

PERSISTENCIA ORIENTADA A OBJETOS

Prof. Mg. Javier Bazzocco

Persistencia Orientada a objetos



Prof. Mg. Javier Bazzocco

2011

Bazzocco, Javier

Persistencia orientada a objetos. - 1a ed. - La Plata : Universidad Nacional de La Plata, 2012.

E-Book.

ISBN 978-950-34-0821-6

1. Informática. 2. Bases de Datos. I. Título

CDD 005.3



Editorial de la Universidad Nacional de La Plata (Edulp)

47 N.º 380 / La Plata B1900AJP / Buenos Aires, Argentina

+54 221 427 3992 / 427 4898

editorial@editorial.unlp.edu.ar

www.editorial.unlp.edu.ar

Edulp integra la Red de Editoriales Universitarias (REUN)

Primera edición, 2011

ISBN N.º 978-950-34-0821-6

Queda hecho el depósito que marca la Ley 11.723

©2011 - Edulp

Impreso en Argentina

Índice

Introducción	6
¿A quién está dirigido?.....	6
Pre-requisitos.....	6
Objetivos y contenidos	7
Capítulo I.....	11
Características deseables de la persistencia de objetos	11
Performance sin compromisos	12
Transparencia	12
Código sql	13
Dependencia de clases específicas	14
Diseño débil o "poco" orientado a objetos	15
Ortogonalidad.....	15
POJO	16
Persistencia por alcance	17
Operaciones C.R.U.D.....	21
Capítulo II	23
Propiedades A.C.I.D.	23
Transacciones	25
Lectura de la información	27
Esquema de lockeo.....	32
Versionamiento	33
Versionamiento de clases	33
Versionamiento de instancias	34
Capítulo III	36
¿Qué es una base de datos orientada a objetos?	36
Dudas más frecuentes.....	37
¿Cuáles son los beneficios de las BDOOs?.....	37
¿Se utilizan en ambientes productivos realmente?.....	39
¿Soportan grandes volúmenes de información?	39
¿Las BDOOs reemplazarán a las bases de datos relacionales?	40
¿Pueden coexistir bases de datos orientadas a objetos y bases de datos relacionales?	41
¿Se pueden utilizar como soporte de persistencia de aplicaciones Web?	41
¿Existe mucha oferta?	42

Manifiesto	42
Características mandatorias	43
Características opcionales	44
Decisiones libres	45
ODMG.....	45
OQL.....	47
Principales características	47
Estructura	48
Path expressions	50
Invocación de operaciones	51
Polimorfismo	52
Capítulo IV	53
Diferencia de impedancia.....	53
Características deseables del mapeo	55
¿Qué se mapea y que no?	57
Mapeo de clases	58
Estrategias de mapeo de jerarquías	59
Mapeo de la jerarquía a una sola tabla	60
Mapeo con una tabla por clase	62
Mapeo de las clases concretas únicamente.....	64
Mapeo de asociaciones, composiciones y agregaciones	66
Asociaciones 1 a 1.....	66
Asociaciones 1 a n.....	67
Asociaciones n a n.....	67
Mapeo del identificador	68
Hibernate	69
Arquitectura.....	70
Forma de trabajo.....	71
Principales componentes.....	72
Ejemplo simple de uso	75
Consultas	82
Capítulo V	85
Introducción	85
Principales elementos.....	87
PersistenceManagerFactory	87

PersistenceManager.....	87
Query.....	89
Transaction.....	89
Modelo de trabajo	90
Consultas.....	92
Iteradores.....	93
JDOQL	94
Ejemplos.....	97
Las operaciones C.R.U.D. y el PersistenceManager.....	98
Productos.....	103
Capítulo VI.....	105
Root object	105
Proxy	108
Lazy loading.....	109
Decorator.....	112
Repository pattern	115
DTO.....	117
Capítulo VII	122
Inversión de control.....	122
Inyección de dependencias.....	122
Spring.....	124
Wicket.....	125
ITeM.....	126
Modelo	127
Mapeo Hibernate.....	129
Base de datos.....	131
Repositorios.....	132
Servicios	133
DTOs.....	135
Formularios	136
Servidor de aplicaciones.....	137
Spring	138
Bibliografía	141
Artículos.....	141
Referencias.....	141

Introducción

¿A quién está dirigido?

Este libro está dirigido a todas aquellas personas que por su profesión o área de estudio tienen vinculación directa o indirecta con el diseño e implementación de aplicaciones bajo el paradigma orientado a objetos y que requieran la persistencia de la información en "bases de datos" o "repositorios".

También está dirigido a todos los alumnos de las materias de grado y postgrado en las que he tenido la suerte de participar, a modo de agradecimiento por todo el conocimiento que me brindaron a lo largo de estos últimos 10 años.

Si bien en algunos tramos del libro se presentan ciertos detalles de carácter técnico, no es objetivo de este libro centrarse solamente en la comunidad de desarrolladores. Diseñadores de objetos, administradores de bases de datos, arquitectos de software e incluso perfiles tradicionalmente no-técnicos como Analistas y/o Líderes de proyecto deberían poder encontrar valor en las enseñanzas aquí recogidas.

Pre-requisitos

Este libro tiene muy pocos requisitos para su comprensión. Básicamente se asumen las siguientes "condiciones" para los lectores:

1. Conocimientos del paradigma orientado a objetos.

2. Conocimientos básicos de patrones de diseño orientado a objetos (no es fundamental pero sí es deseable).
3. Conocimientos básicos de la temática general de bases de datos. Si bien no es necesario un conocimiento profundo, sí se requiere comprender y manejar cómodamente temas como transacciones, propiedades A.C.I.D., bloqueo, etc.
4. Una mente abierta y "hambrienta", capaz de aceptar un nuevo enfoque "no tradicional" para persistir información administrada por sistemas orientados a objetos.
5. Finalmente, es deseable un mínimo conocimiento del lenguaje Java, tanto para interpretar el código que se vaya presentando así como el ejemplo completo que se brinda como muestra de implementación de todos los conceptos de este libro.

Objetivos y contenidos

Prácticamente desde la aparición del paradigma orientado a objetos ha existido la necesidad de persistir la información administrada por los sistemas desarrollados bajo dicho paradigma. Desde entonces la tecnología ha ido evolucionando (y retrocediendo a veces) hasta llegar a un punto en el cual existen múltiples alternativas para lograr la tan ansiada persistencia. Sin embargo, lamentablemente algunas de dichas alternativas se han impuesto por sobre otras de mayor calidad y beneficios. Las razones detrás de este fenómeno se pueden encontrar en la falta (aún hoy en día) de familiaridad con el paradigma orientado a objetos; la falta de conocimientos de técnicas de diseño apropiadas y por sobre todo la carencia de material de estudio que proponga mejores alternativas a las existentes actualmente.

Es realmente interesante notar que de vez en cuando aparecen conceptos, usualmente simples en su definición, pero que tienen un profundo impacto en la forma de diseñar, implementar y por sobre todo plantear soluciones de software. En el campo de la persistencia, y en particular de la persistencia orientada a objetos, existe un concepto

que ha sido pasado por alto, por no decir menospreciado en su importancia. Se trata de un concepto simple, pero que sin embargo será utilizado como base y guía de la propuesta de este libro. En base a este concepto se irán discutiendo cada uno de los puntos importantes que deben ser considerados y tenidos en cuenta al momento de diseñar una aplicación orientada a objetos que requiera persistir su información. Sin más preámbulos, el concepto del cual estamos hablando es **persistencia por alcance**. Más adelante se harán las presentaciones formales del caso, pero como para bajar la ansiedad del lector, este concepto central establece **"todo objeto al cual se pueda navegar a partir de un objeto persistente, debe ser necesariamente persistente a su vez"**. Simple y elegante a la vez, ¿no?

Los siguientes capítulos irán presentando progresivamente temas más avanzados, partiendo de algunos conceptos teóricos elementales hasta temas específicos y técnicas de diseño orientado a objetos avanzadas relacionadas con la persistencia.

El **capítulo I** presenta algunas cuestiones teóricas de fondo que son necesarias para la comprensión de los mecanismos de persistencia, así como al mismo tiempo se van presentando cualidades que como diseñador/desarrollador se desearía que fueran provistas por la solución de persistencia seleccionada. Todos estos temas no se basan en alguna tecnología en particular, como podrían ser las bases de datos orientadas a objetos o soluciones de mapeo-objeto relacional, sino que por el contrario resultan "agnósticas" de la tecnología. Luego los capítulos específicos dedicados a las soluciones concretas harán la adaptación requerida de cada uno de los conceptos esenciales presentes en este capítulo.

En el **capítulo II** se repasan conceptos relacionados con las bases de datos y la concurrencia, como manejo de transacciones, propiedades A.C.I.D. y las diferentes situaciones que pueden presentarse al modificar clases con instancias ya "persistentes". Finalmente se presentan situaciones "extrañas" desde el punto de vista del desarrollador orientado a objetos que no está acostumbrado a trabajar en ambientes con persistencia.

El **capítulo III** presenta una breve reseña de las bases de datos orientadas a objetos, su

evolución, uso y posibilidades, siempre desde el punto de vista de la orientación a objetos. El objetivo de este capítulo no es cubrir completamente el tema de las bases de datos orientadas a objetos, sino el de presentar un esquema de persistencia que podría considerarse "puro", para luego basarnos en los conceptos presentes en este tipo de bases de datos y tratar de "trasladar" sus enseñanzas a otras alternativas de persistencia, particularmente el mapeo objeto-relacional.

Probablemente en la actualidad la alternativa más utilizada sea el mapeo objeto-relacional, razón por la cual el **capítulo IV** presenta sus características esenciales en primer término y luego se analizan las características de uno de los productos de mayor utilización y difusión, Hibernate. Nuevamente es necesario aclarar que este capítulo no pretende ser una guía exhaustiva de Hibernate, sino simplemente presentar una manera elegante y "correcta" de utilizarlo, demostrando que se pueden construir soluciones orientadas a objetos de calidad con esta herramienta.

En todas las ramas de la informática los estándares abiertos son importantes, por lo que en el **capítulo V** se aborda JDO (Java Data Objects). A pesar de que no es un nombre muy "feliz" para un estándar de persistencia de objetos (¡y no de datos!), es interesante conocer sus características y operación de modo de poder construir aplicaciones que no queden "atadas" a productos en particular, como suele suceder en el caso de la utilización de Hibernate. Incluso en este momento de auge de soluciones basadas en el esquema "cloud computing", este estándar toma importancia al ser el mecanismo de acceso preferido para la plataforma AppEngine de Google.

Desde su aparición en escena, los patrones de diseño han marcado un antes y después en el diseño y desarrollo orientado a objetos. Sería de esperar que tuvieran el mismo efecto en el dominio de la persistencia. Si bien de hecho existen muchos patrones de diseño específicos para la persistencia, no son de público conocimiento aún, por lo que el **capítulo VI** humildemente se propone presentarlos con el objetivo de que se comiencen a conocer y utilizar en forma masiva.

El enfoque de este libro en muchos pasajes puede tornarse dudoso, en cuanto a factibilidad, performance, y otros criterios igualmente importantes. La mejor manera de demostrar que este enfoque no sólo es factible, sino que incluso produce soluciones de diseño de más alta calidad, es presentando un ejemplo completo que cubra cada una de las capas de una aplicación "real". El **capítulo VII** contiene detalles de una aplicación simple, de múltiples capas, la cual implementa la totalidad de los conceptos del enfoque propuesto.

Finalmente se presentan referencias bibliográficas útiles, así como artículos y otras fuentes de información relacionadas con la persistencia orientada a objetos que pueden resultar de valor para el lector.

Capítulo I

Resumen: En este capítulo se discuten las características deseables que cualquier capa de persistencia debería proveer, de modo de no generar alteraciones y/o cambios en el modelo de objetos puro. Estas características no están relacionadas con tecnología alguna, de manera que pueden ser aplicables a cualquier alternativa de persistencia orientada a objetos.

Características deseables de la persistencia de objetos

Si se realizara una encuesta acerca de cuáles son las características deseables de las soluciones de persistencia para las aplicaciones, seguramente "performance" sería la palabra que más veces se mencionaría. Si bien es absolutamente necesario contar con una excelente (¿extrema?) performance, aún cuando no haya sido identificada como un aspecto crítico inicialmente, muchas veces se la pondera por sobre otros criterios de calidad de igual importancia.

Podría argumentarse que mantenibilidad del código, claridad del diseño, utilización de patrones (todas cualidades internas del software) no son tan importantes para el usuario (que usualmente sólo se fija en cualidades externas como el rendimiento o usabilidad); sin embargo esto no es totalmente cierto ya que las cuestiones de índole internas suelen ser la base a partir de la cual se obtienen las cualidades externas. En realidad, no es real el enfrentamiento entre "hacerlo bien" y "hacerlo rápido"; en otras palabras, es perfectamente posible conseguir una solución eficiente en términos de rendimiento, y que a la vez cumpla con todos los requisitos de un buen diseño.

Ahora bien, ¿cuáles son las cualidades deseables, más allá de la alternativa técnica concreta que se utilice? Más abajo hay una lista subjetiva de algunas de estas

características.

Performance sin compromisos

Obviamente este factor debía aparecer en la lista. La lectura que se debe hacer de su aparición en esta lista es que la solución que se implemente por ningún motivo debe poner en peligro el mejor rendimiento posible.

Es deseable que la solución posea el mejor diseño posible, así como que sea lo más mantenible y extensible posible, independientemente de la tecnología; pero de nuevo, sin que esto repercuta negativamente en el rendimiento. En conclusión, que esté bien diseñado desde el punto de vista de las técnicas de diseño y/o patrones no significa que sea ¡"lento"!

Transparencia

Esta característica establece que el diseñador/desarrollador no debería tener que estar todo el tiempo preocupándose por los aspectos ligados a la persistencia cuando esté trabajando en la lógica del negocio o dominio concreto. El escenario ideal sería que el desarrollador se enfocase solamente en los detalles propios del dominio, modelando con técnicas aplicables y reconocidas. Todo aspecto de la persistencia debería estar oculto a los ojos del desarrollador, de modo de no "desviar" no sólo su atención sino también para evitar que no se logre el mejor diseño posible en aras de cuestiones como performance (que como se mencionó en el punto anterior, es importante y a su debido momento se analizará y mejorará).

La falta de transparencia puede aparecer expresada de muchas maneras en el código, algunas veces de manera bastante obvia, y en otras ocasiones de manera muy sutil (y por lo tanto difícil de descubrir y mejorar).

Código sql

Probablemente sea muy obvio e incluso una exageración, pero está claro que en pos de obtener transparencia, sería bueno no "ver" código SQL dentro del código de los objetos de la aplicación. ¿Pero cuáles son concretamente las razones para argumentar que escribir código SQL en los objetos de dominio es "malo"?

Algunas de las razones que podemos citar son:

1. Dependencia de un motor de bases de datos en particular. Aún cuando se cuenta con un estándar SQL (SQL 92), es sabido que no todos los proveedores de software de bases de datos lo soportan o respetan totalmente. Por esta razón, el SQL de una base de datos ORACLE seguramente diferirá del específico para MySQL. Un ejemplo simple: en MySQL se puede contar una tabla denominada USER que mantenga la información de todos los usuarios de un sistema, mientras que en ORACLE, USER es una palabra reservada.
2. Dependencia de una tecnología en particular. Además del punto anterior, que pasaría si en vez de bases de datos relacionales se pretendiera utilizar una solución de bases de datos orientadas a objetos o bases de datos XML. La única alternativa sería contar con tantas implementaciones de las clases de dominio como alternativas tecnológicas de persistencia existan¹. Cabe aclarar que se podrían aplicar técnicas de diseño y ciertos patrones de diseño para mejorar la solución, pero sólo se conseguiría minimizar el problema, no eliminarlo.
3. Cambios en una capa que afectan directamente a otra capa. La aplicación de un estándar nuevo para la nomenclatura de los elementos de una base de datos no debería afectar el código de otra capa, particularmente la que contiene la lógica de negocios. Está claro que cambiar el nombre de una columna no debería impactar en cómo se resuelve cierta lógica en los objetos de dominio. Al tener el SQL embebido en los objetos, es usual ver este problema.

¹ La idea sería tener una clase User que sepa persistirse a bases de datos relacionales. Otra clase User que sepa persistirse a bases de datos orientadas a objetos, y así siguiendo. En resumen, una "variante" de cada clase para cada tipo de base de datos.

4. Para lenguajes como JAVA, el SQL no es más que una cadena de caracteres (String), con lo cual hasta que no se lo ejecute es muy difícil saber si es válido o si contiene errores. Los cambios en el esquema de bases de datos no se ven reflejados directamente en las sentencias SQL.

Se podría escribir un capítulo entero sobre las desventajas que tiene este enfoque, pero con los puntos citados anteriormente puede comprenderse el problema en términos generales.

Dependencia de clases específicas

Hace unos años atrás², en un esfuerzo por lograr un estándar para las bases de datos orientadas a objetos, el OMG (Object Management Group) estableció un estándar denominado ODMG. Si bien en términos generales fue muy positivo, imponía ciertas restricciones a la hora de diseñar soluciones persistentes. Para aquellos casos en los que un objeto debería trabajar con una colección de ciertos colaboradores, el tipo de dicha colección debía cambiarse de los tipos tradicionales de JAVA (por ejemplo Collection, Vector, etc) a DVector. Esta última clase era una definición particular del estándar ODMG, la cual debía ser implementada por cada proveedor de bases de datos. La consecuencia directa de este tipo de restricciones es que uno no puede separar totalmente las capas, particularmente las de dominio y bases de datos. Ni siquiera se podrían realizar pruebas de unidad³ ya que cada clase persistente tendría referencias a clases que requieren de un respaldo de una base de datos.

Otras soluciones incluso establecían que las clases persistentes no extendieran de Object, sino de una clase especializada PersistentObject, la cual proveía ciertas implementaciones requeridas para la persistencia.

² Más precisamente entre los años 1993 y 2001.

³ Por ejemplo a través del framework JUnit.

Diseño débil o "poco" orientado a objetos

Si finalmente bajo la capa de dominio tendremos una base de datos, e incluso tendremos también algún código SQL embebido en el código, cabe preguntarse ¿por qué modelar completamente en objetos la lógica de negocios?, si al final de cuentas la resolución de los problemas en parte pasarán por consultas SQL (con sus joins, búsquedas por claves foráneas, etc.). Bajo este supuesto (mucho más habitual de lo que cabría esperar) suelen verse diseños en los cuales hay relativamente pocas clases, que incluso no tienen relaciones con las demás clases, sino que simplemente conocen los "ids" de sus colaboradores. Este es un ejemplo de un modelo débil, que más que ser un buen diseño de objetos no es más que una representación con clases de cada una de las tablas de una base de datos.

Ortogonalidad

Cuando se menciona el término ortogonalidad, hay dos semánticas posibles asociadas a la persistencia. La primera es **ortogonalidad de persistencia**, que es la expresión formal del punto anterior que planteaba la transparencia para el desarrollador respecto de los aspectos de persistencia. Más formalmente entonces, se dice que un concepto A es ortogonal a otro concepto B si al pensar en A no es necesario tener presente B; en otras palabras, una buena solución de persistencia debería permitir que el pensar una solución a un problema de un dominio fuese ortogonal a los problemas de la persistencia.

La otra acepción que se suele relacionar con ortogonalidad es **ortogonalidad de tipo**, que establece que un objeto debería ser persistente más allá de su tipo. Nuevamente, simplificando el concepto tendríamos que toda cualquier instancia de cualquier clase debería ser persistente. Es decir que no hay clases persistentes y clases no persistentes, sino que las instancias de cualquier clase se deberían poder almacenar. Sin embargo, en la práctica este punto suele materializarse a través de la declaración de las clases persistentes en archivos con meta-data (usualmente en archivos XML), ya que no es

posible que la solución de persistencia se dé cuenta "automáticamente" como persistir nuevas instancias de clases desconocidas.

POJO

El término P.O.J.O. significa **Plain Old Java Objects**⁴. Un nombre que no explica mucho ¿no? Este nombre se acuñó años atrás para referirse a las clases Java que se diseñaban y desarrollaban y que no contenían más que el código propio del dominio. Con la explicación anterior tampoco aclara mucho, ya que cabe pensar ¿qué otro código se escribiría en una clase que no fuera código relacionado con el negocio/dominio?

Bien, un poco de historia entonces para aclarar: a partir de la necesidad de poder invocar mensajes en objetos remotos (es decir objetos alojados en espacios de memoria diferentes, posiblemente incluso en máquinas distintas) Sun Microsystems elaboró un estándar denominado **E.J.B. (Enterprise Java Beans)**. Bajo este estándar era posible crear objetos que "publicarían" sus protocolos para que fueran invocados desde cualquier otro lugar. Una de las más importantes características de los EJB era que no eran administrados por el desarrollador, sino que por el contrario se los "instalaba" en un servidor de aplicaciones, el cual se hacía cargo de administrar el ciclo de vida de los *ejbs*, llevándolos a memoria cuando se los requería, pausándolos cuando no se los requería, etc.

Esta administración por parte del servidor, si bien resultó ser una buena idea, su implementación impuso restricciones fuertes tales como la necesidad de escribir métodos en los objetos para que fueran invocados por el servidor a su debido momento. El problema fue que muchos desarrolladores perdieron la tan ansiada transparencia, y terminaron escribiendo métodos "basura" en sus clases, ya que muchas veces no les interesaba escribir estos métodos en su totalidad, pero al utilizar el estándar EJB era lo que se debía hacer.

⁴ Algo así como viejos y comunes objetos Java.

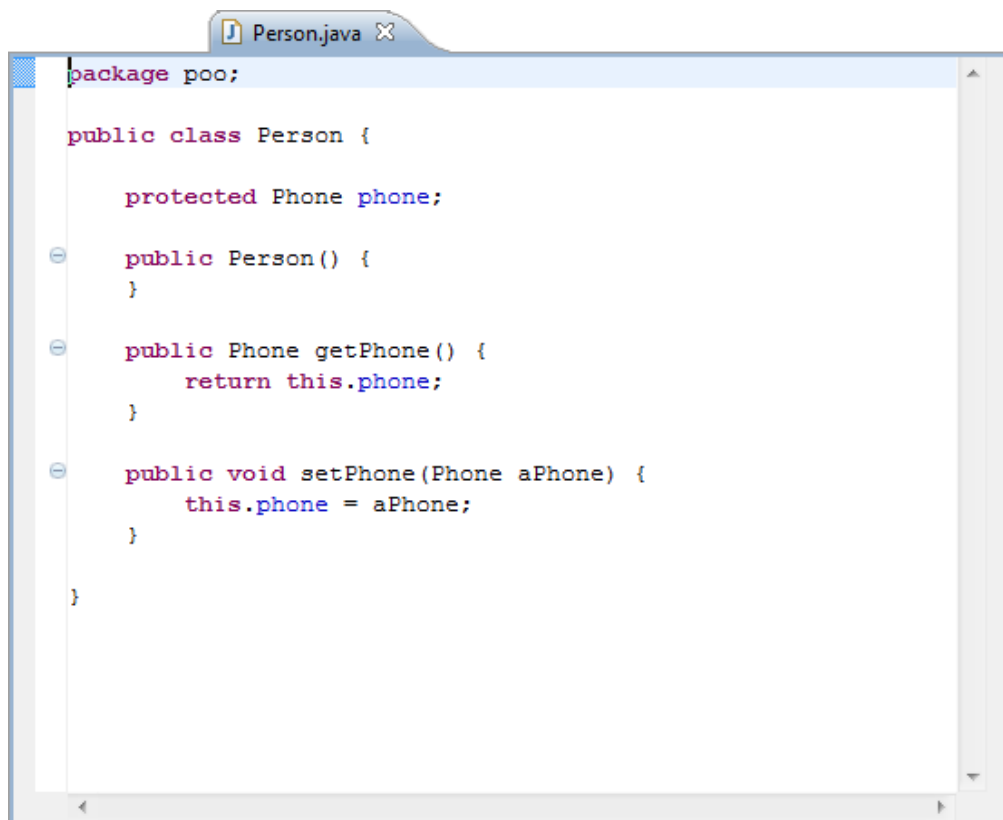
Moraleja, una característica deseable sería no tener que escribir nada más que el código propio de la aplicación, sin distraerse por otros detalles como concurrencia, distribución, persistencia, etc.

Persistencia por alcance

En la introducción hablábamos de que el concepto fundamental y guía de este libro es la **persistencia por alcance** (también se la suele mencionar como persistencia transitiva). Dicho concepto establece que todo **objeto al cual se pueda llegar a partir de un objeto ya persistente, debe ser necesariamente persistente**. Si bien es bastante simple lo que define, en realidad lo que nos interesa es como afectan las implicancias de este concepto al diseño y desarrollo de software, ya que otra forma de interpretar la persistencia por alcance es que a partir de ahora para persistir objetos es necesario solamente vincularlos con algún objeto ya persistente.

A continuación se muestra el código de la clase `Person`⁵, la cual tiene un colaborador perteneciente a la clase `Phone`:

⁵ Si bien soy un ferviente defensor de la utilización de nombres en los idiomas nativos de cada desarrollador, es cierto que al tener todas las palabras claves en inglés, el nombre de la clase también se ajusta mejor al inglés.



```
package poo;

public class Person {

    protected Phone phone;

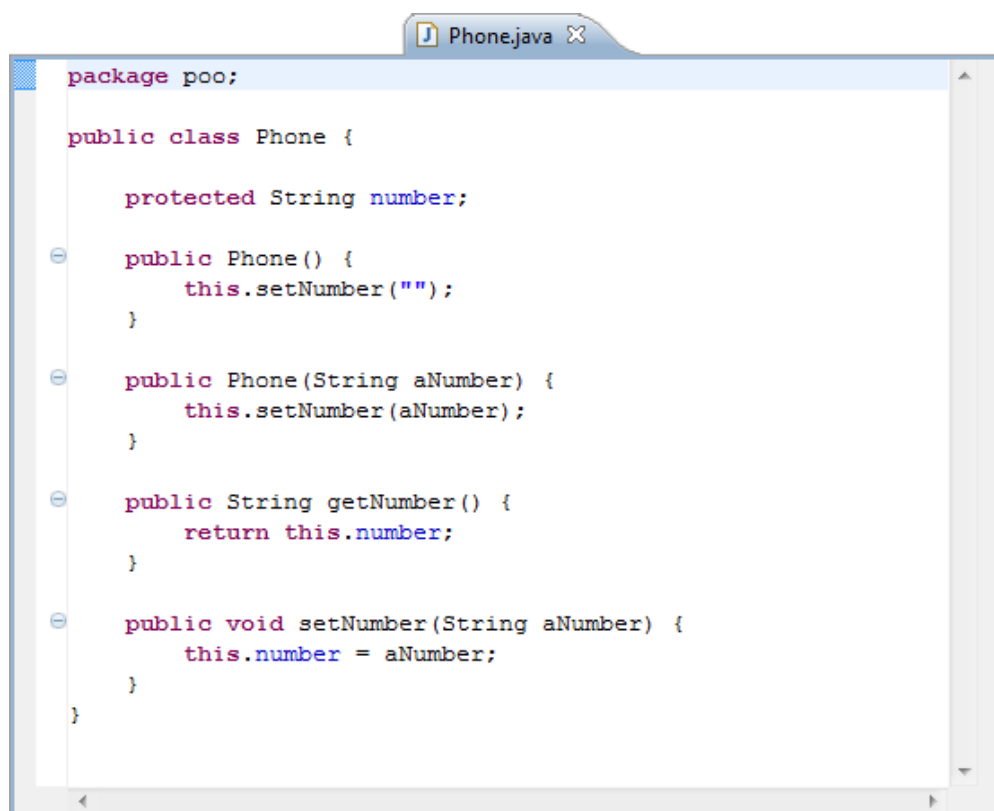
    public Person() {
    }

    public Phone getPhone() {
        return this.phone;
    }

    public void setPhone(Phone aPhone) {
        this.phone = aPhone;
    }

}
```

Figura 1: código fuente de la clase Person



```
package poo;

public class Phone {

    protected String number;

    public Phone() {
        this.setNumber("");
    }

    public Phone(String aNumber) {
        this.setNumber(aNumber);
    }

    public String getNumber() {
        return this.number;
    }

    public void setNumber(String aNumber) {
        this.number = aNumber;
    }

}
```

Figura 2: código fuente de la clase Phone

Teniendo en cuenta el código de ambas clases, y por increíble que parezca, para persistir un nuevo teléfono basta con asignárselo a la persona apropiada mediante las siguientes líneas de código (solamente la línea 12).

```
8      Person aPerson = null;
9      // se recupera a la persona apropiada de la base de datos;
10
11      Phone aPhone = new Phone("123 456 789");
12      aPerson.setPhone(aPhone);
```

Figura 3: Código necesario para persistir un teléfono

Realmente increíble ¿no?

