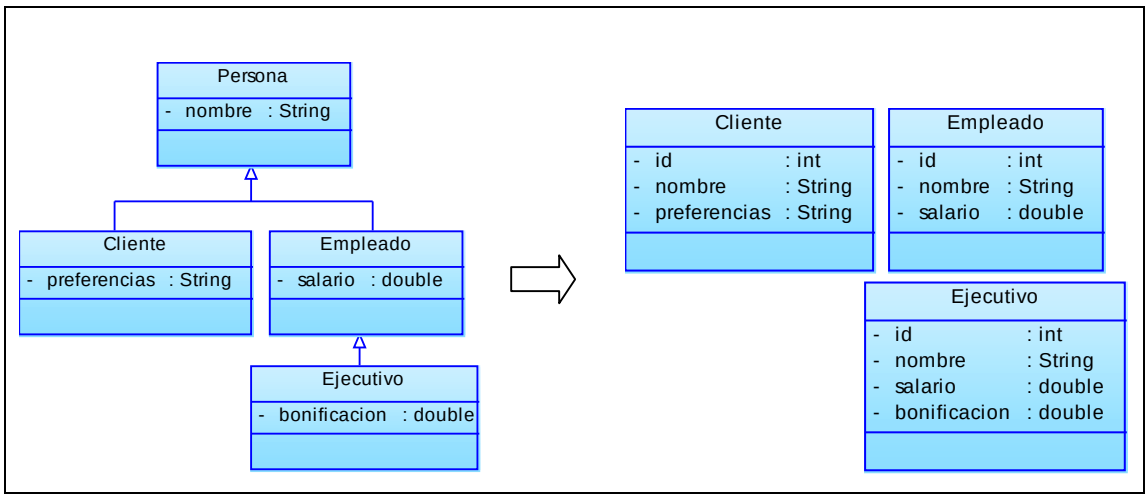


Figura 2.5 Mapeo de todas las clases concretas a una tabla independiente

En el ejemplo: dado que la clase Persona es abstracta no creamos una tabla para la misma. Creamos las siguientes tablas:

tabla CLIENTE [ID : Clave primaria. NOMBRE: VARCHAR PREFERENCIAS: VARCHAR]	tabla EMPLEADO [ID : Clave primaria. NOMBRE: VARCHAR SALARIO: DECIMAL]

Nuevamente es necesario agregar un campo para la clave primaria en la clase Persona (información shadow).

Al evolucionar el modelo se agrega una tabla nueva:

```
tabla EJECUTIVO [
    ID : Clave primaria.
    NOMBRE: VARCHAR
    SALARIO: DECIMAL
    BONIFICACION: DECIMAL
]
```

A simple vista podemos observar que un Empleado ascendido a Ejecutivo requerirá copiar sus datos de la tabla Empleado a la tabla Ejecutivo, incluso es posible que sea necesario mantener la información redundante en las dos tablas para no afectar los procesos de negocios que manipulan Empleado.

Ventajas

- Reportes fáciles de obtener ya que todos los datos necesarios sobre una clase particular se encuentran almacenados en una sola tabla.
- Buena performance para acceder a datos de un objeto simple (sin asociaciones no es necesario el JOIN).

Desventajas

- Poco escalable, si se modifica la clase base se debe modificar en todas las tablas de las clases derivadas para reflejar ese cambio (Ej. Agregar una columna direccion en la Persona implica modificar todas las tablas).
- Actualización compleja, si un objeto cambia su rol (se contrata un Cliente como Empleado) se le asigna un nuevo OID[9] y es necesario copiar todos sus datos a la tabla apropiada.
- Dificultad para soportar múltiples roles y mantener integridad de datos.

¿Cuándo usar?

- Cuando la jerarquía de clases es muy estable (las clases raramente cambian).
- Cuando no existe solapamiento de tipos.

[9] OID: Identificador de objeto. Permite relacionar los objetos y evitar duplicados.

2.4.1.3 Estrategia 3: Mapear cada clase a su propia tabla

Se crea una tabla por cada clase de la herencia, sin redundar los atributos de las superclases.

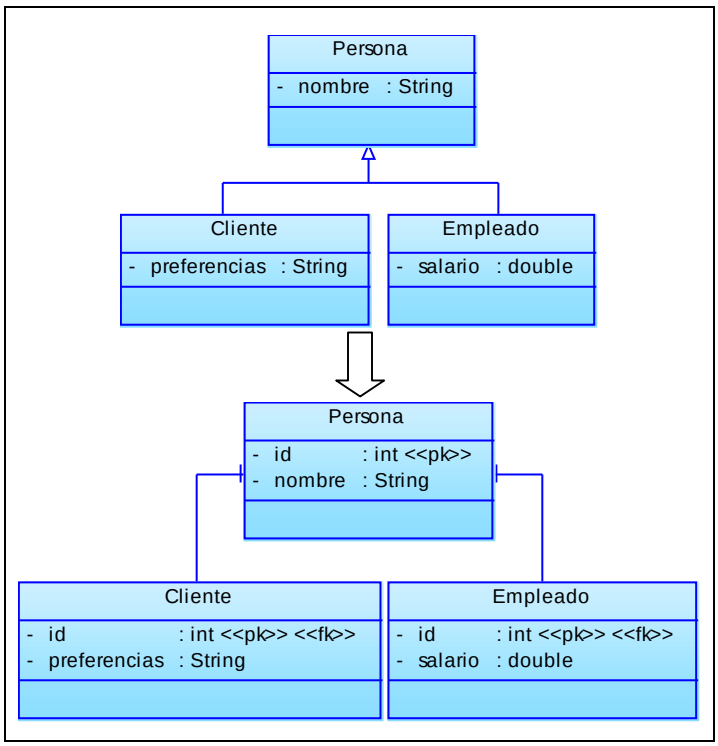


Figura 2.6 Mapeo de todas las clases de herencia a su propia tabla

Para recuperar una instancia es necesario recurrir a JOINS por clave primaria que en las tablas de las subclases son claves foráneas.

En el ejemplo:

```
tabla PERSONA [
    ID : Clave primaria.
    NOMBRE: VARCHAR
]
tabla CLIENTE [
    ID : Clave primaria. // Foreign key a PERSONA.ID
    PREFERENCIAS: VARCHAR
]
tabla EMPLEADO [
    ID : Clave primaria. // Foreign key a PERSONA.ID
    SALARIO: DECIMAL
]
```

Nuevamente es necesario agregar un campo para la clave primaria en la clase Persona (información shadow).

Al evolucionar el modelo se agrega una tabla nueva:

```
tabla EJECUTIVO [
    ID : Clave primaria. // Foreign key a EMPLEADO.ID
    BONIFICACION: DECIMAL
]
```

Para recuperar los datos de un Ejecutivo es necesario realizar el JOIN entre Persona, Empleado y Ejecutivo.

Ventajas

- Fácil de entender, porque es un mapeo uno a uno.
- Soporta muy bien el polimorfismo, ya que tiene almacenado los registros en la tabla correspondiente.
- Fácil de modificar superclases y agregar nuevas subclases (implica modificar o agregar una sola tabla).
- Modelo muy normalizado en el cual el tamaño de los datos crece en proporción directa al número de objetos (instancias) persistidos.

Desventajas

- Muchas tablas en la base de datos (una por clase + tablas de relación).
- Menor perfomance, pues se necesita leer y escribir sobre varias tablas.
- Reportes rápidos difíciles de armar, debido a la necesidad de JOINS, a menos que se agreguen vistas para simular las tablas deseadas.

¿Cuándo usar?

- Cuando hay solapamiento de tipos.
- Cuando las modificaciones a las clases son frecuentes.
- Cuando se quiere evitar la redundancia de datos.

2.4.2 Mapeando asociaciones

Existen tres tipos de asociaciones entre objetos:

- Asociación
- Agregación
- Composición

Por ahora las trataremos igual, aunque existen diferencias en el manejo de las restricciones. Las cuales se pueden clasificar de acuerdo a dos categorías ortogonales:

En base a la multiplicidad:
- One-to-one
- One-to-many (many-to-one según desde donde se lea).
- Many-to-many

En base a la dirección:
- Unidireccionales
- Bidireccional

En la base de datos las relaciones se mantienen mediante el uso de Foreign Keys:
- One-to-one: FK implementada en una de las tablas.
- One-to-many: FK desde la “one table” a la “many table”
- Many-to-many: Es necesario incorporar una tabla asociativa (o de relación)

En cuanto a la direccionalidad, todas las relaciones en la base de datos relacional son efectivamente bidireccionales.

Una configuración importante es si los objetos asociados a otro se leen automáticamente al leer el mismo. Cargar un objeto y todos sus objetos asociados automáticamente en memoria puede llegar a impactar negativamente en el rendimiento del sistema. Para esto se usan técnicas de lazy loading que consisten en cargar los objetos asociados a demanda a medida que son solicitados. Si una asociación nunca es solicitada entonces nunca se lee de la base de datos. [WWW09]

Otra consideración importante tiene que ver con el almacenamiento de colecciones secuenciales de objetos asociados a un objeto padre. En este caso es necesario almacenar en la base la información (por ejemplo un ordinal numérico) que permita recuperar estos objetos en la secuencia correcta.

2.5 Diferentes Técnicas de Mapeo

En el mercado existe una gran variedad de herramientas y productos ORM. Conozcamos algunas de las técnicas más utilizadas por estas herramientas. [WWW08]

Mapeo directo sin identidad: El desarrollador crea clase con atributos de cuyo nombre se derivan las columnas de las tablas en la base de datos (viceversa). Las herramientas examinan estas clases y generan los queries correspondientes. No se mantiene identidad basada en la llave primaria. Esto significa que es posible que en memoria existan dos objetos diferentes que correspondan a la misma hilera de una tabla. Si ambos objetos son actualizados, entonces se perderán los cambios en alguno de ellos. Esto implica que la lógica de la aplicación esté al pendiente de que esto no suceda.

Mapeo directo con identidad: Similar al anterior, pero se mantiene un índice (típicamente un hash table) de todos los registros residentes en memoria. De esta manera se asegura que no puedan existir dos objetos que correspondan a la misma hilera de una tabla. El inconveniente de esta estrategia es que la capa ORM debe responsabilizarse de administrar el ciclo de vida de los objetos, ya que la lógica de la aplicación no tendrá forma de saber cuando se deba eliminar un objeto compartido. Esto puede presentar complicaciones dependiendo del lenguaje de programación utilizado y los mecanismos que provea para administrar el ciclo de vida de los objetos.

Generación a partir de XML: Consiste en generar código a partir de definiciones escritas en XML. Es recomendable que el código generado no sea modificado posteriormente.

2.6 Optimizaciones de rendimiento

A nivel de la base de datos puede ser necesario cambiar el esquema, usualmente desnormalizando porciones del mismo, cambiar los tipos de las columnas clave (numéricos son más efectivos que los strings), reducir la cantidad de columnas que componen una clave, introducir índices o incluso introducir procedimientos almacenados que trasladen a la base ciertos procesos.

También puede ser necesario agregar cachés en memoria que minimicen la cantidad de accesos a la base de datos a los mínimos necesarios.

En cuanto a la optimización de los mapeos siempre hay que tener en cuenta que tengo:

- Maneras de mapear herencia.
- Maneras de mapear relaciones one to one.

- Maneras de mapear atributos de clase.

Además, hay que tener en cuenta que cada vez que se cambia la estrategia de mapping puede ser necesario cambiar el esquema de objetos, el esquema de base de datos o ambos. Por lo que la recomendación es elegir muy bien las estrategias a utilizar.

2.7 Requerimientos de la Capa de Persistencia

Una capa de persistencia encapsula el comportamiento necesario para persistir objetos. Es decir: leer, escribir y borrar objetos en el almacenamiento persistente (base de datos).

Un buen framework de persistencia debería proveer de forma relativamente sencilla:

- Soporte para diversos tipos de mecanismos de persistencia (archivos planos, RDBMS, OODBMS, etc.)
- Encapsulamiento total del mecanismo de persistencia.
- Acciones sobre múltiples objetos (asociaciones, búsquedas, etc.).
- Transacciones: planas o anidadas, locales o distribuidas.
- Extensibilidad: permitir agregar nuevas clases (entidades) de forma sencilla.
- Soportar la manipulación automática de identificadores de objetos (OIDs).
- Cursores que permitan la lectura incremental de objetos desde la base de datos.
- Soporte para proxies que habiliten las técnicas de "lazy loading".
- Registros (Record Sets) para operaciones de bajo nivel.
- Soporte para múltiples arquitecturas (Desktop, Enterprise).
- Soporte para diferentes versiones de una base de datos o diferentes proveedores en forma transparente a la aplicación.
- Soporte para el manejo eficiente de múltiples conexiones a diferentes BD (pooles de conexiones).
- Soporte para drivers de base de datos nativos y no nativos.
- Permitir ejecutar consultas SQL si es necesario
- Proveer un lenguaje para consultas Orientadas a Objetos, más naturales al modelo de información de la aplicación, y transformación automática de éstas a SQL al momento de ejecutar la consulta.

2.8 Herramientas ORM

Hay múltiples alternativas para los programadores de Java cuando se pretende trabajar con mapeadores O/R. Existen tres organizaciones o comunidades que están implicadas en el mundo de la persistencia O/R de Java de forma activa: organizaciones basadas en el estándar, comunidades open source y grupos comerciales.

Las comunidades open source incluyen importantes tecnologías, entre ellas Hibernate y el framework Spring. Las alternativas más importantes basadas en el estándar, son EJB 3.0 y JDO. Entre las implementaciones comerciales se pueden resaltar Oracle's TopLink.

2.8.1 Matriz de Selección

La siguiente tabla muestra los principales puntos que se deben considerar en el momento de seleccionar un framework de persistencia. [WWW18]

Framework	¿Cuándo? (Si la aplicación necesita)	¿Qué Beneficios?	¿Qué obligaciones?
Java Persistente API 	Simple framework de persistencia para aplicaciones Java Standard o empresariales.	• Basada en estándares. • Incorpora las mejores características de otros frameworks.	• No puede utilizarse con versiones de Java anteriores a 5.0
EJB 	Contenedor proporciona seguridad y gestión de transacciones, además de la gestión de persistencia.	• Componentes distribuidos. • Buena escalabilidad.	• Uso intensivo de recursos. • Complejo de aprender y usar. • Menos flexible.
Hibernate 	Si quiere un framework simple y flexible.	• Ninguna adquisición o cuotas de mantenimiento. • Se integra perfectamente con otros frameworks. • Fácil de aprender y usar. • Flexible: Se puede utilizar con o sin Ejes, a nivel de empresa o aplicaciones Java.	• De código abierto.

Toplink 		Tecnología madura.	• Específico de un vendedor.

2.9 Clasificación de las Aplicaciones según ORM

Dependiendo de que tanto se apoyen en ORM, podemos clasificar las aplicaciones en las siguientes categorías:

1. Relacional pura: Toda la aplicación, incluyendo la interfaz de usuario, está diseñada en base al modelo relacional. Es una aproximación válida para aplicaciones simples donde no se va a reutilizar gran cantidad de código, aunque tiene serios contratiempos como la mantenibilidad o la portabilidad.
   En esta categoría, las aplicaciones típicamente suelen hacer uso de procedimientos almacenados, con lo que se deja parte del trabajo fuera de la capa de lógica de negocio y dentro de la base de datos.

2. Mapeo de objetos ligero: En esta aproximación las entidades se representan como clases que después son mapeadas manualmente en las tablas relacionales. Se oculta el código de acceso a la Base de Datos con los patrones de diseño más utilizados. Es una aproximación con mucha aceptación, y muy válida cuando no tenemos muchas entidades. Aunque también puede haber procedimientos almacenados, en este caso, tendrían un peso mucho menor.

3. Mapeo de objetos medio: La aplicación se diseña en base a un modelo de objetos. Las sentencias SQL se generan en tiempo de ejecución a través de un framework o generadores de código. Las asociaciones entre objetos son manejadas por el mecanismo de persistencia y es posible refinar el acceso a datos a través de expresiones en un lenguaje orientado a objetos. Típicamente se maneja un caché de objetos en la capa de persistencia. La mayoría de los productos ORM y soluciones desarrolladas internamente (in-house) soportan este nivel de funcionalidad. Este nivel es adecuado para aplicaciones de complejidad media, donde es importante la portabilidad hacia diferentes RDBMS. Estas aplicaciones usualmente no usan procedimientos almacenados.

4. Mapeo de objetos completo: Se soporta modelado de objetos complejo con composición, herencia, polimorfismo, etc. La capa de persistencia implementa persistencia transparente, esto es la capacidad de manipular los datos almacenados en la RDBMS directamente a través de objetos, sin tener que implementar alguna interfaz especial o heredar una clase. Se utilizan técnicas eficientes para la obtención y cacheo de información, las cuales son transparentes hacia la aplicación. Este nivel de funcionalidad es muy difícil de lograr a través de una solución in-house, ya que requiere meses o tal vez años de desarrollo. [LIB02]

2.10 ¿Cómo implementar ORM?

El primer paso es el más complicado y el que defina nuestro proyecto con objetos persistentes; elegir del mercado el ORM; este paso es muy importante ya que podemos quedar ligados a un producto poco eficiente, que desaparece del mercado o/y carece de mantenimiento.

La experiencia del desarrollador en el manejo de un ORM es determinante en la decisión de implementación de tal o cual solución ya que de nada sirve poseer el mejor ORM si no se lo sabe usar.

Luego de obtener el ORM hay que definir como se trabajará si se usará POJOs[10] o no, definir donde se encontrará el archivo de configuración si el framework ORM permite tener un archivo de configuración; creación de la base de datos, definición de las clases persistentes y no persistentes; definición del mapeo registro-objeto; definir los métodos de acceso a datos y de persistencia de los datos; programar los métodos y probarlos.

Básicamente estos son los pasos para implementar un framework ORM.

2.10.1 ¿Qué debe saber el Framework ORM para unir el modelo relacional con el modelo Orientado a Objetos?

1. Quién está detrás del modelo relacional:

   - ¿Qué gestor de bases de datos está detrás?
   - ¿A qué base de datos me conecto?
   - ¿Cómo me conecto?

[10] POJOs, Simple clase Java.

2. Cómo se emparejan propiedades y campos de tablas:

   Clave primaria:

   - ¿Cuál es la propiedad que se corresponde con la clave primaria de la tabla correspondiente?
   - ¿Qué método deberé utilizar para generar un nuevo valor de la clave primaria?

   Otras propiedades:

   - ¿Cómo empareja una propiedad con un campo de una tabla de la base de datos?
   - ¿Cómo se llama la columna correspondiente?
   - ¿Qué tipo tiene?
   - ¿Es obligatoria?

   Cómo gestiona las relaciones entre tablas

   - ¿Es una relación “uno-a-uno”, “uno-a-muchos”, “muchos-a-muchos”?

Para responder a las preguntas, se utilizan dos archivos distintos y fundamentales:

1. El archivo de propiedades o de configuración: Encargado de determinar los aspectos relacionados con el gestor de bases de datos y las conexiones con él.

2. Los archivos que definen el mapping: Emparejamiento de propiedades con tablas y columnas.

Archivo de Configuración

Muchos frameworks ORM nos permiten definir las configuraciones de la base de datos en un archivo de configuración esto nos permite cambiar el archivo de configuración y usar otra base de datos sin tener que tocar una sola línea de código.

El archivo de configuración es donde se encuentran todos los parámetros necesarios para la conexión entre la aplicación y la base de datos:

JDBC: Recordemos que todas las aplicaciones que se comunican con aplicaciones Java lo hacen por medio del jdbc y que existe jdbc por proveedor de base de datos. Por lo tanto debemos definir cual es la librería jdbc que usaremos y donde se encuentra.

Url: Donde se encuentra el servicio de base de datos.

Usuario y password: De la base de datos.

Dialecto de la base de datos: Es sabido que SQL es un estándar pero que ninguna base de datos lo cumple totalmente siempre.

Este archivo de configuración puede tener diferentes formatos depende del proveedor ORM; por ejemplo Hibernate permite guardar los datos en un archivo XML o en archivo descriptivo ".properties", JPA utiliza el archivo persistence.xml

El archivo de configuración debe estar protegido por normas de seguridad ya que contiene información confidencial y con su avería no se podría conectar la aplicación con la base de datos.

Archivo de Mapeo

Debemos decirle al frameworks ORM cómo persistir nuestras clases. Aquí es donde entra en juego el fichero de mapeo. El fichero de mapeo le dice al frameworks ORM qué debería almacenar en la base de datos y cómo.

La estructura exterior de un fichero de mapeo se parece a la siguiente que es de una aplicación utilizando Hibernate:

```
<?XML version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//hibernate/hibernate Mapping DTD 2.0//EN"
      "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
      <class name="de.gloegl.road2hibernate.Event" table="EVENTS">
            <id name="id" column="uid" type="long">
                  <generator class="increment"/>
            </id>
            <property name="date" type="timestamp"/>
            <property name="title" column="eventtitle"/>
      </class>
</hibernate-mapping>
```

Éste tiene una cabecera fija, típica de todos los documentos XML que no describiremos.

Entre las dos etiquetas <hibernate-mapping>, esta incluido un elemento class, donde podemos declarar a que clase se refiere este mapeo y a qué tabla de nuestra base de datos SQL se debería mapear.

Tendremos que dar al framework ORM la propiedad a utilizar un identificador único. Además, el ORM tiene la posibilidad de generar estos ids. El elemento <id> es la declaración de la propiedad id. name="id" es el nombre de la propiedad - Hibernate usará los métodos getId y setId para acceder a ella. El atributo column le dice a Hibernate que columna de la tabla EVENTS contendrá el id. El atributo type le dice a Hibernate el tipo de la propiedad - en este caso un long.

