UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Título: Problemas de performance en el uso de mapeadores objeto/relacional en aplicaciones con grandes volúmenes de datos
Autores: Belena Raquel Zulaica
Director: Dra. Silvia Gordillo
Carrera: Licenciatura en Informática

Resumen

La performance es uno de los requerimientos no funcionales principales de las aplicaciones enterprise. Este requerimiento puede convertirse en una característica que, de no cumplirse, afecte la usabilidad y la funcionalidad de la aplicación pudiendo causar una mala aceptación del usuario y hasta el fracaso del proyecto.
En la actualidad, la programación orientada a objetos se ha convertido en el paradigma de programación dominante para el desarrollo de software de gran escala y las bases de datos relacionales son el repositorio más popular y con suficiente madurez para almacenar los objetos de dominio. Ambos paradigmas se adaptan muy bien a las necesidades de las aplicaciones enterprise, pero existe una inherente incompatibilidad conceptual entre ellos, conocido como "desajuste por impedancia". El uso de patrones de diseño, frameworks de persistencia y mapeadores objeto/relacional ayudan a resolver el desajuste por impedancia. Pero esto introduce una gama de problemas nuevos y nada triviales de resolver.
La performance es uno de los problemas más frecuentes y críticos en aplicaciones enterprise, debido al volumen y complejidad de los datos que manejan y los tiempos de respuesta que se deben mantener. Es importante considerar aspectos de performance en etapas tempranas de definición y diseño. De todas maneras, aún así los problemas de performance pueden producirse y el equipo debe estar preparado para poder afrontarlos.

Palabras Claves

Aplicaciones Enterprise, problemas de performance, mapeadores objeto/relacional, bases de datos relacionales, modelos orientados a objetos, desajuste de impedancia, volumen de datos, lógica de negocio compleja.

Conclusiones

Adoptando las técnicas correctas de optimización, la utilización de mapeadores objeto/relacional presenta grandes beneficios para las aplicaciones que manejan mucho volumen de datos y lógica de negocio compleja.
Se debe tener en cuenta aspectos relacionados con la performance de la aplicación durante etapas tempranas de diseño y en la definición de la arquitectura, dejando de lado los aspectos puros de diseño de objetos y relacional para rescatar los aspectos positivos de cada uno y combinarlos para lograr los mejores resultados.

Trabajos Realizados

Estudio de las características y arquitecturas de aplicaciones enterprise.
Introducción y análisis al problema de desajuste por impedancia.
Análisis de alternativas de mapeos, frameworks y herramientas existentes en la comunidad para resolver el desajuste por impedancia.
Análisis de problemas de performance en aplicaciones que presentan reglas de negocio complejas y manipulan gran cantidad de datos.
Análisis de posibles soluciones a los problemas planteados anteriormente.

Trabajos Futuros

Análisis de performance de los principales frameworks.
Comparación en el uso mapeadores y bases de datos orientadas a objetos.
Análisis de nuevos problemas de performance y posibles soluciones.
Implementación de las estrategias de optimización.
Análisis del uso de bases de datos noSQL en contextos de aplicaciones con modelos orientados a objetos.
Incorporación de procesos de performance en el ciclo de vida de un proyecto.

Fecha de la presentación: Noviembre 2011

Agradecimientos

Quiero agradecer a mi directora de tesis, Silvia Gordillo por su experiencia y orientación para poder llevar a cabo este trabajo.

Al LIFIA, al cual pertenezco desde hace varios años, por darme la posibilidad de dar mis primeros pasos en la industria y de poder perfeccionarme constantemente tanto en el ámbito profesional como personal.

Al proyecto e-Sidif por los conocimientos y la experiencia que me brindó para poder desarrollar esta tesis. A mis compañeros de trabajo por las charlas técnicas y sus aportes.

A mi familia, mis padres Alberto y Raquel, mis hermanos Melina y Lisandro, por el apoyo incondicional durante toda la carrera, por el aliento y la paciencia que me tuvieron en todo momento.

A mis amigos de toda la vida: Caro, Lucre, Sole, Andrea, Ale, Estela, Kaño, Martín y Hugo. A los amigos cosechados a lo largo de esta carrera, en particular a Flor, Eva, Baby, Lisa, Magui, Ine, Ale y Pau.

Finalmente y en especial a mi amor Santiago, por ser mi sostén en todo momento, por su paciencia infinita y por estar siempre.

Índice

Capítulo I – Introducción

Motivación

Los sistemas informáticos están cada vez más integrados a las necesidades de las empresas modernas, 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. [1]
Uno de los retos más importantes en aplicaciones empresariales es la performance, siendo, en muchos casos, un factor determinante del éxito o fracaso de la aplicación.
Los aspectos relacionados con el rendimiento de la aplicación se deben considerar en cada fase del ciclo de vida de un proyecto: especificación de requerimientos, diseño, implementación, mantenimiento y especialmente en las etapas tempranas de definición de la arquitectura del proyecto.
Tener en cuenta la performance desde el inicio del proceso de software, permite anticiparse a problemas, que de no ser así, recién van a detectarse en la etapa de testing de la aplicación y en el peor caso, cuando el sistema ya se encuentre operativo.[2]

Aplicaciones Enterprise

Las aplicaciones enterprise son sistemas de software que presenta algunas características que las distinguen del resto de las aplicaciones de software:
Trabajan con un conjunto de reglas de negocio complejas. Las reglas de negocio describen las operaciones, definiciones y restricciones que se aplican a una organización para alcanzar sus objetivos. Es necesario contar con herramientas y conceptos que permitan diseñar el sistema de manera clara y lo suficientemente detallado para capturar la complejidad del negocio. El paradigma orientado a objetos posee la extensibilidad y expresividad necesarias para permitir que el proceso de diseño y desarrollo sea flexible y eficiente para adaptarse a los constantes cambios del negocio.[3]
Los datos son persistentes, porque son necesarios en futuros accesos al sistema para responder a la lógica de negocio implementada. Los datos se guardan durante mucho

tiempo, inclusive años; ya que representan parte del patrimonio de la organización que los genera y manipula. Durante este período de tiempo, es probable que en la organización cambien las aplicaciones que los utilizan, el hardware que los creó originalmente, los sistemas operativos y servidores, lo cual no debe alterar los datos ya persistidos.
Manipulan una gran cantidad de datos. Los datos generalmente son complejos y con muchas relaciones entre sí. Los datos que se van generando a través de las distintas operaciones que brindan los sistemas deben ser almacenados La persistencia de los datos asegura que los mismos queden disponibles para futuras operaciones. Los datos deben soportar cambios de hardware y migraciones a nuevas estructuras de persistencia.
Considerando que trabajamos con entornos orientados a objetos y que en la actualidad la industria sigue utilizando base de datos relacionales, es imprescindible trabajar con mapeadores objeto/relacional que permitan "traducir" las clases en tablas (y viceversa) y los atributos en campos de las tablas (y viceversa).
Las aplicaciones enterprise, generalmente, presentan acceso concurrente a los datos. La aplicación debe permitir que gran cantidad de personas puedan acceder al sistema adecuadamente. También debe permitir el acceso de más de una persona al mismo tiempo, controlando las operaciones que intentan realizar, para evitar inconsistencias y errores en los datos y en la aplicación.
Presentan una gran cantidad de interfaces gráficas al usuario para manejar los datos. En aplicaciones enterprise existen distintos tipos de usuarios. Los usuarios habituales que conocen exhaustivamente el funcionamiento del sistema, los usuarios regulares que poseen poca experticia, los usuarios administradores y auditores. Cada uno necesita ver cierta información y de una manera particular para llevar a cabo su gestión.
Las aplicaciones enterprise, comúnmente, necesitan integrarse con otras aplicaciones. Estas aplicaciones puede sen de la misma organización, de otras organizaciones que proveen servicios y con aplicaciones que tienen como fin vender servicios a aplicaciones enterprise entre otro tipo de integración. Hoy en día, el uso de Web Services facilita la integración de aplicaciones gracias a la estandarización de protocolos y formatos utilizados para la comunicación e interpretación de los pedidos y respuestas.

Arquitecturas en aplicaciones enterprise

La arquitectura de software es el soporte necesario para la construcción de los casos de uso, que garantice el cumplimiento de los atributos de calidad, asegure la calidad y maximice la productividad.
En función a las características de la aplicación se elige inicialmente el estilo a utilizar. Si bien existen varios estilos arquitectónicos, la arquitectura con estilo N-Layered es la que prevalece en las aplicaciones enterprise.
Los elementos de este tipo de arquitectura son layers, generalmente traducidas como capas. Cada layer tiene una responsabilidad bien definida que, para llevarla a cabo, requiere comunicación con las capas subsiguientes.
Si bien en la teoría, un layer no debería poder interactuar directamente con otro layer si no está inmediatamente a su lado (arquitecturas layered opacas), en la práctica no siempre ocurre por cuestiones de performance o simplicidad. El hecho de que no sea un modelo puro de layers atentará contra la mantenibilidad. Por este motivo es necesario hacer un análisis de costo-beneficio y definir pautas claras de cuando es posible sobrepasar capas. [3]
Las principales ventajas de este estilo son:

- Desacoplamiento: Se puede desarrollar un layer sin disponer aún de los layers con los que debe colaborar. Alcanza tan solo con conocer el "contrato" de dichos layers. El contrato será el conjunto de servicios que ofrece un layer. Este

desacoplamiento favorece la división de trabajo en el equipo de desarrollo y el testing aislado.
- Es posible cambiar implementaciones de layers (con el mismo contrato) sin modificar los restantes.
- Si se respeta el desacoplamiento y se mantiene el modelo lo más opaco posible, un cambio en el contrato de dos layers no debería afectar a los demás. Por ejemplo, supongamos que se tienen los layers A, B y C. Si cambia el contrato entre B y C no se debería tener que modificar A (asumiendo una arquitectura opaca donde A no consume servicios de C).
- Reuso: Un mismo layer puede ser utilizado o consumido por layers para distintos propósitos.

La performance, es un requerimiento que si no se tiene en cuenta en las decisiones arquitectónicas, puede ser un problema crítico en este tipo de arquitecturas. El estilo N-layered estimula la creación de múltiples layers para obtener todos los beneficios que trae consigo el desacoplamiento de las distintas partes del sistema. Hay que evitar el sobre-diseño de layers ya que puede traer más problemas que soluciones. En ciertos casos es inevitable que la mantenibilidad "ceda un poco de pista" para dar paso a una mejora de la performance.

A continuación se detalla la evolución de este estilo arquitectónico. Comenzaremos desde el momento en que no tenían sentido pensar en layers para llegar a arquitectura n-layered, pasando por arquitectura conocidas como la de 2 y 3 layers.
Durante el desarrollo de sistemas batch, sistemas que solo interactuaban con archivos y no tenían interacción hombre-máquina, no existía la necesidad de layers.

2 layers

A partir de los años 90, comenzó a surgir la idea de layers con los sistemas client-server. Estos sistemas se consideraban "sistemas de dos layers": el layer de cliente y el layer de servidor como muestra la Figura 1.

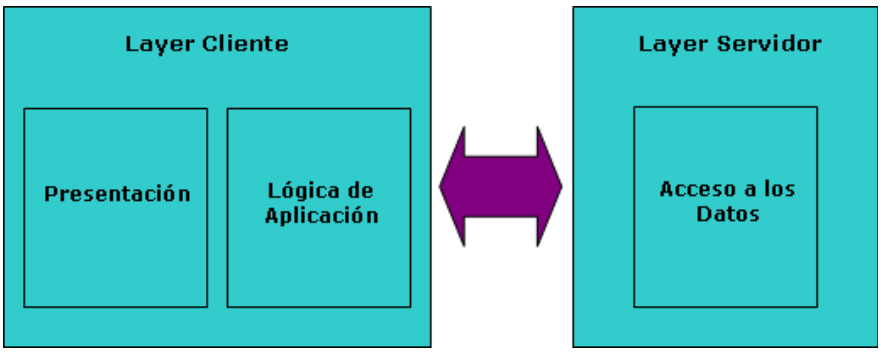


Figura 1: Arquitectura de 2 layers

En los primeros sistemas dos layers, los clientes no solo implementaban la GUI sino también la lógica de la aplicación. El servidor, generalmente, se encargaba solo de la persistencia de los datos. Por lo general era una base de datos. Los datos, podían ser consultados y modificados desde el cliente utilizando algún lenguaje de consultas y de ABM (SQL por ejemplo). [3]
Para sistemas que solo tenían que implementar operaciones de consultas, altas, bajas y modificaciones, este esquema funcionaba bien y era muy productivo. Muchos de los

ambientes de desarrollo brindaban herramientas que facilitaban la construcción de los clientes en este tipo de arquitecturas. Por ejemplo, proveían la posibilidad de realizar fácilmente bindings entre los widgets de la GUI y campos de la tabla. Este binding, luego de configurado, sincronizaba el dato que mostraba el widget con el valor que tenía el campo en la base de datos. Algunos ejemplos: Delphi y Visual Basic.
El problema surgió cuando se quiso tener la lógica de dominio implementada en el cliente y peor aun, cuando esta lógica se tornó más complicada. Una herramienta que fue diseñada para desarrollar GUI e interactuar con datos era ahora utilizada para escribir reglas, validaciones y cálculos complejos. Como consecuencia, la lógica de las GUIs comenzó a ser cada vez más compleja, con código replicado, difícil de mantener y de testear. Además, cuando surgió la idea de poder presentar la misma aplicación en distintos clientes (por ejemplo, cliente de escritorio y cliente web), la solución era reescribir toda la lógica en ambos cliente, solución que implicaba replicar código y complicaba el mantenimiento.
Para evitar algunos de los mencionados problemas, se buscaron alternativas en las cuales la lógica se escriba del lado del servidor y los clientes solo tengan que consumirla para poder dibujar las GUIs.
La primera idea que surgió, considerando que en ese momento el servidor era en general una base de datos, fueron los stored procedures. Este esquema incitaba más a la modularización que el anterior, pues estaba pensado para escribir la lógica y no para dibujar GUIs. También, resolvía el problema de múltiples clientes que necesitan la misma lógica.
Esta alternativa tiene algunas desventajas. El lenguaje que ofrece la base de datos es poco natural para expresar las complicadas reglas, validaciones y cálculos. La base de datos fue diseñada para persistir los datos y no para escribir algoritmos complicados que consuman estos datos. Los store prodecures se escriben en lenguaje propietario, lo cual dificulta el cambio de proveedor de base de datos.

3 layers

Al mismo tiempo que se fue haciendo popular el esquema client-server, el paradigma orientado a objetos empezaba a nacer.[3]
El hecho de que este paradigma sea considerado muy expresivo y natural para diseñar e implementar la lógica de dominio, fue la causa del surgimiento de un nuevo layer que permita separar la lógica, de los datos. Esta capa se denominó “layer de dominio”.
No es obligatorio el uso del paradigma orientado a objetos en este layer, pero es el más utilizado; principalmente en aplicaciones enterprise, en las cuales abundan validaciones, cálculos y reglas complicadas.
Con este nuevo layer nace la arquitectura de tres layers, que se muestra en la Figura 2.

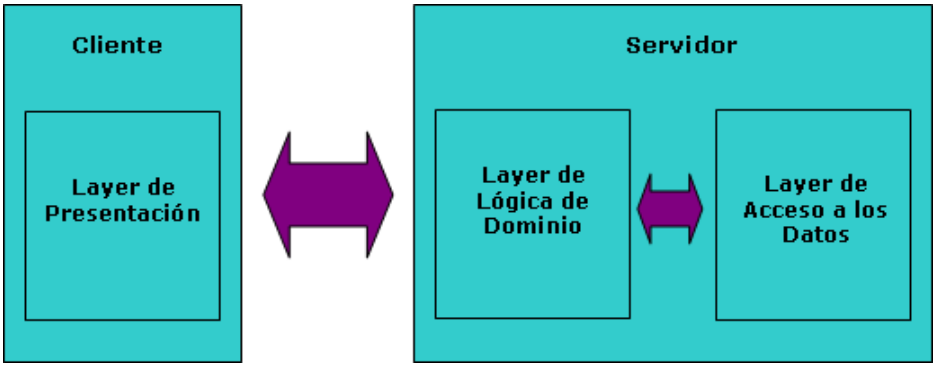


Figura 2: Arquitectura de 3 layers

A continuación se describen las responsabilidades de cada una de las capas:

- Layer de presentación: Interfaces gráficas responsables de mostrar información e interpretar la interacción del usuario para ejecutar los comandos correspondientes que serán quienes consuman la lógica de dominio. Es responsable de hacer validaciones básicas de cliente.
- Layer de lógica de dominio: Es responsable de realizar validaciones de datos ingresados por el usuario y de ejecutar la lógica de dominio correspondiente, que en aplicaciones enterprise frecuentemente incluirá cálculos complejos y reglas del dominio complicadas.
- Layer de acceso a los datos: Se encarga de la comunicación con el mecanismo de persistencia que se utilice; en general, una base de datos. Si se trata de una base de datos relacional, este layer será el responsable de realizar el mapeo objeto/relacional para lograr mapear los objetos en tablas y los atributos en campos de esas tablas.

N layers

A medida que los sistemas se tornaron más exigentes con sus arquitecturas, nuevos layers fueron surgiendo. Este surgimiento de nuevos layers permitió clarificar y distinguir la responsabilidad de cada elemento de una arquitectura en capas, además de reforzar el desacoplamiento necesario para favorecer el mantenimiento y extensión de las aplicaciones. Es así como surgen las arquitecturas n-layered.[3]
En el capítulo 7 se describirá la arquitectura N-layered, en la cual se basa el caso de estudio.

Problema

La performance es uno de los requerimientos no funcionales más problemáticos en el desarrollo de aplicaciones enterprise. En general, las exigencias que presenta este requerimiento no funcional, no son alcanzadas por el producto final o bien, para poder conseguirlo es necesario invertir mucho tiempo.
Existen distintos términos que se utilizan cuando hablamos de performance:

- Tiempo de respuesta: tiempo que tarda el sistema en procesar un pedido externo (por ejemplo, a partir de presionar un botón).
- Tiempo de respuesta al usuario: tiempo que tarda el sistema en responder al usuario, aunque posiblemente no haya terminado de procesar el requerimiento. En un proceso asíncrono, aún con un tiempo de respuesta alto, se podrá bajar el tiempo de respuesta al usuario si se da una respuesta inmediata indicando que su

proceso fue encolado.
- Tiempo de latencia: Es el tiempo mínimo de cualquier respuesta, aunque el trabajo a realizar sea inexistente.
- Throughput: Cuanto trabajo se puede hacer en un tiempo determinado. En aplicaciones enterprise suelen medirse en transacciones por segundo (tps).

La performance puede convertirse en una característica que, de no cumplirse, afecte la usabilidad y las funcionalidades que el mismo brinda, pudiendo causar una mala aceptación del usuario y hasta el fracaso del proyecto. [3]
En muchos casos, el tiempo que demora la aplicación en responder a los requerimientos del usuario de manera correcta y el hecho de asegurar que la respuesta correcta llegue al usuario ante cargas extremas del sistema, es tan importante como retornar el resultado correcto.
Alcanzar buena performance en aplicaciones enterprise es más complicado que en otro tipo de aplicaciones. Entre los factores que pueden afectar el rendimiento se encuentran:
- A nivel físico: el acceso a la base de datos, el enlace de red utilizado y la capacidad de procesamiento de los servidores involucrados.
- A nivel diseño: la arquitectura, las pautas de desarrollo definidas, la utilización de los frameworks, los accesos a la base de datos, la cantidad de datos que se deben mostrar en las interfaces de usuario, la manipulación de grandes volúmenes de datos, entre otros.

Tomar decisiones sobre la performance es una tarea difícil. Es importante considerarla en cualquier decisión de diseño, tanto a nivel de arquitectura como a nivel de diseño y desarrollo de la aplicación.
De todas maneras, aún habiéndola tenido en cuenta en las etapas tempranas de diseño e inclusive en la definición de la arquitectura, los problemas de performance pueden producirse y el equipo debe estar preparado para poder afrontarlos. No alcanzará solo con analizar el diseño sino que se necesitarán herramientas de medición y “profiling” para poder descubrir un posible cuello de botella, un algoritmo ineficiente, un método invocado varias veces de manera innecesaria o cualquier otra causante del problema.

Una de las necesidades más comunes de las aplicaciones enterprise es la persistencia de sus objetos de negocio. Para aplicaciones robustas, transaccionales, escalables y de alta performance se requiere un sistema de manejo de bases de datos para almacenar los objetos de negocio.
El uso de objetos en las capas de servicios y de dominio implica que, conceptualmente, la persistencia es de objetos. Las bases de datos orientadas a objetos son quizás la forma más natural de persistir un modelo de objeto. Una de las ventajas más importantes es que almacenan los datos en el mismo formato con el que los lenguajes orientados a objetos los manipulan, es decir, almacena los datos como objetos. Esto permite que grafos de objetos complejos se persistan directamente. En bases de datos orientadas a objetos trabajar con datos complejos es mas performante que en una base de datos relacional, porque los datos se leen en el mismo formato con el que el lenguaje los utiliza. No se requiere traducción alguna y por lo tanto no existe el conocido problema de desajuste de impedancia.
Sin embargo existen varias razones que hacen que en la actualidad se sigan eligiendo las bases de datos relacionales. Los programadores están familiarizados y tienen mayor experiencia en bases de datos relacionales, los vendedores de bases de datos orientadas a objetos son compañías mas pequeñas y menos conocidas comparadas con las que proveen bases de datos relacionales y fundamentalmente, el mercado de las bases de datos orientadas a objetos es aun pequeño e inestable comparado con el mercado de las bases de datos relacionales.

Por estas razones, las aplicaciones enterprise, en la actualidad utilizan mayormente, bases de datos relacionales. Resulta ser la mejor alternativa a pesar de que genera un desajuste de impedancia, que termina siendo la causa de los problemas de performance, por no tener ciertos recaudos a la hora de diseñar la capa de acceso a datos y la forma en que dicha capa se consume.

Alcance de la tesis

En este tesis se analizarán distintas estrategias para implementar la capa de acceso a datos, en arquitecturas donde la capa de dominio y la capa de servicios utilizan el paradigma orientado a objetos para llevar a cabo el modelado e implementación del sistema. El capítulo 2 describe el desajuste de impedancia objeto/relacional que se genera en sistemas con estas características y las alternativas para poder resolverlo. En el capítulo 3 se describen las estrategias de mapeo objeto/relacional. El capítulo 4 presenta el estado del arte de los mapeadores objeto/relacional, la solución más utilizada en la industria para resolver el desajuste de impedancia. Luego de haber presentado el contexto del problema, en el capítulo 5, se plantean los problemas de performance en el uso de mapeadores objeto/relacional. El capítulo 6 analiza las estrategias de optimización propuestas a los problemas planteados. Finalmente, en el capítulo 7, se presenta el caso de estudio, donde se muestran los resultados de haber aplicado ciertas estrategias presentadas en el capítulo anterior.

Capítulo II - Desajuste de impedancia objeto/relacional

Introducción

Hoy en día, el entorno de programación para sistemas de software mas ampliamente utilizado es la tecnología de objetos y la forma más popular de almacenar y recuperar los datos es mediante el uso de bases de datos relacionales.
Las aplicaciones enterprise se caracterizan por poseer un dominio complejo con lógica de negocio compleja y muchas reglas de negocio, donde las reglas varían con el tiempo, van modificando las reglas actuales y se van nutriendo con nuevas reglas. En este tipo de aplicaciones resulta beneficioso modelar el dominio utilizando el paradigma orientado a objetos. Esto permite obtener un modelo del dominio formado por objetos muy similares a la realidad que se rigen según las reglas de negocio.
Una de las necesidades más comunes de todas las aplicaciones de negocio es la persistencia de sus objetos de negocio. Dentro de las opciones de almacenamiento persistente se encuentran los sistemas de archivos, bases de datos relacionales y bases de datos orientadas a objetos.
Los sistemas de manejo de bases de datos relacionales son un repositorio pragmático y muy popular para almacenar los objetos de negocio debido a la madurez de la tecnología de los sistemas de manejo de base de datos relacionales (RDBMS) y a la disponibilidad de numerosas herramientas para analizar y manejar datos relacionales.
Sin embargo cuando se intenta implementar un software orientado a objetos, ocurre un problema: los objetos no se pueden almacenar y recuperar directamente en una base de datos relacional. Los objetos tienen identidad, estado y comportamiento además del dato en sí mismo, mientras que las bases de datos relacional solo almacenan los datos. Los datos suelen representar un problema en sí mismos, porque frecuentemente no hay un mapeo directo entre los tipos de datos que se manejan en los lenguajes orientados a objetos y los de un sistema de manejo de base de datos relacionales. Además mientras los objetos se relacionan y atraviesan usando referencias directas, las tablas de un RDBMS se relacionan mediante claves primarias y foráneas.
Por estas incompatibilidades entre los paradigmas de objetos y relacional es que, para el desarrollo de aplicaciones orientadas a objetos robustas y escalables se requiere encontrar estrategias de mapeos complejas y eficientes, basándose en un conocimiento sólido de las similitudes y deferencias de ambos modelos. De esta manera, los objetos en memoria se transforman en persistentes en una base de datos relacional.

Modelo de Objetos

Los modelos de objetos describen a los sistemas como un conjunto de objetos. Los sistemas pueden verse como abstracciones de programación que poseen identidad, estado y comportamiento. Los objetos son abstracciones donde los datos y el comportamiento constituyen un concepto unificado. El modelo de objetos también abarca otros conceptos de alto nivel, como abstracción, encapsulamiento, herencia, modularidad y polimorfismo. [4]

Identidad

Los objetos tienen identidad. La identidad permite distinguir un objeto de otros. Cuando se

crea un objeto, este ya es distinguible, incluso de otros objetos que tengan el mismo estado o valores para sus atributos y aparentan ser iguales.

Estado

Si bien un objeto puede distinguirse independientemente de sus valores, un objeto tiene un estado. El estado es el valor actual asociado a su identidad. Los objetos pueden tener un único estado a lo largo de su ciclo de vida o pueden pasar por diferentes estados. Como un objeto está encapsulado, su estado es una abstracción y solo es visible y se puede conocer mediante el comportamiento del objeto.

Comportamiento

Los objetos proveen una abstracción a sus clientes para que estos puedan interactuar con ellos. El comportamiento de un objeto es el conjunto de operaciones que este provee, es decir su interface, las respuestas a estas operaciones que dá al objeto que los invocó y los cambios que causa la operación en el objeto u otros objetos del sistema. Toda interacción con el objeto debe ser a través de su interface y el conocimiento del objeto está dado por su comportamiento al interactuar con su interface.

Encapsulamiento

Es una abstracción que oculta la implementación interna de un objeto, de manera que no pueda ser accedida desde otros objetos. Los clientes interactúan con el comportamiento público del objeto pero no pueden ver como el comportamiento y el estado están implementados.

Tipo

Un tipo es una especificación de una interface que los objetos deben soportar. Un objeto implementa un tipo si proporciona la interface que describe ese tipo. Todos los objetos del mismo tipo pueden interactuar a través de la misma interface. Un objeto puede implementar varios tipos.

Asociaciones

Los tipos pueden ser asociados con otros tipos. Una asociación especifica que objetos de un tipo pueden relacionarse con objetos de otro tipo. Es posible navegar de un objeto a otro asociado, siempre que existe un link o enlace entre ellos.

Clase

Una forma de implementar objetos es tener una clase. Una clase define una implementación para varios objetos. Una clase define que tipos de objetos deberán implementar, como ejecutar el comportamiento requerido por la interface y como recordar la información de estado del objeto. Cada objeto solo necesita recordar su estado individual. Este es el concepto principal del modelo orientado a objetos.

Herencia

La herencia puede aplicarse a tipos o clases. Cuando se aplica a tipos, la herencia especifica que si un objeto es de Tipo B donde Tipo B hereda de Tipo A, entonces el objeto puede usarse como objeto de Tipo A. El Tipo B se dice que sigue el patrón del Tipo A y todos los objetos que son de Tipo B son también de Tipo A.
Cuando se aplica a clases, la herencia especifica que una clase usa la implementación de otra con la posibilidad de sobrescribir su comportamiento.

Abstracción

Es el acto de representar las características esenciales sin tener en cuenta los detalles de contexto y aclaraciones. Las clases usan el concepto de abstracción y se definen como una lista de atributos abstractos.
La abstracción facilita la conceptualización de los objetos del mundo real, extrayendo la información esencial de un objeto y eliminando los detalles innecesarios.

Modularidad

La programación modular es una técnica de diseño de software que consiste en dividir el software en un conjunto de partes separadas llamadas módulos. Un módulo es un componente de un sistema más amplio que debe operar dentro de ese sistema independientemente de las operaciones de los otros componentes.

Polimorfismo

La capacidad que distintos objetos puedan responder de maneras diferentes al mismo mensaje es conocido como polimorfismo.
Simplifica la interface de programación y facilita el mantenimiento y reuso. La interface de programación se describe como un conjunto de comportamientos abstractos y las clases que lo requieren, implementan este comportamiento a su manera.

Modelo Relacional

El modelo relacional describe información mediante lógica de predicados y sentencias de verdad.
Provee una representación lógica de los datos. A partir de este modelo lógico, permite que la base de datos pueda obtener y retornar información original persistida y proveer información derivada. El modelo relacional presenta un conjunto de términos bien definidos como relación, tupla, dominio y base de datos, para traducir las sentencias de los usuarios al modelo de base de datos.

Relación

Una relación es un predicado de verdad. Define que atributos están involucrados en el predicado y cual es el significado del predicado. Generalmente, el significado de la relación no esta representado explícitamente. Por ejemplo,
Persona {#DNI, Nombre, Ciudad}

Atributo

Un atributo identifica un nombre que participa en la relación y especifica el dominio para los valores posibles del atributo. Por ejemplo, Nombre en la relación Persona es un atributo definido sobre el dominio String. En la relación anterior debería explicitarse los dominios de cada atributo,
Persona (#DNI: Integer, Nombre: String, Ciudad: Ciudad}
Los atributos son comúnmente conocidas con el nombre de columnas.

Dominio

Un dominio es simplemente un tipo de datos. El dominio especifica una abstracción de datos, es decir los valores posibles para los datos y las operaciones disponibles sobre los datos. Por ejemplo, un String puede tener cero o más caracteres y existen operaciones de comparación, concatenación y creación de strings.

Tupla

Una tupla es una sentencia de verdad en el contexto de una relación. Una tupla tiene valores de atributos, los cuales se corresponden con los atributos requeridos en la relación y su estado es verdadero. Por ejemplo,
<Persona #DNI=”26557469” Nombre=”Belena Zulaica” Ciudad=”La Plata” >
Las tuplas son valores y dos tuplas son idénticas si su relación y los valores para los atributos son iguales.
Las tuplas son comúnmente conocidas con el nombre de filas.

Valor de Atributo

Un valor de atributo es el valor para un atributo en una tupla particular. Un valor de atributo esta definido por el dominio especificado para ese atributo.

Valor de Relación

Un valor de relación esta compuesto de una relación y un conjunto de tuplas. Todas las tuplas deben tener la misma relación y como forman parte de un conjunto, las tuplas no tienen orden y no existen duplicadas. Por ejemplo,
{ <Persona #DNI=”26557469” Nombre=”Belena Zulaica” Ciudad =”La Plata” >,
<Persona #DNI=”30687956” Nombre=”Marta Morales” Ciudad=”Junín” >,
<Persona #DNI=”22235587” Nombre=”José Martínez” Ciudad=”La Plata” >}
Es más común ver los valores de relación como una tabla:

: Persona

#DNI	Nombre	Ciudad
26557469	Belena Zulaica	La Plata
30687956	Marta Morales	Junín
22235587	José Martínez	La Plata

Variable de relación

Una variable de relación almacena un único valor de relación en un momento determinado, sin embargo su valor puede cambiar en cualquier momento. Las variables de relación son tipadas a una relación particular. Por ejemplo,
empleado: Persona
Usando una tabla, podemos ver la variable de relación y su valor actual.

empleado: Persona

#DNI	Nombre	Ciudad
26557469	Belena Zulaica	La Plata
30687956	Marta Morales	Junín
22235587	José Martínez	La Plata

Las variables de relación son comúnmente conocidas con el nombre de tablas.

Base de datos

Una base de datos es un conjunto de variables de relación. Describe el estado completo de un modelo de información, puede cambiar de estado y puede responder preguntas sobre un estado particular.

Comparación de los modelos de Objetos y Relacional

Una relación es similar a una clase. No obstante, una relación no presenta los conceptos de herencias que se encuentran en una clase. Una tupla es similar a una instancia de un objeto. Aunque, una tupla se limita a la estructura de datos que la contiene y una instancia de un objeto puede almacenar cualquier estructura de datos soportada por el lenguaje. Una columna es similar a un atributo. Una columna está limitada a ciertos tipos de datos. Un atributo puede soportar cualquier tipo de todos los permitidos por el lenguaje, incluyendo referencias a otros objetos.
Un procedimiento almacenado es significativamente diferente a un método. Los métodos son computacionalmente completos porque son escritos en leguajes de programación orientados a objetos. Los lenguajes de los procedimientos almacenados son mucho mas limitados.
La Tabla 1 muestra la comparación entre los conceptos del modelo orientado a objetos y el relacional.[5]

Conceptos del modelo de objetos	**Concepto en el modelo relacional**	**Concepto en el modelo de objetos**	**Beneficio del modelo de objetos**
Abstracción de datos	Relaciones entre entidades e índices para representar relaciones entre tuplas.	OIDS para referencias directas entre objetos.	Esquema más simple para representar datos complejos.
Herencia	Códigos de tipos.	Jerarquía de clases	Representación directa de referencias entre tipos y subtipos. Soporte para el procesamiento específico de cada subtipo.
Encapsulamiento	Código “if then else if” basado en los códigos de tipo y manejo de código con librerías.	El encapsulamiento provee *dispatching* (ejecución del código correcto basado en el tipo de una clase)	Reducción del código de aplicación y reducción de posible errores cuando código incorrecto se ejecuta sobre un dato correcto.

Tabla 1: Comparación entre modelo relacional y modelo de objetos

Desajuste de Impedancia

La tecnología orientada a objetos permite la construcción de aplicaciones basándose en objetos que tienen tanto datos como comportamiento. Las tecnologías relacionales soportan el almacenamiento de datos en tablas y permiten manipular los datos mediante un lenguaje de manipulación de datos (DML). Esta manipulación de datos se hace internamente vía procedimientos almacenados o externamente vía llamadas SQL.
Tanto la tecnología de objetos como la relacional son utilizadas comúnmente en la mayoría de las organizaciones, se seguirán usando por un largo tiempo y fundamentalmente, ambas se utilizan para construir sistemas de software complejos. Es evidente que la correspondencia entre estas dos tecnologías no es perfecta, es decir, que existe un desajuste de impedancia entre ambas. [6]
Para la mayoría de las aplicaciones, almacenar y recuperar información implica alguna forma de interacción con una base de datos relacional. Esto representa un problema fundamental para los desarrolladores porque muchas veces el diseño de datos relacionales y los modelos orientados a objetos comparten estructuras muy diferentes.
Las bases de datos relacionales están estructuradas en una configuración tabular y los modelos de objetos normalmente están relacionados en forma de grafo. Esta inherente incompatibilidad conceptual entre los paradigmas, denominada “desajuste de impedancia objeto/relacional” o simplemente “desajuste de impedancia”, ha llevado a los desarrolladores de varias tecnologías de persistencia de objetos a intentar construir un puente entre el mundo relacional y el mundo orientado a objetos. [7]
El término “desajuste de impedancia” (impedance mismatch) proviene de la ingeniería eléctrica donde la impedancia mide la resistencia del cable a la corriente alterna en un circuito. La forma mas eficiente de intercambio entre sistemas eléctricos ocurre cuando sus impedancias son casi iguales. El desajuste de impedancia es un problema que sucede cuando se conectan dos circuitos con diferentes impedancias, pudiendo ocasionar atenuación o ruido.
La analogía en la ingeniería de software es que los sistemas orientados a objetos frecuentemente tienen un desajuste con los sistemas de bases de datos relacionales. Para clarificar la metáfora, el desajuste de impedancia tanto en la ingeniería eléctrica como de software ocurre cuando uno de los sistemas no puede interactuar eficientemente con el otro.
El desajuste de impedancia objeto/relacional ocurre porque los paradigmas orientado a objetos y relacional tienen diferentes concepciones de los datos. El paradigma orientado a objetos ve los datos principalmente en el contexto de las acciones que se ejecutan sobre ellos. Es decir, los objetos son importantes no solo por el dato que contienen sino también por la posibilidad de ejecutar tareas sobre los datos e intercambiar información con otros objetos. El paradigma relacional está centrado en los datos. El lugar más importante lo ocupan los datos en si mismos y sus relaciones estructurales (no de comportamiento) con otros datos.
El mapeo de objetos a tablas y viceversa puede resultar sencillo cuando ambos modelos son pequeños y los datos son simples. Sin embargo, el mapeo puede tener un alto impacto en la performance de la aplicación cuando los datos a transformar son complejos. Por suerte existen estrategias para mitigar este desajuste de impedancia objeto/relacional. [6]

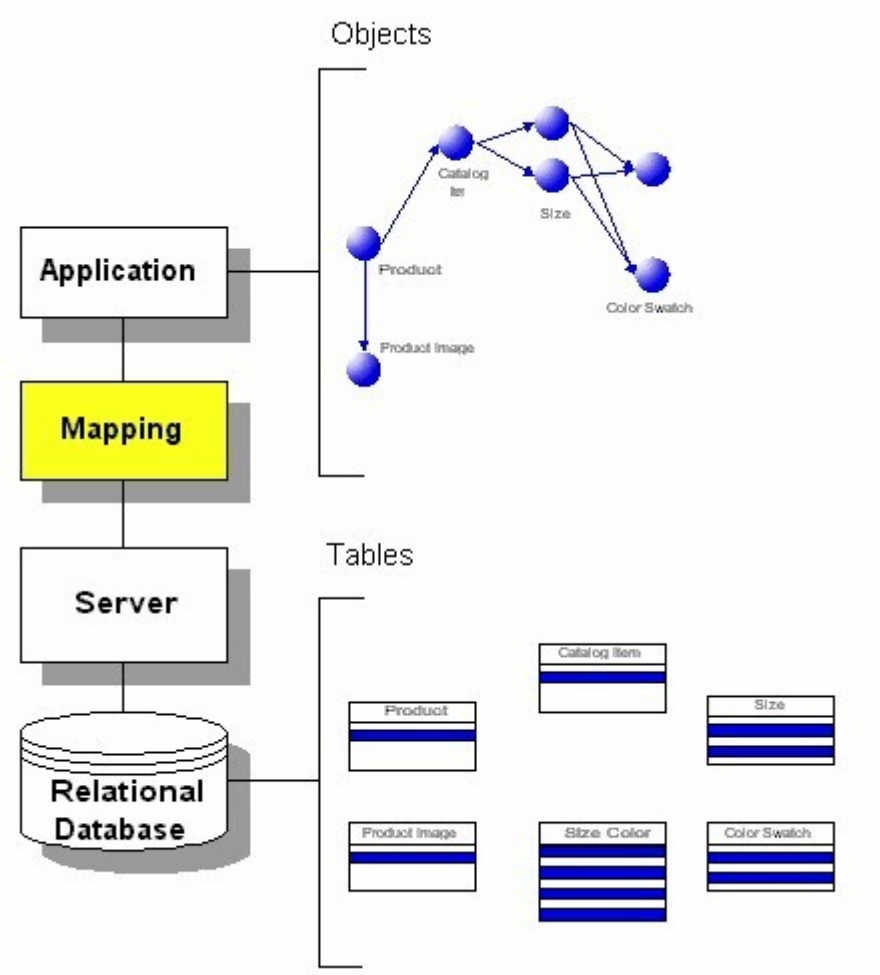


Figura 3: Desajuste de impedancia objeto/relacional [8]

La mayoría de las investigaciones sobre el desajuste de impedancia se centra en las diferencias técnicas entre las tecnologías de objetos y relacional. Esto tiene una justificación, ya que si bien hay algunas similitudes descriptivas también hay diferencias aun más importantes. Desafortunadamente, se le presta menos atención a las diferencias culturales entre la comunidad orientada a objetos y la comunidad de datos. Estas diferencias se ponen de manifiesto cuando los profesionales de ambas comunidades discuten sobre que aproximación elegir para un proyecto.[6]

Desajuste de impedancia técnico

El desajuste de impedancia técnico está relacionado con las diferencias entre los paradigmas objeto/relacional mencionado anteriormente.[9]

Herencia y polimorfismo

Todos los objetos de un sistema completo existen junto a una jerarquía de herencia intrínseca. Podemos decir que todos los objetos derivan de la superclase Object. Sin embargo este concepto no es intrínseco en el modelo relacional, el cual está centrado en las relaciones estructurales de los datos.
Cuando pensamos en mapear una jerarquía de herencia en la base de datos, existen varias alternativas. La mas trivial es mapear cada clase en su propia tabla. Por ejemplo considerando la jerarquía de clases de la Figura 4, el resultado será una tabla para Fruta,

otra para Manzana, otra para ManzanaRojaDeliciosa, y así siguiendo.

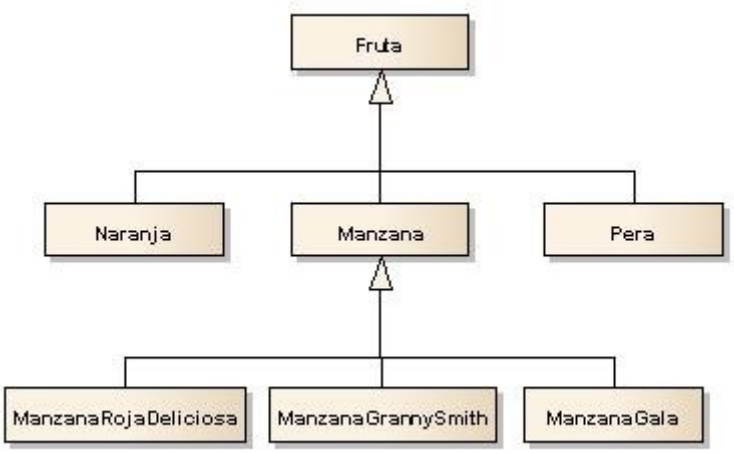


Figura 4: Jerarquía de clases

Sin embargo hay otras alternativas de mapeo que se discutirán mas adelante.
Una de las características mas substanciales que provee el mecanismo de herencia en los lenguajes orientados a objetos es el polimorfismo de subtipos.
Siguiendo con el ejemplo, supongamos que tenemos una clase Persona que modela datos de su nutrición (ingesta calórica, consumo de proteínas, deficiencias vitamínicas) y que existe un método comer que recibe una Fruta como argumento. El polimorfismo de subtipos permite actualizar su información nutricional de manera apropiada según el tipo de fruta que está ingiriendo. En bases de datos relacionales no hay una forma directa de representar este polimorfismo. Por ejemplo, si se quieren persistir las frutas que la persona ha ingerido, es necesario tener una relación entre las tablas Fruta y Persona. Sin embargo, hay que realizar un análisis sobre como trabajar con todas las tablas que almacenan algún subtipo de fruta.

Asociaciones

Las clases en los sistemas orientados a objetos se relacionan entre sí de diferentes maneras. Una forma es enviándose mensajes, es decir invocando métodos de otras clases. Este tipo de relaciones de comportamiento, normalmente no se representa en el modelo relacional. La otra forma de relacionar clases es mediante asociaciones.
Consideremos una aplicación de software que modela los empleados y sus certificaciones. Supongamos que cada empleado tiene una certificación y que varios empleados pueden tener la misma certificación. Hay varias formas de modelar esta relación en un lenguaje orientado a objetos. Primero, podemos referenciar la clase Certificacion en la clase Empleado.

```
public class Empleado {
        private string nombre;
        private string sueldo;
        private Certificacion certificacion;
}
```

Alternativamente, la clase Certificacion podría tener una colección de objetos Empleado, indicando que empleados tienen esa certificación.

```
public class Certificacion {
        private string nombre;
        private int validez;
        IList<Empleado> empleados;
}
```

Si la asociación tiene que ser atravesada en ambas direcciones, es necesario implementar ambas asociaciones. Independientemente de la elección que se haga, se va a mapear en la misma estructura de datos, una tabla Empleado y una tabla Certificacion con una relacion muchos-a-uno como se muestra en la Figura 5.

Figura 5: Relación muchos a uno

Si ahora un empleado puede tener varias certificaciones y la misma certificación la puede obtener más de un empleado al mismo tiempo. En el código orientado a objetos, hay que cambiar en la clase Certificacion, la referencia a la clase Empleado por una referencia a una colección de Certificacion. Sin embargo, el modelo relacional no se acomoda naturalmente a las relaciones muchos-a-muchos. En este caso, es necesario crear una nueva tabla para representar la relación propiamente dicha y relacionar las tablas Certificacion y Empleado con esta nueva tabla como muestra la Figura 6.

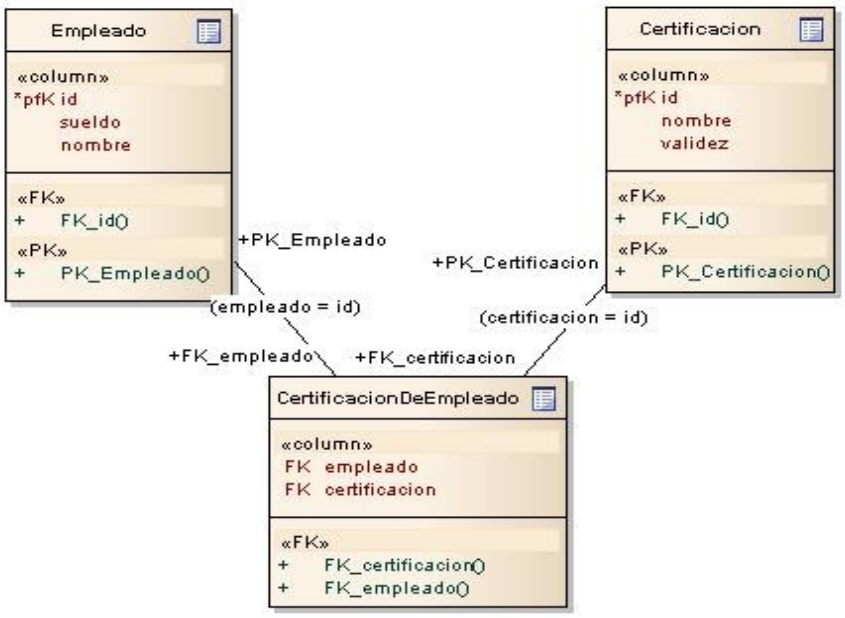


Figura 6: Relación muchos a muchos

Tipos de datos

Los sistemas de manejo de bases de datos y los lenguajes orientados a objetos tienen sus propios tipos de datos y no es completamente claro como hacer un mapeo entre ellos. Algunos lenguajes tiene tipos primitivos que no se mapean directamente a los tipos de SQL estándar.
Es necesario que la aplicación orientada a objetos resuelva estas diferencias para poder comunicarse correctamente con la base de datos.