Las anteriores líneas de código merecen una explicación más detallada para su mejor comprensión:

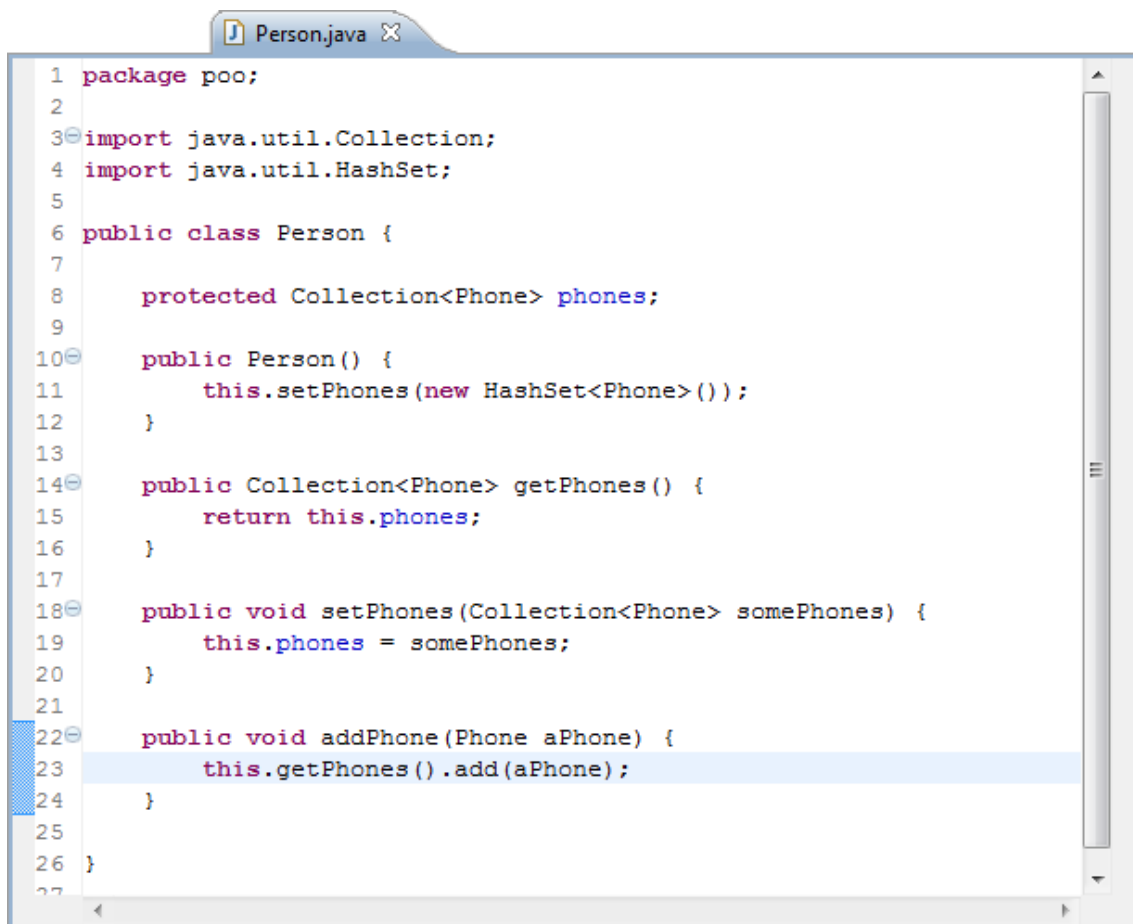
1. Las líneas 8 y 9 de la figura 3 en las que se recupera una persona por el momento no son más que un pseudo-código ya que aquí es justamente donde se debe interactuar con el repositorio de objetos a fin de recuperar una instancia ya persistente de la clase `Person`. Una vez que la instancia se recupera, todo objeto que sea vinculado a ella deberá ser a su vez persistente una vez que se finalice exitosamente la ejecución del método que contiene estas líneas.
2. La línea 11 muestra que no es necesario escribir ningún código especial para que las nuevas instancias de teléfono se almacenen. Los objetos persistentes se crean a través de sus constructores habituales y cuando se los vincule a otro objeto ya existente se persistirán.
3. La línea 12 es la que realmente hace "la magia", pero sin embargo en realidad lo único que ejecuta es el setter de la variable de instancia *phone* (revisar el código mostrado previamente para asegurarse que así sea).

Si revisamos el código de ambas clases se podrá observar que en ninguno de los dos casos tenemos código de base de datos ni tuvimos que alterar nuestro diseño natural de la relación entre las personas y sus teléfonos debido al conocimiento que tenemos de

que finalmente ambas instancias serán persistentes. Este es entonces un buen ejemplo del concepto de transparencia para el desarrollador al que nos referíamos al principio de este capítulo.

Por supuesto que este no es todo el código necesario en una aplicación real, pero sirve como una muestra de lo que implica la **persistencia por alcance** desde el punto de vista del diseño y su posterior implementación.

¿Qué pasaría si ahora nos interesara tener una persona que a su vez tiene un conjunto de teléfonos y no uno sólo como en el ejemplo anterior? En este caso, la clase Person sería la siguiente



```
1 package poo;
2
3 import java.util.Collection;
4 import java.util.HashSet;
5
6 public class Person {
7
8     protected Collection<Phone> phones;
9
10    public Person() {
11        this.setPhones(new HashSet<Phone>());
12    }
13
14    public Collection<Phone> getPhones() {
15        return this.phones;
16    }
17
18    public void setPhones(Collection<Phone> somePhones) {
19        this.phones = somePhones;
20    }
21
22    public void addPhone(Phone aPhone) {
23        this.getPhones().add(aPhone);
24    }
25
26 }
27
```

Figura 4: Código de la clase Person con una colección de teléfonos

Como se puede observar, nuevamente no existe código que de alguna manera esté relacionado con una base de datos. El código de la clase Person continúa siendo puramente "de dominio".

Para persistir una nueva instancia de teléfono bastaría con ejecutar el método **addPhone()** de la clase Person con la instancia de teléfono y listo!

Aquí en este punto es donde es posible vislumbrar los cambios en el diseño, implementación e incluso a nivel de arquitectura de software que tiene la **persistencia por alcance**.

Operaciones C.R.U.D.

En la bibliografía disponible actualmente suele identificarse un conjunto de operaciones necesarias e indispensables para toda aplicación orientada a objetos que deba persistir su información. Dicho conjunto es conocido como operaciones C.R.U.D. (por sus siglas en inglés):

1. **C** (Create): esta operación es la encargada de crear nueva información persistente. A partir del conocimiento de la **persistencia por alcance** esta operación no debería existir como una operación en sí misma, sino que está representada por el mismo código de dominio que crea y asigna las instancias a los objetos que correspondan. En otras palabras, no debería existir una operación "**create**" o "**save**" para almacenar instancias en el repositorio.
2. **R** (Retrieve, Read): esta operación es la encargada de recuperar la información del modelo. Probablemente si se tuviera que optimizar la performance de la aplicación, esta operación sería el principal punto para hacerlo. El modelo es perfectamente capaz de resolverlo, pero como mencionamos más arriba, esta es una buena oportunidad de optimizar la performance.

3. **U** (Update): en una aplicación tradicional, para actualizar un objeto basta con enviar un mensaje (setter) con la información nueva a actualizar. En un modelo persistente, no debería ser diferente si se trata de obtener transparencia. En otras palabras, no es necesario tener operaciones "**update**" diferenciadas para actualizar la información.
4. **D** (Delete): para borrar un objeto del modelo lo único que debería hacerse es "desvincularlo" apropiadamente de sus colaboradores. Nuevamente, no es necesario tener un "**delete**" específico en el modelo.

Más adelante, cuando tratemos el tema del patrón DAO volveremos sobre las operaciones C.R.U.D. para demostrar que dicho patrón de diseño trae más problemas y falencias en el diseño que beneficios.

Capítulo II

Resumen: En este capítulo se repasan conceptos esenciales de ambientes concurrentes y otros conceptos fundamentales como transacciones, esquemas de bloqueo y versionamiento. Dichos conceptos serán siempre abordados desde la problemática que generan en el diseño y desarrollo orientado a objetos que requiere de persistencia en bases de datos.

Propiedades A.C.I.D.

Estas propiedades establecen un marco de trabajo y restricciones que deben ser cumplidas en un ambiente concurrente y multiusuario a fin de mantener la información accesible, confiable, siempre en forma compartida entre todos los usuarios. Cada una de las propiedades está enfocada en un aspecto en particular:

1. A (de Atomicity en inglés o Atomicidad): al interactuar con un repositorio de información (por ejemplo una base de datos relacional) usualmente se invocan series de operaciones más que una sola operación en particular. En este escenario, un requerimiento importante a cumplir es que todo el conjunto de operaciones enviadas a la base de datos se deben ejecutar exitosamente como una sola unidad. En caso de que alguna de estas operaciones falle, todos los cambios producidos por las demás operaciones se deben deshacer a fin de mantener la consistencia de la información. La atomicidad justamente es la propiedad que resguarda este punto al establecer que se ejecuta todo como una unidad indivisible o no se ejecuta nada.
2. C (de Consistency en inglés o Consistencia): esta propiedad es la encargada de mantener la coherencia de la información, asegurando que la información nunca pase a un estado incoherente. Todo conjunto de operaciones siempre debe pasar

la información de un estado coherente a otro, a fin de mantener la confianza en el repositorio.

3. I (de Isolation en inglés o Aislamiento): en un ambiente concurrente y multiusuario, es de esperar que cada usuario envíe una serie de operaciones a la base de datos. En este contexto, no sería adecuado que cada operación de cada usuario se ejecute sobre el ambiente compartido en forma libre ya que podría estar afectando el conjunto de datos que otra operación de otro usuario podría estar leyendo o incluso alterando en forma simultánea. Para evitar este problema, cada conjunto de operaciones se debe ejecutar en forma aislada respecto de los otros conjuntos de operaciones. Una vez que todo el conjunto de operaciones se haya ejecutado en forma completa y satisfactoria, recién en ese momento los cambios serán visibles para los demás usuarios.
4. D (de Durability en inglés o Durabilidad): finalmente esta propiedad establece que un cambio realizado satisfactoriamente a la información mantenida por un repositorio, y que haya cumplido satisfactoriamente las 3 propiedades anteriores, deberá ser mantenido por dicho repositorio soportando cualquier situación anormal, por ejemplo fallas. A partir de la ejecución satisfactoria de un cambio, el nuevo estado de la información se debe mantener como base para cualquier otro cambio que se realice en el futuro.

La forma tradicional de mantener las 4 propiedades es a través de la demarcación de las operaciones utilizando transacciones. Utilizando éstas últimas, el desarrollador irá estableciendo el conjunto de operaciones que deben ser ejecutadas como una sola unidad (atomicidad), las cuales deberán dejar la información consistente (consistencia), siempre en un ambiente aislado (aislamiento) y en forma durable (durabilidad).

En la siguiente sección se discutirán más en detalle las transacciones, presentando las características especiales que hay que tener presente al momento de trabajar con éstas en un ambiente orientado a objetos.

Transacciones

Como mencioné en el punto anterior, las transacciones suelen ser el medio más adecuado para mantener y resguardar la información a través del aseguramiento de las 4 propiedades A.C.I.D.

También se mencionó que el desarrollador es el encargado de demarcar el conjunto de operaciones que deben ser ejecutadas como una unidad. Si bien esto es un tema ya conocido, el problema al que nos enfrentamos es que este conjunto de operaciones no es precisamente un conjunto de sentencias SQL, sino que si estamos en un ambiente orientado a objetos, por lo que estas “sentencias” serán en realidad mensajes enviados a distintos objetos.

Para tratar de aclarar el punto anterior, tenemos que analizar varios puntos relacionados con las transacciones:

1. Si consideramos que estamos trabajando en un ambiente orientado a objetos puro, es de esperar que todo sea un objeto, en particular las transacciones serán objetos. Esto significa que se le podrán enviar mensajes a las transacciones, por ejemplo:
 - a. `begin()` que demarcará el inicio de la transacción.
 - b. `commit()` que notificará a la transacción que trate de bajar a la base los cambios realizados mediante el envío de mensajes a los objetos.
 - c. `rollback()` que indica a la transacción que deshaga los cambios realizados a los objetos.
2. Mencionamos más arriba que es el desarrollador el que debe demarcar en el código el inicio y fin de las transacciones, por lo cual estamos poniendo en peligro el concepto de transparencia de persistencia establecido en el capítulo I. La figura 5 muestra un ejemplo de este punto.

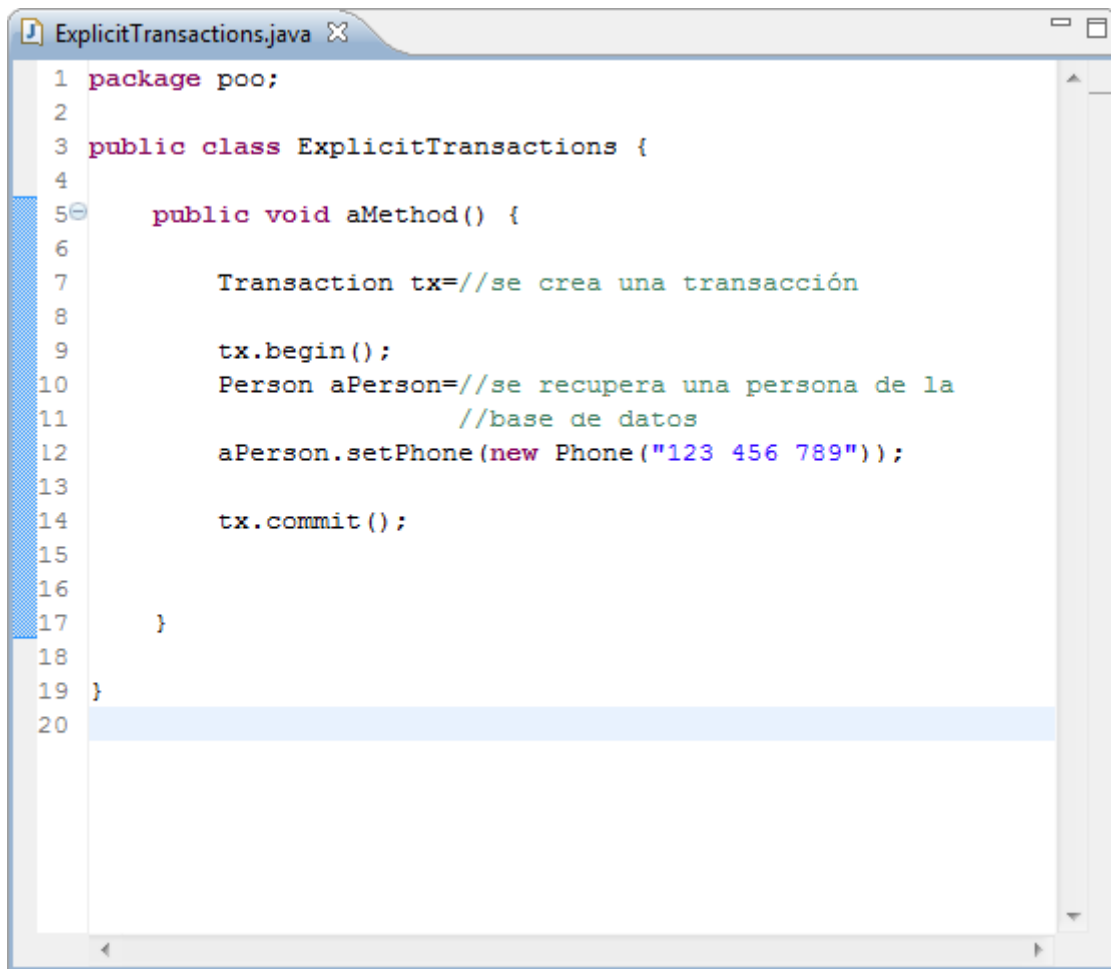


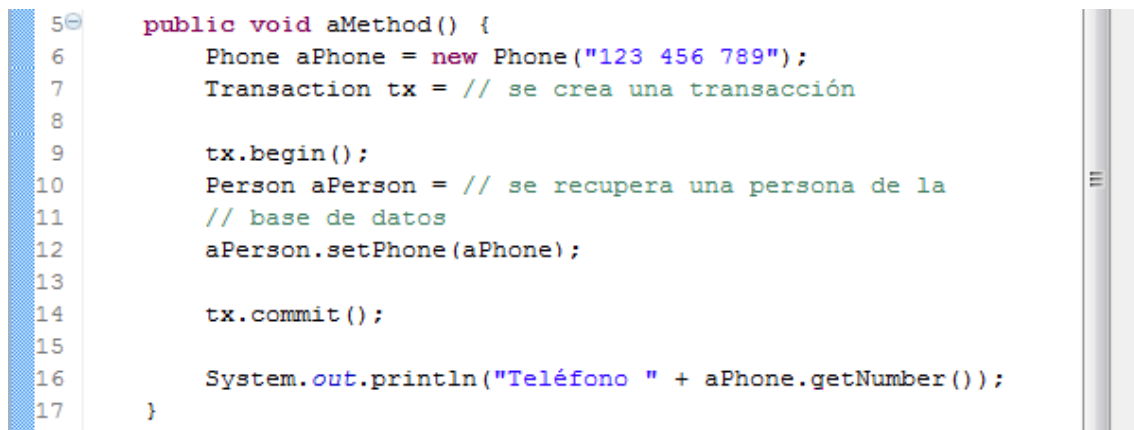
Figura 5: Demarcación explícita de transacciones

3. La figura anterior no sólo muestra la demarcación de transacciones explícita (líneas 7, 9 y 14), sino que también presenta un ejemplo de cómo se envían mensajes a los objetos involucrados en la resolución de cierta lógica de negocios. El problema aquí es que la solución de persistencia que se esté utilizando debe llevar el registro de los cambios que se van realizando a cada uno de los objetos involucrados, tanto a fin poder bajar los cambios como para desestimarlos y volver al estado anterior a cada uno de los objetos afectados.
4. Si se recuerda el concepto de **persistencia por alcance**, al realizar el commit de la transacción la solución de persistencia aplicará dicho concepto e irá recorriendo todos los objetos modificados dentro de la transacción para analizar si corresponde actualizarlo (en caso de que ya sea un objeto persistente) o si por el contrario se trata de un objeto volátil (y por ende se lo debe persistir).

5. Un punto que es necesario aclarar es el relacionado con el “momento” en el cual se crean las nuevas instancias que luego se persistirán. En el ejemplo de la figura 5, en la línea 12 se crea una nueva instancia de clase Phone, pero no hay ninguna restricción respecto de que esto siempre tenga que ser así. Se podría haber pasado por parámetro al método del ejemplo un objeto teléfono para no tener que crearlo.

Lectura de la información

Por lo dicho anteriormente, salvo que se cuente con herramientas adicionales⁶, el código de nuestro diseño tendrá llamadas explícitas a las transacciones. Un punto importante que hay que tener en cuenta es cómo las transacciones ayudan a mantener las propiedades A.C.I.D. y cómo terminan afectando la manera de pensar a la que estamos habituados. Para ello podemos analizar en detalle el código de la siguiente figura.



```
5 public void aMethod() {
6     Phone aPhone = new Phone("123 456 789");
7     Transaction tx = // se crea una transacción
8
9     tx.begin();
10    Person aPerson = // se recupera una persona de la
11    // base de datos
12    aPerson.setPhone(aPhone);
13
14    tx.commit();
15
16    System.out.println("Teléfono " + aPhone.getNumber());
17 }
```

Figura 6: Alcance de las variables afectadas por las transacciones

⁶ Existen herramientas y frameworks que solucionan en parte este problema, y de hecho este libro contiene un capítulo sobre este punto, pero por el momento se asume que no se cuenta con dichas herramientas por lo que agregar transacciones inevitablemente “ensucia” el código.

Analizando en detalle la figura 6, puede verse que el código es bastante similar al de la figura 5, excepto que la instancia de teléfono es creada en forma separada (línea 6) y luego asignada a la persona (línea 12). La demarcación de las transacciones establece que la asignación del teléfono a la persona debe hacerse protegiendo las propiedades A.C.I.D. por lo que al finalizar la transacción (línea 14) se persistirá la nueva instancia de teléfono por persistencia por alcance.

Hasta aquí no hay mayores novedades. Los problemas empiezan cuando se trata de ejecutar la línea 16. En dicha línea básicamente lo que se espera obtener es una impresión en la consola del número de teléfono de la instancia persistida. Si bien no es “nada de otro mundo”, el código de la figura 5 lamentablemente falla⁷. ¿Pero por qué falla si el alcance de la variable que referencia al teléfono llega hasta la línea 17 inclusive?

La razón de esta falla es que con el objetivo de mantener las propiedades A.C.I.D., no está permitido leer información relacionada con la base de datos fuera de las transacciones (recordar que al persistir la información, se está compartiendo la información en un ambiente concurrente multi-usuario). Luego de la línea 14 otro usuario podría estar modificando la información y no se estaría accediendo a dicha información actualizada si no se accediera dentro de un marco transaccional.

Semejante restricción implica cambios profundos en la forma de diseñar y codificar, ya que la intención de persistir información en una base de datos es poder consumirla más tarde, pero ¿cómo se puede acceder a dicha información si cada vez que cerremos las transacciones perderemos la posibilidad de acceder a la misma?

Por suerte existen varias alternativas para salvar este problema:

1. Abrir la transacción el mayor tiempo posible, de modo que toda la recuperación de la información siempre esté dentro del marco de las transacciones. Es obvio que esta alternativa no es válida en una aplicación real, pues implica que toda interacción con el usuario a través de una interfaz gráfica deberá ser dentro de

⁷ Por cuestiones de optimización de rendimiento, en realidad el código tal cual está escrito no falla. Al acceder a atributos básicos usualmente no se dan errores ya que se cuenta con la información “en memoria”. En cambio si lo que se está accediendo es un colaborador (instancia de otra clase) el código en cuestión sí dará una excepción.

transacciones y por ende las mismas deberían tener una duración extremadamente grande (estamos hablando de segundos o incluso minutos para que el usuario tome decisiones que permitan cerrar las transacciones). Durante todo el tiempo que el usuario tiene abierta una transacción se podrían estar generando cambios en la base de datos que dicho usuario no estaría viendo y que recién al tratar de hacer **commit** con su transacción se harían visibles (posiblemente generando excepciones debido a la “vejez” o invalidez de sus datos). Esta alternativa se menciona simplemente por una cuestión de completitud.

2. Utilizar los objetos desasociados⁸. Este nombre se genera a raíz de que esta alternativa plantea la posibilidad de “desasociar” o “desligar” los objetos recuperados de las transacciones en las cuales se los ha obtenido. En otras palabras, según esta alternativa es posible ejecutar satisfactoriamente el código de la figura 6 sin que se produzca un error en la línea 16. Si bien es una alternativa válida, por cuestiones más avanzadas para este capítulo introductorio (como optimización de performance), su aplicación no suele estar exenta de problemas. En capítulos posteriores nos referiremos a esta alternativa con más detalle.
3. Usar DTOs (Data Transfer Objects): como su nombre lo indica, esta alternativa se basa en la utilización de objetos que sirven solamente para transferir datos o información. Básicamente esta alternativa plantea una solución muy simple al problema de la lectura de la información fuera de las transacciones. Dicha solución establece la carga de la información relevante de los objetos recuperados dentro de las transacciones en objetos volátiles que puedan ser leídos fuera del contexto transaccional. La diferencia radical con la alternativa de los “detached objects” es que no se está desasociando a los objetos recuperados de sus respectivas transacciones, sino que se están creando “copias” de los mismos. Al ser copias no estarán vinculados con transacción alguna, por lo que se podrá acceder a su información aún fuera de las transacciones. Las siguientes figuras muestran más claramente este punto y permiten dar un primer

⁸ El término en inglés es ‘Detached Objects’.

vistazo al patrón de diseño DTO (el capítulo VI contiene una explicación detallada de este patrón, sus beneficios y contras).

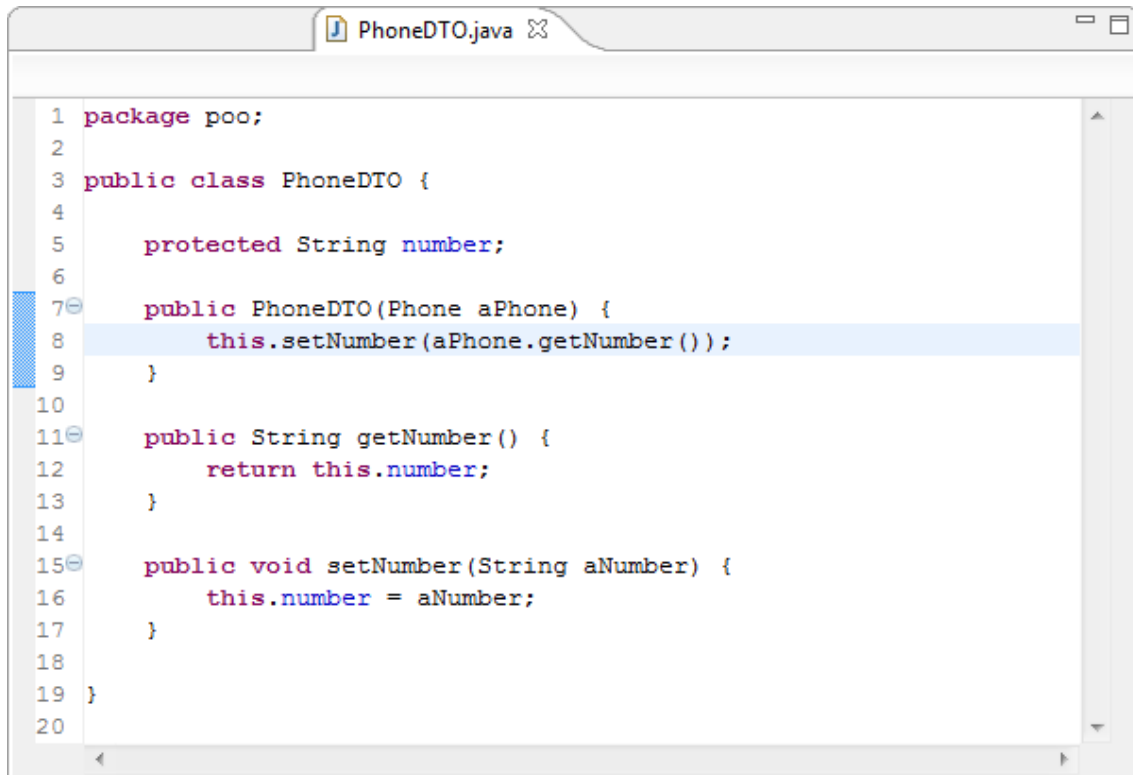


Figura 7: un DTO para una instancia de la clase Teléfono.

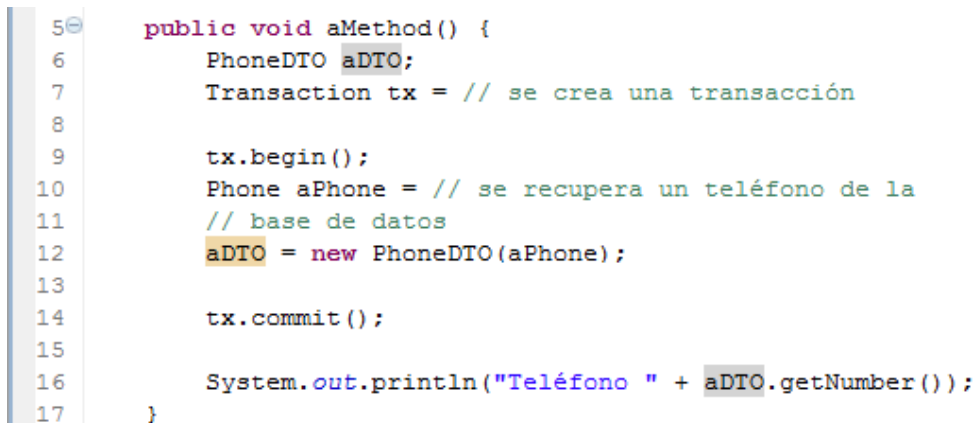


Figura 8: código de ejemplo utilizando los DTOs.

Nuevamente vale realizar algunas aclaraciones sobre las dos figuras anteriores:

1. El código de la clase PhoneDTO (figura 7) es muy similar al código de la clase Phone (figura 2). Sin embargo, la intención de clase PhoneDTO es simplemente transferir información, a través de sus atributos y sus correspondientes getters y setters. En la figura 2 no se muestra, pero es de esperar que la clase Phone contenga lógica de dominio/negocio que amerite la existencia de dicha clase. Otro punto a notar es que el modelo (Phone) no conoce al DTO (PhoneDTO) sino que el conocimiento es a la inversa. Más adelante volveremos sobre este punto a fin de comentar los beneficios de este enfoque.
2. El DTO no contiene absolutamente ninguna lógica, está compuesto únicamente por getters y setters.
3. Los DTOs refuerzan la separación en capas, ya que no presentan más comportamiento que sus getters y setters, por lo que no es posible invocar lógica de negocios sobre ellos. A diferencia de los DTOs, una desventaja importante de un Detached Object es que contiene toda la lógica de negocio, con lo cual al enviarlo por ejemplo al cliente, éste podría equivocarse e invocar cierta lógica en forma indebida.

Finalmente, considerando nuevamente la figura 6 ¿porqué podría fallar el código? La respuesta a esta pregunta suele desconcertar a los desarrolladores que no están habituados a trabajar en ambientes multi-usuario y concurrentes. Además de todos los problemas tradicionales, debemos agregar un nuevo punto que podría generar problemas. Dicho punto se encuentra en la línea 14 (la línea del **commit**). En esta línea es donde la solución de persistencia que se esté utilizando deberá controlar que la información pueda enviarse a la base de datos manteniendo las propiedades A.C.I.D. y es justamente por dicha razón que si la solución de persistencia nota que el usuario está tratando de agregar un nuevo teléfono a una persona que ha sido borrada de la base por otra transacción se levantará una excepción para señalar esta situación irregular.

Como conclusión, cuando se está realizando un debugging de la aplicación, pasar exitosamente las líneas correspondientes a la lógica de negocios no significa que ese código es seguro, ya que finalmente se evaluarán todos los cambios realizados a los

objetos cuando se realice el **commit** y podrían entonces en dicha línea levantarse excepciones no esperadas.

Esquema de lockeo

En la sección anterior se mencionó que al tratar de realizar un commit podría darse un error debido a una edición de los objetos por parte de otro usuario. En realidad, esto podría ocurrir siempre y cuando se permita que dos usuarios puedan acceder y modificar los mismos objetos. De hecho, esta es una de las dos alternativas existentes:

1. Optimista: esta estrategia supone que no existirán conflictos de ediciones concurrentes de la misma información, por diferentes usuarios. A partir de esta suposición se permite el acceso a los usuarios a todos los objetos del repositorio.

La clara ventaja de esta alternativa es que ningún usuario queda bloqueado esperando el acceso a otro objeto porque algún otro usuario ya lo tiene “tomado”. Todo usuario puede acceder a prácticamente todo objeto.

Por el lado de las desventajas tenemos que mencionar los problemas que se dan cuando se detecta un conflicto de edición concurrente. En este caso, probablemente una de las dos ediciones concurrentes esté destinada a fallar.

2. Pesimista: esta alternativa supone que seguramente existirán conflictos entre los diferentes usuarios, por lo que se van bloqueando los objetos a medida que son tomados por los usuarios. De esta manera en todo momento solamente un usuario podrá modificar un objeto dado.

Las desventajas de esta alternativa son dos: por un lado podrían darse problemas de “inanición” para un usuario que está esperando un objeto que nunca es liberado por otro usuario; la segunda desventaja tiene que ver con la degradación muy importante de la performance, ya que todos los accesos a los objetos se deben serializar.

La siguiente sección presenta el concepto de versionamiento, el cual suele utilizarse como mecanismo de control para los ambientes con esquema de lockeo optimista.

Versionamiento

Este concepto suele generar confusión debido a los diferentes significados que se le asignan dependiendo del autor. Aquí presentaremos los dos significados más relevantes, en particular enfocándonos sobre el versionamiento de instancias que permite el funcionamiento de los esquemas optimistas.

Versionamiento de clases

El versionamiento de clases define el conjunto de cambios al esquema de clases del diseño que se soportará cuando se implemente una solución de persistencia. En otras palabras, define el conjunto de cambios que se podrán realizar a las clases una vez que tengan instancias persistidas, así como qué acciones deben ser tomadas para mantener la consistencia de la información. A grandes rasgos, los cambios posibles se dividen en dos grandes grupos:

1. Cambios suaves

Estos son cambios que pueden demorar su aplicación y/o hacerse bajo demanda. Ejemplo típico de este cambio es agregar una nueva variable de instancia a una clase con instancias ya persistidas. ¿qué acciones deberían realizar inmediatamente en las instancias persistentes luego de agregar la nueva variable de instancia en la clase? En realidad ninguna, ya que podría esperarse a ir recuperando cada instancia bajo demanda y asignándole en ese momento un valor por defecto (recordar que en Java cada tipo tiene su valor por defecto).