El elemento <generator> especifica la técnica que se usará para la generación de id -- en este caso se uso un incremento.

Finalmente están incluidas las declaraciones para las propiedades persistentes en el fichero de mapeo.

Tenemos que observar algunas cosas de aquí. Al principio, igual que en el elemento <id>, el atributo name del elemento <property> le dice a Hibernate que métodos get y set utilizar ya que requiere que los métodos para acceder a los datos comiencen con get y métodos para asignar datos set.

En el siguiente capítulo se estudia el framework de persistencia, JPA (Java Persistence API) y se explica el mapeo.

2.11 ¿Por qué ORM?

2.11.1 Beneficios

Productividad

El código relativo a la persistencia es una de las partes más tediosas y difíciles en el desarrollo de una aplicación. ORM elimina gran parte de ese trabajo tedioso y permite centrarse en la propia lógica de negocio, independientemente de la estrategia que se adopte. ORM convenientemente usado reduce de forma importante el tiempo de desarrollo.

Mantenibilidad

Otro de los beneficios es la mantenibilidad. Al haber menos de líneas de código, nuestra aplicación es más fácil de entender y mantener dado que se da énfasis a la lógica de negocio, tal y como he dicho en el párrafo anterior. No obstante, puede ser discutible que el número de líneas de código sea la mejor medida para la mantenibilidad.

Hay otros argumentos en defensa de la mantenibilidad. En las aplicaciones donde la persistencia se implementa "a mano" mediante un driver JDBC y código SQL, los dos mundos (el relacional y el de los objetos) no se llevan muy bien, y si se produce algún cambio en el diseño de uno de ellos, casi con toda seguridad esto implicará algún cambio en el otro. Además, los diseños de ambos están pensados para evitar conflictos entre ellos (generalmente, el diseño de las clases está hecho para evitar conflictos con el diseño Entidad/Relación) cualquier herramienta ORM (Hibernate, TopLink) establece, por así decirlo, un "buffer" entre ambos diseños para permitir una mayor flexibilidad.

Rendimiento

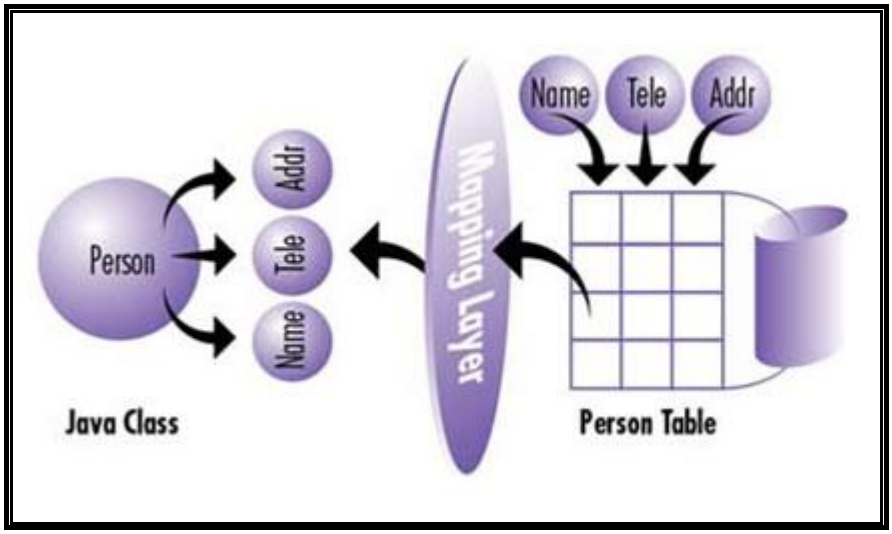
Continuando con los beneficios, otro posible beneficio es el rendimiento. Esto puede resultar paradójico, ya que se suele argumentar que con la persistencia "manual" se obtiene un rendimiento mayor que con herramientas ORM. A priori, esto es cierto (al igual que lo es que un programa escrito en ensamblador es más rápido que uno escrito en Java, C# o cualquier otro lenguaje de alto nivel) lo que ocurre es que hay que tener en cuenta las restricciones de tiempo que existen en el desarrollo de aplicaciones en el mundo real, es decir, la persistencia manual proporciona un rendimiento mayor que una herramienta ORM cuando se dedica a escribir dicha capa de persistencia al mismo tiempo que el resto de la aplicación.

Para terminar con lo referente al rendimiento, resulta lógico a priori pensar que quienes han desarrollado una herramienta ORM han dedicado más tiempo que cualquiera de nosotros a la búsqueda de optimizaciones. Un ejemplo es que al realizar una actualización en una tabla, si sólo actualizamos las columnas que se quieren actualizar y no todas, el resultado que se obtiene es que en algunos motores de bases de datos se consigue un incremento del rendimiento, mientras que en otros sistemas de Bases de Datos Relacionales se degrada el rendimiento. Otro ejemplo es que instanciar poolings de

objetos PreparedStatement hace que el rendimiento del driver JDBC DB2 mejore ostensiblemente, mientras que el de InterBase empeora. [LIB01]

CAPITULO III

Mapeo Objeto Relacional con JPA



"Los logros más importantes no se miden sólo por los resultados, sino por el esfuerzo que ponemos en realizarlos."

3.1 Introducción

Para desarrollar un sistema moderno, sin duda el paradigma más utilizado es el de la orientación a objetos. Pero a la hora de modelar las necesidades de persistencia de datos comerciales, el paradigma que se impone es el de base de datos relacionales (RDBMS). Esta diferencia de enfoques hace que un framework de persistencia u ORM sea un componente crítico de la arquitectura de una aplicación.

En los últimos años, numerosos frameworks de persistencia han evolucionado para simplificar la transición del modelo de objetos al relacional y viceversa. Elegir uno que se ajuste a sus requerimientos no es una tarea trivial. Es por ello que destacaremos uno de los que pretende ser el estándar definitivo, Java Persistence API.

3.2 Java Persistence API (JPA)

3.2.1 Definición

La API de persistencia Java es el estándar de transformación objeto/relacional que permite a los desarrolladores Java manejar datos relacionales en las aplicaciones Java mediante anotaciones [11] o con descriptores XML. [LIB05]

Está construida alrededor de tres áreas principales:

- API de persistencia Java.
- Metadatos de transformación objeto/relacional (Anotaciones).
- Lenguaje de consulta.

3.2.2 Arquitectura

Para utilizar JPA, es necesario elegir un proveedor de persistencia el cual se ocupa de lo relacionado con la carga y almacenamiento de los datos, cuando refrescar cada instancia y de la sincronización entre los objetos. [WWW10]

La arquitectura de JPA, en alto nivel, se muestra en la siguiente figura.

[11] Anotación: Forma de añadir metadatos al código fuente Java que están disponibles para la aplicación en tiempo de ejecución.

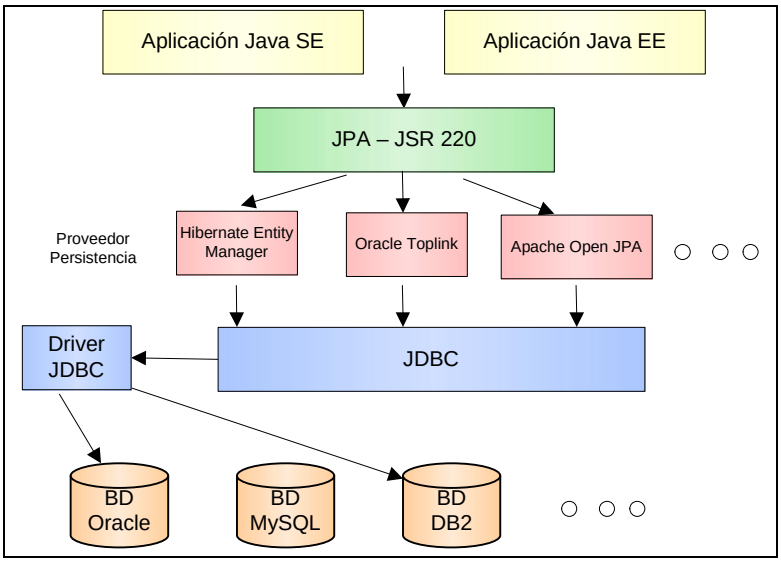


Figura 3.1 Arquitectura externa JPA.

Cualquier aplicación Java (SE o EE) que utiliza JPA no está vinculada a los proveedores de persistencia (incluso si son de código abierto) como Hibernate o el TopLink. En lugar de ello, la aplicación sólo utiliza una especificación estándar de JCP (Java Community Process). [WWW11]

3.2.3 Elementos

Para facilitar la persistencia JPA se basa en los siguientes elementos:

- Entidades
- Contexto de Persistencia
- Unidad de Persistencia
- Administrador de Entidades

La siguiente figura muestra la relación entre estos elementos.

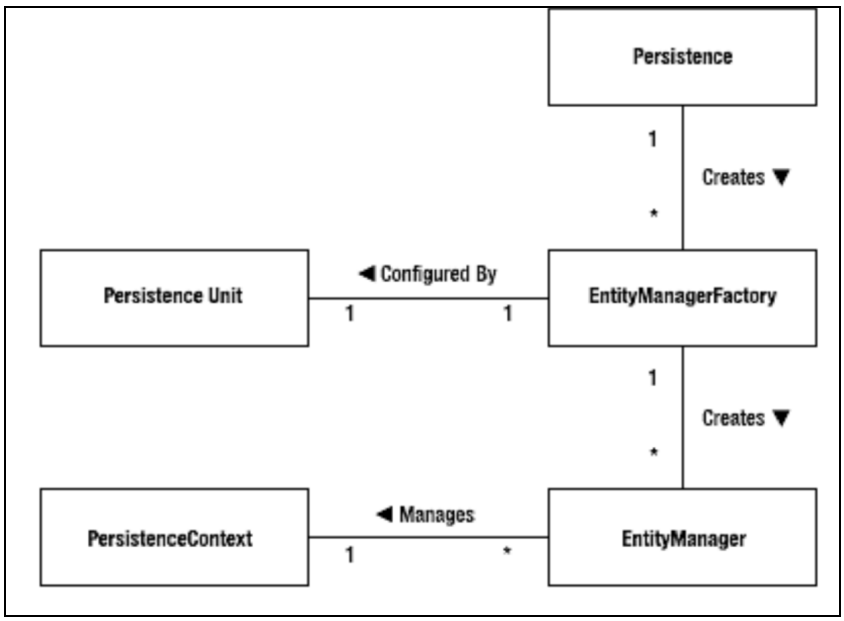


Figura 3.2 Relación entre elementos de JPA. Fuente: [LIB05]

La figura muestra que para cada unidad de persistencia hay un EntitityManagerFactory y que muchos Entity Managers pueden ser creados para un solo EntitityManagerFactory. En tanto que muchos Entity Managers pueden apuntar al mismo contexto de persistencia. [LIB05]

3.2.3.1 Entidades

Una entidad es un objeto persistente en la aplicación. [WWW11]

Entidad = POJO (Plain Old Java Object) + anotaciones

```
@Entity
public class Persona {
...
}
```

- Típicamente representa una tabla de una base de datos relacional.
- Cada instancia de una entidad corresponde normalmente a una fila de la tabla.
- La parte principal de la entidad es la clase Entity, aunque se pueden usar clases auxiliares.

3.2.3.1.1 Requerimientos para la clase Entity

- La clase debe llevar la anotación javax.persistence.Entity.
- Tener un constructor por omisión público o protegido. Sin argumentos.
- La clase, los métodos o las variables de instancia persistentes no deben ser declarados final.
- Si una instancia de un entity es pasada por valor como un objeto debe implementar la interfaz java.io.Serializable.
- Los entities pueden extender tanto clases entity como cualquier otra, incluso clases no entity pueden extender clases entity.
- Las variables persistentes deben ser declaradas private o protected y solamente pueden ser accedidas a través de los correspondientes métodos de acceso: getters o setters. [DOC01]

Un entity debe declarar una llave primaria. Esto se hace marcando el campo con la anotación @Id. Al igual que en las bases de datos relacionales, la llave primaria hace del entity un objeto único. Cuando se requiere de una llave compuesta se puede usar una clase como llave primaria compuesta. Para este tipo de claves, se utilizan las anotaciones javax.persistence.EmbededId y javax.persistence.IdClass.

Los entities no pueden ser accedidos directamente, deben ser desplegados y usados localmente desde JSE[12] o en un contenedor (desde beans de sesión, o de mensajería, o incluso desde cualquier componente web). De cualquier manera, el código cliente debe primero recuperar una instancia particular del entity desde el contexto de persistencia o crear una y adicionarla a dicho contexto. [WWW10]

3.2.4 Contexto de Persistencia

Un contexto de persistencia es un conjunto de instancias de entidades en las que para cualquier entidad hay únicamente una instancia. En el contexto de persistencia, las instancias de las entidades y sus ciclos de vida son administrados. Al decir que una instancia de entidad es administrada significa que está en el contexto de persistencia y que debe ser representada por un EntityManager.

[12] JSE: Java Estándar Edition

Cada contexto de persistencia es asociado con una unidad de persistencia, restringiendo las clases de las instancias administradas al conjunto definido por la unidad de persistencia. [LIB05]

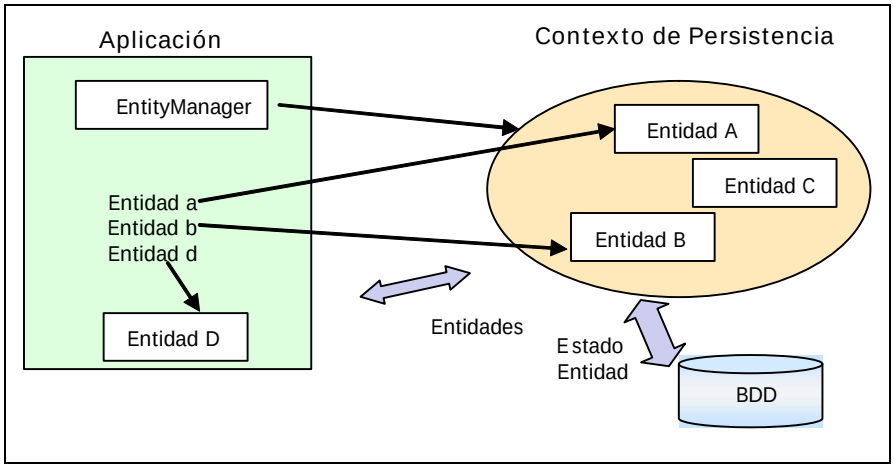


Figura 3.3 Contexto de Persistencia

3.2.5 Unidad de Persistencia

La Unidad de Persistencia consiste en la declaración de entidades que serán mapeadas a una base de datos relacional. Es definida por el archivo persistence.xml.

persistence.xml

Este archivo de configuración es donde se definen los contextos de persistencia de la aplicación. Se debe situar dentro del directorio META-INF.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
 <persistence-unit name="TestPersistence " transaction-type="RESOURCE_LOCAL">
  <provider>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider</provider>
  <class>com.sun.demo.jpa.Player</class>
  <class>com.sun.demo.jpa.Team</class>
  <properties>
   <property name="toplink.jdbc.user" value="league"/>
   <property name="toplink.jdbc.password" value="league"/>
   <property name="toplink.jdbc.url" value="jdbc:derby://localhost:1527/league"/>
   <property name="toplink.jdbc.driver" value="org.apache.derby.jdbc.ClientDriver"/>
   <property name="toplink.ddl-generation" value="create-tables"/>
  </properties>
 </persistence-unit>
</persistence>
```

Los elementos más importantes del archivo son los siguientes:

persistence-unit name: Especifica un nombre para el contexto o persistente. Si únicamente se especifica uno, no habrá; que incluir su nombre cuando se recupere el EntityManager (con la anotación @PersistenceContext o @PersistenceUnit).

transaction-type: El valor de este elemento es JTA o RESOURCE-LOCAL. El tipo de transacción por omisión es RESOURCE-LOCAL para aplicaciones Java SE. Una transacción tipo JTA significa que el administrador de entidades participa en la transacción.

provider: Especifica el nombre del proveedor de persistencia. En el ejemplo se muestra la implementación de GlassFish[13], Toplink Essentials.

class: Lista los nombres de las entidades que son parte de la unidad de persistencia.

properties: Se especifica el tipo de base de datos a utilizar en entornos Java SE si no es posible usar JNDI. Las propiedades de la conexión a la base de datos incluyen el nombre de usuario y contraseña para la conexión, la cadena de la conexión (URL) y el nombre de la clase del driver. Adicionalmente, puedes incluir propiedades de persistencia del proveedor como opciones para crear o borrar-crear nuevas tablas. La propiedad namespace javax.persistence está reservada para propiedades definidas por la especificación. Las opciones de las especificaciones y propiedades del proveedor deben ser usadas para evitar conflictos con la especificación. El archivo persistence.xml mostrado usa la implementación GlassFish. Sus propiedades de la especificación del proveedor son el namespace toplink y no son parte de su propia especificación. Los proveedores de persistencia ignorarán cualquier otra propiedad que no estén en sus especificaciones o que no sean parte de sus propiedades de las especificaciones del proveedor. [WWW13]

3.2.6 Administrador de Entidades (EntityManager)

Esta interfaz es en la que se apoya la API de persistencia y la que se encarga del mapeo entre una tabla relacional y su objeto Java. Proporciona métodos para manejar la

[13] GlassFish: Proyecto open source de SUN.

persistencia de una entidad, permite añadir, eliminar, actualizar y consultar así como manejar su ciclo de vida. Sus métodos más importantes son: [WWW12]

- persist(Object entity): Almacena el objeto entity en la base de datos.
- merge(T entity): Actualiza las modificaciones en la entidad devolviendo la lista resultante.
- remove(Object entity): Elimina la entidad.
- find(Class<T> entity, Object primaryKey): busca la entidad a través de su clave primaria.
- flush(): Sincroniza las entidades con el contenido de la base de datos.
- refresh(Object entity): Refresca el estado de la entidad con su contenido en la base de datos.
- createQuery(String query): Crea una query utilizando el lenguaje JPQL.
- createNativeQuery(): Crea una query utilizando el lenguaje SQL.
- isOpen(): Comprueba si está; abierto el EntityManager.
- close(): Cierra el EntityManager.

Ejemplos:

```
//persist
public Order addNewOrder(Customer customer, Product product) {
Order order = new Order(product);
customer.addOrder(order);
em.persist(order);
return order;
}

//find
public Customer findCustomer(Long customerId) {
Customer customer = em.find(Customer.class, customerId);
return customer;
}

//update
order.setDeliveredDate(date);
...
public Order updateOrder(Order order) {
return em.merge(order);
}

//remove
public void deleteOrder(Long orderId) {
Order order = em.find(Order.class, orderId);
em.remove(order);
}
```

Podemos obtener una referencia al EntityManager a través de la anotación @PersistenceContext. El contenedor nos proporciona el contexto de persistencia

mediante inyección por lo que no tendremos que preocuparnos de su creación y destrucción.

```
@PersistenceContext
EntityManager entityManager;
```

Otra forma de obtener un EntityManager es a través de la factoría EntityManagerFactory con el nombre del contexto de persistencia configurado en el persistence.xml.