Granularidad

Supongamos que tenemos las siguientes clases:

```
public class Cliente {
        private int numero;
        private String nombre;
        private Direccion direccion;
        private String telefono;
}

public class Direccion {
        private String calle;
        private String numero;
        private String ciudad;
        private String codigoPostal;
}
```

En este caso, en lugar de incluir los campos de la direccion en la clase Cliente, los encapsulamos en una nueva clase Direccion, la cual puede utilizarse varias veces en la aplicación. La alternativa mas simple es mapear cada clase a su propia tabla, pero esto puede generar problemas de performance ya que cada vez que se necesita la dirección de un empelado hay que hacer join de ambas tablas. La otra alternativa es incluir toda esta información en una única tabla. Si bien esto no es un espejo de la estructura de las clases en el lenguaje de objetos, esta alternativa muchas veces es elegida, no solo por los beneficios de performance sino también porque resulta mas intuitivo tener toda la información de un empleado almacenada en un mismo lugar. Si el RDBMS soporta tipos definidos por el usuario, se podría representar Direccion como un tipo definido por el usuario, donde la tabla Cliente tenga una columna de tipo Direccion para almacenar los valores de la dirección de manera compacta y eficiente.
Este es un ejemplo de incompatibilidad de granularidad. Las clases en el modelo orientado a objetos tienen muchos niveles de granularidad, desde clases de grano grueso como Cliente, la cual modela objetos de negocio importantes, hasta clases de grano fino como Direccion. Cuando se lleva a cabo el mapeo objeto/relacional, se debe decidir que clases requieren sus propias tablas para persistirse y cuales no. Las clases de grano fino se pueden mapear a tipos definidos por el usuario, a un conjunto de columnas o a una única columna, según el caso.

Identidad

Las claves primarias en la base de datos aseguran que todos los registros de una tabla sean únicos, aunque sean idénticos en cuanto al contenido con otros registros. En decir, la clave primaria da la identidad a cada registro.
Los objetos en el paradigma orientado a objetos tienen dos tipos de identidad, es decir dos maneras mediante las cuales se puede determinar si dos objetos son iguales o no.

Ellas son la comparación por referencia y la comparación por valor. La comparación por referencia chequea si las referencias de ambos objetos apuntan al mismo objeto. En Java por ejemplo, dos objetos a y b se comparan por referencia usando la sintaxis a == b. Esta forma de comparación equivale a comparar si dos objetos ocupan el mismo espacio de memoria.
La comparación por valor compara dos objetos, atributo por atributo. En Java, por ejemplo, dos objetos a y b se comparan por valor mediante la sintaxis a.equals(b). Esta forma de comparación indica si dos objetos almacenan la misma información.
Estos tres tipos de identidad son todos diferentes. La comparación por referencia difiere de la comparación de identidad de base de datos en que dos objetos no necesitan compartir el mismo espacio de memoria para representar la misma entidad de base de datos. La comparación por valor difiere de la comparación por identidad de base de datos porque el punto de la identidad es que dos entidades de base de datos pueden ser diferentes aun si ellas poseen el mismo contenido (aparte de su identidad).
Existen varias formas de resolver este desajuste, pero generalmente es el código orientado a objetos el cual es adaptado para corresponderse con la base de datos relacional. Frecuentemente, la clave primaria de una tabla se transforma en un atributo de la clase orientada a objetos correspondiente.

Cantidad de consulta de base de datos

La navegación del grafo de objetos de una aplicación orientada a objetos puede resultar en una explosión exponencial de consultas a la base de datos.
Supongamos que tenemos una aplicación en la cual cada cliente tiene varias órdenes de compra y que cada orden puede tener muchos ítems, como muestra la Figura 7.

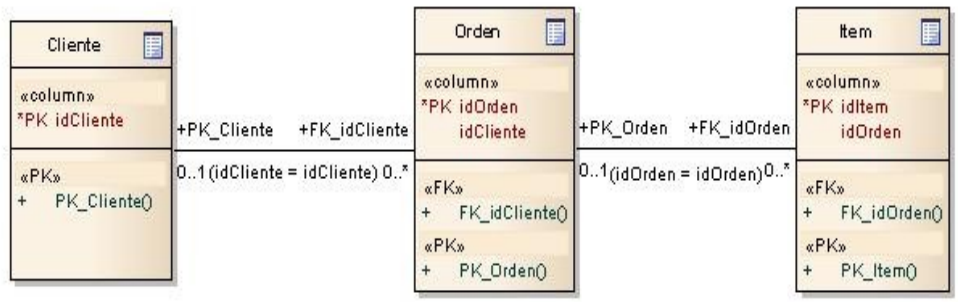


Figura 7: Relación muchos a muchos

Supongamos también, que en la aplicación queremos listar todos los ítems de las órdenes que adquirió un determinado cliente. Para obtener esta información es necesario escribir algún método como el siguiente:

```
void  procesarItemsDeOrdenesDeCliente (Cliente cliente) {
        List<Orden> ordenes = cliente.getOrdenes();
        for (Orden orden: ordenes) {
                List<Item> items = orden.getItems();
                for (Item item: items) {
                        System.out.println(item.toString());
        }
}
```

Cuando se accede a las órdenes del cliente se realiza una consulta a base de datos, lo

mismo ocurre cuando se accede a los ítems de cada una de las órdenes. Si el cliente tiene n ordenes y cada una de ellas tiene m ítems se realizan n*m accesos a la base de datos. Por cada acceso se abre una conexión a la base de datos, se ejecuta la consulta SELECT correspondiente y se cierra la conexión. Por este comportamiento es que la ejecución de este método puede ocasionar problemas de performance.
La alternativa para evitar este problema es reducir la cantidad de accesos a la base de datos. En este caso se podría obtener toda la información necesaria en una única consulta del estilo:

```
SELECT Item.* FROM Cliente
INNER JOIN [Orden]
ON Cliente.idCliente= [Orden].idCliente
INNER JOIN Item ON [Orden].idOrden= Orden.Item.IdOrden
```

Sin embargo, lograr mapeos que permitan acceder a la base de datos mediante consultas simples no es trivial.

Además de las diferencias técnicas mencionadas, existen otras dificultades, tales como:

- Aproximación declarativa versus imperativa. Los lenguajes orientados a objetos son lenguajes imperativos. Tiene métodos que nos dicen como hacer algo a partir de una secuencia de sentencias. En contraposición, SQL es un leguaje declarativo, el cual solo dice que hay que hacer.
- Normalización y modelado. Crear un esquema de base de datos es un proceso diferente al de modelar un modelo de objetos. Comúnmente intentamos tener un esquema de base de datos normalizado, sin redundancia y ambigüedad en los datos. Por otro lado, cuando realizamos un modelo de objetos usamos diferentes técnicas y conceptos, tales como, herencia, polimorfismo y abstracción. El diseño orientado a objetos esta frecuentemente influenciado por análisis orientado a objetos y el uso de patrones de diseño.

Desajuste de impedancia cultural

Ya hablamos del conocido desajuste de impedancia técnico entre las tecnologías orientadas a objetos y las relacionales. También existe, aunque no es tan conocido, el desajuste de impedancia cultural. Este se refiere específicamente a las dificultades que se presentan cuando los desarrolladores especialistas en objetos y los especialistas en datos trabajan juntos en un proyecto y a las disfuncionalidades políticas que presentan estas dos comunidades en la industria.
Algunos de los síntomas que indican que existe un problema de impedancia cultural son:

- Desarrolladores de objetos que argumentan que la tecnología relacional no debe o no puede ser utilizada para almacenar objetos.
- Profesionales de datos (van desde diseñadores a administradores de base de datos) que afirman que el modelo de objetos debe ser conducido por el modelo de datos.
- Desarrolladores de aplicación que argumentan que porque ellos usan framework de persistencia, no necesitan entender la tecnología de objetos subyacente.
- Profesionales de datos que están en desacuerdo con el desorden con los datos que ocasionan los desarrolladores de aplicación. Aunque, rara vez tiene la intención de capacitar a los desarrolladores para hacer este trabajo correctamente.

Esta resistencia de los equipos de desarrollo de interactuar eficientemente con los grupos encargados del manejo de datos agrava los problemas de calidad de los datos en las

organizaciones. Los equipos de desarrollo no son capaces de aprovechar el conocimiento, habilidades y experiencia de los profesionales sobre los datos. Por esta razón ocurren errores como crear una nueva fuente de datos en lugar de aprovechar las fuentes de datos existentes, diseñando una base de datos de bajo nivel debido a no tener los conocimientos necesarios en este área y sin cumplir con las convenciones de nombres de datos y metadata.[6]

¿Que hacer?

El primer paso es el reconocimiento. Ambos grupos deben aceptar que el problema existe y que necesita ser superado. Desarrolladores de aplicaciones y profesionales en datos tienen habilidades diferentes, entornos diferentes, filosofías diferentes y distintas formas de hacer su trabajo. Se debe encontrar la forma de trabajar juntos aprovechando las ventajas de estas diferencias.

El segundo paso es distinguir el tamaño y tipo del proyecto. Diferentes proyectos requieren diferentes aproximaciones y necesitan ser manejadas según el caso. Un proyecto data warehousing será distinto a un proyecto de desarrollo de un sitio web. Un equipo de trabajo de 3 personas será diferente a un equipo de 30 ó de 300. Un equipo de trabajo ubicado físicamente en un mismo lugar será diferente a un equipo distribuido. Un equipo que trabaja en un ambiente regido por reglas políticas y legales trabajará diferente que si el ambiente no posee reglas. Un equipo que trabaja con sistemas legados trabajará distinto que uno que desarrolla un sistema nuevo.

No es suficiente tener el grupo encargado de los datos y el de aplicación correctos sino que es necesario que ambos trabajen en conjunto. Se requiere capacitar a los desarrolladores de aplicaciones en temas relacionados con los datos y a los profesionales de datos en habilidades de desarrollo.

Por supuesto, cada comunidad debe reconocer que la otra tiene un rol importante para alcanzar el éxito en el proyecto. El trabajo en equipo es lo que permitirá construir el puente entre ambos paradigmas.[10]

Soluciones al desajuste de impedancia

El desarrollo de aplicaciones que requieren acceso a bases de datos existe desde hace varias décadas, sin embargo, nos seguimos enfrentando al problema de desajuste de impedancia. Este problema aún no tiene una solución definitiva, pero contamos con varias estrategias de solución que fueron apareciendo a lo largo del tiempo.
Es necesario analizar cada una de las estrategias de implementación de la capa de acceso a datos, para ver cual es la que mejor se adecua a las necesidades del proyecto. [6]

Fuerza bruta

Es la estrategia básica. En esta aproximación los objetos de negocio acceden directamente a la fuente de datos. Básicamente, consiste en escribir cadenas de caracteres que contienen las sentencias SQL en el código de la aplicación. En aplicaciones Java, esto se lleva a cabo mediante el uso de la librería Java Database Connectivity (JDBC).
Si bien esta no es una estrategia de encapsulamiento del acceso a los datos, es una alternativa válida para resolver el acceso a los datos. Es la alternativa mas común porque es simple y provee al programador un control completo sobre cómo los objetos de negocio interactúan con la base de datos. Por otro lado, este enfoque tiene fuertes limitaciones. Las sentencias SQL se evalúan recién en tiempo de ejecución, lo que retrasa el ciclo de desarrollo, además resulta muy difícil asegurar que toda sentencia SQL elaborada sea siempre válida y correcta. No existe separación entre el código de acceso a los datos y la lógica de negocio. Cuando se trabaja con este esquema, se terminan distribuyendo las operaciones de acceso a la base de datos, por todo el código de la aplicación. Esto hace muy costoso de modificar el código cuando requiere mantenimiento. En otras palabras, es un problema de semántica, por un lado se tiene el significado del programa verificado por el compilador, y por otro, la semántica del SQL que sólo conocerá el DBMS en momento de ejecución.
Sin embargo, la simplicidad de esta solución puede convertirla en una buena alternativa en proyectos donde los requerimientos de acceso a la base de datos son sencillos.

Data Access Objects (DAOs)

Una estrategia para “despegar” el código de acceso a los datos de la lógica de aplicación es el uso del patrón de diseño DAO (Data Access Object). En esta alternativa, el modelo de los objetos de negocio queda representado en objetos que se instancian y se ejecutan en una capa lógica conocida como Capa de Negocio. Esta capa implementa todas las reglas del negocio de la aplicación. Por otro lado, la lógica necesaria para crear, recuperar actualizar y eliminar datos (conocido como CRUD por Create, Retrieve, Update y Delete) se implementa en una capa conocida como Capa de Acceso a Datos. Esta capa contiene objetos que implementan CRUD. A estos objetos se los conoce como Objetos de Acceso a Datos o DAOs.
Muchas veces el uso de DAOs se combina con Data Transfer Objects (DTO). Los DTO son objetos que solo tienen getters y setters para transportar datos entre las capas de la aplicación.
Lo que busca el patrón DAO es encapsular las llamadas a la API del DBMS para lograr un menor acoplamiento entre las clases, facilitando los cambios en el código; pero no elimina el problema de la semántica.
Los DAOs de una aplicación son sencillos de desarrollar, aunque también existen

implementaciones estándares en la industria tales Java Data Object (JDO) y ActiveX Data Object (ADO).

Mapeo Objeto/Relacional

En los años 90s surge una nueva alternativa de solución al problema de desajuste de impedancia, a la que se le conoce como Mapeo Objeto/Relacional. Esta es una técnica de programación en la cual se vincula la base de datos con la aplicación orientada a objetos creando objetos virtuales de la base de datos. Mediante mecanismos, ya sea un archivo XML, atributos en el código de las clases y propiedades, se mapean clases en tablas, propiedades en columnas, asociaciones en relaciones.
A lo largo de toda la aplicación, los objetos que modelan las entidades de negocio y que tienen lógica dentro, son objetos absolutamente planos (POJOs, por Plain Old Java Objects). Los mapeadores O/R suelen tener una API con alguna clase principal a la que se le dice qué tipo de objetos se necesitan. Las consultas no se expresan en SQL sino en un dialecto basado en los POJOs. Esta clase principal se encarga de generar las sentencias SQL necesarias para recuperar los datos solicitados y luego mover la información de las columnas a las propiedades (proceso que se conoce como hidratación). Análogamente, una vez que se procesan esos objetos, es decir se modifican, borran, crean nuevos; se los envía al mapeador O/R y él se encarga de generar los comandos que reflejen el nuevo estado en la base de datos.
Actualmente existen alternativas, tanto comerciales como libres, para automatizar la creación de los esqueletos de código necesarios para adoptar este tipo de soluciones. [11]

Frameworks de persistencia

A partir de la aparición de Java en el mercado, en la segunda mitad de los 90s, junto con el resto de los lenguajes que funcionan dentro de una máquina virtual y permiten operaciones de introspección, surgieron otras alternativas de solución al problema de impedancia. Los frameworks de persistencia buscan automatizar la implementación del patrón DAO, mediante diferentes descriptores (o mappings) que permiten determinar de que manera los objetos de negocio deben ser almacenados en tablas de bases de datos relacionales.
Los frameworks de persistencia extienden el concepto de los mapeadores sumando prestaciones y funcionalidad, por ejemplo los más avanzados implementan mecanismos que permiten detectar cuáles objetos de los que se recuperaron en la lectura, han sufrido modificaciones (es decir, propiedades cambiadas, objetos suprimidos, objetos creados). A estos objetos modificados, Fowler los llama "objetos sucios" ("dirty objects"). Los frameworks son capaces de sincronizar la base de datos con el nuevo estado del modelo, invocando los métodos CRUD necesarios. Algunos frameworks proveen funcionalidad avanzada como control de acceso concurrente a los objetos, implementaciones de caché o control de versiones para modificaciones.
Cuando se habla de frameworks de persistencia se puede pensar en dos alternativas. Una es utilizar alguno de los framewoks que se encuentran disponibles en el mercado, ya sean comerciales como open sources. La otra es construir un framework propietario adaptándolo a las necesidades del proyecto y la organización. Generalmente, es preferible la primera opción, ya que es menos costosa y menos propensa a errores.
Ya entrado en el siglo XXI, se introduce una nueva tecnología que busca que la persistencia sea más transparente al programador, conocida como Hibernate. En lugar de escribir el código que implementa la lógica de acceso a la base de datos, se escribe la

metadata que representa los mapeos (generalmente XML). El framework utiliza esta metadata para generar el código de acceso (SQL) a la base de datos necesario para persistir/recuperar los objetos de negocio mapeados, abstrayendo a las aplicaciones de las distintas capacidades de procesamiento que ofrece cada DBMS específico, por ejemplo diferentes implementaciones del ANSI3 SQL para realizar consultas paginadas. Análogamente, en la plataforma .NET existe NHibernate, una migración de Hibernate a .NET. Otras alternativas puede ser el framework de persistencia ADO.NET, Toplink de Oracle, el estándar J2EE EJB 2.0 que en su versión 3.0 soporta integración con Hibernate.

Bases de datos orientadas a objetos

Una alternativa para evitar enfrentarnos al problema de desajuste de impedancia objeto/relacional es utilizar un sistema manejador de bases de datos orientadas a objetos (OODBMS). Como su nombre lo indica, en estos sistemas, las bases de datos están diseñadas para trabajar con objetos. Elimina la necesidad de convertir el modelo de objetos en datos relacionales y viceversa, porque los datos se almacenan con su representación de objetos original y las relaciones se representan directamente sin requerir joins entre las tablas como en el modelo relacional.
Quienes están a favor del uso de OODBMS argumentan que son recomendables para almacenar grafos de objetos complejos, navegar grafos de objetos de manera rápida y reducir el desajuste de impedancia. En contraposición, quienes prefieren el uso de RDBMS afirman que estas permiten mantener la independencia de los datos, facilitan la generación de reportes complejos y mediante un lenguaje de definición de datos permiten un excelente manejo del esquema relacional.[12]
Algunos puntos que nos dan indicio que sería conveniente pensar en un OODBMS son:

- Uso de objetos que requieren mapeos a tablas RDBMS. Si el código necesario para implementar los mapeos aumenta en un tercio o la mitad del código de la aplicación, lo cual incrementa el costo de desarrollo. El costo de mantenimiento es muy alto, porque un cambio en el diseño, implica un cambio en el código de la aplicación, el cual también genera un cambio en el código del mapeo. RDBMS requiere trabajar continuamente a bajo nivel mientras que OODBMS permite trabajar a cualquier nivel, incluyendo el más alto que es el nivel de aplicación.
- Más de 2 ó 3 niveles de joins. Los joins entre tablas en un RDBMS son más lentos que atravesar relaciones directas en un OODBMS.
- Relaciones muchos-a-muchos. Modelar relaciones muchos-a-muchos en un RDBMS requiere tablas extras, ya que es necesario guardar las relaciones en una tabla asociativa adicional.
- Navegación de jerarquías. En RDBMS la creación, actualización o navegación de jerarquías resulta menos performance y más complejo que en un OODBMS.
- Estructuras de datos complejas o recursivas. Un RDBMS soporta tablas simples y planas. Cualquier otra estructura, complejidad y anidamiento de datos requiere “trucos” de modelado complejos, haciéndolos difícil de implementar, propensos a errores y lentos. En un OODBMS estas estructuras son soportadas directamente.

Por otro lado, algunos puntos a favor de los RDBMS son:

- ORM en sistemas legados. ORM es necesario para manejar esquemas de bases de datos existentes o soportar datos legados.
- Los datos tienen mayor durabilidad que las aplicaciones. Los mapeos son necesarios porque los datos siempre durarán mas tiempo que las aplicaciones que los crean.
- ODBMS son inadecuados para la compatibilidad. Los objetos fuertemente tipados

almacenados en una base de datos orientada a objetos son difíciles de usar con múltiples lenguajes. Números y Strings sencillos en un RDBMS se pueden mapear en distintos lenguajes.

- OODBMS no están suficientemente maduros. RDBMS es más estable y maduro que los ODBMS.
- OODBMS no poseen un lenguaje de consultas estándar. Los RDBMS tienen un lenguaje de consultas estándar (SQL) el cual es soportado por muchos vendedores. También posee algunas interfaces como ODBC y JDBC que permiten conectarse correctamente con diferentes lenguajes y una API estable. Los OODBMS no presentan estas características.

Conclusiones

Se puede decir que "cualquiera sea el modelo de programación utilizado, este debe permitir que, complejas e intensivas operaciones en datos, puedan ser lanzadas por los programas para ejecutarse en un DBMS".
Las soluciones propuestas anteriormente atacan dos principios fundamentales:

- Unificación de semántica: Todo el programa, debe estar especificado utilizando una semántica unificada. Es decir, que no existan desconexiones de significado entre las operaciones que se envían al DBMS y lo que se realizan dentro de la aplicación.
- Eficiencia: Buscar la eficiencia en la ejecución delegando las operaciones intensivas sobre los datos al DBMS y las intensivas en cálculos a la aplicación.

Las soluciones al problema del diferencial de impedancia propuestas intentan resolver ambos problemas de manera simultánea, no obstante estas soluciones sólo resuelven una en detrimento de la otra. Por ejemplo, el mapeo objeto/relacional con sus variantes, resuelve la eficiencia en la ejecución, pero rompe con la semántica, en cambio, la persistencia orientada a objetos elimina las diferentes semánticas pero no tiene mecanismos claros para definir dónde se ejecutar las operaciones intensivas de datos.
Analizando solamente la problemática de resolver la diferencia que existe entre un modelo de objetos y la forma en la que los datos de estos objetos son almacenados, las bases de datos orientadas a objetos podría plantearse como la solución natural. El problema surge cuando se analiza la eficiencia, estabilidad y madurez de estas soluciones, es por eso que la solución adecuada para la generalidad de las aplicaciones es la utilización de bases de datos relacionales en conjunto con herramientas de mapeo objeto/relacional.

Capítulo III - Mapeo Objeto/Relacional

Introducción

El mapeo objeto/relacional (también conocido como ORM o OR mapping) es un proceso que consiste en la transformación entre modelos de objetos y relacional y entre los sistemas que soportan estas metodologías.
Para poder realizar esta transformación es necesario tener un amplio conocimiento de las tecnologías orientada a objetos y relacional, así como también de sus similitudes y diferencias.
El modelo relacional está centrado en los datos; el modelo orientado a objetos, en el comportamiento e interacción entre objetos. El problema real surge cuando se intenta unir estos dos modelos en una misma aplicación.

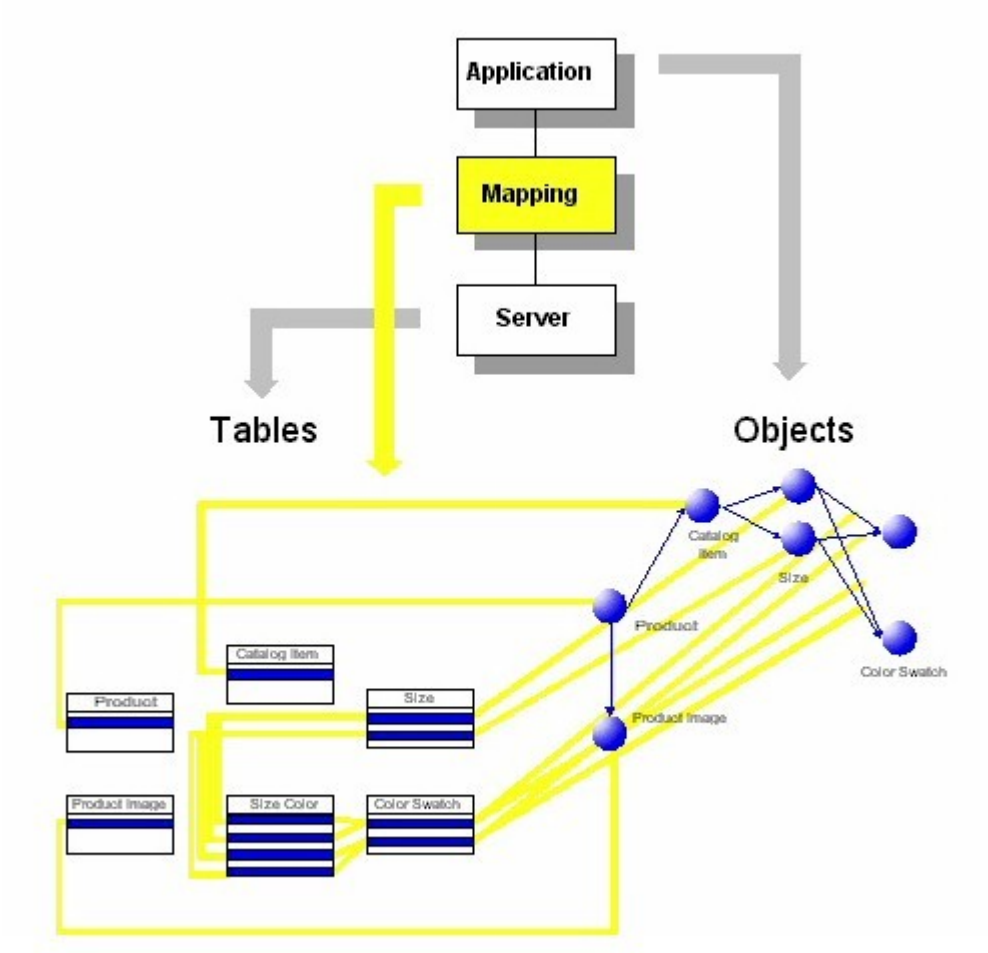


Figura 8: Mapeo objeto/relacional [13]

Utilización de mapeadores

Si se desea implementar una aplicación mediante un lenguaje orientado a objetos como lo es Java y utilizando un RDBMS para almacenar los datos, se podría pensar que la

manera más fácil y rápida de llevarlo a cabo es usando un conjunto de componentes data-aware sobre la interface de la aplicación. Un componente data-aware tiene la capacidad de leer directamente el valor de un campo de una tabla de la base de datos previamente configurado. Sin embargo, si se decide usar estos componentes, se están perdiendo los beneficios del encapsulamiento y a esto se agrega una ardua tarea de mantenimiento más adelante.
Una correcta integración objeto/relacional requiere de una estrategia para realizar el mapeo del modelo de objetos al modelo relacional, con el objetivo de que los objetos del leguaje de programación utilizado, en este caso Java, se conviertan en objetos persistentes en el RDMBS. Un objeto persistente es aquel que puede almacenarse y recuperarse automáticamente en un almacenamiento permanente como es el caso de una base de datos.
La raíz del problema es que los objetos no pueden guardarse ni recuperarse directamente de una base de datos relacional. Los objetos tienen identidad, estado y comportamiento además del dato en si mismo, mientras que las bases de datos relacional solo almacenan los datos. Aun así, los datos solamente suelen representar un problema porque no hay frecuentemente un mapeo directo entre los tipos de datos que se manejan en los lenguajes orientados a objetos y los de un RDBMS. Además mientras los objetos se relacionan y se navegan usando referencias directas, las tablas de un RDBMS se relacionan mediante claves foráneas y claves primarias. Si esto fuera poco, los RDBMS actuales no proveen nada similar a la herencia de objetos, tanto para datos como para comportamiento, como sí lo proveen los lenguajes de programación orientados a objetos.
Finalmente, el objetivo de un modelo relacional es normalizar los datos, es decir, eliminar los datos redundantes en las tablas, mientras que el objetivo del diseño orientado a objetos es modelar los procesos de negocios representando a los objetos del mundo real mediante datos y comportamiento.
El desarrollo de aplicaciones orientadas a objetos robustas requiere construir estrategias de mapeos complejas y eficientes, basándose en un conocimiento sólido de ambos modelos.
Por estas incompatibilidades entre los paradigmas de objetos y relacional es necesario encontrar una forma de mapear el modelo de objetos al relacional. De esta manera los objetos de programación se transforman en persistentes en el RDBMS.

Las herramientas y productos de mapeo objeto/relacional son una alternativa para integrar ambos paradigmas, simplificando la creación de las capas de acceso a los datos, automatizando el acceso a los datos y generando código de acceso a los datos.
El principal objetivo de las herramientas de mapeo objeto/relacional es llevar a cabo el manejo de la persistencia y trabajar a nivel de código con objetos que representan el modelo de dominio en lugar de trabajar con estructuras de datos de la base de datos relacional. Las herramientas establecen un link bidireccional entre los datos de la base de datos relacional y los objetos, basado en una configuración y la ejecución de consultas SQL sobre la base de datos.

Estrategias de mapeo objeto/relacional

Conceptos básicos

Cuando comenzamos con el mapeo de objetos a base de datos relacionales, el punto de inicio son los atributos de cada clase. Un atributo se puede mapear a ninguna o más columnas de una base de datos relacional. No todos los atributos son persistentes,

existen también atributos denominados no persistentes que generalmente son usados para realizar cálculos temporales. Algunos atributos de un objeto son también objetos; esto refleja que la relación entre las dos clases tienen que ser mapeadas y los atributos del segundo objeto también.
La forma mas sencilla de mapeo es un atributo simple a una columna simple y es más directo aun si el atributo y la columna tienen el mismo tipo básico, por ejemplo un atributo String y una columna Char, un atributo Number y una columna Integer.
Como sabemos, las clases implementan tanto comportamiento como datos, mientras que las tablas relacionales solo almacenan datos. Es por esto que además de poder mapear las propiedades de una clase a columnas, también es posible mapear operaciones de una clase a columnas.
Una clase persistente se mapea en su forma mas simple a una tabla, es decir es una relación uno-a-uno. Sin embargo hay muchos casos que no se pueden resolver de forma tan directa.
El mapeo objeto/relacional debe tener ciertas características que aseguren que los conceptos modelados en objetos tengan una traducción correcta al modelo relacional. Este mapeo debe ser idempotente, es decir si CO es un concepto de la programación orientada a objetos, M(x) la función de mapeo y M(CO) = CR (concepto en el modelo relacional), entonces M(M-1(CR)) = CR. Durante esta traducción no puede existir pérdida de información. Una vez realizada la traducción a relacional, la recuperación debe contener la misma información que existía antes de realizar el mapeo.

Información oculta (shadow information)

Información oculta son aquellos datos adicionales a la información del dominio necesarios para que un objeto pueda persistirse. Esto incluye generalmente, información de clave primaria cuando la misma no tiene ningún significado desde el punto de vista del negocio, marcas para el manejo de control de concurrencia como contadores incrementales o timestamps y números de versionado. De esta manera, para que un objeto pueda persistirse, la clase a la que pertenece este objeto necesariamente debe implementar estos atributos ocultos para almacenar los valores correspondientes. La información oculta no necesariamente tiene que ser implementada en los objetos de negocio.[6]

Mapeo de jerarquías de herencia simple

Las bases de datos relacionales no soportan herencia; por este motivo es necesario realizar el mapeo correspondiente entre las estructuras jerárquicas del modelo de objetos y el esquema de base de datos. [6]
Existen diferentes estrategias para mapear herencias a una base de datos relacionales:

- Jerarquía de clases a una única tabla
- Cada clase contracta a su propia tabla
- Cada clase a su propia tabla
- Clases a una estructura de tablas genérica

No todas estas estrategias de mapeo son ideales para todas las situaciones. Veremos al presentar cada caso, cuales son sus ventajas, desventajas y bajo que condiciones conviene usar cada estrategia.
Las tres primeras estrategias pueden combinarse en una misma aplicación. También pueden combinarse en una misma jerarquía compleja.
Para mostrar cada una de las estrategias, usemos la jerarquía de clases que muestra la Figura 9. Esta jerarquía representa distintos tipos de personas. La clase Persona es una clase abstracta, es decir no puede instanciarse. Las clases Cliente, Empleado y Ejecutivo

son clases concretas.

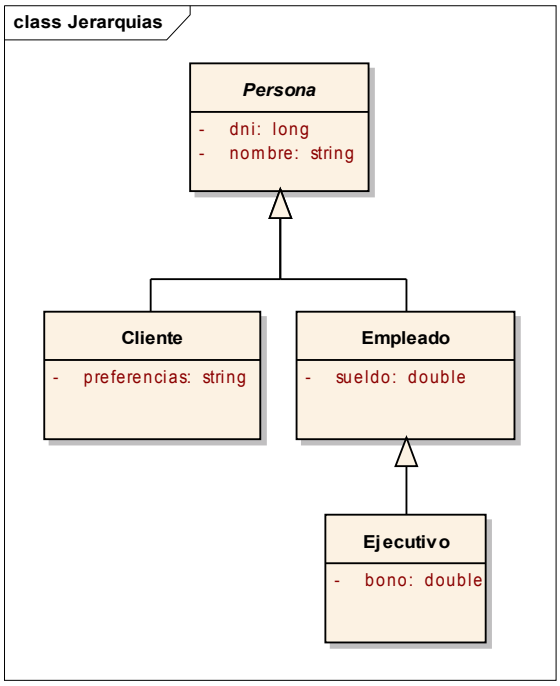


Figura 9: Jerarquía de clases con herencia simple

Jerarquía de clases a una única tabla

Esta estrategia almacena todos los atributos de las clases en una única tabla.

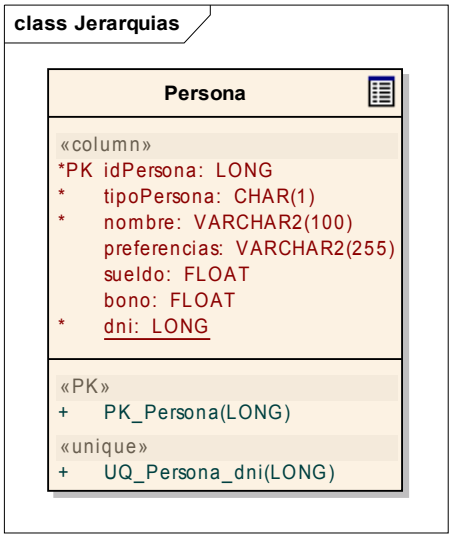


Figura 10: Tabla única Persona

Como se puede observar en la Figura 10, se agregan dos columnas extras a la tabla Persona: idPersona y tipoPersona. La primera es la clave primaria de la tabla. La segunda es un código que identifica si la persona es cliente, empleado o ejecutivo y es necesaria para identificar que tipo de objeto debe instanciarse para cada fila de la tabla.

Ventajas

- Estrategia más simple.

- Si se agregan nuevas subclases a una jerarquía existente, solo deben agregarse nuevas columnas a la tabla para incorporar los atributos particulares de la nueva clase.
- El polimorfismo se mantiene simplemente mediante el tipo determinado para cada fila.
- El acceso a los datos es rápido ya que solo se accede a una única tabla.
- Reportes ad-hoc son fáciles de implementar porque todos los datos están en una única tabla.

Desventajas

- Alto acoplamiento de la jerarquía de clases porque todas las clases están mapeadas en la misma tabla. Un cambio en una clase que implique una modificación en la tabla, puede tener un efecto lateral en las demás clases de la jerarquía.
- Potencial desperdicio de espacio en la base de datos. Una fila representa un objeto de una subclase determinada y las columnas que almacenan datos propios de otra subclase van a tener valores nulos.
- Indicar el tipo de cada fila en la tabla, se dificulta cuando puede existir solapamiento entre los tipos existentes, es decir una misma fila puede ser de dos o mas tipos.
- La tabla puede crecer rápidamente cuando la jerarquía es muy grande.

Cuando usarla

Es una buena alternativa cuando la jerarquía de clases tiene pocos niveles y cuando no existe o es muy poco el solapamiento entre los tipos de la jerarquía.

Cada clase contracta a su propia tabla

Esta estrategia indica que debe crearse una tabla por cada clase concreta de la jerarquía. En cada tabla van a almacenarse los atributos propios de la clase y los atributos heredados de su superclase.

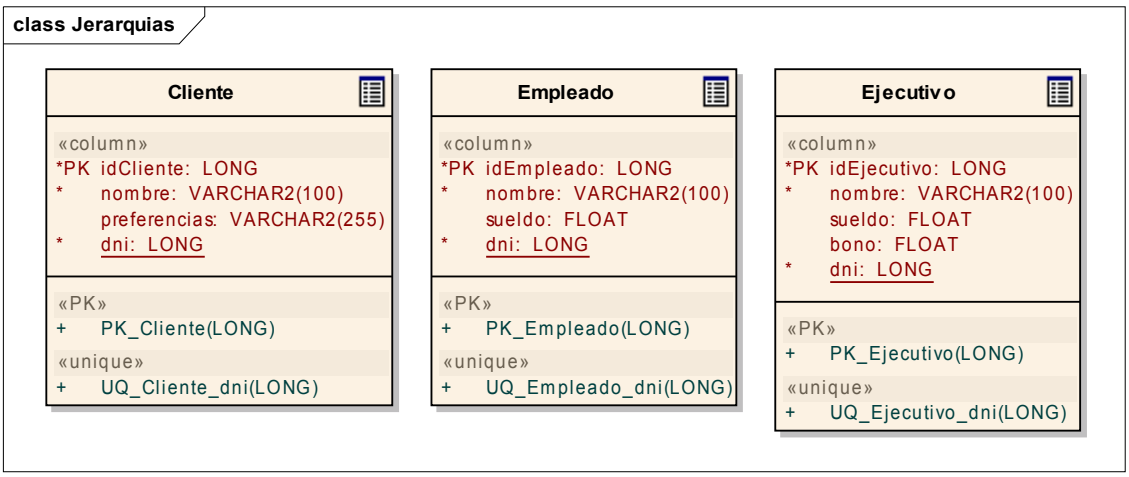


Figura 11: Tabla propia por clase concreta: Cliente, Empleado y Ejecutivo

En este caso cada tabla provee su propia clave primaria, como se observa en la Figura 11. Como cada clase tiene su propia tabla, es fácil instanciar objetos de cada una de ellas. No es posible instanciar objetos de la clase Persona porque esta clase es abstracta.

Ventajas

- Reportes ad-hoc son fáciles de implementar cuando todos los datos necesarios de

una clase están almacenados en una sola tabla.
- Buena performance para acceder a los datos de un único objeto.

Desventajas

- Una modificación en una clase que contiene subclases, implica modificar las tablas en las que se almacenan todas las subclases concretas. Por ejemplo si a la clase Persona se le agrega el atributo mail, es necesario agregar una columna para almacenar el valor de este atributo en las tablas Cliente, Empleado y Ejecutivo.
- Si un objeto cambia de rol (de clase), en necesario sacar el dato de la tabla donde se persistía el objeto y agregarlo en la nueva tabla, asignándole un nuevo ID o reusando algún ID existente.
- Difícil soportar roles múltiples y mantener integridad de datos. Por ejemplo, si una persona fuese cliente y empleado a la vez, ¿dónde debería almacenarse el dni de la persona?

Cuando usarla

Es una buena alternativa cuando es muy poco probable que ocurran cambios de tipos y solapamiento entre los tipos de la jerarquía.

Cada clase a su propia tabla

Con esta estrategia cada clase (abstracta o concreta) se persiste en una tabla. En cada tabla, se almacenan los atributos propios de cada clase y la información oculta que fuese necesaria.

Un punto importante en esta estrategia es la clave primaria. Todas las tablas tienen la misma clave primaria y en el caso de las tablas que mapean a las clases concretas, tienen una clave foránea a la tabla que mapea la clase abstracta para mantener la relación entre ambas tablas.

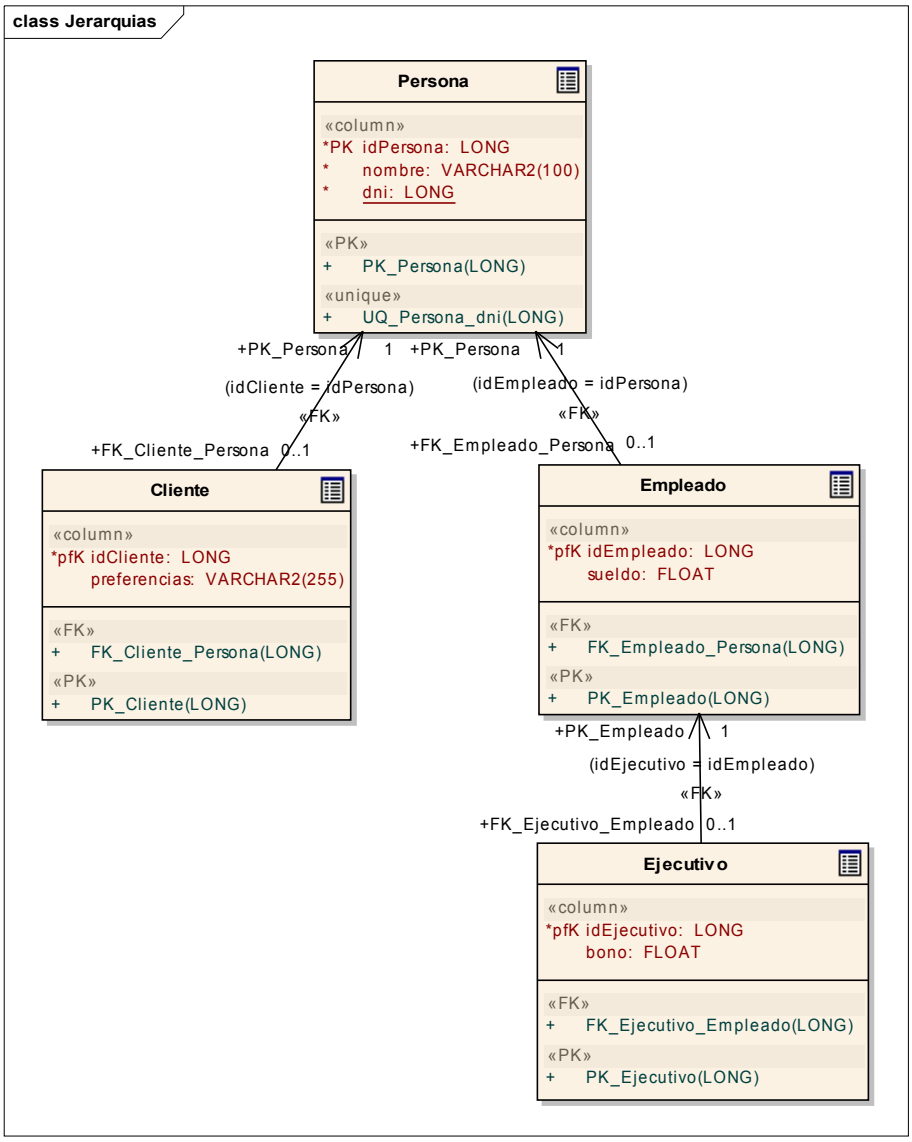


Figura 12: Tabla propia para cada clase: Persona, Cliente, Empleado y Ejecutivo.

En este caso como muestra la Figura 12, la información de la clase Persona se encuentra almacenada en tres tablas, Cliente, Empleado y Ejecutivo, siendo necesario realizar un join entre ellas para obtener toda la información o bien un acceso a cada tabla por separado.
Es común, modificar este mapeo, agregando una columna extra de tipo o boolean en la tabla Persona para indicar de qué tipo de persona es cada tupla almacenada. Si bien esto genera un overhead adicional, muchas veces facilita algunos tipos de consultas. Otra alternativa es usar vistas, en lugar de tener que agregar esta información adicional, ya que su mantenimiento es más sencillo.

Ventajas

- Fácil de entender porque es un mapeo uno a uno.
- Soporta polimorfismo muy bien. Los objetos de una clase se almacenan en las tablas correspondientes para cada tipo.
- Fácil de modificar las superclases e incorporar nuevas subclases, ya que solo se necesita modificar o agregar una sola tabla.
- El tamaño de los datos crece de manera directamente proporcional al incremento del número de objetos.

Desventajas

- Muchas tablas en la base de datos, una por cada clase y eventualmente tablas para mantener las relaciones entre ellas.
- El acceso a los datos es más lento, ya que para leer y escribir datos es necesario acceder a varias tablas.
- Reportes ad-hoc son más complejos, a menos que se usen vistas que joineen las tablas necesarias.

Cuando usarla

Es una buena alternativa cuando es muy común que ocurran cambios de tipos y solapamiento entre los tipos de la jerarquía.

Clases a una estructura de tablas genérica

Esta alternativa de mapeo también es conocida como mapeo orientado a metadata. Esta aproximación no sirve solamente para mapear estructuras jerárquicas, sino para todas las formas de mapeo.

Si bien no nos vamos a detener en la misma, básicamente consiste en tener un esquema de tablas para almacenar la metadata y los datos propiamente dichos.

Un esquema muy sencillo debería consistir en una tabla para almacenar las clases, otra para almacenar los nombres de los atributos, otras para los tipos que pueden tener los atributos (string, long, integer, etc), otra para guardar los valores que van a tener los atributos de una clase y otra para guardar la herencia de clases.

Como indica su nombre, esta alternativa es bien genérica, pero para mapear un solo objeto de una clase es necesario almacenar varias tuplas en todas las tablas involucradas y análogamente la recuperación de la información de un objeto completo, requiere de varios joins entre estas tablas lo cual reduce la performance.

Ventajas

- Funciona bien cuando el acceso a la base de datos están encapsulado por un framework de persistencia robusto.
- Puede extenderse para soportar un amplio rango de mapeos, incluyendo mapeo de relaciones. Es el comienzo de un motor de mapeo de metadata.
- Muy flexible, permitiendo cambiar rápidamente la forma en que se almacenan los objetos, porque lo único que se necesita es actualizar la metadata almacenada en las tablas involucradas.

Desventajas

- La técnica es muy sofisticada y puede resultar difícil de implementar.
- Funciona bien con pequeñas cantidades de datos porque es necesario acceder a varias filas de varias tablas de la base de datos para construir un único objeto.
- Puede ser necesario construir una pequeña aplicación de administración para mantener la metadata.
- Reportes ad-hoc son muy complejos, debido a la necesidad de acceder a varias filas para obtener los datos de un único objeto.

Cuando usarla

Es una buena alternativa para aplicaciones complejas que manejan pequeñas cantidades de datos o para aplicaciones donde el acceso a los datos no es muy común y es posible pre-cargar datos en cachés.

Mapeo de jerarquías de herencias múltiples

Los mapeos de jerarquías de clases vistos hasta ahora, se focalizaron en herencia simple.

La herencia simple es aquella en la que cada subclase hereda directamente de solo de una clase. La herencia múltiple ocurre cuando una subclase tiene dos o más superclases. Por ejemplo, como muestra la Figura 13, el Hidrogeno hereda tanto de NoMetal como de Gas.[6]
La herencia múltiple es un tema bastante cuestionable por los lenguajes orientados a objetos. Existen muchas problemas de dominios donde la herencia múltiple tiene sentido, pero la mayoría de los lenguajes no lo soportan. Por ejemplo Java no lo soporta mientras que C++ y Eiffel, si. Una forma de simular una herencia múltiple es, modelar por un lado una de las jerarquías de herencia simple, crear una Interface con el comportamiento de la superclase que quedó fuera de la jerarquía simple anteriormente mencionada y hacer que todas las subclases de esta jerarquía, además de extender de la segunda superclase, tengan que implementar la interfaz creada.