2. Cambios duros

A diferencia de los cambios suaves, los duros establecen que se debe detener el acceso de los usuarios a la base de datos y realizar inmediatamente el cambio a las instancias. Un ejemplo de este caso es dividir una clase en una jerarquía con clases abstractas y concretas. En este caso no queda otra alternativa que recorrer el conjunto de instancias y proceder a asignarle su nueva “clase”.

Versionamiento de instancias⁹

El versionamiento de instancias por su parte tiene como objetivo asegurar que las ediciones concurrentes sean llevadas a cabo de manera que no afecten la información llevándola a estados inconsistentes.

En este punto nuevamente tenemos dos alternativas: la primera más compleja pero a su vez más abarcativa, mientras que la segunda es muy simple de implementar pero un poco “restringida” en su alcance.

1. Versionamiento integral

El esquema de versionamiento “integral” permite contar con objetos y versiones de éstos. Cuando se crea un nuevo objeto, se considera que éste es el original. Luego, el usuario tiene herramientas para pedir la creación de “copias” o versiones del objeto original. Cada objeto puede contar con un grafo dirigido de versiones. En este esquema hay que resaltar que los demás objetos pueden elegir apuntar a cualquier versión de otro objeto (tanto el original como cualquier versión derivada). Como puede observarse, este esquema es complejo de implementar, pero a la vez muy potente.

2. Versionamiento simple

El esquema es realmente básico. Incluso tampoco es algo novedoso, ya que se lo conoce desde hace mucho tiempo. Lo que se plantea es agregar una nueva variable de instancia a las clases que serán persistentes para almacenar un número que indica a qué versión corresponde una instancia determinada. Cada vez que se modifique exitosamente (es

⁹ El esquema que se menciona aquí es sumamente simple. Otras alternativas más eficientes y complejas pueden ser implementadas. Este esquema simple se comenta con el fin de demostrar cuán fácil puede ser llevar el control de la versión de un objeto en particular. Un punto no cubierto por esta alternativa es el versionamiento de todo un grafo de objetos.

decir se pueda realizar efectivamente el **commit** de la transacción asociada) una instancia, este número se incrementará automáticamente.

Este esquema sigue los siguientes pasos entonces:

- I. Cualquier usuario recupera un objeto de la base de datos, con un número de versión dado. Dicho número de versión es mantenido automáticamente por la solución de persistencia que se esté utilizando.
- II. Cada vez que un usuario edita la información de sus objetos y puede realizar exitosamente un **commit** de la transacción, dicho número de versión se incrementa automáticamente.
- III. El resto de los usuarios que también habían leído la misma instancia con el mismo número de versión, cuando quieran persistir sus cambios lo harán con un número de versión “viejo” por lo que la transacción fallará. En este caso el código debe estar preparado para este “incidente” ya que no es la lógica de negocios la que falló, sino que ocurrió un problema de concurrencia.

Como puede verse en la descripción de pasos anterior, no es más que un esquema típico de “timestamp”, pero aplicado a objetos en esta ocasión y con un control automático por parte de la solución de persistencia utilizada.

Capítulo III

Resumen: En este capítulo se presentan los principales elementos de las bases de datos orientadas a objetos, sus características y beneficios. La idea que se persigue al presentar este tipo de bases de datos es permitir a los usuarios conocer un ambiente de persistencia de objetos muy “puro”, a partir del cual se diseñan soluciones orientadas a objetos de calidad, robustas y escalables. Todo el conocimiento adquirido con este tipo de soluciones de persistencia luego permitirá generar mejores soluciones incluso cuando se utilizan soluciones de mapeo objeto-relacional u otras.

¿Qué es una base de datos orientada a objetos?

Una base de datos orientada a objetos puede ser definida como una colección de objetos, a la cual se le agregan características de persistencia. En otras palabras, es una colección a la que se debe acceder protegiendo en todo momento las famosas propiedades A.C.I.D.; es una colección a la que se debería poder consultar en base a un lenguaje de consultas eficiente y posiblemente declarativo; es una colección de objetos cuyo ciclo de vida se extiende más allá del tiempo de vida de la aplicación que la creó.

Aquí me permito agregar una característica más a las bases de datos orientadas a objetos. Además de todas las características previas citadas, la colección de objetos debe ser tratada en sí misma como un objeto. Aún cuando esto pueda parecer trivial al principio, la mayoría de los autores elige no acceder en forma “orientada a objetos” a las bases de datos, por lo que más adelante volveremos sobre este punto con el fin de promover una mejor utilización de las bases de datos.

Dudas más frecuentes

Cuando se presenta una nueva tecnología es habitual la aparición de muchas dudas e interrogantes sobre la factibilidad y aplicabilidad de dicha tecnología. A continuación se presentan brevemente algunas cuestiones que usualmente se plantean a la vez que se van respondiendo con ejemplos y datos concretos.

¿Cuáles son los beneficios de las BDOOs?

Esta pregunta probablemente sea una de las que más subjetividad contenga, sobre todo considerando mi inclinación hacia todo lo referido a la orientación a objetos. Sin embargo, vale la pena repasar algunas de las cuestiones que a mi juicio pueden plantearse como posibles beneficios de las bases de datos orientadas a objetos.

- Diseño más natural: si uno cuenta con una solución de persistencia que brinda transparencia, óptimo rendimiento y facilidad de uso seguramente podrá enfocarse más tiempo en solucionar los problemas relacionados con el negocio o dominio. Esto trae como consecuencia que la calidad del diseño obtenido mejore, o que por lo menos no se vea afectado por cuestiones ortogonales como la persistencia.
- En el desarrollo, menos código: supongamos un escenario de trabajo muy común: bases de datos relacionales y aplicación orientada a objetos. En este escenario existen dos alternativas para persistir la información: se desarrolla una solución propietaria de mapeo o se utiliza un mapeador objeto-relacional. En ambos casos, en mayor o menor medida, se deberá escribir código (Java, .Net, XML, etc) para poder realizar la traducción objeto-relacional y viceversa. Si en cambio se utilizara una solución puramente orientada a objetos, dicha capa no sería necesaria, por lo que al final de cuentas uno tiene menos código que desarrollar (y mantener!).

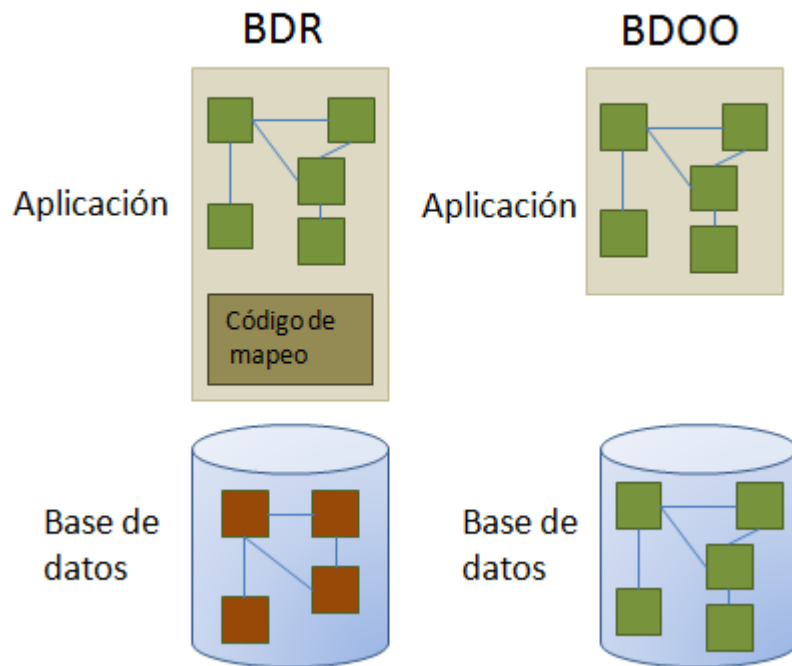


Figura 9: Código extra en el caso del mapeo objeto relacional¹⁰

- En la utilización, mucha performance: usualmente se considera casi a nivel de mito urbano que las aplicaciones orientadas a objetos son “lentas” o con mala performance. Algo de cierto hay sin embargo, sobre todo por la falencia de los diseñadores/desarrolladores al momento de preocuparse por el rendimiento (siempre se deja para más adelante, lo cual finalmente se transforma en un “nunca jamás”). Parte del mito también tiene su base en que se considera que las bases de datos orientadas a objetos utilizan estructuras de datos distintas y menos eficientes que las de bases de datos relacionales, lo cual es falso. Es decir, tanto las bases de datos relacionales como las orientadas a objetos pueden utilizar las mismas estructuras de datos “internas” para almacenar la información. Por lo tanto, bien diseñada e implementada, una base de datos orientada a objetos no tiene porqué ser más lenta que una base de datos relacional (incluso se podría argumentar que hasta hay razones para que sea más rápida, por ejemplo al ejecutar menos código seguramente será más rápida la resolución).

¹⁰ Este gráfico es una adaptación del gráfico presente en el paper Objectivity Technical Overview (10-OTO-1).

Hay un refrán que dice que cuando sólo se tiene un martillo, todo se parece a un clavo. Y esto traducido a este contexto debe interpretarse como que para no todos los casos las bases de datos orientadas a objetos necesariamente tienen que representar la mejor opción. Será responsabilidad de los diseñadores, desarrolladores y arquitectos evaluar la aplicabilidad de esta tecnología.

¿Se utilizan en ambientes productivos realmente?

La mejor manera de responder a esta pregunta es a través de la cita de algunos ejemplos de aplicación de esta tecnología en la industria y gobierno:

1. SLAC (Stanford Linear Accelerator Center) es una instalación científica que estudia la estructura de la materia. Para llevar esto adelante suelen realizar experimentos en los cuales se aceleran partículas básicas para hacerlas chocar luego y estudiar las nuevas partículas que se generan en dicha colisión. Para dar una idea de la magnitud de estos experimentos, la base de datos tiene una tasa de ingesta de 1 Tera por día, llegando a un tamaño aproximado de 900 TB de información comprimida [Base de datos Objectivity/DB].
2. P8A – Multi-Mission Maritime Aircraft es un proyecto de Boeing que utiliza bases de datos orientadas a objetos embebidas como repositorio de la información requerida por sus naves de vigilancia y detección de submarinos [Base de datos DB4O].

¿Soportan grandes volúmenes de información?

En el año 2003, una base de datos orientadas a objetos fue nombrada como la base de datos con más información almacenada. Su tamaño equivalía a la suma de las 200 bases de datos relacionales juntas.

Para dar una idea acabada del tamaño que pueden alcanzar, la última versión de ObjectivityDB puede manejar volúmenes de EXABYTES¹¹. La tasa de “ingesta” (o capacidad de incorporar nueva información) puede acceder a niveles cercanos a 100 Terabytes por día.

Estos volúmenes de información no son habituales pero sí alcanzables, sobre todo por experimentos científicos como los aceleradores de partículas¹².

¿Las BDOOs reemplazarán a las bases de datos relacionales?

La respuesta a esta pregunta es claramente NO. ¿Sorprendido? En realidad no debería existir sorpresa, ya que si bien por todo lo comentado hasta el momento las bases de datos orientadas a objetos tienen un nivel de madurez apropiado para tomar el lugar de las bases de datos relacionales, existen también razones de peso que evitarán el reemplazo total. A continuación hay una pequeña lista de estas razones:

1. Conocimientos de la teoría de objetos. Lamentablemente aún hoy no es un paradigma totalmente adoptado. Mucho menos es la cantidad de recursos formados apropiadamente para incorporarlo en ambientes productivos en forma eficiente.
2. Conocimientos extensos de persistencia orientada a objetos. Si es poca la cantidad de recursos formados apropiadamente en la teoría de objetos, es mucho menor todavía la cantidad de recursos que además de objetos, también conoce del tema de persistencia.
3. Base de sistemas instalados, basados en bases de datos relacionales, que deberían ser modificados para acceder a los objetos. La inmensa mayoría de los sistemas están hoy realizados en base a la tecnología relacional, por lo que deberían ser actualizados o modificados para que puedan conectarse con las bases de datos orientadas a objetos. ¿Tiene sentido invertir recursos humanos, técnicos y económicos en actualizar dichos sistemas? Probablemente no.

¹¹ Para aquellos lectores no acostumbrados, un exabyte equivale a 1.000 petabytes, o 1.000.000 de terabytes, o 1.000.000.000 de gigabytes.

¹² Ver en la sección de Referencias CERN y SLAC.

4. Miedo comercial. Este aspecto afecta directamente a la capacidad de innovación tecnológica de las organizaciones, ya que se prefiere (con razón) comprar software a empresas internacionales de gran porte (Oracle, Microsoft, IBM) con representantes locales antes que soluciones más novedosas pero que comercialmente no son tan conocidas ni respaldadas.
5. Inercia. La inercia es la propiedad de los cuerpos de mantenerse en el estado en que se encuentran. Entonces, si una organización tiene sus necesidades cubiertas con sistemas basados en la tecnología relacional, ¿para qué cambiar por otra tecnología?

¿Pueden coexistir bases de datos orientadas a objetos y bases de datos relacionales?

Hay muchos ejemplos actualmente que demuestran que se pueden utilizar ambas tecnologías en forma conjunta. Un ejemplo simple es el brindado por la bolsa de Chicago, que utilizaba la base de datos orientada a objetos Versant para dar soporte a toda su operatoria diaria (muy compleja por cierto). Una vez por día se pasaba toda la información a una base de datos relacional Oracle para poder realizar análisis estadísticos y hacer explotación de toda la información.

¿Se pueden utilizar como soporte de persistencia de aplicaciones Web?

El modelo tradicional Web está basado en el concepto de navegación. Si bien se utilizan habitualmente muchos listados, es muy habitual que el usuario seleccione un elemento y empiece a navegar el grafo de relaciones que dicho objeto mantiene. Teniendo esto último en mente, es fácil ver que resulta muy apropiado entonces el uso de una tecnología que se basa prácticamente en el mismo concepto. Las bases de datos orientadas a objetos almacenan justamente las relaciones de los objetos con los demás objetos, para favorecer la navegación entre los mismos más que la ejecución de consultas que deben hacer “joins” lentos.

¿Existe mucha oferta?

En esta parte del libro vamos a tratar de ser interactivos, para lo cual proponemos un ejercicio muy simple. Propongo que el lector trate de listar todas las bases de datos relacionales que conoce, o mejor aún, de las cuales ha escuchado que existen. Me arriesgo a decir que no puede listar más de 10 o 15 (lo cual es muy poco teniendo en cuenta que estamos tratando con una tecnología que tiene más de 40 años de existencia).

Bien, más abajo hay una lista (no completa) de algunas bases de datos orientadas a objetos o que soportan la persistencia de objetos:

Versant	Gemstone	Objetivity/DB
Prevayler	VOSS	Orion
ODE	ObjectStore	GBase
VBase	Exodus	Jasmine
Matisse	Ontos	O2
FastObjects	Jade	Caché
Perst	Ozone	Zope
EyeDB	10Gen	SiagoDB

Tabla 1: Oferta de bases de datos orientadas a objetos

Bueno, por lo menos se debe admitir que hay una variedad interesante de productos disponibles (algunos comerciales, otros open-source).

Seguramente no todos puedan cubrir las necesidades o requisitos de ciertos sistemas, pero por lo menos podemos asegurar que tenemos variedad.

Manifiesto

Al hablar de objetos, lamentablemente no todo es tan estándar como se podría esperar, y mucho menos al hablar de repositorios orientados a objetos. Por esta razón, ya en 1995

se escribió un manifiesto que establecía las características que debía cubrir todo repositorio que quisiera considerarse como “orientado a objetos”.

Características mandatorias

Como su nombre lo indica, estas características deben ser obligatoriamente provistas por el motor de bases de datos orientado a objetos. Entre estas características tenemos:

1. Objetos complejos: el sistema debe permitir la creación de objetos “complejos” a partir de la composición de objetos más simples. Operaciones como “copia” o “clonación” también deben ser provistas.
2. Identidad de objeto: el sistema debe permitir la existencia de un objeto más allá del valor que este represente. En otras palabras, un objeto existe porque tiene una identidad única que lo define e identifica.
3. Encapsulamiento: el sistema debe permitir mantener oculta la implementación de los conceptos, permitiendo a los objetos operar en base a sus interfaces y no sus implementaciones.
4. Tipos y clases: los sistemas de bases de datos orientadas a objetos deben soportar tanto el concepto de tipo (como en el lenguaje C++) como el concepto de clase (como en Smalltalk).
5. Jerarquías de clases y tipos: la herencia es una herramienta poderosa que debe ser soportada por la base de datos O.O.
6. Overriding, overloading y late binding: todos estos conceptos son centrales en sistemas orientados a objetos para brindar polimorfismo. En consecuencia, un sistema de bases de datos O.O. debería soportarlos eficientemente.
7. Completitud computacional: el usuario debería poder definir cualquier función computable. SQL por ejemplo no es funcionalmente completo.
8. Extensibilidad: además de los tipos predefinidos del sistema de bases de datos, se debería contar con la posibilidad de definir nuevos tipos. El uso entre los tipos predefinidos y los definidos por el usuario debería ser indistinto.

9. Persistencia: este requerimiento es bastante obvio ¿no?
10. Administración de un repositorio secundario: este requerimiento define que el sistema de bases de datos orientado a objetos debería poder mantener grandes volúmenes de información, utilizando posiblemente herramientas como índices, clustering, etc.
11. Concurrencia: otro requerimiento bastante obvio nuevamente ¿no?
12. Recuperación de fallos: seguimos con los requerimientos conocidos.
13. Posibilidad de consultas Ad-hoc: el sistema debería proveer alguna manera de realizar consultas para poder recuperar la información persistida.

Características opcionales

Dentro de las características opcionales a ser provistas por las bases de datos orientadas a objetos podemos encontrar:

1. Herencia múltiple: queda como característica opcional el soportar jerarquías en las cuales es posible utilizar herencia múltiple.
2. Chequeo de tipos e inferencia de tipos: cuanto más soporte para esta característica exista mejor, pero se deja abierto para ser implementado en el grado que el sistema decida pertinente.
3. Distribución: la base de datos opcionalmente podría estar distribuida en múltiples nodos.
4. Transacciones largas: el sistema opcionalmente puede soportar esquemas de manejo de transacciones en los cuales es posible definir anidamiento o transacciones que implican tareas del usuario.
5. Versionamiento: la información administrada por el sistema de bases de datos podría versionar la información para permitir acceder a distintas “fotos” en distintos momentos de la misma información.

Decisiones libres

Finalmente, dentro de las características que se dejaron abiertas para ser seleccionadas y eventualmente implementadas tenemos:

1. Paradigma de programación: conceptualmente no hay restricciones para que el único modelo de programación sea el orientado a objetos. Otros paradigmas podrían ser soportados.
2. Sistema de representación: este sistema queda definido por el conjunto de tipos atómicos y sus constructores. Este conjunto podría extenderse arbitrariamente.
3. Sistema de tipos: no existen restricciones conceptuales para minimizar el sistema de tipos que se utilice.
4. Uniformidad: ¿cuán orientado a objetos debería ser el sistema? Múltiples niveles podrían ser definidos para no tener que implementar todo como objetos.

Los tres niveles definidos para las características según el manifiesto se mueven desde lo muy pragmático (mandatorias) hasta las más teóricas o conceptuales (características libres).

ODMG

En el año 1991, O.M.G. decide crear un subgrupo enfocado en los temas relacionados con las bases de datos orientadas a objetos. Este grupo, denominado O.D.M.G. (Object Data Management Group) tenía como principal objetivo crear y fomentar estándares para posibilitar el desarrollo de aplicaciones con persistencia orientada a objetos (tanto bases de datos orientadas a objetos como las basadas en mapeo objeto-relacional), que fueran portables entre productos y hasta plataformas.

Para lograr semejante objetivo, tanto el esquema de datos, así como los lenguajes de definición y manipulación de la información y hasta los adaptadores¹³ para cada lenguaje debían soportar esta “portabilidad”. Sutilmente se ha puesto aquí una de las características más controversiales planteadas por este grupo: la posibilidad de que la información fuera persistida desde una plataforma de desarrollo y que fuera luego accedida desde otra. La figura 9 muestra un esquema de este punto.

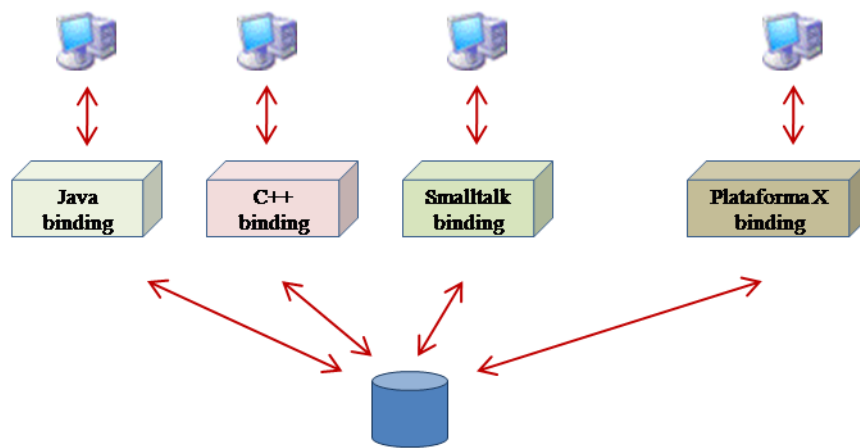


Figura 9: Adaptadores para plataformas.

Según este estándar, mediante la utilización de diferentes “bindings” era posible crear información y persistirla desde la plataforma Java, para luego recuperarla y modificarla desde Smalltalk. Este planteo, si bien bastante más cercano a lo conceptual que a lo práctico, es muy interesante.

Cuenta la historia que hasta el año 2000 se publicaron varias revisiones del estándar, llegando a publicar la versión definitiva bajo el nombre ODMG 3.0. Luego de la publicación de esta versión el grupo se disolvió.

¿Cómo? ¿Se disolvió ODMG? Esto es lo que se preguntaron muchos en aquel momento, y una de las principales razones fue que pocos de los “bindings” tuvieron éxito. De hecho el que más suceso tuvo fue Java. Tanto fue su éxito que gente de

¹³ El término “adaptador” se utiliza como traducción del término “binding” en inglés.

ODMG migró a Sun Microsystems y terminó generando el nuevo estándar de persistencia para Java denominado J.D.O. (Java Data Objects).

OQL

El estándar ODMG contaba con varios componentes: un modelo de objetos, el cual sentaba las bases que debían ser respetadas por todas las implementaciones y las aplicaciones; lenguajes de especificación, que permiten definir por ejemplo los nuevos tipos de objetos; un lenguaje de consulta (OQL), el cual permite consultar y modificar los objetos de manera declarativa; y finalmente “adaptadores” para cada uno de los principales lenguajes orientados a objetos. Claramente el que más impacto ha tenido a lo largo del tiempo es el componente ligado a las consultas. Por esta razón, OQL merece una sección propia en la que se comenten sus principales características.

Principales características

- Similar a SQL’92. Al crear un lenguaje de consultas, la mejor manera de asegurarse que sea fácil de aprender y se utilice rápidamente es que se parezca en lo posible a otro lenguaje ya reconocido. En el caso de OQL, ese otro lenguaje reconocido es el SQL ’92. Teniendo en cuenta la base de clientes instalada y los años que se ha estado utilizando el SQL, ODMG utilizó su estructura y la adaptó para definir su propio lenguaje de consultas orientado a objetos.
- No es computacionalmente completo. En otras palabras, se trata simplemente de un lenguaje de consulta fácil de utilizar. No todas las operaciones pueden ser representadas con este lenguaje.
- Lenguaje funcional completamente ortogonal. En este lenguaje los operadores pueden ser compuestos libremente, siempre y cuando se respeten los tipos definidos en el lenguaje. Esto tiene como consecuencia directa que toda

consulta, al devolver siempre tipos propios del lenguaje, puede ser tomada a su vez como nueva fuente para realizar nuevas consultas.

- Puede ser fácilmente optimizado debido a su naturaleza declarativa. Las optimizaciones necesarias para poder resolver las consultas de la manera más óptima se realizan de manera más directa al tratarse de un lenguaje declarativo, en el cual se expresa solamente lo que se quiere obtener, y no cómo se debe obtener.
- No cuenta con un operador específico para actualizar los objetos (UPDATE). Este punto merece un punto aparte por su importancia ya que al contrario que en SQL, la modificación de los objetos no se realiza mediante una sentencia UPDATE propia del lenguaje. En OQL los updates se realizan invocando mensajes a los objetos.

Estructura

Como se mencionó más arriba, la estructura de una consulta OQL es sumamente similar a la del SQL. Por esta razón, como es de esperar, dicha estructura respeta la forma básica¹⁴

SELECT [distinct] proyecciones

FROM conjunto candidato

[**WHERE** condiciones]

[Cláusula de agrupamiento]

[Cláusula de ordenamiento]

Si bien a primera vista la estructura es idéntica al SQL, hay importantes diferencias:

- En el SQL, el conjunto candidato está compuesto por las tuplas de las tablas intervinientes en la consulta. En una BDOO, al no existir tablas, se debe buscar

¹⁴ Las partes opcionales se encuentran encerradas entre “[“ y “]”.

un concepto que represente al conjunto candidato. Este concepto es la “clase”; en otras palabras, al realizar una consulta con OQL, el conjunto candidato queda representado por la clase a la que pertenecen los objetos que deben ser visitados. La forma entonces de especificar el conjunto candidato es

```
SELECT c  
  
FROM Car c
```

En esta consulta, se está accediendo a todas las instancias de la clase Car, las cuales se referencian a través de la variable “c”.

Un punto importante es el referido a la herencia. En una BDOO se trabaja bajo el paradigma de objetos, por lo que hay que tener muy presente el concepto de la herencia. Suponiendo que se cuenta con un diseño en el cual existe una jerarquía como la siguiente:

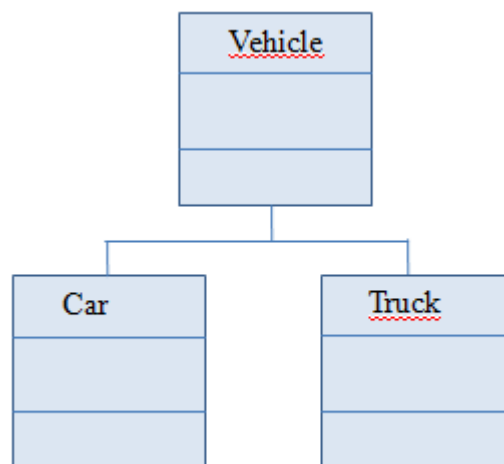


Figura 10: Jerarquía de clases en una consulta OQL.

Por lo comentado en este punto, resulta claro que al consultar por la clase Car, todas las instancias de los autos serán accedidas. ¿Ahora, que pasaría si en vez de Car se pusiera Vehicle? En el paradigma de objetos, todo auto es a su vez también un vehículo, del mismo modo que todo camión también lo es. De este modo, si la consulta utilizara la clase Vehicle como conjunto candidato, lo que se obtendría sería el conjunto de todos los vehículos, tanto autos como camiones.

Esto es ciertamente muy interesante, ya que bajo ciertas circunstancias permite obviar la necesidad de la operación UNION. Otra consecuencia práctica de este punto es que resulta sumamente sencillo conocer la cantidad de objetos almacenados. Sólo basta consultar la cantidad de instancias de la clase Object

SELECT count(o)

FROM Object o

- Trabajando bajo el paradigma de objetos, obtener cierta “parte” de un objeto a través de una proyección no es algo que esté muy bien visto. Sobre todo teniendo en cuenta la cuestión del encapsulamiento. Esto trae como consecuencia que en general luego del SELECT no se vean muchas proyecciones, sino que se devuelvan los objetos “enteros”¹⁵.
- Como se trata de un lenguaje funcional, el resultado de una consulta puede ser utilizado a su vez como conjunto candidato de otra consulta. Esta es una alternativa para escribir consultas complejas, pero fáciles de leer.

Path expressions

En el SQL, el mecanismo para vincular tuplas correspondientes a diferentes tablas es a través del uso de atributos los cuales se “matchean” en operaciones de JOIN. Un punto importante de estas operaciones es que son lentas (aún con la utilización de índices). Es por esto que contar con un mecanismo que permita evitar la utilización de JOINS sería muy bienvenido. Las “*path expressions*” representan un mecanismo por el cual es posible “navegar” las relaciones de los objetos. Con este mecanismo, es posible recuperar una persona, luego navegar a su esposa, luego al auto perteneciente a la esposa para luego acceder a la patente de dicho auto. En OQL esto se escribiría de la siguiente manera:

¹⁵ Por el momento basta con comentar que al devolver un objeto “entero” no es está incurriendo en problemas de performance, ya que luego se realizan optimizaciones para traer a memoria solamente lo requerido. Más adelante cuando se comente el patrón de diseño Proxy se ahondará en este punto.

```
SELECT aPerson.getSpouse().getCar().getPlate()

FROM Person aPerson
```

Las *path expressions*, además de ser mucho más legibles que los JOINS representan una optimización importante, ya que no es necesario iterar sobre colecciones de elementos para realizar el matching por algún atributo, sino que directamente se accede al objeto referenciado por la relación.

Otro ejemplo de *path expression* permite recuperar todos los hijos de todas las personas:

```
SELECT c

FROM Person p, p.children c
```

Invocación de operaciones

En el ejemplo en el cual se accede a la esposa de una persona se muestra una de las características más importantes y a la vez controversiales de OQL: el envío de mensajes a los objetos. Este envío de mensajes permite la ejecución de operaciones propias de los objetos. El resultado de la ejecución de un método pasará a formar parte del resultado de la consulta.

En principio no hay porqué limitar el uso de mensajes a los getters, dando lugar al envío de cualquier tipo de mensajes. Si el mensaje enviado cambia al objeto receptor ya no se requiere tener una operación explícita de UPDATE. Basta con enviar el mensaje al objeto correspondiente y la actualización se habrá realizado.

Si bien este último punto puede parecer interesante, también es cierto que acarrea potenciales problemas. El punto en cuestión es que al tratarse de un lenguaje de consultas, una de las características que debe cumplir como tal es la idempotencia¹⁶.

¹⁶ La idempotencia establece que la ejecución retirada de una consulta sobre el mismo conjunto candidato debe retornar siempre el mismo resultado. Como consecuencia se puede observar que la consulta no debe “modificar” el conjunto de objetos sobre los que se aplica.

Prácticamente en ninguno de los lenguajes en los que se implementó OQL (Smalltalk, C++ y Java) es posible marcar a un método de los objetos como “no mutador” (es decir, que su ejecución no cambiará al objeto receptor). Incluso un getter podría estar mal escrito y modificar algún atributo del objeto sobre el que se ejecuta. Por esta razón, la mayoría de las implementaciones sólo permiten la ejecución de ciertos métodos bien conocidos, no permitiendo la ejecución de consultas que contengan el envío de mensajes correspondientes a clases desarrolladas por el usuario¹⁷.