```
EntityManagerFactory emf = Persistence.createEntityManagerFactory("TestPersistence");
EntityManager em = emf.createEntityManager();
```

En el método createEntityManagerFactory de la clase Persistence se debe pasar el nombre del contexto definido en el persistence.xml. [WWW12]

3.2.7 Ciclo de vida de una Entidad

Engloba dos aspectos: la relación entre el objeto entidad y su contexto a persistir y por otro lado la sincronización de su estado con la base de datos. Para realizar estas operaciones la entidad puede encontrarse en cualquiera de estos cuatro estados: [WWW12]

- New: Nueva instancia de la Entidad en memoria sin que aún le sea asignado su contexto persistente almacenado en la tabla de la base de datos.
- Managed: La Entidad dispone de contenido asociado con el de la tabla de la base de datos debido a que se utilizó el método.
- Persist(): Los cambios que se produzcan en la Entidad se podrán sincronizar con los de la base de datos llamando al método flush().
- Detached: La Entidad se ha quedado sin su contenido persistente. Es necesario utilizar el método merge() para actualizarla.
- Removed: Estado después de llamarse al método remove() y el contenido de la Entidad será; eliminado de la base de datos.

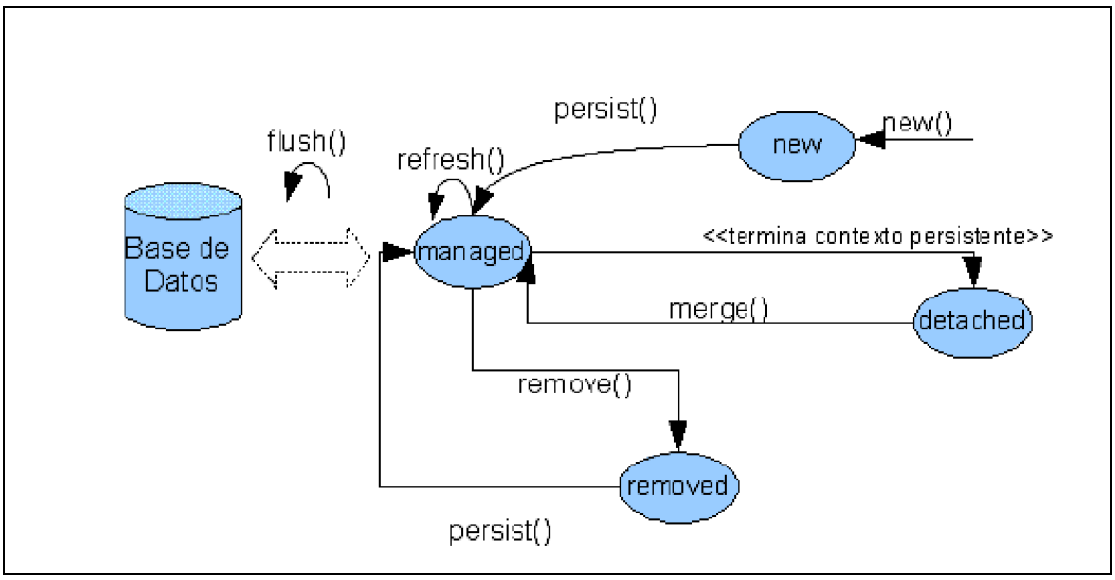


Figura 3.4 Ciclo de vida de entidades. Fuente: [WWW12]

3.2.8 Anotaciones

Una anotación o metadato proporciona un recurso adicional al elemento de código al que va asociado en el momento de la compilación. Cuando la máquina virtual de Java (JVM) ejecuta la clase busca estos metadatos y determina el comportamiento a seguir con el código al que va unido la anotación. [WWW12]

A continuación se describen las principales anotaciones.

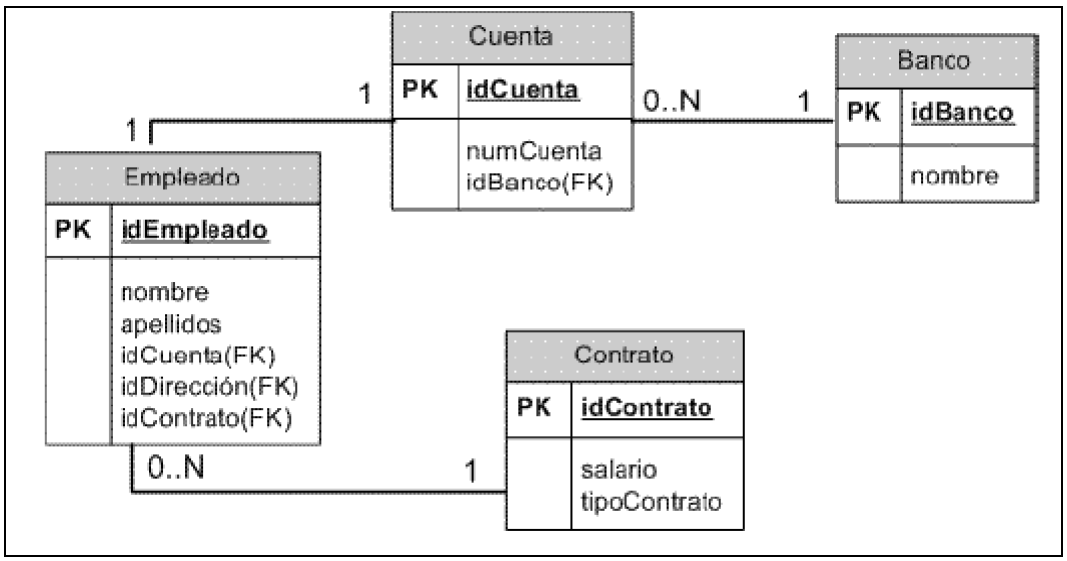


Figura 3.5 Diagrama de tablas. Fuente: [WWW12]

@Entity: Indica que es una Entidad.

name: Por defecto el nombre de la clase pero se puede especificar otra diferente.

@Table: Específica la tabla principal relacionada con la entidad.

name: Nombre de la tabla, por defecto el de la entidad si no se especifica.

catalog: Nombre del catálogo.

schema: Nombre del esquema.

uniqueConstraints: Constrains entre tablas relacionadas con la anotación @Column y @JoinColumn.

@SecondaryTable: Especifica una tabla secundaria relacionada con la entidad si éste englobara a más de una. Tiene los mismos atributos que @Table.

@SecondaryTables: Indica otras tablas asociadas a la entidad.

@UniqueConstraints: Específica que una única restricción se incluya para la tabla principal y la secundaria.

@Column: Especifica una columna de la tabla a mapear con un campo de la entidad.

name: Nombre de la columna.

unique: Si el campo tiene un único valor.

nullable: Si permite nulos.

insertable: Si la columna se incluirá; en la sentencia INSERT generada.

updatable: Si la columna se incluirá; en la sentencia UPDATE generada.

table: Nombre de la tabla que contiene la columna.

length: Longitud de la columna.

precision: Número de dígitos decimales.

scale: Escala decimal.

@JoinColumn: Especifica una campo de la tabla que es foreign key de otra tabla definiendo la relación del lado propietario.

name: Nombre de la columna de la foreign key.

referenced: Nombre de la columna referencia.

unique: Si el campo tiene un único valor.

nullable: Si permite nulos.

insertable: Si la columna se incluirá; en la sentencia INSERT generada.

updatable: Si la columna se incluirá; en la sentencia UPDATE generada.

table: Nombre de la tabla que contiene la columna.

@JoinColumns: Anotación para agrupar varias JoinColumn.

@Id: Indica la clave primaria de la tabla.

@GeneratedValue: Asociado con la clave primaria, indica que ésta se debe generar por ejemplo con una secuencia de la base de datos.

strategy: Estrategia a seguir para la generación de la clave: AUTO (valor por defecto, el contenedor decide la estrategia en función de la base de datos), IDENTITY (utiliza un contador, Ej: MySQL), SEQUENCE (utiliza una secuencia, Ej: Oracle, PostgreSQL) y TABLE (utiliza una tabla de identificadores).

generator: Forma en la que general la clave.

@SequenceGenerator: Define un generador de claves primarias utilizado junto con la anotación @GeneratedValue. Se debe especificar la secuencia en la entidad junto a la clave primaria.

name: Nombre del generador de la clave.

sequence: Nombre de la secuencia de la base de datos del que se va a obtener la clave.

initialValue: Valor inicial de la secuencia.

allocationSize: Cantidad a incrementar de la secuencia cuando se llegue al máximo.

@TableGenerator: Define una tabla de claves primarias generadas. Se debe especificar en la anotación @GeneratedValue con strategy = GenerationType.TABLE.

name: Nombre de la secuencia.

table: Nombre de la tabla que guarda los valores generados.

catalog: Catalogo de la tabla.

schema: Esquema de la tabla.

pkColumn Name(): Nombre de la clave primaria de la tabla.

valueColumn Name(): Nombre de la columna que guarda el último valor generado.

pkColumn Value(): Valor de la clave primaria.

initialValue: Valor inicial de la secuencia.
allocationSize: Cantidad a incrementar de la secuencia.
uniqueConstraints: Constrains entre tablas relacionadas.

@AttributeOverride: Indica que sobrescriba el campo con el de la base de datos asociado.
name: Nombre del campo
column: Columna del campo

@AttributeOverrides: Mapeo de varios campos.

@EmbeddedId: Se utiliza para formar la clave primaria con múltiples campos.

@IdClass: Se aplica en la clase entidad para especificar una composición de la clave primaria mapeada a varios campos o propiedades de a entidad.

@Transient: Indica que el campo no se debe persistir.

@Version: Se utiliza a la hora de persistir la entidad en base de datos para identificar las entidades según su versión. Se actualiza automáticamente cuando el objeto es mapeado en la base de datos.

@Basic: Mapeo por defecto para tipos básicos: tipos primitivos, wrappers de los tipos primitivos, String, BigInteger, BigDecimal, Date, Calendar, Time, Timestamp, byte[], Byte[], char[], Character[], enumerados y cualquier otra clase serializable.

fetch: Determina la forma en que se cargan los datos (relaciones entre tablas): FetchType.LAZY (carga de la entidad únicamente cuando se utiliza), FetchType.EAGER (carga de todas las entidades relacionadas con ella).
optional: Si permite que el campo sea nulo. Por defecto true.

De acuerdo a la Figura 3.5:
FechType.EAGER - En el momento de acceder a la tabla 'Empleado' también se consultan las tablas Cuenta y Contrato y las entidades que a su vez contuvieran éstas ya que están relacionadas.

FechType.LAZY - En el momento de acceder a la tabla Empleado únicamente se carga su contenido y en el momento de necesitar alguno de los campos idCuenta o idContrato, es en ese momento cuando se accede a la tabla correspondiente.

@OneToOne: (1:1) Indica que un campo está; en relación con otro (Ej: dentro de una empresa, un empleado tiene un contrato de trabajo).

cascade: Forma en que se deben actualizar los campos: ALL, PERSIST, MERGE, REMOVE y REFRESH.
fetch: Determina la forma en que se cargan los datos: FetchType.LAZY (carga de la entidad únicamente cuando se utiliza), FetchType.EAGER (carga de todas las entidades relacionadas con ella).
optional: Si la asociación es opcional.
mappedBy: El campo que posee la relación, únicamente se especifica en un lado de la relación.

@ManyToOne: (N:1) Indica que un campo está; asociado con varios campos de otra entidad (ej: las cuentas que posee un banco o los empleados que pertenecen a un departamento).

cascade, fetch y optional: Igual que la anterior anotación.

@OneToMany: (1:N) Asocia varios campos con uno (Ej: varios departamentos de una empresa).

cascade, fetch y optional: Igual que la anterior anotación.
mappedBy: El campo que posee la relación. Es obligatorio que la relación sea unidireccional.

@ManyToMany: (N:M) Asociación de varios campos con otros con multiplicidad muchos-a-muchos (Ej: relaciones entre empresas).

cascade, fetch y mappedBy: Igual que la anterior anotación.

@Lob: Se utiliza junto con la anotación @Basic para indicar que un campo se debe persistir como un campo de texto largo si la base de datos soporta este tipo.

@Temporal: Se utiliza junto con la anotación @Basic para especificar que un campo fecha debe guardarse con el tipo java.util.Date o java.util.Calendar. Si no se especificara ninguno de estos por defecto se utiliza java.util.Timestamp.

@Enumerated: Se utiliza junto con la anotación @Basic e indica que el campo es un tipo enumerado (STRING), por defecto ORDINAL.

@JoinTable: Se utiliza en el mapeo de una relación ManyToMany o en una relación unidireccional OneToMany.

name: Nombre de la tabla join a donde enviar la foreing key.

catalog: Catalog de la tabla.

schema: Esquema de la tabla.

joinColumns: Columna de la foreign key de la tabla join que referencia a la tabla primaria de la entidad que posee la asociación (sólo en un lado de la relación).

inverseJoinColumns: Columnas de la foreign key de la tabla join que referencia a la tabla primaria de la entidad que no posee (lado inverso de la relación)

uniqueConstraints: Constraints de la tabla.

@MapKey: Específica la clave de una clase de tipo java.util.Map.

@OrderBy: Indica el orden de los elementos de una colección por un ítem específico de forma ascendente o descendente.

@Inheritance: Define la forma de herencia de una jerarquía de clases entidad, es decir la relación entre las tablas relacionales con las entidades.

@NamedQuery: Especifica el nombre del objeto query utilizado junto a EntityManager.

name: Nombre del objeto query.

query: Especifica la query a la base de datos mediante lenguaje Java Persistence Query Language (JPQL).

@NamedQueries: Específica varias queries como la anterior.

@NamedNativeQuery: Específica el nombre de una query SQL normal.

name: Nombre del objeto query.

query: Especifica la query a la base de datos.

resultClass: Clase del objeto resultado de la ejecución de la query.

resultSetMapping: Nombre del SQLResultSetMapping definido.

@NamedNaviteQueries: Específica varias queries SQL.

3.2.9 Relaciones entre entidades

3.2.9.1 Relaciones de multiplicidad

Hay cuatro tipos de multiplicidades en las relaciones entre entidades:

- Uno a uno.
- Uno a muchos.
- Muchos a uno.
- Mucho a Muchos.

Uno a uno: en esta relación cada instancia de un entity está relacionada a una única instancia de otro entity. Se identifica la relación con la anotación: javax.persistence.OneToOne.

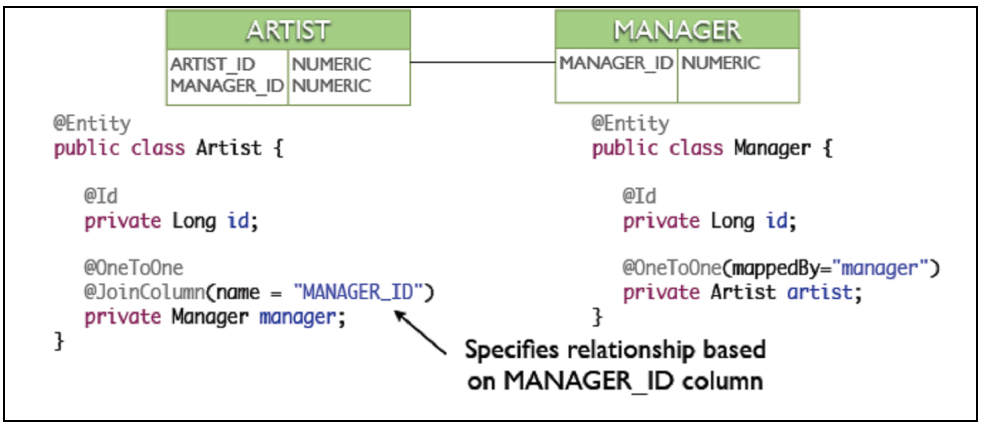


Figura 3.6 Ejemplo de @ OneToOne. Fuente: [WWW16]

Uno a muchos: una instancia de un entity puede estar relacionada con múltiples instancias de otros entities. La anotación javax.persistence.OneToMany caracteriza esta situación.

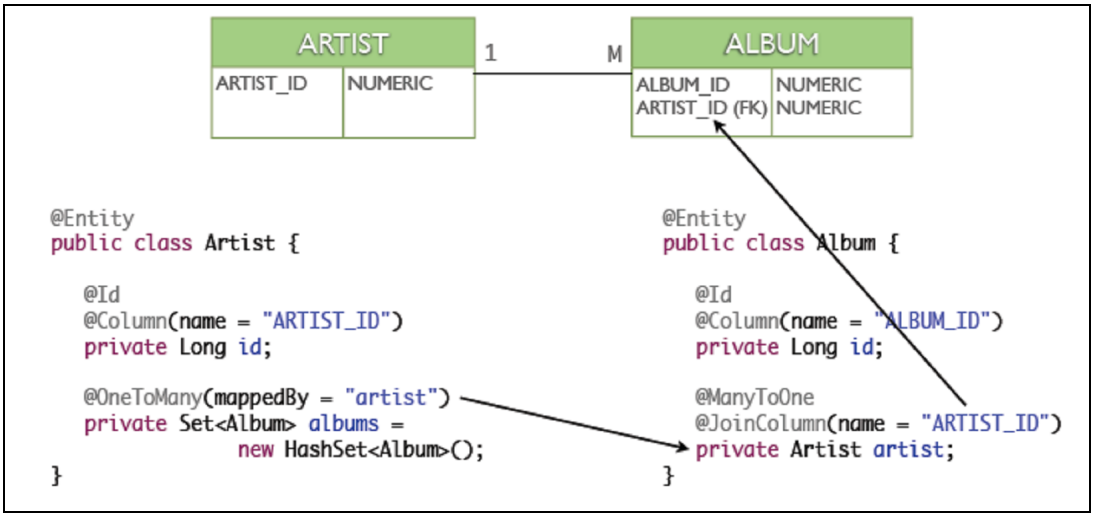


Figura 3.7 Ejemplo de @ OneToMany. Fuente: [WWW16]

Muchos a uno: múltiples instancias de un entity pueden estar relacionadas con una única instancia de otro entity. Es la opuesta a la relación uno a muchos. Se usa la anotación javax.persistence.ManyToOne.

Muchos a muchos: instancias de un entity pueden estar relacionadas con múltiples instancias de otros entities. Se identifica mediante la anotación javax.persistence.ManyToMany. [WWW16]

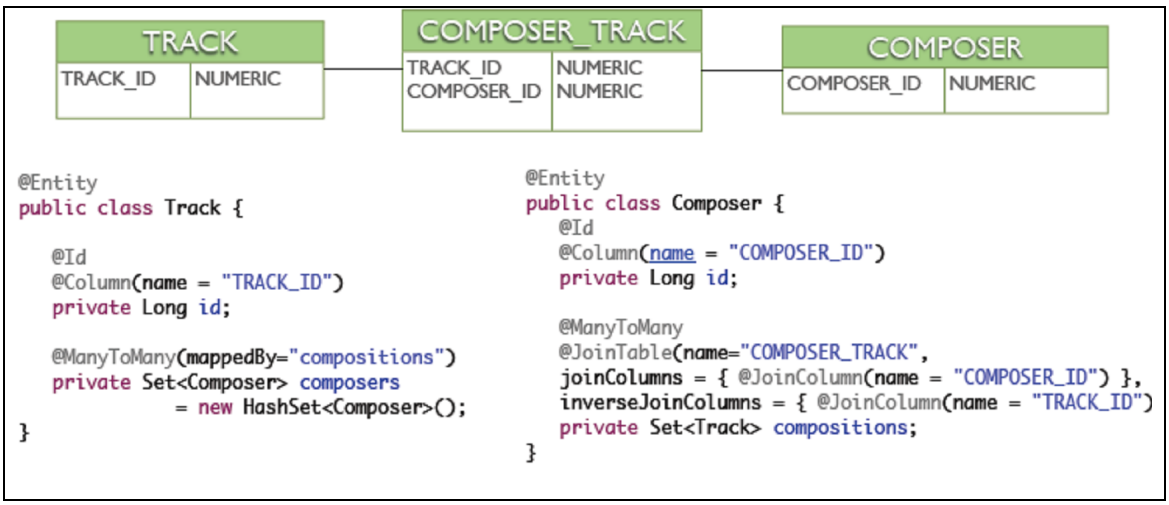


Figura 3.8 Ejemplo de @ ManyToMany. Fuente: [WWW16]

3.2.9.2 Dirección de las relaciones entre entidades

La dirección de una relación determina si una consulta puede navegar de una entidad a otra.

En las entidades que usan relaciones es frecuente que se dependa de la existencia de la otra entidad. Cuando este es sea el caso, el borrado, la construcción, o modificación del entity dominante debería desencadenar el borrado, construcción, o modificación de los dominados: relaciones en cascada. Este tipo de operación se especifica con el elemento cascade en las anotaciones @OneToOne y en @OneToMany.

Bidireccional: Se presenta cuando un entity conoce a otro y este a su vez lo conoce a él. Se dice que se tiene un lado propietario de la relación y un lado inverso.

Unidireccional: Únicamente uno de los entities conoce al otro. En este caso solamente se tiene el lado propietario.
Es el lado propietario de una relación el que determina como se hacen las actualizaciones a la relación en la base de datos.

Reglas de las Relaciones bidireccionales

Las siguientes son las reglas que se deben tener en cuenta para definir el lado propietario de una relación:

- El lado inverso de una relación bidireccional debe referirse al lado propietario por medio del elemento mappedBy de la anotación correspondiente. Este elemento designa la propiedad o campo en el otro entity que es el propietario de la relación.

- El lado muchos, en relaciones bidireccionales muchos a muchos y muchos a uno es siempre el lado propietario; por tanto, no debe definir el elemento mappedBy.

- En las relaciones bidireccionales uno a uno, el lado propietario corresponde al bean que contiene la correspondiente llave foránea.

- Cuando la relación es muchos a muchos, cualquier lado puede ser el lado propietario.

3.2.10 Lenguaje de Consulta

El Java Persistence Query Language (JPQL) es usado para definir queries para las entidades y su estado persistente. Esto le permite al desarrollador especificar la semántica de los queries de manera portable, independiente de una base de datos.

Java Persistence Query Language es una extensión de EJB QL; al igual que EJB QL, es un lenguaje al estilo de SQL.

Una sentencia JPQL puede ser un select, un update o un delete. Cualquiera de ellas puede ser construida dinámicamente o puede ser estáticamente definida con metadatos XML o anotaciones (@NamedQuery, @NamedNativeQuery). Además pueden tener parámetros definidos por nombre o por posición.

Hay que resaltar que el EntityManager es fábrica para objetos Query CreateNamedQuery, createQuery, createNativeQuery; métodos para controlar el máximo de resultados, la paginación, y modo de vaciado (flush).

3.2.10.1 Sintáxis

Una consulta de select tiene 6 elementos: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY.
SELECT y FROM son requeridos los demás son opcionales. La forma general de una consulta de select es:

QL_statement ::= select_clause from_clause
[where_clause][groupby_clause][having_clause][orderby_clause]

Las consultas de delete o update tienen la siguiente forma:
update_statement :: = update_clause [where_clause]
delete_statement :: = delete_clause [where_clause]

La where_clause tiene el mismo significado que en el caso del SELECT. [DOC01]

3.2.10.2 Ejemplos JPQL

- Queries que navegan a entidades relacionadas. [WWW14]

```
SELECT DISTINCT p FROM Player p, IN(p.teams) t
SELECT DISTINCT p FROM Player p JOIN p.teams t
SELECT DISTINCT p FROM Player p WHERE p.team IS NOT EMPTY
```

- Queries que navegan usando relaciones con valores sencillos.

```
SELECT t FROM Team t JOIN t.league l WHERE l.sport = 'soccer' OR l.sport ='football'
```

- Filtrado en los campos relacionados.