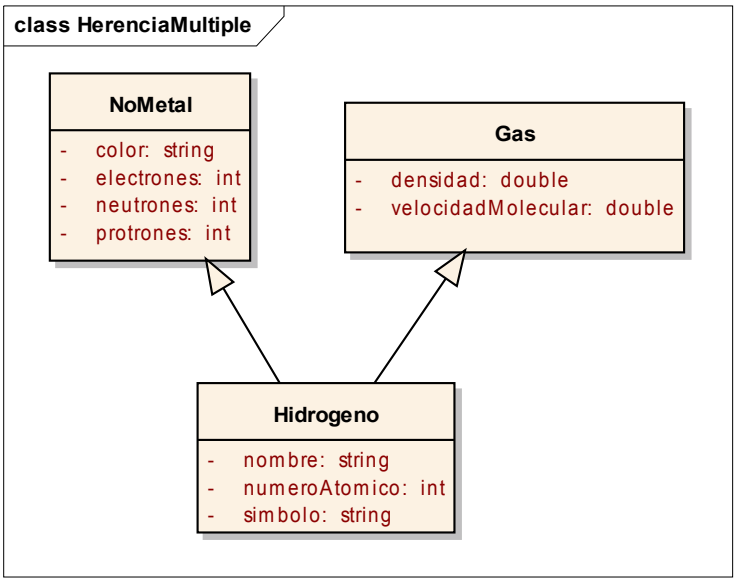


Figura 13: Jerarquía de clases con herencia múltiple

Las tres estrategias mencionadas anteriormente para jerarquías simples de clases, pueden aplicarse para el caso de herencias múltiples. Las ventajas y desventajas también aplican a estos casos.
La Figura 14 muestra como se almacenan los datos de la jerarquía de herencia múltiple si se utiliza la estrategia de almacenar toda la jerarquía en una sola tabla. La Figura 15 muestra el almacenamiento de los datos si se elige la estrategia de almacenar cada clase concreta en una tabla. Finalmente la Figura 16 muestra las tablas donde se almacena la jerarquía si se elige persistir cada clase en su propia tabla.

Jerarquía de clases a una única tabla

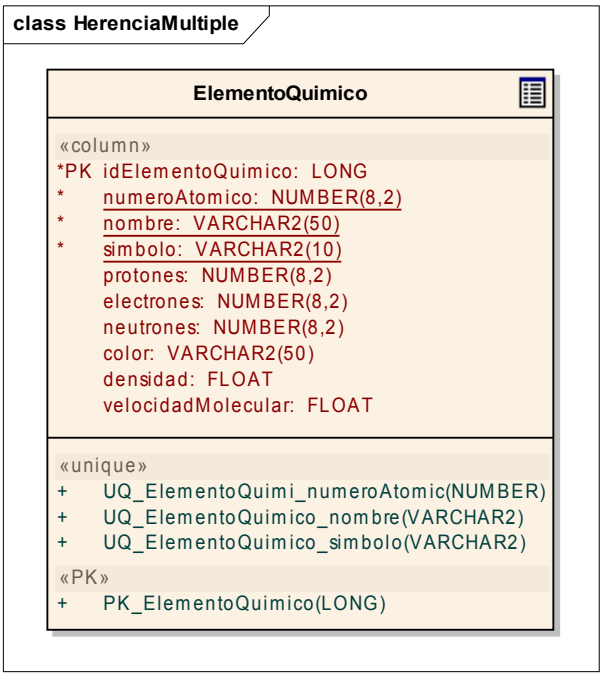


Figura 14: Tabla única ElementoQuimico

Cada clase concreta a su propia tabla

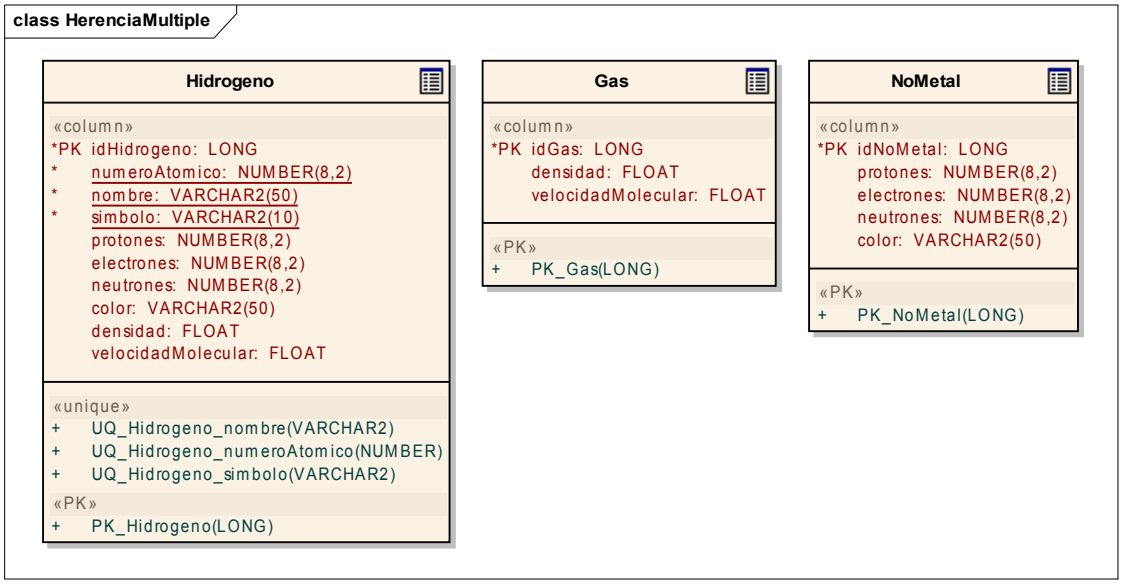


Figura 15: Tabla propia por clase concreta: Hidrogeno, Gas y NoMetal

Cada clase a su propia tabla

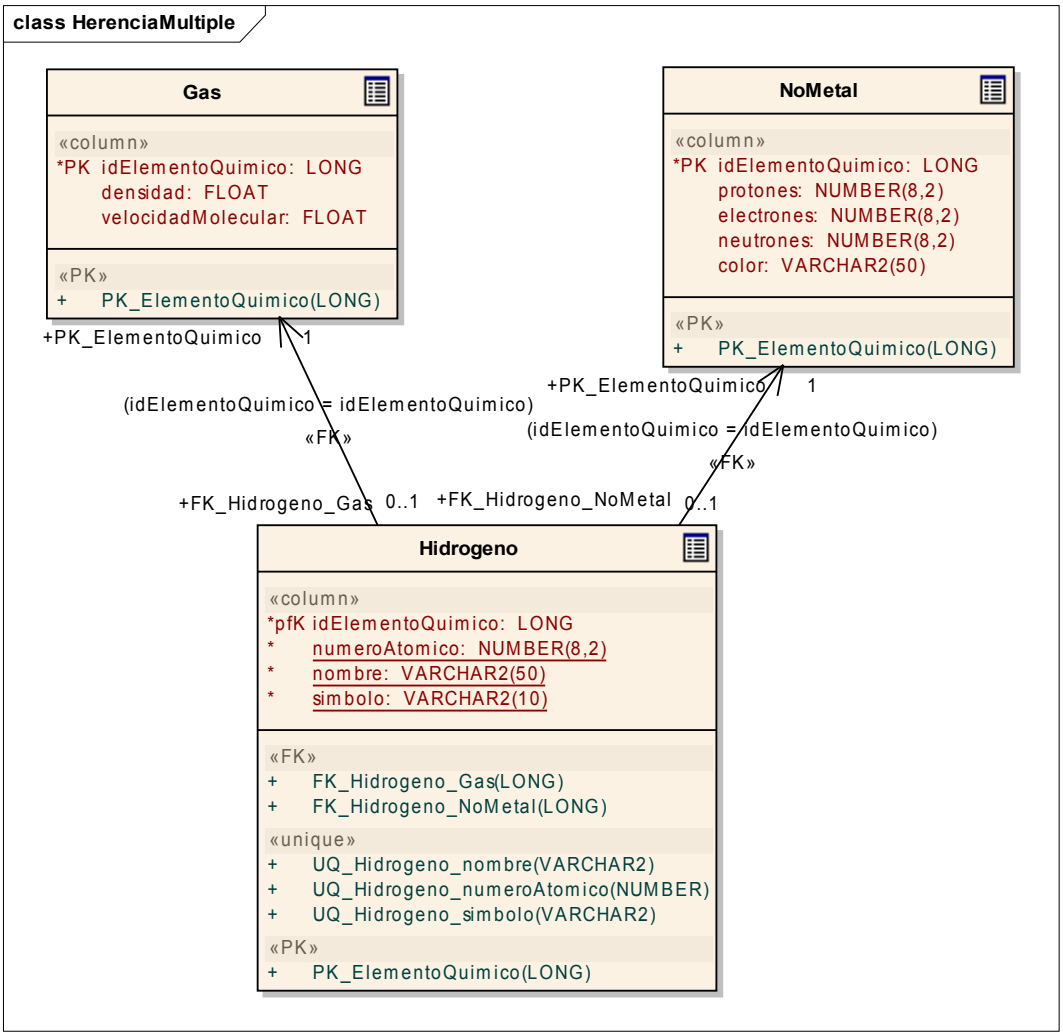


Figura 16: Tabla propia para cada clase: Hidrogeno, Gas y NoMetal

Mapeo de relaciones

Existen tres tipos de relaciones entre objetos que son necesarias mapear: asociación, agregación y composición. Una asociación es una relación débil e independiente entre dos objetos. Una agregación es una relación más fuerte que una asociación pero aún independiente. Una composición es a la vez una relación fuerte y dependiente entre dos objetos. [6]

Tipos de relaciones

Existen dos categorizaciones que se deben tener en cuenta cuando se mapean relaciones de objetos.
La primer categorización se basa en la multiplicidad de la relación y presenta tres tipos:

- Relaciones uno-a-uno: El máximo de cada una de las multiplicidades es uno. En la Figura 17, un empleado posee una y solo una posición y una posición la puede tener uno o ningún empleado.
- Relaciones uno-a-muchos: También conocida como relación muchos-a-uno. En este caso el máximo de una de las multiplicidades es uno y el de la otra es más de uno. En la Figura 17, un empleado trabaja en una sola división y en una división

trabajan uno o más empleados.

- Relaciones muchos-a-muchos: El máximo de ambas multiplicidades es más de uno. En la Figura 17, a un empleado se le asignan una o más tareas y una tarea puede asignarse a ninguno o varios empleados.

La segunda categorización se basa en la direccionabilidad de la relación. Los tipos posibles son:

- Relaciones unidireccionales: Un objeto conoce a los objetos con los que se encuentra relacionado, pero estos objetos no conocen al objeto original. Por ejemplo, en la Figura 17, la relación entre Empleado y Posicion.
- Relaciones bidireccionales: Los objetos que participan de la relación se conocen mutuamente. Por ejemplo, en la Figura 17, la relación entre Empleado y Division. Este tipo de relación es más difícil de implementar que la unidireccional.

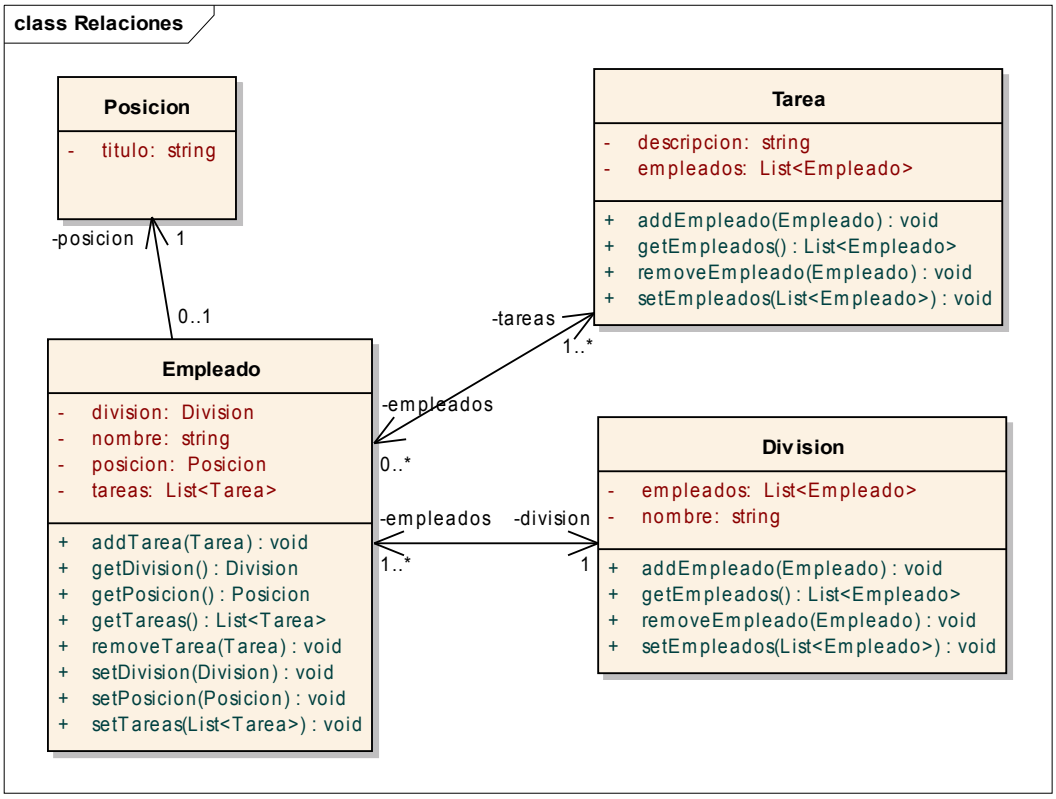


Figura 17: Relaciones uno_a_uno, uno_a_muchos y muchos_a_muchos en un modelo de objetos

Las seis combinaciones se pueden presentar en el mismo modelo de objetos. Sin embargo un aspecto del desajuste de impedancia es que la tecnología relacional no soporta el concepto de relaciones unidireccionales. En las bases de datos relacionales todas las relaciones son bidireccionales. Las relaciones son implementadas vía claves foráneas, la cuales pueden ser joineadas en ambas direcciones.

Relaciones en objetos

Las relaciones entre objetos se implementan mediante una combinación de referencias a objetos y métodos. Cuando la multiplicidad es uno (por ejemplo 0..1 ó 1) la relación se implementa como una referencia a un objeto y los métodos setter y getter. La Figura 17 muestra que el hecho que un empleado trabaja en una única división, se representa mediante el atributo division y los métodos getDivision() y setDivision() de la clase

Empleado.
Cuando la multiplicidad es muchos (por ejemplo n, 0..*, 1..*) la relación se implementa mediante una atributo colección (Array, List, Hash en Java) y métodos para manipular esta colección. Por ejemplo, la clase Division implementa el atributo empleados como List<Empleado> y los métodos getEmpleados(), setEmpleados(), addEmpleado() y removeEmpleado()
Cuando la relación es unidireccional, el código es implementado por el objeto que conoce a los objetos relacionados. Por ejemplo, en la relación entre Empleado y Posicion, la asociación solo es implementada por la clase Empleado. En relaciones bidireccionales, el código es implementado por ambas clases intervinientes en la relación. Por ejemplo, en la relación entre Empleado y Tarea.

Relaciones en bases de datos

Las relaciones en bases de datos relacionales se implementan mediante el uso de claves foráneas. Una clave foránea es un atributo de una tabla que es parte de la clave o clave de otra tabla. Cuando la relación es uno-a-uno, la clave foránea debe ser implementada por una de las tablas. Por ejemplo, en la Figura 18 se observa que la tabla Posicion tiene una columna idEmpleado, la cual es una clave foránea a la tabla Empleado. Otra alternativa para implementar esta relación es mediante la columna idPosicion en la tabla Empleado.

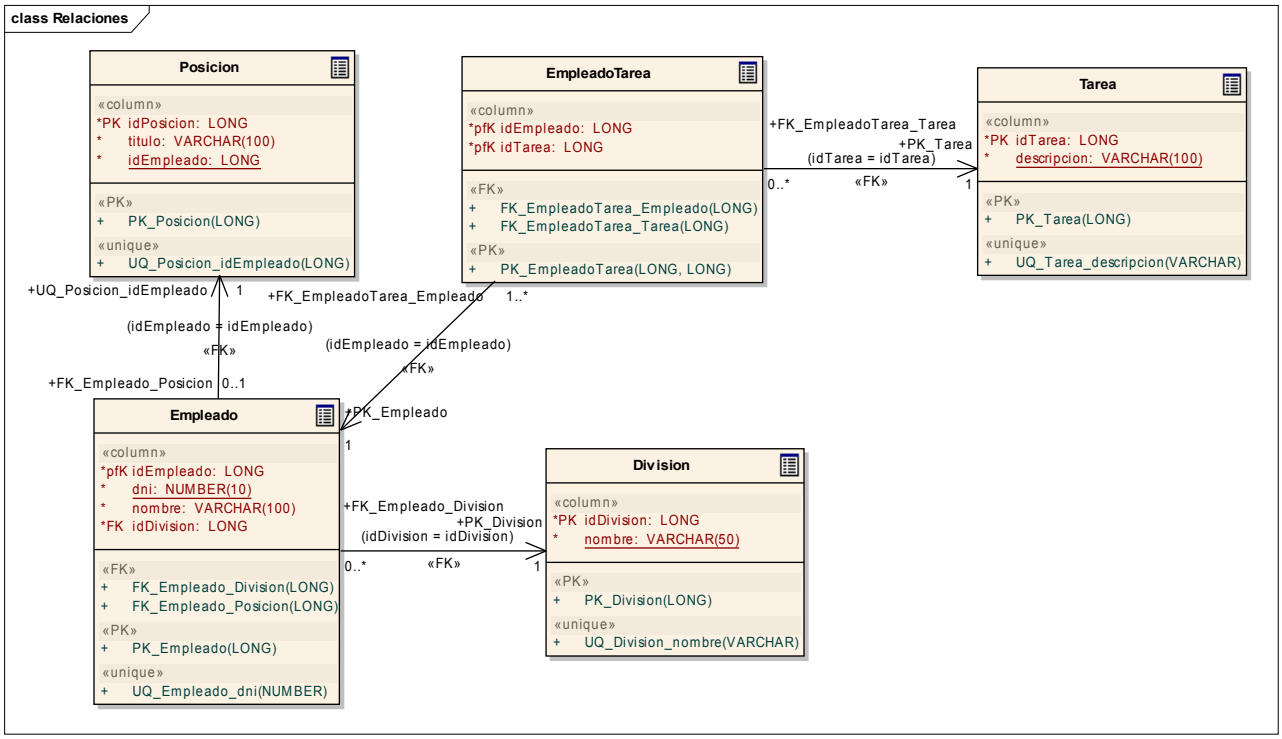


Figura 18: Relaciones uno-a-uno, uno-a-muchos y muchos-a-muchos en un modelo de datos

Cuando la relación es uno-a-muchos, se debe implementar una clave foránea desde la "tabla de los unos" a la "tabla de los muchos". Por ejemplo, la tabla Empleado posee la columna idDivision para implementar la relación trabaja en una División. También se puede implementar esta relación mediante una tabla asociativa, donde la clave primaria de esta tabla asociativa, será la clave primaria de la "tabla de los unos".
Para implementar una relación muchos-a-muchos se debe crear una tabla asociativa cuya clave primaria estará formada por las claves primarias de las tablas que vincula, tal como

muestra la tabla EmpleadoTarea en la Figura 18.
Todas las relaciones en una base de datos relacional son siempre bidireccionales porque las claves foráneas se utilizan para joiner las tablas. Por ejemplo, el código SQL para joinear las tablas Empleado y Posicion vinculadas por la relación uno-a-uno de la Figura 18, es:

```
SELECT * FROM Empleado e, Posicion p WHERE p.idEmpleado = e.idEmpleado;
```

Mientras que el código SQL para joinear las tablas de la misma relación pero si la clave foránea está implementada en la otra tabla, tiene el siguiente aspecto:

```
SELECT * FROM Empleado e, Posicion p WHERE e.idPosicion = p.idPosicion;
```

Estrategias de mapeo de relaciones

La regla general al mapear relaciones dice que se debe mantener la multiplicidad de las relaciones. Una relación de objetos uno-a-uno se debe mapear a una relación uno-a-uno de datos, una uno-a-muchos mapea a una uno-a-muchos y una muchos-a-muchos mapea a una muchos-a-muchos. Esto no impide que una relación de objetos uno-a-uno se mapea en una relación de datos uno-a-muchos o incluso en una muchos-a-muchos. Esto es posible porque la relación uno-a-uno es un subconjunto de la uno-a-muchos y esta es un subconjunto de la muchos-a-muchos.

Relación uno-a-uno

Supongamos la relación uno-a-uno entre Empleado y Posicion. Asumamos que cuando la aplicación lee un objeto Posicion o Empleado a memoria, automáticamente atraviesa la relación posee ellos y lee el objeto correspondiente. La otra opción consiste en que la relación se atraviese manualmente, implementando la aproximación lazy read, donde el objeto relacionado se lee cuando es requerido por la aplicación.
Los objetos de la relación son persistidos en la base de datos en una misma transacción, porque ellos mantienen integridad referencial.
Aunque la dirección de la relación entre Empleado y Posicion en el modelo de objetos está implementada desde Empleado a Posicion, en el esquema de base de datos está implementado de Posicion a Empleado. Inclusive podría implementarse la clave foránea en la otra tabla y no habría diferencia. Si se tiene un potencial requerimiento donde la relación podría cambiar a uno-a-muchos, esto debería motivar la implementación de la clave foránea del lado correcto para poder reflejar el potencial requerimiento.

Relación uno-a-muchos

Supongamos la relación uno-a-muchos trabaja entre Empleado y Division, donde la relación de Empleado a Division debería atravesarse automáticamente (frecuentemente llamada lectura en cascada), pero no en la otra dirección. También son posibles el guardado y el borrado en cascada.
Cuando un empleado es cargado en memoria, la relación se atraviesa automáticamente para cargar la división en la que trabaja. Si varios empleados trabajan en la misma división, se quiere referenciar en memoria al mismo objeto división y no tener varias copias del mismo. Una estrategia para implementar este requerimiento es tener una caché que garantice que solo una copia de cada objeto se encuentre en memoria o que la clase Division implemente su propia colección de instancias en memoria.
El guardado de la relaciones uno-a-muchos funciona de la misma manera que lo hacen

las relaciones uno-a-uno. Cuando los objetos son guardados, también se guardan los valores de las claves primaria y foráneas, por lo tanto la relación se guarda automáticamente.

Relación muchos-a-muchos

Para implementar relaciones muchos-a-muchos es necesaria una tabla asociativa, es decir una entidad de datos cuyo único propósito es mantener las relaciones entre dos o más tablas relacionales. Por ejemplo en la Figura 17 vemos una relación muchos-a-muchos entre Empleado y Tarea y en la Figura 18, la tabla asociativa EmpleadoTarea que implementa la relación entre las tablas Empleado y Tarea, donde idEmpleado y idTarea forman la clave primaria de la tabla asociativa.
Supongamos que un objeto empleado está cargado en memoria y que necesitamos la lista de todas las tareas asignadas. La aplicación necesitará crear un SQL select joineando EmpleadoTarea con Tarea para traer las tareas asociadas al idEmpleado del empleado en cuestión. Luego deberá generar los objetos Tarea a partir de los registros obtenidos. Parte de esta tarea consiste en ver si cada objeto Tarea está en memoria, si lo está es posible actualizar sus datos teniendo en cuenta el manejo de concurrencia. Cuando la operación Empleado.getTareas() en invocada, la aplicación retorna la lista de objetos Tarea.
Un proceso similar ocurre al cargar en memoria los empleados involucrados en una tarea.
En las relaciones muchos-a-muchos, dos clases son mapeadas a tres tablas, lo cual requerirá trabajo extra para obtener el resultado correcto.

Mapeo de colecciones ordenadas

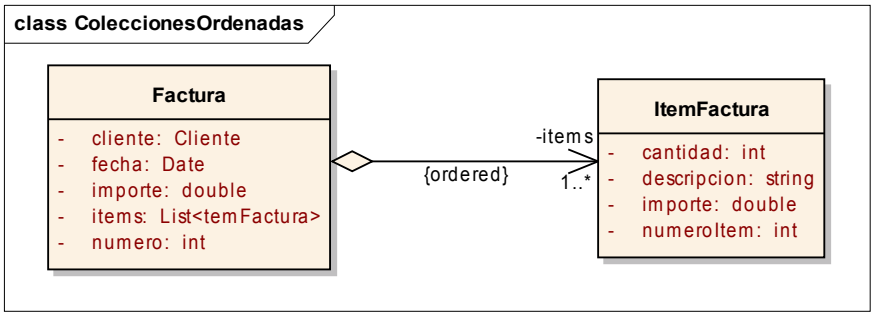


Figura 19: Colección ordenada en modelo de clases

La Figura 19 muestra una agregación entre las clases Factura e ItemFactura, donde una particularidad que se observa es la constraint {ordered} sobre la relación. Esto representa que los ítems de la factura deben aparecer en orden. Para mapearlo a una base de datos relacional se necesita agregar una columna adicional donde guardar esta información. Por ejemplo, en la Figura 20 se observa la columna itemSecuencia.

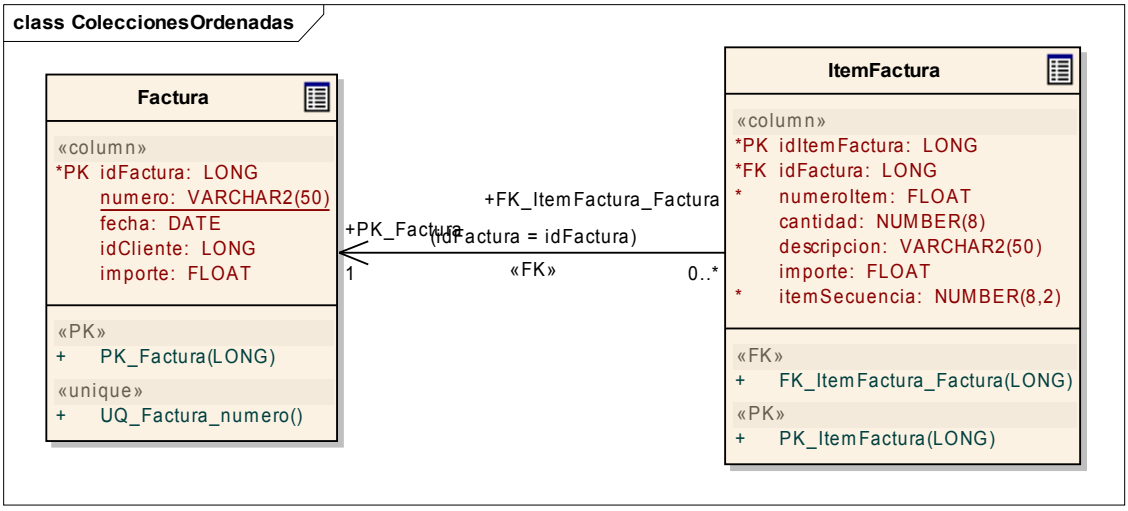


Figura 20: Colección ordenada en modelo de datos

Algunas consideraciones a tener en cuenta:

- Lectura de los datos en el orden correcto. El tipo del atributo que implementa la relación debe ser una colección que permita ordenamiento secuencial. Cuando se lleva a memoria una factura con sus ítems, estos últimos se deben obtener en el orden correspondiente.
- No incluir el número de secuencia en la clave. Supongamos que tenemos una factura con 5 ítems en memoria, a la cual se le agrega un nuevo ítem entre el segundo y el tercero y luego se persiste este cambio. En este caso, es necesario renumerar los números de secuencia de los ítems y grabar este cambio en todos los ítems aunque no hayan tenido otro cambio. Si el número de secuencia es parte de la clave de la tabla ItemFactura, tendremos problemas con las tablas que referencien a un ítem de factura mediante la clave foránea que incluye itemSecuencia. Una mejor aproximación es usar idItemFactura como clave primaria.
- ¿Actualizar los números de secuencia luego de borrar un ítem? Si borramos el quinto ítem de una lista de seis ítems, se puede actualizar los números de secuencia o dejarlos así. Los números de secuencia pueden trabajar con “agujeros” en el medio, pero no puede utilizarse como indicadores de posición dentro de la colección.
- Considerar números de secuencia con saltos (gaps) mayores a uno. En vez de asignar números de secuencia consecutivos (1, 2, 3,…) a los ítems, asignar números tales como 10, 20, 30,…Con esta estrategia se puede evitar la actualización de los números de secuencia de los ítems ya existentes cuando se agrega un nuevo ítem. Por ejemplo, si se quiere agregar un ítem en segundo lugar, se le puede asignar el número de secuencia 15. Puede ocurrir luego de varias inserciones que se necesita insertar un ítem entre el 17 y 18. En este caso se deberá reordenar los números de secuencia. Una alternativa para evitar esto es usar gaps mas grandes (50 ,100, 150,…), pero este problema no desaparecerá completamente.

Mapeo de relaciones recursivas

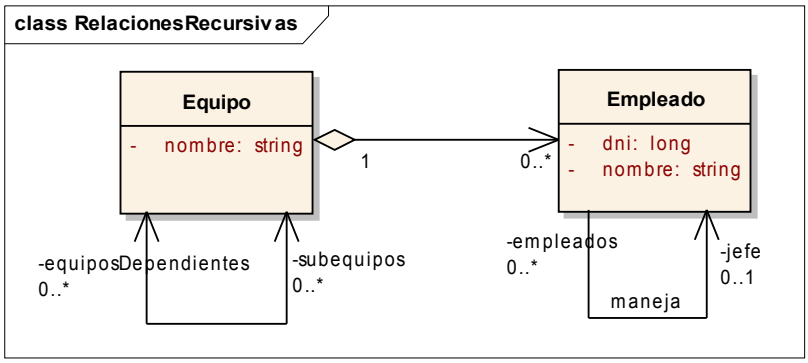


Figura 21: Relación recursiva en un modelo de clases

Una relación recursiva es aquella donde la misma entidad se encuentra en ambos extremos de la relación. Por ejemplo en la Figura 21, la relación maneja es recursiva y representa que un empleado coordina a varios empleados. La relación de la clase Equipo también es recursiva, donde un equipo puede formar parte de uno o más equipos.
La relación recursiva muchos-a-muchos se mapea a una tabla asociativa RelacionEquipo, de la misma forma que se mapea una relación muchos-a-muchos normal. La única diferencia es que las columnas de la tabla asociativa son claves foráneas de la misma tabla.[6]
Análogamente, la relación recursiva uno-a-muchos maneja se mapea igual que una relación uno-a-muchos normal. En el ejemplo, la columna idJefe hace referencia a un registro de la tabla Empleado.

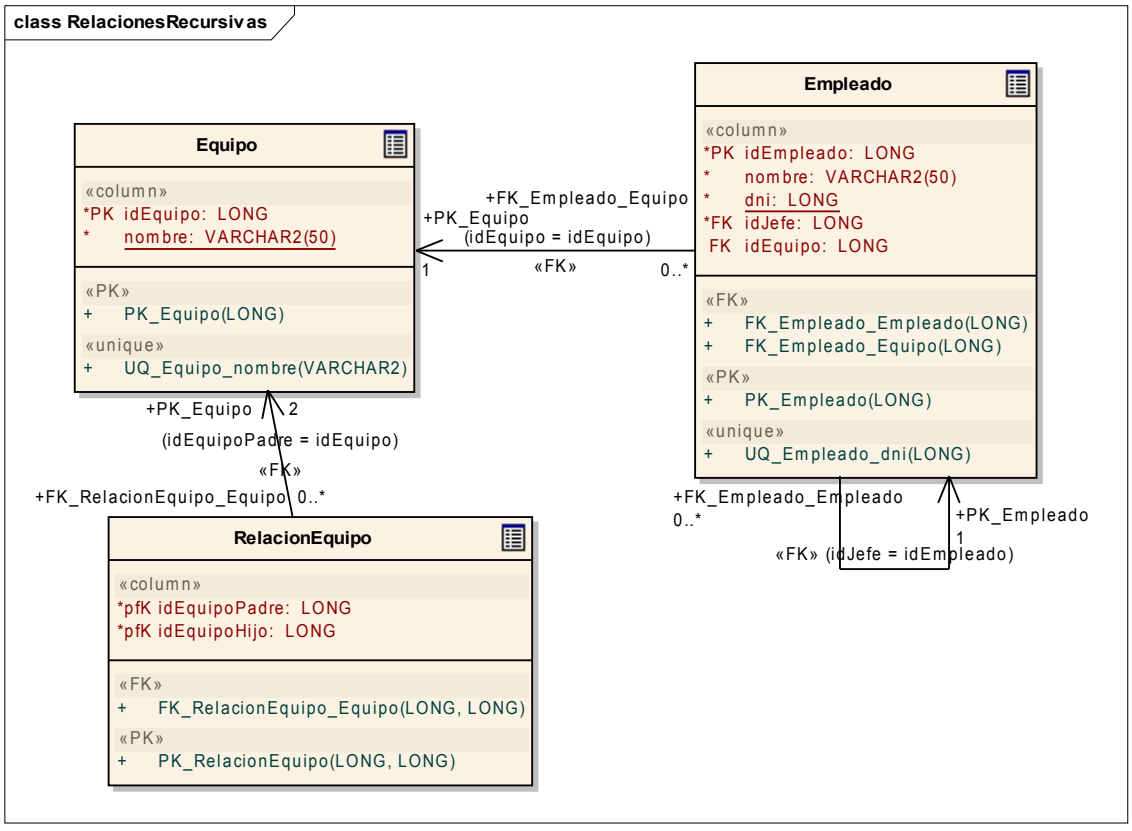


Figura 22: Relación recursiva en un modelo de datos

Conclusiones

Todas estas estrategias de mapeo objeto/relacional son, sin duda, problemas recurrentes en muchos desarrollos de aplicaciones enterprise. En este capítulo se analizaron cada una de las estrategias de mapeo de manera informal, sin embargo existe un conjunto bien definido de patrones de diseño que tratan formalmente cada uno de los aspectos del mapeo analizadas anteriormente. Estos patrones básicamente se relacionan con la comunicación entre la lógica del dominio y los repositorios de datos.[14] A continuación se listan los patrones más importantes relacionados con el mapeo objeto/relacional.

Patrón	Descripción
Class Table Inheritance	Representa una jerarquía de herencia de clases mediante una tabla por cada clase.
Concrete Table Inheritance	Representa una jerarquía de herencia de clases mediante una tabla por cada clase concreta. de la jerarquía.
Single Table Inheritance	Representa una jerarquía de herencia de clases mediante una única tabla que posee columnas para todos los campos de las diferentes clases.
Inheritance Mappers	Una estructura para organizar los mapeadores de base de datos que manejan jerarquías de herencia.
Identity Field	Graba un campo ID de la base de datos en el objeto para mantener la relación unívoca entre los objeto en memoria y su correspondiente fila en la base de datos.
Foreign Key Mapping	Mapea una asociación entre objetos cuando existe una referencia de clave foránea entre tablas.
Association Table Mapping	Persiste una asociación como una tabla con claves foráneas a las tablas que almacenan los objetos relacionados por la asociación.
Embedded value	Mapea un objeto a varios campos de la tabla de otro objeto.
Metadata Mapping	Guarda la información del mapeo objeto – relacional como meta información.
Serialized LOB	Guarda un grafo de objetos como su serialización en un objeto grande único (LOB) y almacena el LOB en un campo de la base de datos.
Identity Map	Asegura que cada objeto se cargue una sola vez, manteniendo cada objeto cargado en un mapa. Los objetos son buscados en el mapa cuando se hace referencia a ellos.
Lazy Load	Un objeto que no contiene todos los datos solicitados pero sabe como obtenerlos. Se lleva el objeto a memoria solo cuando este es requerido.

Tabla 2: Patrones de mapeo objeto/relacional

Cada estrategia de mapeo presenta varias soluciones posibles, cada una con sus ventajas y desventajas. Se debe analizar en cada caso cual de estas soluciones es la que mejor de adecúa a los requerimientos y necesidades de la aplicación y tambien cual representa menor impacto en cuanto a su implementación.

Capítulo IV – ORM - Estado del arte

Productos de mapeo objeto/relacional

Hoy en día existen una gran variedad de frameworks y productos de mapeo objeto/relacional, tanto open source como comerciales, para distintos lenguajes e incluso para diferentes plataformas. La plataforma J2EE tiene una oferta muy rica y variada de mapeadores. La plataforma .NET tiene menos frameworks O/R-M disponibles.
Elegir cual será el producto de mapeo que vamos a utilizar en una aplicación no es una tarea sencilla. Se deben evaluar detalladamente todas las características, analizando cuales se consideran críticas y cuales no, para poder tomar la decisión correcta.

Mapeadores objeto/relacional open source

- Hibernate es la solución de mapeo O/R para Java más popular. Hibernate es un servicio de persistencia y consultas objeto/relacional muy poderoso y de alta performance. Hibernate permite desarrollar objetos persistentes siguiendo un idioma común, Java; incluyendo asociaciones, herencia, polimorfismo, composición y colecciones. Ofrece un lenguaje de consultas denominado HQL (Hibernate Query Lenguage), diseñado como una extensión orientada a objetos "mínima" de SQL. HQL es un puente entre los mundos relacional y de objetos. Hibernate además permite expresar consultas usando SQL nativo o Criteria basado en Java.[15]
- ObJectRelationalBridge (OJB) es una herramienta de mapeo objeto/relacional que permite persistencia transparente para objetos Java contra una base de datos relacional. OJB fue diseñado para amplio rango de aplicaciones, desde sistemas embebidos a aplicaciones con cliente rico, y hasta arquitecturas J2EE multi-capas. OJB es parte del proyecto Jakarta. OJB provee una API ODMG 3.0 y también una API JDO. OJB puede utilizarse con JSPs, Servlets y SessionBeans. OJB provee sopoerte especial para Bean Managed EntityBeans (BMP).
- iBATIS SQL Maps provee formas muy simples y flexibles para mover datos entre los objetos Java y una base de datos relacional. Usa la potencia del SQL sin una línea de código JDBC. El framework SQL Maps ayuda a reducir significativamente la cantitad de código Java que normalmente se necesita para acceder a una base de datos relacional. Este Framework mapea JavaBeans a sentencias SQL utilizando simples descriptores XML. Su simplicidad la principal ventaja de SQL Maps sobre otros frameworks y herramientas de mapeo objeto/relacional. Para usar SQL Maps solo se necesita estar familiarizado con XML, SQL y JavaBeans.
- Cayenne es un poderoso y completo framework de mapeo objeto/relacional para Java. Es open source y completamente gratis. Está certificado bajo la licencia Apache. Una de las principales distinciones de Cayenne es que provee herramientas GUI multiplataforma.
- TriActive JDO (TJDO) es una implementación open source de la especificación JDO de Sun (JSR 12), diseñada para soportar persistencia transparente usando cualquier base de datos JDBC-compatible.
- Jaxor es una herramienta para generar la capa de mapeo objeto/relacional la cual toma información referente a las entidades relacionales que van a ser mapeadas, definida en XML y genera clases, intefaces y objetos finder los cuales pueden ser

utilizados por cualquier aplicación Java. La generación de código es manejada por templates Velocity en lugar de mecanismos fijos de la herramienta. Esta flexibilidad permite que se ajuste y modifique fácilmente el formato así como también el código generado.

- JDBM es un motor de persistencia para Java. Se puede utilizar para almacenar un mix de objetos y BLOBs y todas las actualizaciones se realizan en forma transaccional y de manera segura. JDBM también permite estructuras de datos escalables, por ejemplo HTree y B+Tree, para soportar persistencia de grandes colecciones de objetos.
- pBeans es una capa de mapeo objeto/relacional. Está diseñada con el objetivo de ser simple de utilizar y complemente automatizada. El objetivo es ahorrar tiempo y esfuerzo para centrarse simplemente en escribir clases Java, y no preocuparse por el mantenimiento de scripts SQL, esquemas basados en XML y de la generación de código. pBeans permite persistir JavaBeans con muy poca asistencia del desarrollador.
- SimpleORM es un sistema de mapeo objeto/relacional con funcionalidad completa para Java. Provee una implementación simple pero efectiva de mapeos objeto/relacional por encima de JDBC a bajo costo y con bajo overhead. No es necesario configurar ningún archivo XML. SimpleORM es 100% Java puro, tiene mínimo uso de reflection, no posee pre procesamiento y ningún byte code de post procesamiento. SimpleORM permite accesos a JDBC directo y a base de datos no-Java sin comprometer la integridad de la base de datos. Lockeo optimista es utilizado para llevar a cabo el lockeo entre transacciones.
- JDBCPersistence es un framework de persistencia objeto/relacional. Se diferencia de otros productos similares en que genera el bytecode necesario para mapear una clase a una tabla. Es rápido de cargar, soporta CLOB y BLOBs y brinda una API compacta
- Persistent Applications Toolkit (PAT) simplifica el desarrollo de la capa de persistencia para las aplicaciones empresariales. Provee un entorno orientado a objetos para persistir los objetos: POJOs. PAT proporciona una capa de datos casi transparente. Emplea un conjunto de técnicas avanzadas para lograrlo, como son OO, AOP, Java, Ant, JUnit, Log4j, @@annotations, etc. También coopera en aplicaciones web: Struts y otros framwors web, Tomcat, JBoss, etc.

Mapeadores objeto/relacional comerciales

- JDO Genie (de Hemisphere Technologies) es una implementación de JDO de alta performance que permite trabajar con bases de datos relacionales open source y comerciales. JDO Genie tiene opciones de mapeo objeto/relacional flexibles e incluye una GUI Workbench. JDO Genie soporta JBoss, WebLogic, WebSphere, Tomcat, aplicaciones 2-capas y aplicaciones distribuidas.[15]
- LiDO (de LIBeLIS) es una implementación de JDO que a diferencia de las demás herramientas de mapeo O/R, LiDO no solo provee acceso a bases de datos relacionales sino también a otras fuentes de datos, tales como XML, Mainframe, ODBMS y archivos planos.
- JDO Toolkit (de MVCSoft) JDO es la forma estándar de persistir objetos en Java. JDO Toolkit permite persistir de manera transparente los objetos de la aplicación en la base de datos, sin tener que romper el modelo orientado a objetos con código procedural. Brinda soporte para todas las feactures de JDO y feactures adicionales. Incluye GUI Workbench, soporte para Ant y XDoclet.
- IntelliBO (de Signsoft) es una implementación del estándar Java Data Objects

(JDO). Con intelliBO se puede manejar el acceso a los datos de manera simple, flexible y performance. En otras palabras, intelliBO permite un acceso inteligente a los datos, de donde deriva su nombre intelliBO - intelligent Business Objects. Puede ser utilizado en aplicaciones de escritorio sencillas, en aplicaciones web o aplicaciones enterprise complejas. Permite persistir datos en bases de datos relacionales, bases de datos orientadas a objetos o XML.

- Kodo JDO (de SolarMetric) es una implementación robusta, de alta performance y con funcionalidad completa, de la especificación JDO para lograr persistencia transparente. A diferencia de muchas herramientas de mapeo objeto/relacional propietarias, Kodo JDO provee acceso a las bases de datos relacionales mediante el estandar JDO, permitiendo a los desarrolladores Java utilizar la base de datos relacional existente sin necesidad de conocer SQL o ser un experto en diseño de bases de datos relacionales. Puede ser utilizado con esquemas de bases de datos existentes o puede generar automáticamente un nuevo esquema.
- POET, ahora denominado FastObjects (de Poet Software) proporciona una tecnología de persistencia orientada a objetos para aplicaciones Java y C++. Permite al desarrollador concentrarse en el modelo de la aplicación, en lugar de preocuparse en escribir el código de mapeo. El post procesador determina el esquema de base de datos a partir de un archivo de configuración y las definiciones de clases. Crea la base de datos y expande bytecodes Java para agregar la funcionalidad de bases de datos necesaria. FastObjects trabaja sobre las especificaciones estandares JDO 1.0 y ODMG 3.0, provee alta performance aun con modelos de objetos complejos e integración transparente con bases de datos relacionales.
- TopLink (por Oracle) es uno de los frameworks de mapeo O/R mas respetados. Permite almacenar objetos Java en bases de datos relacionales o convertir objetos Java en documentos XML. Tiene buenos manuales y herramientas para sincronizar con los esquemas de bases de datos y una buena interfaz para el desarrollador. [9]
- TopLink Essentials es la versión open source de TopLink. Provee una implementación de Java Persistence API (JPA), pero es una versión limitada del producto propietario.

Mapeadores para .NET

- Ojb.Net es una herramienta de persistencia objeto/relacional para la plataforma .Net. Está basado en OJB de la plataforma Java. Permite a las aplicaciones, recuperar y almacenar transparentemente, objetos .NET usando bases de datos relacionales. Es un proyecto open source. [16]
- NHibernate es el framework Hibernate pero para la plataforma .NET. Maneja la persistencia de objetos planos .NET hacia y desde una base de datos relacional. Dada una descripción XML de sus entidades y relaciones, NHibernate genera automáticamente SQL para recuperar y almacenar los objetos. Opcionalmente, es posible describir la metadata de mapeo con atributos en el código fuente basándose en librerías de persistencia de objetos a bases de datos relacional. Es un proyecto open source.
- iBATIS.NET Mapeador de datos que ayuda a crear la capa de persistencia en aplicaciones .NET. Es un framework open source.
- NEO es un framework para desarrolladores .NET que permite escribir aplicaciones enterprise mediante un modelo de dominios basado en objetos. Esta es una de las diferencias más importantes con otras soluciones ORM. En Hibernate y TopLink, los desarrolladores de aplicaciones, crean objetos planos para representar el

dominio y luego la capa de persistencia toma estos objetos y opera sobre ellos para persistir su estado. En Neo, el modelo de dominio se describe de manera abstracta (en un archivo XML) y Neo genera las clases y todo SQL necesario para manejar la persistencia del los objetos. Neo está basado a data set ADO.NET y el modelo de dominio generado es independiente del lugar de almacenamiento. Es un framework open source.

- Gentle.NET es un framework de persistencia de objetos independiente del RDBMS, diseñado específicamente para .NET. Utiliza programación basada en atributos para generar la metadata, posee independencia del motor de bases de datos, generación automática de SQL, control de concurrencia, ayudas en la creación de DataViews, construcción de objetos automática y clases para la gestión de relaciones 1:n y n:m. Es un framework open source.
- ADO.NET es un conjunto de librerías base que permiten la conexión a datos. Es la forma más básica y a más bajo nivel de como acceder a datos, requiere de esfuerzo para hacerla funcionar en una aplicación.
- SubSonic es un mapeador objeto/relacional open source para tecnología .Net que simplifica y reduce al máximo el trabajo de desarrollo de la capa de datos. A partir de un esquema de base de datos, genera código fuente que conforma lo que se conoce como Data Access Layer (DAL). Data Access Layer representa una capa en la aplicación que contiene todo el comportamiento y lógica para acceder a los datos a partir de los objetos.
- Castle ActiveRecord es una implementación del patrón ActiveRecord para .NET. El patrón ActiveRecord consiste en, un objeto que contiene los datos que representan a un renglón (o registro) de nuestra tabla o vista, además de encapsular la lógica necesaria para acceder a la base de datos. De esta forma el acceso a datos se presenta de manera uniforme a través de la aplicación. Castle ActiveRecord es una capa implementada encima de NHibernate, pero el mapeo basado en atributos libera a los desarrolladores de escribir archivos XML para mapear objetos a datos relacionales, lo cual es necesario cuando se usa NHibernate directamente.