Polimorfismo

Finalmente, otro punto importante presente en OQL es la posibilidad de hacer uso del polimorfismo. Esto permite, entre otras cosas por ejemplo, aprovechar las jerarquías de clases y realizar consultas a colecciones de objetos heterogéneos sabiendo que en última instancia todos corresponden a la superclase.

Considerando entonces nuevamente la jerarquía de la figura 10, y suponiendo que cada clase de esta jerarquía cuenta con un método polimórfico *getMaxSpeed()*, el cual calcula la velocidad máxima permitida para el tipo de vehículo en cuestión, se podría realizar la siguiente consulta en OQL y dependiendo de la clase de cada objeto en particular se ejecutará la implementación del mensaje que corresponda.

```
SELECT v.getMaxSpeed()  
  
FROM Vehicle v
```

¹⁷ Como ejemplo de estos métodos “bien conocidos” se puede tomar el método *startsWith()* de la clase String, el cual obviamente no es desarrollado por el usuario sino por el proveedor de la máquina virtual de Java, por lo que se podría asegurar que es un método seguro.

Capítulo IV

Resumen: Como se comentó al inicio de este libro, la situación más probable al desarrollar un nuevo sistema de software es que se utilicen dos paradigmas distintos: el orientado a objetos para el diseño y construcción de la aplicación, y el relacional para la persistencia de la información. Por esta razón en este capítulo se presentan los principales conceptos del mapeo objeto/relacional, primero en forma y luego se comenta brevemente el producto Hibernate a modo de ejemplo de implementación de los conceptos previos.

Diferencia de impedancia

El primer punto importante que se debe considerar es la diferencia existente entre los dos paradigmas que se están haciendo coexistir. Por un lado tenemos el paradigma relaciona, basado fuertemente en nociones matemáticas; por otro lado tenemos al paradigma orientado a objetos, el cual tiene sus fundamentos en la Ingeniería de Software. Las diferencias resultantes entre los dos paradigmas fueron bautizadas en los '90 como “diferencia de impedancia”.

Al trabajar con objetos, mapeándolos a una base de datos relacional, es necesario conocer y tener bien en claro dichas diferencias, a fin de poder diseñar y desarrollar aplicaciones que se integren de la mejor manera posible.

Como ejemplo de las diferencias entre ambos paradigmas, suele considerarse el mecanismo de acceso a la información: en objetos, la forma tradicional de acceder a la información es a través de la navegación de los colaboradores, mientras que en relacional la forma recomendada es a través de *joins* entre diferentes tablas. Cierta no es el mismo mecanismo en ambos paradigmas.

Existen otras diferencias más sutiles y a la vez básicas que conviene tener en mente:

- Datos y lógica separados: en el paradigma orientado a objetos la información y la lógica encargada de operar sobre dicha información se encuentran en un solo lugar: los objetos. En cambio en el paradigma relacional, la información se encuentra en tablas, mientras que la lógica se modela en procedimientos, funciones o “triggers”. En el caso del relacional, el modo de trabajo referencia mucho al paradigma procedural.
- Objeto que representa el sistema¹⁸: una práctica común en el diseño orientado a objetos es modelar con un objeto en sí mismo al sistema que se está modelando. En otras palabras, si estoy modelando un sistema para un banco, probablemente la primera clase que aparezca en mi diseño sea Banco. De la misma manera, al modelar un sistema para un aeropuerto, la clase Aeropuerto seguramente aparecerá. El objetivo de dicha clase es representar al sistema bajo diseño, siendo generalmente responsable por ejemplo de las colecciones de objetos importantes, por ejemplo ¿dónde se definiría la lógica de administración de clientes de un banco si no es en la clase Banco? Ahora bien, en el paradigma relacional si uno intenta crear una entidad Banco en un diagrama Entidad-Relación, lo más probable es que termine recursando la materia de bases de datos¹⁹.
- Tipos de datos: otro punto con notables diferencias es el relacionado con los tipos de datos. En Java por ejemplo tenemos el tipo String, mientras que en Oracle existe el tipo Varchar2. No sólo no son el mismo tipo, sino que incluso ambos tipos cuentan con funcionalidad y restricciones diferentes (el tipo Varchar2 tiene un tamaño máximo de 2500 caracteres).
- Colecciones en objetos y relacional: en el paradigma orientado a objetos se cuenta con colecciones, las cuales modelan distintas situaciones, como por ejemplo conjuntos (Set) que no permiten elementos repetidos. En cambio en el paradigma relacional no sólo no existe el concepto de colección, sino que el

¹⁸ En el capítulo dedicado a los patrones de diseño en persistencia se presenta un patrón que establece justamente este punto.

¹⁹ El motivo que se esgrime en relacional es que no tiene sentido modelar una entidad Banco, que se sabe que luego al transformarse en tabla contendrá únicamente una sola tupla, que difícilmente varíe. Notar que lo mismo pasa en el paradigma orientado a objetos, salvo que en este caso la única instancia tiene lógica asociada.

único mecanismo para representarlo (las relaciones) no tienen toda la semántica asociada. Como muestra de esto último para modelar un conjunto es necesario definir restricciones o “*constraints*” que velan por la ausencia de tuplas con ciertos atributos repetidos.

- Jerarquías en objetos y en relacional: según lo que se vio en el capítulo anterior, al realizar una consulta con una superclase como destino en OQL uno puede obtener todas las instancias, tanto de la superclase como de todas las subclases de la misma. En cambio, al tener una jerarquía en el modelo relacional, no sólo es un concepto teórico que no tiene correlato en la implementación, sino que además en el SQL no es posible realizar una consulta a la tabla que contiene las tuplas de la super-entidad y que automáticamente se incluya a las tuplas de las entidades hijas.
- Identificación de los objetos y de las tuplas: en el modelo relacional se enseña que cada entidad tiene su identidad definida en base a sus atributos (por ejemplo el dni+nombre+apellido define a una persona). En cambio en objetos la identidad de un objeto queda definida por su OID, no siendo editable éste último por el usuario o desarrollador. Incluso, el OID ayuda a identificar unívocamente a una instancia en todo el sistema, mientras que en relacional el identificador suele restringirse a la tabla a la que pertenece, pudiendo repetirse el mismo identificador para diferentes tablas. Finalmente otro punto a señalar es que en relacional es posible identificar tuplas en base a identificadores de otras tuplas (claves foráneas). En objetos esto último no sería posible ya que se estaría ante una flagrante violación del encapsulamiento.

Características deseables del mapeo

Aún teniendo que enfrentarse con los problemas surgidos a partir de la diferencia de impedancia antes comentada, si se intentará construir una aplicación bajo el paradigma

de objetos con soporte relacional, ¿cuáles serían las características deseables de la función de mapeo entre ambos paradigmas?

Entre dichas características seguramente tendríamos:

- Idempotencia: como se mencionó en un capítulo previo, la idempotencia permite aplicar infinitas veces una función, sin que ésta altere el conjunto de datos sobre los que fue aplicada, dando como resultado también todas las veces el mismo resultado. Esto es sumamente importante al momento de mapear objetos a una base de datos. El mapeo no debe alterar nunca los objetos que está persistiendo, así como también siempre debe mapearlos de la misma manera.
- Transparencia para el sistema O.O. de los detalles de la persistencia; es decir, que el desarrollador de la aplicación O.O. no debería preocuparse (por lo menos no directamente) de los detalles de performance, de representación de la información, de cuantas tablas se están consultando, ni de las transacciones y otros conceptos propios de las bases de datos relacionales.
- Control de todas las propiedades A.C.I.D. A modo de repaso, estas propiedades establecen que un conjunto de operaciones debe ser ejecutado como un todo (A, de atomicity) y en caso de no poder ejecutar alguna operación todo el conjunto de operaciones tiene que ser deshecho; el estado en el que se deja la base de datos debe ser consistente, nunca en un estado indefinido (C, de consistency); todo cambio que se haga debe ser en absoluta independencia de los cambios que otros usuarios pudieran estar realizando al mismo tiempo (I, de isolation) y finalmente que todo cambio que llegue a la base debe ser durable en el tiempo (D, de durability).
- Performance sin compromisos. Si el desarrollador de la aplicación OO tiene que tomar decisiones respecto a cómo implementar una solución teniendo en cuenta aspectos de performance por sobre otros aspectos como por ejemplo patrones de diseño, entonces no tendríamos transparencia y por lo tanto no sería una situación buena.
- Integración fácil y sin limitaciones. Si los tipos de datos a ser utilizados están restringidos por el producto que se encargue del mapeo, entonces el diseñador

no es libre de plantear la mejor solución desde el punto de vista del diseño, por lo que no se cuenta ni con transparencia ni con independencia de la base de datos.

- Meta-data. Nuevamente persiguiendo la transparencia y la mantenibilidad de la aplicación, todos los aspectos relacionados con la persistencia deberían estar lo más separados posible del código en sí. Tener la meta-data requerida para el mapeo en archivos externos es una alternativa que permite variar el mapeo sin tener que alterar el código fuente.

¿Qué se mapea y que no?

Considerando que el mapeo no es más que una función que permite traducir objetos desde y hacia una base de datos relacional, y que en definitiva lo que se está haciendo es vinculando dos paradigmas distintos (el relacional y el orientado a objetos), cabe entonces preguntarse ¿cuáles de los elementos de cada uno es posible mapear al otro?

En objetos básicamente se cuenta con dos grandes “partes”: por un lado tenemos el comportamiento (lo principal) y por otro lado la estructura (funcional al comportamiento, y por lo tanto secundaria). Obviamente tanto el comportamiento como la estructura se podrían mapear, sin embargo el mapeo de cada uno de estas partes tiene ventajas y desventajas. Si tenemos en cuenta que lo que siempre se trata de evitar es repetir lógica en distintos lugares, debería quedar claro que no tiene muchas ventajas prácticas mapear el comportamiento. Si hay cuestiones de performance de por medio, puede ser necesario tener el comportamiento en la base de datos²⁰, pero entonces en este caso no se tendría el mismo comportamiento replicado en los objetos.

El caso de la estructura en cambio es diferente, ya que la estructura tiene como fin representar la información, la cual sí debe ser persistida en la base de datos ya que si no lo hace será siempre de carácter volátil. Es cierto que en un momento dado habrá

²⁰ Por ejemplo a través de funciones o stored procedures en PL/SQL.

información replicada, pero esto no representa un problema ya que en memoria siempre habrá copias locales y restringidas de la información consistente almacenada en la base de datos.

De los dos párrafos anteriores entonces se puede concluir que la función de mapeo suele tener más importancia para transformar la estructura de objetos en una estructura de tablas y tuplas más que para transformar el comportamiento.

Mapeo de clases

El escenario más simple es tener que mapear la estructura de una sola clase. En este caso, dicha clase estará compuesta solamente por atributos simples (recordar que se trata de un escenario con una sola clase). Dichos atributos probablemente correspondan con tipos simples como String, boolean, etc. En este caso la función de mapeo establece que a cada atributo simple le corresponderá una columna de una tabla y a cada instancia de dicha clase le corresponderá una nueva tupla de la tabla resultante.

La figura 11 muestra este escenario.

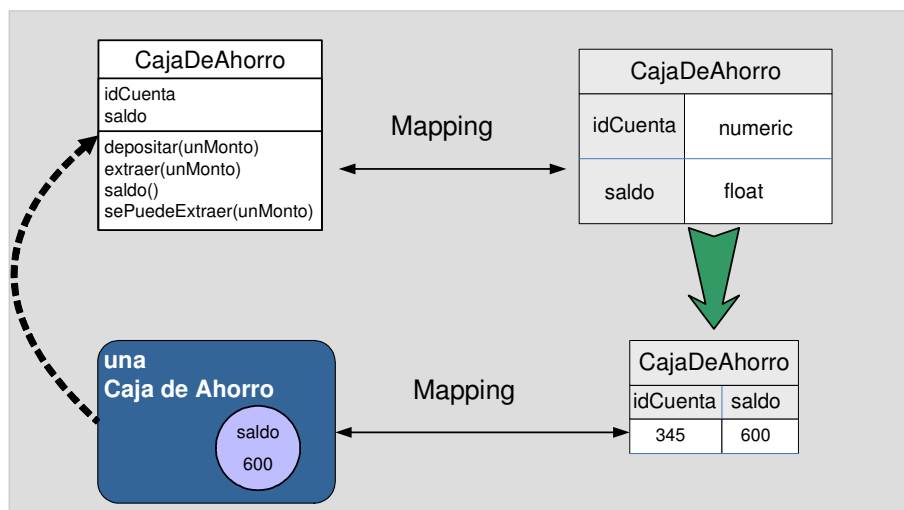


Figura 11: Mapeo de una clase a una tabla.

En conclusión se verifican las siguientes “igualdades”²¹

Clase = Tabla

Instancia = Tupla

Atributo = Columna

Notar que en el mapeo realizado en la figura 11 no contamos con una transparencia total, ya que hemos tenido que agregar una columna en la tabla para almacenar el identificador de la tupla (pk^{22}). Luego, este atributo tiene que ser reflejado también en la estructura de la clase (atributo *idCuenta*).

Estrategias de mapeo de jerarquías

En cualquier diseño de un sistema real es muy probable que no existan clases “sueltas”, sino que seguramente estarán organizadas algunas veces dentro de jerarquías de clases. En esta situación es necesario analizar las 3 estrategias de mapeo de jerarquías, ya que cada una tiene ventajas y desventajas que deben ser conocidas y comprendidas para poder tomar una decisión en cada oportunidad que se presente.

Para todos los ejemplos tomaremos la misma jerarquía de ejemplo, sobre la que se irán comentando los detalles de cada estrategia de mapeo y presentando pros y contras de cada alternativa.

La figura 12 presenta la jerarquía de clases de ejemplo.

²¹ Para evitar polémicas, estos conceptos NO son “iguales” en términos de su significado, sino que son igualados a partir de la aplicación de la función de mapeo.

²² Pk=Primary Key. Clave primaria que identifica unívocamente a una tupla dentro de la tabla.

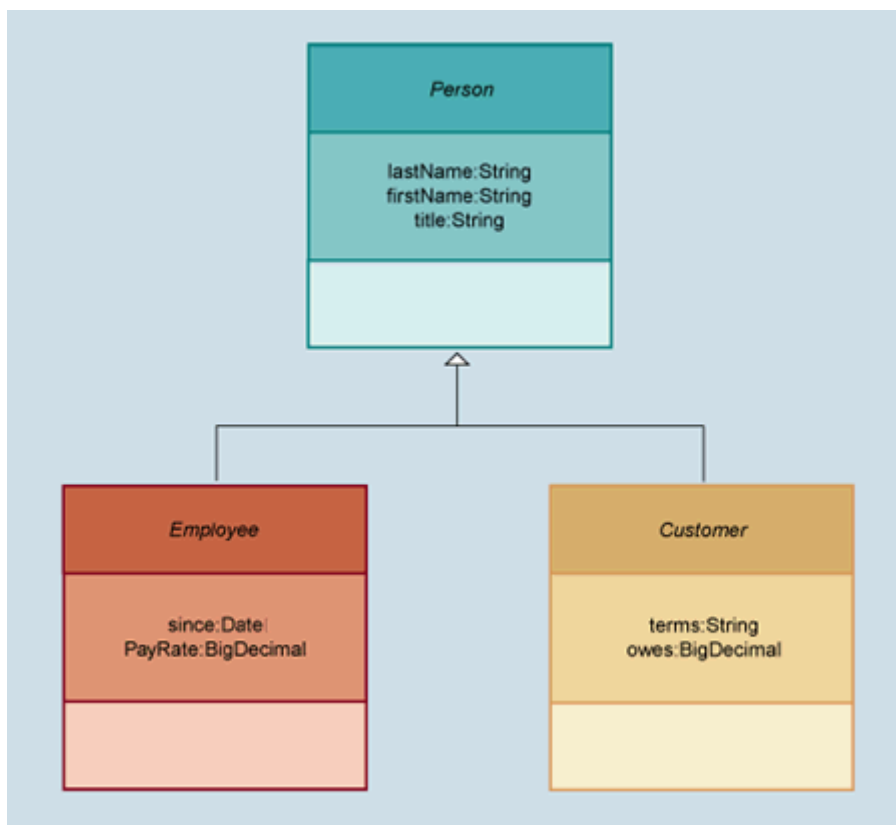


Figura 12: Jerarquía de clases.

Mapeo de la jerarquía a una sola tabla

En esta alternativa todas las clases pertenecientes a una jerarquía (incluso las clases abstractas) se mapean todas “juntas” a una sola tabla destino, que contendrá tantas columnas como atributos distintos existan en toda la jerarquía.

De acuerdo con lo anterior, para la jerarquía de personas, en esta alternativa tendríamos la tabla de la figura 13

Person
<u>OID</u> name phoneNumber customerNumber preferences startDate objectType

Figura 13: Tabla resultante al mapear toda la jerarquía a una sola tabla

Como puede verse en la figura 13, a cada uno de los atributos de las diferentes clases le corresponde una columna propia para mantener dicha información.

A. Ventajas de esta solución

1. Toda la información se encuentra en una sola tabla, por lo que las consultas que deban recorrer todos los elementos de una jerarquía (por ejemplo “*recuperar todas las personas que....*”) serán extremadamente rápidas ya que todas las tuplas están en la misma tabla y por lo tanto no es necesario ejecutar **joins**.
2. Cuando se realiza un cambio en la estructura de la clase al tope de la jerarquía y se quiere propagar este cambio a las tablas involucradas en el mapeo, es muy simple ya que solamente se debe agregar la nueva columna a la única tabla presente.

B. Desventajas de esta solución

1. La tabla siempre contiene tuplas con algunas columnas vacías. Si la tupla corresponde a un empleado (Clase Employee) todos los campos correspondientes a la clase Customer estarán vacíos, lo cual es una pérdida de espacio al menos.
2. Se torna difícil cuando existe el mismo atributo en dos clases de la misma jerarquía, pero con diferente semántica o rango de valores posibles.
3. Es necesario incorporar una nueva columna a la tabla que especifique a qué clase de la jerarquía pertenece la tupla que se está leyendo²³.

²³ En la figura 13 se muestra esta nueva columna con el nombre objectType.

4. Cada vez que se quiera hacer una consulta que recorra las instancias de una clase concreta (por ejemplo Employee) se estarán recorriendo también las instancias de las demás clases.
5. Al realizar cambios en la estructura de una clase siempre se termina impactando en el mapeo de todas las clases, ya que todas las clases comparten la misma tabla destino.

Mapeo con una tabla por clase

En esta alternativa cada una de las clases recibe una tabla que albergará su información, incluso para el caso de las clases abstractas. Siguiendo con el ejemplo de la jerarquía de Personas, el mapeo resultaría en las siguientes tablas:

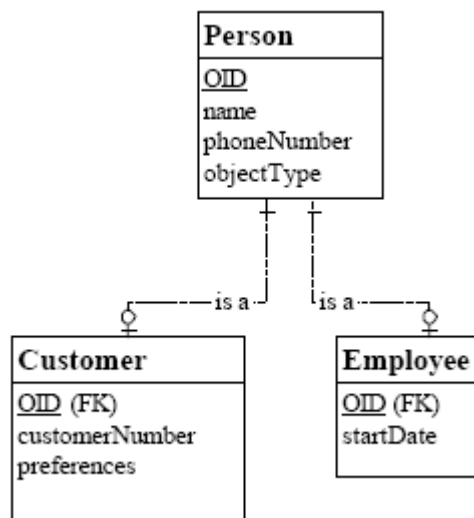


Figura 14: Mapeo de todas las clases con su correspondiente tabla

Como puede observarse, la tabla Person es la que mantiene la identificación de las instancias, y luego cada tabla “concreta” (es decir, cada tabla perteneciente a una clase concreta) tiene un identificador como clave foránea (FK) a la tabla Person, aumentando la información de la tabla Person con los datos propios.

A. Ventajas de esta solución

1. Desde el punto de vista de la orientación a objetos, esta alternativa es la que más se asemeja. Se tiene una tabla que representa la clase abstracta, en la que se definen los atributos comunes a todas las subclases; y luego tablas para las clases concretas que “apuntan” a la tabla de la clase abstracta.
2. Cuando se realiza un cambio que debe propagarse a cada clase concreta es necesario alterar solamente la clase abstracta. El resto de las tablas no debe ser modificado. En este sentido es similar a la alternativa anterior.
3. Cuando se requiere consultar el conjunto de instancias de una clase en particular, simplemente basta con apuntar la consulta a la tabla asociada sin tener que recorrer inútilmente el resto de las instancias (siempre y cuando no se requieran datos que están en la tabla Person).
4. Si hay un atributo con el mismo nombre en dos o más clases, cada una lo puede mapear como más le convenga.
5. Para aquellas consultas que deban apuntar a todas las instancias de la clase Person (recordar que todas las instancias de las clases Employee y Customer son en definitiva también instancias de la clase Person) basta con dirigir la consulta a una sola tabla (Person). Esto suponiendo que dicha tabla contiene toda la información requerida.

B. Desventajas de esta solución

1. Quizás la mayor desventaja de esta solución de mapeo resida en la necesidad de realizar siempre joins para poder recuperar toda la información que compone a una instancia. Esto se debe a que la parte común a todas las instancias se encuentra en la tabla Person y la parte correspondiente a la Clase Customer está en la tabla Customer.

Mapeo de las clases concretas únicamente

En esta alternativa se mapean solamente las clases concretas, por lo que es necesario “mover” todos los atributos presentes en la clase abstracta a cada una de sus subclases concretas. Con este esquema, las tablas resultantes serían las que se muestran en la figura 15

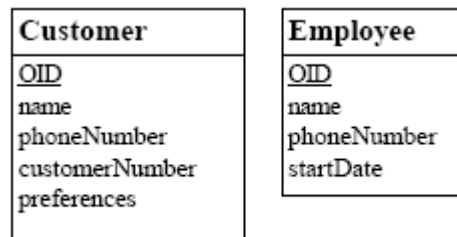


Figura 15: Tablas resultantes del mapeo de clases concretas únicamente

Como puede observarse, los atributos *name* y *phoneNumber* (definidos en la clase abstracta Person) ahora aparecen en las tablas Customer y Employee.

A. Ventajas de esta solución

1. La principal ventaja de esta propuesta es que de ahora en más para recuperar toda la información de una instancia basta con recuperar una tupla de la tabla que corresponda. No es necesario ejecutar **joins** entre varias tablas para “armar” nuevamente una instancia.
2. Otra de las ventajas es que los conjuntos de instancias están separados. Esto permite recorrer solamente alguno de los conjuntos al tratar de recuperar una instancia (o conjunto de instancias) determinadas. Por ejemplo, si se quiere recuperar un cliente, no es necesario recorrer los empleados (en el caso del mapeo de toda la jerarquía a una sola tabla esto suele suceder, sobre todo si no se cuenta con índices apropiadamente definidos).

B. Desventajas de esta solución

1. Cuando se requiere devolver el “extent” de una clase (es decir, el conjunto de todas sus instancias), y dicha clase es la que fue “dividida” (en nuestro ejemplo sería la clase Person) se debe ejecutar una de las operaciones más caras del SQL: UNION. Entonces para recuperar todas las personas habría que realizar la siguiente consulta

*select * from Customer*

UNION

*select * from Employee*

2. Otra desventaja tiene que ver con la estructura repetida en varias tablas. Supongamos el siguiente caso: se desea agregar un nuevo atributo a la clase abstracta Person. En este caso se debe obviamente agregar el atributo en la clase y luego recordar en qué tablas se repartió la información de esta clase para agregar tantas columnas como sea necesario. Para ser más concretos, habría que recordar que existen dos tablas (Customer y Employee) que representan a las instancias de las clases Customer y Employee respectivamente, y que ambas deben contener ahora un nuevo atributo. Con 2 o más tablas no representa un gran problema, pero en diseños de sistemas grandes en donde se cuenta con 2000 tablas o más esto se vuelve un problema grave (sobre todo considerando que la documentación no es precisamente el foco de nuestra atención). Para el caso del borrado de un atributo sucede lo mismo.

Mapeo de asociaciones, composiciones y agregaciones

Una asociación no es más que un vínculo existente entre dos instancias. Por otro lado, la manera de vincular tuplas de una base de datos relacional es a través de relaciones, por lo tanto para mapear asociaciones se deberán utilizar relaciones (1 a 1, 1 a n, etc.).

Antes de comenzar con el análisis de cada una de las posibles situaciones al mapear asociaciones conviene refrescar los conceptos de agregación y composición del paradigma orientado a objetos:

- **Agregación:** es una asociación entre dos instancias, en la que una de las instancias forma parte de la otra. No existe relación entre los tiempos de vida de cada una de las instancias.
- **Composición:** es una asociación “fuerte” entre dos instancias, en donde se establece la relación “es parte de” y por lo tanto se vinculan los tiempos de vida de ambas instancias. En otras palabras, al desaparecer el objeto “contenedor” debe desaparecer el objeto “parte”.

¿Qué diferencias entonces aparecen al mapear, mediante relaciones, una agregación o una composición? La respuesta es, en realidad, ninguna o casi ninguna. “Ninguna” en el sentido de que si uno mira únicamente las tablas y relaciones resultantes, no se puede distinguir si se trata de una agregación o de una composición. “Casi ninguna” porque puede ser que aparezcan otros elementos de las bases de datos relacionales, como Stored Procedures o Triggers que se disparen en cascada para eliminar una tupla (parte) cuando se elimina otra tupla (contenedor).

Asociaciones 1 a 1

Este tipo de asociaciones se mapea directamente agregando en una tabla una columna más que mantendrá la clave primaria de tupla de la otra tabla referenciada. En el caso de que se trate de una asociación bidireccional, ambas tablas deberán agregar una columna

extra para referenciar la clave primaria de la otra entidad. Es el caso más simple ya que no es necesario crear una tabla intermedia.

Asociaciones 1 a n

Este tipo de asociaciones generalmente está relacionado con el mapeo de las colecciones de objetos. En este caso se debe seguir la misma idea que al pasar a tablas un diseño relacional en donde se tienen relaciones 1 a n, es decir “pegando” del lado de n la clave primaria del lado de 1. Para ser más explícitos, si tenemos una clase *Persona*, que tiene varios Teléfonos (en otras palabras la clase *Persona* tiene una colección de teléfonos), ambas clases serán mapeadas a sendas tablas *Tabla_Persona* y *Tabla_Teléfono* respectivamente. Adicionalmente a la tabla *Tabla_Teléfono* se debe agregar una columna extra para referenciar la clave primaria de *Tabla_Persona*, de modo de poder determinar para cada teléfono quién es la persona dueña del mismo. Nuevamente no ha sido necesaria la creación de una tercera tabla.

Asociaciones n a n

Este caso es bastante parecido al anterior, con la salvedad de que ahora no basta con agregar columnas a las tablas resultantes del mapeo, sino que se debe crear una tercera tabla que mantenga las relaciones entre las tuplas de la relación “n a n”.

Siguiendo con el ejemplo, si tenemos la clase *Persona* y la clase *Cuenta* (en donde una persona puede tener varias cuentas, y a su vez una cuenta pueda pertenecer a varias personas) en mapeo resultante podría ser:

Tabla_Persona (**persona_pk**,...)

Tabla_Cuenta (**cuenta_pk**,...)

y finalmente la relación que une a cada persona con sus cuentas y viceversa,

Tabla_Persona_Cuenta(**persona_pk**, **cuenta_pk**, ...).

Un punto adicional a discutir es el relacionado con las diferencias que se deben tener en cuenta al mapear colecciones ordenadas. En este caso, al recuperar los elementos de una colección debemos poder recuperarlos exactamente en el mismo orden en el que se encontraban al momento de persistirlos.

¿Cómo se puede lograr mantener el orden? En realidad es sumamente sencillo. Basta con agregar una nueva columna que mantenga el “orden” de la tupla dentro de la colección de un objeto (representado a través de su clave primaria). En este punto hay detalles que pueden hacer bastante más complejo este tema: ¿qué sucede al eliminar elementos de la colección ordenada? y ¿cómo afecta la presencia de un orden a la performance? Ambos puntos serán comentados con mayor profundidad en los próximos capítulos.

Mapeo del identificador

El OID es el identificador de una instancia, generalmente oculto al desarrollador, que permite identificar unívocamente al objeto. Si la información de un objeto será almacenada en una tupla, entonces es necesario definir alguna manera de identificar unívocamente también a dicha tupla. Esto puede lograrse mediante la definición de una clave primaria.

Si bien parecen ser exactamente los mismos conceptos, hay una diferencia importante entre OID y clave primaria. En la teoría de bases de datos relacionales se suele proponer que la clave primaria esté compuesta por valores del dominio, en cambio los OIDs son administrados enteramente por el ambiente, haciendo muchas veces imposible siquiera el poder consultarlo.

Esto último plantea el primer problema importante al momento de mapear un objeto a una tupla. Supongamos el siguiente caso: existe una instancia de la clase *Persona*, cuya información se almacena en la tabla *TPersona*. Al modificar la información en la instancia y persistir dichos cambios, ¿cómo se hace para determinar cuál de todas las tuplas de la tabla que corresponda es la que representa a la instancia que estoy modificando (recordar que desde el mundo de objetos no suele tenerse acceso al OID)?

La solución más simple que se suele adoptar es agregar un nuevo atributo a la clase. Dicho atributo tendrá como finalidad almacenar el valor de la clave primaria, de modo que cuando se desee persistir el objeto se podrá acceder rápidamente a la clave que identifica la tupla asociada. Esta solución es simple y funciona. Pero no es del todo elegante desde el punto de vista de objetos ya que terminamos por romper una de las primeras características deseables del mapeo: la transparencia. A partir de ahora todo diseñador OO “sabe” explícitamente que las instancias de una clase se está persistiendo en una base de datos relacional.

Obviamente existen otras alternativas para solucionar este problema pero todas, en mayor o menor medida, terminan por romper la transparencia, algunas agregando incluso bastante más complejidad.

Hibernate

Unos años atrás cuando uno se proponía desarrollar una aplicación orientada a objetos con persistencia relacional, en forma indirecta terminaba desarrollando software para realizar el mapeo entre ambos paradigmas. No sólo es tedioso y complicado desarrollar este tipo de software, sino que además probablemente uno podía hacer realmente poco reuso entre aplicaciones distintas, por lo que se debía volver a desarrollar dicha capa de mapeo una y otra vez.

Adicionalmente, como el objetivo de la aplicación es resolver problemas de un dominio determinado, seguramente los desarrolladores conozcan más dicho dominio que el dominio de la persistencia. Por esta razón, es evidente que las soluciones de persistencia “caseras” no eran óptimas, y mucho menos estándares.

Todo eso comenzó a cambiar a fines del año 2001 con la publicación en SourceForge de un proyecto open-source denominado Hibernate, el cual a través de los años se convertiría prácticamente en el estándar de facto para el mapeo de objetos a bases de datos relacionales.