```
SELECT DISTINCT p FROM Player p, IN (p.teams) t WHERE t.league.sport = :sport
```

- LIKE

  ```
  SELECT p FROM Player p WHERE p.name LIKE 'Mich%'
  ```

- IS NULL

  ```
  SELECT t FROM Team t WHERE t.league IS NULL
  ```

- IS EMPTY

  ```
  SELECT p FROM Player p WHERE p.teams IS EMPTY
  ```

- BETWEEN

  ```
  SELECT DISTINCT p FROM Player p WHERE p.salary BETWEEN :lowerSalary AND :higherSalary
  ```

- Operadores de comparación.

  ```
  SELECT DISTINCT p1 FROM Player p1, Player p2 WHERE p1.salary > p2.salary AND p2.name = :name
  ```

Creando Consultas

- Consulta Dinámica

  ```
  public Album findById(Long id) {
  String jpql = "select distinct a from Album a left join fetch a.artist art "
  + "left join fetch art.genre left join fetch a.tracks where a.id = :id"
  Query query = getEntityManager().createQuery(jpql);
  query.setParameter("id", id);
  return (Album) query.getSingleResult();
  }
  ```

- Consulta Estática

  ```
  @NamedQuery(name="artist.all",
  query="select distinct a from Artist a left join fetch a.albums")
  public List<Artist> findAll() {
  Query query = getEntityManager().createNamedQuery("artist.all");
  return query.getResultList()
  }
  ```

3.2.11 Transaccionalidad

Una aplicación empresarial típica accede y almacena información en uno o más bases de datos. Como esta información es crítica para las operaciones de negocios, debe ser precisa, actual, y confiable. Las transacciones controlan el acceso concurrente a los datos por parte de múltiples programas y en evento en que ocurra una falla del sistema, las transacciones aseguran que después de la recuperación los datos estén en un estado consistente.

Una transacción es una unidad indivisible de trabajo que puede finalizar con un commit o con un rollback.
Los métodos begin (), commit () y rollback () demarcan los límites de la transacción.

Las implementaciones JPA realizan automáticamente rollback si cualquier excepción ocurre durante el proceso commit.

3.2.12 Ventajas de JPA

La Java Persistence API constituye un nuevo concepto de programación que:

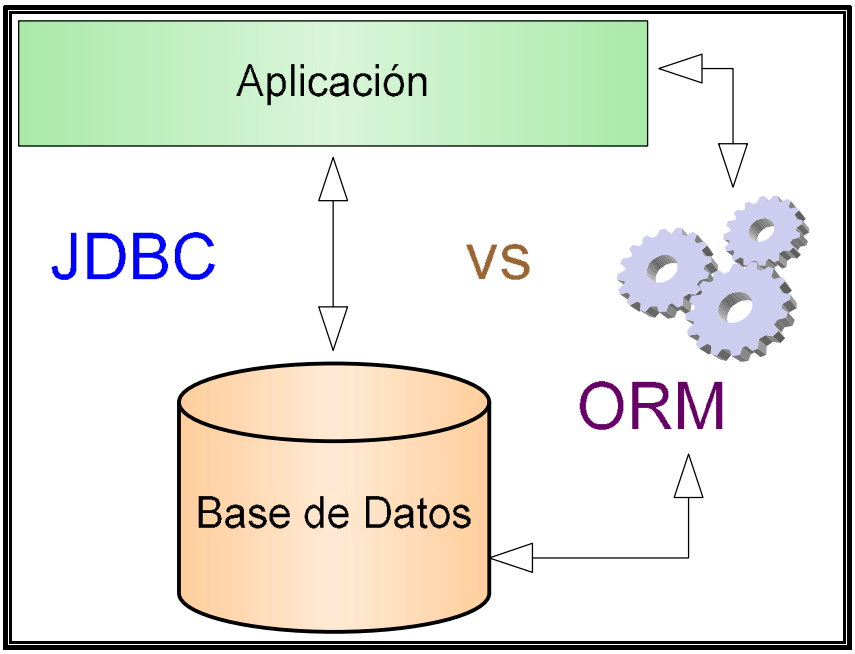
- Integra conceptos de muchas infraestructuras existentes como Hibernate, Toplink y JDO.
- Se puede usar tanto en entornos Java SE, así como en Java EE.
- Permite que diferentes proveedores de persistencia se puedan usar sin afectar el código del entity.
- Evita tener que capturar información de la pantalla y construir nuestras sentencias SQL, ya que mediante una configuracion xml y anotaciones sobre el código fuente podemos mapear directamente de objetos en memoria a tablas en bases de datos. Esto, a diferencia de un API como JDBC, permite pensar más en objetos que en tablas y como acceder a los datos en ellas.
- Simplicidad: Una única clase para declarar la persistencia (con la ayuda de anotaciones).
- Facilidad de aprendizaje.
- Transparencia: Las clases a persistir son simples POJOs.
- No hay restricciones con respecto a relaciones entre objetos (herencia, poliformismo).

3.2.13 Desventajas de JPA

- Se limita principalmente a la comunidad Java, pero es posible que el API de persistencia aparezca en las otras plataformas en el futuro.

CAPITULO IV

Persistencia Manual vs ORM



"La vida se escribe cada día y siempre tendremos la oportunidad de elegir caminos que nos llenen de esperanza y satisfacción". Anónimo.

4.1 Introducción

Se ha establecido que las alternativas más usuales de persistir objetos Java en base de datos relacionales es a través de JDBC directamente conectado a la base de datos mediante ejecuciones de sentencias SQL y ORM a través del uso de metadatos.

En este capítulo se pretende comparar ORM frente a JDBC considerando facilidad, productividad, rendimiento y curva de aprendizaje.

4.2 Comparativa

4.2.1 Facilidad

A continuación se enumeran los pasos generales para la interacción de JDBC y ORM con una base de datos relacional.

4.2.1.1 Pasos para la interacción con RDBMS

JDBC

a) Cargar el driver JDBC.
b) Conectarse a la Base de Datos utilizando la clase Connection.
c) Crear sentencias SQL, utilizando objetos de tipo Statement.
d) Ejecutar las sentencias SQL a través de los objetos de tipo Statement.
e) En caso que sea necesario, procesar el conjunto de registros resultante utilizando la clase ResultSet.

Ejemplo de interacción con la RDBMS Oracle usando JDBC: Recupera lista de patologías de la tabla BOD_TAB_PROCEDIMIENTO.

```
(a) Cargar el driver
Class.forName("oracle.jdbc.OracleDriver").newInstance();

(b) Abrir una conexión a la base de datos
Connection conn = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:GENERAL", "BIENESTAR", "DBU");
```

```
(c) Crear consulta SQL
Statement  stmt = conn.createStatement();


(d) Ejecutar la sentencia SQL
ResultSet rs = stmt.executeQuery("SELECT *FROM BOD_TAB_PROCEDIMIENTO WHERE ID_CATEGORIA=11");


(d) Procesar los registros
String patologia="", area="";
int icono=0;

while ( rs.next() ) {
        patologia = rs.getString("NOMBRE_PROCEDIMIENTO");
        icono = rs.getInt("ICONO_PROCEDIMIENTO");
        area = rs.getString("AREA_OCUPA");
        System.out.println(patologia + ", " + icono + ", " + area);  // Visualizar los datos
}
```

ORM

a) Configurar el archivo de acceso. Ejemplos: persistence.xml para JPA, hibernate.cfg para Hibernate.
b) Obtener conexión a la base de datos a partir del archivo de configuración.
c) Crear la instancia para administrar los objetos persistentes.
d) Definir el mapeo, para lo cual debe:
    - Identificar clases persistentes y no persistentes
    - Especificar atributos persistentes y sus relaciones.
e) Gestionar las clases persistentes a través de consultas o métodos.

Ejemplo de interacción con la RDBMS Oracle usando ORM, particularmente el framework JPA: Recupera lista de patologías de la tabla BOD_TAB_PROCEDIMIENTO.

```
(a)  Configuración del archivo de acceso, persistence.xml


<?xml version="1.0" encoding="UTF-8"?>
<persistence version="1.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
  <persistence-unit name="DentalUTNPU" transaction-type="RESOURCE_LOCAL">
    <provider>oracle.toplink.essentials.ejb.cmp3.EntityManagerFactoryProvider</provider>
...
.
.
    <class>com.dbu.odontologico.model.entities.BodTabProcedimiento</class>
    <class>com.dbu.odontologico.model.entities.BodTabCategoria</class>
...
.
.
    <properties>
      <property name="toplink.jdbc.url" value="jdbc:oracle:thin:@localhost:1521:GENERAL"/>!— URL para Oracle --
      <property name="toplink.jdbc.user" value="BIENESTAR"/> !--Nombre de usuario con acceso a la bdd--
      <property name="toplink.jdbc.driver" value="oracle.jdbc.driver.OracleDriver"/> !-- Driver --
```

```
      <property name="toplink.jdbc.password" value="DBU"/> !--Contraseña de dicho usuario--
    </properties>
  </persistence-unit>
</persistence>
```

(b) Establecer conexión a la base de datos. En este caso se crea una instancia del EntityManagerFactory usando el archivo de configuración

```
EntityManagerFactory  emf = Persistence.createEntityManagerFactory("DentalUTNPU ");
```

(c) Crear la instancia para administrar los objetos persistentes. Para el ejemplo el EntityManager.

```
EntityManager em = emf.createEntityManager();
```

(d) Definición del mapeo

```
//Clase persistente BodProcedimiento.java
package com.dbu.odontologico.modelo.entities;

import java.io.Serializable;
import java.util.Collection;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToMany;
import javax.persistence.Table;

//Indica que es una entidad
@Entity
@Table(name = "BOD_TAB_PROCEDIMIENTO") //Tabla a mapear
public class BodProcedimiento implements Serializable {
    @Id //Indica clave primaria de la clase persistente
    @Column(name = "id_procedimiento", nullable = false)
    private Short idProcedimiento;
    @Column(name = "nombre_procedimiento")
    private String nombreProcedimiento;
    @Column(name = "icono_procedimiento")
    private Short iconoProcedimiento;
    @Column(name = "area_ocupa")
    private Character areaOcupa;

    @JoinColumn(name = "id_categoria", referencedColumnName = "id_categoria")
    @ManyToOne
    private BodCategoria idCategoria;

    @OneToMany(cascade = CascadeType.ALL, mappedBy = "bodProcedimiento")
    private Collection<BodOdontograma> bodOdontogramaCollection;
```

Atributos persistentes.

Relaciones entre Clases Persistentes

```
    //Creación de una nueva instancia de BodProcedimiento. Constructor sin argumentos, requisito de una entidad.
    public BodProcedimiento() {
    }
    public BodProcedimiento(Short idProcedimiento) {
        this.idProcedimiento = idProcedimiento;
    }

//Métodos get y set de la clase persistente
    public Short getIdProcedimiento() {
        return this.idProcedimiento;
    }
    public void setIdProcedimiento(Short idProcedimiento) {
        this.idProcedimiento = idProcedimiento;
    }
    public String getNombreProcedimiento() {
        return this.nombreProcedimiento;
    }
    public void setNombreProcedimiento(String
nombreProcedimiento) {
        this.nombreProcedimiento = nombreProcedimiento;
    }
    public Short getIconoProcedimiento() {
        return this.iconoProcedimiento;
    }
    public void setIconoProcedimiento(Short iconoProcedimiento) {
        this.iconoProcedimiento = iconoProcedimiento;
    }
    public Character getAreaOcupa() {
        return areaOcupa;
    }
    public void setAreaOcupa(Character areaOcupa) {
        this.areaOcupa = areaOcupa;
    }
    public BodCategoria getIdCategoria() {
        return this.idCategoria;
    }
    public void setIdCategoria(BodCategoria idCategoria) {
        this.idCategoria = idCategoria;
    }
    public Collection<BodOdontograma>
getBodOdontogramaCollection() {
        return this.bodOdontogramaCollection;
    }
    public void setBodOdontogramaCollection(Collection<BodOdontograma> bodOdontogramaCollection) {
        this.bodOdontogramaCollection = bodOdontogramaCollection;
    }
```

Métodos de acceso para atributos persistentes

```
    // Generación de identificador único para el objeto. Utiliza método hash basándose en clave primaria
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (this.idProcedimiento != null ? this.idProcedimiento.hashCode() : 0);
        return hash;
    }

   //Determina si otro objeto es igual al objeto BodProcedimiento.
    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not set
        if (!(object instanceof BodProcedimiento)) {
            return false;
        }
        BodProcedimiento other = (BodProcedimiento)object;
        if (this.idProcedimiento != other.idProcedimiento && (this.idProcedimiento == null ||
!this.idProcedimiento.equals(other.idProcedimiento))) return false;
        return true;
    }
}
```

(e) Gestionar las clases persistentes. Ejecuta la consulta y obtiene el resultado en forma de un objeto Java

```
List listaPatologias= em.createQuery("select o from BodProcedimiento o where o.idCategoria.idCategoria =
11").getResultList(); //Consulta JPQL

// Mostrar resultados, todas las patologías
BodProcedimiento pat = new  BodProcedimiento(); //Instancia un objeto de la clase persistente
for(int i=0; i<listaPatologias.size(); i++){
        pat = (BodProcedimiento) listaPatologias.get(i);
        System.out.println(pat.getNombreProcedimiento() + ", " + pat.getIconoProcedimiento() + ", " +
pat.getAreaOcupa()); // Visualizar los datos
}
```

Conclusión de Facilidad

Con JDBC el establecer directamente en el código fuente la relación entre los atributos del objeto y las columnas de la base de datos además de ser una tarea muy laboriosa, no es para nada escalable ya que cualquier modificación sobre la clase requerirá buscar y modificar la parte del código fuente apropiada, añadiendo nuevos métodos, cambiando el esquema de la base de datos, etc. Si además queremos disponer de una mayor funcionalidad aparte de la correspondencia clase-objeto, la labor de desarrollo llevaría aún más tiempo. Mientras que usando ORM una vez dispuestos los ficheros de mapeo, se puede trabajar con la misma facilidad con la que se codifica con cualquier librería Java.

4.2.2 Productividad

El número de líneas de código para materializar la funcionalidad necesaria, se configura como un indicador del esfuerzo de desarrollo que permite la comparación entre cada una de las alternativas consideradas. Menos líneas de código, significa menor tiempo de desarrollo, mayor facilidad de mantenimiento, menos costes de desarrollo y consecuentemente mayor productividad.

A continuación se analizará la reducción en el número de líneas contrastando la manera de consultar y actualizar la base Datos de un ORM frente a JDBC.

4.2.2.1 Consultar la base de Datos

JDBC

Para recuperar información de una de base de datos se utiliza la clase Statement , cuyos objetos se crean a partir de una conexión y tienen el método executeQuery para ejecutar consultas SQL de tipo SELECT devolviendo como resultado un conjunto de registros en un objeto de la clase ResultSet.

Supongamos que se tiene una conexión a la base de datos BIENESTAR cuyas tablas son BSA_HISTORIA_MEDICA y BOD_CONSULTA_ODONTOLOGICA y que queremos obtener los registros de la tabla BOD_CONSULTA_ODONTOLOGICA de un paciente, el código sería:

```
...
//Obtener Consultas
try {
            st =con.createStatement();
            rs = st.executeQuery("SELECT *FROM BOD_TAB_CONSULTA_ODONTOLOGICA WHERE NUMERO_HISTORIA = "
            + historia);
            System.out.println("Tabla abierta");
            ...
} catch (SQLException e) {
System.out.println("Error al Abrir tabla ");
...
```

Una sentencia creada de esta forma devuelve un ResultSet en el que sólo puede haber desplazamiento hacia adelante. Para luego acceder al conjunto de registros que se encuentran en el ResultSet, es necesario el método next del objeto ResultSet para movernos por los registros y métodos específicos para extraer la información de cada tipo de campo con la forma getXXXX.

Para cada método getXXXX, el driver JDBC debe hacer conversiones entre el tipo de la base de datos y el tipo Java equivalente. El driver no permite conversiones inválidas aunque si permite que todos los tipos puedan ser leídos desde Java como cadenas con el método getString.

Por otra parte, cuando se recorre el ResultSet es necesario conocer cuando se llega al final del mismo y esto se controla con el método next que además de moverse al siguiente registro, devuelve falso cuando se ha pasado del último registro.

```
...
//Mostrar datos de consulta odontológica
try {
        while (rs.next()) {
        int numHistoria = rs.getInt("NUMERO_HISTORIA");
        String motivoCons = rs.getString("MOTIVO_CONSULTA")
        String evolucion = rs.getString("EVOLUCION");
        System.out.println(numHistoria + ", " + motivoCons + ", " + evolucion);
        }
} catch (Exception e) {
        System.out.println("Error al visualizar datos");
}
...
```

ORM

Para ejecutar una consulta de los objetos almacenados en la base de datos ORM proporciona varias maneras:

- API de consulta.
- Lenguaje de consulta orientado a objetos.

```
//API de consulta
@Entity
@Table(name = "BOD_TAB_CONSULTA_ODONTOLOGICA")
@NamedQueries( {
@NamedQuery(name = "BodConsultaOdontologica.findByNumeroHistoria", query = "SELECT d FROM BodConsultaOdontologica d WHERE d.bodConsultaOdontologicaPK.numeroHistoria = :numeroHistoria"),
...

//Consulta estática
public List listarConsultas (BigDecimal historia){
Query q = em.createNamedQuery("BodConsultaOdontologica.findByNumeroHistoria"); //Llama a la consulta
q.setParameter("numeroHistoria", historia); //Asigna parámetro
return  q.getResultList(); //Devuelve datos de la consulta, una lista.
}

//Lenguaje de Consulta
public List listarConsultas (BigDecimal historia){
Query q = em.createQuery("select o from BodConsultaOdontologica as o WHERE o.bodConsultaOdontologicaPK.numeroHistoria = :numeroHistoria");
q.setParameter("numeroHistoria", historia);
return  q.getResultList(); //Devuelve datos de la consulta, una lista.
}

//Mostrar consultas odontológicas usando API de Consulta o Lenguaje de Consulta
...
List consultas = cof.listarConsultas(num_hc);
        if(consultas!=null){
          for( i=0; i< consultas.size(); i++){
              co = (BodConsultaOdontologica)consultas.get(i);
              System.out.println(num_hc  + "," + co.getMotivoConsulta() + "," + co.getEvolucion());
          }}
...
```

Observamos aquí que las consultas se formulan en base a objetos, pedimos objetos y expresamos las condiciones de la consulta objetualmente.

De acuerdo a lo expuesto JDBC se diferencia de ORM, en la codificación de la operación que muestra las consultas odontológicas. La diferencia esencial es el aumento de líneas de código necesarias para lograr recuperar el estado de los objetos con JDBC, hecho de relevancia para la productividad.

4.2.2.2 Consultas de Actualización

JDBC

Para actualizar la base de datos con sentencias SQL de tipo UPDATE, INSERT o DELETE, es necesario al igual que en el caso de SELECT, tener una Connection y crear una Statement a partir de la misma. La diferencia es que en vez de llamar al método executeQuery para ejecutar la consulta, se llama al método executeUpdate que no devuelve un ResultSet como resultado, sino que devuelve la cantidad de registros afectados. A continuación se muestran varios métodos con ejemplos de este tipo de consultas.

```
/*Método para modificar la tabla BOD_TAB_CONSULTA_ODONTOLOGICA pasando el número de historia, id de consulta que se quiere modificar y el tipo de consulta. Las cadenas en la condición es necesario ponerlas entre comillas simples. */

public void modificar(int historia, int idCons, String tipoConsulta)
{
	try {
		Statement s2=con.createStatement();
		s2.executeUpdate("Update BOD_TAB_CONSULTA_ODONTOLOGICA set TIPO_CONSULTA ='"+ tipoConsulta + "' where NUMERO_HISTORIA = " historia + " and ID_CONSULTA_ODONTOLOGICA =" + idCons);
		System.out.println("Elemento modificado correctamente");
	}catch (SQLException e) {
		System.out.println("Error al modificar");
	}
}

/*Método para borrar la consulta odontológica cuya historia e id se pasa como argumento */
public void borrar(int historia, int idCons)
{
	try{
		Statement s2=con.createStatement();
		s2.executeUpdate( "DELETE FROM BOD_TAB_CONSULTA_ODONTOLOGICA where NUMERO_HISTORIA = " historia + " and ID_CONSULTA_ODONTOLOGICA =" + idCons);
		System.out.println("Elemento Borrado");
	}catch(SQLException e) {
		System.out.println("Error al Borrar");
	}
}

/*Método que permite insertar un nuevo registro en la tabla BOD_TAB_CONSULTA_ODONTOLOGICA, pasándole como argumento número de historia, id, fecha, tipo de consulta, estado */
public void insertar(int historia, int idCons, String tipoConsulta, String estado) {
	try{
		Statement s1 = con.createStatement();
		s1.executeUpdate( "INSERT INTO BOD_TAB_CONSULTA_ODONTOLOGICA (NUMERO_HISTORIA, ID_CONSULTA_ODONTOLOGICA, FECHA_CONSULTA_ODONTOLOGICA, TIPO_CONSULTA, ESTADO) values (" + historia + "', " + idCons + ", '" + fecha + "'," + tipoConsulta + ",'" + estado + "')");
		System.out.println("Elemento insertado");
	} catch(SQLException e) {
		System.out.println("Error al insertar ");
	}

}
```

ORM

Para actualizar un ORM se sirve de métodos, comprobando el campo clave para identificar el registro que se va a actualizar y sustituyendo todos los valores que haya en la tabla por los que se encuentran poblando los atributos del objeto. Lo mismo sucede con instrucciones de inserción y borrado.