Historia de los mapeadores objeto/relacional

La siguiente es una breve reseña histórica de los mapeadores objeto/relacional relacionados con la plataforma J2EE [17]:

- 1994: Sale a la venta el producto TopLink para Smalltalk
- 1996: Una versión Java de TopLink se agrega a la línea de productos, denominada TopLink para Java.
- 1997: 3M es la primera compañía que oficialmente compra TopLink para Java.
- 1998: IBM lanza su propia herramienta para mapeo objeto/relacional para Smalltalk, llamada ObjectExtender.
- 1999: Se crea TopLink para WebLogic basado en POJOs.
- 1999: Comienza a desarrollarse la especificación Enterprise JavaBeans 2.0.
- 2000: Comienza a desarrollarse la especificación Java Data Objects.
- 2001: Release de Enterprise JavaBeans 2.0 en septiembre de 2001.
- 2001: Se crea Enterprise JavaBeans 2.1.
- 2001: En noviembre comienza el proyecto SourceForge. Un proyecto open sorce destinado a construir un sistema de mapeo objeto/relacional para Java .
- 2002: Release final de JDO 1.0
- 2002: Release de Hibernate 1.0

- 2003: Release final de la especificación Enterprise JavaBeans 2.1 [18]
- 2003: En junio se lanza el release de Hibernate 2.0
- 2003: En agosto comienza a desarrollarse JDO 2.0
- 2003: En octubre, desarrolladores de Hibernate son contratados por JBoss.
- 2006. Release de JDO 2.0
- 2005: En febrero se lanza el release de Hibernate 3.0
- 2006: Release de EJB 3.0
- 2006: En mayo se lanza el release de Hibernate 3.1 y en octubre Hibernate 3.2.
- 2008: En Febrero finaliza JDO 2.1, desarrollado por el proyecto Apache JDO
- 2008: En agosto se lanza el release de Hibernate 3.3
- 2008: En octubre se lanza el release de JDO 2.2
- 2009: Release de EJB 3.1
- 2010: En abril se lanza el release de JDO 3.0
- 2010: Release de Hibernate 3.5

Tipos de mapeadores objeto/relacional

Dependiendo de las diferentes vistas sobre los datos que tenga la aplicación, es factible categorizar los mapeadores en tres tipos [19]:

Aproximación de Tuplas o Tablas

Esta aproximación no se basa en ningún modelo abstracto, solo en un conjunto de tablas. La aplicación simplemente lee registros desde el RDBMS a “record sets” y luego trabaja con estos record sets. Los datos no se leen a objetos.
Esta estrategia no es recomendada para el desarrollo de sistemas grandes porque la lógica de negocio esta fuertemente acoplada con el formato de los datos en la base de datos. Por ejemplo cuando un atributo se mueve (por algún refactoring) de una tabla a otra, todos los lugares del código fuente de la aplicación donde se usa este atributo, deben cambiar.

Aproximación de Entidad (Chen / Yourdon)

Esta aproximación se basa en el modelo relacional. Como en la aproximación anterior, la aplicación lee datos a record sets y luego las entidades de negocio se crean con los datos de estos record sets. Las entidades de negocio tienen una estructura similar a la de tablas para las bases de datos relacionales. Por ejemplo DataSets o DataTables de ADO.Net. Un DataTables es una estructura tabular que puede tener restricciones de claves primarias (PK) y foráneas (FK). Un DataSet es un conjunto de DataTables, entre los cuales pueden definirse relaciones, con posibilidades de update o delete en cascada. Como se observa, la funcionalidad es similar a la que provee un RDBMS.
Un inconveniente de esta aproximación es que las entidades de negocio no tienen métodos, es decir comportamiento. Solo pueden encontrarse métodos simples asociados al procesamiento de los datos. Por ejemplo métodos como checkConstraints().
La lógica del negocio esta ubicada en las llamadas clases manejadoras o administradoras. Las entidades de negocio son los argumentos de los métodos de dichas clases administradoras.
De esta manera las entidades de negocio pasan a ser más pasivos y los objetos que modelan los procesos de negocio se convierten en activos. Se preserva lo relacional,

pensando a nivel de objetos.
Esta aproximación es ampliamente utilizada en sistemas de gran magnitud desde fines de los 70'.

Aproximación de Modelo de Dominio (Fowler / Evans)

Esta aproximación es similar a la de entidad. La diferencia está en las entidades de negocio. En este caso las entidades de negocio son objetos reales con datos y comportamiento. Los objetos no solamente contienen los datos de las tablas de bases de datos. También pueden utilizarse la herencia, polimorfismo y otros conceptos de la orientación a objetos. La lógica de negocios esta ubicada en los métodos de las entidades de negocio. No se necesitan clases manejadores como en el caso anterior.

Elección de aproximación

Es difícil decir que la aproximación de Entidad es mala, ya que hace 25 años los sistemas de software la usan, aunque también una gran cantidad de sistemas de software hoy están implementados en Java y utilizan la aproximación de modelo de dominios, la cual funciona muy bien.
La aproximación a elegir depende fundamentalmente, de cómo se ven y como se quiere trabajar con los datos en el sistema. Por ejemplo, "Un cliente obtiene la tarjeta gold cuando el total de sus compras ha superado el valor $25.000 en un mes. ¿Dónde se pone esa lógica? ¿En qué clases?". ¿Dentro del objeto del cliente, obteniendo los datos de la orden del objeto del cliente para probar la regla? ¿O en un CustomerManager que ejecuta reglas y consume el cliente y objetos de la orden?
Si el sistema está claramente en el campo de Fowler/Evans, no utilizar la solución relacional porque será un trastorno ya que la manera del pensamiento no cabe en la herramienta usada.
Si el sistema está claramente en el campo de Yourdon, usar un mapeador puro de O/R puede generar dolores de cabeza a la hora de implementar reportes o si se utilizan listas que combinan atributos de múltiples entidades. En este caso se necesita una funcionalidad que permite realizar consultas escalares, y una aproximación que permita que permita pensar a partir del modelo relacional.

Otra posible división de los mapeadores objeto/relacional se basa en la perspectiva que el mapeador puede asumir [9]:

Orientado a la metadata

La única entrada del desarrollador de la aplicación para el proceso de mapeo es la metadata (información sobre los datos). El sistema ORM por si mismo es el encargado de generar el código.
La base de datos puede existir previamente y es descripta por la metadata o el mapeador puede generar la base de datos como lo hace con el código. Esto varía de aplicación en aplicación. En sistemas que están construidos desde cero, es conveniente que el mapeador construya la base de datos. Si en cambio, la aplicación usa una base de datos legacy, el mapeador no necesita generar la base de datos, pero si debe escribir el código necesario para que sea compatible con el esquema existente.
La ventaja de este método es la simplicidad para el desarrollador. No solamente se despreocupa del mecanismo de persistencia, sino también del trabajo interno que lleva a

cabo para mapear los objetos a tablas de la base de datos y viceversa (excepto en la medida que el desarrollador necesite especificar algunos de estos detalles en la metadata)
La desventaja es la pérdida de flexibilidad. Como el código es generado por el mapeador, el desarrollador generalmente no lo modifica y está limitado a la funcionalidad que provea el mapeador. Análogamente, como la base de datos es generada por el mapeador o bien ya existe, el desarrollador no puede modificarla directamente.

Orientado a la aplicación

El desarrollador de la aplicación escribe el código orientado a objetos para la aplicación completa, incluyendo los objetos que van a ser mapeados (menos el código asociado con la persistencia de los mismos). El mapeador, generalmente apoyado por metadata adicional provista por el usuario, escribe el código asociado a la persistencia de dichos objetos. El mapeador puede crear la base de datos en la cual los objetos van a persistirse o puede usar una base de datos existente o una creada específicamente para la aplicación.
La ventaja de este método es la flexibilidad, ya que este modelo permite al desarrollador escribir lógica en las clases que van a ser mapeadas. Este modelo además tiene una ventaja teórica que permite a los desarrolladores de la aplicación focalizarse en su dominio de experticia, es decir el paradigma orientado a objetos, dejando de lado, un campo menos familiar y mas distante de la lógica del negocio, como es en este caso, la base de datos. Debido a estas ventajas, muchos de los sistemas ORM más populares de hoy en día, son orientados a la aplicación.
Un requerimiento obvio para los mapeadores orientados a la aplicación es que sean capaces de leer el código. Hay tres aproximaciones para esto. La primera, es que el mapeador pueda parsear e interpretar el código fuente. La segunda, que el mapeador pueda parsear e interpretar un código resultante de compilar el código fuente (por ejemplo, código maquina o bytecode), La tercera es que pueda usar reflection para acceder a la estructura del código en tiempo de ejecución (runtime). El último método es el más sofisticado y el que mayor cantidad de recursos utiliza.

Orientada a la base de datos

El desarrollador de la aplicación construye la base de datos y el mapeador examina la base de datos e infiere el modelo de clases de la misma (frecuentemente asistido por metadata). Finalmente genera el código para el modelo de objetos.
Así como los mapeadores orientados a la aplicación tienen la ventaja de hablar el mismo lenguaje que los desarrolladores (orientado a objetos), los mapeadores orientados a base de datos tienen la desventaja de hablar el lenguaje incorrecto y de no proveer flexibilidad para editar las clases mapeadas.
Por otra parte, este modelo puede adaptarse bien a aplicaciones donde la estructura de datos y las relaciones entre los diferentes tipos de datos son especialmente importantes y donde la lógica de negocio es menos importante.

Problemas con esta clasificación

No todos los mapeadores encajan perfectamente en esta taxonomía. Algunos mapeadores tienen más de un modo de operación. En algunas situaciones, los desarrolladores de aplicaciones pueden desarrollar las tres componentes: código, metadata y base de datos, dejando al mapeador con la única tarea de agregar la

funcionalidad de persistencia al código existente, usando la metadata provista para referenciar la base de datos existente.

Beneficios de usar un mapeador objeto/relacional

- ORM separa la interface de usuario de la base de datos subyacente y de esta manera se logra un completo encapsulamiento. Pueden realizarse cambios en la base de datos de la aplicación sin afectar la interface de usuario.
- ORM permite que las aplicaciones sean escalables.
- Las aplicaciones que usan ORM son mucho más mantenibles. La arquitectura separa las responsabilidades en deferentes capas. Gracias a esta separación de responsabilidades, las aplicaciones son más fáciles de modificar cuando el negocio cambia.
- ORM elimina el requerimiento de escribir SQL para cargar y persistir el estado de los objetos. Aunque aún se tiene que escribir consultas, una buena herramienta ORM debería hacerlo más sencillo.[20]
- ORM puede realizar detección automática de cambios, eliminando una tarea propensa a errores del ciclo de vida de los desarrolladores.
- ORM puede mejorar la portabilidad entre bases de datos reduciendo la dependencia del SQL específico de la base de datos, que se puede abstraer detrás de la herramienta ORM.

Cuando no utilizar un mapeador objeto/relacional

Un ORM no es siempre la mejor opción. Algunas situaciones en las cuales no es apropiado elegir un ORM son [20]:

- Aplicaciones que realizan frecuentes actualizaciones de numerosos registros.
- Aplicaciones OLAP, porque normalmente las bases de datos ofrecen mecanismos de programación nativos.
- Las bases de datos o entornos operativos repletos de código SQL y llamadas a procedimientos almacenados como único mecanismo para recuperar y actualizar datos.
- Aplicaciones que se adaptan mejor con aproximaciones directas basadas en SQL. En estas aplicaciones la lógica de negocio está codificada dentro de la base de datos, o forzadas por restricciones de integridad, y le dan a los usuarios una "ventana de datos" permitiendo que el usuario los edite. En aplicaciones como éstas, ORM tiene menos que ofrecer porque los objetos en general, tienen menos que ofrecer; más allá de una ilusión de orientación a objetos, se puede ganar muy poco modelando las tablas de la base de datos como objetos de dominio.

Elección de un mapeador objeto/relacional

Al momento de elegir un mapeador objeto/relacional existen dos grandes áreas en las cuales hay que focalizarse: los mecanismos de mapeo y la API de persistencia. La API de persistencia no solo actúa como una capa de indirección para la base de datos, sino que también oculta los mecanismos de mapeo entre los objetos y las tablas de la base de datos. Una buena API de persistencia debería realizar esto, sin forzar el modelado de objetos en términos de los tipos de datos y sus relaciones.[21]
Algunas características a tener en cuenta, a la hora de elegir un mapeador

objeto/relacional son:

- Tener en cuenta que la herramienta elegida no tenga demasiadas restricciones de modelado. Por ejemplo, algunas herramientas no soportan relaciones a clases abstractas. Una alternativa a esto, es duplicar las relaciones sobre las clases concretas que extienden de dicha clase abstracta; aunque esta no es una solución muy prolija teniendo en cuenta la orientación a objetos.
- Tener en cuenta que el mapeador permita modelar visualmente (preferentemente usando UML)
- Si UML es considerada una herramienta importante en el desarrollo de la aplicación, asegurarse que sea posible importar UML al mapeador O/R y exportar UML desde el mapeador O/R.
- Examinar el modelo de programación del mapeador O/R y ver si es compatible con lo que se espera obtener de él.
- Observar todas las posibilidades de mapeo que provee para asegurarse que el tipo de relaciones que se imaginaron entre los objetos y tablas son soportados. Generalmente la mayoría de los mapeadores soportan un amplio rango de posibilidades, pero no todos los mapeadores soportan todos los tipos de relaciones.
- Evaluar la performance, aunque se piense en que no se van a llegar a tener muchos requerimientos que degraden el rendimiento.
- Si se desea o necesita utilizar un esquema de base de datos existentes y luego mapear los objetos a partir de este, evaluar si el mapeador O/R soporta el sistema de base de datos que se vaya a utilizar.
- Chequear las capacidades de consultas que provee la interface de persistencia del mapeador O/R. Fundamentalmente, verificar si soporta el uso de funciones agregadas y si es posible consultar valores particulares en lugar de objetos completos. Los objetos completos pueden contener demasiada información cuando, por ejemplo solo se necesita el valor de unas pocas propiedades para llenar una grilla.

Selección de la API de persistencia

La elección de una buena API para persistir objetos es tan importante como la elección del mapeador O/R. La API de persistencia es la parte que posee mayor visibilidad al equipo de desarrollo. La funcionalidad del mapeo O/R es conocida por un grupo reducido del equipo de trabajo, el cual está dedicado al mantenimiento de la capa de persistencia, en cambio la API de persistencia define la interfaz que el equipo de desarrollo completo debe utilizar.[21]
Las API de persistencia se dividen en dos categorías: transparentes y no transparentes.

APIs de persistencia transparentes

Una API de persistencia transparente oculta completamente la persistencia. No necesita proveer de muchos métodos, métodos como load y save son suficiente en la mayoría de los casos. Se define declarativamente en lugar de proceduralmente. Hibernate y JDO son ejemplos de APIs de persistencia transparentes.
Por ejemplo, un objeto Seguro puede contener 0-n objetos Garantía. La aplicación cliente actualiza un atributo de un objeto Seguro. Semánticamente, una Garantía está contenida en un Seguro, de manera que cuando se actualiza un Seguro es factible que implícitamente se actualicen sus Garantías. Según los requerimientos, esto podría ser un

diseño correcto, pero podría tener un impacto negativo sobre la performance, especialmente si no se es conciente de que una actualización implícita de los objetos Garantía está ocurriendo. Cuando se modifica solo un atributo del objeto Seguro, debería ser posible limitar el manejo de persistencia de esta funcionalidad.
La ventaja de esta alternativa es la manera "mágica" en la cual el manejador de persistencia conoce que hacer. El lado negativo es que el manejador de persistencia decide cómo hacerlo.

APIs de persistencia no transparentes

Una API de persistencia no transparente hace menos cosas "mágicas". Comparada con una API transparente, tiene una API más rica, ofreciendo más control al usuario.
Por ejemplo, supongamos una API de persistencia transparente con un solo método persist(). Este método realiza todo lo que parece mágico para el usuario, como chequear si hay asociaciones que potencialmente necesitan persistirse. Aunque esto podría sonar atractivo, cuando se seleccione una API de persistencia hay que asegurarse que puedan realizarse optimizaciones. Cuando se persiste un objeto, es posible que estos objetos asociados no necesiten ser persistidos aun. ¿Qué debería hacer el método persist(), persistir automáticamente estos objetos asociados o dejarlo en consideración del cliente?
Para ilustrar el poder de una API de persistencia no transparente, veamos un ejemplo de la API SimpleORM. Supongamos una clase Seguro con dos métodos. El primer método carga explícitamente todos los hijos asociados a este objeto:

```
seguro.getAllChildren(seguro.getGarantias())
```

El segundo método solo lista aquellos ítems que ya se habían recuperado de la base de datos. Es decir no retorna los ítems que fueron agregados al objeto en memoria:

```
seguro.getRetrievedChildren(seguro.getGarantias())
```

La ventaja en este caso es que el usuario de la API puede "ver" que es lo que hace la API y puede tomar decisiones teniendo en cuenta los costos de performance. La desventaja es que la interfaz es más compleja que la de la API transparente.
Una variante, una vez elegida una API de persistencia, es wrapear esta API implementando los ajustes necesarios en los mapeos según las necesidades de la aplicación.

Razones para wrapear una API de persistencia

Una vez seleccionado el mapeador O/R, se debe decidir si se va a utilizar directamente su API de persistencia o si se va a wapear.[21]
Algunas buenas razones para wrapear la API de persistencia son:

- Si se quiere agregar lógica extra (por ejemplo, validaciones no son parte de un mapeador O/R, pero se necesita que esto sea una parte inherente de los objetos persistentes)
- Aplicar el principio de subsistema, evitando un fuerte acoplamiento con un mapeador O/R específico. De esta manera, el mapeador se trata como un subsistema y se trabaja con una interfaz.
- Se quieren limitar las características que los clientes puedan usar. Por ejemplo, el mapeador podría soportar el uso de SQL directo pero no se quiere exponer este aspecto.
- Se quiere exponer ciertos servicios de diferentes maneras. Por ejemplo, se quiere

introducir un objeto consulta en lugar de exponer una interfaz sobre consultas OQL.
- Una razón errónea para wrapear una API de persistencia es pensar que esto mejorará la performance. Una mejor manera de atacar los problemas de performance es usar patrones o estrategias de diseño adecuados para cada situación.

Otra herramienta de mapeo objeto/relacional

Los mapeadores no son la única herramienta para manejar la persistencia. Otra alternativa son los generadores de código.
Los generadores de código generan código fuente automáticamente, basándose en metadata de alto nivel. El mayor beneficio de los generadores de código es que se libera a los desarrolladores del desarrollo del mapeo y se reduce la cantidad de código propietario en la aplicación resultante. Una desventaja es que los generadores de código no tienen la lógica suficiente para refactorizar el código repetido que generan.
Las principales diferencias entre las herramientas de mapeo objeto/relacional y los generadores de códigos es que:

- Consultas dinámicas solo son manejadas por los mapeadores.
- Los mapeadores usan reflection, lo cual hace que sea mas lento que el código compilado producido por los generadores de código
- Dependiendo la situación, puede adecuarse mejor las herramientas de generación de código o herramientas de mapeo. También existen herramientas que combinan ambas alternativas para ofrecer lo mejor de ambos mundos.

Conclusiones

El uso de mapeadores objeto/relacional presenta las mejores ventajas de los dos modelos:

- Es posible programar con orientación a objetos, aprovechando las ventajas de flexibilidad, mantenimiento y reusabilidad que aporta el lenguaje.
- Es posible usar una base de datos relacional, aprovechando de su madurez y su estandarización así como de las herramientas relacionales que hay para ella.

En general, en los proyectos de software enterprise, las bases de datos relacionales son elegidas por diferentes razones:

- los datos a ser utilizados por la nueva aplicación, ya existen en una o mas bases de datos relaciones
- si bien los datos son nuevos, es decir hay que crearlos, existen razones políticas o técnicas de la organización para usar una base de datos relacional.

Se calcula que usar una producto de mapeo objeto/relacional puede reducir el código de una aplicación en un 30% a 40%, haciéndola menos costosa de desarrollar. Además, el código que se obtiene programando de esta manera es más limpio y sencillo y, por lo tanto, más fácil de mantener y más robusto. Además, la capa de mapeo no sólo simplifica la programación, porque se encarga de realizar la conversión objeto a tabla y tabla a objeto, sino que permite hacer ciertas optimizaciones de rendimiento que serían difíciles de programar.
También es posible, escribir una capa de mapeo objeto/relacional propia, ajustada a las necesidades de la aplicación que se está desarrollando. Sin embargo, a menos que los mapeos que requiera realizar la aplicación sean muy sencillos, esta no es una buena idea. Una capa de mapeo es una pieza de software genérica y resulta difícil explicitar todos los aspectos realmente importantes (ya sea para construirlos o para no focalizarse en ellos)

que requiere una aplicación.
Implementar una capa de mapeo propia presenta varios problemas. Requiere mucho esfuerzo escribir código que no está relacionado con la problemática del negocio sobre la que está impulsado el desarrollo de la aplicación. Mediante los modelos que permiten medir como aumenta la cantidad de código defectuoso con el código total de la aplicación, se puede observar que una cantidad significativa de defectos adicionales, no están relacionados directamente con el problema de negocio sino con el desarrollo de la capa de mapeo. Es necesario llevar a cabo buenas políticas de testing para garantizar la calidad del producto. Se debe tener documentación y soporte para que nuevos desarrolladores aprendan a utilizar la herramienta y facilite el mantenimiento.

Capítulo V - Problemas de performance

Introducción

La mayoría de las aplicaciones enterprise de hoy son desarrolladas mediante técnicas de modelos orientados a objetos pero siguen usando bases de datos relacionales para persistir los datos. Como vimos en los capítulos anteriores, por este motivo, resulta necesario implementar una capa de mapeo objeto/relacional.
Algunas de las características que presentan las aplicaciones enterprise, como el volumen de datos que manejan, la complejidad de los objetos de negocio, las necesidad de realizar operaciones masivas sobre los datos, llevar a cabo consultas complejas, por ejemplo para generar reportes y la necesidad de guardar todos los cambios sobre los datos para poder luego consultar la historia de los mismos; hacen que los diseñadores y desarrolladores de este tipo de aplicaciones, deban enfrentarse a una amplia variedad de problemas de performance.
Independientemente de la estrategia elegida para implementar la capa de mapeo objeto/relacional, es necesario resolver los problemas de performance que se presentan, ya que muchas veces de ello depende el éxito o fracaso de un proyecto.

Problema N+1

El problema N+1 es muy conocido en todas las aplicaciones, pero especialmente en aquellas que utilizan mapeadores objeto/relacional. En este último grupo, es común la aparición de este problema dado que al trabajar en un mundo de objetos perdemos la visión de lo que sucede a nivel de base de datos.
El problema se puede traducir a: “ejecución de más consultas de las necesarias, especialmente en modelos de objetos con relaciones de listas involucradas”.
Supongamos que tenemos un modelo de objetos con dos clases A y B. La relación entre ambas clases consiste en que un objeto A “tiene” muchos B y un objeto B sólo es conocido por un objeto A. Desde el punto de vista de los mapeadores la relación de A hacia B es de uno-a-muchos.
En este contexto debemos plantearnos que queremos que suceda cuando recuperemos una lista de B. El inconveniente surge cuando no se realiza un análisis de que tipo de consulta se va a realizar y cómo se encuentra mapeada la relación entre las dos clases.
El problema que puede surgir es que cuando se ejecute la consulta que recupere todos los objetos B, por cada uno de estos se realice una consulta para recuperar el objeto A correspondiente.

Ejemplos

Tenemos un modelo en donde nuestras clases principales son Proveedor y Producto. El Proveedor tiene una lista de Producto, que representa los productos que un proveedor puede vender, y el Producto conoce a un Proveedor, donde esta última relación representa que cada producto puede ser vendido por un único proveedor.
En nuestro modelo de datos esto se representa con dos tablas, una tabla con los proveedores y una tabla con los productos. En la tabla de los productos tenemos el ID del proveedor.

Desde el punto de vista de los mapeos objeto/relacional, el Proveedor tiene una relación de uno-a-muchos con el Producto y el Producto tiene una relación de muchos-a-uno con el Proveedor.
Supongamos que en un momento se debe recuperar la lista de todos los proveedores de la empresa. Lo que vamos a ejecutar es una consulta del estilo:

```
session.createQuery("from Producto")
```

Esta consulta HQL va a generar una consulta SQL que recuperará de la tabla de Productos todos los registros que cumplan con el criterio (en este caso vacío):

```
Select * from Producto
```

Este paso se realiza siempre, independientemente de cómo esta mapeada la relación. Pero el motor de mapeo relacional debe decidir como recuperará el proveedor asociado a cada producto. Dependiendo de diversos factores de configuración en algunos casos puede suceder que por cada uno de los productos recuperados el motor vaya a recuperar el proveedor asociado, dando como resultado N+1 accesos, es decir 1 acceso para recuperar todos los productos y N accesos para recuperar el proveedor de cada uno de los productos.

Supongamos otro ejemplo donde nuestro modelo de negocio está compuesto por dos clases principales, Proveedor y Contacto. El Proveedor tiene asociado un contacto y cada Contacto puede estar asociado a un Proveedor. El Contacto representa la persona de referencia de un proveedor y por motivos propios del negocio puede suceder que una misma persona sea el contacto de varios proveedores.
Desde el punto de vista de los mapeos objeto/relacional, el Proveedor tiene una relación muchos-a-uno con el Contacto.
Si en nuestra aplicación necesitamos recuperar un Proveedor, realizaremos una consulta similar a la siguiente:

```
session.createQuery("from Proveedor")
```

Esto se traduce a una consulta SQL que realiza un select de la base de datos del estilo

```
Select * from Proveedor
```

Pero además, puede pasar que por cada proveedor se recupere el contacto, es decir que además de la consulta anterior se realice una consulta:

```
Select * from Contacto Where id = 1234
```

donde 1234 es el id del contacto asociado al proveedor obtenido en la primer consulta. Esto genera dos consultas cuando podría optimizarse en solo una, pero el real problema surge cuando entramos en un ciclo, dado que el motor debe recuperar el contacto del proveedor y esto implica que se deba ejecutar el ciclo de vida de carga del objeto Contacto completo. Esto implica recuperar todos los proveedores del Contacto, es decir que estamos recuperando un grafo demasiado grande y estamos trayendo los demás proveedores del contacto que en este caso puntual no nos interesan.

Conclusión

El problema de las consultas N+1 se presenta en relaciones aparentemente simples, por ejemplo entre un producto y su proveedor, en donde cuando recuperamos una lista de productos ejecutamos N consultas para recuperar cada uno de los proveedores. También se presenta en relaciones de listas (uno-a-muchos) pero esto es fácilmente solucionable

con lazy loading y es muy común que se presente siempre que se tengan relaciones muchos-a-uno.
Este problema tiene muchas alternativas de solución. Puede solucionarse con técnicas de configuración de los mapeos como fetch join haciendo que las relaciones se manejen en una sola consulta, lazy loading para que no se carguen las relaciones a menos que estas sean requeridas o haciendo uso de un caché para que el motor no realice un join entre las tablas sino que recupere la relación desde la caché (en el primer ejemplo, sería una caché para los Proveedores).
Este problema de las consultas N+1 debe ser controlado muy de cerca, fundamentalmente en aplicaciones que recuperan grandes volúmenes de información dado que puede afectar drásticamente la performance de una solución.

Operaciones masivas

Una operación masiva es la ejecución de una misma operación unitaria sobre un subconjunto de entidades del mismo tipo, utilizando alguna política de ejecución.
Llamamos operación unitaria a la ejecución de la operación sobre uno de los elementos.
Durante la ejecución de una operación masiva se deben realizar las siguientes actividades:

- Identificar los elementos sobre los cuales se va a ejecutar la operación masiva.
- Iterar sobre los elementos seleccionados.
- Ejecutar la operación unitaria sobre cada elemento.

El problema ante este tipo de situaciones en que generalmente no se conoce desde fuera de la entidad, los cambios en el estado interno de la entidad que fueron realizados, es decir que no se conoce si en el objeto se modificó sólo el valor de un atributo o si se agregaron múltiples objetos a una asociación con otras entidades. En resúmen podemos decir que el problema radica en no saber cuántas actualizaciones en la base de datos se pueden realizar, lo cual obliga a implementar mecanismos de dirty check u simplemente siempre actualizar todo.

Ejecución de operaciones unitarias

Se pueden distinguir dos tipos de operaciones unitarias: alteración del estado interno de la entidad y operación de negocio sobre la entidad.
La alteración del estado interno (o edición) consiste en la ejecución de una operación básica aplicada a una propiedad de una entidad dada, como por ejemplo, operaciones aritméticas o reemplazo de valores. Este tipo de operaciones se presenta cuando desde el proceso conocemos exactamente cual es la modificación al estado interno de la entidad que se va a realizar (por ejemplo que propiedades se van a modificar). Una edición unitaria tiene dos posibles estrategias para su ejecución:

- En la Base de Datos. Se realiza la edición de una manera eficiente, pero no se ejecuta ningún tipo de validación sobre la entidad al realizar la modificación. (UPDATE sobre la entidad)
- En Memoria. El objetivo de hacerlo en memoria es que se ejecuten validaciones y chequeo de integridad previos a realizarle la modificación. En este caso el elemento es cargado en memoria.

La ejecución de operación de negocio consiste en la ejecución de operaciones con lógica compleja sobre una entidad, la cual puede alterar dicha entidad, modificar entidades relacionadas, generar nuevas entidades o cambiar el estado de la entidad. A diferencia de la edición no conocemos desde fuera de la entidad cuales son los cambios que se

producirán dentro del estado interno de la entidad. En este caso, durante la ejecución de la operación de negocio, se podrían crear nuevas entidades, eliminar entidades dependientes, modificar propiedades simples, etc.

Subconjunto de entidades

La especificación del conjunto de entidades al cual se le va a aplicar la operación masiva puede ser por comprensión o por extensión.

- Por comprensión, permite definir un filtro a partir del cual se podrá obtener el subconjunto de entidades a las cuales se aplicará la operación masiva. Por ejemplo una consulta HQL, SQL o cualquier lenguaje de expresión que me permita recuperar entidades que cumplan una determinada condición.
- Por extensión, permite seleccionar el conjunto de entidades concretas a las cuales se aplicará la operación masiva. Por ejemplo seleccionándolas de una grilla.

Políticas de ejecución

Las diferentes estrategias para iterar sobre los elementos a aplicar la operación masiva son: transacción única con recorrido completo, transacción única con recorrido parcial y transacción por elemento.

- Transacción única, recorrido completo: Consiste en una única transacción para el conjunto de elementos, donde si la operación sobre un elemento falla, se deshacen todas las operaciones exitosas anteriores. Aunque un elemento falle, se sigue ejecutando la operación unitaria sobre el resto de los elementos para recolectar los errores, y así saber sobre que entidades falla la operación
- Transacción única, recorrido parcial: Si la operación sobre un elemento falla, se deshacen todas las operaciones exitosas anteriores. Finaliza el proceso y se realiza roollback de la transacción si falla la operación sobre algún elemento, retornando el error encontrado. A diferencia del anterior en este caso no podemos conocer los posibles errores de procesamiento futuro.
- Transacción por elemento: Las operaciones unitarias se ejecutan en transacciones independientes. Aunque un elemento falle, se sigue ejecutando la operación unitaria sobre el resto de los elementos, recolectando los errores encontrados.

Grandes grafos de objetos

Cuando nos referimos a un gran grafo de objetos estamos haciendo referencia a modelos estáticos y dinámicos compuestos por una gran cantidad de “partes”. Los mayores problemas de performance se presentan con los grafos de objetos en su vista dinámica, es decir cuando la cantidad de instancias que forman el grafo de objetos es grande.
En todos los modelos de objetos podemos identificar al objeto principal o Root Object, este objeto es el punto desde el que se desencadenan la mayor cantidad de operaciones (desde allí es desde donde se disparan las operaciones) y es el objeto que tiene el mayor significado semántico para el usuario.
Por ejemplo, si nuestro modelo de objetos esta formado por las clases País, Ciudad y Persona. Cada país tiene una lista de habitantes (instancias de la clase Persona), donde esta relación forma una composición y cada país posee una lista de ciudades (instancias de la clase Ciudad) donde la relación forma una agregación.
Sería lógico pensar que por cada operación de guardado ante una actualización del País (root Object del modelo) se requiera el guardado en cascada de todas sus dependencias,

es decir todos los habitantes del país y todas las ciudades. Tengamos presente que estamos hablando de millones de operaciones de actualización.
En el caso de la creación de un país podríamos pretender armar nuestro modelo de objetos en memoria (el país con todas sus ciudades y todos sus habitantes) y almacenar nuestro root object para que luego en cascada se almacenen todos los objetos dependientes.
En el caso de que se requiera por ejemplo mostrar los datos de un país por pantalla, la alternativa inicial es la de recuperar el root object País pero esto puede traer como consecuencia que se consulten todas sus dependencias, es decir, todos los habitantes del país y todas sus ciudades.
En el contexto de la eliminación del país la lógica común indica que se deberían eliminar en cascada todas las relaciones de composición pero no las de agregación.
Estas situaciones describen problemas comunes que tienen las aplicaciones que trabajan con grandes grafos de objetos y se reducen a que en ciertos casos y operaciones sólo se modifica o consulta una pequeña parte de este grafo (por ejemplo, solo la descripción del país y no sus 40 millones de habitantes). Por este motivo, es necesario tomar decisiones que permitan realizar operaciones discriminadas en solo una parte del grafo y no en todo el grafo en su conjunto e incluso romper con el encapsulamiento de los objetos y trabajar con parte de la estructura interna de estos y no con el objeto entero.

Acceso concurrente al mismo objeto

Una de las características de las aplicaciones enterprise es que son multiusuarios, esto significa que la información será accedida por varios usuarios a la vez. El problema ocurre cuando más de un usuario accede concurrentemente al mismo dato.
Los problemas de concurrencia afectan a dos aspectos fundamentales de un sistema: la integridad de los datos y la performance de la aplicación.
Al permitir el acceso de múltiples usuarios a la misma información, existe la posibilidad de que un usuario modifique parte de la información, la cual puede ser sobreescrita sin intención, por los cambios de otro usuario. Si esta situación ocurre, los datos quedan inconsistentes y el sistema muestra información errónea o peor aún, se pueden producir errores inesperados. Por otro lado, las técnicas utilizadas para prevenir este tipo de pérdidas, puede reducir drásticamente la performance de la aplicación, ya que algunos usuarios tendrán que esperar a que otros usuarios completen sus operaciones, para poder continuar.
Veamos los conceptos básicos relacionados con concurrencia:
Una transacción es una unidad indivisible de trabajo que impacta sobre ciertos datos. Todas las modificaciones que realiza una transacción se aplican de manera uniforme a la base de datos mediante la sentencia COMMIT o los datos afectados por los cambios de la transacción retornan a su favor inicial mediante la sentencia ROLLBACK. Cuando una transacción es comineada, los cambios hechos por la transacción pasan a ser permanentes y son visibles al resto de las transacciones y usuarios.
Durante el acceso concurrente se deben garantizar las 4 propiedades ACID: Atomicidad, Consistencia, Aislamiento y Durabilidad.

- Atomicidad. Una transacción debe ser tratada como una unidad de procesamiento indivisible. Por lo tanto, todas las acciones de la transacción se realizan o ninguna de ellas se lleva a cabo. Asimismo, si una transacción se interrumpe por una falla, sus resultados parciales deben ser desechados.
- Consistencia. Los recursos del sistema deben estar en un estado consistente y no-corruptos al comenzar la transacción y finalizada su ejecución. Si la transacción no puede alcanzar un estado final estable, debe retornar el sistema a su estado inicial

en el que se encontraba antes de comenzar la transacción.

- Aislamiento. Una transacción en ejecución no puede revelar sus resultados a otras transacciones concurrentes antes de su completitud. Si varias transacciones se ejecutan concurrentemente, los resultados deben ser los mismos que si ellas se hubieran ejecutado de manera secuencial (seriabilidad). La propiedad de aislamiento no asegura cual transacción se ejecutará primero sino que las transacciones no se interferirán unas con otras.
- Durabilidad. Se debe asegurar que una vez que una transacción finaliza, sus resultados son permanentes y no pueden deshacerse.

Los DBMS utilizan un sistema de lockeo para evitar que los efectos de una transacción interfieran en los efectos de otras. Existen dos tipos de lockeos: exclusivo y compartido. Un lockeo exclusivo o de escritura es utilizado para realizar modificaciones sobre datos dentro de una transacción. Un lockeo exclusivo solo puede ser usado por un usuario al mismo tiempo. Un lockeo compartido o de lectura puede ser utilizado por más de un usuario, ya que mediante este lockeo los usuarios solo pueden leer datos.
Existen varios grados de aislamiento de transacciones para controlar el grado de lockeo para el acceso a los datos. Las transacciones de base de datos pueden construirse sin requerir un alto nivel de aislamiento, reduciendo el overhead que genera el lockeo en el sistema. En contraposición, si las transacciones se construyen con altos niveles de aislamiento, aumenta la posibilidad de deadlock, lo cual requiere un cuidadoso análisis e implementación de técnicas de programación para evitarlos.
El estándar ANSI/ISO SQL define cuatro niveles de aislamiento, en base a tres factores que deben evitarse al ejecutar transacciones concurrentes[22]:

- Lecturas sucias (dirty reads). Ocurre cuando una transacción lee datos escritos por otra transacción aun no comiteada.
- Lecturas no repetibles (non-repeatable reads). Ocurre cuando una transacción vuelve a leer datos que había leído previamente y encuentra que los datos fueron modificados por otra transacción (fueron comiteados luego de la lectura inicial).
- Lectura “fantasma” (phantom read). Ocurre cuando una transacción vuelve a ejecutar una misma consulta, la cual retorna un conjunto de registros que satisface una condición de búsqueda determinada y encuentra que el conjunto de registros retornados es diferente, debido a que otra transacción comiteó recientemente.

Los cuatro niveles de aislamiento y sus comportamientos se describen en la Tabla 3.

	Lecturas sucias	**Lecturas no repetibles**	**Lectura “fantasma”**
Lectura no comiteada (read uncommitted)	Posible	Posible	Posible
Lectura comiteada (read committed)	Imposible	Posible	Posible
Lectura repetible (repeatable read)	Imposible	Imposible	Posible
Serializable	Imposible	Imposible	Imposible

Tabla 3: Niveles de aislamiento

Serializable es el nivel más alto de aislamiento para el manejo del acceso concurrente. Como su nombre lo indica, las transacciones serializables parecen ejecutarse en serie y siguiendo un orden. Para el usuario cada transacción serializable luce como si tuviera uso exclusivo de la base de datos durante la ejecución de la transacción. Las transacciones

serializables son reproducibles y predecibles.
Se debe tener en cuenta la estrategia de aislamiento implementada porque una simple transacción puede causar importantes problemas de performance si lockea por ejemplo, un conjunto de datos que otras transacciones necesitan para su completitud. Esta interferencia causada por conflictos de lockeo se denomina contención. A mayor contención, mayor es el tiempo de respuesta.
En general ocurre que al incrementar el acceso concurrente a los datos, se incrementa la contención y se decrementa la performance, tanto en tiempo de respuesta como en throughput.

Consultas complejas

Las aplicaciones que utilizan mapeadores objeto/relacional resuelven el acceso y las consultas a su modelo de objetos utilizando lenguajes de consulta de alto nivel (por ejemplo Criteria, HQL, etc). Las sentencias de estos lenguajes son interpretadas por los motores ORM para generar las sentencias SQL que son enviadas al motor de base de datos.
Estos lenguajes de consulta de alto nivel permiten abstraer a los programadores de los mapeos realizados y del modelo de base de datos subyacente. También abstrae a las aplicaciones de las tablas que luego serán accedidas por el motor ORM por medio de sentencias SQL.
La secuencia que ocurre cuando se requiere recuperar uno o más objetos utilizando mapeadores objeto/relacional puede resumirse de la siguiente manera:
La aplicación necesita recuperar uno o varios objetos, para eso se genera un modelo de consulta de alto nivel que trabaja con objetos. Esta consulta puede ser explícita, por ejemplo con HQL o implícita utilizando la API del motor (por ejemplo invocando el método get de la API o navegando un objeto lazy). Este modelo se envía al motor ORM para su procesamiento. El motor toma esta consulta y en base al mapeo resuelve la consulta de bajo nivel y genera un conjunto de sentencias SQL. Estas consultas SQL son enviadas al motor de base de datos, quien retorna los resultados en formato cursor. El motor ORM toma los resultados e hidrata el o los objetos solicitados por la aplicación. Mediante este proceso los objetos que se quieren recuperar con la consulta de alto nivel son instanciados y su estado interno es llenado utilizando los datos retornados por las consultas SQL ejecutadas. Finalmente se retornan a la aplicación los objetos solicitados.

Si bien en la mayoría de los casos las consultas de alto nivel son implícitas, es decir se utiliza la API del mapeador, en otros casos se requiere recuperar un subconjunto de nuestros objetos de dominio que cumplan cierto criterio, pudiendo ser este criterio un filtro sobre los estados internos de diferentes objetos del grafo. Este problema es el que llamamos consultas complejas explícitas. Otro caso de este tipo de consultas se presenta cuando se requiere recuperar o comparar múltiples subconjuntos de múltiples grafos de objetos, por ejemplo para determinar cambios y alteraciones históricas en los objetos. O cuando se requiere realizar una consulta con lógica de negocio compleja, por ejemplo para generar un reporte en nuestra aplicación.
En resumen, los problemas de performance generados por consultas se pueden dividir en dos grandes grupos:

Consultas de alto nivel simple que generan operaciones de bajo nivel complejas
Por ejemplo, supongamos un modelo para representar la factura de un comercio. Vamos a tener la cabecera de factura que contiene los datos propios de la factura, como su tipo, número, fecha de emisión, nombre, domicilio, cuit del cliente y su condición frente al IVA.

La factura está compuesta por un conjunto de ítems y el precio final. Cada ítem se compone de una cantidad, el producto y el precio del mismo. Cada producto está definido por un código, descripción, rubro, marca, modelo y proveedor.
Supongamos que los mapeos de la factura y sus ítems no son lazy.
Si desde la aplicación necesitamos buscar la factura número 1000 se generará una consulta de alto nivel simple e implícita en este caso, ya que se obtiene invocando el método get de la API del ORM indicando la clase Factura y el ID de la factura 1000.
El motor generará un gran conjunto de consultas SQL que serán enviadas al motor de base de datos, para recuperar no solo el objeto Factura, sino también el objeto Cliente con sus objetos Domicilio, Ciudad, Provincia, País, Condición IVA.
Además se van a recuperar todos sus ítems y de cada objeto ítem, el objeto rubro, marca, modelo y el proveedor, es decir que se recuperará el grafo completo del modelo.
Esta misma situación ocurre si desde la aplicación se hace una búsqueda de todas aquellas facturas emitidas entre el 01/01/2010 y 21/01/2010. En este caso, la consulta de alto nivel es simple y consiste en un HQL o un Critera indicando la clase Factura y la condición de filtro. La consulta de bajo nivel que genera es la misma que se mencionó anteriormente para una factura pero para recuperar todas y cada una las facturas cuyas fechas de emisión están dentro de las fechas del filtro.

Consultas de alto nivel complejas que generar operaciones de bajo nivel complejas
Por ejemplo, continuando con el modelo para representar la factura de un comercio, supongamos que se guarda la siguiente información de los proveedores: su razón social, el CUIT, el objeto que representa su domicilio (calle, número, ciudad y provincia).
Desde nuestra aplicación se necesita hacer una consulta para obtener aquellas facturas emitidas entre el 01/01/2010 y 21/01/2010, en las cuales se vendió algún producto cuyo proveedor pertenezca a la ciudad de La Plata. Esta consulta de alto nivel que se genera es compleja. En este caso, la consulta se puede generar mediante un HQL o si resulta muy complejo puede usarse directamente SQL, perdiendo en este último caso, la abstracción sobre el modelo de base de datos.
Las consultas SQL que se generan, son muy complejas ya que requieren hacer joins de varias tablas (las tablas donde se persiste la cabecera de la factura, los ítems, los proveedor, los domicilios y las ciudades); e incluso utilizar operaciones como el exists o in las cuales se caracterizan por tener un impacto negativo en la performance.

Versionado de objetos e historia de modificaciones

Las aplicaciones orientadas a objetos trabajan sobre un conjunto de datos semánticamente complejos que varían en el tiempo. Manejar y administrar estos cambios es una tarea difícil y costosa.
El versionado de objetos es una estrategia que permite a una aplicación administrar los cambios y acceder a la historia de los objetos (es decir a estados anteriores de los objetos). La implementación de estos mecanismos representa desafíos tecnológicos que requieren concentrarse en la eficiencia de tiempo y espacio para guardar los estados anteriores de los objetos. Cuando nuestros grafos de objetos son complejos el versionado se transforma en un problema a resolver.
Debemos mencionar que el versionado de objetos puede ser utilizado para implementar políticas de acceso concurrente a los objetos, pero en esta sección nos vamos a concentrar en el manejo del historial de cambios de los objetos.

Estrategias

Existen diferentes estrategias para implementar el versionado de objetos, la más sencilla consiste en guardar en el objeto, el período de tiempo durante el cual se considera válido. Luego se utilizará este período para obtener el objeto adecuado en un momento de tiempo determinado. La desventaja de esta alternativa es que es explícita, quien use este objeto, cualquiera sea el contexto, tiene conocimiento que el objeto contiene en si mismo el aspecto temporal. En algunos casos puede convenir ocultar las cuestiones temporales cuando no se necesitan, dejando el modelo de objetos agnóstico de estos aspectos.
Otra alternativa consiste en modelar un objeto con un conjunto de versiones explícitas del mismo, donde cada versión contiene los cambios que se hacen a través del tiempo. Esto permite referenciar al objeto actual o a una versión específica en un punto determinado del tiempo. También permite ver fácilmente, como fue cambiando el objeto a lo largo del tiempo.