Hibernate ha sido tan importante, primero en el mundo Java, que incluso ha sido portado a otras plataformas como .Net con su implementación open-source NHibernate.

Arquitectura

La arquitectura de este producto ha ido evolucionando a lo largo de los años, pero básicamente sigue respetando la división inicial en tres grandes componentes:

- Administración de las conexiones: Hibernate administra las conexiones a las bases de datos de manera de optimizar este recurso tan costoso.
- Administración de las transacciones: el mantenimiento de las propiedades ACID es primordial para el correcto funcionamiento del sistema, por lo que Hibernate no lo deja librado al usuario sino que es directamente responsable de administrar las transacciones.
- Mapeador: este componente es el encargado de insertar, actualizar, borrar y recuperar la información persistida. Se apoya en los otros componentes así como en otros recursos que deben ser definidos por el usuario.

La figura 16 muestra un esquema simplificado de la arquitectura de Hibernate

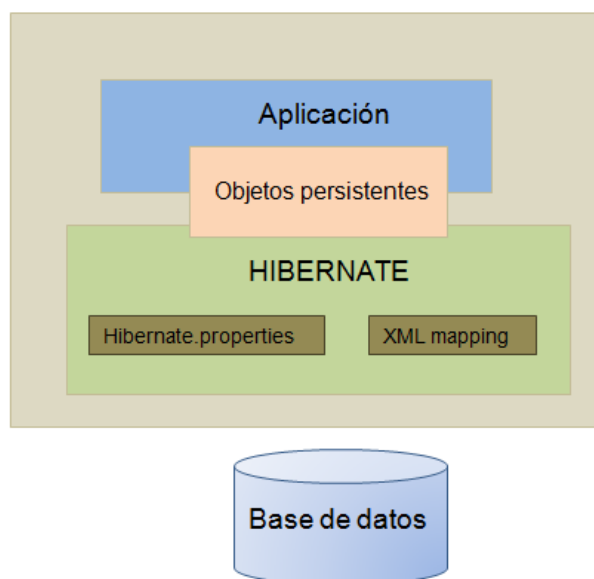


Figura 16: Arquitectura de Hibernate

Como puede observarse a partir de la figura anterior, no hay más interacción directamente entre la aplicación y la base de datos. La aplicación sólo trabaja con los objetos persistentes, los cuales al estar administrados por Hibernate, son persistidos en forma “cuasi-automática”.

Forma de trabajo

Para trabajar con Hibernate es necesario realizar las siguientes actividades:

- Crear las clases correspondientes al modelo. Este paso no tiene demasiadas variantes respecto de lo que se haría en un caso sin persistencia, siempre y cuando se tomen los recaudos suficientes. Por recaudos necesarios nos referimos tanto a cuestiones obvias a estas alturas, como no escribir SQL en las clases del dominio; como a cuestiones más bien sutiles, como por ejemplo el uso de annotations. Estas últimas, sin bien en algunos casos pueden resultar de utilidad, al utilizarlas relacionadas con la persistencia traen más desventajas que ventajas (al final de este capítulo se encuentra una breve discusión sobre este punto). A diferencia de otras alternativas de persistencia, con Hibernate no es necesario post-compile las clases o aplicarles ningún otro proceso.
- Crear archivos de mapeo: para que Hibernate conozca cuales son las tablas a las que se debe enviar la información de cada clase, qué estrategia de mapeo de jerarquía se decidió utilizar, y toda otra información oportuna, se deben crear archivos de mapeo. Estos archivos contendrán para cada clase (o jerarquía de clases) toda la información requerida para poder persistir y luego recuperar los datos. Estos archivos mantienen el nombre de la clase pero su extensión es “.hbm.xml”. Entonces si estamos mapeando la clase Car, tendremos el archivo de mapeo Car.hbm.xml. Estos archivos suelen ser ubicados en los mismos directorios que las clases que mapean, aunque con configuración este punto puede ser alterado.
- Configurar las librerías de Hibernate. Aunque resulte obvio este punto, hay que incluirlo en la lista ya que muchas veces los errores que se levantan a raíz de la utilización de este producto tienen que ver con la falta de librerías para su correcto funcionamiento. Para facilidad del usuario, en estos momentos las

librerías de Hibernate están divididas teniendo en cuenta si se tratan de extensiones o es parte del núcleo básico requerido.

- Configurar Hibernate. Además de los mapeos propiamente dichos hay que configurar Hibernate con información del esquema de base de datos al cual debe acceder, con qué protocolo de SQL debe hacerlo, si debe utilizar caché o no, etc. Todos estos elementos son configurados en un archivo denominado `hibernate.cfg.xml`, el cual es leído una única vez al levantar la aplicación para optimizar el proceso de arranque y configuración.
- Abrir una sesión de trabajo y trabajar con los objetos. Finalmente, el último paso que resta es abrir una conexión a la base de datos a través de Hibernate, operar con los objetos del dominio y al cerrar la sesión de trabajo, si todo está configurado correctamente, los cambios deberían verse reflejados en la base de datos.

Principales componentes

En esta sección se comentarán brevemente los principales componentes de Hibernate más utilizados.

- `SessionFactory`: es una caché (inmutable) de mapeos compilados para una base de datos. Además su función es permitir la creación de sesiones de trabajo.
- `Session`: representa una “conversación” entre la aplicación y el almacenamiento persistente. Permite persistir y recuperar objetos y crear transacciones. Además mantiene una caché de objetos persistentes, usada cuando se navega el grafo de objetos o para buscar objetos por identificador.

Dentro del protocolo de `Session` tenemos un método muy relevante:

```
public Serializable save(Object object) throws HibernateException
```

Este método es la base para implementar un mecanismo de persistencia transparente ya que para almacenar un objeto solamente es necesario invocar a este método pasándole el objeto que se desea persistir.

Ahora bien, si tenemos en cuenta que Hibernate provee **persistencia por alcance**, la pregunta es ¿cuántas veces es necesario invocar a este método dentro de la aplicación?

Aunque suene extraño, la respuesta es 0 (cero). ¿Cómo es esto de que se puede persistir información a través de Hibernate invocando exactamente cero veces el método `save()`? Bien, para responder a esta pregunta podemos analizar escenarios y ver que todos reducen siempre al mismo caso base.

Escenario 1: supongamos que tenemos una aplicación que invoca n veces al método `save()`. En este caso hay que notar que Hibernate realizará un análisis de si la instancia que se está pasando como parámetro es nueva o si en cambio ya está persistida y sólo hay que actualizarla. En cualquier caso se guardará la información de esta instancia, pero además Hibernate debe asegurarse que todos los objetos que “relacionados” con dicha instancia también estén actualizados. Por ejemplo, si tenemos un Banco y estamos agregando un nuevo Cliente a dicho banco (llamando al `save()` con el cliente), hay que notar que Hibernate revisará la instancia de Banco para ver si no debe actualizarla también. Como conclusión podemos ver que aún cuando se llame al `save()` con un objeto determinado, Hibernate debe revisar los objetos relacionados para actualizarlos posiblemente.

Escenario 2: ¿qué pasa si en vez de llamar directamente al `save()` utilizamos persistencia por alcance?. En este caso deberíamos recuperar la instancia del Banco y ejecutar el método `add()` de la colección de clientes del Banco. Al realizar esto, el banco quedará marcado como modificado y Hibernate persistirá todos los cambios automáticamente (además de insertar al nuevo cliente). En este escenario podríamos considerar que solamente es necesario llamar una sola vez al método `save()` (para persistir la primer instancia del Banco).

Escenario 3: si bien es cierto y muy probable el escenario 2, todavía tenemos un `save()`. Si ahora consideramos que el Banco y otros objetos son objetos requeridos para la correcta ejecución de la aplicación, uno puede darse cuenta de que ya deberían estar creados antes de la primera ejecución de la aplicación. Si seguimos con este razonamiento pronto nos daremos cuenta de que la aplicación

puede “asumir” que el Banco ya existe y no necesitará nunca llamar al método **save()**²⁴. Todo esto se debe nuevamente a la posibilidad que da la **persistencia por alcance**.

- Transaction: define una unidad de trabajo atómica.

Ahora que conocemos los elementos más importantes podemos ver un pseudo-código que muestra la utilización de Hibernate. La figura 17 muestra dicho código

```
Session sess = factory.openSession();
Transaction tx;
try {
    tx = sess.beginTransaction();
    //do some work
    ...
    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    sess.close();
}
```

Figura 17: Pseudo-código de utilización de Hibernate²⁵

Como puede observarse en la figura 17, la forma de trabajo es relativamente simple

1. Se crea una nueva session a través de una instancia de SessionFactory (en la figura se denomina *factory*).
2. Para comenzar a trabajar con los objetos se abre una transacción a partir de la session.

²⁴ El banco y los demás objetos requeridos pueden ser creados a través de scripts de SQL directamente en la base de datos o a través de instaladores.

²⁵ Este pseudo-código figura en la API de la clase Session provista por Hibernate.

3. Al finalizar el trabajo se cierra la transacción (*commit()*) y si todo funciona bien los cambios se persisten a la base de datos.

En la siguiente sección se muestra un ejemplo simple pero completo para ejemplificar el uso básico de Hibernate.

Ejemplo simple de uso

A continuación se presentan todos los artefactos relacionados con un ejemplo simple de utilización de Hibernate²⁶. Si bien hay partes que no fueron explicadas explícitamente en las secciones anteriores, son relativamente simples los conceptos por lo que el lector no debería tener problemas en comprenderlos.

La figura 18 presenta el diagrama de clases que se desea persistir.

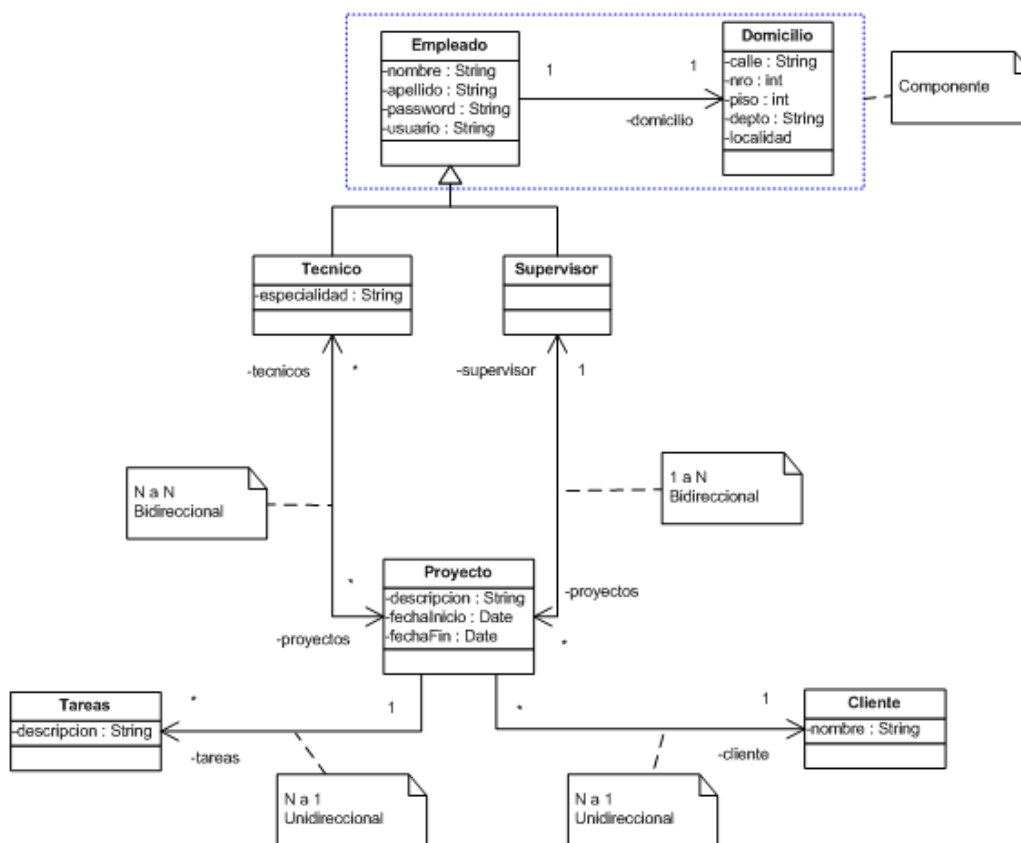


Figura 18: Diagrama de clases del ejemplo.

²⁶ El código de las clases del modelo no se muestra aquí ya que no tienen ninguna complejidad adicional. El proyecto en SourceForge que acompaña este libro contiene este código en caso de que el Lector quiera obtenerlo.

Como puede observarse en la figura anterior, se trata de un modelo de objetos simple y pequeño. El único detalle que merece la atención es la declaración del domicilio de un empleado como un “componente”, lo que permite recuperar al domicilio en el mismo momento que se recupera al empleado relacionado.

En la siguiente figura puede verse el archivo de configuración de Hibernate, hibernate.cfg.xml, en el que se pueden ver todos los elementos que se deben configurar para poder utilizar apropiadamente este mapeador. Entre los parámetros que se deben configurar tenemos el driver de conexión a la base de datos, usuario y clave para la misma y adicionalmente se deben definir todos los archivos de mapeo que se deben importar.

Usualmente durante la etapa de desarrollo es habitual configurar el parámetro “*show_sql*” para poder ver en la consola las sentencias sql que genera automáticamente Hibernate.

```
<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>
        <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property name="connection.username">example</property>
        <property name="connection.password"></property>
        <property name="connection.url">jdbc:hsqldb:file:data/DB</property>
        <property name="show_sql">true</property>

        <mapping resource="hibernate/Empleado.hbm.xml" />
        <mapping resource="hibernate/Cliente.hbm.xml" />
        <mapping resource="hibernate/Tarea.hbm.xml" />
        <mapping resource="hibernate/Proyecto.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Figura 19: Archivo de configuración de Hibernate.

La figura 20 muestra el archivo de mapeo correspondiente a la clase Tarea, Tarea.hbm.xml. Como todo archivo XML debe ser bien formado y respetar las etiquetas de Hibernate adecuadamente. En la siguiente figura puede apreciarse como se configura

una columna para la clave primaria (*oid*) además de otras columnas para el nombre y la descripción (en el caso de estas dos últimas columnas, se tomará el nombre del atributo presente en la clase para el nombre de la columna en la tabla a falta de una definición de dicho nombre en el archivo de mapeo).

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="modelo">
    <class name="Tarea">
        <id name="oid" type="long" column="oid_tarea" unsaved-value="0">
            <generator class="increment" />
        </id>
        <property name="nombre" />
        <property name="descripcion" />
    </class>
</hibernate-mapping>
```

Figura 20: Archivo de mapeo de la clase Tarea.

En la figura 21 tenemos el archivo de mapeo de la clase Empleado, Empleado.hbm.xml.

En este archivo tenemos no sólo el mapeo correspondiente a la clase Empleado, la cual es abstracta, sino que también hemos puesto las definiciones de los mapeos de las subclases Supervisor y Técnico.

Otro punto a notar es la definición del domicilio como *componente* del empleado. En Hibernate esto significa que siempre que se recupere un empleado se recuperará también su domicilio sin necesidad de ejecutar una consulta adicional. También hay que notar que un domicilio no puede recuperarse sólo, sino que siempre debe ser accedido a través de su empleado ya que no cuenta con un identificador propio.

Finalmente, se puede ver como algunos atributos generales son mapeados en la clase Empleado (como *nombre* y *apellido*); mientras que otros atributos propios del supervisor son declarados únicamente para esta clase (colección *proyectos*) y en forma similar para los técnicos (atributo *especialidad*).

```

<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="modelo">
  <class name="Empleado">
    <id name="oid" type="long" column="oid_empleado" unsaved-value="0">
      <generator class="increment" />
    </id>
    <discriminator column="emp_tipo" type="string" />

    <property name="nombre" />
    <property name="apellido" />
    <property name="password" />
    <property name="usuario" />

    <property name="dni" />

    <component name="domicilio" class="Domicilio">
      <property name="calle" />
      <property name="nro" />
      <property name="piso" />
      <property name="depto" />
      <property name="localidad" type="string" />
    </component>

    <subclass name="Supervisor" discriminator-value="S">
      <set name="proyectos" cascade="all">
        <key column="oid_empleado" />
        <one-to-many class="Proyecto" />
      </set>
    </subclass>

    <subclass name="Tecnico" discriminator-value="T">
      <set name="proyectos" table="tecnicos_proyectos" cascade="all" >
        <key column="oid_empleado" />
        <many-to-many column="oid_proyecto" class="Proyecto" />
      </set>
      <property name="especialidad" />
    </subclass>
  </class>
</hibernate-mapping>

```

Figura 21: Mapeo de la clase Empleado.

La figura 22 muestra el mapeo correspondiente a la clase Cliente, Cliente.hbm.xml. A diferencia del mapeo de empleados, este mapeo es sumamente simple.

Hibernate permite utilizar una gran variedad de alternativas para la generación de las claves para las tuplas; en estos ejemplos se ha estado utilizando la estrategia “*increment*” que indica a Hibernate que se haga cargo de generar automáticamente un número incremental para asignar la clave a cada tupla que se inserte.

```

<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="modelo">
    <class name="Cliente">
        <id name="oid" type="long" column="oid_cliente" unsaved-value="0">
            <generator class="increment" />
        </id>

        <property name="nombre" />
        <property name="dni" />

    </class>

</hibernate-mapping>

```

Figura 22: Mapeo de la clase Cliente.

El mapeo correspondiente a la clase Proyecto, Proyecto.hbm.xml, se puede apreciar en la figura 23.

En esta figura nuevamente contamos con un mapeo mucho más interesante debido a su complejidad, ya que se cuenta con diferentes mapeos:

- de atributos simples (*nombre* y *descripción*);
- colecciones unidireccionales 1 a n con el supervisor (un supervisor supervisa muchos proyectos);
- colecciones bidireccionales (relación entre *proyectos* y *técnicos*);
- colecciones 1 a muchos (colección de *tareas*).


```

<?xml version="1.0"?>

<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping package="modelo">
    <class name="Proyecto">

        <id name="oid" type="long" column="oid_proyecto" unsaved-value="0">
            <generator class="increment" />
        </id>

        <property name="nombre" />
        <property name="descripcion" />
        <property name="fechaInicio" />
        <property name="fechaFin" />

        <many-to-one name="supervisor" column="oid_empleado" not-null="true"
        cascade="all"/>

        <set name="tecnicos" table="tecnicos_proyectos" cascade="all" inverse="true">
            <key column="oid_proyecto" />
            <many-to-many column="oid_empleado" class="Tecnico" />
        </set>

        <set name="tareas" cascade="all">
            <key column="oid_proyecto" />
            <one-to-many class="Tarea" />
        </set>

        <many-to-one name="cliente" column="oid_cliente" not-null="true" cascade="all" />

    </class>
</hibernate-mapping>

```

Figura 23: Mapeo correspondiente a la clase Proyecto.

Finalmente, para ver un extracto de código en el que se utiliza Hibernate para persistir información del modelo tenemos que analizar la siguiente figura en la que se realizan varias actividades, desde configurar Hibernate, crear objetos de dominio para finalmente persistirlos. Es de esperar que en una aplicación productiva todas estas actividades no estén codificadas en un solo lugar, sino que cada una pertenezca a una capa bien identificada.

```

1      Configuration cfg = new Configuration();
2      cfg.configure();
3      new SchemaExport(cfg).create(true, true);
4      SessionFactory sessions = cfg.buildSessionFactory();
5      Session session = sessions.openSession();
6      Domicilio dom1 = new Domicilio("Calle1", 1, 1, "Dpto1", "La Plata");
7      Cliente cli1 = new Cliente("Juan Perez", "1");
8      Supervisor sup1 = new Supervisor("Maria", "Ru", "mr", "1", "1");
9      Tecnico tec = new Tecnico("Juan", "Martin", "ja", "14", "5", "elec");
10     sup1.setDomicilio(dom1);
11     Tarea t1 = new Tarea("Analisis", "Tarea1");
12     DateFormat dfm = new SimpleDateFormat("yyyy-MM-dd");
13     Date d = dfm.parse("2006-02-26");
14     Date d2 = dfm.parse("2008-09-26");
15     Proyecto p1 = new Proyecto("Proyecto1", "Descripcion1", d, d2,
16                               cli1, sup1);
17     sup1.agregarProyecto(p1);
18     p1.agregarTecnico(tec);
19     tec.agregarProyecto(p1);
20     p1.agregarTarea(t1);
21     Transaction tx = null;
22     try {
23         tx = session.beginTransaction();
24         session.save(p1);
25         tx.commit();
26     } catch (Exception e) {
27         if (tx != null)
28             tx.rollback();
29         session.close();
30     }
31     session.disconnect();

```

Figura 24: Utilización de Hibernate.

Algunas aclaraciones en cuanto a la figura 24:

- En las líneas 1 a 3 se configura Hibernate, indicándole que lea su archivo de configuración y que además cree el esquema requerido en la base de datos.
- En las líneas 4 y 5 se realiza la conexión a la base de datos y se obtiene una sesión de trabajo para poder trabajar con la base de datos.
- Las líneas 6 a 20 crean diferentes instancias del modelo y los vinculan.
- La línea 23 abre una transacción a partir de la sesión de trabajo.
- La única línea requerida para persistir todo el modelo recientemente creado es la línea 24 a través de la invocación del método **save()**. En este caso no contábamos con un modelo persistente por lo que no pudimos hacer uso de la persistencia por alcance en su forma más potente, sin embargo notar que

solamente se está pidiendo explícitamente que se almacene el proyecto y en cascada todos los demás objetos también se persisten.

- La línea 25 realiza el commit de la transacción.
- Las restantes líneas son necesarias para el caso de que ocurra algún error en la persistencia.

Consultas

Esta solución de mapeo brinda 4 maneras de recuperar la información:

- Iteradores: esta es la manera más “orientada a objetos”. Si uno recupera un objeto cualquiera, y dicho objeto tiene colaboradores que son colecciones, entonces es posible obtener un iterador y a partir de éste recorrer todos los elementos. Es necesario aclarar que dicho iterador no recupera inmediatamente todos los elementos de la colección, lo que algunas veces significa una mejora en la performance y otras veces puede perjudicarla. En el capítulo dedicado a los patrones de diseño se verá con más detalle este punto.
- SQL: en algunas circunstancias es necesario contar con la posibilidad de ejecutar consultas SQL. A través de la conexión que se obtiene de la sesión de trabajo la aplicación puede enviar todas las sentencias SQL que requiera. Obviamente esta alternativa debe ser utilizada en casos de extrema necesidad y no como método preferido.
- HQL: tomando las mejores características de OQL, Hibernate presenta un lenguaje de consulta “orientado a objetos”, con el cual se pueden recuperar los objetos persistidos. Obviamente no todas las características del OQL se han implementado, pero por lo menos las más importantes se encuentran disponibles. Todas las consultas en HQL finalmente son traducidas a SQL para poder ser ejecutadas por el motor de la base de datos relacional. La performance de esta alternativa es muy buena, incluso comparándola con SQL, razón por la cual es la forma preferida de realizar consultas.

Más abajo se encuentran algunas consultas de ejemplo en HQL para demostrar su potencia²⁷.

1. Listar los nombres de todos los proyectos

```
SELECT p.nombre  
FROM modelo.Proyecto p
```

Notar que se debe utilizar el nombre de la clase y no el nombre de la tabla para especificar lo que se quiere consultar.

2. Listar los nombres de los proyectos que tienen como cliente, al cliente cuyo nombre es “*Carlos Rodríguez*”

```
SELECT p.nombre  
FROM modelo.Proyecto p  
WHERE p.cliente.nombre = 'Carlos Rodriguez'
```

En este caso hay que remarcar que no es posible utilizar métodos, sino que se debe acceder directamente a la estructura.

3. Listar los técnicos que tienen especialidad “*electrónica*”, que están en los proyectos con tarea “*Análisis*” y supervisados por el supervisor con apellido “*Ruiz*”

```
SELECT new list(tecnico.nombre,tecnico.apellido)  
FROM modelo.Proyecto p  
    join p.tareas as tarea  
    join p.tecnicos as tecnico  
WHERE tecnico.especialidad = 'electrónica'  
    and tarea.nombre = 'Análisis'  
    and p.supervisor.apellido='Ruiz'
```

En este ejemplo se devuelve un nuevo tipo de resultado (una lista con dos atributos por cada técnico); además se utiliza la palabra **join** para especificar que se quiere ligar a cada tarea de un proyecto con la variable tarea y en forma similar con la variable técnico para luego verificar condiciones sobre estas variables.

²⁷ Para las consultas de ejemplo se utilizará el modelo de objetos de la figura 18.

- Criterias: la principal desventaja que tiene HQL es que es dentro del ambiente Java es considerado simplemente como un String, por lo que no es posible detectar errores en su definición. Se debe ejecutar para verificar que esté correcto y que devuelve lo que uno espera. Para solucionar estos problemas, y otros más sutiles, Hibernate también presenta un framework de consulta totalmente orientado a objetos que permite la creación y verificación dinámica de correctitud de la consulta (por lo menos en términos sintácticos). Criterias es traducido a HQL para su ejecución, por lo cual es una de las alternativas más “lentas”, además de que no todo lo que se puede escribir en HQL es posible lograrlo con HQL.

Capítulo V

Resumen: En este capítulo se aborda el estándar Java Data Objects (JDO), el cual puede ser considerado como una extensión del trabajo realizado por ODMG para tratar de establecer un estándar para los repositorios orientados a objetos. La particularidad de JDO es que considera a Java como único lenguaje de programación, mientras que permite múltiples tipos de repositorios.

Introducción

Como se mencionó en el capítulo III, en la sección dedicada al ODMG, muchas de las plataformas para las cuales fue pensada dicha arquitectura no prosperaron más allá de un planteo meramente teórico o conceptual. Sin embargo, una plataforma que no sólo prosperó, sino que terminó derivando en el establecimiento de un nuevo estándar, fue Java.

Inicialmente desarrollado por Sun Microsystems y luego donado en el año 2005 a la fundación Apache, JDO se establece como uno de los primeros estándares para persistencia de objetos con llegada real al mercado.

Si bien se lo puede considerar como un subproducto “indirecto” de ODMG, JDO tiene algunas diferencias sustanciales:

1. Está enfocado exclusivamente en la plataforma Java. Al contrario de ODMG que contemplaba plataformas muy distintas como Smalltalk, Java, C, C++, Delphi, y otras más, JDO solamente está definida para la plataforma Java.
2. Define la posibilidad de persistir la información en múltiples tipos de bases de datos o repositorios. Una de las diferencias más radicales que plantea JDO es independizar la aplicación del “tipo” de repositorio en donde se estén persistiendo los datos. La misma aplicación sin cambios importantes debería

poder almacenar su información en bases de datos relacionales, orientadas a objetos o archivos planos.

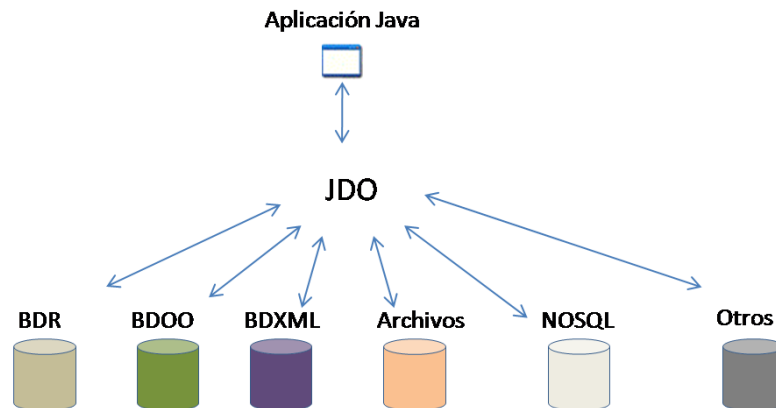


Figura 24: Distintos tipos de repositorios de JDO

3. Tiene como objetivo no crear nuevos tipos. Un defecto importante que tuvo el esquema planteado por ODMG es que requería la utilización de nuevos tipos, como por ejemplo DVector en vez de Vector (clase presente en el JDK estándar de Java). En cambio, JDO no requiere la utilización de nuevas clases.
4. Cero o casi ninguna restricción para persistir clases. Este es un aspecto muy cuidado en este estándar. En todo momento se trata de no imponer restricciones a ser cumplidas por las clases para poder persistir sus instancias. Al contrario de otras alternativas de persistencia, en JDO no es necesario extender una clase²⁸ particular para que las instancias de las clases sean persistentes.
5. No pretende reemplazar a JDBC, sino que es un complemento. En aquellas implementaciones que estén basadas en bases de datos relacionales incluso es posible ejecutar sentencias SQL desde JDO a través del driver de JDBC.

²⁸ Usualmente esta clase suele ser PersistentObject.

Principales elementos

Probablemente una de las mejores características que puedan mencionarse sobre JDO es su simplicidad y pocos elementos que lo componen, lo cual es de gran valor ya que permite un aprendizaje extremadamente rápido y fácil.

PersistenceManagerFactory

Este *Singleton*²⁹ tiene fundamentalmente dos objetivos

1. Permitir la configuración de todos los aspectos requeridos para poder conectarse con el repositorio de información. Parámetros como usuario, dirección ip del servidor, claves, etc. son configuradas a través de esta clase.
2. Actuar como una fábrica de instancias de la clase *PersistenceManager*. Cada vez que se requiera interactuar con el repositorio se deberá solicitar a esta clase una nueva instancia de la clase *PersistenceManager*.

PersistenceManager

Esta clase representa al repositorio de información. Es a través de la misma que se puede acceder a la funcionalidad provista por la implementación de JDO, tal como persistir objetos, recuperar instancias persistentes, realizar consultas.

Aún cuando es central a JDO, su protocolo es relativamente pequeño. Dentro de la funcionalidad más relevante tenemos:

1. **<T> T makePersistent(T pc):** persiste el objeto recibido dentro del marco de este administrador de persistencia. Si la transacción asociada puede realizar el **commit** exitosamente el objeto será almacenado en el repositorio.
2. **void makeTransient(java.lang.Object pc):** remueve la instancia persistente de este administrador de persistencia, volviéndola volátil nuevamente. Esto

²⁹ En toda la aplicación debe existir únicamente una sola instancia de este objeto, ya que es el punto de partida para la creación de las conexiones a la base de datos.

significa que todos los cambios que se realicen a esta instancia no serán reflejados en el repositorio.