```
//Inserción
public void insertarConsulta(BodConsultaOdontologica co){
        try{
            em = EntityManagerHelper.getEntityManager();
            em.getTransaction().begin();
            em.persist(co);
            em.flush();
            em.getTransaction().commit();
        }finally{
            em.close();
        }
}

//Actualización
public void editarConsulta(BodConsultaOdontologica co){
        BodConsultaOdontologica consp = new BodConsultaOdontologica();
        try{
            em = EntityManagerHelper.getEntityManager();
            em.getTransaction().begin();
            em.flush();
            consp = em.merge(co);
            em.getTransaction().commit();
        }finally{
            em.close();
        }
}

//Borrado
public void borrarConsulta (BodConsultaOdontologicaPK consultaOdontologicaPK) {
BodConsultaOdontologica cons = em.find(BodConsultaOdontologica.class, consultaOdontologicaPK);
em.remove(cons);
}
```

Conclusión de Productividad

La persistencia manual se caracteriza por la utilización del SQL, como fórmula para acceder a los datos, esto implica que el desarrollador tendrá que codificar a mano como mínimo todas las operaciones de consulta, actualización y borrado que se realicen sobre los objetos. Con ORM, el acceso transparente evita la necesidad de utilizar un lenguaje de consulta por todas partes, solo cuando es necesario en la localización de los objetos iniciales, desde los cuales se alcanzan otros con la navegación por las referencias.

4.2.3 Rendimiento

Es de esperar que con un modelo de objetos muy simple, un ORM debiera ser más lento que la solución JDBC sin modelo de objetos, debido a que JDBC no tiene la sobrecarga de un ORM, con sus cachés, control de modificaciones, etc. No obstante, en los modelos complejos es donde los mecanismos para mejorar el rendimiento desempeñan su papel, superando a una aplicación JDBC no preparada.

4.2.4 Curva de aprendizaje

JDBC

Requiere tener sólidos conocimientos de lenguaje SQL.

ORM

El aprendizaje de un ORM supone más trabajo, sobre todo si se quiere aprovechar todo el potencial, sin embargo es relativamente fácil dada la documentación disponible.

Conclusión de Curva de aprendizaje

Los costos de aprendizaje de un ORM en términos de tiempo dedicado serán recompensados desde el primer día con el ahorro en tiempo de desarrollo y depuración de código JDBC, lo cual permite centrar esfuerzos en desarrollar la funcionalidad de la aplicación.

4.3 Análisis

Comencemos analizando JDBC como solución. El mayor beneficio es el rendimiento dado que programado adecuadamente, con store procedures y al no tener capas intermedias que pasar, el rendimiento aumenta a sacrificio de muchas características que son de suma importancia en una arquitectura empresarial. La aplicación es dependiente de la base de datos esto se podría solucionar con el patrón DAO dado que encapsulando el acceso de datos en un objeto, cambiando el DAO podemos modificar la Base de datos, pero de igual forma deberíamos programar todo el objeto de acceso a datos, dado que el lenguaje SQL

no es estándar para todas las bases de datos. Usando solamente JDBC, con el patrón DAO o sin él, es muy costoso cambiar de base de datos, aunque con el patrón DAO tenemos más independencia con la base de datos.

Con JDBC tenemos que enfrentarnos con la diferencia de impedancia solos, es decir, que tenemos que programar el SQL para acceder a los registros de la base de datos, convertir los registros en objetos llenando cada propiedad del objeto con los campos del registro recuperado.

El uso de JDBC y consecuentemente de SQL es más eficiente, pero no es flexible, ni portable, ni escalable. Por lo que es preferible utilizarlo en una aplicación pequeña que no necesita las características de una aplicación empresarial.

JDBC es una API de bajo nivel y sus prestaciones son para funciones de bajo nivel. Cuando el modelo de datos es simple la solución más fácil es JDBC

Los frameworks ORM ya están consolidados, probados en el mercado y existen muchas herramientas para desarrollar con estos frameworks. La mayoría de ellos permiten la definición de las entidades generalmente a través de ficheros XML o anotaciones. A partir de esa definición los ORM son capaces de extraer la definición de esquema de la base de datos necesaria para representar ese modelo de objetos. Los ORM permiten el mapeo de estos esquemas de base de datos a objetos de modo que a partir de las tablas de base de datos se pueden generar las clases Java, necesarias para modelar dicho esquema con todas las relaciones de jerarquía que estén presentes en el mismo.

Obviamente esto supone una ventaja grandísima ya que la cantidad de código necesaria para realizar todas esas funciones es realmente considerable. Además, de este modo, se puede cambiar de manera sencilla el esquema de base de datos modificando el fichero XML y en un momento tener toda la estructura de clases Java y la base de datos actualizada con el mínimo coste de desarrollo para el programador que tan sólo tendrá que preocuparse por añadir lógica de negocio para soportar los cambios.

Para obtener un nivel de abstracción y facilitar la programación, es aconsejable utilizar un framework de persistencia.

4.4 Conclusiones Finales

- JDBC define una solución de bajo nivel para acceder a la base de datos. No funciona al nivel de objetos Java. Los desarrolladores que usamos JDBC debemos escribir métodos que contienen declaraciones SQL y convierten el registro de la base de datos a objetos. ORM no intenta reemplazar JDBC, pero hace uso eficaz de él y aísla a los desarrolladores de los detalles de bajo nivel propios de JDBC. Desarrolladores de la aplicación trabajan con objetos que usan frameworks de persistencia, en lugar de las filas y SQL que usa las llamadas de JDBC.

- ORM es una técnica con la que es más fácil programar la persistencia que con JDBC solo. Con ORM se necesitan menos líneas de código, de un menor esfuerzo de programación, para una gran mayoría de casos, a excepción de cuando hay que elaborar consultas SQL complicadas. El uso de ORM no excluye usar JDBC, cuando sea preciso y conveniente.

- El acceso mediante JDBC conviene utilizarlo cuando tenemos pocas clases, escasos conocimientos, escaso tiempo para formación, modelo de datos muy desnormalizado. Para todo lo demás ORM es una solución madura y dinámica que nos permite simplificar a la vez que robustecer nuestro acceso a bases de datos.

- Si hablamos de curva de aprendizaje y rapidez de desarrollo, ORM es un claro ganador, la curva es relativamente corta, las ventajas evidentes y los tiempos de desarrollo, líneas de código y archivos que se necesitan para que la aplicación funcione se reducen considerablemente.

CAPITULO V

Análisis, Diseño e Implementación del "Sistema de Gestión de Información Odontológica para el Departamento de Bienestar Universitario de la UTN"

"El fin de una etapa es sólo el comienzo de otra, los riesgos sorteados son la preparación necesaria para pasar mejor la próxima etapa" Paulo Coelho.

5.1 Introducción

En este capítulo se trata el desarrollo del Sistema de Gestión de Información Odontológica aplicando el tema de investigación descrito en este documento, ORM. Dicho desarrollo se basa en la metodología RUP[14] de la que se destaca los artefactos[15] esenciales que contribuyen a la culminación del proyecto.

5.2 Estudio de Viabilidad

5.2.1 Antecedentes

El Departamento de Bienestar Universitario de la Universidad Técnica del Norte es una dependencia que contribuye al desarrollo socio económico, académico y de salud del personal estudiantil, docente y administrativo de la institución.

Esta dependencia busca prestar servicios a los miembros de la comunidad universitaria con personal idóneo, para ser un punto de apoyo en el logro del mejoramiento de la calidad de vida y la búsqueda de la excelencia académica.

Coherente con el objetivo de la universidad de formar integralmente nuevos profesionales que deberán ser los protagonistas del desarrollo de nuestro país, cumple un rol fundamental. Esta acción de carácter primordial se traduce básicamente en la promoción, organización y realización de actividades de salud y formación humana.

Diariamente el Departamento de Bienestar presta sus diversos servicios, debiendo mantener información exclusiva de cada una de las personas que acuden, como son: estudiantes (7106), docentes (326), empleados (324) y familiares (412)[16]; cuyo volumen de información representa una gran responsabilidad.

Tomando en consideración lo anterior, es necesario contar con herramientas de software que garanticen el correcto tratamiento de la información y ayuden a proporcionar un buen servicio.

[14] RUP: Rational Unificate Proccess. Metodología de desarrollo de Software.
[15] Artefacto: Entregables del proyecto.
[16] Datos extraídos del Sistema de Recaudación SARE.

5.2.2 Descripción del Problema

El Departamento de Bienestar Universitario está conformado por cuatro áreas: Salud, Asistencia Social, Orientación Profesional, Orientación Académica.

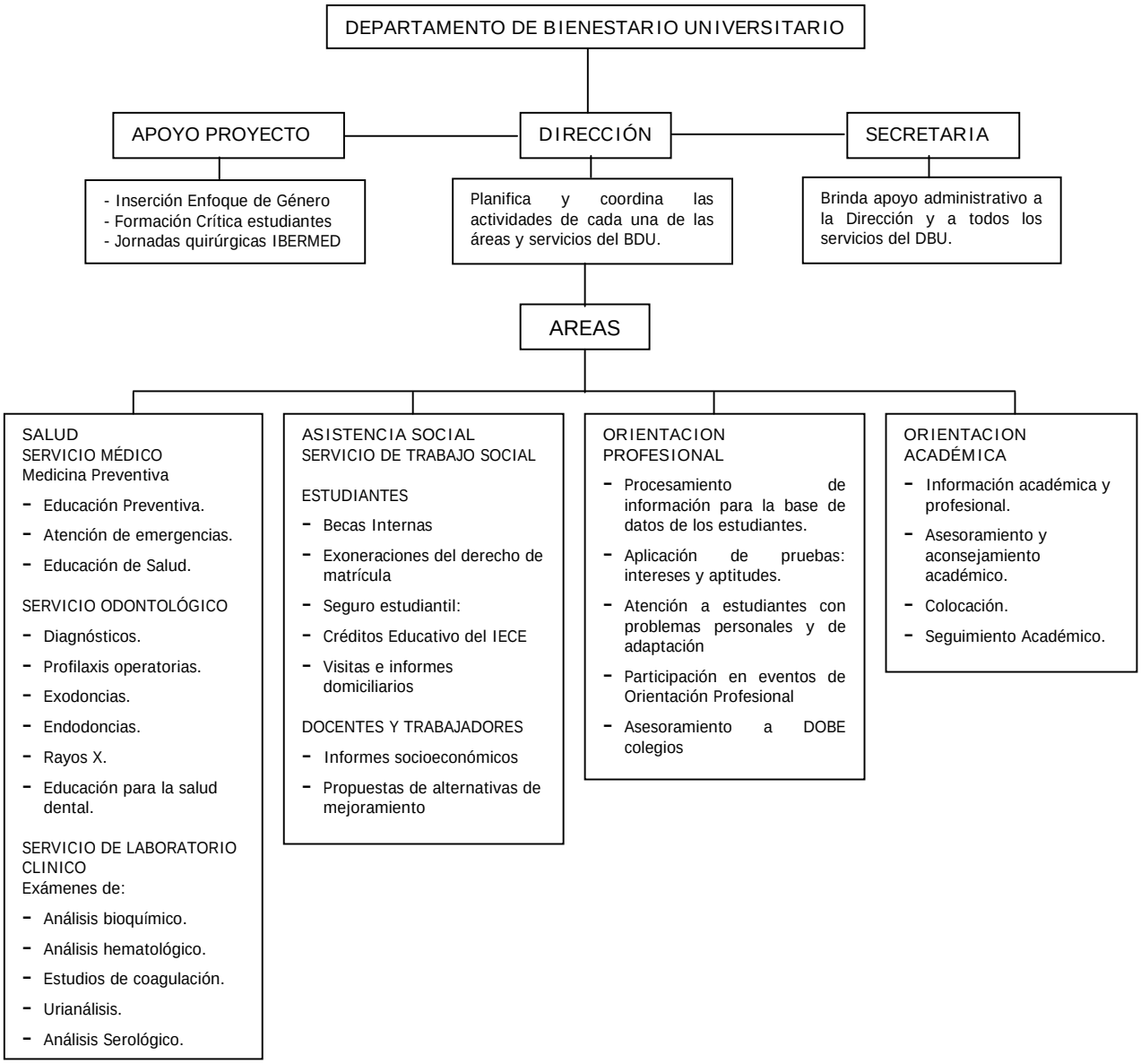


Figura 5.1 Estructura Organizativa del Departamento de Bienestar Universitario

ÁREA DE SALUD

El área de Salud a su vez está estructurado de la siguiente forma: Servicio Médico, Servicio Odontológico y Servicio de Laboratorio Clínico.

Esta área presta atención personalizada a estudiantes, docentes, personal administrativo y sus familiares, tanto del colegio como de la universidad.

El área de Salud maneja toda la información de sus pacientes en formularios, hojas de trabajo, registros diarios, registros mensuales y anuales manualmente; además lleva un registro de ingresos y egresos de medicamentos y materiales mediante cartones de kardex. La inexistencia de un sistema automatizado en esta área, ha ocasionado una serie de inconvenientes tales como:

- Información aislada y redundante.- Al no existir un control de historias clínicas automatizado impide el acceso a la información que mantiene el servicio de Laboratorio Clínico el cual cuenta con un sistema de ingreso de resultados. Además la información existente en el sistema de Ficha Psicosocial es redundante a la que contiene la ficha historia clínica.

- Falta de agilidad en la atención al paciente.- La historia clínica de cada paciente está constituida por varios formularios llenados a mano y archivados en un mueble, de donde son extraídos al momento de iniciar la consulta. Este tedioso proceso requiere mayor atención y tiempo, ocasionando molestias tanto al personal del departamento como a los pacientes.

- Dificultad en la generación de informes y estadísticas.- Al no existir un proceso automático para registrar la atención al paciente en el departamento se lleva diariamente un registro manual que luego es contabilizado para informes mensuales, lo que requiere gran cantidad de tiempo. Además para la generación de cuadros estadísticos, es necesario que el personal del departamento obtenga los datos examinando cada uno de los formularios de historias clínicas de los pacientes existentes; siendo ésta una tarea compleja y poco precisa.

- Pérdida de información.- Los datos de historias clínicas y los resultados de exámenes es información confidencial y de vital importancia, sin embargo es manipulada por varias personas, y archivada en lugares poco propicios, lo cual facilita su pérdida.

- Desconocimiento de medicamentos y materiales existentes.- El servicio médico y odontológico manejan cierta cantidad de medicamentos de emergencia, los cuales son contabilizados manualmente mediante kárdex sin embargo no se lleva un control adecuado del mismo dado que los datos no son actualizados

constantemente. Además el servicio de laboratorio y el odontológico manejan materiales de uso exclusivo los mismos que no son inventariados. [DOC02]

5.2.3 Propuesta de Desarrollo en Base a Requerimientos

De acuerdo al problema descrito anteriormente, a continuación se detallan los módulos de solución para el servicio odontológico.

Historia Clínica Odontológica

- Obtendrá datos del módulo de Servicio Médico como: datos personales, antecedentes patológicos y signos vitales.
- Registro de anamnesis.
- Registro del examen bucal.
- Registro de datos de endodoncia y oclusión.
- Creación del odontograma inicial al momento de la primera consulta y modificaciones para las subsecuentes.
- Emisión de prescripción y certificados.
- Registro de imágenes de RX.

Inventario de Materiales

- Sistema de inventario de material odontológico.

Reportes y Estadísticas

- Generación automática de reportes y estadísticas de acuerdo a las necesidades del departamento como pacientes atendidos por fechas, por tipo de paciente (estudiante, empleado, docente y familiar), por edades, por sexo, por patologías, etc.

5.2.4 Requisitos Tecnológicos

De Telecomunicaciones:

Debido a que el área de Salud del DBU[17] no contaba con equipos de computación ni red de datos que soporte la implementación de un sistema se solicita la instalación de puntos de red Figura 5.2 y la adquisición de equipos Tabla 5.1.

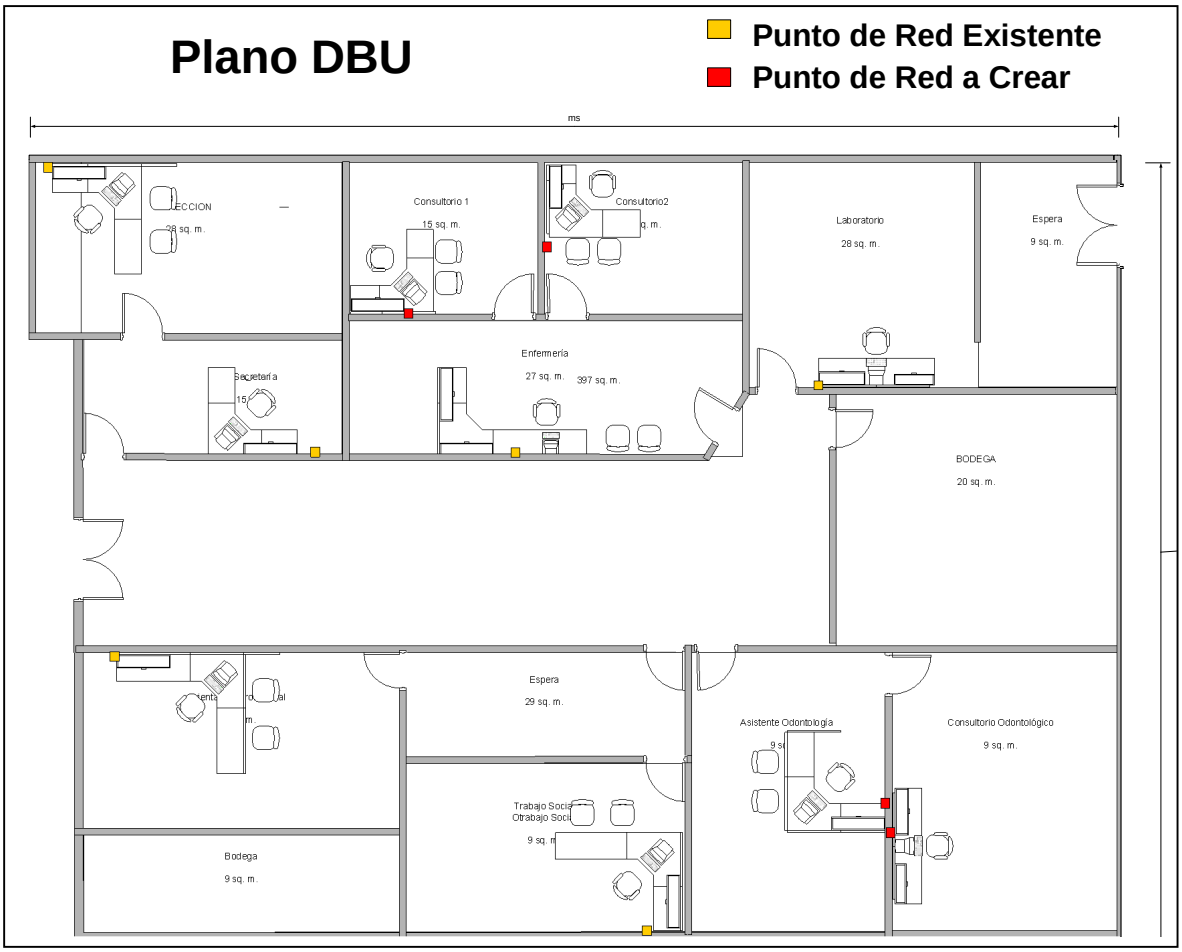


Figura 5.2 Cableado estructurado del DBU.

Recurso	Cantidad
Equipos de Computación	5
Impresoras Láser	7
Escáner	1

Tabla 5.1 Recursos para el DBU.

[17] DBU: Departamento de Bienestar Universitario.

De Hardware:

La siguiente tabla muestra las características principales mínimas que los equipos de computación requieren para la implementación del sistema.

Equipo	Servidor	Clientes
Procesador	Intel Core 2 Duo 2.33 Ghz	Intel Pentium IV 2.8 Ghz
Memoria	1 Gb	512 Mb
Disco Duro	250 Gb	80 Gb
Tarjeta de Red	Sí	Sí

Tabla 5.2 Características de Hardware para el DBU.

De Software:

Para el desarrollo e implementación del sistema se plantea inicialmente:

	Servidor	Clientes
Sistema Operativo	Windows 2003 Server	XP
Base de Datos	PostgreSQL 8.1	No
Servidor de Aplicaciones	Apache Tomcat 5.5.17	No
Software Base	Sí	Sí

En la etapa de análisis se establece el software definitivo, principalmente en base de datos.