Alcance del versionado

En aplicaciones donde el grafo de objetos es complejo, es decir muchos objetos relacionados, independientemente de la estrategia de versionado elegida, debemos analizar sobre que objetos es necesario guardar la historia de los cambios y cómo se quiere explotar dicha historia. Surgen dos opciones bien definidas:

- versionar solo el objeto raíz del grafo
- versionar todo el grafo de objetos

Si se versiona solo el objeto raíz, solo se almacenará la historia de los cambios de este objeto. Si cambia el estado de alguno de los objetos relacionados al objeto raíz, a cualquier nivel de distancia, estas modificaciones no se persisten. La ventaja de esta alternativa es que requiere almacenar menor volúmen de información, las operaciones para almacenar y recuperar la historia son más simples y como consecuencia, los tiempos de procesamiento son menores. La desventaja radica en que se tiene menos control de los cambios, dado que no se almacena el historial de TODO el grafo de objetos.
Por ejemplo, supongamos que tenemos el objeto Persona como raíz del grafo, el cual se relaciona con el objeto País mediante una relación de agregación, donde esta relación representa el país de residencia de la persona. En este caso, resulta natural adoptar la alternativa de versionar solo la raíz del grafo, dado que ante un cambio en el estado del objeto persona solo se necesita guardar los valores anteriores al cambio, de los atributos que representan a la persona, no es necesario guardar la historia del país porque el mismo no ha cambiado.
Si se versiona todo el grafo de objetos, ante el cambio en un objeto se almacenará en cascada el estado anterior al cambio de todos los objetos que componen el grafo, cuya raíz es el objeto que se modificó directamente. En este caso, la ventaja es que se tiene mayor control, dado que se almacenan todas las modificaciones de todos los objetos. Las desventajas que se presentan son que el tiempo de procesamiento es mucho mayor, dado que ante un pequeño cambio en un objeto, se requieren guardar una versión de todo el grafo completo y las operaciones necesarias para recuperar el estado del grafo de objetos en un momento determinado de tiempo son complejas y costosas.
Por ejemplo, supongamos que tenemos un objeto raíz Factura compuesto por una colección de objetos Item de factura, donde la relación es de composición. Si cambia algún atributo de la factura, es necesario guardar el estado anterior de la factura y todos los objetos que la componen.
Cuando el grafo de objetos es complejo y de gran volúmen, cuando se requiere versionar el grafo completo y las modificaciones a los objetos son muy frecuentes, el versionado de los objetos, la persistencia de los cambios y la recuperación del estado de todos los

objetos en un momento de tiempo determinando son operaciones que van a tener un impacto negativo en la performance. Por este motivo, es necesario analizar en primer lugar, sobre qué objetos se requiere mantener el historial de cambios, para qué y cómo se va a utilizar esta información histórica almacenada.
La información histórica puede ser utilizada con diversos fines como por ejemplo hacer reportes esporádicamente, fuente de información de auditoría ante alguna inconsistencia de datos, resolver dudas sobre el estado actual de ciertos datos o si es necesario analizar diariamente los cambios que va sufriendo un objeto. En este último caso las consultas sobre los datos pueden ser muy costosas y complejas de resolver. Cualquiera sea el caso, es necesario encontrar estrategias que nos permitan mejorar el rendimiento de estas operaciones y como vimos anteriormente la elección de la estrategia correcta dependerá del objetivo del versionado y del tipo de relación que existen entre los objetos.

Lecturas completas y simples de objetos

Unas de las grandes ventajas que tenemos al utilizar mapeadores objetos/relacionales es que podemos abstraernos de los detalles de cómo los objetos son almacenados en la base de datos. Si bien esta abstracción representa grandes beneficios en ciertos casos puede traer acarreado consecuencias indeseadas.
El contexto más simple y habitual donde se presentan problemas con esta abstración es el de la lectura o recuperación de objetos simples y está asociado a casos en donde se requiere recuperar un objeto pero solo se va a “utilizar” algunas partes de su estado interno.
Por ejemplo supongamos que tenemos el objeto Pais que posee una lista de Provincia, es decir que nuestro modelo de objetos esta compuesto por países y cada país tiene una lista de las provincias. Desde el punto de vista de objetos esta composición de objetos es siempre así y no debería darse el caso en donde el Pais no esté representado por sus datos y todas sus provincias.
En este sentido nuestra aplicación dispondrá de un API (en algún layer de la arquitectura) que me permita recuperar un país por medio de algún identificador, o recuperar todos los países.
Pero ¿que pasa cuando mi aplicación requiere en algún punto mostrar solo una lista de países con su código y su descripción? La solución debería ser que nuestra aplicación muestre en ese caso el país sin la lista de provincias cargadas (rompiendo el encapsulamiento del objeto).
Pero ¿Qué pasa cuando mi aplicación requiere en algún punto mostrar el país con todas sus provincias? Entonces en ese caso la aplicación deberá entrar el modelo de objetos completos.
En grandes aplicaciones el diseño de las APIs de acceso al negocio son diseñadas y construidas de manera independiente de quien las vaya a utilizar, es decir que se diseñan mecanismos para recuperar el país pero no se sabe exactamente quien las va a utilizar y tampoco para que las van a utilizar. Por ejemplo, el layer de negocio presta servicios al layer de presentación, pero el layer de negocio no sabe exactamente para que le están pidiendo un país, por ende no sabe si lo tiene que retornar completo o parcial.
Este problema es muy común en grandes aplicaciones, en donde se requiere recuperar objetos completos o de manera parcial dependiendo el caso. Y en muchos casos ni siquiera se puede predecir que tipo de recuperación se debe realizar.
Para solucionar estos problemas se recurre a diseños de API de recuperación de objetos muy específicas o a técnicas de lazy loading y proxy que veremos más adelante.

Conclusiones

Al diseñar y desarrollar aplicaciones enterprise utilizando un lenguaje orientado a objetos para modelar la lógica de la aplicación, una base de datos relacional para la persistencia de los datos y como consecuencia, una capa de mapeo objeto/relacional, nos vamos a enfrentar inevitablemente con alguna de las situaciones anteriormente mencionadas o alguna situación derivada de ellas, donde la performance de la aplicación se vea degradada y sea el principal problema a resolver.

Resolver estos problemas requiere dejar de ser puristas y estrictos con las metodologías de trabajo adoptadas y pensar alternativas de solución donde se aproveche al máximo las ventajas que presenta cada una de ellas.

Es de suma importancia estar conscientes de que cuando se trabaja en grandes aplicaciones, utilizando técnicas orientadas a objetos y mapeadores se debe analizar en fases muy tempranas de la aplicación cuales son los requerimientos no funcionales en cuanto a performance en el acceso a datos. En base a estas necesidades se decidirá si se tomarán decisiones de compromiso en el diseño del modelo de objetos, técnicas de mapeo, diseño de API de acceso al negocio y tunning.

La solución a los problemas de perfomance de ORM no solo radica en el tunning de las herramientas sino que generalmente son producidos en gran medidas por una falta de coherencia en el diseño de modelos de objetos no alineados con la realidad de la aplicación, el voluúen de datos que esta maneja y el tipo de arquitectura.

Capítulo VI - Estrategias de optimización

Introducción

En este capítulo analizaremos diferentes técnicas de optimización que pueden ser utilizados para abordar los problemas mencionados anteriormente y otro tipo de problemas. Las técnicas de optimización no deben ser tomadas de manera reactivas (cuando el problema está presente) sino que deben ser tenidas en cuenta al momento de comenzar el diseño de una aplicación de manera de poder prevenir los problemas que pudieran presentarse.
Las técnicas planteadas incluyen recomendaciones de diseño de objetos, mecanismos puntuales para optimización de productos de mapeo, patrones de integración de mapeadores con otros productos y patrones de arquitectura que pueden permitir a las aplicaciones adaptar las funcionalidades de los ORM a las necesidades puntuales de cada caso manteniendo la performance deseada.

Lazy loading

Lazy loading (carga perezosa) es una estrategia que consiste en demorar la carga de un objeto hasta que este sea requerido. Es decir, la carga del objeto se realiza de manera explícita cuando este es invocado.
Lazy loading se utiliza en aquellos casos en los que la aplicación necesita acceder solo a una parte de un objeto, a un subgrafo del objeto o a algunos atributos de un objeto.
Por ejemplo, cuando queremos mostrar solo la lista de nombres de los proveedores, no es necesario recuperar ningún otro atributo de los proveedores, como sus domicilios, ciudades, provincias y países asociados.
La estrategia por defecto, en los mapeadores objeto/relacional consiste en que cuando se carga un objeto a memoria, se cargan también todos los objetos del grafo relacionados. En aplicaciones donde el modelo de objetos es complejo, es decir tiene muchas relaciones y todos los objetos están interconectados, si no se utiliza lazy laoding, cargar un objeto a memoria significaría cargar una gran cantidad de objetos (ejecutando un gran número de consultas) en memoria cuando solo necesitábamos sólo un objeto.
Utilizando lazy loading se minimiza la cantidad de consultas a la base de datos. En lugar de generar una consulta para cada objeto o colección asociado al objeto que se quiere cargar, al marcar como lazy estos atributos, solo se ejecuta un número reducido de consultas SQL a la base de datos para recuperar el objeto en cuestión y la menor cantidad de relaciones posibles. Las consultas SQL necesarias para obtener los objetos asociados recién se ejecutarán cuando se requiera acceder a dichas asociaciones. En lugar de cargar en memoria la entidad o la colección asociada al objeto, se crea y asocia un proxy.
Un objeto proxy contiene la información suficiente para que el sistema identifique el objeto que está representando, por ejemplo los atributos que forman parte de la identificación del objeto, y contiene información suficiente para que los usuarios identifiquen el objeto. La idea es que en lugar de traer todos los datos de cada objeto del conjunto resultado de la consulta, solo trae la información de identificación de estos objetos. Esta información es la que se muestra al usuario o al sistema. Cuando se selecciona un proxy para operar con él, el sistema recién ejecuta las consultas SQL necesarias para retornar la información completa del objeto de negocio seleccionado y permite continuar trabajando con el objeto.

Por ejemplo, tenemos el objeto Empleado que tiene asociado un objeto Domicilio y una colección de Departamento en los que trabaja.
Si en el mapeo del objeto Empleado indicamos que las asociaciones con su domicilio y sus departamentos son lazy, cuando queramos cargar a memoria el objeto Empleado se genera una consulta SQL a la base de datos para recuperar el empleado.
No se generan las consultas SQL para recuperar el domicilio y los departamentos en los que trabaja, sino que se crea un objeto proxy para el domicilio y otro para la colección de departamentos.
El objeto proxy representa la asociación no inicializada. Para asociaciones de tipo colección, como el caso de la propiedad departamentos, es una instancia de la clase Collection. Para las asociaciones de valor simple, como el caso de la propiedad domicilio, el proxy toma la forma de una subclase de la clase persistente relacionada (Domicilio) generada en runtime.
Recién en el momento que se necesite acceder al domicilio del empleado, el proxy va a disparar la consulta SQL para recuperarlo. Si el objeto Domicilio nunca es usado durante el ciclo de vida del objeto Empleado, nunca se ejecutará la consulta para recuperarlo.
En el caso de la colección de departamentos, la consulta SQL no se ejecuta cuando se accede a la colección mediante el getter correspondiente, sino que se va a ejecutar cuando se intente acceder a alguno de los objetos de la colección o si se quiere conocer el tamaño de la colección (getDetaptamentos().size()) o verificar si una instancia determinada de Departamento está en la colección (contains(getDepartamentos(), unDepartamento)). Para estas dos últimas operaciones, algunos mapeadores objeto/relacional proveen alternativas para evitar traer a memoria todos los objetos de la colección (extra-lazy en Hibernate).
Otra situación en la cual el uso de lazy loading resulta provechoso, es cuando un objeto tiene atributos pesados (por ejemplo un texto grande o una imagen) y que son poco usados. En la mayoría de los casos que se accede al objeto, no se necesitan estos atributos y cargarlos a memoria hace que la inicialización del objeto sea más lenta y consuma espacio en memoria innecesario.
Por ejemplo, si se almacena la foto de Empleado ocupará alrededor de 100k mientras que el resto de los atributos no llegan, en total, a 1k; y las fotos raramente son accedidas.
Lazy loading es una estrategia que debe utilizarse siempre que sea posible, porque solo los datos que son necesarios se cargan en memoria. Sin embargo, agrega algunas complicaciones. Cuando una asociación lazy (no inicializada) es accedida (y el objeto asociado es recuperado de la base de datos) mucho después que el objeto padre fue cargado y la sesión con la cual se cargó este objeto no se encuentra abierta para poder ejecutar la consulta, se lanza una excepción inesperada. En el caso de Hibernate, esta excepción es LazyInitializationException. Para resolver este problema es necesario mantener la sesión abierta durante más tiempo, el tiempo que dure vivo el objeto.
Otro punto a tener en cuenta es la cantidad de consultas a la base de datos que se ejecutan cuando se atraviesan las asociaciones lazy. La primera vez que se accede a una asociación lazy de un objeto, se ejecuta una consulta a la base de datos. Esto no parece costoso cuando se trata de una sola instancia del objeto, pero si iteramos sobre una lista de varios objetos, generará una cantidad importante de sentencias SQL para ejecutar.
Volviendo al ejemplo anterior, donde un objeto Empleado tiene una asociación con un objeto Domicilio.
Si nuestra aplicación necesita imprimir el domicilio de 100 empleados, asumiendo que la asociación es lazy, se ejecutarán 101 consultas (una consulta para recuperar la lista de empleados y una por cada empleado para recuperar su domicilio).
Esto puede ocasionar el “Problema de n+1 Selects”. En situaciones donde se presente este problema, se debe analizar la lógica de la aplicación, ya que puede ocurrir que

utilizar la estrategia de lazy loading sea menos performante que no usarla. Esta alternativa es de utilidad cuando las asociaciones a un objeto no se leen inmediatamente después de recuperar el objeto. Es la estrategia más común.

Eager fetching

Eager fetching (carga activa) es la técnica opuesta a lazy loading. Cuando se carga un objeto a memoria, aquellas asociaciones marcadas como eager, se levantan junto con el objeto cargado. Con esta estrategia, la carga de un objeto simple, genera más consultas a la base de datos que con lazy loading, pero como las consultas se ejecutan inmediatamente una tras otra y generalmente utilizando la misma conección o sesión en ciertos casos puede ser mas óptimo que realizarlo posteriormente.
La filosofía de este patrón de solución es: adelantar la carga de objetos del grafo que seguramente voy a utilizar en un futuro.
Otra de las ventajas de esta alternativa es que ayuda a evitar el problema de "n+1 Selects", aunque no lo elimina. Si el objeto Empleado tiene asociado el objeto Domicilio como eager, cuando la aplicación necesita recuperar los 100 empleados, se ejecutarán solo 2 consultas: una para recuperar las instancias de los 100 empleados y otra para recuperar las instancias de Domicilio asociados a los objetos Empleado.
Utilizar eager fetching por defecto probablemente no sea una buena decisión. Si se usa esta estrategia en asociaciones de objetos con uso frecuente, fácilmente vamos a llevar a memoria grandes subgrafos de objetos innecesarios. Sin embargo es una buena alternativa para aquellas asociaciones a un objeto que siempre se requieren llevar a memoria cuando se accede a dicho objeto.
A diferencia de la alternativa anterior, la asociación se cargada en memoria inmediatamente después que el objeto asociado es accedido, aun después que la sesión del ORM se encuentre cerrada.
Esta alternativa es de utilidad cuando las asociaciones a un objeto siempre se leen después de recuperar el objeto y cuando ejecutar una consulta separada es mas eficiente que ejecutar un join.
Una ventaja tanto de la alternativa eager fetching como de lazy loading es que no se necesita cambiar el modelo de objetos ni el modelo de datos; sólo hay que modificar los mapeos de los objetos, indicando cómo se quiere cargar en memoria cada asociación que tiene el objeto.

Fetching

Una mejora a la estrategia de eager fetching consiste en llevar a memoria la entidad o la colección asociada al objeto que se quiere recuperar mediante un solo acceso a la base de datos. Es decir, se ejecuta una sola sentencia SELECT utilizando JOINs para recuperar todos los datos en una única consulta SQL.
Esta estrategia presenta las ventajas de la estrategia eager fetching y además reduce la cantidad de accesos a la base de datos, en pos de mejoras de performance.
Un punto a tener en cuenta es que puede ocurrir que se lleven a memoria asociaciones de un objeto que nunca van a ser utilizadas. Además como la consulta que se ejecuta es un join, el conjunto resultado que retorna la base de datos podrá contener gran cantidad de datos repetidos. Si se define más de una asociación del mismo objeto con esta estrategia, se va a crear un producto cartesiano de los datos de todas las colecciones y esto generará un conjunto resultado muy grande.
Esta alternativa es de utilidad cuando las asociaciones a un objeto siempre se leen inmediatamente después de recuperar el objeto y cuando utilizar una consulta de tipo join

no genera demasiado overhead.
Los distintos mapeadores objeto/relacional, proveen varias estrategias de fetching para recuperar objetos asociados cuando la aplicación necesita navegar estas asociaciones. Como vimos hasta ahora estas estrategias se definen en los mapeos, pero también es posible utilizarlas en consultas realizadas en lenguajes de consulta provistos por las herramientas de mapeo. Por ejemplo en Hibernate se pueden ejecutar consultas en HQL o Criteria. Este tema se analizará mas adelante.

Caché

Una de las características más importantes que presenta los ORM es el uso de caché. La caché es la encargada de retener los datos que son solicitados frecuentemente por los usuarios con el objetivo de minimizar la cantidad de accesos a la base de datos, minimizar el tráfico por la red, minimizar la carga del motor ORM y reducir los tiempos de respuestas al usuario.
El uso de una caché permite reemplazar la recuperación de datos mediante llamadas SQL a la base de datos por búsquedas en memoria, logrando reducir los accesos a la base de datos y aumentando la performance del sistema. Además, varios usuarios pueden compartir la caché del mismo sistema, viendo incrementado todavía más el rendimiento de sus aplicaciones, al aprovecharse los datos ya cargadas a memoria por otros usuarios.
Sin embargo, el uso de cualquier caché tiene asociados una serie de desventajas. La primera es que aumenta la complejidad de la aplicación porque se requiere lógica adicional para manejar la caché. Por ejemplo para manejar la consistencia entre la caché y la base de datos o para sincronizar distintas caché. La segunda es que si se actualizan los datos de la base de mediante otro camino que no sea caché, por ejemplo otra aplicación o desde la misma aplicación pero salteando el uso de la caché, la caché no se entera de estos cambios y va a almacenar y trabajar con datos obsoletos. Estos problemas afloran especialmente al aumentar la concurrencia de las aplicaciones y pueden llegar incluso a hacer que la caché se vuelva en contra, y que el rendimiento de las aplicaciones decrezca notoriamente.
El procedimiento habitual de uso de caché es el siguiente:

1. Un usuario requiere un objeto (ya sea mediante una consulta o la navegación de un grafo de objetos)
2. Se comprueba si los objetos están en la caché.
3. Si los objetos están en la caché se devuelven directamente.
4. Si los objetos no están en la caché, se recuperan los datos de la base de datos, se hidratan los objetos, se almacenan en la caché para que futuros accesos se ahorren este paso, que es sin duda es muy costoso; y finalmente se retornan.

Existen dos tipos de caché:

- Caché local o a nivel de proceso: Se utiliza para cachear datos de sólo lectura o datos con actualizaciones poco frecuentes. Solo es adecuada para datos que se actualizan frecuentemente pero en el contexto de una única JVM.
- Caché distribuida: Se utiliza para cachear objetos a través de varias JVM. Es adecuada para datos de lectura y escritura.

Antes de utilizar una caché se debe elegir la estrategia de caché según los requerimientos de la aplicación:

- Caché de sólo lectura. Esta estrategia es útil cuando los datos son leídos con frecuencia y nunca se actualizan. Es la más simple y performante.
- Caché de lectura y escrituras poco frecuentes: No ofrece ninguna garantía de consistencia entre la caché y la base de datos. Con esta estrategia, tenemos un intervalo de tiempo, en el cual existe el riesgo de obtener objetos desfasados. Es

adecuada cuando los datos se leen con frecuencia y son actualizados ocasionalmente. Ideal para almacenar datos que no sean demasiado críticos.

- Caché de lectura/escritura: Esta estrategia se utiliza cuando los datos necesitan ser actualizados con frecuencia. Es la más compleja y menos performante por el procesamiento extra que presenta. Se recomienda su uso cuando no podamos permitirnos datos que queden desfasados.

A medida que bajamos en la lista anterior, el rendimiento disminuye, ya que aumenta la necesidad de sincronización. Cuando solo una aplicación o un usuario utiliza los datos, la sincronización de la caché no es un problema. Si una segunda aplicación o usuario accede a la misma base de datos, necesitamos analizar y tomar decisiones sobre cómo vamos a manejar la sincronización.
Veamos algunas situaciones frecuentes al utilizar caché:

¿Qué sucede si nos descargamos el objeto A de la caché y otro usuario (que lo tiene también en su propia caché) lo modifica?
La modificación concurrente es el problema más habitual en las cachés. Tal es la magnitud de este problema, que normalmente se recomienda el uso de cachés para datos de sólo lectura, o para datos que se actualizan con poca frecuencia. En nuestro ejemplo, si el segundo usuario utilizas la misma caché, no tendríamos ningún tipo de problema, ya que dicha caché ya reflejaría los cambios realizados por el primer usuario. Sin embargo los problemas surgen cuando las cachés son diferentes, ya que en este caso el primer usuario encontraría que el objeto A ya está en su caché local y no lo recargaría de la base de datos, perdiendo la actualización realizada por el segundo usuario. Asimismo nos encontramos con la engorrosa situación de tener dos usuarios trabajando con objetos diferentes, y por si fuese poco, uno de los objetos está desincronizado con respecto a la base de datos y seguramente causará problemas.
Las soluciones a este problema son variadas. El mecanismo de caché puede intentar realizar un chequeo cada cierto tiempo para comprobar el estado de los objetos en base de datos; también se podría tratar de sincronizar las cachés; otra solución sería que el primer usuario eliminase los objetos de la caché antes de realizar la consulta, de modo que se fuerce siempre un fallo en la caché y se acceda al recurso externo; otra solución sería realizar un acceso directo al recurso externo y evitar la caché en los casos en que sospechemos que pueden producirse actualizaciones concurrentes. Cualquiera sea la solución, necesitamos algún mecanismo para controlar este problema.

¿Qué sucede en el caso anterior si nuestra caché es distribuida?
Otro problema importante es el que sucede cuando nuestra caché está distribuida en diferentes servidores.
Las cachés distribuidas no sólo tienen la carga asociada a la búsqueda de un objeto dentro de sus estructuras internas, sino que es necesario tener en cuenta también el tiempo empleado para sincronizar los diferentes nodos de la caché en las diferentes operaciones realizadas con la misma. Analizar el funcionamiento de las cachés distribuidas no es alcance de esta tesis, pero es necesario comprender que el costo es mucho mayor que el presente en una caché con un único nodo. Los sistemas de caché distribuida pueden ser muy variados. Podemos tener una caché replicada en todos los nodos, en cuyo caso tendríamos un costo asociado de sincronización de los datos; podemos tener una caché repartida entre los nodos, en cuyo caso tendríamos un costo asociado a la petición de objetos a otros nodos; o quizás simplemente cachés independientes que se sincronicen periódicamente con una base de datos común. Sea como sea, tenemos asociados una serie de costos de comunicación que antes no estaban presentes.
El peor costo al trabajar con las cachés distribuidas es el asociado a las actualizaciones.

En muchos casos, cada actualización realizada en un nodo, implica el comunicarse con el resto de los nodos para que actualicen sus estructuras de datos. Evidentemente, esta actualización no será trivial, y traerá asociada una serie de transacciones entre los diferentes nodos. Resumiendo, habrá una serie de operaciones que incrementan considerablemente la latencia de nuestra aplicación.
Como se puede apreciar, las cachés distribuidas tienen una serie de problemas graves, tanto si son redundantes como si no, que pueden ocasionar efectos contraproducentes en nuestro sistema. Muchas personas, podrían creer que simplemente por el hecho de implantar una caché distribuida en su sistema, todo va a ir más rápido. Sin embargo, si implantamos una caché en un sistema donde predominan las actualizaciones, seguramente nos encontraremos con el efecto contrario, y la latencia originada por dichas actualizaciones hará que el rendimiento no sólo no sea mayor, sino que se vea gravemente afectado.

¿Qué sucede si el usuario que ha modificado el objeto A lo ha hecho desde una aplicación que no usa la caché?
Nuevamente nos encontramos frente al problema de las actualizaciones remotas. En el caso de que el segundo usuario no utilice una caché, los problemas que surgen son muy similares a los que se plantearon en el primer caso. Aquí, ya no podemos compartir la caché entre los dos usuarios, porque las aplicaciones son diferentes, y probablemente hasta estén escritas en diferentes lenguajes o funcionen sobre diferentes plataformas.
En el caso de que aplicaciones diferentes, y que no controlamos, puedan acceder a los datos que hemos almacenado en la caché, la cosa se agrava, hasta el punto de que no se recomienda utilizar cachés para acceder a este tipo de datos compartidos. Por ejemplo, supongamos que tenemos una caché que accede a algún listado histórico de datos que se suponía que nunca cambiaban, pero de pronto se instala una nueva aplicación que permite la modificación a mes vencido de dichos datos que suponíamos eran invariables. A partir de ese momento nuestra aplicación pasará a ofrecer datos potencialmente erróneos, con los graves problemas que esto puede acarrear.
Básicamente, debemos recordar la regla de que si no somos dueños de los datos, entonces es mejor no guardarlos en la caché. Guardar en caché datos que no controlamos puede ser muy peligroso, por mucha confianza que tengamos en que no cambien.
Existen distintas alternativas de cómo utilizar una caché en una aplicación:

- Caché programática. En este caso la caché es administrada de manera explícita por la aplicación. En lugar de dejar al motor ORM que trabaje con la caché según la estrategia elegida y que esto sea transparente a la aplicación, es la aplicación la que decide dónde y cuándo agregar, eliminar o modificar los objetos de la caché o bien sincronizar los objetos de la caché con la base de datos. La ventaja de esta alternativa es que la aplicación tiene control absoluto sobre la caché. La desventaja es que implementar esta lógica es una tarea difícil y costosa.
- Caché de objetos. En la caché de objetos se almacenan los objetos o entidades de dominio. Cuando ocurre un hit en la caché de objetos, no es necesario realizar ningún proceso de conversión o hidratación, el motor ORM simplemente retorna el objeto de la caché. La ventaja de esta alternativa es que los datos se almacenan en el mismo formato con el que son usados en la aplicación, con lo cual se reduce la hidratación de los objetos. La caché de objetos se utiliza para almacenar objetos muy pesados, es decir objetos que tienen muchas relaciones a otros objetos y que son conocidos como “tablas de referencias”. Por ejemplo supongamos que el objeto Persona tiene asociado el objeto País al cual pertenece. País es una tabla de referencia ya que tiene un conjunto finito de valores y bastante reducido.

- Caché de relaciones. En la caché de relaciones se almacenan solo las composiciones de los objetos de dominio. Esta caché es de utilidad para relaciones de uno-a-muchos y de muchos-a-muchos. Las relaciones de uno-a-uno y muchos-a-uno generalmente no necesitan ser almacenados en esta caché porque referencian el ID de los objetos. En la caché de relaciones se almacenan las referencias a los objetos, no los objetos en si mismos. Por ejemplo, un objeto País tiene asociada una colección de objetos Provincia. En esta caché se almacenan las asociaciones del país con sus provincias, no se almacenan las provincias ni el país. Cuando ocurre un hit en la caché de relaciones, retorna la referencia del objeto. Con la identificación del objeto, el motor lo recupera utilizando primero la caché de objetos y si no se encuentra, accede a la base de datos. Las ventajas de esta alternativa es que se reduce las consultas SQL ya que se evitan los joins necesarios para traer los objetos relacionados y se reducen las hidrataciones de las relaciones. La caché de relaciones se utiliza para almacenar relaciones que cambian con poca frecuencia.
- Caché de consultas parametrizadas. Se almacenan en la caché las consultas parametrizadas y precompiladas que el motor ORM utiliza para recuperar los objetos. Esta caché es de utilidad para aquellas consultas que se ejecutan muchas veces con la misma estructura pero parámetros diferentes. Estas consultas se compilan una sola vez y se guardan en caché para ejecutarse posteriormente con otros valores en sus parámetros. Las sentencias SQL utilizadas para guardar, recuperar, borrar y actualizar (operaciones CRUD) los objetos de dominio de la aplicación que se deducen de los mapeos son consultas candidatas a guardarse en esta caché. Cuando ocurre un hit en la caché de consultas parametrizadas, se retorna la consulta para su posterior ejecución con un conjunto de parámetros definidos por la aplicación. Luego se envía la consulta a la base de datos y esta retorna las referencias de los objetos resultado y con estas referencias recupera los objetos utilizando la caché de objetos y de relaciones o accediendo a la base de datos en el peor de los casos. Las ventajas de esta alternativa son que se reduce el tiempo de generación de las sentencias SQL ya que se compilan una única vez aprovechando la potencia de algunos motores de bases de datos que optimizan las consultas al compilarlas y se guardan precompiladas para sus posteriores ejecuciones. Los accesos a la base de datos no se evitan, ya que son necesarios para ejecutar la consulta.
- Caché de consultas. La caché de consultas almacena el resultado de una consulta en lugar de los objetos en si mismo. En este caso se guarda la consulta ejecutada junto con sus parámetros y los resultados obtenidos al ejecutarla. Cuando se ejecuta la consulta por primera vez, va a buscar los objetos resultado a la caché de objetos y de relaciones, si no los encuentra accede a la base de datos y guarda la consulta junto con el resultado en la caché de consultas. Durante las próximas ejecuciones de la misma consulta, el motor ORM verifica si la consulta ya se ejecutó anteriormente y retorna el resultado de la caché. La ventaja de esta estrategia es que elimina los accesos a la base de datos. Esta estrategia sirve para aquellas consultas que se ejecutan en la aplicación con mucha frecuencia y con los mismos parámetros. El principal problema que presenta la caché de consultas, al igual que con el almacenamiento en caché en general, son los datos obsoletos. La caché de consultas normalmente interactúa con una caché de objetos para asegurar que los objetos al menos estén sincronizados con los de la caché de objetos.

Vistas SQL

Cuando trabajamos en contextos de aplicaciones de tipo enterprise, es decir aplicaciones que afectan a muchos recursos de la organización, es común encontrarnos con situaciones donde la base de datos a utilizar ya existe y ya se disponen de datos cargados. En estos casos se debe tomar la decisión de utilizar la base de datos existente o de diseñar un nuevo modelo de datos derivado del modelo de objetos.
En otros casos existen organizaciones en donde si bien las bases de datos no existen, se presentan dificultades (muchas veces políticas) al momento de definir cual es el modelo de datos que la aplicación utilizará. Estos contextos se presentan en organizaciones en donde existen áreas de DBA o bases de datos muy fuertes o que fueron o son áreas estructurales para los departamentos de sistemas.
En cualquier caso se deben elegir diferentes alternativas de abordaje:

- Tomar el diseño de base de datos y en base a eso definir el modelo de objetos.
- Diseñar el modelo de objetos y crear la base de datos que corresponda con este modelo.

Normalmente en el primero de los casos estamos frente a los problemas mencionados en las diferencias de impedancia, es decir que el modelo de base de datos que tenemos diseñado no es el adecuado para el modelo de objetos deseable en la aplicación. Normalmente ante estos casos se puede aplicar diferentes soluciones:

- Generar un modelo de objetos que mapee directamente a las tablas existentes. El modelo de objetos no será el ideal pero respetaremos el modelo de datos existente o diseñado.
- Generar un modelo de objetos “ideal” que si bien no se puede mapear directamente a las tablas del modelo de datos, podremos mapearlo utilizando modelos complementarios.

En el primer caso estamos frente a una decisión de compromiso que no representa el ideal de los casos en aplicaciones que utilicen tecnologías orientadas a objetos. Si bien es normal que se presenten el resultado de esta técnica de abordaje se debe más a decisiones que escapan a la tecnología y están basadas en temas políticos del proyecto o la organización.
En el segundo caso estamos frente a la dificultad de unir dos mundos (objeto y relacional) que son diferencias entre sí, y tenemos que hacerlos hablar el mismo lenguaje. Para esto es posible utilizar herramientas intermedias como pueden ser modelos de objetos intermedios o recursos de las bases de datos que transformen el modelo existente en un modelo más acorde a nuestro modelo de objetos.
Por ejemplo en muchos casos es normal encontrarse en bases de datos legacy con relaciones uno-a-uno entre diferentes tablas. Esto no representa puntualmente un problema o un error de diseño, pero normalmente este tipo de relaciones desaparecen cuando las bases de datos son creadas como consecuencia de un modelo de objetos. Si nuestro modelo de objetos consiste en una clase Cliente que tiene una relación con una clase DatosPersonales, es muy probable que queramos mapearla a una sola tabla. En ciertas organizaciones como la Banca o Telefonía es normal encontrar estos “datos” en dos tablas distintas.
Ante esta situación podemos crear una vista que presente la información de estas dos tablas como si fuera una sola de modo que nuestro mapeador deba acceder solo a una tabla, reduciendo la complejidad de las consultas y mejorando la performance (menos joins = menos tiempo).
Las vistas también pueden ser utilizadas en el caso de que se requiera recuperar una gran cantidad de objetos en donde los filtros de búsquedas sean muy complejos de definir. Es conocido que los motores de bases de datos pueden resolver de manera más eficiente la unión (joins) y filtrado de datos (where) de manera interna que cuando se los

solicita puntualmente. Es por eso que si nuestra aplicación sabe que debe realizar una consulta que involucra muchas tablas y con parámetros de filtrado predefinido, es conveniente crear vistas que representen esta consulta, de modo que la consulta generada en la aplicación sea mas simple y el motor puede optimizar de manera interna el acceso a los datos.
La utilización de vistas no debe tomarse como un patrón de solución habitual y se debe tender a ser utilizada solo para consultas de lectura, dado que no todos los motores soportan vistas de escritura y en los casos en donde se requiere, su eficiencia no es óptima.
Las vistas puede generar reducciones en la complejidad del modelo de objetos, las consultas a nivel objeto (por ejemplo HQL) y la complejidad de los mapeos, pero también son indicadores de que es posible que el modelo de datos con el que se está trabajando no sea el adecuado para todos los casos.

Procedimientos almacenados

Un punto controversial en aplicaciones que utilizan mapeadores objeto/relacional es el uso de procedimientos almacenados. Cuando en la industria se comenzaron a utilizar los ORM, los equipos que promovían su utilización encontraban gran resistencia por parte de las organizaciones que tendencias y costumbre de uso de procedimientos almacenados para la construcción de aplicaciones.
Tradicionalmente las organizaciones utilizaron procedimientos almacenados debido a dos grandes motivos:

- La plataforma y lenguajes de programación utilizados fomentaban su utilización y facilitaban el trabajo, es decir que en base a la tecnología con la que se construían las aplicaciones era más “fácil” escribir procedimientos almacenados que hacer que las aplicaciones ejecutaran las consultas SQL y las estructuras de control contenidas en los procedimientos almacenados.
- Por otro lado, en otros casos se promovió el uso de procedimientos almacenados por su seguridad en el tratamiento de transacciones y por la performance. En estos casos se utilizaban procedimientos almacenados para modelar los procesos de negocio core de la organización, aquellos en donde el tratamiento de la información era crítico, donde se requerían una alta performance o donde las transacciones eran demasiado largas o complejas.

Con la introducción de los ORM los equipos de desarrollo tomaron la tendencia de “eliminar” el uso de procedimientos almacenados para incluir toda la lógica contenida en esos procedimientos dentro de las nuevas aplicaciones orientadas a objetos. La justificación de esta estrategia estuvo fuertemente basada en las ventajas que promueven las tecnologías orientadas a objeto.
Si bien es cierto que en general es conveniente tender a la no utilización de procedimientos almacenados para las taras comunes de las aplicaciones como consultas y acceso a datos simples de información, hay otros casos en donde puede ser conveniente utilizarlos.
El caso más común es cuando una aplicación debe utilizar bases de datos ya existentes que contienen procedimientos almacenados con procesos de negocio muy complejos ya implementados. En este caso se debe medir la relación costo/beneficio de reutilizar los procedimientos almacenados y seguir manteniéndolos, contra el costo de re implementar la lógica en objetos y mantenerla en la nueva tecnología. Esta evaluación es muy puntual para cada caso y depende mucho del tipo de organización. Pero como regla general se puede decir que en organizaciones en donde existen equipos fuertes con alto conocimiento del negocio y de tecnologías de procedimientos almacenados es

conveniente la reutilización de estos procedimientos y no la re implementación de los mismos.
Otro caso es cuando, si bien se está utilizando una tecnología orientada a objetos, y la norma de la aplicación indica que “todo” debe ser implementado en objetos, se determina para ciertos casos que es conveniente utilizar procedimientos almacenados. Un claro ejemplo de este caso es la actualización masiva de datos que no puede resolverse con una simple consulta SQL.
Veamos un ejemplo. Supongamos que nuestra aplicación resuelve la problemática de gestión de cobranzas y cuentas corrientes de una empresa. En cierto proceso de negocio se requiere determinar cuales clientes tienen mora (es decir disponen de facturas impagas) y en caso de que así sea, se deberán copiar cada una de las facturas impagas de una tabla a otra. Si bien este procesamiento puede ser resuelto a nivel aplicación en objetos, pude resultar demasiado costoso, dado que la aplicación deberá recuperar todos los clientes, recuperar las facturas impagas de cada cliente y realizar un insert por cada factura impaga en la segunda tabla. Esto genera demasiado tráfico en la red y por consiguiente problemas de performance. Una estrategia de solución para este caso puede ser la implementación de todo el proceso en un procedimiento almacenado.
La contra que tiene la utilización de procedimientos almacenados es que todo lo que estos resuelvan queda fuera del alcance de la aplicación y el mapeador. Por ejemplo si la aplicación está utilizando algún caché o motor de indexación de objetos y el procedimiento almacenado modifica algún dato, se podrán generar inconsistencias.
Las ventajas radican en que para casos de procesamientos de grandes volúmenes de información en donde se requieran performance, los procedimientos almacenados en general proveen mejores prestaciones, dado que puede generan menor tráfico de información (entre la aplicación y la base de datos) y gestionan las conexiones y transacciones de manera interna en el motor.

Serialización

Cuando se trabaja con mapeadores objetos/relacional, la estrategia tradicional de utilización implica que todo atributo de todo objeto se corresponderá con alguna columna de alguna tabla. Como regla general se intenta que todos los objetos tengan mapeada su estructura completa al modelo de datos correspondiente, almacenando en la base de datos los valores de los atributos como atributos de cada una de las tablas, es decir llegando a la granularidad mas baja posible.
Esta estrategia es la adecuada para la gran mayoría de los casos dado que nos permite poder acceder a los valores de los atributos de manera directa por ejemplo para realizar consultas y búsquedas de objetos. En esta estrategia, si un atributo de un objeto no está representado como una columna de alguna tabla no seremos capaces de realizar búsquedas de los objetos de acuerdo al valor de este atributo.
Pero también es cierto que algunos objetos contienen información que no es necesaria para realizar consultas, es decir hay algunos casos en donde no se justifica almacenar todo su estado interno como columnas de tablas. Estos casos en general se presentan con objetos que no son el root object de algún modelo.
Por ejemplo, supongamos que nuestro modelo de objetos consiste en una clase Computadora y que esta clase tiene relaciones con un objeto Procesador que modela las características y prestaciones del chip del procesador. Dadas las características de la aplicación el objeto Procesador no es utilizado para realizar consultas y toda la información que contiene es utilizada por el modelo de objetos para resolver el negocio propio de la aplicación (por ejemplo darle prioridad a un proceso o calcular los Mhz disponibles).

Una estrategia que puede utilizarse en este caso es que el objeto Procesador sea serializado utilizando el API nativa del lenguaje de programación utilizado (por ejemplo Java), dando como resultado un stream de bytes, y luego indicamos al ORM que almacene el objeto como un stream de bytes. Es decir que para este objeto el mapeador no realiza la conversión de atributos a columnas y tablas sino que serializa el objeto entero en un arreglo de bytes y almacena en la base de datos el arreglo de bytes como un dato único. Para la lectura simplemente realiza el proceso inverso.
Las desventajas de esta estrategia son básicamente dos: en primer lugar no seremos capaces de realizar consultas al ORM basadas en el estado interno de la clase Procesador, y por otro lado podremos tener problemas a la hora de realizar cambios en la clase Procesador si en la base de datos se encuentran almacenados objetos con versiones anteriores de la clase.
La ventaja es que la base de datos se mantiene simple, almacenando en una columna todo un objeto entero y por consiguiente el mapeo también se mantiene relativamente simple. Esta estrategia es recomendable solo en casos en donde el grafo de objetos sea demasiado complejo, generando un modelo de datos muy complejo y donde los datos del estado interno de estos objetos no sean utilizados para ninguna consulta.

Motores de indexación de objetos

El poder de los índices para acceder a los datos almacenados en una base de datos son bien conocidos, los motores utilizan índices para optimizar el acceso a los datos de acuerdo a sus atributos. Cuando se trabaja con ORM los “datos” que conforman a los objetos son almacenados en bases de datos de modo que los índices de las bases de datos son utilizados cuando se consultan. Estas consultas son derivadas a partir de consultas de más alto nivel, generadas en lenguajes ORM (como Criteria, HQL, o JPQL) , quienes toman el modelo de consulta en objetos y generan consultas en SQL que luego son interpretadas por el motor de base de datos.
El motor luego procesa esa consulta SQL y retorna toda la información que la consulta determina, para que luego el motor “hidrate” los objetos con los datos retornados.
Cuando se trabaja con grandes volúmenes de datos, el mayor costo en este procedimiento se encuentra en el procesamiento de la consulta por parte del motor y en el posterior proceso de creación de los objetos por el ORM.
Si nos centramos específicamente en consultas de solo lectura, y mas específicamente consultas utilizadas para “buscar” objetos, puede que el SQL que se generen en la base de datos sea muy complejo para su procesamiento, que involucre una gran cantidad de tablas y un gran volumen de datos.
Ejemplo, supongamos que nuestro modelo de objetos consta de una clase Provincia que tiene una lista de Ciudad, además la clase Provincia tiene relaciones con el País al que pertenece y una relación con una clase EstadoFinanciero que si bien contiene atributos con datos, su mayor valor es que contiene métodos que pueden retornar el PBI (el PBI no es un atributo sino que es el resultado de un método implementado dentro de la clase).
Que características tiene nuestro modelo:

- El grafo de objetos es complejo, es decir esta formado por muchas clases y por consiguiente por muchos datos.
- El modelo de datos necesario para soportar este modelo de objetos esta formado por muchas tablas
- Formar el modelo de Provincia completo implica realizar joins entre múltiples tablas.

Supongamos que nuestra aplicación requiere buscar todos los países que cumplan con las siguientes condiciones:

- La cantidad de ciudades es mayor a 5
- El nombre del presidente del país al que pertenece comienza con “M”
- El PBI de la provincia es mayor a USD10.000.000.000

El ejemplo sirve para ilustrar una situación compleja, pero independientemente de esto, ante cualquier consulta de tipo “búsqueda” cuando utilizamos ORM se deben resolver dos grandes aspectos:

- Determinar cuales son los objetos que cumplen con las condiciones
- Recuperar todos los datos necesarios para recrear el modelo de objetos completo

Sin importar la complejidad y el costo de ejecución Los primeros dos puntos pueden resolverse utilizando SQL, si bien la complejidad de las consultas es muy alta y el costo de su ejecución también, podemos resolverlo. El tercer punto no podrá ser resuelto a nivel SQL, dado que el dato no existe en la base de datos sino que es el resultado del negocio modelado por el objeto.

Este es un ejemplo en donde el problema de “determinar cuales son los objetos que cumplen con las condiciones” es demasiado costoso de resolver a nivel SQL. Entonces, que pasaría si utilizáramos otras “herramienta” que le permita al ORM:

- Determinar cuales son todos los objetos que cumplen con ciertas condiciones sin tener la necesidad de acceder a la base de datos.
- Entregar todos los identificadores de los objetos que cumplan con esas condiciones.
- Dejar que el mapeador recupere los objetos de la base de datos utilizando directamente el ID.

Para poder implementar esta estrategia de solución se pueden utilizar motores de indexación de objetos (por ejemplo Solr, Lucene, etc). Estos motores permiten almacenar información que luego va a ser utilizada para realizar búsquedas de los objetos.

Tomando como base el modelo planteado anteriormente podríamos almacenar en el motor de indexación una estructura como:

<idProvincia, nombrePresidente, cantidadCiudades, PBI>

La contra de esta solución es que ante cualquier actualización del objeto Provincia (o su grafo) debemos notificar al motor de indexación para que actualice el índice, es decir que debemos ser responsables de mantener actualizado el índice.

La ventaja es que la búsqueda de información será mucho mas performante que realizarla a nivel SQL, dado que la estructura del índice estará específicamente diseñada para el tipo de consultas que queremos realizar, y (para estos casos) delegaremos en el motor solo la recuperación de los datos por el ID, dado que el motor nos ayudó a reducir la complejidad y el costo de filtrar los objetos de la consulta en base a su estructura interna.

Repositorios Online/Offiline

Las aplicaciones que trabajan con grandes volúmenes de información pueden agruparse en dos grandes tipos de acuerdo a la utilización que hacen de cada uno de los datos según el estado temporal de los datos:

- Uso unificado de todos los datos, son las aplicaciones que utilizan todos los datos (o la mayoría de estos) en todo momento, es decir que todos los datos son accedidos independientemente de la edad del dato, no importa si el dato es muy nuevo o muy viejo, la aplicación los va a acceder con la misma periodicidad independientemente de esto.
- Uso segmentado temporalmente de los datos, son las aplicaciones que tienen tendencia a consumir los datos “nuevos” más habitualmente que los datos más viejos. En este tipo de aplicación se puede realizar un segmentado horizontal de

los datos para determinar que datos son más factibles de acceder que otros.

En este sentido cabe aclarar que con mas nuevo o mas viejo no nos referimos a la cantidad de tiempo que tiene el dato dentro del repositorio de datos, sino que el concepto depende de cada aplicación, puede ser que el concepto de viejo signifique que el dato pertenece a un trámite ya finalizado, y el trámite puede haber finalizado hoy, y en contrapartida puede que un trámite iniciado hace 2 años y que no este finalizado, para la aplicación es más nuevo que el anterior.

El concepto de repositorio de datos online/offline implica una serie de cuestiones y nuevos conceptos o desafíos que una aplicación debe incorporar y resolver.

La infraestructura física de los datos deberá estar física o virtualmente particionada, es decir que se deberá poder determinar a nivel base de datos si un dato es considerado viejo o nuevo. Los datos viejos son considerados datos dentro del repositorio offline de la aplicación, es decir que son datos que no estarán disponibles para el trabajo normal de la aplicación, la operatoria del día a día. Los datos del repositorio online son datos con los que trabajará la aplicación durante la operatoria normal. El objetivo de está estrategia es reducir la cantidad de datos con los que trabaja la aplicación.