3. ***java.lang.Object getObjectById(java.lang.Object oid)***: retorna la instancia correspondiente al **oid** recibido. Esta es la forma más eficiente de recuperar objetos, pero impone la restricción importante de tener que conocer los oids de los objetos que se pretende recuperar. En ciertos casos sin embargo, como por ejemplo para recuperar los objetos raíz³⁰ del modelo es sumamente útil.
4. ***java.lang.Object getId(java.lang.Object pc)***: retorna un objeto que representa la identidad JDO del objeto recibido como parámetro. En otras palabras, este método permite recuperar el identificador de un objeto persistente. Lo notable de este caso es que el identificador no es persistente en sí mismo, por lo que puede ser almacenado y leído fuera de las transacciones. Este hecho lo hace perfecto para ser almacenado por ejemplo en la sesión HTTP de una aplicación Web para poder recuperar luego información de los objetos en forma eficiente.
5. ***<T> Extent<T> getExtent(java.lang.Class<T> persistenceCapableClass, boolean subclasses)***: retorna la extensión³¹ de la clase recibida. El parámetro booleano establece si se deben considerar o no las instancias de las subclasses. Este método usualmente es utilizado al realizar consultas, ya que en JDO se debe especificar un conjunto de instancias “candidatas” sobre las cuales se ejecutará una consulta filtrando por una condición dada.
6. ***void deletePersistent(java.lang.Object pc)***: elimina del repositorio al objeto recibido como parámetro. Esto permite no sólo hacer un borrado lógico (que se logra fácilmente al desasociar a los objetos correspondiente) sino que también elimina físicamente al objeto de la base de datos.

³⁰ Por ejemplo la única instancia de la clase Banco en un sistema bancario, o la única instancia de la clase Aeropuerto en un sistema para aeropuertos, etc.

³¹ La extensión (traducción de “extent”) se define como el conjunto de todas las instancias de una clase dada.

Como puede observarse a partir de la lista anterior, con sólo 7 métodos es posible ya construir una aplicación orientada a objetos que persista su información en un repositorio a través de la API de JDO.

Un comentario aparte merece el primer mensaje de la lista. ¿Cuántas veces se debería enviar este mensaje como parte de una aplicación “real”? Al igual que en el caso de Hibernate (con el método **save()** de la clase Session), este método debería invocarse realmente en pocas ocasiones. De hecho, como se comentó en el capítulo anterior se podría crear una aplicación que realice una sola invocación o incluso yendo un paso más allá se podría tener una aplicación que nunca invoque a este mensaje y que sin embargo almacene objetos a partir de la presencia de la **persistencia por alcance**.

Query

Por más innovadores que podamos ser al explicar su finalidad, es indudable que el objetivo de esta clase es permitir realizar consultas al repositorio para recuperar instancias que satisfagan ciertos criterios o condiciones de búsqueda. Su existencia se justifica ya que el acceso a grandes cantidades de instancias puede ser una operación muy cara en términos de performance, por lo que contar con una facilidad de este tipo suele ser muy valioso.

Sin embargo, hay que utilizarla con mucho juicio, ya que las consultas suelen tender a ser poco “orientadas a objetos” y por lo tanto se termina pagando el costo de “ensuciar” el modelo perdiendo la transparencia e independencia de las cuestiones de persistencia.

Transaction

Como se viene discutiendo en varios capítulos, al persistir los objetos en un repositorio se los está publicando a un ambiente multi-usuario y concurrente. Por este motivo es necesario utilizar transacciones que permitan demarcar las operaciones que se deben realizar en forma atómica. Las instancias de esta clase permiten iniciar (**begin()**) y finalizar las transacciones (exitosamente a través de **commit()** o en caso de fallas a través de **rollback()**).

Un punto importante a notar de la arquitectura de JDO es que cada instancia de `PersistenceManager` tiene asociada exactamente una sola instancia de la clase `Transaction`. Esto tiene como consecuencia que no se puedan crear transacciones anidadas por ejemplo.

Modelo de trabajo

Ahora que conocemos los grandes elementos de JDO podemos tratar de ordenarlos gráficamente para dar indicios de su interacción en una arquitectura típica. La figura 9 muestra tal orden.

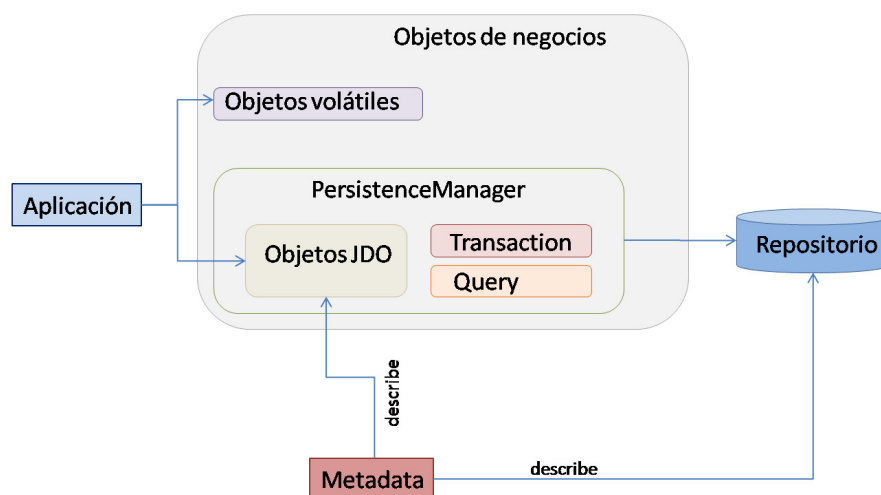


Figura 25: Interacción entre los elementos de JDO.

En la figura 25 la aplicación interactúa con los objetos de negocio. Sería deseable que desde su punto de vista no se pudieran distinguir entre los objetos volátiles (aquellos que desaparecerán cuando se cierre la aplicación) y los objetos persistentes. Para que esto sea posible, un `PersistenceManager` debe recuperar de la base de datos objetos que sean totalmente polimórficos con los objetos que alguna vez se almacenaron.

Como también puede verse en la figura anterior, un punto importante que no mencionamos todavía es el relacionado con la forma de expresar todos los aspectos correspondientes a la persistencia. Si pretendemos contar con las virtudes de la transparencia, es preciso contar con un mecanismo basado en meta-data, y en particular externa basada en XML.

JDO requiere que todos los detalles concernientes a la persistencia se expresen en archivos con extensión “.jdo”³², en los cuales se definen cuales serán las clases que admitirán instancias persistentes. También en ocasiones es necesario declarar las relaciones existentes entre dichas clases. En la sección de ejemplo se presenta un archivo **.jdo** para mostrar su contenido típico.

A continuación veremos todos los pasos involucrados en un “uso típico” de JDO a través de la figura 26.

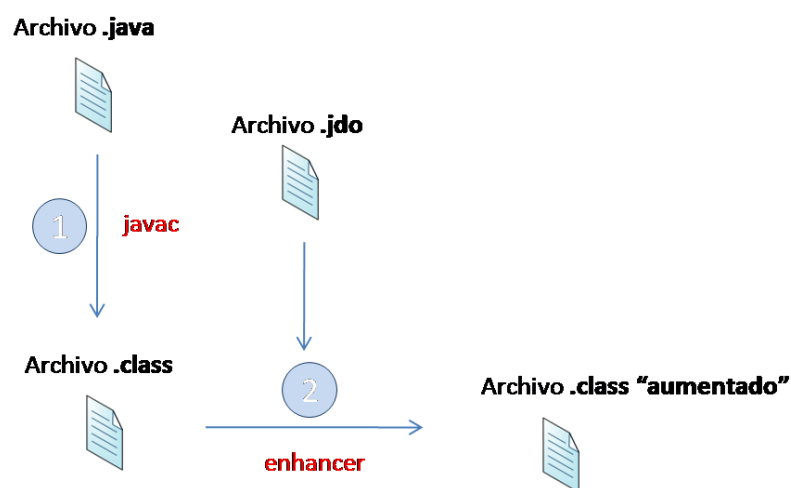


Figura 26: Pasos al utilizar JDO.

- a) El desarrollador crea el archivo **.java** con el código fuente de la clase (para nuestro ejemplo, un archivo de 47 líneas).

³² Es posible declarar un archivo “.jdo” por cada clase, pero se suele recomendar hacer un archivo .jdo por paquete.

- b) A través del compilador³³ (1) se genera el archivo **.class** correspondiente a la clase. Este archivo contiene los bytecodes de Java.
- c) Ahora tomando la información de la meta-data en el archivo **.jdo** y el archivo **.class**, se utiliza un programa conocido como “enhancer” (2) que genera un nuevo archivo **.class** “aumentado” con código requerido para poder persistir las instancias (en el ejemplo, el archivo resultante puede tener unas 440 líneas). Este proceso de “aumento” del código debe ser realizado cada vez que el archivo **.java** original es alterado³⁴.

Ahora bien, ¿qué significa exactamente que los objetos recuperados sean polimórficos con los que se almacenaron? Como se vio más arriba, cuando se persiste con JDO es necesario “aumentar” las clases. Esto significa que el código que se ejecuta en nuestras clases no es precisamente el mismo código que nosotros escribimos, sino que es bastante “parecido”. El proceso de aumento es necesario para que la implementación de JDO pueda incorporar el control de transacciones y otras optimizaciones para recuperar los objetos (en especial cuando se trabaja con colecciones). El punto importante a mencionar aquí es que cuando se recuperan los objetos de la base de datos, éstos usualmente no pertenecen a la nuestra clase, sino a una implementación propia de JDO que tiene mejoras, pero que sigue siendo totalmente polimórfica con nuestra clase. En el capítulo dedicado a los patrones de diseño seguiremos trabajando sobre este punto.

Consultas

Toda solución de persistencia es inútil si no permite luego recuperar la información previamente almacenada.

³³ A estas alturas es poco probable que se utilice directamente el compilador. Probablemente se utilice un IDE como Eclipse.

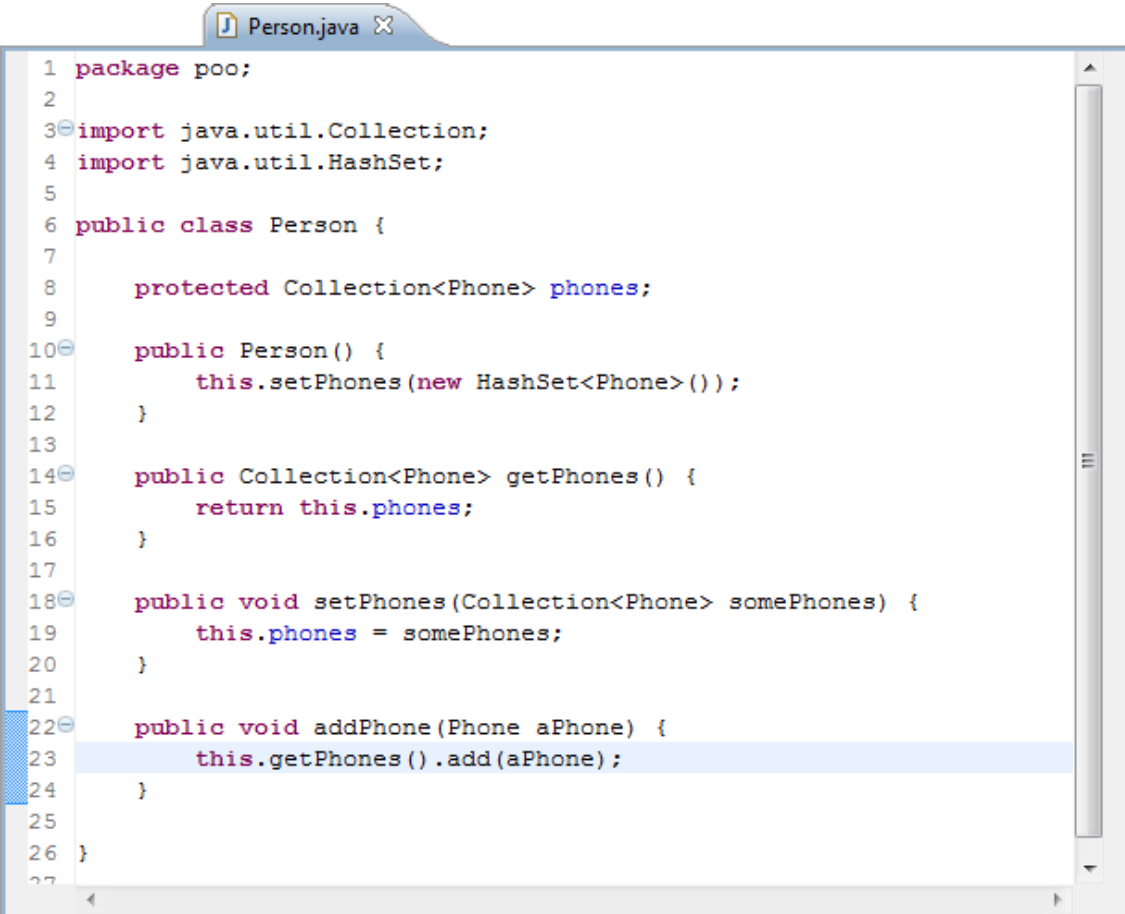
³⁴ Para ser completamente sincero, este proceso de “enhancement” es sumamente tedioso, pero nuevamente al utilizar un IDE con los plugins adecuados esta tarea puede realizarse en forma automática y transparente.

Con JDO existen dos maneras de acceder a la información. Una simple, orientada a objetos pero ineficiente; y por otro lado la otra manera menos orientada a objetos pero sumamente eficiente.

Iteradores

Como vimos anteriormente, un `PersistenceManager` permite recuperar instancias individualmente a través de método `getObjectById()`. Para poder utilizar esta alternativa de consulta, basta con poder recuperar un objeto previamente persistido e iterar sus colecciones en búsqueda de objetos. Como lo que se pretende es mantener siempre la transparencia, la única manera permitida es recorrer dichas colecciones en base a sus iteradores.

La figura 27 recuerda el código de la clase `Persona`, que mantiene una colección de sus teléfonos.



```
1 package poo;
2
3 import java.util.Collection;
4 import java.util.HashSet;
5
6 public class Person {
7
8     protected Collection<Phone> phones;
9
10    public Person() {
11        this.setPhones(new HashSet<Phone>());
12    }
13
14    public Collection<Phone> getPhones() {
15        return this.phones;
16    }
17
18    public void setPhones(Collection<Phone> somePhones) {
19        this.phones = somePhones;
20    }
21
22    public void addPhone(Phone aPhone) {
23        this.getPhones().add(aPhone);
24    }
25
26 }
27
```

Figura 27: Código fuente de una clase que mantiene una colección.

A esta clase ahora deberíamos agregar un nuevo método que permita buscar un teléfono por su número. La figura 28 muestra una alternativa.

```
21 public Phone findPhoneByNumber(String aNumber) {  
22     Iterator<Phone> anIterator = this.getPhones().iterator()  
23     Phone aPhone = null;  
24     Phone currentPhone = null;  
25  
26     while (anIterator.hasNext() && aPhone == null) {  
27         currentPhone = anIterator.next();  
28         if (currentPhone.getNumber().equals(aNumber)) {  
29             aPhone = currentPhone;  
30         }  
31     }  
32  
33     return aPhone;  
34 }
```

Figura 28: iteración para recuperar un teléfono por su número.

Como puede verse en las figuras 27 y 28, no existe código de “bases de datos”, lo cual aporta a la transparencia deseada. Cabe aclarar que la ejecución de este método debe realizarse dentro de un marco de transacciones. El hecho de que no se muestren aquí se debe a que probablemente las transacciones hayan sido abiertas en otras capas antes de invocar a estos objetos.

El código anterior puede considerarse “puro” desde el punto de vista orientado a objetos, sin embargo si no se toman recaudos, su ejecución podría resultar muy lenta e ineficiente para grandes colecciones.

JDOQL

Esta alternativa explota la funcionalidad de consultas provista por JDO. Esta facilidad, denominada JDOQL toma prestada la forma de OQL y la adapta para su utilización con Java a través de la clase Query y sus clases asociadas.

JDOQL a su vez presenta dos alternativas:

1. Forma declarativa

Esta forma permite al desarrollador ir construyendo programáticamente una consulta, como un objeto al que se puede ir agregando sus componentes. Para resolver la búsqueda del teléfono por su número la consulta sería la que se muestra en la siguiente figura.

```
4 Query q = pm.newQuery(org.jpox.Person.class,  
5     "lastName == \"Jones\" && age < age_limit");  
6 q.declareParameters("double age_limit");  
7 List results = (List)q.execute(20.0);  
8
```

Figura 29: Consulta declarativa.

2. Forma como String

La forma como String permite crear una consulta pasándole un solo String que contiene la definición de la consulta. La misma consulta de la figura 29 se muestra como String en la figura 30

```
3 Query q = pm  
4     .newQuery("SELECT FROM org.jpox.Person " +  
5         "WHERE lastName == \"Jones\" " +  
6         "&& age < :age_limit " +  
7         "PARAMETERS double age_limit");  
8  
9 List results = (List) q.execute(20.0);
```

Figura 30: Consulta como String.

En ambos casos se está tratando de recuperar una instancia de la clase **org.jpox.Person** cuyo apellido sea “**Jones**” y su edad sea menor que el límite pasado como parámetro³⁵.

Como se puede observar tanto en la forma declarativa como la basada en Strings, el resultado de una consulta es una lista de instancias de la clase solicitada (eventualmente

³⁵ Notar como se deben declarar al igual que en Java cada uno de los parámetros utilizados.

podrían incluirse instancias de alguna subclase). Si la consulta no obtuviera resultado alguno, aún se seguiría retornando una lista, vacía, pero lista al final.

La estructura completa de una consulta JDOQL como String es la siguiente:

```
① SELECT [UNIQUE] [<result>] [INTO <result-class>]
② [FROM <candidate-class> [EXCLUDE SUBCLASSES]]
③ [WHERE <filter>]
④ [VARIABLES <variable declarations>]
⑤ [PARAMETERS <parameter declarations>]
⑥ [<import declarations>]
⑦ [GROUP BY <grouping>]
⑧ [ORDER BY <ordering>]
⑨ [RANGE <start>, <end>]
```

Figura 31: Componentes de una consulta JDOQL.

Como puede observarse en la figura anterior, la estructura de la consulta es similar a la de OQL y/o SQL, con las siguientes particularidades:

- 1) Al realizar una proyección de la estructura de las instancias, en vez de devolver atributos sueltos es posible inicializar instancias de la clase **<result-class>** cargándolas con dichos atributos.
- 2) Al igual que en el caso de OQL, la cláusula **from** debe completarse con el nombre de la clase candidata a ser consultada³⁶. Eventualmente pueden excluirse las instancias de sus subclases del conjunto consultado.
- 3) Los filtros de las instancias se expresan en base a sus atributos (posiblemente también navegando las relaciones a través del operador “.”). En OQL se planteaba la posibilidad de enviar mensajes a las instancias, lo cual sería lo correcto, pero debido a la imposibilidad de marcar a un método como “no mutador”³⁷, esto no está permitido en JDOQL.

³⁶ Recordar que desde la teoría de objetos se establece que una de las responsabilidades de las clases es conocer y mantener al conjunto de sus instancias.

³⁷ Un método no mutador es aquel cuya ejecución no altera el objeto sobre el cual se ha ejecutado. Un *getter* escrito correctamente debería ser no-mutador, mientras que todo *setter* debería cambiar al objeto sobre el que se ejecuta.

- 4) Esta sección permite declarar variables para referenciar a las instancias dentro de la consulta.
- 5) JDOQL requiere la declaración con su tipo de cada uno de los parámetros que se utilicen como parte de una consulta. La declaración se realiza de la misma manera que en Java, es decir **tipo nombreVariable**.
- 6) Si dentro de la consulta se utilizarán tipos no conocidos por Java, en esta sección se los debería declarar, nuevamente al estilo Java tradicional.

Ejemplos

En esta sección se presentarán algunos ejemplos simples para demostrar las posibilidades de JDOQL.

La figura 32 muestra la utilización de parámetros e importación de tipos.

```
3 Query query = pm.newQuery("SELECT FROM org.jpox.samples.store.Product "
4     + "WHERE endDate > best_before_limit "
5     + "PARAMETERS Date best_before_limit "
6     + "import java.util.Date ORDER BY endDate DESC");
7
8 List results = (List) query.execute(my_date_limit);
```

Figura 32: parámetros y tipos en JDOQL.

La figura 33 muestra la ejecución de una consulta creada programáticamente que recibe múltiples parámetros.

```
7 Query query = pm.newQuery(Person.class);
8 query.declareParameters("int age1, int age2");
9 query.setFilter("this.age >= age1 && this.age <= age2");
10 Collection result = (Collection) query.execute(new Integer(20),
11     new Integer(60));
12
```

Figura 33: Múltiples parámetros en una consulta.

La siguiente consulta muestra la utilización de funciones de agregación (max()) y la utilización de la cláusula “having”.

```

7      Query q = pm.newQuery(Bid.class, "bidder.user.userName==name");
8      q.setResult ("max(bidAmount), item.description");
9      q.setGrouping ("item having max(bidAmount) > 10.00");
10     q.declareParameters ("String name");
11     Collection results = (Collection) q.execute("Sophie");
12

```

Figura 34: Having y max() en una consulta JDOQL.

La figura 35 muestra dos características importantes de JDOQL, la posibilidad de declarar variables y su uso junto con la posibilidad de ejecutar métodos sobre los objetos (notar el mensaje **contains()**). Como se mencionó en párrafos anteriores, no es posible ejecutar cualquier método, sino sólo aquellos que no cambian al objeto receptor. En el caso de método **contains()** no hay forma de asegurar que no cambia la colección a la que es enviado, pero como es una implementación del proveedor de la base de datos (es decir que no fue desarrollado por el usuario) se confía en que es realmente “no-mutador”.

```

7      Query q = pm.newQuery(Person.class);
8      q.declareVariables("Person child");
9      q.setFilter("this.children.contains(child) && child.address.city== \"London\"");
10     Collection results = (Collection) q.execute();
11

```

Figura 35: Ejecución de métodos.

Las operaciones C.R.U.D. y el PersistenceManager

En el capítulo I se mencionó hacia el final el tema de las famosas operaciones C.R.U.D³⁸. Ahora que contamos con conocimientos acerca de las cuestiones básicas de la persistencia de objetos, e incluso conocemos tanto un producto de mapeo objeto/relacional como Hibernate y un estándar de persistencia como JDO, conviene

³⁸ Por sus siglas en inglés C (Create), R (Retrieve), U (Update), D (Delete).

volver a retomar el tema las 4 operaciones y presentar una alternativa de como debería ser su implementación a partir del hecho de contar con persistencia por alcance.

Vayamos por parte entonces, comenzando por la primera operación, es decir **Create**.

- La operación **Create** tiene como objetivo final la inserción de la información en el ambiente persistente, es decir la base de datos. Ahora bien, en realidad tenemos un escenario confuso aquí ya que por un lado tenemos las instancias de la clase **PersistenceManager** (en J.D.O.), cuyo protocolo incluye el mensaje *makePersistent()*, el cual permite almacenar las instancias volátiles convirtiéndolas en persistentes; por otro lado, basándose en buenas prácticas de diseño orientado a objetos, debería ser el mismo modelo el que decida e implemente la creación de las diferentes instancias. Entonces la pregunta que naturalmente surge es ¿cómo utilizar las facilidades del **PersistenceManager** a la vez que se mantiene la transparencia e independencia del modelo de dominio?

Analicemos pues 3 escenarios posibles, los cuales van variando en cuantas veces se invoca el mensaje *makePersistent()* de la clase **PersistenceManager**:

- N veces: en este escenario, el más inmediato, cada vez que se debe crear una nueva instancia de alguna clase de dominio, alguna clase importante del dominio creará la instancia requerida, pero no la persistirá. Para que la nueva instancia se persista en algún momento se deberá invocar explícitamente al método *makePersistent()*. La principal desventaja de este escenario es que aún cuando exista el estándar J.D.O., al cambiar a Hibernate por ejemplo habría que cambiar todas las llamadas al *makePersistent()* por el mensaje *save()* de la clase **Session**.
- 1 vez: un escenario mucho mejor es basarse en la persistencia por alcance. A partir de este punto sabemos que si tomamos una instancia ya persistente y la vinculamos con una instancia volátil, esta última deberá ser a su vez persistente para que se conserven las propiedades A.C.I.D. Entonces, analizando en detalle este escenario podemos ver que solamente se requiere un solo llamado al *makePersistent()* o *save()* en toda la aplicación. Este mensaje se debe enviar solamente para persistir alguna instancia distinguida del dominio, a partir del cual se puede

acceder a todo el resto del modelo. En un dominio como el bancario bien podría ser la única instancia de la clase **Banco**.

- 0 (cero) veces: este escenario es simplemente una extensión del caso anterior en donde nuestra aplicación asume en forma segura que ya existe la instancia de la clase distinguida. ¿Pero es real que nuestra aplicación puede asumir tal circunstancia? En realidad es muy viable es escenario ya que basta con poner la restricción de que antes de que se ejecute nuestra aplicación se haya corrido otra aplicación, usualmente un instalador, que cargue todos los objetos requeridos para la ejecución. Si se piensa en detalle, ahora nuestra aplicación no tendrá ni una sola llamada a *makePersistent()* o *save()*, solamente se recuperarán las instancias persistentes, luego se las vinculará con las no persistentes y al final estas últimas se persistirán automáticamente.

En conclusión, ahora podemos argumentar que evidentemente no es necesario que nuestra aplicación o su modelo de objeto subyacente tenga operaciones *Create* distinguidas o especiales.

- Por su parte la operación *Update* tiene como fin el actualizar la información de una instancia ya persistente. Esta operación es probablemente la que menos sentido tiene de existir en forma distinguida, es decir como una operación por separado de las clases que se supone que debe modificar. Es usual ver diseños en los cuales se cuenta con ciertas clases dedicadas a manejar los detalles de persistencia de otras, por ejemplo para la clase **Person** existe la clase **PersonDAO**³⁹. Es común ver entonces dentro del protocolo de la clase **PersonDAO** mensajes como

```
public boolean update(long aPersonId, String newName, String  
newSurname)
```

Dicho mensaje al ser invocado actualiza el nombre y apellido de una persona cuya identificación se recibe. La pregunta es ¿y dónde quedó el modelo

³⁹ DAO: Data Access Object.

orientado a objetos, en el cual la única responsable de modificar la información de la clase **Person** es justamente la misma **Person**? Noten que usualmente en el diseño de los DAOs se suele poner la lógica que verifica si es válida o no la actualización, en vez de asignarla a la clase que corresponde.

Un enfoque mucho más natural es no tener ninguna operación de actualización distinguida. Solamente basta con poder recuperar un objeto persistente determinado e invocar sobre éste los *setters* apropiados con la información pertinente y listo.

De las 4 operaciones C.R.U.D. se ha visto más arriba que tanto **Create** como **Update** no requieren ser implementadas como operaciones distinguidas. En cambio las dos restantes, **Retrieve** y **Delete** alguna dependencia con el producto de persistencia suelen mostrar.

- La operación que probablemente sea más utilizada es la de recuperación (**Retrieve**). Teniendo esto en mente, y considerando que al utilizar un acceso en objetos “puro” (a través de iteradores) puede acarrear problemas serios de performance, es que se toma la decisión informada de contar con un mecanismo especial para recuperar los objetos. Nuevamente conviene explicar este punto: no es que las soluciones actuales no permitan obtener los objetos persistentes sin tener que escribir código dependiente de la solución de mapeo. El problema es que el código escrito para recuperar en forma de iteradores sufre problemas de performance cuando se trata de colecciones con muchos elementos.

Por lo expuesto más arriba entonces es usual encontrar operaciones de búsqueda y recuperación que son utilizadas como mecanismo anexo y preferido para obtener instancias persistentes, ya sea mediante su *oid* o mediante la ejecución de consultas.

Si bien se utilizan las facilidades de consulta de la clase **PersistenceManager** (o **Session** en el caso de Hibernate) esto no significa que se pueda utilizar libremente este protocolo desde cualquier clase. En particular el objetivo en este punto es ocultar lo más posible su utilización en la capa de modelo. Volviendo a la idea de los DAOs, se puede pensar en tener una solución que siga la misma idea original pero que ahora solamente tenga la operación **R**, de este modo para

dar de alta y/o actualizar se utiliza solamente el protocolo de los objetos de dominio, mientras que para realizar una recuperación se utilizan clases y métodos especializados para obtener la mayor performance posible. En el capítulo dedicado a los patrones de diseño se explicará en detalle el patrón Repository como forma elegante (en realidad relativamente elegante) de realizar búsquedas.

- Finalmente, la operación Delete es la que plantea interrogantes. Si todo lo que debe hacer la aplicación es desvincular objetos entre sí, sin borrarlos nunca de la base de datos, entonces no es necesario tener una operación de borrado distinguida. En este caso, para borrar una instancia de **Client** de la clase **Bank** basta con enviar el siguiente mensaje al banco

public boolean removeClient(Client aClient)

El método lo único que tiene que hacer es acceder a la colección interna de clientes y remover el cliente recibido. Una vez que se haga **commit** de la transacción la asociación entre la instancia de **Bank** y **Client** habrá desaparecido.

Un caso distinto se da cuando además de los pasos anteriores es necesario eliminar físicamente la instancia de la clase **Client**. En este caso es necesario invocar protocolo propio de la clase **PersistenceManager**. Bajo estas circunstancias la aplicación sí debería invocar el protocolo de la clase **PersistenceManager** y por ende perdería algo de su transparencia respecto de los aspectos de persistencia. Sin embargo, no debería en ningún caso ser el modelo el que invoque a dicho protocolo, sino las capas pertenecientes a los servicios⁴⁰ o aplicación.

⁴⁰ Servicios, en el sentido propuesto por S.O.A. (Service Oriented Architecture).

Productos

Cuando se trata de estándares, en gran medida su éxito puede medirse de acuerdo a la cantidad de productos o implementaciones que los adopten. La siguiente tabla⁴¹ presenta un breve resumen de los productos que al momento de publicación.

Nombre	Licencia	Versión del estándar	Repositorio
DataNucleus Access Platform	No comercial	1.0, 2.0, 2.1, 2.2, 3.0	BDR, DB4O, Ldap, Excel, XML, ODF, Google Big table, Amazon S3, Hadoop
JDOInstruments	No comercial	1.0	JDOInstruments
JPOX	No comercial	1.0, 2.0, 2.1	BDR, DB4O
KODO	Comercial	1.0, 2.0	BDR, XML
ObjectDB for Java/JDO	Comercial	1.0, 2.0	ObjectDB
Objectivity	Comercial	1.0	ObjectivityDB
Orient	Comercial	1.0	Orient
TJDO	No comercial	1.0	BDR
Versant	Comercial ⁴²	1.0, 2.0	Versant Object Database
XCalia	Comercial	1.0,2.0	BDR, XML, Versant, BDOO, etc.

⁴¹ Esta tabla fue extraída del sitio de la Fundación Apache dedicado al estándar JDO, y luego fue aumentada con información de otros sitios.

⁴² Versant también cuenta con una versión no comercial muy básica que puede ser descargada gratuitamente,

Como puede observarse en la tabla anterior, no sólo hay una variedad interesante de productos, sino que también hay empresas de renombre soportante y utilizando este estándar (particularmente Google a través de DataNucleus y Objectivity, la cual se utiliza en el CERN como repositorio para sus experimentos de aceleración de partículas).