5.2.5 Plan de Desarrollo

A continuación se presenta el plan tentativo de inicio y finalización del proyecto.

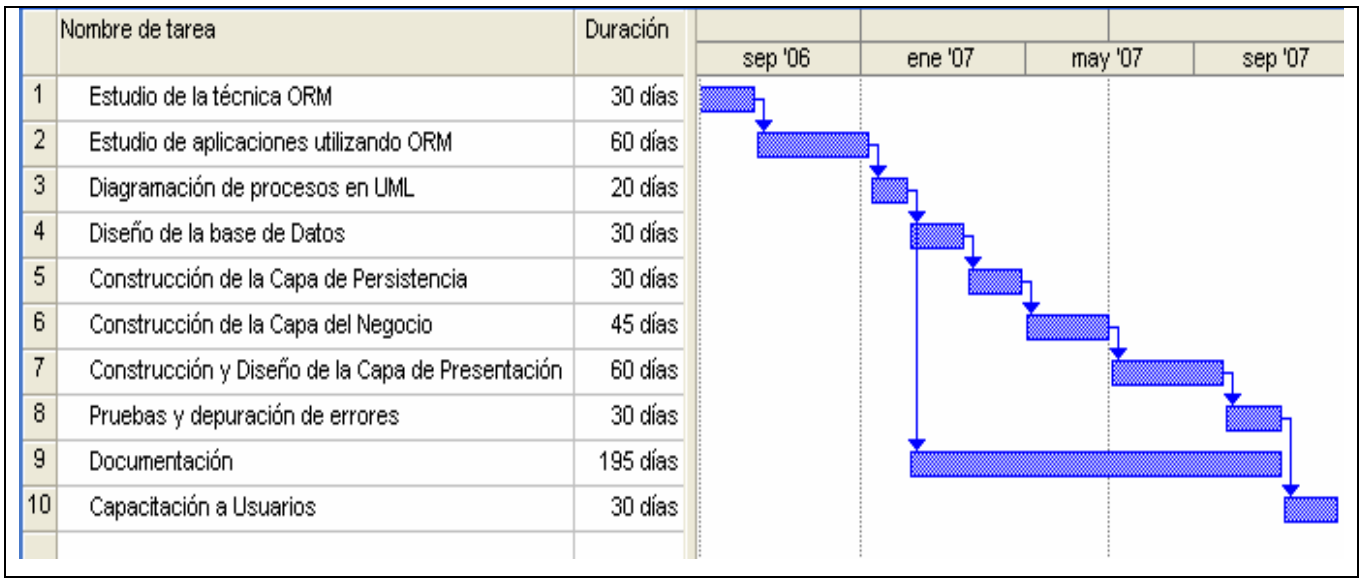


Figura 5.3 Plan de Desarrollo del sistema.

Fecha Inicio: 04/09/2006

Fecha Fin: 14/12/2007

5.3 Análisis

5.3.1 Flujo de Trabajo

Conforme a los módulos definidos en la propuesta, el flujo para el servicio odontológico es el que se muestra a continuación.

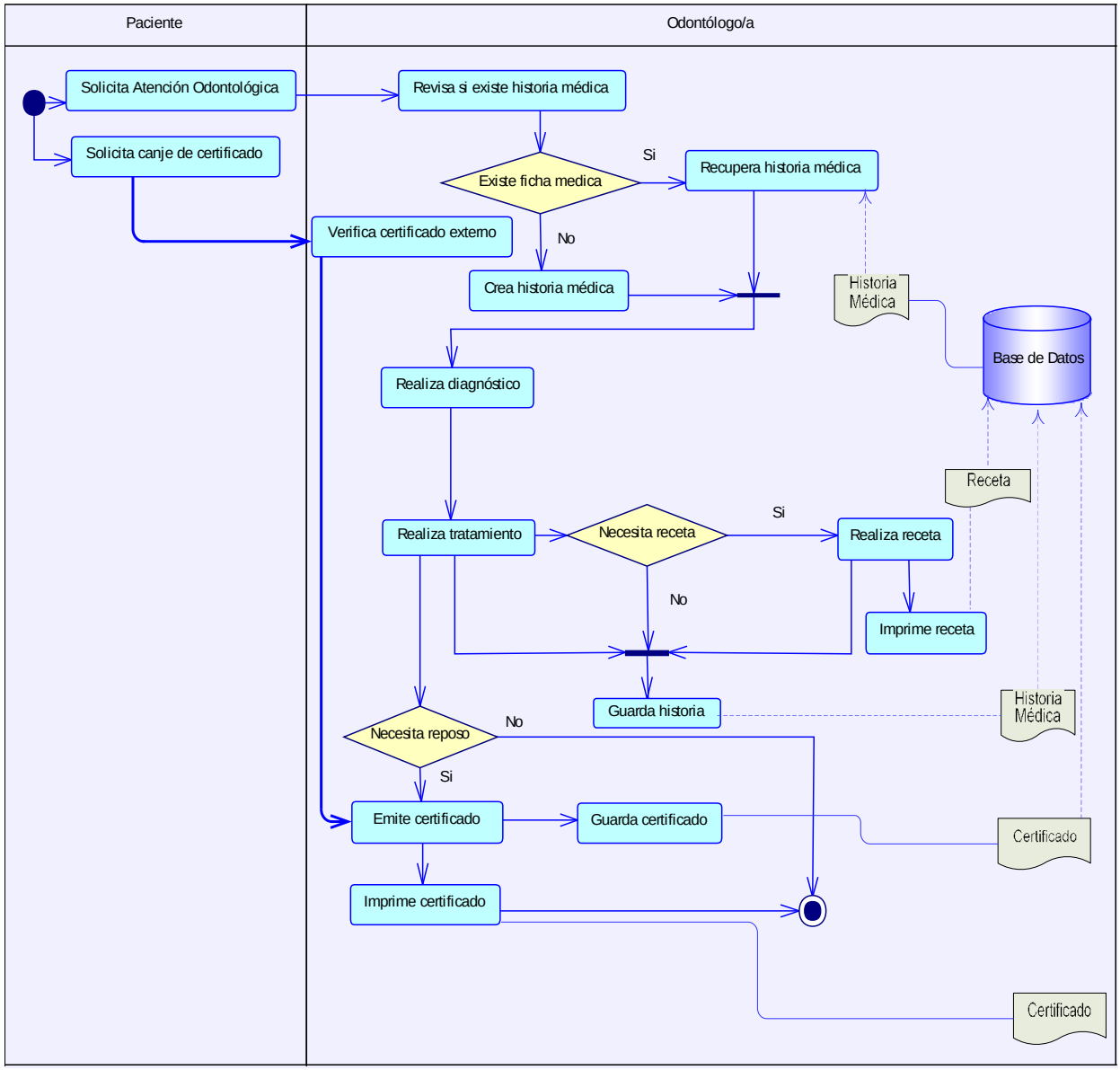


Figura 5.4 DFD Gestión Odontológica

5.3.2 Diagrama de Casos de Uso

Los diagramas de casos de uso definen un conjunto de funcionalidades que el sistema (o una clase) debe cumplir para satisfacer todos los requerimientos. Representan las funciones principales que la aplicación puede realizar.

A continuación se muestran las que corresponden al servicio odontológico.

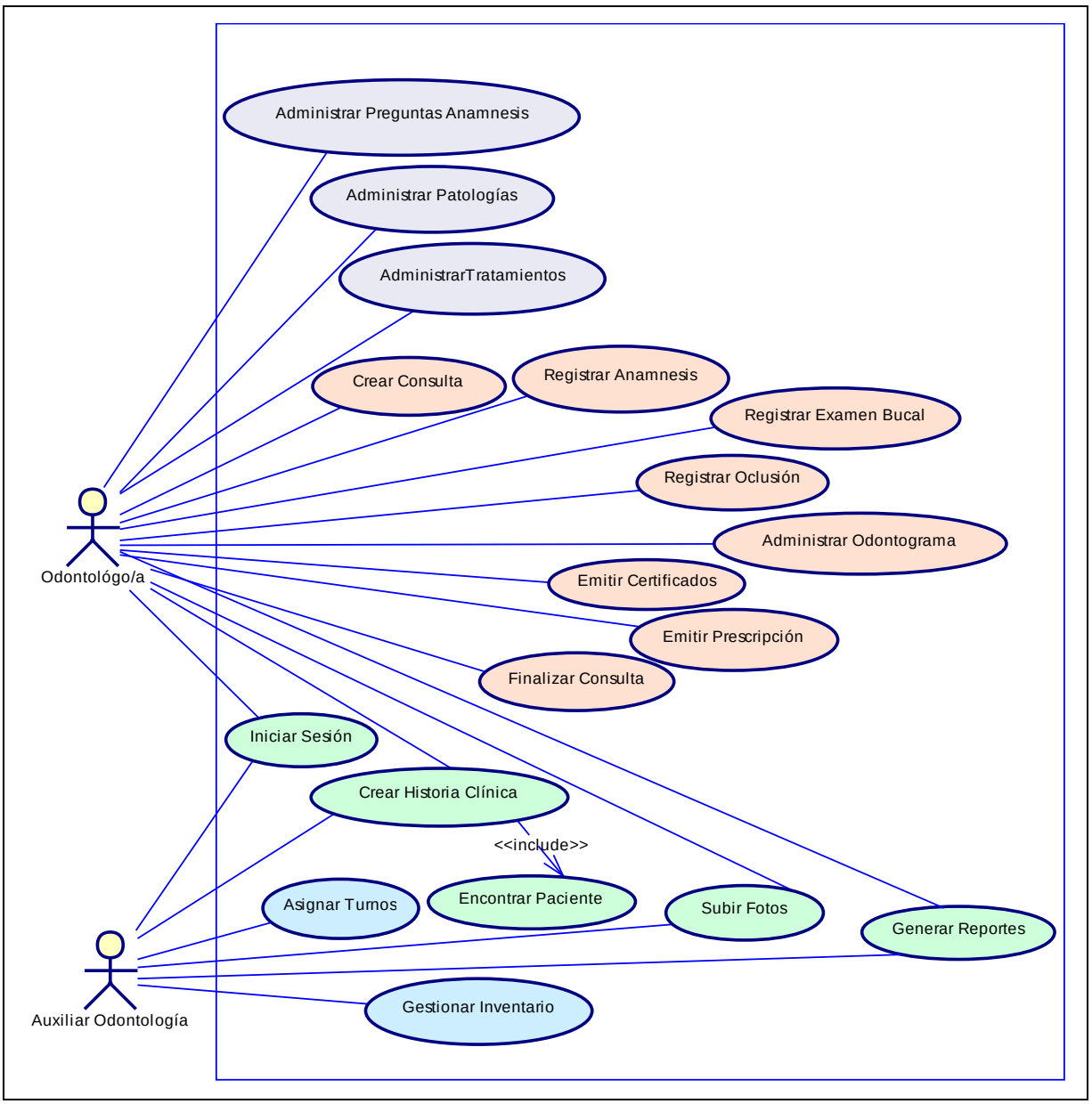


Figura 5.5 Diagrama de Caso de Uso del Servicio Odontológico

5.3.3 Tecnología

Una vez consolidados todos los requisitos, la tecnología que se usa para desarrollar e implementar el sistema es:

- Oracle 10g 10.2.0.1.0: Base de Datos.
- Toad for Oracle Version 8.5: Administrador de Base de Datos.
- ojdbc14.jar: Driver JDBC para Oracle.

En este punto es indispensable mencionar que inicialmente el sistema estaba siendo desarrollado con la base de datos PostgreSQL 8.1 sin embargo, se debió migrar a la base de datos Oracle 10g debido a que la Universidad Técnica del Norte en coordinación con el Departamento de Informática adquirió el licenciamiento de Oracle como plataforma tecnológica de la institución

- Apache Tomcat 5.5.17: Servidor de Aplicaciones.

- Netbeans 5.5: IDE de desarrollo.
- JVM 1.5.0_09: Máquina Virtual de Java.
- Java Persistence API con implementación de GlassFish Toplink Essentials: ORM
- iReport-2.0.4: Diseñador de reportes.
- Jasper Report: Librería para compilar reportes.

- Macromedia Fireworks 8: Tratamiento de imágenes, especialmente para la elaboración de iconos.

Con respecto a lenguajes:

- Java: Lenguaje de programación Orientado a Objetos.
- JSP: Contenido web dinámico.
- JavaScript: Lenguaje de programación para la validación de formularios.
- Css: Hojas de estilo.

5.3.4 Arquitectura de la Aplicación

Patrones de Diseño

Durante el desarrollo de cualquier proyecto es de vital importancia el uso de patrones de diseño. El uso de patrones acelera el desarrollo del proyecto, y proporciona seguridad, ya que estamos trabajando con soluciones demostradas.

En este trabajo se especifica dos patrones: MVC y Facade.

Descripción de los patrones usados

MVC - Modelo Vista Controlador

MVC es un patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos. [WWW17]

Patrón Facade

Este patrón sirve para proveer de una interfaz unificada sencilla que haga de intermediaria entre un cliente y una interfaz o grupo de interfaces más complejas.

Facade puede:

- Hacer una librería de software más fácil de usar y entender, ya que Facade implementa métodos convenientes para tareas comunes.
- Hacer el código que usa la librería más legible, por la misma razón;
- Reduce la dependencia de código externo en los trabajos internos de una librería, ya que la mayoría del código lo usa Facade, permitiendo así mayor flexibilidad en el desarrollo de sistemas;
- Envuelve una colección mal diseñada de APIs con un solo API bien diseñado.

En la siguiente figura se muestra la arquitectura del sistema y enseguida se describe la interacción entre cada componente.

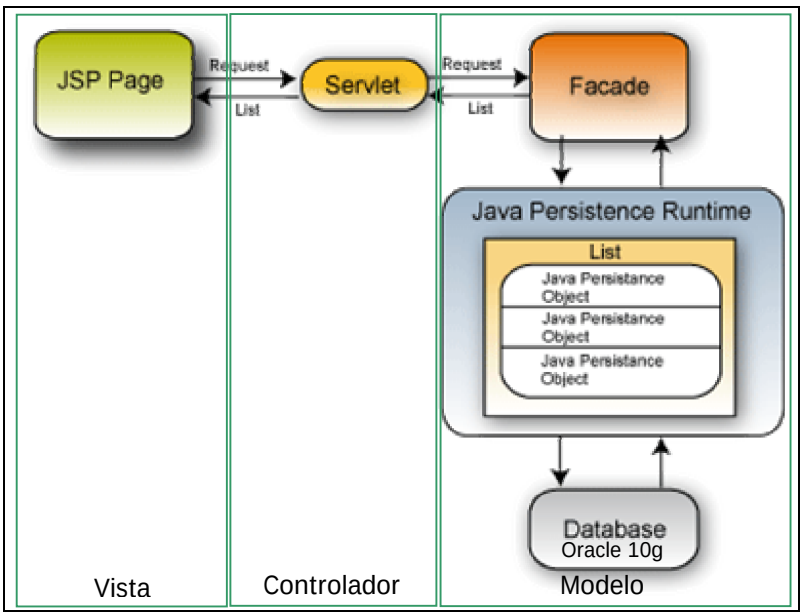


Figura 5.6 Diagrama que muestra la Arquitectura de la Aplicación DentalUTN.

- La página principal JSP envía la petición a un servlet.
- El servlet dirige la solicitud a un facade.
- El facade crea una consulta mediante JPQL para encontrar todos los datos en la base de datos.
- El Java Runtime Persistence ejecuta la consulta, realiza el mapeo objeto-relacional, construye las listas, y realiza algunas otras tareas.
- La base de datos ejecuta una consulta SQL y devuelve una lista de los resultados, cada uno de los cuales representa un elemento (es decir, una fila) en la base de datos.
- El facade pasa la lista al servlet.
- Finalmente, el servlet devuelve los resultados a la página JSP.

5.3.5 Diagrama de Componentes

Conforme a la arquitectura definida la siguiente figura muestra los componentes del sistema.

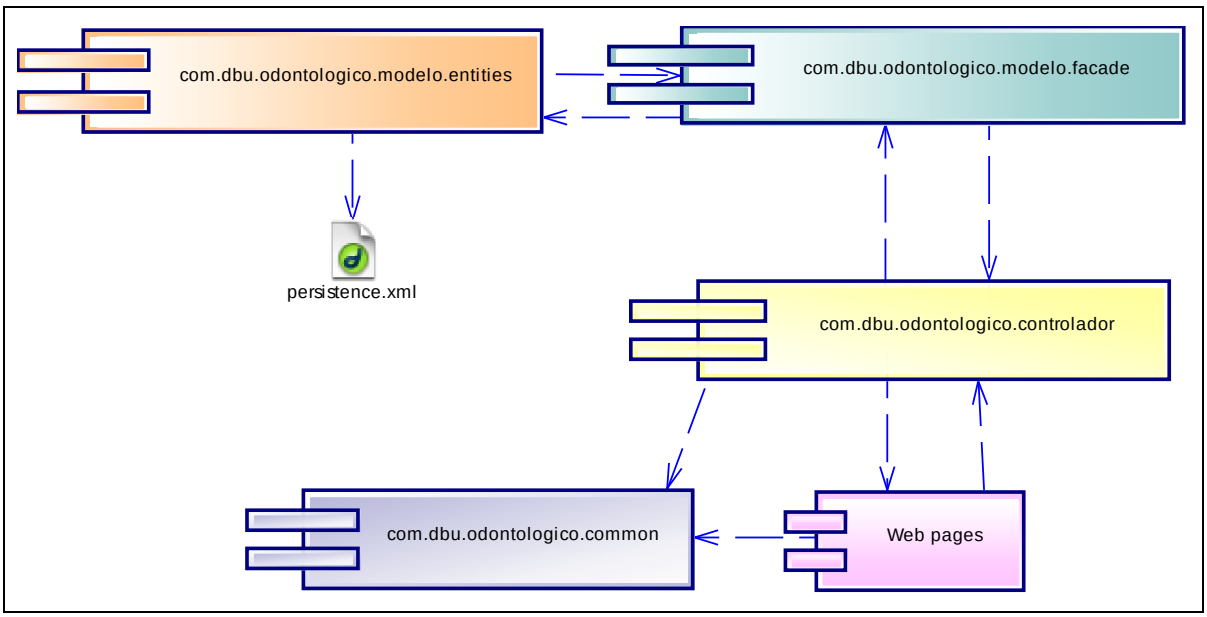


Figura 5.7 Diagrama de Componentes para la Aplicación DentalUTN.

El paquete com.dbu.odontologico.modelo.entities contiene todos los archivos que definen el mapeo. Algunos de los archivos son:

- InsPersona.java
- BsaHistoriaMedica.java
- BodPiezaDental.java
- BodOdontograma.java
- BodOdontogramaPK.java
- BodConsultaOdontologica.java
- BodCategoria.java
- BodCaraPiezaDental.java
- BodAnamnesisPK.java
- BodAnamnesis.java
- ImagenPaciente.java
- ImagenPacientePK.java

Pero principalmente incluye el archivo que administra todas las entidades denominado EntityManagerHelper.java el mismo que se sincroniza con el archivo de configuración de base de datos llamado persistence.xml

El paquete com.dbu.odontologico.modelo.facade contiene las clases que permiten interactuar entre la capa de modelo y el controlador. Ellas son:

- AnamnesisFacade.java

- ConsultaOdontologicaFacade.java
- HistoriaMedicaFacade.java
- ImagenFacade.java
- OdontogramaFacade.java
- PersonaFacade.java
- PreguntasAnamnesis.java
- PrescripcionFacade.java
- UsuarioFacade.java

El paquete com.dbu.odontologico.controlador contiene los servlets que implementan los procesos para la gestión de información odontológica. Enseguida se enumeran varias:

- Ficha.java
- HistoriaOdontologica.java
- GuardarArticulo.java
- GuardarConsultaOdontologica.java
- GuardarImagen.java
- GuardarHistoria.java
- Odontograma.java
- Patologias.java
- Tratamientos.java

El paquete com.dbu.odontologico.common contiene las clases de uso común para la aplicación, ellas son:

- Fecha.java
- Template.java
- Y Reporteador.java

Finalmente, el paquete Web pages contiene las páginas html y jsp que corresponden a la capa de presentación del sistema. Dichas páginas son:

- alarmasPaciente.jsp
- ampliarImg.jsp
- anamnesis.jsp
- articulos.jsp
- auxiliar.jsp
- buscar_paciente.jsp
- cerrarSesion.jsp
- certificados.jsp
- certificadosAnteriores.jsp
- configurarPatologias.jsp
- configurarPreguntasAnamnesis.jsp
- configurarTratamientos.jsp
- consultas.jsp
- consultasSuspendidas.jsp
- datos_personales.jsp
- egreso_articulo.jsp

- egreso_detalle_material
- egreso_material.jsp
- examen_bucal.jsp
- imagenesPaciente.jsp
- imprimir.jsp
- index.jsp
- kardexMateriales.jsp
- listarOclusiones.jsp
- listarPatologias.jsp
- listarTratamientos.jsp
- odontograma.jsp
- registroDiarioAtencion.jsp
- registroMensualAtencion.jsp
- registroSemanalAtencion.jsp
- verCertificados.jsp
- verImagenes.jsp
- verOdontograma.jsp
- verPrescripcion.jsp
- calendar.html
- principal.htm
- principalAuxOdontologia.htm

5.4 Diseño

5.4.1 Modelo de Datos

A continuación se muestra el modelo físico para la base de datos del servicio odontológico la misma que está integrada con el Sistema de Gestión Médica, y el sistema de Gestión Socioeconómica; además se utiliza algunas tablas que forman parte del área Académica.

5.4.2 Diccionario de Datos

Dado que el Sistema de Gestión de Información Odontológica requiere información que generan otros sistemas fue preciso establecer una nomenclatura que permita diferenciar las tablas que pertenecen a los diferentes sistemas o módulos. Para ello, conjuntamente con el personal técnico del Departamento de Informática se establece un estándar, el mismo que se describe en la siguiente tabla.

Proceso	Identificativo	Nomenclatura
Servicio Médico	me	bme_tab
Servicio Odontológico	od	bod_ta
Laboratorio Cínico	la	blc_tab
Salud Integral	sa	bsa_tab
Bienestar Universitario	bie	bie_tab
Gestión socioeconómica	se	bse_tab
Recaudación	rec	rec_tab
Académico	aca	aca_tab
Uso Institucional	inst	inst_tab

Tabla 5.3 Nomenclatura de base de datos

Enseguida se presenta el diccionario de datos para el Servicio Odontológico.

TABLAS DE USO COMÚN - SALUD

BIE_TAB_PERSONAL	Registra datos del personal médico del DBU				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_personal	integer	•	Números enteros	Registra un número que se irá incrementando para identificar al personal médico	
ci_pasaporte	varchar2(13)		Caracteres solo números	Cédula o pasaporte de la persona	INS_TAB_PERSONA
Alias	varchar2(10)				
Cargo	varchar2(50)		Caracteres	Registra si la persona es Médico, Odontólogo, Enfermera, etc.	

BSA_TAB_HISTORIA_MEDICA	Registra datos médicos del paciente				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	number	•	Números enteros	Registra un número que se irá incrementando para identificar a cada paciente	
tipo_paciente	char(1)		Una letra	Identificador de tipo de paciente	

id_grupo_sanguineo	integer		Números enteros	Identificador de grupos sanguíneos	BSA_TAB_GRUPO_SANGUINEO
id_subperiodo	integer		Números enteros	Identifica el periodo en el que fue creada la historia clinica	DAC_SUBPERIODO
ci_pasaporte	varchar(13)		Caracteres solo números	Cédula o pasaporte de la persona	INS_TAB_PERSONA
fecha_creacion	date		Fecha	fecha de creacion del registro	
antecedentes_personales	varchar2(1000)		Caracteres ilimitados	Enfermedades que haya tenido el paciente	
antecedentes_familiares	varchar2(1000)		Caracteres ilimitados	enfermedades que tengan o hayan tenido los familiares del paciente	
habitos	varchar2(1000)		Caracteres ilimitados	habitos y costumbres del paciente	
alergia_medicamentos	varchar2(1000)		Caracteres ilimitados	nombre de medicamentos a los que el paciente es alérgico	
h_iess	number		Caracteres solo números	número de historia clinica del IESS	
examen_visual	varchar2(500)		Caracteres ilimitados	resultados del examen visual	
anamnesis_alimenticia	varchar2(1000)		Caracteres ilimitados	toda la anamensis alimenticia reportada	

BIE_TAB_FAMILIAR — Registra datos de un familiar del paciente con quien hablar en caso de emergencia

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_familiar	number	•	caracteres solo números	Identificador del familiar	
cod_parentesco	char(1)		Una letra	Identifica el tipo de parentesco que mantiene con el paciente	BIE_TAB_PARENTESCO
ci_familiar	varchar(13)		Caracteres solo números		
apellido	varchar(60)		caracteres solo letras	nombre de un familiar o con quien hablar en caso de emergencia	
nombre	varchar2(60)		caracteres solo letras	apellido de un familiar o con quien hablar en caso de emergencia	
direccion	varchar2(90)		caracteres números y letras	dirección del familiar o persona referente en caso de emergencia	
telefono	varchar2(9)		caracteres solo números	telefono del familiar o persona referente en caso de emergencia	
ci_pasaporte	varchar2(13)		Caracteres solo números	Cédula o pasaporte de la persona	INS_TAB_PERSONA

BSA_TAB_GRUPO_SANGUINEO — Registra información de los grupos sanguineos existentes

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_grupo_sanguineo	integer	•	Un caracter	Identificador para cada tipo de grupo sanguineo	
descripcion_grupo_sanguineo	varchar2(5)		Caracteres y simbolos	Nombre del grupo sanguíneo	

BSA_TAB_PRESCRIPCION — Registra las prescripciones del paciente

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_consulta_medica	integer	•	Números enteros	Identificador de cada consulta de un paciente	BME_TAB_CONSULTA_MEDICA
numero_historia	number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_TAB_HISTORIA_MEDICA
id_area	char(1)		Un caracter	Identificador de cada area	BSA_TAB_AREA
medicamento_cantidad	varchar2(1000)		Caracteres ilimitados	Nombre del medicamento y cantidad	
indicaciones	varchar2(1000)		Caracteres ilimitados	Indicaciones del medicamento	

BSA_TAB_IMAGEN_PACIENTE	Registra la dirección de las imágenes del paciente

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_consulta_medica	integer	•	Números enteros	Identificador de cada consulta de un paciente	BME_TAB_CONSULTA_MEDICA
numero_historia	number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_TAB_HISTORIA_MEDICA
id_area	char(1)		Un caracter	Identificador de cada area	BSA_TAB_AREA
fecha_imagen	date		Fecha	Fecha en la que sea incluida la imagen	
imagen	varchar2(100)		Caracteres	Dirección de ubicación de la imagen	

HISTORIA ODONTOLOGICA

BOD_CONSULTA_ODONTOLOGICA	Registra datos de consulta odontológica para cada paciente.

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_HISTORIA_MEDICA
id_consulta_odontologica	integer	•	Números enteros	Identificador de cada consulta de un paciente	
id_personal	integer		Números enteros	Identificador del médico con el cual se realiza la consulta	BSA_PERSONAL_MEDICO
fecha_consulta_odontologica	timestamp		Fecha y hora	Fecha de consulta odontológica	
motivo_consulta	varchar2(1000)			Motivo por el cual el paciente asiste a la consulta	
tipo_consulta	char(1)		Un caracter	Primera (P), Subsecuente (S), Emergencia (E)	
estado	char(1)		Un caracter	Estado de la consulta. Puede ser: I (Iniciada), S (Suspendida), A (Atendida)	
certificado	char(1)		Un caracter	Se Entregó Certificado. Sí (S), No(N)	

BOD_ANAMNESIS	Cuestionario odontológico de cada paciente

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_HISTORIA_MEDICA
id_pregunta	integer	•	Números enteros	Número de pregunta	BOD_PREGUNTA_ANAMNESIS
respuesta	char(1)		Un caracter	Respuesta a la pregunta formulada: S o N	
detalle	varchar2(1000)		Caracteres ilimitados	Alguna observación referente a la respuesta afirmativa	

BOD_PREGUNTA_ANAMNESIS	Listado de preguntas odontológicas

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_pregunta	integer	•	Números enteros	Número de pregunta	
pregunta	varchar2(1000)		Caracteres ilimitados	Pregunta	
alarma	char(1)		Un caracter	La pregunta es motivo de alarma Sí(S) o No(N)	

BOD_EXAMEN_BUCAL_PACIENTE	Registra problemas del examen bucal de cada paciente

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_HISTORIA_MEDICA
id_consulta_odontologica	integer	•	Números enteros	Identificador de cada consulta de un paciente	BOD_CONSULTA_ODONTOLOGICA