Esta segmentación puede realizarse de dos grandes formas:

- Físicamente: los datos se segmentan horizontalmente y se almacenan en dos bases de datos diferentes, una con los datos online (con los que opera normalmente la aplicación) y otras con los datos offline (con los que opera en modo offline la aplicación). Esta segmentación no es lo mismo que la segmentación horizontal de las bases de datos. Se trata de mantener dos bases de datos diferentes.
- Virtualmente: se especifica en cada dato si es nuevo o viejo, es decir si debe ser incluido en el modo online y modo offline. Esta táctica puede ser implementada tan simplemente como con una columna que especifique este atributo. También puede combinarse con técnicas de vistas para crear una vista que solo contengan los datos online de la aplicación.

En primer lugar la aplicación podrá contar con dos modos de operación:

- Modo online: En este modo la aplicación trabajará solo con los datos “nuevos” o los datos que son utilizados para la operatoria de los “trámites” actuales del negocio. En este modo la aplicación no sabrá de la existencia de información histórica o de trámites terminados. Bajo esta modalidad el volumen de datos con los que la aplicación trabaja se reduce solo a los trámites activos.
- Modo offline: Este modo es el inverso al anterior, bajo este modo la aplicación no conocerá en absoluto los datos actuales, es decir los datos nuevos o los trámites en curso. Esta modalidad la aplicación accede a información “vieja” o de trámites terminados. Este modo suele ser utilizado para consultar información y ante esta situación la aplicación se enfrenta a trabajar con grandes volúmenes de datos.

Un contexto de uso muy habitual de estos modos es en el contexto de “Switching transaccional” relacionado con las empresas dedicadas al procesamiento de transacciones de pago electrónico. Las aplicaciones que realizan el procesamiento de cada transacción electrónica que entra al circuito, no necesitan conocer información histórica de las transacciones históricas procesadas, solo necesitan conocer información relacionadas con transacciones procesadas durante un período de tiempo muy corto. Por ejemplo para el caso de realizar cancelaciones de transacciones, en general las cancelaciones se dan en un marco temporal acotado (no mayor a un par de horas). En este contexto las transacciones que tengan una edad mayor a 24 horas son “pasadas” a un repositorio offline y las nuevas transacciones son almacenadas en un repositorio “online”. De modo que todo el procedimiento de autorización y procesamiento de transacciones (la aplicación) trabaja solo con los datos de las transacciones de las últimas

24 horas. Las anteriores simplemente no le sirven.

Batching Update

Toda aplicación que utiliza base de datos y que requiera realizar una operación con datos almacenados en ésta, debe realizar una serie de pasos básicos y obligatorios:

1. Obtener una conexión a la base de datos
2. Abrir una transacción
3. Enviar una a una las consultas SQL para modificar los datos
4. Cerrar la transacción (commit o rollback)
5. Cerrar la conexión a la base de datos

El paso 2 es, en general el que presenta problemas de performance cuando la cantidad de acciones que se deben realizar son muchas.
Supongamos que nuestra aplicación no utiliza un ORM y necesita realizar una operación que implica insertar dos registros en dos tablas distintas. La aplicación realizará las siguientes operaciones:

1. Obtener una conexión/transacción
2. Enviar el insert en la tabla 1
3. Enviar el insert en la tabla 2
4. Cerrar la conexión/transacción

Los pasos 2 y 3 si bien están ejecutados dentro de la misma conexión y transacción representan dos paquetes diferentes de datos, es decir que el motor los procesará de manera independiente (dentro de la misma transacción).
Una técnica que puede utilizarse con algunos motores de bases de datos es enviar las consultas como un solo paquete de consultas, de modo que el motor tenga conocimiento de todas las consultas (o gran parte de las consultas) que deben realizarse para completar la operación. El motor planificará estas consultas y las ejecutará como un solo bloque, de modo que lo que pasará realmente será:

1. Obtener una conexión/transacción
2. Enviar el insert en la tabla 1 y el insert en la tabla 2 (como un bloque único)
3. Cerrar la conexión/transacción

En este ejemplo, el paso 2 se denomina batching update, es el procedimiento mediante el cual le enviamos al motor varias consultas juntas y no una a la vez, reduciendo el tráfico de red entre la aplicación y el motor y permitiendo que el motor pueda tener mayor información al momento de planificar la ejecución de las consultas.
Los motores ORM permiten indicar que las actualizaciones de un objeto se realicen utilizando esta técnica, e incluso permiten que se utilice esta técnica para todas las interacciones que el ORM tenga con la base de datos dentro de una misma transacción.
En algunos motores podemos incluso indicar la cantidad de consultas que debe enviar al motor de base de datos como un bloque “batch”, por ejemplo le podemos indicar que “intente” enviar bloques de 10 consultas, por lo que el ORM intentará demorar el envío de las consultas hasta juntar 10 o hasta que se realice un cierre de conexión o commit de transacción.
Por ejemplo, podemos indicar al ORM que cuando realice una actualización del objeto Provincia (y la cascada de todas sus dependencias como Ciudad, País, etc) las consultas correspondientes las envíe en modo batch, es decir como un solo bloque de consultas al motor.
La ventaja de esto es que proveemos de mayor información al motor para planificar la ejecución de estas consultas y reducimos la cantidad de bloques de información que fluye entre la aplicación y la base de datos. La desventaja radica es que se suma mayor complejidad y costo al motor para recolectar las consultas y enviarlas al motor, además de

que los paquetes de información que se envían al motor si bien son menos en cantidad, son mayor en tamaño.

Desnormalización

Cuando se utilizan bases de datos para almacenar información siempre es deseable que el modelo de datos se encuentre normalizado. Si bien esto presenta grandes beneficios desde el punto de vista del diseño, en ciertos casos pueden presentar algunas dificultades a la hora de obtener buena performance en la aplicación.
Para comprender el concepto de desnormalización como estrategia de optimización de performance vamos a tomar como ejemplo el caso de las jerarquías de clases. Existen distintas alternativas tanto para el mapeo de jerarquías como para relaciones y cada una de ellas presenta ventajas y desventajas. Teniendo en cuenta las características de la aplicación y analizando los pros y contras de cada alternativa, se puede proveer mejoras de performance en el acceso a los datos, simplemente cambiando los mapeos.
Por ejemplo, supongamos que en nuestra aplicación tenemos una jerarquía de objetos donde tomamos la decisión de utilizar la estrategia de una tabla por clase para mapear esta jerarquía. Testeando la aplicación nos damos cuenta que recuperar todos los objetos de la jerarquía es demasiado lento.
Por ejemplo, nuestro modelo de objetos consta de una jerarquía representada por tres clases, la clase A (como clase abstracta) y las clases B y C como subclases. Analizamos las siguientes estrategias de mapeo:

- Normalizado: Crearemos una tabla para cada clase concreta más una tabla para la clase abstracta. En la tabla de la clase abstracta almacenaremos los datos comunes de todas las clases y en cada una de las tablas B y C almacenaremos los datos específicos de cada clase.
- Desnormalizado: Crearemos solo una tabla para almacenar todas las instancias de cualquiera de las clases de la jerarquía. En esta estrategia la tabla creada tendrá una gran cantidad de columnas, contendrá la suma de los atributos de A+B+C y posiblemente algun atributo de tipo “discriminador”. Muchos de ellos estarán vacíos o inutilizados dependiendo de la clase que represente el registro. Pero todos los datos estarán almacenados en una sola tabla.

Si bien la primera estrategia representa ventajas obvias desde el punto de vista del diseño y del mantenimiento de los datos, presenta algunas dificultades de performance a la hora de acceder a la información. Para recuperar una objeto de la clase C se requiere realizar un join entre las tablas A y C. Tomando la misma estrategia si se requieren recuperar todas las instancias de cualquier clase de la jerarquía, inevitablemente se deberán realizar dos consultas SQL, A join B y A join C.
Por el otro lado, utilizando la estrategia desnormalizada, perdemos las ventajas desde el punto de vista de diseño pero todos los accesos a datos pueden ser más performante, al menos en cuanto a la complejidad de las consultas realizadas y a la cantidad de tablas involucradas.
Es importante tener presente que algunos cambios en las estrategias de mapeo puede presentar cambios en el modelo de objetos. El ejemplo expuesto con las diferentes estrategias de mapeo de jerarquías es un ejemplo en el que el modelo de objetos no se altera pero si se altera el modelo de datos.

Conclusiones

Como pudimos analizar en los capítulos anteriores, las aplicaciones que utilizan ORM se enfrentan a diferentes tipos de problemas, estos problemas dependerán de diferentes

factores tanto técnicos como de contexto (los equipos de trabajo y las costumbres de la organización).
La determinación de las estrategias de optimización elegida dependerá de estos dos grandes factores, y se deberá analizar puntualmente cada caso para determinar la factibilidad en la aplicación de las estrategias elegidas.
Algunos equipos de trabajo pueden no estar preparados para implementar determinadas estrategias de solución, esta situación puede darse por resistencias al cambio o simplemente porque no se cuenta con el conocimiento suficiente para implementar la estrategia, por ejemplo equipos en donde no se encuentran los skill necesarios para implementar herramientas de caché distribuidos en cluster o programación de procedimientos almacenados.
Del mismo modo las mismas estrategias pueden no ser implementables debido a trabas o barreras técnicas, por ejemplo en ciertos casos los motores de bases de datos utilizados soportan procedimientos almacenados o el ORM no provee soporte para implementar estrategias de lazy loading, y el costo de implementarlo inhouse es demasiado alto.
Cada estrategia planteada en este capítulo es aplicable en diferentes contextos y situaciones y provee ventajas y desventajas. Lo cierto es que ante la presencia de desafíos que involucren aplicaciones enterprise, grandes volúmenes de datos y ORM no se debe perder de vista que será inevitable encontrarse con múltiples problemas y que deberán ser aplicadas diferentes estrategias de optimización.
Como conclusiones general podemos decir que, en el contexto global de aplicaciones enterprise no es factible la utilización de ORM al estilo “out of the box”, sin implementar algunas de las técnicas de optimización aquí planteadas.

Capítulo VII - Caso de estudio

Introducción

Sancionada la Ley Nº 24.156 de Administración Financiera y de los Sistemas de Control del Sector Público Nacional el 30 de septiembre de 1992 se desarrolla un sistema integrado de información financiera (SIDIF) cuyo principal propósito es la formulación del presupuesto nacional y registro de la ejecución presupuestaria.
El SIDIF fue concebido como un sistema integrado, compuesto por diversos subsistemas y módulos dentro de una visión funcional, que contempla la distribución de la base de datos lógica, en una base de datos central y tantas bases institucionales como SAF existentes (Servicios de Administración Financiera).
Bajo este esquema operativo, la Secretaría de Hacienda se comunicaba, a través del SIDIF Central con los sistemas periféricos instalados en los organismos (sistemas locales), con el sistema de gestión para la Unidades Ejecutoras de Préstamos Externos y demás aplicaciones que interactúan con la base de datos central. A través de esta comunicación la base central concentra el registro de la ejecución presupuestaria. La información de gestión permanece en las bases locales.
Dentro del SIDIF convivían una variedad de sistemas locales en las distintas entidades con características diferentes. Esta diversidad de aplicativos incrementó el costo de mantenimiento y de replicación de las adecuaciones en los sistemas locales, lo que implicaba en algunas oportunidades demoras para su disponibilidad.
Ante esta situación se da comienzo a una etapa de transformación que consiste en un proceso de homogeneización de los sistemas locales mediante la implementación de un nuevo sistema SIDIF Local Unificado.
Este proceso, que aún se está llevando a cabo, implica reemplazar los sistemas locales existentes en distintos organismos por una versión unificada con mejoras en cuanto a la provisión de información confiable para la alta gerencia de las entidades, el ajuste de procedimientos de compras, presupuesto, contabilidad y tesorería y la reducción de aplicativos complementarios requeridos para soportar la gestión.
Habiendo cumplido esta primera etapa de transformación que ha introducido importantes mejoras a través del SLU, se plantea un nuevo desafío, la denominada segunda transformación que se materializa con la extensión del alcance y renovación tecnológica con el desarrollo del sistema e-Sidif.
Es este sistema el que busca mejorar la calidad del SIDIF incorporando las innovaciones y posibilidades que en materia de tecnología de comunicaciones se han producido en la última década, aprovechando la flexibilidad y reducción de costos que posibilita la modernización, introduciendo cambios que sirvan como base para una mayor orientación de la gestión pública a resultados y una ampliación del alcance y vinculación del sistema.
eSidif es un sistema enterprise y como tal, tiene la complejidad necesaria para que se presenten muchas situaciones en las cuales la performance del sistema se vea comprometida. Por esta razón eSidif es utilizado como caso de estudio de esta tesis.

Arquitectura eSidif

El estilo arquitectónico que presenta el caso de estudio, es el N layers. Esto significa, como se describió en capítulos anteriores, organizar el sistema en layers que se comunican para poder completar la funcionalidad requerida. Cada layer tiene una

funcionalidad y un alcance bien definidos.
La Figura 23 muestra la arquitectura esidif:

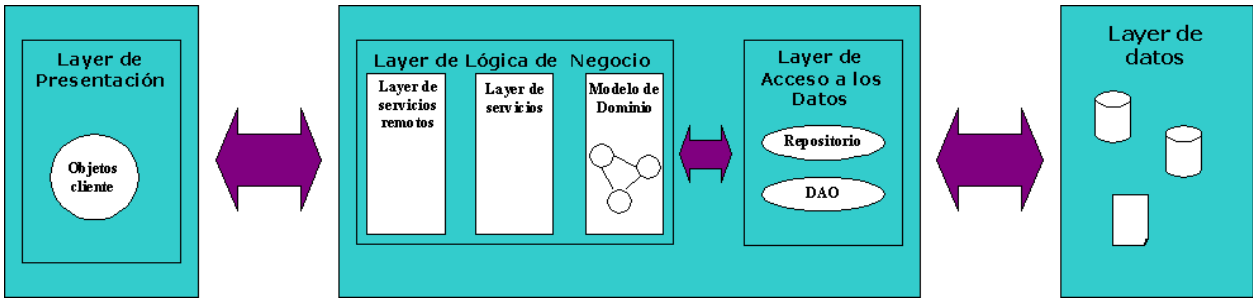


Figura 23: Arquitectura eSidif N-layer

A continuación se describen las responsabilidades de cada una de las capas:

- Layer de presentación: La lógica de presentación conforma parte de la "vista" o interfaz gráfica [GUI] de la aplicación y de cómo ésta organiza la interfaz de usuario conectándose con el layer de la lógica de negocios. El hecho de tener la presentación como un layer separado de la lógica de negocios permite desacoplar la lógica de negocios de la presentación permitiendo alterar la interfaz de la aplicación o agregar un nuevo canal sin necesidad de alterar la lógica de negocios.
- Layer de lógica de negocios: En esta capa se ejecutarán las reglas específicas del negocio. Recibirá peticiones desde el layer de presentación que deberá resolver aplicando las reglas que correspondan. Llevar a cabo la lógica implicará comunicarse con la capa de acceso a datos con el propósito de recuperar los objetos involucrados en las reglas que haya que aplicar y persistir los cambios realizados.
  Esta capa se divide en tres layers:
  - Modelo de dominio: Tiene la lógica de dominio de la aplicación. Usa el paradigma orientado a objetos para implementar las reglas del negocio.
  - Layer de servicios: Fachada sobre el modelo de dominio. Coordina una operación interactuando con la capa de acceso a datos y delegando en el modelo de dominio. Define las precondiciones que deben darse para que el servicio se ejecute.
  - Layer de servicios remotos: Mantiene toda la lógica de aplicación. En este caso la lógica de aplicación incluye exposición de los servicios para poder ser ejecutados remotamente, manejo de transaccionalidad, seguridad, jogging.
- Layer de acceso a datos: Es responsable de proveer acceso a los datos y objetos que necesita el layer de lógica de negocio para poder llevar a cabo su cometido y de llevar a cabo la persistencia de los cambios realizados por el layer de la lógica de dominio. Siendo que el medio de persistencia de la aplicación será una BD relacional, deberá encargarse del mapeo objeto/relacional.
- Layer de datos: Es responsable de almacenar los datos de la aplicación. El medio de persistencia principal de la aplicación será una base de datos relacional. Además existirán otras fuentes/destinos de datos, entre las cuales se encuentra el sistema legado.

Tecnologías utilizadas en e-Sidif

Se presentan los frameworks y tecnologías de la comunidad utilizados en las distintas capas arquitectónicas del sistema eSidif.

Tecnologías del Layer de Presentación

RICH CLIENT PLATFORM (RCP)
RCP es una plataforma de desarrollo de aplicaciones desktop que permite utilizar un grupo de artefactos que facilitan el desarrollo de software.
RCP está compuesto por las siguientes partes:

- Un CORE que contiene la base de la plataforma
- Un Framework estándar de desarrollo
- Un toolkit de widgets portables
- Editores de texto y manejadores de archivos y texto
- Un ambiente de ejecución (Workbench) que permite trabajar con editores, vistas, perspectivas y wizards
- Una herramienta de actualización automática

STANDARD WIDGET TOOLKIT (SWT)
SWT es una librería de elementos visuales que se utiliza sobre la plataforma Java. Fue desarrollado por IBM y actualmente es actualizado y mantenido por el grupo de fundación Eclipse. SWT es una alternativa dentro de las librerías visuales existentes tales como AWT y Swing provistas por Sun Microsystems y que son parte de la plataforma estándar de Java (JSE).
SWT se utiliza para la visualización de elementos de interfaz gráfica. Para esto, se accede a las librerías nativas del sistema operativo utilizando JNI (Interfaz Nativa de Java). Las aplicaciones que utilizan SWT son portables, sin embargo, si bien son herramientas desarrolladas en Java, su implementación es específica para cada plataforma. Este set de herramientas posee licencia pública de Eclipse y abierta.

Tecnologías del layer de acceso a datos

HIBERNATE
Hibernate es una herramienta de mapeo objeto/relacional para la plataforma Java que facilita el mapeo entre una base de datos relacional tradicional y el modelo de objetos de una aplicación, mediante archivos declarativos (XML) que permiten establecer estas relaciones.
Hibernate es open source y está distribuido bajo los términos de la licencia GNU LGPL.
La utilización de Hibernate facilita la solución al problema de la diferencia entre los dos modelos usados para organizar y manipular datos. Hibernate ofrece también dos lenguajes de consulta de datos:HQL (Hibernate Query Language)y Criteria, además de permitir la ejecución de consultas en SQL nativo.

Tecnologías del layer de lógica de dominio

SPRING
El Spring es una estructura de soporte para desarrollar aplicaciones sobre la plataforma Java.
El Framework Spring adopta un conjunto de características que mencionamos a continuación:

- Spring pone foco en brindar facilidades para el manejo de los objetos de negocio.
- Es modular. Está estructurado en componentes, los que pueden utilizarse por separado sin perder consistencia. Por ejemplo, se puede utilizar el Framework para simplificar los accesos de JDBC solamente o bien se puede utilizar para manejar todos los objetos de negocio.
- Basado en el patrón IOC para construir código fácil de testear.
- Posee una muy buena integración con distintas tecnologías.

Spring es una tecnología que permite construir aplicaciones utilizando POJOs que ayuda tanto al modelado estructural como al de comportamiento y las reglas de negocio a las que se enfrentan en el caso de estudio.
El modelo de dominio se representa mediante clases planas y simples que no dependen de ningún framework en especial, las cuales posteriormente mediante Hibernate son persistidos en una base de datos Oracle. Contienen además la lógica de negocio de la aplicación.

Tecnologías del layer de datos

ORACLE
Oracle es un sistema de gestión de base de datos relacional (o RDBMS por el acrónimo en inglés de Relational Data Base Management System), desarrollado por Oracle Corporation.
Se considera a Oracle como uno de los sistemas de bases de datos más completos, destacando su:

- Soporte de transacciones.
- Estabilidad.
- Escalabilidad.
- Multiplataforma.

Casos

En eSidif se presentan muchos casos concretos de algunos de los problemas de performance que se describieron en el capítulo cinco. A continuación veremos un subconjunto de estos casos y la aplicación de algunas de las estrategias de optimización planteadas.

Caso: Vistas SQL

El sistema e-Sidif provee varios listados y reportes que permiten visualizar y analizar la información almacenada y generada por el sistema. Los reportes se dividen en dos grandes categorías: operativos y gerenciales.
Los reportes operativos son aquellos que se utilizan periódicamente junto con el registro diario de la información. Estos reportes muestran información necesaria para llevar a cabo la gestión diaria. También sirven como respaldo y resumen del registro de datos realizado. El requerimiento principal que debe cumplir la aplicación es que los reportes operativos se generen de manera rápida y en tiempo real.
Los reportes gerenciales por su parte muestran información consolidada, la cual se obtiene de relacionar datos de diferentes negocios (subsistemas) y períodos de tiempo más grandes que los reportes operativos. Estos reportes son utilizados para análisis y posterior toma de decisiones. Los reportes gerenciales consultan grandes volúmenes y diversidad de datos, lo cual hace que el tiempo de generación sea elevado. Si bien, estos

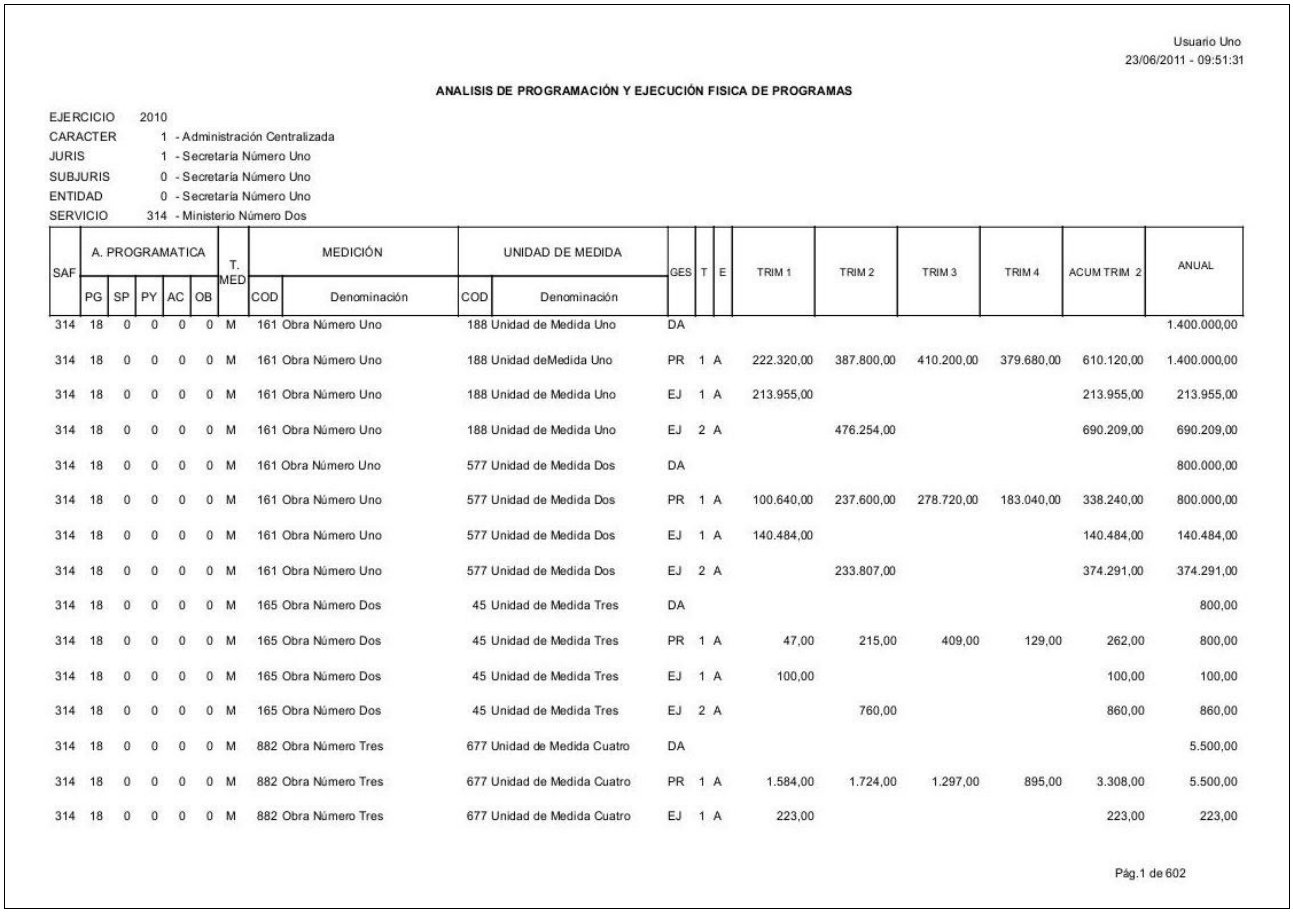
reportes no son utilizados con mucha frecuencia, se requiere que su tiempo de ejecución no sea excesivo .
En el sistema e-Sidif, el modelo de objetos de dominio es complejo, posee una gran cantidad de objetos con muchas relaciones a otros objetos (grafo grande y completo). Este modelo de objetos se persiste en un conjunto de tablas normalizadas.
Los reportes del sistema deben consultar diferentes partes del grafo (subgrafos) de objetos y esto hace que el mapeador objeto/relacional ejecute gran cantidad de consultas a la base de datos para recuperar toda la información solicitada. Cuando la performance de la aplicación se ve degradada por este motivo, una alternativa es la utilización de vistas SQL que reduzcan la cantidad de tablas a acceder y consultas a realizar desde la aplicación.
Las vistas se encargan de resolver parte de la lógica para obtener los datos del reporte, aprovechando la potencia del motor de base de datos y retornar los datos con una estructura esperada por el usuario o una estructura similar a esta.
El caso mas sencillo es que cada fila que forma parte del resultado de la ejecución de la vista se mapee directamente a un objeto y que el reporte simplemente se limita a mostrar una colección de estos objetos. Un caso más complejo consiste en que se necesiten datos de más de una fila de la vista para poder mostrar información en el reporte. En este caso, si bien cada fila resultado de la vista se mapea a un objeto, luego es necesario realizar un post procesamiento de estos objetos y armar un modelo de objetos que se adecúe a las necesidades del reporte.

Ejemplo: Listado de Análisis de Programas

El módulo PEF del e-Sidif tiene como objetivo registrar la programación, ejecución y cierre de ejercicio correspondiente a la información física, su relación con los estados financieros y el seguimiento en el Órgano Rector. Este registro de información se realiza en comprobantes eSidif.
Un comprobante consta de una cabecera e ítems. La cabecera contiene la identificación del comprobante (tipo y número de comprobante, gestión y entidad emisora), el evento (programación, ejecución o cierre) y período (trimestre1, 2, 3, 4 u año). La estructura de la cabecera es igual para los comprobantes de programación, ejecución y cierre.
Los ítems de los comprobantes de programación, ejecución y cierre difieren levemente. Todos los ítems referencian a una imputación física (asociación de los clasificadores institución, servicio, apertura programática, medición física y unidad de medida).

- Los ítems de programación representan lo proyectado de avance físico de los proyectos u obras en cada uno de los trimestres.
- Los ítems de ejecución representan lo ejecutado en cada trimestre y los motivos o causas de desvío cuando existe diferencia entre lo programado y lo ejecutado.
- Los ítems de cierre contienen la ratificación o rectificación de la información correspondiente a la ejecución informada periódicamente a lo largo del ejercicio, a fin de incluir, en términos anuales, los avances de proyectos de inversión, los comentarios de la ejecución y la explicación de los desvíos.

El listado de Análisis de Programas es uno de los reportes definidos en el módulo. Este reporte muestra lo valores correspondiente a lo programado, ejecutado y el cierre, por cada imputación física del ejercicio. La Figura 24 muestra un ejemplo de la salida del reporte.



Usuario Uno
23/06/2011 - 09:51:31

ANALISIS DE PROGRAMACIÓN Y EJECUCIÓN FISICA DE PROGRAMAS

EJERCICIO 2010
CARACTER 1 - Administración Centralizada
JURIS 1 - Secretaría Número Uno
SUBJURIS 0 - Secretaría Número Uno
ENTIDAD 0 - Secretaría Número Uno
SERVICIO 314 - Ministerio Número Dos

SAF	A. PROGRAMATICA					T. MED	MEDICIÓN		UNIDAD DE MEDIDA		GES	T	E	TRIM 1	TRIM 2	TRIM 3	TRIM 4	ACUM TRIM 2	ANUAL
	PG	SP	PY	AC	OB		COD	Denominación	COD	Denominación									
314	18	0	0	0	0	M	161	Obra Número Uno	188	Unidad de Medida Uno	DA								1.400.000,00
314	18	0	0	0	0	M	161	Obra Número Uno	188	Unidad deMedida Uno	PR	1	A	222.320,00	387.800,00	410.200,00	379.680,00	610.120,00	1.400.000,00
314	18	0	0	0	0	M	161	Obra Número Uno	188	Unidad de Medida Uno	EJ	1	A	213.955,00				213.955,00	213.955,00
314	18	0	0	0	0	M	161	Obra Número Uno	188	Unidad de Medida Uno	EJ	2	A		476.254,00			690.209,00	690.209,00
314	18	0	0	0	0	M	161	Obra Número Uno	577	Unidad de Medida Dos	DA								800.000,00
314	18	0	0	0	0	M	161	Obra Número Uno	577	Unidad de Medida Dos	PR	1	A	100.640,00	237.600,00	278.720,00	183.040,00	338.240,00	800.000,00
314	18	0	0	0	0	M	161	Obra Número Uno	577	Unidad de Medida Dos	EJ	1	A	140.484,00				140.484,00	140.484,00
314	18	0	0	0	0	M	161	Obra Número Uno	577	Unidad de Medida Dos	EJ	2	A		233.807,00			374.291,00	374.291,00
314	18	0	0	0	0	M	165	Obra Número Dos	45	Unidad de Medida Tres	DA								800,00
314	18	0	0	0	0	M	165	Obra Número Dos	45	Unidad de Medida Tres	PR	1	A	47,00	215,00	409,00	129,00	262,00	800,00
314	18	0	0	0	0	M	165	Obra Número Dos	45	Unidad de Medida Tres	EJ	1	A	100,00				100,00	100,00
314	18	0	0	0	0	M	165	Obra Número Dos	45	Unidad de Medida Tres	EJ	2	A		760,00			860,00	860,00
314	18	0	0	0	0	M	882	Obra Número Tres	677	Unidad de Medida Cuatro	DA								5.500,00
314	18	0	0	0	0	M	882	Obra Número Tres	677	Unidad de Medida Cuatro	PR	1	A	1.584,00	1.724,00	1.297,00	895,00	3.308,00	5.500,00
314	18	0	0	0	0	M	882	Obra Número Tres	677	Unidad de Medida Cuatro	EJ	1	A	223,00				223,00	223,00

Pág.1 de 602

Figura 24: Formato del listado de análisis de programas

Para obtener la información de este listado es necesario:

1. Recuperar los ítems de los comprobantes de programación, ejecución y cierre de programas.
2. Ordenar la información por evento y período (múltiples joins con tablas de dominio principales).
3. Recuperar las descripciones de los clasificadores que componen la imputación física (múltiples joins con tablas de refernecia).
4. Agruparlos por imputación física.
5. Hacer corte de control por los clasificadores que componen la imputación física.

Se creó una vista que resuelve parte (puntos 1, 2, 3) de la problemática anterior con el objetivo de reducir la cantidad de consultas a realizar por la aplicación por medio del mapeador y delegar en el motor de base de datos la planificación de todas las consultas.
Un fragmento de la vista que se construyó para implementar este listado se muestra en las Figuras 25 y 26.

```
CREATE OR REPLACE FORCE VIEW prod_esidif.pef_vn_analisis_programas (xl_caracter_institucional, xl_jurisdiccion, xl_subjurisdiccion,
                                                                   xl_entidad, c_ejercicio, c_sector, c_subsector,
                                                                   ...
                                                                   c_servicio, xl_servicio, c_programa, c_subprograma, c_proyecto,
                                                                   ...
                                                                   ID, ges, trimestre, estado_cpte, e_comprobante,
                                                                   triml, trim2, trim3, trim4,
                                                                   acum_triml, acum_trim2, acum_trim3, acum_trim4, anual)
 AS
   SELECT   (SELECT d.xl_sector_institucional FROM tb_bsectorinstitucional d WHERE d.id_ejercicio = tmp.id_ejercicio
               AND d.c_sector = tmp.c_sector AND d.c_subsector = tmp.c_subsector
               AND d.c_caracter_institucional = tmp.c_caracter_institucional
               AND d.id_version IS NULL) AS xl_caracter_institucional,
            ...
            tmp.c_ejercicio, tmp.c_sector, tmp.c_subsector,
            tmp.c_caracter_institucional, tmp.c_jurisdiccion,
            tmp.c_subjurisdiccion,
            ...
            TO_NUMBER (CONCAT (TO_CHAR (tmp.id_imputacion_fisica),
                               TO_CHAR (tmp.id_comprobante)
                              )
                      ) ID,
            tmp.ges, tmp.trimestre, tmp.estado_cpte, tmp.e_comprobante,
            tmp.triml, tmp.trim2, tmp.trim3, tmp.trim4, tmp.acum_triml,
            tmp.acum_trim2, tmp.acum_trim3, tmp.acum_trim4, tmp.anual
       FROM (
            ...
            union all
             SELECT emi.OID id_emisor, emi.c_emisor,
                    impf.id_ejercicio c_ejercicio, si.c_sector,
                    si.c_subsector, si.c_caracter_institucional,
                    ins.c_jurisdiccion, ins.c_subjurisdiccion,
                    ...
                    cpte.id_comprobante,
                    CASE
                       WHEN (cpte.cl_evento LIKE '%PROGRAMACION')
                          THEN 'PR'
                       WHEN (cpte.cl_evento = 'EJECUCION')
                          THEN 'EJ'
                       WHEN (cpte.cl_evento = 'CIERRE')
                          THEN 'CI'
                    END ges,
                    CASE
                       WHEN (cpte.cl_evento = 'CIERRE')
                          THEN '4'
                       ELSE TO_CHAR (per.c_periodo)
                    END periodo,
                    CASE
                       WHEN (cpte.e_comprobante = 'aceptado')
                          THEN 'A'
                       WHEN (cpte.e_comprobante = 'aceptado_con_observaciones'
                            )
                          THEN 'O'
                       ...
                       ELSE ''
                    END estado_cpte,
                    cpte.e_comprobante,
                    CASE
                       WHEN (    (cpte.cl_evento = 'PROGRAMACION')
                             AND (per.c_periodo = 1)
                            )
                          THEN (SELECT detpro.fl_programado
                                  FROM pef_dperprofis detpro
                                 WHERE detpro.id_item = item.id_item
                                   AND detpro.id_periodo = 13)
                       WHEN (    (cpte.cl_evento = 'REPROGRAMACION')
                             AND (per.c_periodo > 1)
                             AND (   (cpte.e_comprobante = 'aceptado')
                                  OR (cpte.e_comprobante =
                                                        'aceptado_con_observaciones'
                                     )
                                 )
                            )
                          THEN (SELECT dd.fl_programado
                                  FROM pef_tgesfis c,
                                       pef_dgesfis d,
                                       pef_dperprofis dd
                                 WHERE c.id_comprobante = d.id_comprobante
                                   AND d.id_item = dd.id_item
                                   AND dd.id_periodo = 13
```

Figura 25: Fragmento de la vista para el listado de análisis de programas

```
                        )
                    )
                    THEN pef_utils.calculartotal
                                    (c_totalizador_avance_fisico,
                                     (SELECT dtot.fl_programado
                                        FROM pef_dpertotfis dtot
                                       WHERE dtot.id_tot_fis =
                                                    tot.id_tot_fis
                                         AND dtot.id_periodo = 13),
                                     NULL,
                                     NULL,
                                     NULL,
                                     impf.n_precision_cantidad
                                    )
                  WHEN (    (cpte.cl_evento = 'EJECUCION')
                    ...
                  WHEN (    (cpte.cl_evento = 'CIERRE')
                    ...
                END acum_trim1,
                ... acum_trim2,
                ... acum_trim3,
                ... acum_trim4,
                item.fl_ejecutado anual
           FROM tb_bimputacionfisica impf, tb_binstitucion ins, tb_bsectorinstitucional si,
                pef_ttotfis tot, tb_bunidaddescentralizada ud, tb_bservicio ser,
                tb_baperturaprogramatica apg, tb_bmedicionfisica mf, tb_btipomedicionfisica tmf,
                tb_bunidadmedida um, tb_btotalizadoravancefisico taf, pef_tgesfis cpte,
                tb_v_bemisor emi, pef_dgesfis item, tb_bperiodo per
          WHERE impf.id_version IS NULL
            AND impf.id_institucion = ins.id_institucion  AND ins.id_sector_institucion = si.id_sector_institucional
            AND tot.id_imputacion_fisica(+) = impf.id_imputacion_fisica AND impf.id_medicion_fisica = mf.id_medicion_fisica
            AND mf.id_tipo_medicion_fisica = tmf.id_tipo_medicion_fisica AND tot.id_unidad_descentralizada =
            ud.id_unidad_descentralizada AND impf.id_servicio = ser.id_servicio AND impf.id_apertura_programatica =
            apg.id_apertura_programatica AND impf.id_tot_ava_fis = taf.id_totalizador_avance_fisico
            AND tmf.c_tipo_medicion_fisica NOT IN ('N', 'S') AND um.id_unidad_medida = impf.id_unidad_medida
            AND cpte.id_comprobante = item.id_comprobante AND cpte.id_entidad_emisora = emi.OID
            AND item.id_imputacion_fisica = impf.id_imputacion_fisica AND item.id_unidad_descentralizada =
            ud.id_unidad_descentralizada AND cpte.id_periodo = per.id_periodo AND cpte.e_comprobante IN
                  ('aceptado', 'aceptado_con_observaciones',
                   'sustituido', 'en_analisis', 'devuelto_onp')) tmp
ORDER BY tmp.c_ejercicio, tmp.c_sector, tmp.c_subsector, tmp.c_caracter_institucional,
         tmp.c_jurisdiccion, tmp.c_subjurisdiccion, tmp.c_entidad, tmp.c_servicio,
         tmp.c_programa, tmp.c_subprograma, tmp.c_proyecto, tmp.c_actividad, tmp.c_obra,
         tmp.c_tipo_medicion_fisica, tmp.c_medicion_fisica, tmp.c_unidad_medida,
         DECODE (tmp.ges, 'DA(*)', 0, 'DA', 1, 'PR', 2, 'EJ', 3, 'CI', 4),
         tmp.estado_cpte, tmp.trimestre;
```

Figura 26: Fragmento de la vista para el listado de análisis de programas (continuación)

Se creó un modelo de objetos intermedio para interpretar el resultado. Cada fila resultado de la vista se mapea a un objeto FilaAnalisisPrograma con la estructura que se muestra en la Figura 27.

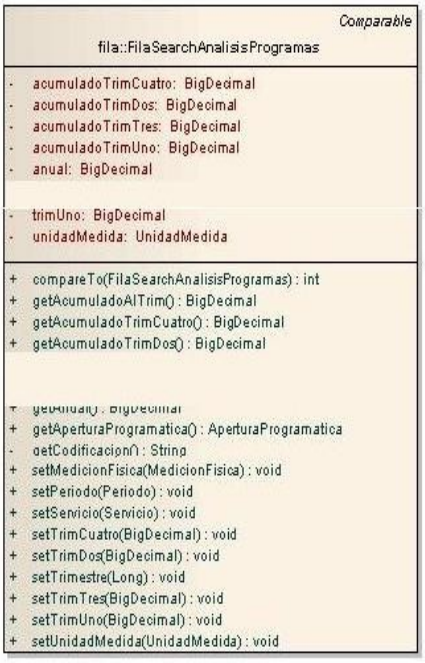


Figura 27: Clase FilaSearchAnalisisPrograma

El listado se implementó utilizando Jasper Report, teniendo como soporte la componente de Reportes provista por la arquitecture e-Sidif. La componente de Reportes provee un servicio encargado de mostrar el listado a partir de una colección de objetos que se envían como parámetro. En este caso, se trata de una colección de objetos FilaAnalisisPrograma, creados exclusivamente para el reporte y que son una representación simple de un modelo de dominio mucho mas grande, similar a objetos DTO.
En este ejemplo se logró reducir la complejidad y cantidad de consultas a realizar por la aplicación (originalmente implicaba una enorme cantidad de joins con diversas tablas), delegando esta tarea a una vista del motor de base de datos Oracle, lo cual generó grandes benficios en la performance. Dado que esta vista retorna datos no directamente relacionados (o no mapeables directamente) con el modelo de dominio, se creó un modelo intermedio para facilitar la renderización del reporte.

Caso: Procedimientos almacenados

El sistema e-Sidif se caracteriza por modelar procesos de negocio que requieren lógica de aplicación compleja para ser resueltos. Estos procesos requieren la interacción no solo de gran cantidad de objetos sino también de muchas instancias de ellos. Para poder resolver estos procesos de negocio, la aplicación necesita disponer en memoria de un gran grafo de objetos, y para lograr esto inevitablemente se requieren ejecutar muchas consultas a la base de datos.
Cuando nos encontramos con esta situación, fue necesario analizar el costo/beneficio de implementar estos procesos de negocio como procedimientos almacenados del motor de base de datos para lograr una importante mejora de performance.

Ejemplo: Copia de Escenario de Simulación

El módulo de Formulación Presupuestaria del e-Sidif es el encargado de la elaboración de las distintas gestiones de la formulación del Presupuesto Nacional que realiza la ONP (Oficina Nacional de Presupuesto) y los distintos organismos. Durante el proceso de

formulación se generan distintos escenarios o simulaciones de presupuestación que reflejan las opciones de política presupuestaria definidas por las autoridades de cada organismo. El análisis de las distintas alternativas permite elaborar el anteproyecto de presupuesto en cada organismo. La ONP centraliza la recepción de estos anteproyectos y con esta información elabora el Proyecto de Ley del Presupuesto Nacional.
El registro de la información necesaria para elaborar las distintas gestiones de la formulación se realiza en un Escenario de Simulación. Este escenario consiste de una cabecera, componentes, elementos y etapas de escenario.

- La cabecera contiene información de configuración tal como, año de ejercitación, entidad emisora y de proceso del escenario, gestión, año de presupuestación, descripción del escenario, año y versión de referencia para los clasificadores presupuestarios.
- Una componente de escenario está definida por un tipo de componente (por ejemplo crédito, recurso, cargo), una estructura presupuestaria (define el nivel de agregación de los clasificadores que intervienen en los elementos de la componente) y la lista de elementos de escenario.
- Un elemento de escenario está compuesto por una imputación presupuestaria (de crédito, recurso, cargo según la componente a la que pertenece el ítem) y el saldo inicial, final y ajustes.
- Una etapa permite identificar los distintos pasos por los que atraviesa una imputación presupuestaria. En cada etapa se registra el ajuste que sufre la imputación presupuestaria.

Las Figuras 28 y 29 muestran el diagrama de clases del modelo de escenarios de simulación y las Figuras 30, 31, 32 y 33 muestran el modelo de base de datos donde se persisten los datos de los escenarios.