Capítulo VI

Resumen: el objetivo de este capítulo es hacer un breve recorrido por los patrones de diseño más importantes al momento de persistir información de una aplicación orientada a objetos. Estos patrones no están ligados directamente con una tecnología en particular, sino que pueden ser utilizados por la mayoría de éstas con pequeñas adaptaciones⁴³.

Root object

El primer patrón de diseño que produce grandes cambios al aplicarse, sobre todo en la manera de trabajar y de pensar soluciones bajo el paradigma orientado a objetos con persistencia es el RootObject. Este patrón además establece prácticas que desde el punto de vista resultan naturales, pero que desde el punto de vista del modelo entidades y relaciones podría considerarse “*incorrecto*”.

En términos simples este patrón establece que todo modelo debería contar con puntos bien reconocidos de acceso, que permitan a la aplicación interactuar con todos los objetos relevantes del sistema. Esto tiene implicancias no sólo a nivel de diseño, sino también en la forma en la que se accede a los objetos.

Siguiendo estos lineamientos, por ejemplo si estuviéramos modelando una aplicación para un Video Club, una de las primeras clases en ser identificadas y diseñadas debería ser la clase **VideoClub**, cuya principal responsabilidad sería modelar al sistema que se está desarrollando.

Habitualmente cuando se desarrollan aplicaciones orientadas a objetos se requiere funcionalidad para dar de alta usuarios. Bajo este escenario surge entonces la siguiente

⁴³ Para este capítulo se asume que el lector tiene un conocimiento básico de los patrones aquí presentados.

pregunta: ¿dónde se debe poner la responsabilidad de verificar que sólo se pueda agregar un usuario cuando su documento no exista previamente? (en otras palabras no se permiten repetidos). Podemos analizar tres posibles soluciones a este problema:

- 1) En la definición de la clase **VideoClub** se podría utilizar un conjunto (**Set**) como tipo de la colección de usuarios propios del video club. Luego, sabiendo que los conjuntos verifican la unicidad de sus elementos mediante el envío del mensaje "*equals()*", bastaría con reimplementar en la clase **User** dicho método y estaríamos consiguiendo los resultados esperados.
- 2) La segunda alternativa pasa por incorporar en la lógica de control dentro de la clase **User**. Así, de este modo cada usuario es responsable de verificar si cumple o no con las condiciones para su alta.
- 3) La última alternativa es delegar esta responsabilidad en el objeto que representa al sistema en desarrollo, es decir la clase **VideoClub**. En definitiva, esta clase es responsable de mantener la colección de usuarios, por lo que parece apropiado colocar la lógica de verificación aquí.

De las tres alternativas anteriores, si bien todas son viables técnicamente, algunas son mejores desde el punto de vista de diseño. La segunda alternativa es por lejos la peor, ya que ¿cómo haría un usuario para verificar que su nombre o documento no se repita en una colección de usuarios que en principio no conoce? La primera alternativa funciona, pero se basa mucho en cuestiones de implementación, ¿Qué pasaría si se tuviera que redefinir el método *equals()* para cubrir alguna otra necesidad? En cambio, desde el punto de vista de diseño O.O. la tercera alternativa es la más correcta. Como mencionamos anteriormente, un videoclub es responsable de mantener sus usuarios (incluso en la vida real). ¿Quién mejor que esta clase para administrarlos considerando que en esta clase está definida la colección?

Ahora bien, si vemos este patrón desde el punto de vista del modelo entidades y relaciones, estamos proponiendo que en un modelo de este tipo aparezca una entidad llamada **VideoClub**, la cual finalmente sería traducida a una tabla que contendría únicamente (y siempre) una sola tupla. Claramente es un mal diseño desde el punto de vista del modelo entidades y relaciones, pero en cambio en objetos está muy bien. ¿Dónde está entonces la diferencia? Pues bien, en realidad la diferencia se debe a que en

el modelo orientado a objetos uno modela comportamiento además de “estructura” para almacenar la información, por lo que debe asignar a alguna clase la responsabilidad de albergar dicho comportamiento. En el caso del ejemplo citado más arriba, no hay otro lugar mejor que la clase **VideoClub** misma.

Como punto extra conviene advertir que cuando sean muy grandes los sistemas es probable que se requieran varias clases que actúen como raíz para acceder al modelo.

¿Qué cambios produce la aplicación de este patrón? Principalmente el cambio se puede resumir de la siguiente manera: para operar con el modelo persistente se debe recuperar la raíz y de ahí navegar a los objetos requeridos. En código, esto se ve de la siguiente manera:

```
Tx

//recuperamos la raíz

//operamos con el modelo
//recuperado y los nuevos
objetos

//tx
```

Como puede verse en el pseudo-código, toda la operatoria con el modelo se lleva a cabo dentro de un marco transaccional, de modo de poder persistir automáticamente los cambios realizados. Luego, toda operatoria siempre arranca con la recuperación del objeto (u objetos) raíz a partir del cual se podrá ir navegando a los demás objetos requeridos⁴⁴.

El requisito que aparece entonces es poder recuperar de manera eficiente el objeto raíz. Este es relativamente fácil y sumamente eficiente. Basta con almacenar el OID del objeto raíz y luego utilizar la funcionalidad provista por la clase **PersistenceManager** (en el caso de JDO) o **Session** (en el caso de Hibernate). Los métodos en cuestión son

- *public Object getObjectById(), de la clase PersistenceManager*
- *public Object load(), de la clase Session*

⁴⁴ Recordar que si por ejemplo agregamos un nuevo objeto a una colección del objeto raíz, al contar con persistencia por alcance, el nuevo objeto se persistirá automáticamente.

En ambos casos basta con saber el OID del objeto raíz para poder recuperarlo de manera eficiente. Notar que este punto es hasta trivial, ya que si se está desarrollando una aplicación web, se podría almacenar en el contexto de la aplicación un atributo que referencie al identificador del objeto raíz⁴⁵.

Proxy

En “el” libro de patrones, el objetivo del patrón de diseño Proxy se define como

Provide a surrogate or placeholder for another object to control access to it.

Una traducción podría ser

Proveer un sustituto de un objeto para controlar el acceso a este.

Con esta traducción no queda realmente claro cuál puede ser el aporte de este patrón para los problemas relacionados con la persistencia de objetos, y sin embargo es sumamente útil.

Un punto importante que debe ser considerado es el costo que conlleva la recuperación de los objetos de la base de datos. Si a esto se suma el hecho de que al recuperar un objeto en principio no se conoce si serán requeridos o no sus colaboradores, entonces cualquier estrategia que ayude a recuperar solamente lo necesario empieza a cobrar sentido. Y es justamente esto lo que aporta el patrón Proxy: la idea detrás de este patrón es mejorar el rendimiento recuperando lo mínimo necesario y reemplazando los demás colaboradores por proxies, los cuales son más “baratos” de construir y devolver. Esta técnica se conoce como **lazy loading**.

⁴⁵ En el último capítulo se ahondará sobre este punto a través de la aplicación web desarrollada como ejemplo.

Lazy loading

Para poder explicar el funcionamiento de esta técnica, supongamos que contamos con el siguiente diseño

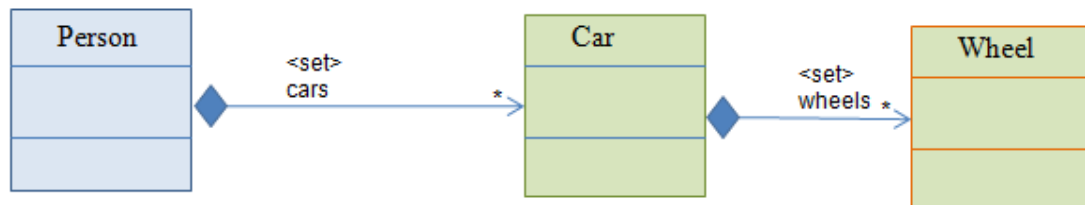


Figura 36: Una persona con una colección de autos, los cuales a su vez están compuestos de ruedas.

Según la figura anterior, cada vez que se persiste una persona, se guarda en cascada su colección de autos asociada. Este comportamiento es el esperado y funciona perfectamente. En cambio cuando se recupera la instancia de la clase **Person**, ¿se deben recuperar todos sus autos, o sus ruedas? La técnica de **lazy loading** responde a esta pregunta con un rotundo NO.

Al recuperar una instancia de la clase **Person**, la solución de mapeo tiene varias alternativas para mejorar la performance:

- 1) No realizar ninguna optimización, por lo que recuperará la persona completa, es decir tanto los atributos propios de la persona como el resto de sus colaboradores, y en cascada seguirá recuperando toda la información de estos objetos y sus colaboradores. Como puede observarse esto podría concluir con la recuperación a memoria de todo el modelo persistido, por lo que no es una situación deseable. La siguiente imagen muestra el conjunto de instancias que sería recuperado al utilizar esta alternativa.

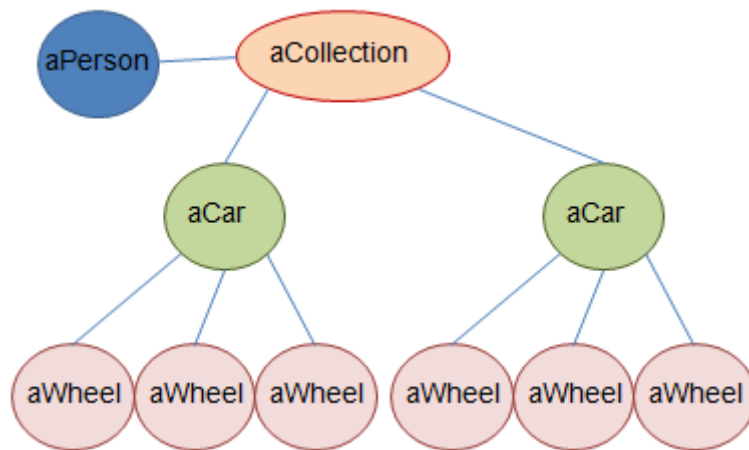


Figura 37: Todos los objetos recuperados con la primera estrategia.

- 2) La primera optimización que puede realizar es recuperar la instancia de la persona, todos sus atributos simples (boolean, strings, etc) y todos los colaboradores. Pero en vez de recuperar en cascada a dichos colaboradores, solamente recupera los atributos básicos. Llegado el caso de necesitar de esta información, bajo demanda se irá recuperando dicha información. Con este simple esquema se logran muchos beneficios en términos de performance y control de la concurrencia⁴⁶. La siguiente figura muestra nuevamente el diagrama de instancias recuperadas con esta alternativa.

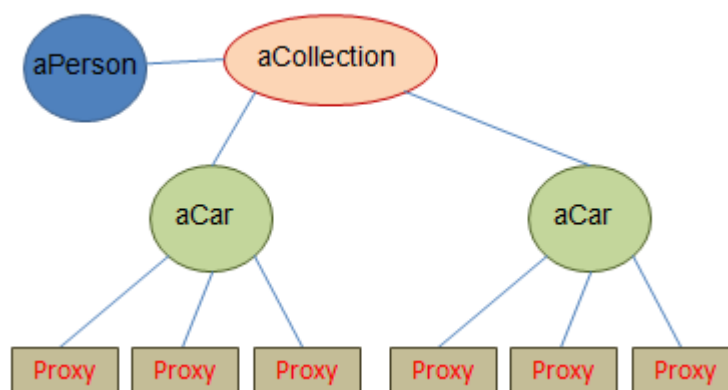


Figura 38: Recuperación del primer nivel de objetos.

⁴⁶ Con la utilización de lazy loading se facilita el control de los cambios concurrentes ya que toda la información que se lleva a memoria en forma de proxy, y que cambia en la base de datos, no se debe controlar si el usuario la está alterando o no.

- 3) El problema de la alternativa anterior es que la instancia de la persona se está configurando con una colección que contendrá muchos objetos proxy que representan a los autos. Una alternativa más eficiente es directamente recuperar una persona y configurarla con un proxy a la colección de autos, de modo de no recuperar ni la más mínima información de los mismos. En cuando a velocidad de acceso y recuperación, esta es la mejor alternativa. La figura siguiente muestra el diagrama de instancias recuperadas.



Figura 39: Recuperación de Proxy.

Como es de esperar, cada una de las 3 alternativas anteriores tiene también sus desventajas. Por ejemplo, considerando las alternativas 2 y 3 cada vez que requiera información de un teléfono se deberá cambiar el proxy por la información propia del objeto, por lo que se incurrirá en un acceso a la base de datos. Analizando un caso de una persona con 10.000 teléfonos, para recuperar toda la información se deberán realizar 10.001 accesos lo cual es terriblemente costoso.

Si recordamos las características de transparencia que solicitábamos en los primeros capítulos, hay que establecer que la selección de cualquiera de las alternativas anteriores debería ser totalmente transparente para el desarrollador. La siguiente figura muestra la configuración de un archivo de mapeo de Hibernate que define que se utilice la alternativa 3 para recuperar la colección de usuarios (instancias de la clase `zinbig.item.model.users.User`) de una clase dada.


```
<bag name="users" lazy="extra" cascade="all">
  <key column="oid_tracker" not-null="false" />
  <one-to-many class="zinbig.item.model.users.User" />
</bag>
```

Figura 40: definición de un proxy en un archivo de mapeo.

Como puede observarse, definir la alternativa de lazy loading que se debe utilizar es relativamente simple y transparente.

Decorator

Nuevamente remitiéndonos al libro de patrones, el objetivo del patrón Decorator es

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Una posible traducción al castellano podría ser

Agregar dinámicamente a un objeto responsabilidades adicionales. Un Decorator provee una alternativa flexible a la sub-clasificación como mecanismo de extensión de funcionalidad.

Una de las tantas oportunidades de aplicación que posee este patrón está relacionada con la demarcación de transacciones dentro de una aplicación orientada a objetos.

A lo largo de los capítulos anteriores hemos ido discutiendo en varias oportunidades la necesidad de mantener la mayor transparencia posible respecto de los detalles de la persistencia, siempre enfocándonos en la capa del modelo. En este momento debería ser

claro que no hay razón para “ensuciar” el código de nuestro modelo con líneas de código relacionadas con transacciones, llamados al *persistence manager* o ejecución de consultas. Podríamos entonces enfocarnos en las otras capas de la aplicación para analizar donde estaría por ejemplo la demarcación de las transacciones.

Supongamos entonces que estamos aplicando los conceptos de SOA⁴⁷, bajo los cuales la funcionalidad de un modelo se expone mediante “servicios”. En particular, cada implementación de un servicio por lo general no mantiene el estado de la conversación, por lo que debe volver a recuperar toda la información desde la base de datos.

Además si recordamos el patrón de diseño RootObject, es probable que también tengamos modelado al sistema en sí, cuya instancia deberá ser recuperada cada vez que se acceda a un servicio para luego interactuar con los objetos pertinentes del modelo. Pero esto implica que el acceso se debe realizar dentro del marco de las transacciones para preservar las propiedades ACID.

El patrón Decorator nos permite agregar a los servicios nuevo comportamiento en forma dinámica, siendo este comportamiento ni más ni menos que la gestión del marco transaccional. La figura 38 nos presenta el esquema de uso de este patrón.

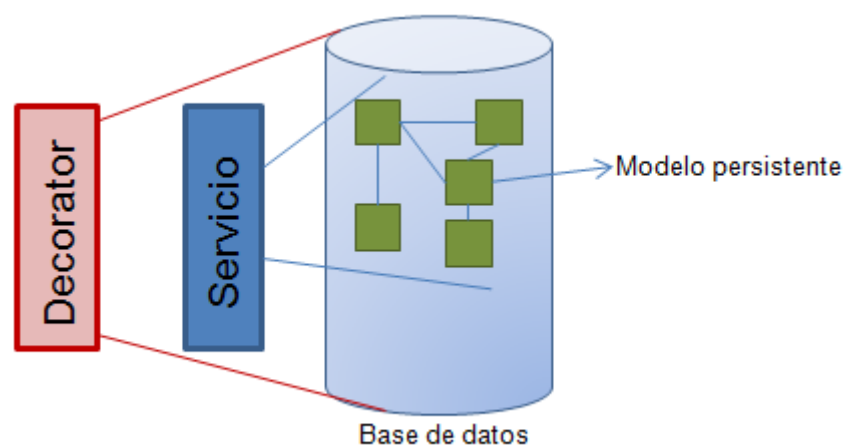
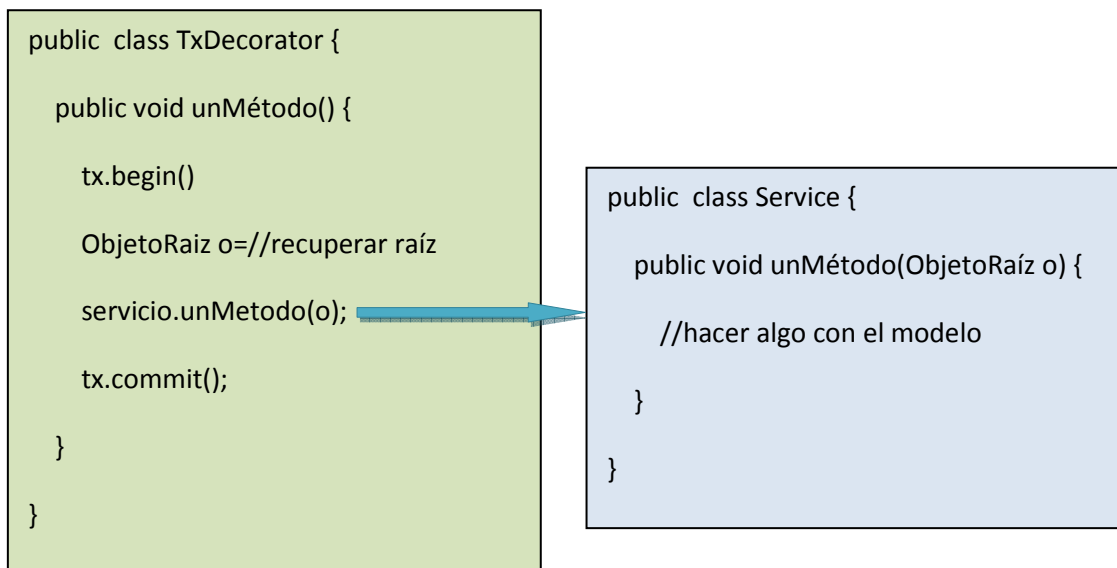


Figura 41: Utilización del patrón de diseño Decorator.

⁴⁷ SOA: Service Oriented Architecture, estilo arquitectónico de creciente aceptación.

A través del uso del patrón Decorator, ahora podemos tener incluso hasta la capa de servicios totalmente transparente respecto de los detalles de persistencia. Esto es singularmente interesante, ya que permite probar la capa del modelo y de servicios primero en forma independiente y luego en forma conjunta, sin necesidad de contar con el ambiente persistente. Más adelante se agrega el **decorator** con la demarcación requerida de transacciones.

Un detalle en el que conviene detenerse es como “hace” el servicio para saber qué objeto raíz recuperar. Si lo recuperásemos dentro del servicio entonces no tendríamos un ambiente transparente, por lo que la única alternativa que nos resta es que se lo pase como parámetro en la invocación que realiza el **decorator**, con lo cual el pseudo-código de un servicio y de su **decorator** tendrían las siguientes formas:



Cabe destacar que esta no es la única forma de resolver la cuestión de la demarcación de las transacciones en forma transparente. De hecho en el capítulo IV lo haremos mediante inversión de control y Spring.

Repository pattern

Cuando se lee material acerca de DDD es usual encontrar referencias a un patrón llamado Repositorio. Dicho patrón tiene como objetivo simular una colección clásica de objetos en memoria, pero que en realidad tiene sus elementos en la base de datos.

Si uno analiza cuáles son los mensajes más utilizados de una colección, probablemente sean solamente 3: agregar, remover y buscar.

Entonces volviendo al planteo previo, este patrón indica que uno debería construir una clase que entienda este protocolo de 3 mensajes básicos, y que cuando uno agregue un elemento a la colección lo inserte en la base de datos. Al remover un objeto de la “colección”, ésta hará lo propio con la base de datos. Finalmente, el mensaje buscar() recorrerá todas las instancias referenciadas por la colección y devolverá aquellos que cumplan con alguna condición definida.

¿Pero cuando se utiliza una solución como JDO o un mapeador como Hibernate, en donde se hace un uso extenso de implementaciones de los patrones **Decorator** y **Proxy**, qué sentido tiene implementar estos métodos, si justamente la solución de mapeo ya los está reimplementando por nosotros?

En realidad, hay una pequeña diferencia que amerita la creación de repositorios. Un producto como Hibernate, para ser ampliamente aceptado debe replicar la interfaz más genérica posible, por lo que termina implementando la interface Collection. Dicha interface no tiene un protocolo para realizar búsquedas, sino que lo único remotamente parecido que ofrece es un iterador, que permite recorrer los elementos en forma externa y averiguar si cada uno cumple o no con una determinada condición.

Si uno quisiera optimizar la búsqueda de objetos referenciados por la colección podría armar un criterio de búsqueda y pasarlo a la colección para que aplique este criterio sobre todos los objetos que referencia. Esto último representa la única razón por la que conviene pensar en este patrón.

En definitiva entonces, este patrón permite crear objetos que se “parecen” a las colecciones, pero que en realidad lo único que hacen es recibir criterios de búsqueda para filtrar sus elementos en forma eficiente. Recordar que a partir del hecho de contar

con persistencia con alcance, los mensajes *add()* y *remove()* dejan de tener sentido dentro del protocolo del repositorio.

Una dificultad que plantea este patrón es que se debe decidir cuán genérico se quiere diseñar e implementar al repositorio. A lo que nos referimos es que si el repositorio será utilizado por diferentes clases para almacenar instancias de muchas clases diferentes será necesario contar con un diseño de filtros y criterios de búsquedas que pueda ser aplicado a los repositorios, sin importar a qué clase corresponden los objetos de la colección. Esto es muchas veces prohibitivo en términos de tiempo, por lo que se termina diseñando repositorios especializados que tienen un protocolo ajustado a los objetos que contienen. La siguiente figura muestra el protocolo de un repositorio que se utiliza para acceder en forma eficiente a las instancias de una clase Proyecto:

ProjectRepository
<pre>public boolean containsProjectWithName(String aName); public Project findProjectByName(String aName); public Collection<Project> getProjects();</pre>

Figura 42: Repositorio de Proyectos.

Como puede observarse en el protocolo del repositorio de la figura 39, es muy concreto y exclusivo para instancias de la clase Project. Si bien esto dista mucho de ser un buen diseño desde el punto de vista de la orientación a objetos, su implementación es simple y directa.

El último punto a comentar es respecto a cómo es la implementación de cada uno de los métodos. En realidad, la única ventaja que se busca es la mejora en performance, por lo cual es usual implementar estos métodos utilizando los lenguajes de consulta provistos por los mapeadores, por ejemplo HQL. Entonces un repositorio no es más que una máscara que en realidad permite ocultar el envío de una consulta a la base de datos.

La figura 40 muestra una implementación posible que utiliza consultas nombradas para devolver un proyecto con un nombre dado.

```
public Project findProjectWithName(Tracker aTracker, String aName)
    throws ProjectUnknownException {

    Query aQuery = this.getNamedQuery("projectByNameQuery");

    aQuery.setParameter("aName", aName);
    aQuery.setMaxResults(1);

    Project result = (Project) aQuery.uniqueResult();

    if (result == null) {

        throw new ProjectUnknownException();

    }
    return result;
}
```

Figura 43: Implementación de un método del repositorio con HQL.

Con la presentación de este patrón podemos dar por cerrado entonces el tema de las 4 operaciones CRUD, argumentando que la única operación que requiere de un trato “especial” es la recuperación (R), que puede ser implementada como un repositorio.

DTO

Como vimos en los capítulos anteriores, particularmente en la figura 5, la demarcación de transacciones trae algunas consecuencias extrañas, por ejemplo el hecho de declarar una variable dentro de un método y luego de cerrar la transacción no poder acceder más al valor de dicha variable. Esto plantea una interrogante importante, ¿cómo se debe hacer entonces “devolver” la información administrada por el sistema? En otras palabras, tenemos un sistema que almacena información, pero que al momento de

recuperar la misma, dentro del marco transaccional, no nos permite leer dicha información más allá de la transacción.

Obviamente una primer alternativa podría ser leer y operar con toda la información dentro de las transacciones, prolongando éstas todo el tiempo que sea necesario como para que el usuario obtenga la funcionalidad deseada. Debería resultar claro que esta alternativa dista de ser la mejor posible, incluso presentando algunas importantes desventajas:

- Se mezclan las capas, ya que para permitir que el usuario interactúe con la información es necesario presentar una interfaz gráfica. Por lo tanto estamos hablando de que la lógica de la aplicación debe abrir una transacción, invocar al modelo y luego abrir una interfaz, dejando abierta la transacción durante todo el tiempo que el usuario desee⁴⁸.
- Como vimos en capítulos previos, a medida que se van recuperando los objetos de la base de datos a memoria, también se va incrementando la probabilidad de tener algún conflicto de versiones. En otras palabras, cuanta más cantidad de objetos se recupera, más probable es que alguno de ellos al intentar ser actualizado resulte en una excepción debido a que no se cuenta ya con la última versión disponible del objetos.
- Si consideramos que todo lo que se modifique en memoria recién será persistido cuando se cierren las transacciones, entonces podemos pensar que hasta que el usuario no cierre la última interfaz gráfica no se persistirá la información que ha estado modificando desde el inicio del día. Sinceramente no me gustaría ser el que tenga que decirle al usuario que todo lo que hizo en un día no se pudo almacenar y lo tiene que volver a hacer.

Evidentemente debe existir alguna otra alternativa para poder recuperar la información de la base de datos, presentarla al usuario, que éste la modifique y que finalmente se

⁴⁸ Notar que el usuario tarde en general entre algunos segundos y varios minutos para realizar su operación, mientras que las transacciones deberían cerrarse en pocos milisegundos.

persistan los cambios sin los inconvenientes presentados más arriba. Una de tales soluciones es el DTO, por sus siglas en inglés de Data Transfer Object.

Como su nombre lo indica, estos objetos son encargados solamente de transferir información de un lado a otro, o mejor dicho, de una capa a otra. La idea básica, explicada en varios pasos, es la siguiente:

1. Abrimos la transacción
2. Recuperamos los objetos del modelo que se requieran. Esto lo hacemos mediante su OID o utilizando una consulta.
3. Invocamos los métodos que sean necesarios, indicando a los objetos que interactúen.
4. Antes de cerrar la transacción copiamos toda la información pertinente para poder mostrarla luego.
5. Cerramos la transacción lo más rápido posible.

Si bien son simples y conocidos ya los pasos citados más arriba, el punto importante está ahora en paso número 4, en el cual se especifica que se debe “copiar” la información. En este punto es importante saber que si copiamos la información de un objeto A a un nuevo objeto B, siendo A persistente, no necesariamente el objeto B debe ser persistente, razón por la cual este esquema realmente sirve: de ahora en adelante nos basta con copiar la información que se quiere presentar al usuario en otros objetos, distintos de los objetos persistentes, y luego de cerrar la transacción se podrá seguir leyendo dicha información.

Como en otras ocasiones, es necesario aclarar un par de cuestiones antes de utilizar el DTO en forma indiscriminada:

- Clase de los nuevos objetos en donde se copia la información: al copiar la información de una instancia persistente de la clase Persona a otro objeto, surge naturalmente la pregunta acerca de la clase a la que debería pertenecer la nueva instancia. Una alternativa es utilizar una nueva instancia de la misma clase, es decir Persona. El problema con esta alternativa es que el desarrollador que

invoca un método en el modelo y recibe como respuesta una instancia del “modelo” no tiene forma de saber si el envío de un mensaje se llevará a cabo en forma correcta o no, simplemente por el hecho de ver el protocolo público del objeto (recordar que el objeto devuelto es de la clase *Persona*, por lo que se le pueden enviar todos los mensajes). Ahora bien, si tuviéramos un buen diseño, seguramente una instancia de *Persona* debería tener muchos colaboradores para poder realizar sus responsabilidades, así que en última instancia deberíamos cargar tanta información en la copia de la persona que prácticamente convendría devolver directamente la persona y en consecuencia no cerrar la transacción. Una alternativa más liviana en este sentido es crear una nueva clase, que solamente contenga la estructura que permita almacenar información (con sus correspondientes getters y setters), de modo que se cargue la información requerida y los desarrolladores de las demás capas que reciban este objeto no tengan dudas de que solamente se trata de un objeto dedicado a transferir información, y por lo tanto no tiene sentido ni siquiera enviarle otro protocolo.

Para comprender mejor este patrón de diseño conviene acceder al código de la clase **zinbig.item.util.dto.ProjectDTO** del proyecto de ejemplo explicado en el capítulo IV.

- Cantidad de información a copiar: una duda habitual tiene que ver con la cantidad de información que se debe copiar en un DTO. La respuesta a esta pregunta en realidad está dada por los requerimientos del caso de uso que se esté modelando. Cabe destacar que probablemente se utilice un mismo DTO para resolver las necesidades de varios casos de uso distintos, por lo que en algunas ocasiones algunos de los atributos de los DTO no tendrán información debido a que dicha información no es necesaria para el caso de uso en particular.
- Identificación: uno de los usos que suele darse a un DTO es el de representar a un objeto persistente para que la información pueda ser editada por el usuario. Cuando el usuario envíe nuevamente la información al servidor, debería ser muy eficiente poder recuperar la misma instancia persistente por lo que usualmente el DTO suele tener un atributo extra, el OID del objeto persistente. A partir de este hecho entonces es posible recuperar en forma muy rápida los objetos

persistentes, utilizando los mensajes **load()** y **getObjectById()** de Hibernate y JDO respectivamente.

Capítulo VII

Resumen: Este capítulo contiene una detallada explicación de una aplicación de muestra desarrollada con el enfoque de este libro, a fin de que el lector pueda tomar contacto de primera mano con los detalles importantes de la implementación.

Inversión de control

Conocido también como “El principio Hollywood⁴⁹”, la inversión de control plantea que dados una aplicación y un framework, la aplicación no debería invocar al framework directamente, sino al revés, es decir el framework debería invocar cierta funcionalidad que en forma predeterminada la aplicación provee. En forma más general, la inversión de control establece una manera de programación en la que no se debe invocar explícitamente procedimientos, funciones y/o métodos, sino que éstos son llamados a partir de situaciones o eventos que se van sucediendo y que el framework va detectando. Típicamente estos eventos son generados por frameworks concretos, como mapeadores tipo Hibernate o frameworks de desarrollo web. En la siguiente sección tomaremos una de las formas más conocidas de la inversión de control para explicar su funcionamiento.

Inyección de dependencias

Un problema habitual que tienen las aplicaciones orientadas a objetos es que necesitan instanciar objetos. Este punto, si bien puede parecer trivial, en lenguajes como Java significa escribir una sentencia dentro de la clase **Person**

Car aCar=new Car();

⁴⁹ No nos llame, nosotros lo llamaremos.