cod_examen_bucal	integer	•	Números enteros	Identificador de enfermedad bucal	BOD_EXAMEN_BUCAL
estado_bucal	char(1)		Un carácter	Indica el estado del examen bucal. N: Normal, A: Anormal.	
Observación	varchar2(1000)		Caracteres ilimitados	Respecto al examen bucal del paciente.	

BOD_EXAMEN_BUCAL	Listado de enfermedades bucales

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
cod_examen_bucal	integer	•	Números enteros	Identificador de enfermedad bucal	BOD_EXAMEN_BUCAL
tipo	varchar2(30)		Caracteres	Descripción de enfermedad bucal	

BOD_OCLUSION_PACIENTE	Registra problemas de oclusión para cada paciente.

Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_HISTORIA_MEDICA
id_consulta_odontologica	integer	•	Números enteros	Identificador de cada consulta de un paciente	BOD_CONSULTA_ODONTOLOGICA
cod_oclusion	integer	•	Números enteros	Identificador de oclusion	BOD_OCLUSION
estado_oclusion	char(1)		Un carácter	Indica el estado de oclusion. N: Normal, A: Anormal.	

observacion	varchar2(1000)		Caracteres ilimitados	Observaciones respecto a oclusión del paciente.	
ortodoncia	char(1)		Un carácter	Indica si el paciente requiere tratamiento de ortodoncia S(Sí) o N(No)	

BOD_OCLUSION	Registra tipos de oclusión existentes				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
cod_oclusion	Integer	•	Números enteros	Identificador de oclusion	
tipo_oclusion	varchar2(30)		Caracteres	Descripción oclusión	

BOD_ENDODONCIA_PACIENTE	Registra los problemas de endodoncia de cada paciente.				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_HISTORIA_MEDICA
id_consulta_odontologica	integer	•	Números enteros	Identificador de cada consulta de un paciente	BOD_CONSULTA_ODONTOLOGICA
cod_endodoncia	integer	•	Números enteros	Identificador de endodoncia	BOD_ENDODONCIA
estado_endodoncia	char(1)		Un caracter	Indica el estado de endodoncia. N: Normal, A: Anormal.	
observacion	varchar2(1000)		Caracteres ilimitados	Observaciones respecto a endodoncia del paciente.	

BOD_ENDODONCIA	Registra enfermedades de endodoncia				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
cod_endodoncia	int2	•	Números enteros	Identificador de endodoncia	
tipo_endodoncia	varchar2(30)		Caracteres	Descripción de endodoncia	

BOD_PERIODONTOGRAMA	Registra el estado de las piezas dentales para cada paciente.				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_HISTORIA_MEDICA
id_consulta_odontologica	integer	•	Números enteros	Identificador de cada consulta de un paciente	BOD_CONSULTA_ODONTOLOGICA
cod_pieza	integer	•	Números enteros	Código de pieza dental	BOD_PIEZA_DENTAL
cod_enf_periodontal	Integer	•	Números enteros	Código que identifica enfermedad periodontal	BOD_ENFERMEDAD_PERIODONTAL
grado				Grado de afección de enfermedad periodontal.	
observacion	varchar2(10)		Caracteres ilimitados	Observaciones respecto a periodontograma del paciente.	

BOD_ENFERMEDAD_PERIODONTAL	Registra enfermedades periodontales				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias

cod_enf_periodontal	integer	•	Números enteros	Código que identifica enfermedad periodontal	BOD_ENFERMEDAD_PERIODONTAL
tipo_enf_periodontal	varchar2(30)		Caracteres	Descripción de enfermedad periodontal	

BOD_PIEZA_DENTAL	Registra datos de la pieza dental				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
cod_pieza	integer	•	Números enteros	Código de pieza dental	
nombre_pieza	varchar2(15)		Caracteres	Nombre de pieza dental	

BOD_ODONTOGRAMA	Registra el estado de las piezas dentales para cada paciente.				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_HISTORIA_MEDICA
id_consulta_odontologica	integer	•	Números enteros	Identificador de cada consulta de un paciente	BOD_CONSULTA_ODONTOLOGICA
cod_pieza	integer	•	Números enteros	Código de pieza dental	BOD_PIEZA_DENTAL
Id_cara	char(1)		Un carácter	Identificador de cara de pieza dental	BOD_CARA_PIEZA_DENTAL
id_procedimiento	integer	•	Números enteros	Identificador del procedimiento a realizar	BOD_PROCEDIMIENTO
observacion	varchar2(10)		Caracteres ilimitados	Observaciones respecto a periodontograma del paciente.	

BOD_CARA_PIEZA_DENTAL	Contiene todas las caras de la pieza dental				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
Id_cara	char(1)	•	Un carácter	Identificador de cara de pieza dental	
nombre_cara	varchar2(15)		Caracteres	Nombre de cara de pieza dental	

BOD_PROCEDIMIENTO	Registra los procedimientos que se realizan según categoría.				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
Id_procedimiento	integer	•	Números enteros	Identificador del procedimiento a realizar	
Id_categoria	char(1)	•	Un carácter	Identificador de categoría	BOD_CATEGORIA
nombre_procedimiento	varchar2(60)		Caracteres	Descripción del procedimiento a realizar	

BOD_CATEGORIA	Registra especialidades odontológicas (Endodoncia, Periodoncia, etc)				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_categoria	char(1)	•	Un carácter	Identificador de categoría	
nombre_categoria	varchar2(20)		Caracteres	Nombre de categoría (Periodoncia, Endodoncia, etc)	

HISTORIA MEDICA

BME_TAB_CONSULTA_MEDICA	Registra datos de cada consulta del paciente				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_consulta_medica	Integer	•	Números enteros	Identificador de cada consulta de un paciente	
numero_historia	Number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_TAB_HISTORIA_MEDICA
fecha_consulta_medica	Timestamp		Fecha y hora	Fecha y hora exactas al momento de la consulta	
tipo_consulta	char(1)		Un carácter	Identifica si es consulta del IESS o no (iess I ó dbu U)	
id_personal	integer		Números enteros	Identificador del medico con el cual se realiza la consulta	BIE_TAB_PERSONAL
id_subperiodo	integer		Números enteros	Identifica el periodo en el que se realiza la consulta	dac_subperiodo
Peso	float4		Números decimales	Peso del paciente	
Pulso	integer		Números enteros	Pulso del paciente	
Temperatura	integer		Números enteros	Temperatura del paciente	
tension_arterial	varchar2(7)		Caracteres números y /	Tensión arterial del paciente	
Talla	float4		Números decimales	Talla del paciente	
motivo_consulta	varchar2(1000)		Caracteres ilimitados	Síntomas del paciente	
Tratamiento	varchar2(1000)		Caracteres ilimitados	Ingresa medicamentos y recomendaciones	

INVENTARIO: Material/Medicamento

BSA_TAB_AREA	Registra las areas del departamento de bienestar (Medico, Odontologico,Laboratorio)				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_area	char(1)	•	Un caracter	Identificador de cada area	
nombre_area	varchar2(20)		Caracteres	Nombre del área (Médico)	

BSA_TAB_ARTICULO	Registra el material/medicamento con el que cuenta el DBU				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
cod_articulo	varchar2(5)	•	Caracteres	Código que identifica a cada material	
id_area	char(1)	•	Un carácter	Identificador de cada area	BSA_TAB_AREA
nombre_articulo	varchar2(60)		Caracteres	Nombre del material	
Presentación	varchar2(50)		Caracteres	Presentacion del material (frascos, tubos, sobres)	
Max	integer				

Min	integer		Números enteros	Cantidad de material existente	

BSA_TAB_KARDEX	Registra los ingresos/egresos de material o medicamentos al DBU				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_doc	integer	•	Números enteros	Indentificador de cada ingreso registrado	
id_area	char(1)	•	Un caracter	Identificador de cada area	BSA_TAB_AREA
cod_articulo	varchar2(5)		Caracteres	Código que identifica a cada material/medicamento	BSA_TAB_ARTICULO
Fecha	timestamp	•	Fecha y hora	Fecha y hora exactas al momento del ingreso	
Transacción	char(1)		Un caracter	Ingreso/Egreso de medicamento/material	
Concepto	varchar2(30)		Caracteres	Cantidad de material que ingresa	
Cant	integer		Números enteros	Origen de los medicamentos (almacen)	
Saldo	integer		Números enteros		
Responsable	integer		Números enteros	Identificador del personal medico que registra el ingreso/egreso	BIE_TAB_PERSONAL

TURNOS

BSA_TAB_TURNO	Registra los turnos (1,2,3....)de los pacientes por fecha y por médico				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
fecha_turno	date	•	Un caracter	Fecha cada	
numero_historia	number	•	Caracteres	Número de historia que se le ha asignado a cada paciente	BSA_TAB_HISTORIA_MEDICA
id_personal	integer	•	Números enteros	Identificador del médico	BIE_TAB_PERSONAL
num_turno	integer	•	Números enteros	Número del turno del paciente (1,2,3,....)	
id_practica	integer	•	Números enteros	Identificador de la practica a realizar	BSA_TAB_PRACTICA
id_estado	integer	•	Números enteros	Indentificador del estado	BSA_TAB_ESTADO

BSA_TAB_PRACTICA	Todas las practicas posibles a realizar				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_practica	integer	•	Números enteros	Identificador de la practica a realizar	
descripcion_practica	varchar2(30)	•	Caracteres	Descripción de la práctica (1a consulta, seguimiento)	

BSA_TAB_ESTADO	Estados de la consulta (Atendido, Esperando, etc)				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias

id_estado	integer	•	Números enteros	Identificador del estado de la consulta	
descripcion_estado	varchar2(20)	•	Caracteres	Descripción del estado (esperando. Atendido, etc)	

BSA_TAB_HORARIO_MEDICO	Horario de atención de cada médico durante la semana				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_personal	integer	•	Números enteros	Identificador del médico	BIE_TAB_PERSONAL
id_dia	char(1)	•	Un carácter	Identificador de los dias de la semana	BSA_TAB_DIA
jornada1inicio	time		Hora (hh:mm:ss)	Hora de inicio de su primera jornada	
jornada1fin	time		Hora (hh:mm:ss)	Hora de finalización de su primera jornada	
jornada2inicio	time		Hora (hh:mm:ss)	Hora de inicio de su segunda jornada	
jornada2fin	time		Hora (hh:mm:ss)	Hora de finalización de su segunda jornada	
id_duracion_consulta	integer		Números enteros	Identificador de duración de consulta	BSA_TAB_DURACION_CONSULTA
total_consultas	integer		Números enteros	Número	

BSA_TAB_DURACION_CONSULTA	Duración de la consulta. (5min, 10 min.)				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_duracion_consulta	integer	•	Números enteros	Identificador de duración de consulta	
duracion_consulta	integer		Números enteros	Duración de las consultas (5min,10min,...)	

BSA_TAB_DIA	Días de la semana				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_dia	char(1)	•	Un caracter	Identificador de los días de la semana	
nombre_dia	varchar2(10)		Caracteres	Nombre del día de la semana	

BSA_TAB_DIA_EXCLUIDO	Fecha que no se asignan turnos.				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_personal	integer	•	Números enteros	Identificador del médico	BIE_TAB_PERSONAL
fecha_excluida	date	•	Fecha "2006-10-09"	Feriados y días excluidos	
Habilitado	char(1)		Un carácter	Indica el estado del día. H: Habilitado, D: Deshabilitado.	

BSA_TAB_DIA_ASIGNADO					
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
fecha_asignada	date	•	Una fecha (Año, mes, dia)		

id_personal	integer	•	Números enteros	Identificador del médico	BIE_TAB_PERSONAL
jornada1inicio	time		Hora (hh:mm:ss)	Hora de inicio de su primera jornada	
jornada1fin	time		Hora (hh:mm:ss)	Hora de finalización de su primera jornada	
jornada2inicio	time		Hora (hh:mm:ss)	Hora de inicio de su segunda jornada	
jornada2fin	time		Hora (hh:mm:ss)	Hora de finalización de su segunda jornada	
id_duracion_consulta	integer		Números enteros	Identificador de duración de consulta	BSA_TAB_DURACION_CONSULTA
total_consultas	integer		Números enteros	Número	

BSA_TAB_SOBRETURNO	Registra los sobreturnos de los pacientes por fecha y por médico				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	number	•	Caracteres	Número de historia que se le ha asignado a cada paciente	BSA_TAB_HISTORIA_MEDICA
fecha_sobreturno	date	•	Una fecha	Fecha en la que sucede el sobreturno.	
hora_turno	time	•	Hora (hh:mm:ss)	Hora que se produce el sobreturno	
id_practica	integer		Números enteros	Identificador de la practica a realizar	BSA_TAB_PRACTICA
id_estado	integer		Números enteros	Indentificador del estado	BSA_TAB_ESTADO
id_personal	integer		Números enteros	Identificador del médico	BIE_TAB_PERSONAL

CERTIFICADOS

BSA_TAB_CERTIFICADO_PACIENTE	Registra certificados entregados al paciente				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
numero_historia	Number	•	Números enteros	Número de historia que se le ha asignado a cada paciente	BSA_TAB_HISTORIA_MEDICA
fecha_emite_certificado	timestamp	•	Fecha y hora	Fecha de entrega de certificado	
id_certificado	integer		Números enteros	Identificador de certificado	BSA_TAB_CERTIFICADO
observaciones	Varchar2(1000)		Caracteres ilimitados	Observaciones respecto a	
tipo_certificado	Number				
id_area	char(1)		Un caracter	Identificador de cada area	BSA_TAB_AREA

BSA_TAB_CERTIFICADO	Registra tipos de certificados existentes				
Columna	Tipo de dato	Clave primaria	Datos a almacenar	Descripción	Referencias
id_certificado	Number	•	Números enteros	Identificador de certificado	
descripción_certificado	Varchar2(1000)		Caracteres ilimitados	Información respecto al certificado	
id_area	char(1)		Un caracter	Identificador de cada area	BSA_TAB_AREA

5.5 Implementación

5.5.1 Instalación del Sistema

Al finalizar el desarrollo, la aplicación se publica en el servidor Tomcat para lo cual del directorio dist de la aplicación se transfiere el archivo DentalUTN.war al directorio webapps del servidor.

5.5.2 Funcionalidad del Sistema

El sistema consiste de una aplicación Web para uso cotidiano, accesible a través de la intranet de la UTN para los usuarios autorizados. Se integra de manera transparente con el Sistema de Gestión de Información Médica y con el Sistema de Gestión Socioeconómica.

Se trata de DentalUTN, un sistema de Gestión de Información Odontológica exclusivo para el Departamento de Bienestar Universitario de la UTN, enfocado especialmente en agilitar el proceso de atención odontológica, generación de reportes y estadísticas de manera oportuna para el usuario.

Una guía detallada del manejo y funcionalidad de la aplicación se encuentra en el manual de usuario.

A continuación se describe las opciones del sistema de acuerdo al usuario.