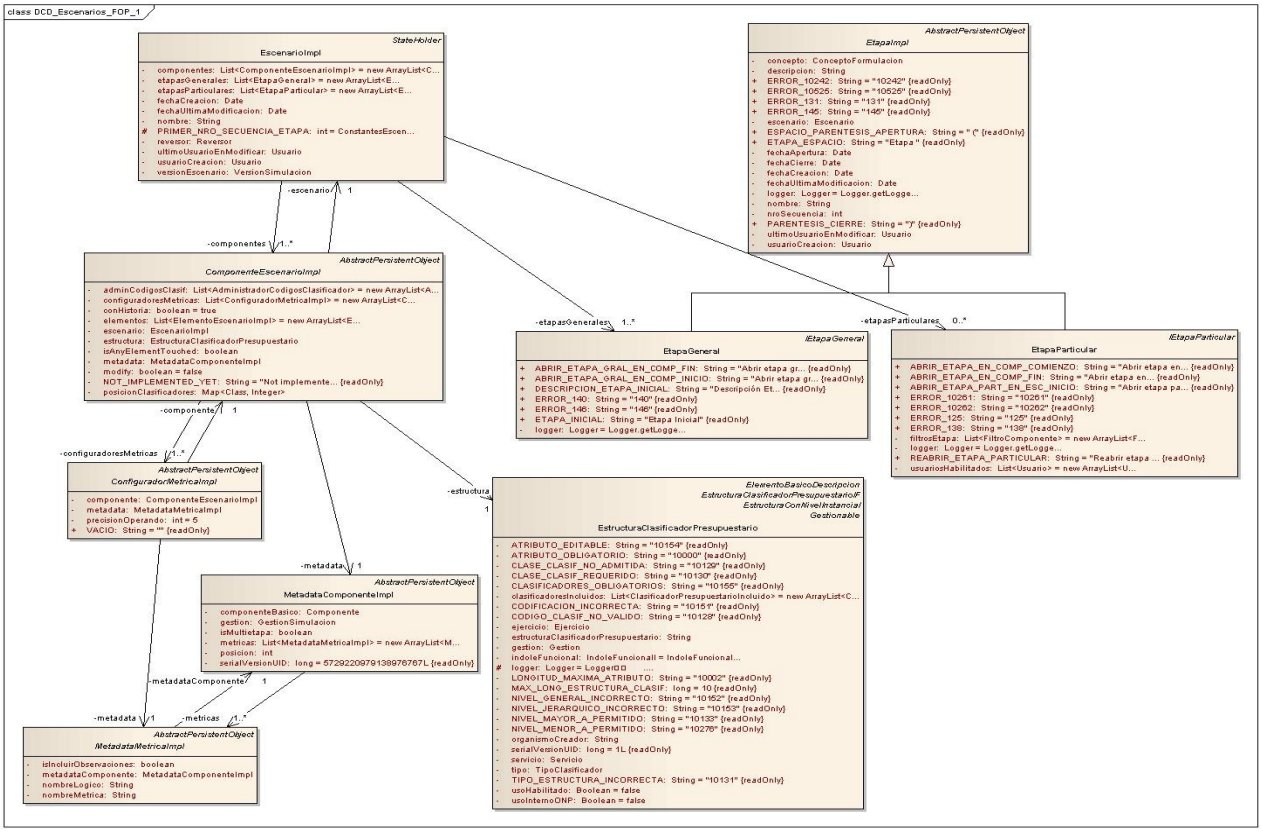


Figura 28: Modelo de clases de Escenario de Simulación

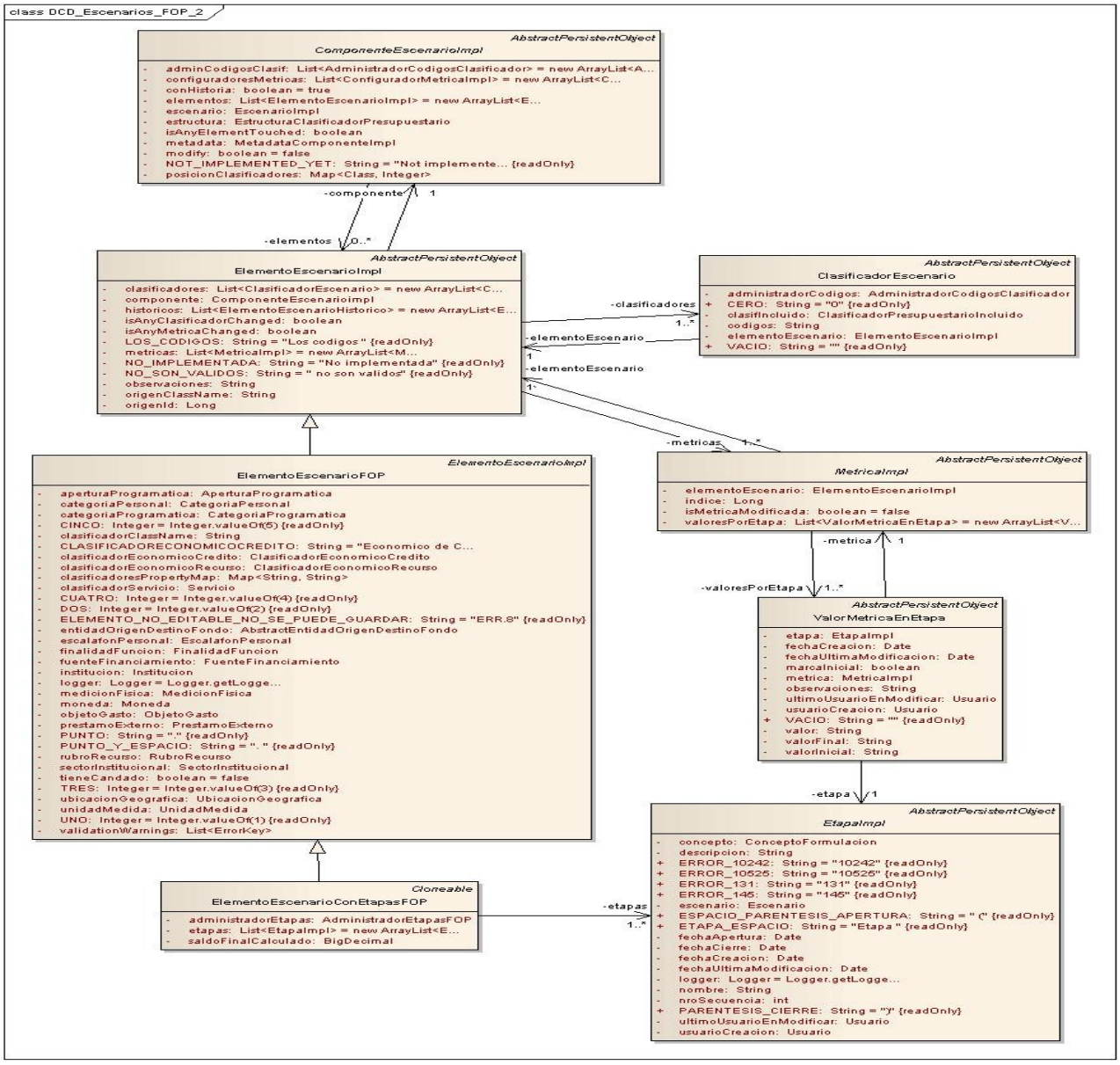


Figura 29: Modelo de clases de Escenario de Simulación (continuación)

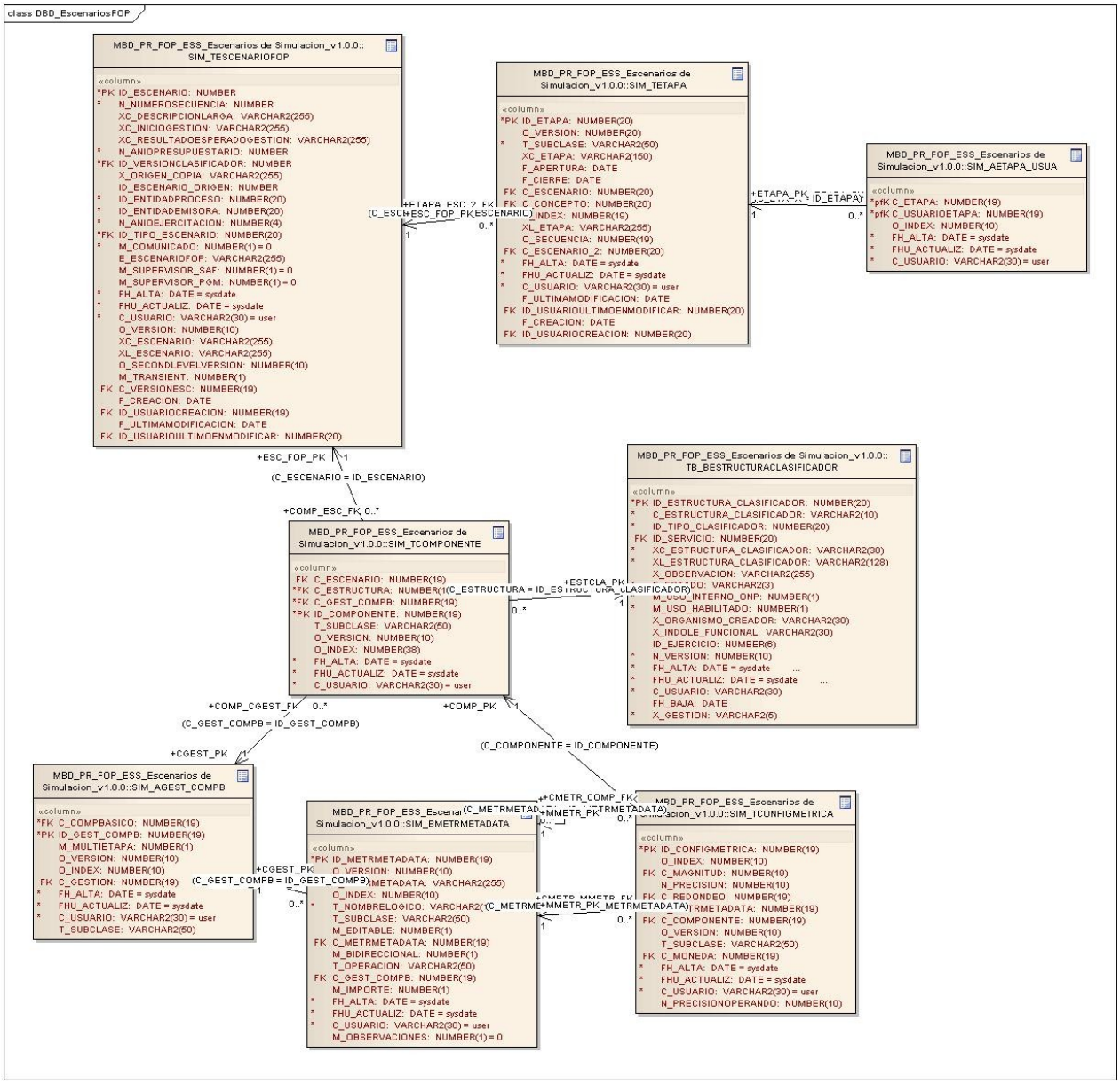


Figura 30: Modelo de datos de Escenario de Simulación

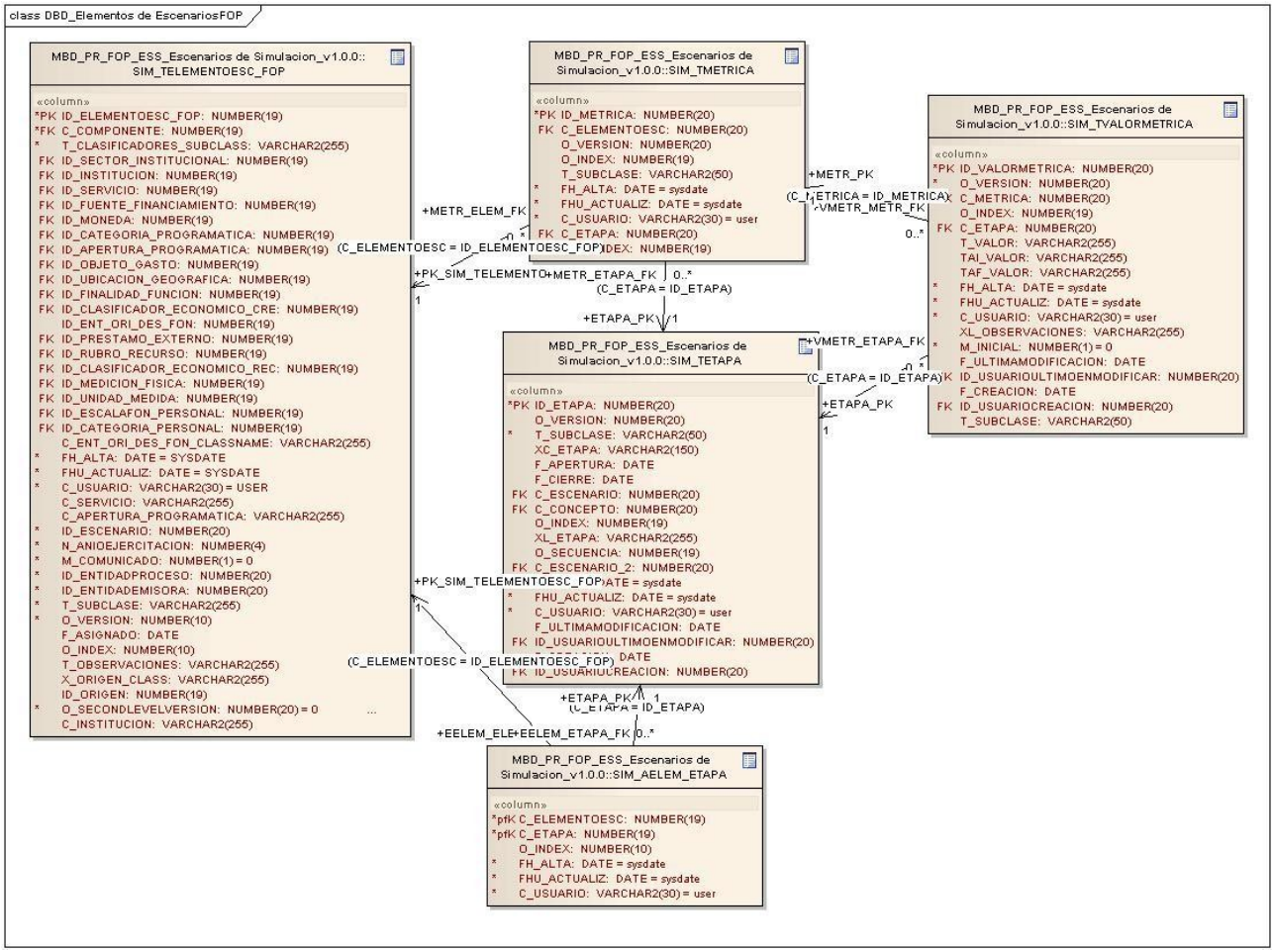


Figura 31: Modelo de datos de Escenarios de Simulación (continuación)

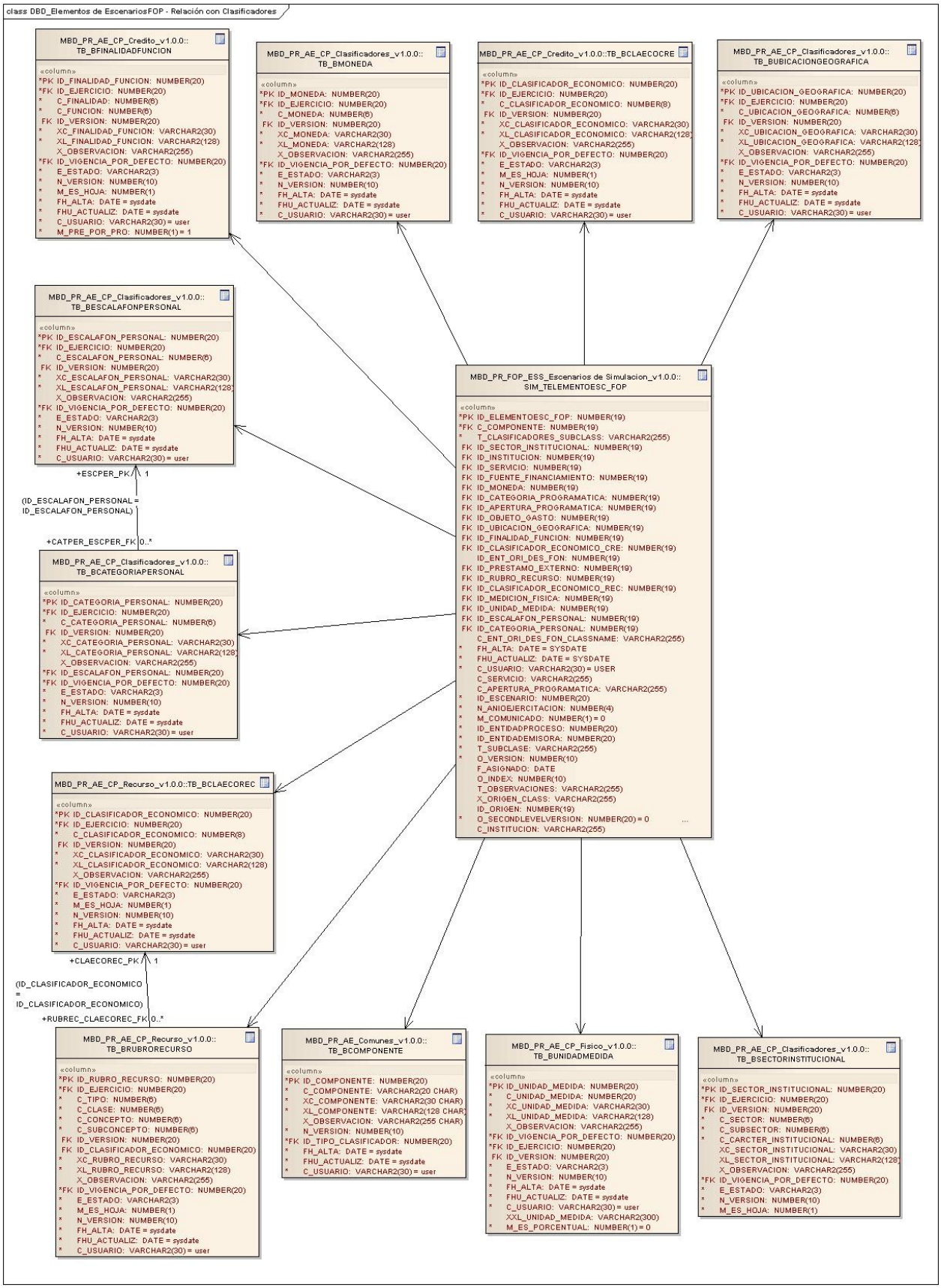


Figura 32: Modelo de datos de Elementos de Escenarios

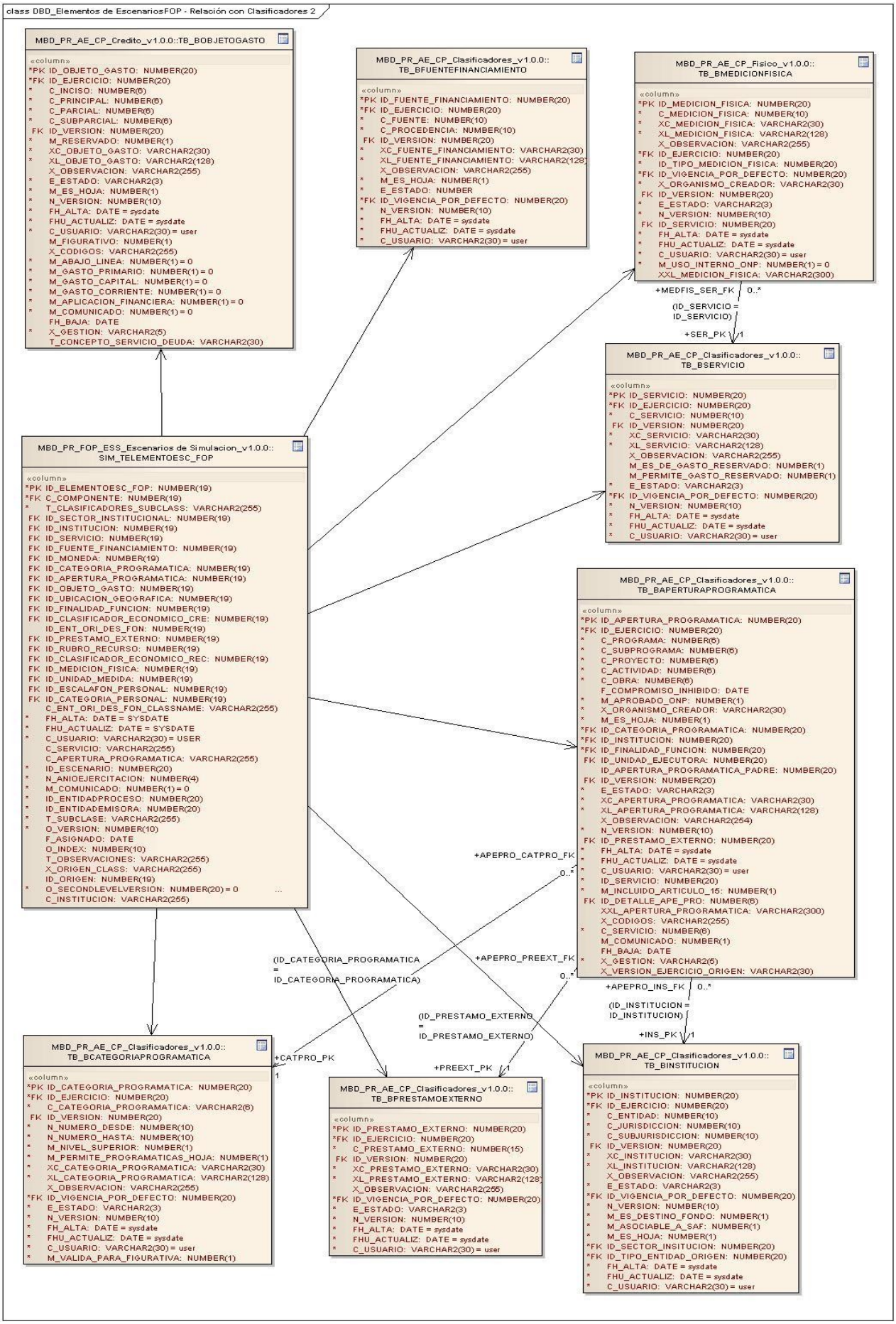


Figura 33: Modelo de datos de Elementos de Escenarios (continuación)

Un escenario de simulación se puede crear manualmente, es decir dar de alta la cabecera del escenario, agregar las componentes con sus estructuras de clasificador, cargar los elementos en cada componente, crear las etapas y hacer los ajustes necesarios sobre los elementos en las distintas etapas.
La creación de un escenario es una tarea ardua y compleja y más aún teniendo en cuenta que un escenario puede contener aproximadamente 70.000 elementos.
Existe funcionalidades específicamente creadas para agilizar la carga de los elementos de un escenario. Por ejemplo, mediante la importación masiva de elementos de escenario desde un archivo Excel o la importación de elementos desde otro escenario. Pero también nos encontramos con situaciones donde el escenario que se quiere crear es muy similar a otro ya existente, no solo en cuanto a sus elementos, sino también su configuración, etapas y ajustes. Para dar soporte a esta situación se implementó un nuevo requerimiento que es la copia de escenario de simulación.
La copia de escenarios crea un nuevo escenario de simulación con la misma configuración, estructura, elementos, etapas y ajustes de los elementos en las distintas etapas pero con otra descripción y número de escenario.
Teniendo en cuenta la complejidad del modelo de objetos, el volúmen de datos y el procesamiento necesario para llevar a cabo esta operación, se decidió implementarla mediante un procedimiento almacenado en la base de datos. Este procedimiento almacenado se invoca desde la aplicación con los parámetros necesarios.
Un fragmento del procedimiento almacenado para copiar un escenario de simulación se muestra en las Figuras 34 y 35.

```
CREATE OR REPLACE PACKAGE PROD_ESIDIF.SIM_SPA_COPIA_ESCENARIO_SIMPLE IS
   PROCEDURE ESCENARIO(NUEVO_N_NUMEROSECUENCIA IN SIM_TESCENARIOFOP.N_NUMEROSECUENCIA%TYPE , ID_USER_NUEVO IN SIM_TESCENARIOFOP.ID_USUARIOCREACION%TYPE );
   PROCEDURE CAMBIOESTADO;
   PROCEDURE CLAUSULA;
   PROCEDURE CLAUSULANIVEL;
   PROCEDURE COMPONENTE;
   PROCEDURE CONFIGMETRICA;
   PROCEDURE ELEMENTO;
   PROCEDURE ELEMENTO_FOP;
   PROCEDURE ELEMENTO_HISTORICO;
   PROCEDURE EST_ESCENARIO;
   PROCEDURE ETAPA_ELEM;
   PROCEDURE ETAPA;
   PROCEDURE ETAPA_USUARIO;
   PROCEDURE FILTRO_CLASIFICADOR;
   PROCEDURE FILTRO_COMPONENTE;
   PROCEDURE LOGESTADO;
   PROCEDURE METRICA;
   PROCEDURE VALOR_METRICA;
   PROCEDURE VALOR_METRICA_HISTORICO;
   FUNCTION MAIN(P_ID_ESCENARIO IN SIM_TESCENARIOFOP.ID_ESCENARIO%TYPE,
      NUEVO_N_NUMEROSECUENCIA IN SIM_TESCENARIOFOP.N_NUMEROSECUENCIA%TYPE , ID_USER_NUEVO IN SIM_TESCENARIOFOP.ID_USUARIOCREACION%TYPE )
  RETURN SIM_TESCENARIOFOP.ID_ESCENARIO%TYPE;
END;
/

FUNCTION MAIN
     (P_ID_ESCENARIO IN SIM_TESCENARIOFOP.ID_ESCENARIO%TYPE,
      NUEVO_N_NUMEROSECUENCIA IN SIM_TESCENARIOFOP.N_NUMEROSECUENCIA%TYPE , ID_USER_NUEVO IN SIM_TESCENARIOFOP.ID_USUARIOCREACION%TYPE )
  RETURN SIM_TESCENARIOFOP.ID_ESCENARIO%TYPE IS
   NEW_ID_ESCENARIO SIM_TESCENARIOFOP.ID_ESCENARIO%TYPE;
   BEGIN
   DELETE CP_TEMP_MAPEO_COPIAS;
   DBMS_OUTPUT.PUT_LINE('Borrado de temporal');
   DBMS_APPLICATION_INFO.SET_MODULE('SIM_SPA_COPIA_ESCENARIO_SIMPLE','MAIN');
   SELECT SIM_SPA_S_TESCENARIO.NEXTVALUE INTO NEW_ID_ESCENARIO FROM DUAL;
   INSERT INTO CP_TEMP_MAPEO_COPIAS(X_TABLA, ID_ORIGEN, ID_DESTINO)
   VALUES ('SIM_TESCENARIOFOP', P_ID_ESCENARIO, NEW_ID_ESCENARIO);
   ESCENARIO(NUEVO_N_NUMEROSECUENCIA,ID_USER_NUEVO);
   RETURN NEW_ID_ESCENARIO;
   EXCEPTION
   WHEN OTHERS THEN
   RAISE;
   END;
END;
```

Figura 34: Fragmento del procedimiento almacenado para copiar escenarios

```
PROCEDURE ESCENARIO(NUEVO_N_NUMEROSECUENCIA IN SIM_TESCENARIOFOP.N_NUMEROSECUENCIA%TYPE , ID_USER_NUEVO IN SIM_TESCENARIOFOP.ID_USUARIOCREACION%TYPE ) IS
BEGIN
    -- Se copia el estado escenario
    EST_ESCENARIO();
    -- Se copia e inserta el nuevo escenario fop
    INSERT INTO SIM_TESCENARIOFOP (ID_ESCENARIO, N_NUMEROSECUENCIA, XC_DESCRIPCIONLARGA, XC_INICIOGESTION,
                                   XC_RESULTADOESPERADOGESTION, N_ANIOPRESUPUESTARIO, ID_VERSIONCLASIFICADOR,
                                       X_ORIGEN_COPIA, ID_ESCENARIO_ORIGEN, ID_ENTIDADEMISORA, ID_ENTIDADPROCESO, N_ANIOEJERCITACION, ID_TIPO_ESCENARIO, M_COMUNICADO, E_ESCENARIOFOP ,
                                       M_SUPERVISOR_SAF , M_SUPERVISOR_PGM , O_VERSION ,   XC_ESCENARIO , XL_ESCENARIO , O_SECONDLEVELVERSION  , M_TRANSIENT, C_VERSIONESC , F_CREACION,
                                       ID_USUARIOCREACION, F_ULTIMAMODIFICACION , ID_USUARIOULTIMOENMODIFICAR)
    SELECT TE.ID_DESTINO, NUEVO_N_NUMEROSECUENCIA, E.XC_DESCRIPCIONLARGA, E.XC_INICIOGESTION, E.XC_RESULTADOESPERADOGESTION,
           E.N_ANIOPRESUPUESTARIO, E.ID_VERSIONCLASIFICADOR,  NULL,          TE.ID_ORIGEN,
           E.ID_ENTIDADEMISORA, E.ID_ENTIDADPROCESO, e.N_ANIOEJERCITACION, e.ID_TIPO_ESCENARIO, e.M_COMUNICADO, e.E_ESCENARIOFOP , 0 , 0 , 0 , e.XC_ESCENARIO || ' (Copia ' ||
           TO_CHAR(SYSDATE, 'YYYYMMDDHHMISS') || ')', e.XL_ESCENARIO, 0,
        e.M_TRANSIENT, e.C_VERSIONESC, ing_spar_fecha.fecha_bd , ID_USER_NUEVO , ing_spar_fecha.fecha_bd , ID_USER_NUEVO
    FROM SIM_TESCENARIOFOP E, CP_TEMP_MAPEO_COPIAS TE
    WHERE TE.X_TABLA = 'SIM_TESCENARIOFOP'
        AND E.ID_ESCENARIO = TE.ID_ORIGEN ;
     dbms_output.put_line('CANT fop ='||sql%rowcount);
    -- Copia de hijos de la entidad
    COMPONENTE();
    ETAPA();
    FILTRO_COMPONENTE();
    ELEMENTO();
    ETAPA_ELEM();
END;
```

Figura 35: Fragmento del procedimiento almacenado para copiar escenarios (continuación)

Sin bien la utilización de procedimientos almacenados no es una situación “común” en aplicaciones orientados a objetos, en este ejemplo vimos que este recurso tecnológico se convierte en un recurso válido y que su utilización favorece enormemente no solo a la performance de la aplicación sino también a la estabilidad de la misma, reduciendo el consumo de memoria que sería necesario si se debieran cargar todo el grafo de objetos en memoria. Esta solución de compromiso posee más ventajas que desventajas para la salud de la aplicación.

Caso: Mapeos diferenciales

El e-Sidif es un sistema que permite registrar la gestión de las distintas etapas del presupuesto y posteriormente consultar la información registrada mediante la generación de listados y reportes. El proceso de registración requiere el ingreso de una gran cantidad de datos, sin embargo solo algunos de estos datos son necesarios al momento de mostrar resultados de una búsqueda o la emisión de un reporte.

- Para ejecutar algunas funcionalidades del sistema nos encontramos con la necesidad de tener en memoria objetos de dominio complejos con muchos atributos y relaciones, por ejemplo para la edición de estos objetos de dominio.
- Por otro lado existen otras requerimientos en la aplicación donde solo se necesita mostrar algunos pocos atributos de estos mismos objetos de dominio, por ejemplo para visualizar posibles valores en un combo de selección, mostrar el resultado de una búsqueda o generar un reporte resumido del sistema.

De un mismo objeto, en algunos casos necesitamos cargar todo el subgrafo completo y en otros casos es suficiente disponer de algunos atributos y relaciones. Podemos resolver ambas situaciones con una sola solución que es cargar a memoria el objeto completo, aunque en el segundo caso estamos penalizando al sistema con la carga de muchos datos que no son necesarios y esto puede ocasionar problemas de performance. Una solución que evita cargar en memoria objetos innecesarios es trabajar con *mapeos alternativos*.

El uso de mapeos alternatios o mapeos diferenciales significa que un mismo objeto de dominio pueda tener diferentes archivos de mapeos, que determinan diferentes alcances de los datos a cargar. En cada mapeo se define que atributos y relaciones se quieren cargar, si son lazy o eager, atributos calculados y cualquier otra variante que el ORM soporte y se adapte mejor a cada situación en pos de mejorar el rendimiento de la aplicación. Luego cada caso de uso configura dinámicamente que mapeo quiere utilizar según las caracterísitcas de la funcionalidad a implementar.

Ejemplo: Escenarios de Simulación

Como mencionamos en el caso de Procedimientos Almancenados, el módulo de Formulación Presupuestaria de e-Sidif se encarga del registro de la información necesaria para elaborar las distintas gestiones de la formulación presupuestaria. El registro de esta información se lleva a cabo en un Escenario de Simulación. Las Figuras 28 y 29 muestran el modelo de objetos de un Escenario de Simulación.
La funcionalidad de alta o modificación de un escenario de simulación requiere tener en memoria el modelo completo de un escenario, es decir cabecera, componentes, etapas, elementos, métricas. Las Figuras 36 y 37 muestran el formato del archivo de mapeo de Escenarios y Elementos de Escenarios respectivamente para este caso.

```
<hibernate-mapping>
 <class name="ar.gov.mecon.esidif.escenarios_fop.server.domain.EscenarioFOP" table="SIM_TESCENARIOFOP" lazy="false"
    polymorphism="explicit">
        <id name="id" column="ID_ESCENARIO" type="long" unsaved-value="0">
            <generator class="org.hibernate.id.SequenceHiLoGenerator">
                <param name="sequence">SIM_S_TESCENARIO</param>
                <param name="max_lo">1000</param>
            </generator>
        </id>
        <version name="version" column="O_VERSION"/>
        <property name="secondLevelVersion" column="O_SECONDLEVELVERSION"  type="integer"/>
        <property name="nombre" access="field" type="string" column="XC_ESCENARIO" />
        <property name="fechaCreacion" access="field" type="timestamp" column="F_CREACION"/>
        <property name="numeroSecuencia" access="field" type="long" column="N_NUMEROSECUENCIA" />
        <property name="descripcionLarga" access="field" type="string" column="XC_DESCRIPCIONLARGA" />
        <property name="inicioGestion" access="field" type="string" column="XC_INICIOGESTION" />
        <property name="resultadoEsperadoGestion" access="field" type="string" column="XC_RESULTADOESPERADOGESTION" />
        <property name="anioPresupuestario" access="field"  type="integer" column="N_ANIOPRESUPUESTARIO" />
        <property name="origenDeCopia" access="field" type="string" column="X_ORIGEN_COPIA" />
        <property name="nombreEstado" type="string" column="E_ESCENARIOFOP" not-null="true" />
        <property name="anioEjercitacion" access="field" type="integer" column="N_ANIOEJERCITACION" />
        <property name="marcaASupervisorDestino" access="field" type="boolean"  column="M_SUPERVISOR_SAF" />
          <property name="marcaASupervisorPrograma" access="field" type="boolean"   column="M_SUPERVISOR_PGM" />
        <property name="fechaUltimaModificacion" access="field" type="timestamp" column="F_ULTIMAMODIFICACION"/>
        <many-to-one name="entidadEmisora" column="ID_ENTIDADEMISORA" class="ar.gov.mecon.esidif.interfaces.Emisor"
        entity-name="EmisorEmisor" cascade="save-update" />
        <many-to-one name="entidadProceso" column="ID_ENTIDADPROCESO" class="ar.gov.mecon.esidif.interfaces.Emisor"
        entity-name="EmisorEmisor" cascade="save-update" />
        <many-to-one name="tipoEscenario" column="ID_TIPO_ESCENARIO"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.TipoEscenario" cascade="save-update" />
        <many-to-one name="ultimoUsuarioEnModificar" access="field" column="ID_USUARIOULTIMOENMODIFICAR"
        class="ar.gov.mecon.esidif.seguridad.model.user.UsuarioImpl" />
        <many-to-one name="versionEscenario" access="field" column="C_VERSIONESC"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.VersionSimulacion" />
        <many-to-one name="usuarioCreacion" access="field" column="ID_USUARIOCREACION"
        class="ar.gov.mecon.esidif.seguridad.model.user.UsuarioImpl" />
        <many-to-one name="versionClasificador" access="field" column="ID_VERSIONCLASIFICADOR"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.VersionClasificador"
            cascade="save-update" />
        <list name="componentes" access="field" cascade="all-delete-orphan" lazy="true" batch-size="5">
            <key column="C_ESCENARIO"/> <index column="O_INDEX"/>
            <one-to-many class="ar.gov.mecon.esidif.domain.escenarios.componentes.ComponenteEscenarioImpl"/>
        </list>
        <list name="etapasGenerales" access="field" cascade="all-delete-orphan" lazy="true" batch-size="5">
            <key column="C_ESCENARIO"/> <index column="O_INDEX"/>
            <one-to-many class="ar.gov.mecon.esidif.domain.escenarios.etapas.EtapaGeneral"/>
        </list>
        <list name="etapasParticulares" access="field" cascade="all-delete-orphan" lazy="true" batch-size="5">
            <key column="C_ESCENARIO_2"/> <index column="O_INDEX"/>
            <one-to-many class="ar.gov.mecon.esidif.domain.escenarios.etapas.EtapaParticular"/>
        </list>
 </class>
</hibernate-mapping>
```

Figura 36: Archivo de mapeo de Escenario de Simulación

```
<hibernate-mapping>
    <class
        name="ar.gov.mecon.esidif.escenarios_fop.server.domain.elementos.ElementoEscenarioFOP"
        table="SIM_TELEMENTOESC_FOP"
        discriminator-value="ElementoSinEtapasFOP" lazy="true"
        polymorphism="explicit">
        <id name="id" column="ID_ELEMENTOESC_FOP" type="long"
            unsaved-value="0">
            <generator class="org.hibernate.id.SequenceHiLoGenerator">
                <param name="sequence">SIM_S_TELEMENTOESC</param>
                <param name="max_lo">1000</param>
            </generator>
        </id>
        <discriminator column="T_SUBCLASE" type="string" />
        <version name="version" column="O_VERSION" />
        <property name="secondLevelVersion" column="O_SECONDLEVELVERSION" type="integer" />
        <many-to-one name="componente"  class="ar.gov.mecon.esidif.domain.escenarios.componentes.ComponenteEscenarioImpl"
            column="C_COMPONENTE" lazy="proxy" cascade="save-update" not-null="true"/>
        <property name="observaciones" type="string" access="field" column="T_OBSERVACIONES" />
        <property access="property" name="clasificadorClassName" column="T_CLASIFICADORES_SUBCLASS" type="string" />
        <any lazy="true" access="field" name="entidadOrigenDestinoFondo" meta-type="string" id-type="long">
            <column name="C_ENT_ORI_DES_FON_CLASSNAME" /> <column name="ID_ENT_ORI_DES_FON" />
        </any>
        <many-to-one lazy="proxy" access="field" name="sectorInstitucional" column="ID_SECTOR_INSTITUCIONAL"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.SectorInstitucional"/>
        <many-to-one lazy="proxy" access="field" name="institucion" column="ID_INSTITUCION"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.Institucion"/>
        <many-to-one lazy="proxy" access="field" name="clasificadorServicio" column="ID_SERVICIO"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.Servicio"/>
        <many-to-one lazy="proxy" access="field" name="fuenteFinanciamiento" column="ID_FUENTE_FINANCIAMIENTO"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.FuenteFinanciamiento"/>
        <many-to-one lazy="proxy" access="field" name="moneda" column="ID_MONEDA"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.Moneda"/>
        <many-to-one lazy="proxy" access="field" name="categoriaProgramatica" column="ID_CATEGORIA_PROGRAMATICA"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.CategoriaProgramatica"/>
        <many-to-one lazy="proxy" access="field" name="aperturaProgramatica" column="ID_APERTURA_PROGRAMATICA"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.AperturaProgramatica"/>
        <many-to-one lazy="proxy" access="field" name="objetoGasto" column="ID_OBJETO_GASTO"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.ObjetoGasto"/>
        <many-to-one lazy="proxy" access="field" name="ubicacionGeografica" column="ID_UBICACION_GEOGRAFICA"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.UbicacionGeografica"/>
        <many-to-one lazy="proxy" access="field" name="finalidadFuncion" column="ID_FINALIDAD_FUNCION"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.FinalidadFuncion"/>
        <many-to-one lazy="proxy" access="field" name="clasificadorEconomicoCredito" column="ID_CLASIFICADOR_ECONOMICO_CRE"
        ="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.ClasificadorEconomicoCredito"/>
        <many-to-one lazy="proxy" access="field" name="prestamoExterno" column="ID_PRESTAMO_EXTERNO"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.PrestamoExterno"/>
        <many-to-one lazy="proxy" access="field" name="rubroRecurso" column="ID_RUBRO_RECURSO"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.RubroRecurso"/>
        <many-to-one lazy="proxy" access="field" name="clasificadorEconomicoRecurso" column="ID_CLASIFICADOR_ECONOMICO_REC"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.ClasificadorEconomicoRecurso"/>
        <many-to-one lazy="proxy" access="field" name="medicionFisica" column="ID_MEDICION_FISICA"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.MedicionFisica"/>
        <many-to-one lazy="proxy" access="field" name="unidadMedida" column="ID_UNIDAD_MEDIDA"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.UnidadMedida"/>

        <many-to-one lazy="proxy" access="field" name="escalafonPersonal" column="ID_ESCALAFON_PERSONAL"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.EscalafonPersonal"/>
        <many-to-one lazy="proxy" access="field" name="categoriaPersonal" column="ID_CATEGORIA_PERSONAL"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.CategoriaPersonal"/>
        <list name="metricas" access="field" cascade="all-delete-orphan" lazy="true" batch-size="100">
            <key column="C_ELEMENTOESC" />  <index column="O_INDEX" />
            <one-to-many class="ar.gov.mecon.esidif.domain.escenarios.metricas.MetricaImpl" />
        </list>
        <property name="origenClassName" type="string" access="field"    column="X_ORIGEN_CLASS" />
        <property name="origenId" type="long" access="field"    column="ID_ORIGEN" />
        <subclass
            name="ar.gov.mecon.esidif.escenarios_fop.server.domain.elementos.ElementoEscenarioConEtapasFOP"
            discriminator-value="ElementoConEtapasFOP" lazy="true">
            <list name="etapas" table="SIM_RELEM_ETAPA" access="property" cascade="save-update" batch-size="100" lazy="true">
                <key column="C_ELEMENTOESC" />  <index column="O_INDEX" />
                <many-to-many column="C_ETAPA"  class="ar.gov.mecon.esidif.domain.escenarios.etapas.EtapaImpl" />
            </list>
        </subclass>
    </class>
</hibernate-mapping>
```

Figura 37: Archivo de mapeo de Elemento de Escenario

Por otro lado, la búsqueda de escenarios de simulación a partir de un conjunto de filtros definidos por el usuario no necesita mostrar toda la información que se definió en el mapeo anterior. Solo necesita mostrar información que permita identificar a cada

escenario. No se requiere mostrar todos los elementos, las métricas y las etapas. La Figura 38 muestra el archivo de mapeo de escenarios para este caso.

```
<hibernate-mapping>
    <class name="ar.gov.mecon.esidif.escenarios_fop.server.domain.EscenarioFOP"
        table="SIM_TESCENARIOFOP" lazy="false"
        entity-name="EscenarioFOPSearch">
        <id name="id" column="ID_ESCENARIO" type="long"
            unsaved-value="0">
            <generator class="org.hibernate.id.SequenceHiLoGenerator">
                <param name="sequence">SIM_S_TESCENARIO</param>
                <param name="max_lo">1000</param>
            </generator>
        </id>
        <version name="version" column="O_VERSION" />   <property name="anioEjercitacion" access="field"
                type="integer" column="N_ANIOEJERCITACION" />
        <property name="secondLevelVersion"     column="O_SECONDLEVELVERSION" type="integer" />
        <property name="nombre" access="field" type="string"    column="XC_ESCENARIO" unique="true" />
        <many-to-one name="versionEscenario" access="field"     column="C_VERSIONESC"
            class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.VersionSimulacion" />
        <property name="fechaCreacion" access="field" type="timestamp"  column="F_CREACION" />
        <property name="fechaUltimaModificacion" access="field" type="timestamp" column="F_ULTIMAMODIFICACION" />
        <many-to-one name="usuarioCreacion" access="field"  column="ID_USUARIOCREACION"
            class="ar.gov.mecon.esidif.seguridad.model.user.UsuarioImpl" />
        <many-to-one name="ultimoUsuarioEnModificar" access="field" column="ID_USUARIOULTIMOENMODIFICAR"
            class="ar.gov.mecon.esidif.seguridad.model.user.UsuarioImpl" />
        <property name="marcaASupervisorDestino" access="field" type="boolean"  column="M_SUPERVISOR_SAF" />
         <property name="marcaASupervisorPrograma" access="field" type="boolean"    column="M_SUPERVISOR_PGM" />
        <property name="numeroSecuencia" access="field" type="long" column="N_NUMEROSECUENCIA" />
        <property name="descripcionLarga" access="field" type="string" column="XC_DESCRIPCIONLARGA" />
        <property name="inicioGestion" access="field" type="string" column="XC_INICIOGESTION" />
        <property name="resultadoEsperadoGestion" access="field" type="string" column="XC_RESULTADOESPERADOGESTION" />
        <property name="anioPresupuestario" access="field"  type="integer" column="N_ANIOPRESUPUESTARIO" />
        <property name="origenDeCopia" access="field" type="string" column="X_ORIGEN_COPIA" />
        <property name="nombreEstado" type="string" column="E_ESCENARIOFOP" not-null="true" />
        <many-to-one name="versionClasificador" access="field"  column="ID_VERSIONCLASIFICADOR"
            class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.VersionClasificador" cascade="save-update" />
        <many-to-one name="entidadEmisora"  column="ID_ENTIDADEMISORA"  class="ar.gov.mecon.esidif.interfaces.Emisor"
            entity-name="EmisorEmisor" cascade="save-update" />
        <many-to-one name="entidadProceso"  column="ID_ENTIDADPROCESO"  class="ar.gov.mecon.esidif.interfaces.Emisor"
            entity-name="EmisorEmisor" cascade="save-update" />
        <many-to-one name="tipoEscenario" column="ID_TIPO_ESCENARIO"
        class="ar.gov.mecon.esidif.presupuesto.entidadesBasicas.domain.TipoEscenario" cascade="save-update" />
    </class>
</hibernate-mapping>
```

Figura 38: Archivo de mapeo de Escenario para la búsqueda

Se define un único archivo de mapeo por cada objeto persistente en la configuración de la aplicación. Si embargo se puede modificar programáticamente el archivo de mapeo (la vista) asociado a un objeto. En nuestro caso, el primer archivo de mapeo es el archivo por defecto que utiliza la aplicación. Para modificar el archivo de mapeo en el caso de la búsqueda, se debe modificar la propiedad entityName del objeto Search que representa las búsquedas. La Figura 39 muestra un fragmento de código de la clase que implementa el comando para abrir la ventana de búsqueda, seteando el entityName.

```
@Override
public boolean doExecute(Parameters parameters) {

    OnSelectionCommand onSelectionCommand = new OpenEditorOnSelectionCommand();
    Class<? extends Composite> searchCompositeClass = EscenariosFOPSearchCompositeWithScrolling.class;
    Class<? extends EscenarioModel> escenarioModelClass = EscenarioFOPModel.class;
    String escenarioTargetClass = AR_GOV_MECON_ESIDIF_ESCENARIOS_FOP_SERVER_DOMAIN_ESCENARIO_FOP;
    EscenariosFOPDytoSearch search = new EscenariosFOPDytoSearch(
            searchCompositeClass, escenarioModelClass,
            escenarioTargetClass, onSelectionCommand);
    String dialogTitle = EscenariosRcpMessages.BUSQUEDA_ESCENARIOS;
    search.setDialogTitle(dialogTitle);
    search.getDialogQueriesWithoutResults().open();
    search.setEntityName("EscenarioFOPSearch");
    return this.proceed();
}
```

Figura 39: Framento de código donde se abre la ventana de búsqueda de escenarios

De esta forma podemos disponer de diferentes vistas del mismo modelo de objeto, una vista completa que contenga toda la información y una vista simple que contiene un subconjunto de la información del modelo. El disponer de una vista resumida (implementada con mapeos diferenciales) nos permite poder recuperar objetos sin la penalización de tener que cargar todos los datos del modelo, cuando en ciertos casos la gran mayoría de los datos no son utilizados.

Caso: Redundancia de datos

Los modelos de objetos que se generan siguiendo las buenas prácticas de objetos son modelos bien desagregados permitiendo una buena expresividad del dominio que se quiere representar. Lo mismo ocurre con los modelos de base de datos, los cuales luego de aplicar una serie de reglas se generan modelos normalizados, evitando persisitir datos redundantes, protegiendo la integridad y evitando problemas de actualización de datos.
Sin embargo, cuando llevamos la teoría a la práctica nos encontramos con numerosas situaciones donde surge la necesidad de adoptar soluciones de compromiso al momento de diseñar los modelos en pos de ganar en performance de la aplicación. En el modelo de base de datos la estrategia es desnormalizar, es decir almacenar el mismo atributo en en diferentes lugares (redundar). Esta estrategia se aplica para aquellos datos que se consultan siempre juntos y de no estar en la misma tabla, requieren que se realicen joins de varias tablas para obtener el valor. El objetivo de esta estrategia es evitar optimizar ciertas consultas reduciendo la cantidad de tablas incluídas en el query de la consulta, almacenando en la tabla root de la consulta los atributos “más” accedidos habitualmente.
En el modelo de objetos se presenta la misma situación, en ciertos casos los modelos de objetos, especialmente en grandes aplicaciones, tienden a generar grafos complejos, o grafos de grandes longitudes (muchos objetos anidados) por lo que en ciertos casos algunas consultas pueden requerir acceder a atributos del objeto padre del grafo y a atributos que se encuentran a una distancia considerable del objeto padre. Esta situación a nivel de objetos es resuelva de manera simple, navegando por el grafo de objetos hasta llegar al objeto que contiene el atributo deseado . Por ejemplo si nuestro modelo contiene 3 clases con relaciones de asociación A->B->C, para acceder a un atributo de la clase C desde el objeto A (root del modelo) en notación puntual esto puede ser representado por el pseudocódigo: objetoA.objetoB.objetoC.atributo1.
Cuando estos modelos son traducidos a tablas (mapeados) generalmente se asocia una clase con una tabla en el modelo relacional, lo que implica que para realizar una consulta del objeto A filtrada por el atributo1 del objeto C implica realizar una consulta del estilo

"Select * From A …. Join B … Join C … Where atributo= ...". Esta consulta requiere realizar 3 joins entre tres tablas diferentes para poder retornar un objeto que mayormente está persistido en la tabla A. Claramente esto puede presentar problemas de performance que pueden ser evitados manteniendo una copia del atributo1 de la clase C en la clase A, lo cual trae como consecuencia que la consulta puede representarse en notación puntual como "objetoA.atributo1" y a nivel SQL como "Select * From A Where atributo1 = ….".