Si bien es simple la sentencia, uno podría mejorarla a través de la aplicación del principio de sustitución de Liskov, obteniendo

Vehicle aVehicle=new Car();

Como se ve en la sentencia anterior, si bien podemos mejorar (y mucho) la parte de la declaración de la variable, aún seguimos con el problema de dejar en forma explícita el nombre de la clase que se está instanciando (en nuestro caso la clase **Car**). ¿Qué pasaría si se requiriese instanciar objetos de otra clase⁵⁰?

Obviamente existen muchas soluciones posibles, por ejemplo utilizando Reflection. Sin embargo, todas las alternativas suelen pasar por soluciones de carácter muy técnico, no a nivel de diseño.

La inyección de dependencias establece por su parte que los objetos no deberían ser responsables de crear sus colaboradores, sino que deberían “recibirlos”, o para utilizar el mismo vocabulario, a cada objeto se le debería “inyectar” el conjunto de colaboradores.

¿Cuál es la principal ventaja entonces al utilizar inyección de dependencias? Principalmente independencia y transparencia. En nuestro ejemplo, la clase en cuestión debería tener solamente la declaración de la variable *aVehicle*, y sus correspondientes getters y setters. A través de la utilización de la inyección de dependencias, cuando se requiera una instancia de la clase **Person**, ésta ya vendrá configurada con su atributo *aVehicle* asignado a algún colaborador. ¿Cuál será la clase de dicho colaborador? En realidad no hay forma de saberlo, solamente podemos asegurar que será de una clase que pertenezca a la jerarquía de vehículos y nada más, lo cual es sumamente interesante y positivo ya que se refuerza muchísimo el polimorfismo.

¿Ahora bien, como se utiliza “en la práctica” la inyección de dependencias, en un ambiente productivo? La siguiente sección presenta muy brevemente un framework que entre otras cosas permite configurar objetos a través de la inyección de dependencias.

⁵⁰ Obviamente siempre estamos hablando de alguna clase que herede de *Vehicle*.

Spring

Una alternativa para poder instanciar objetos de alguna clase, y que éstos “vengan” configurados con sus correspondientes colaboradores requiere de al menos dos pasos:

- Una especificación que permita establecer qué atributos deben ser inyectados y a qué clase deberían pertenecer las instancias inyectadas. Esto generalmente se puede lograr mediante un archivo de propiedades (.properties) o XML.
- Una manera de interponerse en la creación efectiva de los nuevos objetos, para poder inyectar apropiadamente los colaboradores antes de que el nuevo objeto esté disponible para su utilización. Este punto puede lograrse con cierto esfuerzo utilizando **class loaders** en Java.

Otra alternativa más elegante es utilizar un framework que realice todas estas tareas de manera transparente y poco intrusiva. Aquí es entonces en donde entra a jugar Spring, el cual nos permite realizar varias tareas complejas de manera sencilla:

- Definir los colaboradores que deberán ser inyectados en algunos objetos importantes de la aplicación, para que cuando sean obtenidos a partir de Spring, éstos ya estén apropiadamente configurados, de forma totalmente transparente.
- Definir detalles de conectividad, de modo que la aplicación sea independiente de estos detalles.
- Decidir qué implementación de diferentes estrategias de resolución de consultas es la más adecuada para resolver la búsqueda de objetos y pasar de una a otra de manera simple, solamente configurando el archivo de propiedades.
- Publicar la funcionalidad del modelo a través de la definición de “servicios”⁵¹.
- “decorar” los servicios, definiendo en forma declarativa el manejo de transacciones que debe aplicarse a cada método invocado del servicio.

⁵¹ Implementaciones del patrón de diseño Facade.

- Definir una capa que “adapta” excepciones técnicas, como problemas relacionados con la base de datos o de conectividad, para presentarlos como excepciones propias de la aplicación.

Los puntos citados más arriba son solamente algunos de los usos que se puede hacer del framework Spring, pero bastan como para mostrar rápidamente la mejora que se puede lograr a partir de su utilización. Más adelante se presentarán con más detalle algunas secciones importantes del archivo en donde se declaran todos los puntos precedentes.

Wicket

De los tantos frameworks para desarrollo web que se pueden encontrar hoy en día para desarrollar aplicaciones bajo la plataforma Java, Wicket se distingue por el nivel de implementación de los conceptos de objetos y por la potencia que brinda. Seguramente habrá frameworks que sean más simples de utilizar y/o aprender, pero definitivamente este framework de la fundación Apache merece atención.

Una de las principales características que posee es la clara separación entre vistas y controladores. Básicamente, cualquier página estará compuesta inicialmente por dos archivos:

- Un archivo HTML que contendrá la estructura de la página y definirá la manera en la que se visualizará la información.
- Por otro lado habrá una clase Java que respaldará la página HTML creando tantos componentes como elementos de HTML se hayan declarado. En otras palabras, si en el primer archivo existe un tag **<form>**, entonces en este segundo archivo se deberá instanciar un objeto que represente a dicho formulario.

El siguiente ejemplo ayuda a clarificar un poco más este punto:

HelloWorld.html

```
<html>
<body>
  <span wicket:id="message" id="message">Message goes here</span>
</body>
</html>
```

HelloWorld.java

```
import org.apache.wicket.markup.html.WebPage;
import org.apache.wicket.markup.html.basic.Label;

public class HelloWorld extends WebPage {
    public HelloWorld() {
        add(new Label("message", "Hello World!"));
    }
}
```

Figura 46: Clásico Hello World en Wicket

Como puede observarse en la figura anterior, en el archivo HTML solamente es necesario identificar al elemento **** con el mismo nombre que luego se utilizará en la clase Java cuando se cree una instancia del componente **Label**. Wicket luego parsea el código HTML reemplazando el contenido del **** por el texto que genere el componente Java.

En el caso de contar con un formulario, Wicket provee ciertas clases que implementan el protocolo esperable de los formularios, y que suelen ser extendidos por las aplicaciones para representar apropiadamente las necesidades de los formularios de cada aplicación particular.

Con esta muy breve explicación del funcionamiento de Wicket y de Spring en la sección anterior, ya nos encontramos en condiciones de presentar la herramienta prototipo que demuestra todos los conceptos y tips planteados a lo largo de este libro.

ITeM

El objetivo, a nivel funcional, de esta herramienta, es el permitir la gestión de defectos y/o requerimientos para proyectos de desarrollo de software. Herramientas como Jira, Mantis, Bugzilla son Buenos ejemplos de este tipo de herramienta. En las siguientes

secciones se irán presentando las diferentes capas a fin de ir comentando los detalles más relevantes.

Para acceder a todo el código para poder probar mínimamente esta herramienta es posible acceder al sitio SourceForge a través de la siguiente URL

<http://sourceforge.net/projects/issuetracker3/>

Por último, antes de comenzar con el código, una última aclaración. Si bien el código de la aplicación actualmente es mucho mayor del que tiene sentido presentar aquí, se ha seleccionado una parte representativa que alcanza perfectamente para demostrar cada uno de los puntos que se han señalado a lo largo del libro. El resto del código de la aplicación simplemente sigue la misma estructura. Por lo tanto en las próximas secciones se presentará el caso de uso de ingreso de un usuario. Nuevamente aclaro que se muestra un modelo con “recortes” por una cuestión de claridad y espacio.

Modelo

La principal responsabilidad de esta capa es representar los elementos del dominio, tanto sus relaciones como su comportamiento. Toda la lógica del “negocio” está contenida exclusivamente en esta capa.

La figura 47 presenta la parte del modelo relevante para modelar los usuarios y su administración.

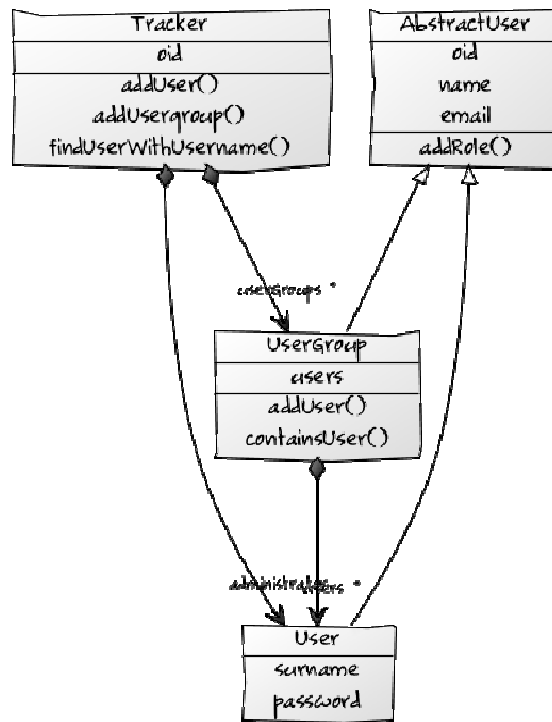


Figura 47: La capa del modelo de usuarios.

En la figura 48 a su vez se puede apreciar el código de la clase Tracker asociado con el ingreso de un usuario al sistema.

```

public User loginUser(String anUsername, String aPassword,
    EncryptionStrategy encryptionStrategy)
    throws PasswordMismatchException, UserUnknownException {

    User user = this.getUsersRepository().findUserWithUsername(this,
        anUsername, "C");

    if (user != null) {

        String encryptedPassword = encryptionStrategy.encrypt(aPassword);

        if (!encryptedPassword.equals(user.getPassword())) {

            throw new PasswordMismatchException();
        }
    } else {

        throw new UserUnknownException();
    }
    return user;
}

```

Figura 48: Ingreso de un usuario al sistema.

Vale la pena detenerse y explicar algunos puntos del código de más arriba:

- Este código pertenece a la clase **Tracker**, es decir que el tracker es “el” responsable del ingreso de los usuarios.
- Para poder validar el ingreso de un usuario, el tracker en primera medida trata de recuperar una instancia de la clase **User** que coincida con el *username* provisto. Ahora bien, aún cuando el tracker cuenta con la colección de usuarios, la búsqueda de un usuario se debe hacer en forma eficiente, por lo que en vez de usar iteradores se utiliza una implementación de repositorio específico para usuarios.
- El último parámetro indica que solamente se debe tratar de ubicar un usuario cuyo estado sea consolidado (“C”).
- Si el usuario es encontrado, entonces recién se compara su clave encriptada con el resultado de encriptar la clave recientemente ingresada⁵². En caso de no tener coincidencia se levanta una excepción de negocio, al igual que en el caso de no encontrar un usuario. Notar que la estrategia de encriptación⁵³ es pasada como parámetro al tracker, de modo que éste nunca sabe exactamente como encriptar sino que lo delega en la estrategia recibida.

Mapeo Hibernate

Esta capa contiene todas las decisiones respecto de las alternativas para “mapear” los objetos del modelo a sus correspondientes tuplas y tablas de la base de datos relacional.

La figuras 49 y 50 presentan los mapeos necesarios para persistir las clases del modelo.

⁵² Esto se debe a que la encriptación de la clave del usuario es asimétrica y no es posible desencriptarla.

⁵³ Implementación del patrón de diseño Strategy para realizar encriptación con diferentes algoritmos.

```

5 <hibernate-mapping package="zinbig.item.model.users">
6   <class name="AbstractUser" optimistic-lock="version" table="ABSTRACT_USER"
7     abstract="true">
8     <id name="oid" type="long" column="OID_USER" unsaved-value="0">
9       <generator class="org.hibernate.id.TableHiLoGenerator" />
10    </id>
11    <version column="version_id" name="version" />
12    <property name="name" length="50" />
13    <property name="deletable" column="is_deletable" />
14    <property name="email" length="50" />
15    <property name="creationDate" column="creation_date" />
16
17    <union-subclass name="User" table="ITEM_USER">
44    </union-subclass>
45
46    <union-subclass name="UserGroup" table="USER_GROUP">
66    </union-subclass>
67
68  </class>
69
70
71 </hibernate-mapping>

```

Figura 49: Archivo de mapeo de la clase **AbstractUser**.

Puntos importantes a considerar en este archivo de mapeo:

- Línea 6: se establece que el control de concurrencia será optimista basado en las versiones de los objetos.
- La línea 7 establece que la clase **zinbig.item.model.users.AbstractUser** debe ser considerada abstracta, por lo que ya se puede ir viendo la estrategia de mapeo que se utilizará para sus subclases concretas.
- La línea 11 define que la variable de instancia versión debe ser utilizada para el control de concurrencia.
- De las líneas 17 a 44 se realiza el mapeo de la subclase **User**, la cual representa usuarios simples o atómicos. En este caso se eligió utilizar la alternativa de mapeo para clases concretas únicamente.
- Las líneas 46 a 66 mapean la clase **UserGroup**, la cual modela los grupos de usuarios. Nuevamente se utiliza el mapeo de clases concretas, como en el caso de la clase **User**.

```

5 <hibernate-mapping package="zinbig.item.model">
6   <class name="Tracker" mutable="true">
7
8     <id name="oid" type="long" column="oid_tracker" unsaved-value="0">
9       <generator class="org.hibernate.id.TableHiLoGenerator" />
10    </id>
11
12    <bag name="users" lazy="extra" cascade="all">
13      <key column="oid_tracker" not-null="false" />
14      <one-to-many class="zinbig.item.model.users.User" />
15    </bag>
16
17    <bag name="userGroups" lazy="extra" cascade="all">
18      <key column="oid_tracker" not-null="false" />
19      <one-to-many class="zinbig.item.model.users.UserGroup" />
20    </bag>
21
22  </class>
23 </hibernate-mapping>

```

Figura 50: Archivo de mapeo de la clase **Tracker**.

Como puede verse en la figura anterior, las líneas 12 a 15 contienen la definición de la relación entre la instancia de la clase **Tracker** y sus usuarios, mientras que las líneas 17 a 20 hacen lo propio con las instancias de la clase **UserGroup**. En ambos casos se ha asignado a la propiedad *lazy* el valor “extra” para determinar que dichas colecciones se carguen utilizando un proxy en vez de la colección en sí misma. Asimismo, en ambos casos se ha seleccionado la actualización en cascada para todos los casos (*cascade=all*).

Base de datos

Esta capa es la que contiene propiamente los datos, en la forma de tablas. La única responsabilidad de esta capa es la citada más arriba. No hay lógica de negocios en la forma de stored procedures y/o triggers.

En la figura 51 se pueden apreciar los scripts necesarios para crear las tablas que almacenarán la información de las instancias de las clases **Tracker**, **User** y **UserGroup** respectivamente.

<pre> CREATE TABLE ITEM_USER (OID_USER NUMBER(19) NOT NULL, VERSION_ID NUMBER(10) NOT NULL, NAME VARCHAR2(50 BYTE), IS_DELETABLE NUMBER(1), EMAIL VARCHAR2(50 BYTE), CREATION_DATE DATE, PASSWORD VARCHAR2(50 BYTE), USERNAME VARCHAR2(50 BYTE), STATUS VARCHAR2(3 BYTE), LANGUAGE VARCHAR2(2 BYTE), SURNAME VARCHAR2(50 BYTE), OID_TRACKER NUMBER(19)) </pre>	<pre> CREATE TABLE USER_GROUP (OID_USER NUMBER(19) NOT NULL, VERSION_ID NUMBER(10) NOT NULL, NAME VARCHAR2(50 BYTE), IS_DELETABLE NUMBER(1), EMAIL VARCHAR2(50 BYTE), CREATION_DATE DATE, OID_TRACKER NUMBER(19)) </pre>	<pre> CREATE TABLE TRACKER (OID_TRACKER NUMBER(19) NOT NULL, OID_ADMINISTRATOR NUMBER(19) NOT NULL) </pre>
---	---	--

Figura 51: Script para la base de datos Oracle.

Como puede observarse los scripts no tienen ningún aspecto extraordinario. Simplemente el único punto interesante para resaltar es el hecho de cómo se le termina “pegando” la clave de la tupla de la tabla TRACKER a las tuplas de las tablas ITEM_USER⁵⁴ y USER_GROUP respectivamente.

Repositorios

El objetivo de esta capa es ofrecer una alternativa eficiente para recuperar los diferentes objetos del modelo persistente, encapsulando posiblemente algunos aspectos muy relacionados con la base de datos y con la persistencia en general.

En la figura 52 se puede apreciar el código que ejecuta una implementación de repositorios de usuario. En este caso se trata de una implementación particular para Hibernate (clase

zinbig.item.repositories.impl.hibernate.HibernateUsersRepository).

⁵⁴ En general es una buena práctica utilizar el mismo nombre para las tablas y las clases en un mapeo, pero en este caso, al utilizar la base de datos Oracle esto no es posible ya que “user” es una palabra reservada de la base de datos.

```

@Override
public User findUserWithUsername(Tracker aTracker, String anUsername,
    String status) throws UserUnknownException {

    Query aQuery = this.getNamedQuery("userByUsernameQuery");

    aQuery.setParameter("anUsername", anUsername);
    aQuery.setParameter("anStatus", status);
    aQuery.setMaxResults(1);

    User result = (User) aQuery.uniqueResult();

    if (result == null) {
        if (aTracker.getAdministrator().getUsername().equals(anUsername)) {
            result = aTracker.getAdministrator();
        } else {
            throw new UserUnknownException();
        }
    }
    return result;
}

```

Figura 52: Implementación de un repositorio de usuarios con Hibernate.

Es fácil notar el uso de consultas para encontrar rápidamente al usuario con el *username* dado. Sin embargo, hay que notar que las consultas no están escritas en el código, sino que han sido externalizadas para su mayor flexibilidad.

Nuevamente, en caso de no hallar al usuario se levanta una excepción de “negocio” propia de la aplicación, enmascarando cualquier evidencia de la presencia de una base de datos para las capas superiores.

Servicios

Esta capa es la que permite presentar a los clientes la funcionalidad provista por el modelo. Tiene como responsabilidad actuar tanto como un Facade (ya que presenta la funcionalidad del modelo en forma agregada, ocultando los detalles internos de los distintos módulos) y como un Adapter (ya que se encarga de transformar los elementos recibidos de la capa Web en sus correspondientes objetos del modelo y finalmente transforma a éstos últimos en DTOs para poder retornar valores a las subsiguientes capas).

A continuación se puede observar cómo un servicio de usuarios responde a los pedidos de ingreso de los usuarios.

```

@Override
public UserDTO loginUser(String username, String password) throws Exception

    Tracker aTracker = this.getTrackerRepository().findTracker();

    User anUser = aTracker.loginUser(username, password, this
        .getEncryptionStrategy());

    UserDTO anUserDTO = this.getDtoFactory().createCompleteDTOForUser(
        anUser);

    return anUserDTO;
}

```

Figura 53: Ingreso de usuarios en el servicio de usuarios.

Bien, a explicar algunas cuestiones:

- El primer paso que realiza el servicio es recuperar la única instancia de la clase **Tracker**, ya que como dijimos más arriba, ésta es la responsable de validar el ingreso de usuarios. Para recuperar en forma eficiente la única instancia de esta clase se utiliza una implementación del patrón Repositorio, específico para recuperar instancias de la clase **Tracker**.
- El servicio no instancia el repositorio que usará, sino que éste es inyectado a través de Spring para lograr flexibilidad e independencia respecto de las implementaciones posibles de este patrón.
- Luego, el servicio envía el mensaje *#login* al tracker. Si éste tiene éxito en el proceso de ingreso del usuario, una instancia de la clase **User** será devuelta. Aquí es preciso resaltar que tanto la instancia de **Tracker** como el usuario devuelto son objetos persistentes, por lo que se debe tener una transacción abierta para poder operar con ellos.
- Notar que en el código de la figura no es posible detectar las transacciones. Esto se debe a que su manejo se realiza en el archivo de configuración de Spring en forma declarativa.

- El último paso consiste en crear una representación del usuario, apta para ser utilizada una vez que se salga del contexto de las transacciones abiertas para el servicio. Esto lo hace a través de un colaborador, el cual implementa el patrón AbstractFactory.
- Nuevamente se debe recalcar que el servicio no contiene lógica de negocio, solamente adapta “ida y vuelta” las diferentes capas, en este caso Web y Modelo.

DTOs

Estos objetos son utilizados para intercambiar la información entre las diferentes capas, particularmente entre la capa Web y los diferentes servicios que se invocan.

Probablemente la capa de los DTOs sea la más simple y la que en términos generales requiera menos explicación, pero por las dudas se presenta en la siguiente figura cómo se crea un DTO para representar a un usuario.

```
public UserDTO createDTOForUser(User anUser, boolean hasOid,
    boolean mustCreateCompleteDTO) {
    UserDTO dto;

    // carga de los datos básicos
    dto = new UserDTO(anUser.getUsername(), this.getEncryptionStrategy()
        .decrypt(anUser.getPassword()), anUser.getLanguage(), anUser
        .getEmail(), anUser.getName(), anUser.getSurname(), anUser
        .isDeletable(), anUser.getVersion(), false);

    if (hasOid) {
        dto.setOid(anUser.getOid());
    }

    if (mustCreateCompleteDTO) {
        // carga información de las preferencias del usuario.

        dto.getUserPreferences().putAll(anUser.getUserPreferences());
    }

    return dto;
}
```

Figura 54: Creación de un DTO de un usuario.

En la figura anterior se crea un DTO para representar al usuario recibido como parámetro, sin demasiados secretos. La única salvedad es que algunas veces es necesario indicar a este objeto “creador de DTOs” si el usuario en cuestión tiene o no un id que lo identifique en la base de datos. Esto se debe principalmente a que cuando se

está pidiendo la creación de un DTO de usuario al darlo de alta, todavía no tiene un id asignado por la capa de persistencia, por lo que se debe controlar esta situación.

Formularios

La capa de formularios permite representar como objetos a los formularios con los que se componen las diferentes páginas de la aplicación.

Como parte de una de las últimas capas que merecen una explicación tenemos a los formularios, cuyas instancias son encargadas de la interacción entre el usuario y los servicios. En el caso particular del ingreso al sistema, tenemos un formulario (instancia de la clase **zinbig.item.application.forms.LoginForm**) en cual ante el evento `#onSubmit`⁵⁵ invoca al servicio de usuarios. La siguiente figura muestra el código correspondiente.

```
public void onSubmit() {  
  
    try {  
  
        UserDTO dto = ServiceLocator.getInstance().getUsersService()  
            .loginUser(this.getUsernameLF(), this.getPasswordLF());  
  
        // se registra de al usuario en la sesión  
        ((ItemSession) this.getSession()).setUserDTO(dto);  
  
    } catch (UserUnknownException e) { // usuario no encontrado  
  
        aTextField.error(this.getString("UserUnknownException"));  
        String classAttr = "background-color:#FF0000";  
        aTextField.add(new SimpleAttributeModifier("style", classAttr));  
        logger.debug("error", e);  
  
    } catch (PasswordMismatchException e) {  
  
        aPasswordField.error(this.getString("PasswordMismatchException"));  
        String classAttr = "background-color:#FF0000";  
        aPasswordField.add(new SimpleAttributeModifier("style", classAttr));  
        logger.debug("error", e);  
  
    } catch (Exception e) {  
        setResponsePage(ErrorPage.class);  
        logger.debug("error", e);  
    }  
}
```

Figura 55: Invocación al servicio desde un formulario.

⁵⁵ Este es un buen ejemplo de Inversión de Control, ya que es el framework Wicket el que invoca a este método del formulario cuando detecta que el usuario ha oprimido el botón “Enviar”.

La figura anterior muestra lo simple que es para un formulario, de la capa de la vista, invocar a un servicio. Lo único que debe hacer es obtener una referencia al mismo a través de una instancia de la clase **ServiceLocator**. Esta clase es en realidad un Singleton, a la cual se le “inyectan” las instancias de los servicios mediante Spring.

Servidor de aplicaciones

En esta capa, si bien no se trata formalmente de una, se especifican todas las cuestiones relacionadas con la integración de los diferentes frameworks, como Wicket y Spring para que colaboren en forma conjunta para la ejecución de la aplicación.

La figura 56 presenta un “resumen” del archivo de configuración web.xml.

```
<filter>
  <filter-name>Item</filter-name>
  <filter-class>org.apache.wicket.protocol.http.WicketFilter</filter-class>
  <init-param>
    <param-name>applicationFactoryClassName</param-name>
    <param-value>org.apache.wicket.spring.SpringWebApplicationFactory</param-value>
  </init-param>
</filter>
<filter-mapping>
  <filter-name>Item</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:applicationContext.xml</param-value>
</context-param>
```

Figura 56: Archivo web.xml de configuración.

En este archivo es donde se establece el filtro para que la aplicación ITeM funcione adecuadamente, vinculándolo con Spring a través de los parámetros *applicationFactoryClassName* y *contextConfigLocation*. El segundo parámetro en particular establece la ubicación del archivo applicationContext.xml, corazón de Spring.

Spring

El objetivo aquí es poder unir todos los componentes de la manera más práctica e independiente posible, prefiriendo siempre la alternativa basada en XML versus la alternativa “compilada”.

El último punto que nos toca ver es el archivo “corazón” de Spring, en el cual se vinculan prácticamente todas las capas anteriormente presentadas.

Como suele tratarse de un archivo de dimensión considerable, solamente se presentarán sus principales partes, dejando los detalles de menor importancia afuera.

Arrancamos entonces con la figura 57, la cual muestra un detalle de la configuración del “bean” llamado *sessionFactory*, que luego será utilizado para configurar los diferentes repositorios.

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource">
    <ref local="dataSource" />
  </property>
  <property name="mappingResources">
    <list>
      <value>zinbig/item/model/users/AbstractUser.hbm.xml</value>
      <value>zinbig/item/model/Tracker.hbm.xml</value>
      <value>zinbig/item/model/Operation.hbm.xml</value>
      <value>zinbig/item/model/projects/Project.hbm.xml</value>
      <value>zinbig/item/model/Item.hbm.xml</value>
      <value>zinbig/item/model/projects/Priority.hbm.xml</value>
      <value>zinbig/item/model/projects/PrioritySet.hbm.xml</value>
      <value>zinbig/item/util/i18n/I18NMessage.hbm.xml</value>
      <value>zinbig/item/util/i18n/ItemLocale.hbm.xml</value>
      <value>zinbig/item/util/SystemProperty.hbm.xml</value>
      <value>zinbig/item/model/workflow/WorkflowNode.hbm.xml</value>
      <value>zinbig/item/model/filters/Filter.hbm.xml</value>
      <value>zinbig/item/model/workflow/WorkflowDescription.hbm.xml</value>
      <value>zinbig/item/model/workflow/WorkflowNodeDescription.hbm.xml</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">${hibernate.dialect}</prop>
      <prop key="hibernate.show_sql">>false</prop>
    </props>
  </property>
</bean>
```

Figura 57: Configuración de la “fábrica” de conexiones de Hibernate.

Bien, el siguiente punto importante para ver es la configuración del objeto que será responsable de controlar las transacciones. Este “bean” se llama *transactionManager* y lo podemos ver en detalle en la siguiente figura.

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  <property name="sessionFactory">
    <ref local="sessionFactory" />
  </property>
</bean>
```

Figura 58: Detalle del objeto que administra transacciones.

Notar cómo es posible ir componiendo objetos complejos a partir de objetos más simples en Spring, en este caso configurando el administrador de transacciones con el objeto creador de sesiones de trabajo de Hibernate.

En la figura 59 por su parte se puede apreciar la configuración de los primeros repositorios, los cuales sólo requieren bean *sessionFactory*.

```
<bean id="trackerRepository"
      class="zinbig.item.repositories.impl.hibernate.HibernateTrackerRepository">
  <property name="sessionFactory">
    <ref local="sessionFactory" />
  </property>
</bean>

<bean id="usersRepository"
      class="zinbig.item.repositories.impl.hibernate.HibernateUsersRepository">
  <property name="sessionFactory">
    <ref local="sessionFactory" />
  </property>
</bean>
```

Figura 59: Configuración de repositorios.

Basta con presentar la configuración de dos repositorios diferentes para notar que todos tienen la misma forma: solamente se requiere inyectarles el objeto responsable de la creación de sesiones de trabajo de Hibernate y todo listo.

El último que nos toca ver es como se configuran los diferentes servicios que brindan acceso al modelo. La figura 60 nos permite apreciar la configuración de dos “beans”, el servicio propiamente dicho (bean “*userServiceTarget*”) y un proxy que agrega las características de transaccionalidad requeridas (bean “*userService*”).

```
<bean id="userServiceTarget" class="zinbig.item.services.impl.UsersServiceImpl"
    parent="baseServiceImpl">
    <property name="encryptionStrategy">
        <ref local="encryptionStrategy" />
    </property>
</bean>
<bean id="userService"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref local="transactionManager" />
    </property>
    <property name="target">
        <ref local="userServiceTarget" />
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
    <property name="preInterceptors">
        <list>
            <bean class="zinbig.item.util.spring.ServiceExceptionInterceptor" />
        </list>
    </property>
</bean>
```

Figura 60: Configuración de un servicio.

Obviamente el archivo de configuración cuenta con muchas más partes que se requieren para poder configurar apropiadamente todas las diferentes funciones de nuestro sistema, pero a modo de ejemplo los segmentos comentados alcanzan para mostrar la facilidad y claridad que nos brinda Spring.

Bibliografía

Artículos

Ambler, Scott, Mapping objects to relational databases.

< <http://www.ambyssoft.com/essays/mappingObjects.html>>. En línea. Consulta: 28 de septiembre de 2010.

Referencias

Catell, R.G.G., Barry, Douglas, The object data standard: ODMG 3.0, San Francisco, California, USA, Morgan Kaufmann Publishers, 1999.

Chaudhri, Akmal, Loomis, Mary, Object databases in practice, Prentice Hall, 1997.

Db4o. <<http://www.db4o.com>>. En línea. Consulta: 28 de septiembre de 2010.

Fowler, Martin. Patterns of enterprise application architecture. Addison – Wesley. 2002.

Gamma, Eric, Helm, Richard, Johnson, Ralph, Vlissides, John. Design patterns, elements of reusable object-oriented software. Addison- Wesley. 1995.

Hibernate. <<http://www.hibernate.org>>. En línea. Consulta: 28 de septiembre de 2010.

Java Data Objects. <<http://db.apache.org/jdo/index.html>>. En línea. Consulta: 28 de septiembre de 2010.

Kim, Won, Moderna database systems: the object model, interoperability, and beyond, Addison-Wesley, New York, USA, 1995.

ODBMS. < <http://www.odbms.org>>. En línea. Consulta: 28 de septiembre de 2010.

Spring framework. <<http://www.springsource.org>>. En línea. Consulta: 28 de septiembre de 2010.

Versant. <<http://www.versant.com>>. En línea. Consulta: 28 de septiembre de 2010.

Walls, Craig. Spring in action. Manning. 2008.

Wicket. <<http://wicket.apache.org>>. En línea. Consulta: 28 de septiembre de 2010.



ISBN: 978-950-34-0821-6