Usuario Odontólogo/a

Contiene las siguientes opciones:

- Autentificación: Mediante usuario y contraseña única.
- Configuraciones: Corresponde a la definición de parámetros generales como: lista de preguntas para anamnesis, patologías y tratamientos.

- ♣ Búsqueda de Pacientes: Permite buscar la Ficha Odontológica de un paciente mediante tres criterios: número de historia, cédula, nombres y apellidos.

- ♣ Turnos: Muestra la lista de pacientes que serán atendidos por un profesional de odontología en la fecha actual; con opción a cancelar el turno de ser necesario.

- ♣ Consulta odontológica: Se refiere al registro de:
  - Motivo de consulta.
  - Examen Bucal.
  - Anamnesis.
  - Oclusión.
  - Odontograma: El cual contiene Patologías y Tratamientos.
  - Notas de Evolución.
  - Prescripción.
  - Certificados.

- ♣ Imágenes: Permite añadir y ver imágenes del paciente.

- ♣ Reportes: Es la lista de pacientes que fueron atendidos por el profesional de odontología durante el día (Registro diario), entre un rango de fechas (Registro semanal), en un mes (Concentrado mensual), patologías frecuentes, etc.

Se destaca en las siguientes figuras la administración del odontograma.

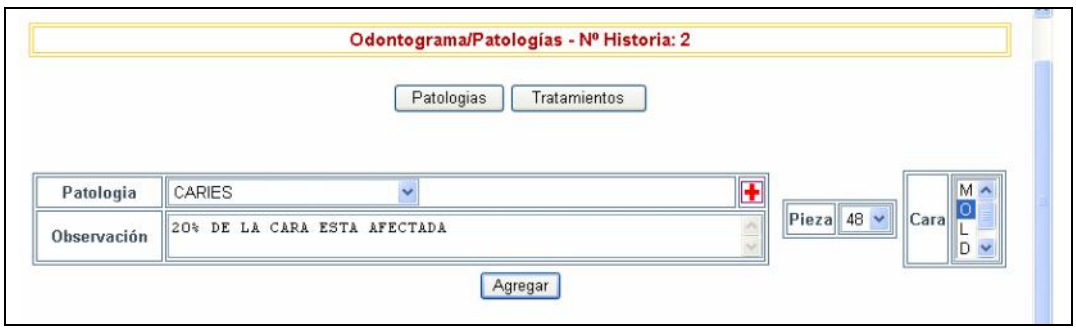


Figura 5.8 Registro de Patología DentalUTN.

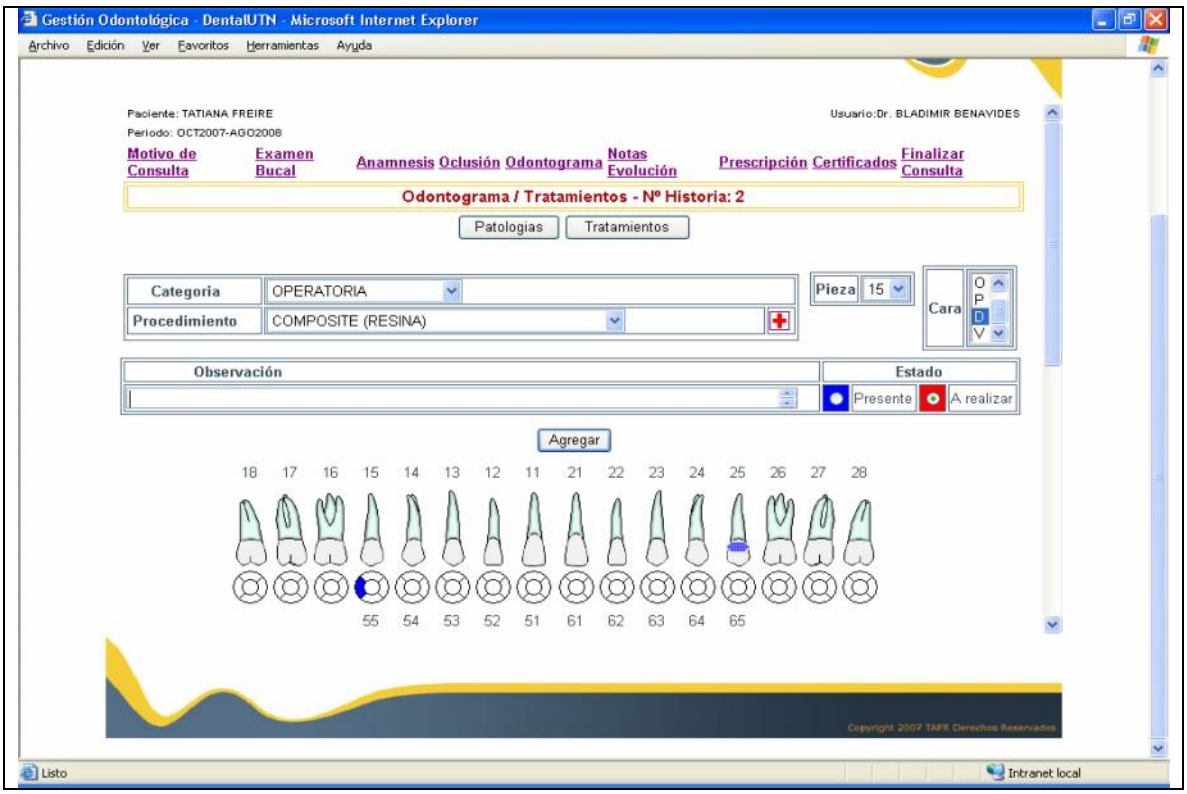


Figura 5.9 Registro de Tratamiento DentalUTN.

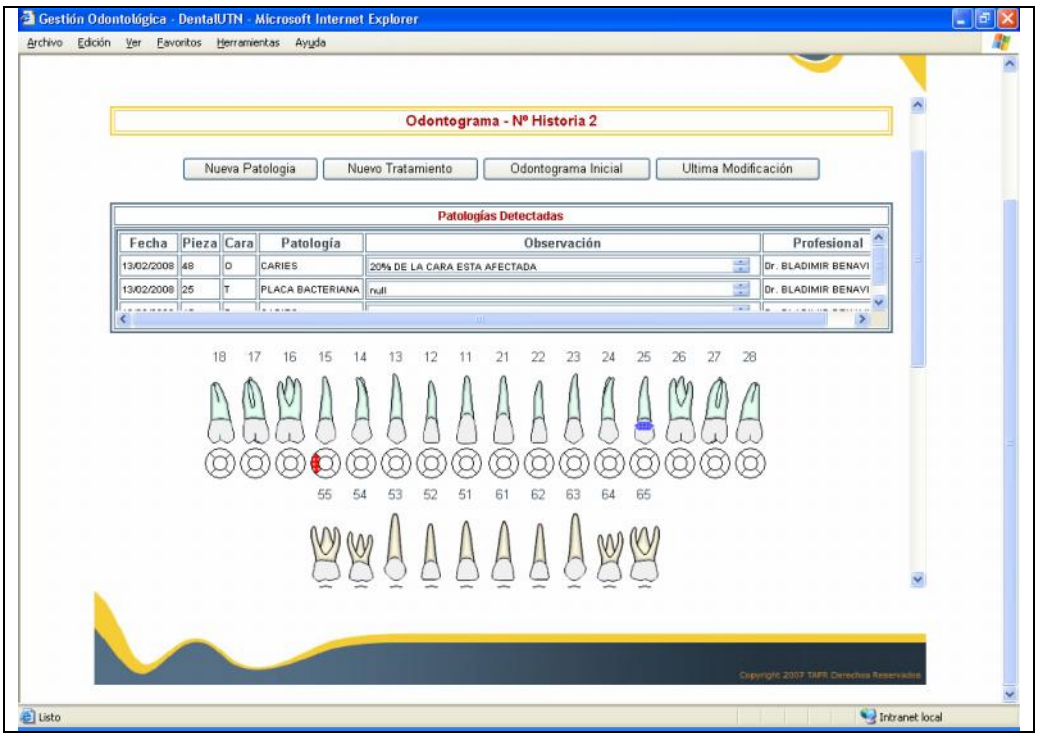


Figura 5.10 Odontograma General

Usuario Auxiliar de Odontología

Para este usuario se describen las siguientes opciones:

♣ Autentificación

Mediante usuario y contraseña única.

♣ Configuraciones

Corresponde a la definición de parámetros generales como: lista de preguntas para anamnesis, patologías y tratamientos.

♣ Búsqueda de Pacientes

Permite buscar la Ficha Odontológica de un paciente mediante tres criterios: número de historia, cédula, nombres y apellidos.

♣ Turnos:

Permite asignar al paciente un turno en la fecha actual conforme al orden de llegada y con el profesional seleccionado.

Los pacientes con turno se encuentran en lista de pacientes en espera y tienen opción a cancelar el turno de ser necesario.

♣ Inventario:

Se trata de llevar un control del ingreso y egreso de materiales de odontología.

♣ Reportes

Permite generar reportes como: Parte diario, concentrado mensual o entre fechas de cada odontólogo/a, materiales existentes, etc.

♣ Imágenes

Permite añadir y ver imágenes de RX del paciente.

5.6 Pruebas

Una vez finalizado el desarrollo de la aplicación se inicia un periodo de pruebas durante el cual se contempló los siguientes aspectos:

- Instalación de puntos de red en el área médica y odontológica, entrega de computadores, impresoras y escáner adquiridos, e instalación y configuración del software base.

- Verificación de la integración de sistemas: socioeconómico, médico y odontológico; desarrollados paralelamente.

- Modificación leve a la interfaz de usuario.

- Y definición de parámetros: preguntas de anamnesis, patologías, tratamientos y simbología del odontograma, aspectos fundamentales para el correcto funcionamiento del sistema.

5.7 Capacitación a Usuarios

En coordinación con la directora del Departamento de Bienestar Universitario y los desarrolladores de los sistemas: médico y socioeconómico, se estableció un cronograma de capacitación. El mismo que inicialmente se lo definió para una semana, no obstante se prolongó durante un mes debido a la falta de conocimientos básicos de computación por parte de los usuarios. Para ello se procedió a la entrega del manual de usuario respectivo y mediante sesiones de trabajo se enseñó el correcto uso del sistema.

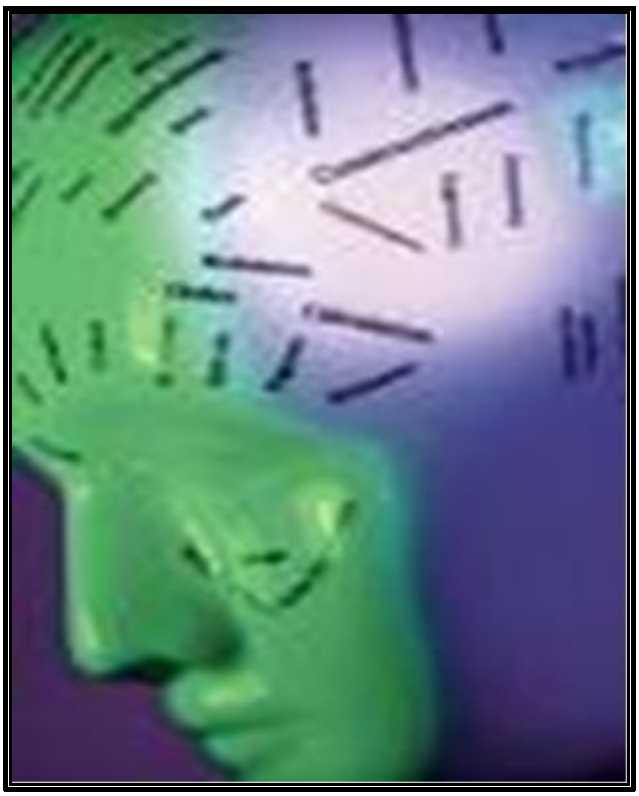
Cabe resaltar la colaboración de todos los usuarios durante esta etapa y la predisposición de los desarrolladores en brindar el soporte respectivo a los sistemas.

5.8 Puesta en marcha

El sistema de Gestión de Información Odontológica se encuentra funcionando normalmente desde el 4 de marzo del 2008.

CAPITULO VI

Conclusiones y Recomendaciones



6.1 Verificación de Hipótesis.
6.2 Conclusiones.
6.3 Recomendaciones.

"Lo que sabemos es una gota de agua; lo que ignoramos es el océano."

Isaac Newton.

6.1 Verificación de Hipótesis

La hipótesis planteada al inicio de esta Tesis:

ORM permite desarrollar aplicaciones de manera sencilla, de fácil mantenimiento y con un nivel de independencia respecto del modelo de datos.

Al término de este trabajo de investigación y el Software desarrollado se demuestra que la hipótesis ha sido VERIFICADA TOTALMENTE dado que:

Se desarrolló el Sistema de Gestión de Información Odontológica para el Departamento de Bienestar Universitario de la UTN utilizando ORM, específicamente con el framework JPA (Java Persistence API) y actualmente el sistema se encuentra operando con normalidad.

6.2 Conclusiones

- La forma más usual para persistir un modelo de objetos es usando una base de datos relacional.

- La técnica de persistencia no sólo puede determinar el rendimiento de la aplicación, sino que también influirá enormemente en la cantidad de esfuerzo requerido para desarrollar y mantener la aplicación; y a menos que se tome las decisiones de diseño correctas desde el principio, podría ser difícil revisar esta parte del diseño después de haber terminado la aplicación.

- ORM (Mapeo Objeto Relacional) es una buena técnica para la persistencia de objetos en bases de datos relacionales. Dado que encapsula la complejidad de la traducción (conexiones a bases de datos, catálogos, esquemas, tipos de datos, diferentes fuentes de información) a través de archivos xml u anotaciones. Es reutilizable. Soporta la generación automática de código SQL.

- Actualmente ORM, es la mejor opción para implementar una aplicación de software: por una parte, se hace uso del paradigma objetual, aprovechando las ventajas de flexibilidad, mantenimiento y reutilización. Por otra parte, se dispone una base de datos relacional, aprovechando su madurez y su estandarización así como de las herramientas relacionales que hay para ella.

- Existen muchas herramientas en el mercado, tanto comerciales como opensource. La elección de un framework de persistencia dependerá de factores como las características requeridas y los costos de adquisición y mantenimiento, además de requerimientos no-funcionales como el rendimiento.

- El servicio Odontológico del Departamento de Bienestar Universitario cuenta con una herramienta que garantiza el correcto tratamiento de la información odontológica.

6.3 Recomendaciones

- Los sistemas debemos construirlos en función de patrones y estándares, y reutilizando al máximo. Esto nos permite centrarnos en la lógica de negocio, reducir los tiempos de desarrollo, y aumentar la calidad del resultado.

- Es fundamental conocer bien como funcionan las tecnologías que utilizamos. En el caso de ORM hemos visto que dependiendo de como hagamos las cosas puede afectar directamente al rendimiento de la aplicación.

- Es necesario que la Universidad Técnica del Norte defina políticas que regulen el desarrollo de sistemas de manera que se aproveche la tecnología e infraestructura adquirida.

- En el servicio de odontología se manejan equipos odontológicos que han cumplido su vida útil; por lo que se recomienda la adquisición de nuevos equipos con características avanzadas que permitan recuperar datos del estado bucal del paciente e integrarla al sistema de Gestión de Información Odontológica.

Glosario

Aplicación web: Aplicación informática que los usuarios utilizan accediendo a un servidor web a través de Internet o de una intranet.

API: Conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. Representa una interfaz de comunicación entre componentes software.

Anotación Java: Forma de añadir metadatos al código fuente Java que están disponibles para la aplicación en tiempo de ejecución.

Arquitectura: Representación abstracta de los componentes de un sistema y su comportamiento

Byte: Unidad de información formada por ocho bits.

Clase: Definición de un objeto.

DAO: componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo.

EntityManager: Interfaz que define los métodos que son usados para interactuar con el el contexto de persistencia.

Facade: Patrón de Diseño, sirve para proveer de una interfaz unificada sencilla que haga de intermediaria entre un cliente y una interfaz o grupo de interfaces más complejas.

Framework de Persistencia: Componente de software encargado de traducir entre objetos y registros (de la base de datos relacional). Es decir es el encargado de que el programa y la base de datos se “entiendan”.

Hibernate: Herramienta de Mapeo objeto-relacional para la plataforma Java (y disponible también para .Net con el nombre de NHibernate) que facilita el mapeo de atributos entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) que permiten establecer estas relaciones.

Java: Lenguaje de Programación orientado a objetos desarrollado por Sun Microsystems a principios de los años 90.

JDBC: Java DataBase Connectivity es el API de Java que define cómo una aplicación cliente accederá a una base de datos, independientemente del motor de base de datos al que accedamos.

JPA: Framework de Persistencia.

JCP (Java Community Process): Organismo que crea y mantiene las especificaciones para Java.

JSR (Java Specification Request): Son documentos formales que describen las especificaciones y tecnologías propuestas para que sean añadidas a la plataforma Java.

Mapping: Proceso de conectar objetos/atributos a tablas/columnas.

Metadatos: Descripciones estructuradas y opcionales que están disponibles de forma pública para ayudar a localizar objetos.

MVC: Patrón de arquitectura de software que separa los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

Persistencia: Capacidad de almacenar y recuperar el estado de los objetos, de forma que sobrevian a los procesos que los manipulan.

POJO(Plain Old Java Object): Simple clase Java que tiene métodos get y set para cada uno de los atributos.

Patrón de diseño: Un patrón de diseño es una solución a un problema de diseño no trivial que es efectiva (ya se resolvió el problema satisfactoriamente en ocasiones anteriores) y reusable (se puede aplicar a diferentes problemas de diseño en distintas circunstancias).

Objeto: Entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos). Se corresponde con los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa). Es una instancia a una clase.

Objeto persistente: Objeto que sobrevive a la ejecución del proceso que lo creó.

Objeto transiente: Objeto que deja de existir cuando el proceso que lo creó termina su ejecución.

ORM (Object/Relational Mapping): Técnica que realiza la transición de una representación de los datos de un modelo relacional a un modelo orientado a objetos y viceversa.

RESOURCE_LOCAL: Tipo de transacción para aplicación con soporte JPA. Indica que el proveedor de persistencia se hará cargo de la gestión de transacciones.

Serialización: Secuencia de bytes escrita en un fichero en disco.

SQL: Es un lenguaje declarativo de acceso a bases de datos relacionales que permite especificar diversos tipos de operaciones sobre las mismas.

ToplinkEssentials: Implementación de JPA.

Bibliografía

Libros

- [LIB01]

Hibernate in Action. Practical Object/Relational Mapping. Bauer, Christian; Gavin King. Manning Publications © 2004.

- [LIB02]

Java Persistence for Relational Databases. Richard Sperko Apress © 2003.

- [LIB03]

Mapping objects to relational databases: O/R mapping in detail. Scott W. Ambler, 2006. URL http://agiledata.org/essays/mappingObjects.html.

- [LIB04]

Java Persistente with Hibernate. Bauer, Christian; Gavin King. Manning Publications © 2007.

- [LIB05]

Pro EJB 3: Java Persistence API. Mike Keith and Merrick Schincariol. Apress © 2006.

- [LIB06]

Beginning Hibernate: From Novice to Professional. Dave Minter, Jeff Linwood. Apress © 2006

Internet

- [WWW01]

http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=PersistenciaJava

- [WWW02]

http://es.wikipedia.org/wiki/Persistencia_de_objetos

- [WWW03]

http://msdn2.microsoft.com/es-es/library/ms233836(VS.80).aspx

- [WWW04]
http://www.db4o.com/espanol/db4o%20Whitepaper%20-%20Bases%20de%20Objetos.pdf

- [WWW05]
http://k7k0.apihuella.com/tesis/mapeo_objeto_relacional.htm

- [WWW06]
http://msevents.microsoft.com/CUI/WebCastEventDetails.aspx?EventID=1032309114&EventCategory=5&culture=es-MX&CountryCode=MX

- [WWW07]
http://www.iit.upcomillas.es/pfc/resumenes/450955e7368ca.pdf

- [WWW08]
http://www.softwareguru.com.mx/downloads/SG-200502.pdf

- [WWW09]
http://pizarropablo.googlepages.com/ORM-ObjectRelationalMapping-PizarroP.pdf

- [WWW10]
http://sophia.javeriana.edu.co/~javila/pregrado/arquitectura/JPA.pdf

- [WWW11]
http://blog.marcomendes.com/category/tecnologias-java/j2ee/

- [WWW12]
http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=AnotacionesEJB3

- [WWW13]
http://java.sun.com/developer/technicalArticles/J2SE/Desktop/persistenceapi/?feed=JSC

- [WWW14]
http://www.fing.edu.uy/inco/cursos/tsi/TSI2/teorico/Clase5a-JPA.pdf

- [WWW16]

http://www.intertech-inc.com/resource/usergroup/Exploring%20the%20JPA.pdf

- [WWW17]

http://es.wikipedia.org/wiki/Modelo_Vista_Controlador

- [WWW18]

http://today.java.net/pub/a/today/2007/12/18/adopting-java-persistence-framework.html

- [WWW19]

http://download.oracle.com/docs/cd/B32110_01/web.1013/b28218/undtl.htm

Documentos de trabajo

- [DOC01]

JSR 220: Enterprise JavaBeans Version 3.0, Java Persistence API, EJB 3.0 Expert Group, Sun Microsystems. ejb-3_0-fr-spec-persistence.pdf http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html

- [DOC02]

Enríquez Evelin y Freire Tatiana. “Solución De Un Sistema De Gestión De Salud”. UTN/DBU. Propuesta. Ibarra, Agosto 2006.

Anexos

Digitales

1. Código fuente de la Aplicación

   Ubicado en el directorio: \Aplicativo\DentalUTN

2. Manuales

   Ubicado en el directorio: \Aplicativo\ Manuales\

   - Manual de Usuario: Odontólogo/a y Auxiliar de Odontología.
   - Manual Técnico.