Ejemplo: Estado actual de un Escenario de Simulación

Volvamos al módulo de Formulación Presupuestaria de e-Sidif, específicamente a los Escenario de Simulación. Mencionamos que los escenarios de simulación permiten registrar la información necesaria durante el proceso de la formulación del presupuesto. Un escenario de simulación consta de una cabecera, componentes, elementos y etapas de escenario. Además todo escenario de simulación tiene un workflow de estados y transiciones, permitiendo definir validaciones y acciones a ejecutar en cada transición entre dos estados. Un escenario siempre se encuentra en un estado, cuando se crea se encuentra en un estado inicial y a lo largo de su ciclo de vida va pasando por distintos estados, incluso puede volver varias veces a un mismo estado hasta llegar a un estado final. Es necesario registrar los estados por los cuales va pasando el escenario y la fecha del cambio de estado. Siguiendo las buenas prácticas de modelado, un EscenarioFOP tiene una colección de objetos LogEstadoEscenario, donde cada objeto LogEstadoEscenario tiene el Estado, la fecha de cambio de estado y el usuario que lo realizó. Naturalmente los escenarios, el log de estados y los usuarios se persisten en tablas diferentes. Para recuperar los estados por los que pasó un escenario, será necesario realizar los joins correspondientes sobre las tablas mencionadas.
En el eSidif, existen muchas validaciones que necesitan consultar el estado actual del escenario. Utilizando el modelo mencionado, cada vez que se quiera obtener el estado actual se deberá hacer los joins de las 3 tablas y recuperar el log de estado con la fecha más reciente. Por este motivo, se decidió redundar el estado actual del escenario en la cabecera del mismo. Este atributo se actualiza cada vez que el escenario cambia de estado. La Figura 40 muestra el modelo de objetos de un Escenario de Simulación con el atributo redundado y la asociación con el Log de cambio de estados.

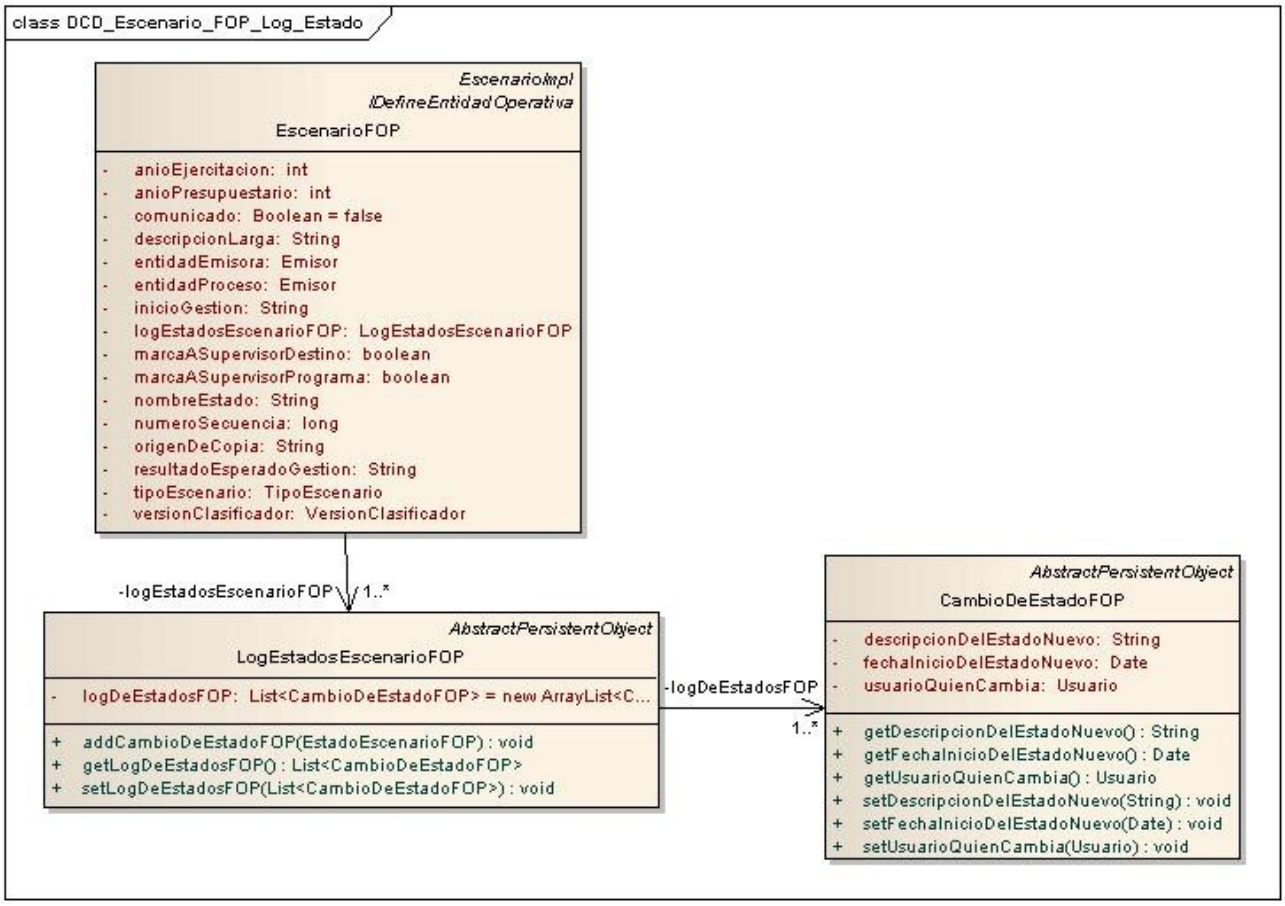


Figura 40: Clase Escenario y LogEstadoEscenario

Ejemplo: Consulta de Estado del Crédito

Veamos otro ejemplo de redundancia de datos en e-Sidif. El presupuesto está organizado operativamente en Partidas Presupuestarias. Cada Organismo tiene sus propias Partidas Presupuestarias para ejecutar, y a su vez organizadas en sus distintas Unidades Ejecutoras. El presupuesto se ejecuta trimestralmente. El crédito de las Partidas Presupuestarias de cada trimestre se denomina Cuota.

Una Partida Presupuestarias se conforma con la combinación de siete Clasificadores Presupuestarios:

- El Institucional indica a que Organismo pertenece la Partida Presupuestaria. Está formado por los códigos: jurisdicción, subjurisdicción y entidad.
- El Objeto del Gasto indica en qué puede ejecutarse el crédito del Partida Presupuestaria. Está formado por los códigos: inciso, principal, parcial y subparcial.
- La Apertura Programática indica que acción se lleva a cabo con el crédito de la Partida Presupuestaria. Está formado por los códigos: programa, subprograma, proyecto, actividad y obra.
- La Ubicación Geográfica indica el impacto geográfico de la ejecución de la Apertura Programática. Está representado por un código numérico.
- La Finalidad y Función indica la finalidad y función del programa indicado en la Apertura Programática. Está formada por un código numérico para finalidad y otro para función.
- La Fuente de Financiamiento indica de donde se obtiene el crédito de la Partida Presupuestaria. Está formada por dos códigos: procedencia y fuente.
- La Moneda utilizada en las Partidas Presupuestarias siempre es la nacional. Está

representado por un código numérico.

- El Clasificador Económico indica el tipo de gasto que se ejecuta con la Partida Presupuestaria.Hay tres tipos de gastos: Gastos Corrientes, Gastos Capitales y Aplicaciones Financieras. Está codificado con un número de ocho dígitos.

El sistema e-Sidif permite consultar el Estado del Crédito y la Ejecución de las Partidas Presupuestarias a partir de filtros definidos por el usuario. La Figura 41 muestra los filtros de búsqueda.

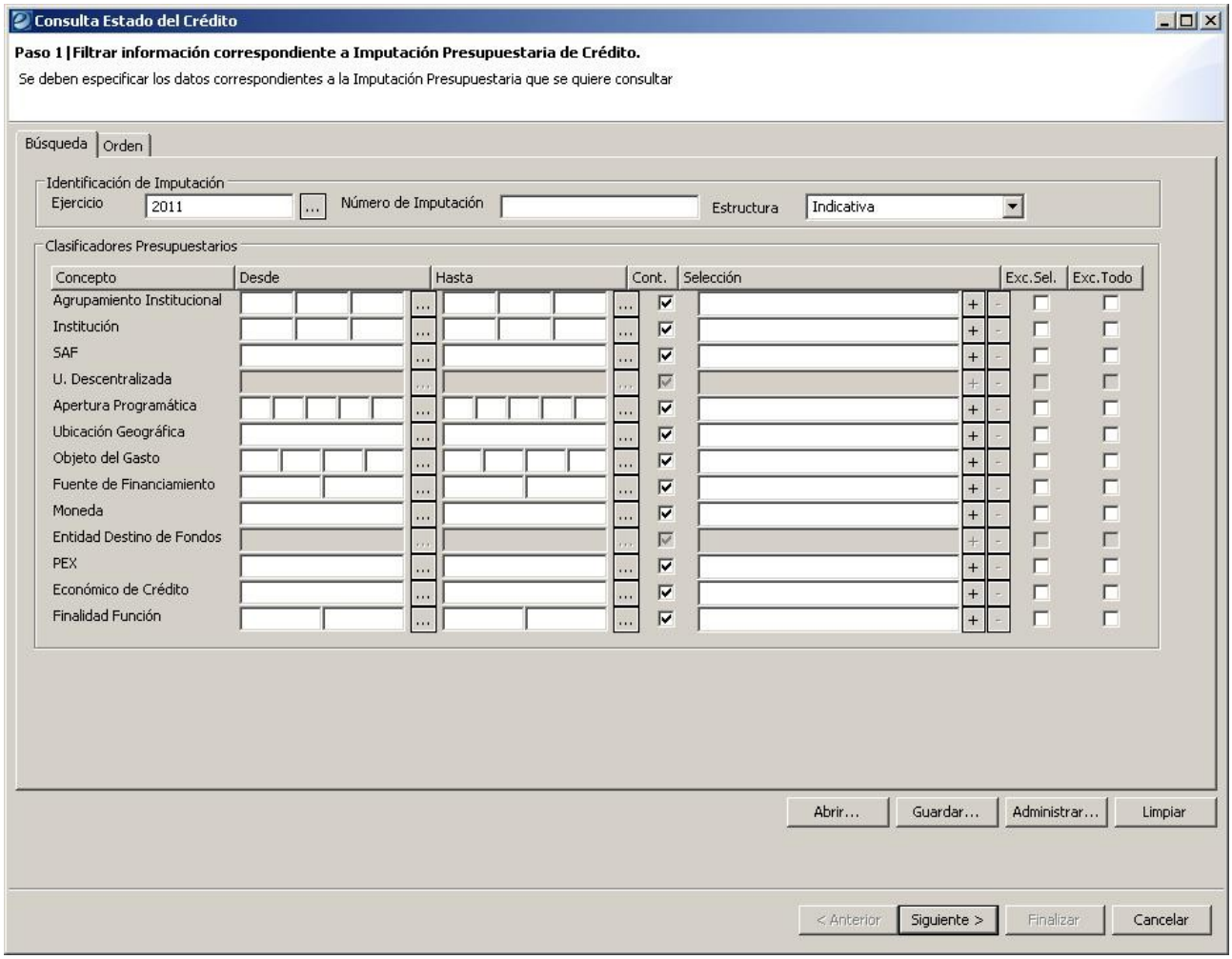


Figura 41: Filtros de la consulta de Estado del Crédito

La Figura 42 muestra un ejemplo de la consulta del crédito. Como se puede observar ademas de mostrar el estado del crédito y la ejecución del gasto de cada partida presupuestaria seleccionada, es posible ver información más detallada. Por ejemplo el detalle de movimientos diarios, la evolución del crédito, el estado de la cuota.

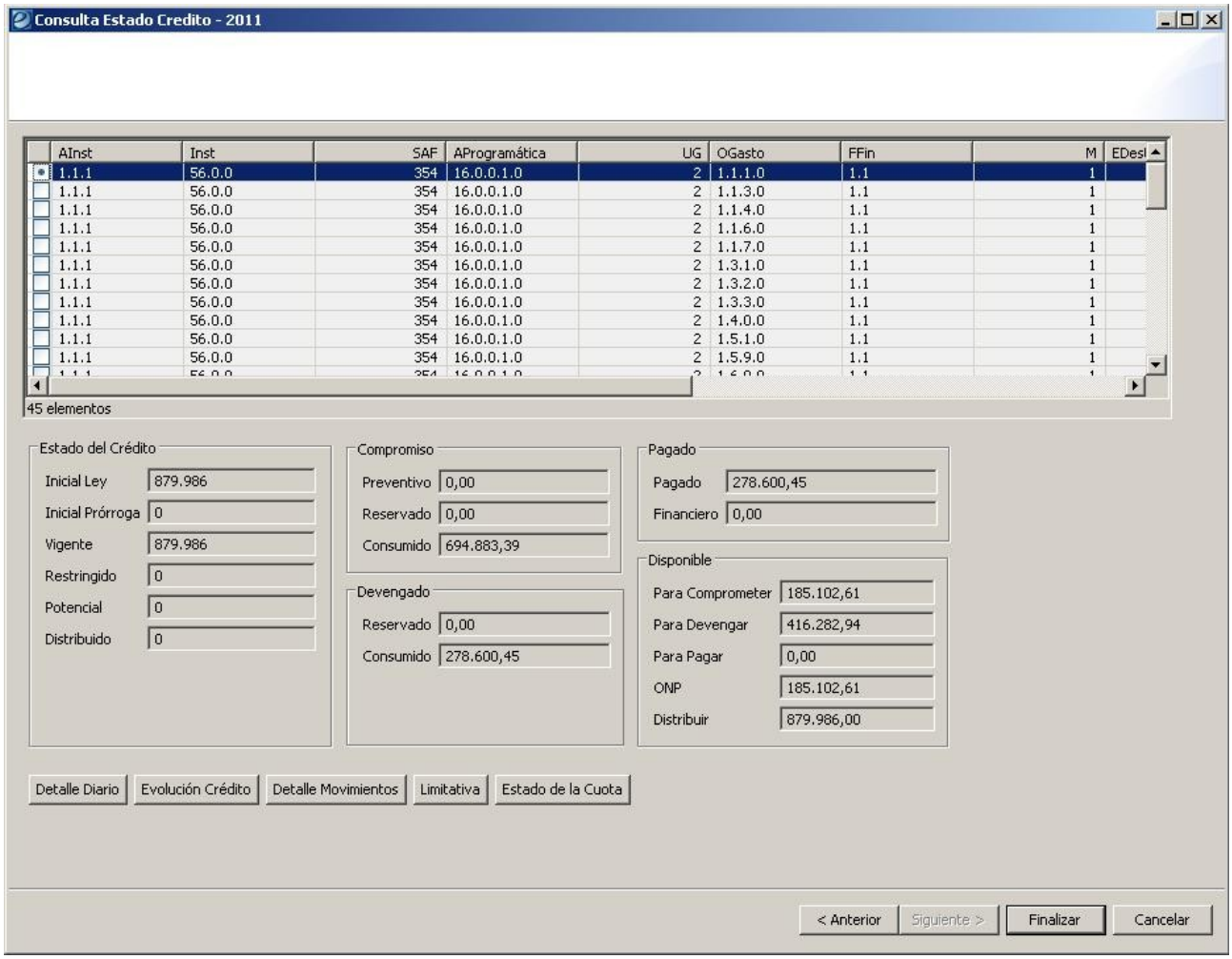


Figura 42: Ejemplo de la Consulta de Estado del Crédito

Las partidas presupuestaria de crédito se modelaron como objetos que tienen asociaciones a cada uno de los clasificadores presupuestarios que la componen. Para resolver muchas operaciones en e-Sidif, es necesario cargar en memoria gran cantidad de imputaciones presupuestarias y junto con ellas se necesitan cargar todos sus clasificadores presupuestarios. Esto es indispensable para resolver la lógica de negocio de estas operaciones. Sin embargo hay otros situaciones tambien frecuentes donde solo alcanza con cargar en memoria algunos atributos de los clasificadores presupuestarios asociados a cada imputación, en particular los códigos funcionales. En este caso, traer el grafo completo de objetos asociados a cada imputación presupuestaria resulta innecesario y puede ocasionar problemas en la performance en la aplicación cuando la cantidad de imputaciones en memoria es considerable.
Un ejemplo de funcionalidad donde ocurre esta última situación es en la consulta del Estado del Crédito. Para implementar esta funcionalidad solo necesitamos en memoria los códigos funcionales de los clasificadores de cada imputación presupuestaria para que el usuario pueda identificar visualmente cada imputación en la consulta y ver el estado del crédito y la ejecución del gasto asociados. La solución a este problema fue redundar los códigos funcionales de los clasificadores en la imputación presupuestaria de crédito, es decir agregar como atributos de la clase ImputacionPresupuestariaCredito los códigos de cada clasificador. La Figura 43 muestra la clase ImputacionPresupuestariaCredito con los clasificadores propiamente dichos y los códigos redundados.

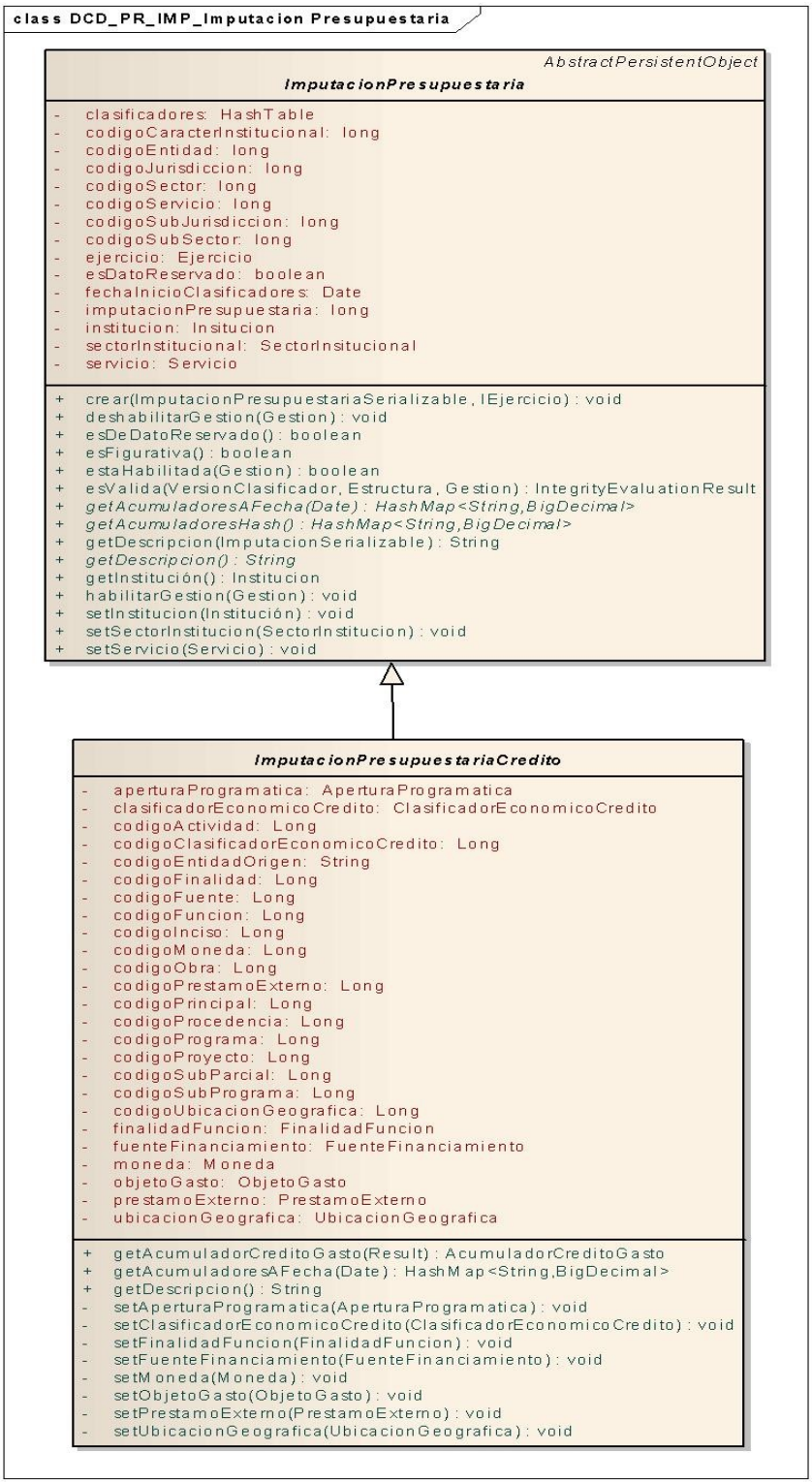


Figura 43: Clase ImputaciónPresupuestariaCredito

De esta manera solo se requiere cargar en memoria los objetos ImputacionPresupuestariaCredito sin necesidad de cargar sus clasificadores presupuestarios. Esto reduce condiderablemente la cantidad de objetos en memoria y en consecuecia mejora la performance de la consulta.
Como beneficio podemos ver que la cantidad de tablas que se accede puede ser considerablemente menor con una mejora sustancial en la performance. Como contra genera la necesidad de manejar la copia de los atributos redundados en todos los procesos de la aplicación. Esta estrategia debe ser implementada con sumo cuidado y

solo en los casos en los que la performance sea muy crítica para la funcionalidad de la aplicación afectada por este modelo, en cualquier caso es de suma importancia que la decisión de adoptar esta estrategia sea tomada siempre en etapas de diseño previas a la implementación de modo que se puedan preveer todas los implicancias que la redundancia trae en otros procesos y que esta estrategia no sea adoptada como solución posterior a la detección de un problema de performance.

Caso: Consultas en base de datos y en memoria

El sistema e-Sidif, como toda aplicación enterprise, maneja un gran volumen de datos, esta característica nos exige manejar con cuidado las consultas que se generan a partir de las operaciones de tipo Retrieve que realizamos utilizando el ORM. En nuestro caso de estudio el ORM utilizado es Hibernate.
Todo lo especificado en los archivos de mapeo a partir de las declaraciones de clases persistentes será utilizado por Hibernate para generar las sentencias SQL sobre las tablas necesarias para la recuperación de los objetos. El lenguaje de mapeo debe ser suficientemente flexible, ofreciendo facilidades para optimizar las consultas SQL generadas.
La carga de un objeto a memoria se puede realizar mediante:

- Retrieve(oid): Recupera un objeto a partir de su id
- Navegar un objeto ya cargado a través de sus relaciones

Estos dos mecanismos de recuperación de objetos son suficientes para alcanzar cualquier objeto que necesitemos. Pero aquí aparece nuevamente el problema de la performance.
La forma más natural y simple de resolver las consultas es disponer de todo el modelo de objetos en memoria y escribir las consultas usando las buenas prácticas y mecanismos del paradigma orientado a objetos. Sin embargo, en aplicaciones con grandes volúmenes de datos ésta no es una buena solución.
Por ejemplo, si se requiere filtrar un conjunto de objetos que cumplen ciertas condiciones, siendo que solo 100 de 100000 son los que la cumplen, no parece ser una buena solución a nivel performance, cargar los 100000 objetos a memoria y filtrar en memoria aquellos objetos que cumplen las condiciones. Una alternativa mejor sería realizar una consulta que solamente recupere de la base de datos los 100 objetos que cumplen las condiciones.

Componente Filtrado

Existen situaciones donde las condiciones por las cuales se necesita filtrar no son simples y por este motivo no pueden traducirse directamente a una condición WHERE en una consulta SQL. Ante estas situación donde no es posilbe posible filtrar los 100 objetos en una sola consulta, la alternativa sería hacer una consulta que recupere un conjunto de elementos que cumplen con algunas de las condiciones, reduciendo la cantidad de elementos que se levantan en memoria, y terminar de filtrar en memoria las condiciones que no se pudieron hacer en la base de datos.
Para poder dar soporte a esta última alternativa de solución, en e-Sidif se implementó una clase denominada *Search* que representa una búsqueda. Si bien el corazón de esta clase Search, es la clase Criteria de Hibernate, se agrega funcionalidad adicional para permitir dar soporte a búsqueda complejas que pueden poner en riesgo la performance de la aplicación.
Una funcionalidad adicional es la capacidad de poder configurar una condición para que pueda ser ejecutada en memoria o en base de datos. De esta manera, cuando se solicita la ejecución de un Search, se colectarán las condiciones de base de datos a partir de las

cuales se armará el criteria correspondiente. Una vez ejecutado el criteria, y su consecuente consulta SQL a la base de datos, se aplicarán cada una de las condiciones (filtros) de memoria al resultado de haber ejecutado el criteria a través de Hibernate. Esto es muy útil para los casos donde existen:

- Condiciones que son mucho más simples de expresar en memoria o que no puede ser resueltas en la base de datos y que no son condiciones que hacen el filtro grueso (volúmen)
- Condiciones más sencillas de traducir a SQL y que realizan el filtro más grande de los datos, razón por la cual deberían ser aplicadas en la base de datos para no perjudicar la performance.

Ejemplo: Aperturas Programáticas

Las Aperturas Programáticas son clasificadores presupuestarios que permiten reflejar "el proceso de producción o provisión de la jurisdicción o entidad". Las aperturas programáticas se definen jerárquicamente mediante la combinación de estas categorías: programa, subprograma, proyecto, actividad y obra.
Las Imputaciones Físicas reflejan la vinculación entre las Aperturas Programáticas y las producciones de cada jurisdicción o entidad y la unidad de medida de las mismas, permitiendo el seguimiento presupuestario y la cuantificación de metas físicas "de los resultados de los procesos productivos, de los insumos requeridos por los mismos, de los bienes y servicios que produce y provee" cada organismo del Estado.
Una imputación física se identifica por un conjunto de clasificadores presupuestarios y entidades básicas (servicio, institución, apertura programática, unidad de medida, medición física, ubicación geográfica y moneda) y por un ejercicio, una versión de clasificadores de simulación y una gestión a la cual pertenece.
Cada imputación física puede corresponder a un ejercicio presupuestario o a una versión de clasificadores de simulación. Las gestiones permiten administrar la vigencia por gestión de las entidades básicas y clasificadores. Las gestiones definen un árbol jerárquico que se muestra en la Figura 44.

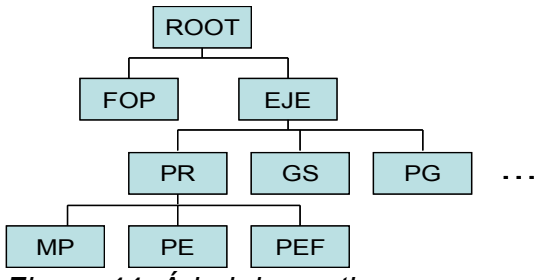


Figura 44: Árbol de gestiones

Si la imputación física pertenece a un ejercicio, la versión de clasificadores es nula y la gestión será alguna de las gestiones del subárbol EJE. Si la imputación tiene versión de clasificadores de simulación, el ejercicio respresenta al ejercicio de la versión y su gestión es FOP.
La forma de determinar si una imputación física se encuentra vigente para una gestión, depende de la gestión. Si la gestión en alguna de las gestiones del subárbol EJE, se dice que la imputación está vigente si su estado es 'Habilitado'. Si la gestión es FOP, la imputación está vigente si su estado es 'Habilitado' y la fecha de baja es mayor a la fecha de hoy.
La Figura 45 muestra el modelo de clases de las imputaciones físicas.

Figura 45: Clase ImputacionFisica

En nuestro ejemplo, queremos recuperar las imputaciones físicas que pertenecen al ejercicio 2011, al programa 16 y que se encuentren vigentes en la gestión EJE (es decir en la gestión EJE o cualquiera de sus dependientes en la jerarquía).
Un dato importante es que existen aproximadamente 100000 imputaciones físicas por ejercicio y solo 1500 cumplen las condiciones de filtrado. Esto imposibilita traer a memoria todas las instancias de imputaciones físicas y filtrar en memoria aquellas que cumplen nuestras condiciones de negocio. Tampoco es posible traducir todas las condiciones para que sean aplicadas en la base de datos y traer a memoria solo las imputaciones que cumplen todas las condiciones. En este caso necesitamos implementar un solución mixta, con algunas condiciones en momoria y otras en base de datos.

- Las condiciones *(ejercicio = 2011) and (programa = 16) and (estado = Habilitado)* se implementarán como conditions del search que se ejecutarán en la base de datos.
- La condición que filtra si la apertura está vigente para la gestión EJE se implementará en memoria.

El método que implementa el seach completo, es decir con todas las condiciones se muestra en la Figura 46.

```
static public ISearch searchImputacionesFisicas(Long ejercicio, Long programa, Gestion gestion) {
  ISearch search = Search.forTargetClass(ImputacionFisica.class);
  search.addCondition(Conditions.eq("ejercicio.ejercicio", ejercicio));
  search.addCondition(Conditions.eq("aperturaProgramatica.programa", programa));
  search.addCondition(Conditions.eq("estado", ElementoEstado.INGRESADO));
  Condition condicionGestion = new ElementoBasicoVigenteEnGestionCondition(gestion);
  condicionGestion.setInMemory(true);
  search.addCondition(condicionGestion);
  return search;
}
```

Figura 46: Fragmento de código que implementa el Seach

A continuación, la Figura 47 muestra un fragmento de la clase que implementa la condición de filtrado de los elementos básicos que están vigentes en una gestión determinada.

```
ElementoBasicoVigenteEnGestionCondition extends AbstractCondition implements Condition {
...
public boolean evaluate(final Object element) {
      if (this.getGestion() == null) {
       }
       if (((ElementoBasico) element).vigenteEnGestion(gestion)) {
         return true;
       }
       return false;
}
...
```

Figura 47: Fragmento de código que implementa la condición en memoria

La Figura 48 muestra fragmentos de la clase ElementoBasico, superclase de ImputacionFisica, donde se implementa la lógica de negocio necesaria para saber si un elemento básico está vigente en una gestión.

```
public boolean vigenteEnGestion(final Gestion gestion) {
  if (this.getEnFop()) && this.estaVigente())
    return true;
  else
    return (this.getEnEjecucion(gestion));
}


private boolean estaVigente() {
  Date fechaYHoraActuales = ((DateTimeServiceBI) (GenericServiceFactory.getInstance()
      .getService(DateTimeServiceBI.class))).getDateTime();
  return (this.getFechaBaja() == null || this.getFechaBaja().compareTo(fechaYHoraActuales) > 0);
}
```

Figura 48: Fragmento de código que implementa la lógica de negocio utilizada por la condición en memoria

Este es un ejemplo de un gran conjunto de casos del sistema e-Sidif en los cuales se aplica esta solución. El problema que motivó a implementar esta solución fue la necesidad de cargar en memoria gran candidad de objetos como resultado de consultas sobre estos objetos con un conjunto de condiciones de filtrado. Las condiciones de filtrado que se presentan en e-Sidif son muy variadas y complejas. Existen condiciones fijas definidas por la lógica de dominio, condiciones dinámicas definidas por el usuario durante la operatoria de la aplicación, filtros de valores, filtros de rango de valores, filtro de listas de valores, filtro sobre códigos jeráquicos de los clasificadores presupuestarios, etc.

La complejidad en la lógica de los filtros y la cantidad de objetos que se deben recuperar en memoria para filtrarlos posteriormente son situaciones que generan problemas de performance. Por este motivo se implementó esta solución mixta, permitiendo reducir la cantidad de objetos que se traen a memoria y aprovechando la expresividad de los objetos y mensajes para realizar los filtros complejos o filtros que no se pueden implementar utilizando los lenguajes de consultas orientado a objetos, HQL ó SQL.

Capítulo VIII - Conclusiones y trabajos futuros

En los capítulos anteriores se analizaron las características principales de las aplicaciones que deben trabajar con grandes volúmenes de datos, los problemas de performance mas comunes y las dificultades a las que se enfrentan estas aplicaciones cuando se utilizan mapeadores objeto/relacional. En esta sección veremos un resumen de las conclusiones generales de los problemas y alternativas de solución planteadas.

Conclusiones finales

En esta sección se presenta una serie de tareas realizadas en este trabajo junto con un resumen del aporte realizado durante la elaboración del mismo.

En el primer capítulo analizamos las características principales de las aplicaciones de tipo enterprise, es decir aplicaciones que contienen gran cantidad de funcionalidad y que para los objetivos de esta tesis trabajan con grandes volúmenes de datos. Dentro de estas aplicaciones se presentaron las arquitecturas clásicas, siendo el foco principal de esta tesis las aplicaciones N-capas, donde existe una clara separación entre la lógica de presentación, lógica de negocio y lógica de acceso a datos. Pudimos observar que tanto en aplicaciones con estas características arquitectónicas como en otro tipo de aplicaciones la separación y abstracción total del acceso a datos de la lógica de negocio y la lógica de acceso a datos es prácticamente imposible, esto hace que sea de suma importancia tener en cuenta el volúmen de datos de los procesos de negocio en el momento del diseño de la aplicación.

En el segundo capítulo analizamos en detalle los modelos fundamentales de las aplicaciones orientadas a objetos que trabajan con bases de datos relacionales, es decir, mencionamos las características principales de los modelos de objetos y los modelos relacionales. Una vez presentados ambos modelos destacamos que existe una diferencia entre ellos lo que obliga a definir un mecanismo para romper esta diferencia (ajuste de impedancia). Este proceso tradicionalmente generaba un conflicto entre ambos mundos, pero durante esta tesis vimos técnicas que permiten tomar decisiones de compromiso para generar una biyección entre ambos modelos. Como fue presentado anteriormente, en aplicaciones que hacen uso intensivo de base de datos y especialmente aquellas que requieren trabajar con grandes volúmenes de datos, es necesario que el proceso de diseño de objetos esté adaptado para lograr una optimización del acceso a datos. Si bien vimos que existen mecanismos que permiten mantener modelos de objetos puros, es decir modelos que no fueron sometidos a decisiones de compromiso que “rompen” las premisas básicas del diseño orientado a objetos, se analizó que las mejores alternativas son aquellas en donde ambos modelos (objeto y relacional) son adaptados y acercados uno a otro, generando modelos relacionales no totalmente normalizados y modelos de objetos que puedan ser mapeados utilizando mapeadores de modo que el acceso a datos sea lo más performante posible.

En el tercer y cuarto capítulo analizamos los conceptos principales de las técnicas de mapeo objeto/relacional y las características técnicas generales de los principales framework de mapeo relacional. Como pudimos ver, los principales frameworks permiten mapear casi cualquier características de un modelo de objetos a un modelo relacionar

permitiendo romper la barrera que separa estos dos mundos, analizamos las diferentes aproximaciones para lograr esta unión entre los modelos y los problemas que existen con cada aproximación. También enumeramos los beneficios que provee un mapeador objeto/relacional así como las situaciones ante las cuales no se recomienda su uso. Si las técnicas utilizadas son las correctas y se diseña un modelo de objetos adaptado a las necesidades de performance para el tipo de aplicaciones que hace foco esta tesis, el uso de los mapeadores contribuye notablemente a la eficiencia en la construcción de aplicaciones, generando una buena separación entre la lógica de negocio y la complejidad del acceso a datos y permitiendo a los procesos trabajar con modelos orientados a objetos y todos sus beneficios. Sin embargo, existen casos en donde a priori se deben considerar otras alternativas a los mapeadores, por ejemplo, aplicaciones que requieran análisis tipo OLAP, donde las bases de datos no se encuentran normalizados y la diferencia entre los modelos es demasiado grande y aplicaciones que utilizan bases de datos legadas (legacy) que ya contienen gran parte de la lógica de negocio implementada y estructurada en la forma de prodedimientos almancenados. Como punto final vimos que dentro de los aspectos a analizar al momento de elegir un mapeador hay algunos que son de suma importancia como las prestaciones y lenguajes de consultas, el estilo de mapeo (metadata, código, autogestionado), el impacto que tiene en el mantenimiento de la aplicación especialmente en aplicaciones que presentan modelos de dominio con gran cantidad de clases y el overhead que se genera durante el proceso de hidratación del modelo de objetos a partir del modelo relacional. Se debe tener presente que la buena elección de un mapeador objeto/relacional puede generar una reducción del 30 a 40% del código de la aplicación.

En el quinto capítulo analizamos los problemas clásicos que se presentan en aplicaciones que utilizan frameworks de mapeo objeto/relacional y especialmente los problemas que se generan en aplicaciones que deben manejar grandes volúmenes de datos. Analizamos el problema de los N+1 accesos que se presenta típicamente en modelos de composición con listas de objetos. Este problema es recurrente en todos los frameworks de mapeo y como se presenta en el sexto capítulo, todos proveen alternativas de solución trasparente para el modelo de objetos, también analizamos el problema de la ejecución de operaciones masivas, acceso concurrente al mismo objeto y versionado, y problemas relacionadas con la ejecución de consultas complejas siendo este último un problema “doble” que contiene los problemas asociados con consultas complejas en el modelo de objetos y consultas simples en el modelo de objetos que generan consultas complejas en el modelo relacional. Como conclusiones generales de los problemas presentados pudimos determinar que cada uno tiene un foco de ataque distinto, en algunos casos es necesario prevenir el problema desde la etapa de diseño del modelo de objeto (tomando decisiones de compromiso) y en otros se solucionan realizando optimizaciones al framework de mapeo objeto/relacional que se está utilizando. En todos los casos, cada uno de estos problemas debe ser analizado en etapas muy tempranas del ciclo de vida del desarrollo de los sistemas, eligiendo las técnicas correctas de solución de manera preventiva y evitando tener que realizar ajustes cuando el problema ya esté instalado.

En el sexto capítulo presentamos diferentes patrones y estrategias de solución a los problemas de performance descriptos en el capítulo anterior, determinando que la aplicación de cada patrón puede solucionar uno o varios de los problemas anteriormente mencionados. Dentro de las estrategias analizadas algunas atacan los problemas desde la definición de restricciones para el proceso de diseño del modelo de objetos, mecanismos de optimización puntuales para los frameworks de mapeo y patrones de arquitectura que pueden influenciar positivamente a la performance de aplicaciones enterprise. Entre las estrategias descriptas se encuentran las técnicas de lazy loading que

permite realizar una carga diferida de parte de un modelo de dominio hasta su utilización concreta o técnicas de edger-fetching que por el contrario permiten realizar una carga anticipada de un modelo, optimizando la cantidad de accesos a la base de datos. También analizamos estrategias relacionadas con la utilización de caché tanto la utilización de caché para una operación puntual como la relación que tiene esta técnica con aplicaciones enterprise que generalmente son utilizadas por gran cantidad de usuarios de manera concurrente. Otra estrategia presentada es la utilización de vistas SQL para realizar mapeos objeto/relacional contra abstracciones o simplicaciones de modelos de datos complejos, para casos puntuales como la consulta de objetos necesario para la generación de reportes, utilización de procedimientos almacenados para operaciones como actualizaciones masivas en pro de la optimización de los tiempos de respuestas y un mejor control transaccional. También analizamos la adopción de motores de indexación de objetos que permiten reducir la complejidad de las consultas a la base de datos para la determinación de los objetos que cumplan con determinadas condiciones, generalmente complejas, y diferir el acceso a la base de datos solo para recuperar objetos concretos y no para realizar búsquedas complejas. Otra estrategia presentada fue la adopción de repositorios (bases de datos) offlline, que almancenen información histórica permitiendo que el repositorio online contenga solo información de soporte a los procesos diarios de la aplicación logrando que la aplicación trabaje con un menor volumen de objetos en su operatoria normal. También analizamos diferentes criterios de desnormalización de la base de datos que permiten obtener modelos de objetos “puros” mapeados a bases de datos no normalizadas con el objetivo de lograr un acceso a los datos más eficiente y analizamos las consecuencias que esta técnica tiene en la lógica de la aplicación.

Como conclusiones generales podemos decir que adoptando las técnicas correctas de optimización, la utilización de mapeadores objeto/relacional presenta grandes beneficios para las aplicaciones, pero es necesario trener en cuenta algunos aspectos fundamentales a la hora de analizar los requerimientos funcionales y no funcionales y al momento de comenzar el diseño de la aplicación:

- Es necesario diseñar los modelos de objetos y relacional de modo que la diferencia que existe entre ambos tenga la distancia suficiente como para que pueda ser unida con un mapeador y que se puedan aplicar las técnicas de optimización correcta. Esto implica que los modelos de objetos pueden no ser puros y las bases de datos pueden no estar completamente normalizadas.
- La utilización de mapeadores provee grandes beneficios a las aplicaciones pero no son la solución a todos los casos de acceso a datos. En este tipo de aplicaciones existen situaciones en donde es conveniente adoptar técnicas de acceso a datos que no inluyen mapeadores objeto/relacional.
- La arquitectura general de las aplicaciones deben estar diseñada e implementada de modo que las técnicas mencionadas puedan ser adoptadas sin requerir una reingeniería de la aplicación.

El uso de mapeadores es un factor determinante de una aplicación enterprise y se debe tener en cuenta en todo el proceso de diseño del modelo de dominio y la arquitectura de la aplicación, debemos dejar de lado los aspectos puros de diseño de objetos y diseño relacional para obtener los aspectos positivos de cada uno y conbinarlos para lograr los mejores resultados.

Trabajos Futuros

De los aspectos analizados y mencionados en esta tesis se desprenden varios temas que pueden ser considerados como una ampliación de este trabajo. Algunos de estos temas están relacionados al análisis, otros temas se refieren a actividades prácticas como tareas de definición en etapas de diseño e implementación y otros hacen referencia a tareas relacionadas con el proceso de toma de decisión sobre la utilización de tecnología de mapeadores.

A continuación se presenta una lista de los temas que pueden ser analizados como continuación de esta tesis:

- **Análisis de performance de los principales frameworks**: Realizar un análisis detallado de performance de los principales frameworks de mapeo objeto/relacional, analizar como resuelven cada uno de estos framework aspectos como ejecución de consultas, recuperación de objetos, hidratación, lazy loading, integración con cachés.
- **Mapeadores vs BDOO**: Comparación entre mapeadores objeto/relacional y bases de datos orientadas a objetos, analizando las diferentes consideraciones que se deben tener en cuenta en las aplicaciones para adoptar cada una de estas tecnologías, realizar una análisis comparativo de las prestaciones de las principales herramientas que dan soporte a estos conceptos.
- **Nuevos problemas de performance y soluciones:** Presentar detalladamente otros problemas de performance que pueden presentarse en aplicaciones enterprise y no fueron abordados en este trabajo junto a sus posibles soluciones.
- **Implementación de las estrategias de optimización:** Implementar algunas de las estrategias de solución planteadas en la tesis, de manera tal que dichas soluciones sirvan para resolver problemas similares en aplicaciones con las mismas características.
- **Bases de datos noSQL + objetos**: Analizar las características de las bases de datos no relacionales y como éstas pueden ser utilizadas en contextos de aplicaciones con modelos orientados a objetos. Determinar la posibilidad de adaptar frameworks de mapeo existenes para persistir información en bases de datos no estructuradas tipo key-value de alta performance.
- **Performance en el ciclo de vida de un proyecto:** Incorporar procesos de performance a lo largo del ciclo de vida de la aplicación, lo cual permitirá alcanzar y encontrar métricas de performance efectivas para la organización.

Referencias bibliográficas

1: Vishy Narayan, Enterprise Application Performance Management: An End-to-End perspective, 2006, http://www.infosys.com/infosys-labs/publications/Documents/enterprise-application-performance-management.pdf
2: Melek Oktay, Ayşe Betül Gülbağcı, Mustafa Sarıöz, Architectural, Technological and Performance Issues in Enterprise Applications, 2007, http://www.waset.org/journals/waset/v27/v27-40.pdf
3: Martín Fowler, Patterns of Enterprise Application Architecture, 2003
4: Douglas K. Barry, Web Services and Service-Oriented Architectures, 2011, http://www.service-architecture.com/
5: Mark L. Fussell, Foundations of Object Relationa Mapping, 1997
6: Scott W. Ambler, Techniques for Successful Evolutionary/Agile Database Development, 1997-2010, http://www.agiledata.org/
7: Pablo Pizzaro, Herramientas, Recursos Humanos, Planificación y etc. Todo lo referido en la arquitectura de un Software., 2007, http://arquitectura-de-software.blogspot.com/
8: Douglas K. Barry, Impedance mismatch when mapping from a relational database, 2000-2011, http://www.service-architecture.com/object-oriented-databases/articles/impedance_mismatch.html
9: Jeffrey M. Barnes, Object-Relational Mapping as a Persistence Mechanism for Object-Oriented Applications, 2007
10: Scott W. Ambler, Crossing the Object-Data Divide, 2000, http://drdobbs.com/architecture-and-design/184414587
11: ceyusa, El problema de la diferencia de impedancia, 2006, http://www.glib.org.mx/article.php?story=20060611151541892
12: Andrew E. Wade, How to Avoid the Wall - Object Oriented Databases vs Relational Databases, 2000-2011, http://www.objectivity.com/pages/object-oriented-database-vs-relational-database/how-to-avoid-the-wall.html
13: Douglas K. Barry, Mapping layer, 2000-2011, http://www.service-architecture.com/object-relational-mapping/articles/mapping_layer.html
14: Object Architects, Patterns for Object / Relational Mapping and Access Layers, 2011, http://www.objectarchitects.de/ObjectArchitects/orpatterns/
15: Rose India Technologies, What is Persistence Framework?, 2005, http://www.roseindia.net/enterprise/persistenceframework.shtml
16: Mario Alberto Chavez, Accesos a base de datos: Patrón ActiveRecord, 2008, http://mario-chavez.blogspot.com/2008/09/accesos-base-de-datos-patrn.html
17: Antonio Goncalves, A brief history of Object Relational Mapping, 2008, http://www.jroller.com/agoncal/entry/a_brief_history_of_object
18: Anónimo, Enterprise JavaBean, 2001-2011, http://en.wikipedia.org/wiki/Enterprise_JavaBean
19: Frans Bouma, Solving the Data Access problem: to O/R map or not To O/R map, 2004, http://weblogs.asp.net/fbouma/archive/2004/10/09/240225.aspx#tableapproach
20: Rod Johnson y Jim Clark, Of Persistence and POJOs: Bridging the Object and Relational Worlds, 2005
21: Mario Van Damme, Best Practices for Object/Relational Mapping and Persistence APIs, 2006
22: Oracle, Oracle9i Database Concepts: Data Concurrency and Consistency, 2003