Instituto de Computación
Facultad de Ingeniería
Universidad de la República

MECANISMOS DE PERSISTENCIA EN SISTEMAS ORIENTADOS A OBJETOS

Informe de Proyecto de Grado para la obtención del grado de Ingeniero en Computación de la Facultad de Ingeniería, Universidad de la República

8 de junio de 2007
Montevideo - Uruguay

Autores:

Pablo Miranda
Joaquín Prudenza
Andrés Segurola

Supervisores:

Ing. Daniel Calegari
Ing. Jorge Corral

Resumen

La orientación a objetos se ha convertido en el paradigma dominante para el desarrollo de sistemas de información. En este contexto, el mecanismo utilizado para resolver la persistencia de los datos es de vital importancia, no solo por su impacto en el desempeño final del sistema sino también a los efectos de asegurar atributos deseables de calidad como la mantenibilidad, reusabilidad y escalabilidad. Existen numerosas alternativas a la hora de seleccionar un mecanismo para acceder a los datos y posteriormente manipularlos. Estas se diferencian en los enfoques utilizados, los contextos a los que aplican, su impacto a nivel arquitectónico, entre otros. La amplia gama de posibilidades puede resultar abrumadora a la hora de seleccionar uno acorde a las necesidades del proyecto. Por tal motivo, es de suma importancia contar con una visión general, objetiva y bien estructurada del problema. Este trabajo presenta un relevamiento de los mecanismos existentes para estructurar el acceso a datos persistentes en un sistema orientado a objetos, y en base a esto, una clasificación de mecanismos que mitiga carencias detectadas en clasificaciones existentes. Esta clasificación incluye tanto mecanismos de uso industrial (Bases de Datos Relacionales, Mapeadores Objeto-Relacional, etc.) como mecanismos aún emergentes académicamente (AOP, Prevalencia, etc.). Asimismo, se definen criterios de comparación entre mecanismos y entre productos que implementan cierto mecanismo. En base a estos criterios se realiza una comparación entre los mecanismos de la clasificación así como entre los productos Hibernate, JPA/Toplink Essentials y LLBLGen Pro correspondientes al mecanismo Mapeadores Objeto-Relacional. Los resultados obtenidos permiten mejorar la toma de decisiones a la hora de seleccionar el mecanismo (y producto) a utilizar por parte de un desarrollador.

Índice general

Capítulo 1

Introducción

El acceso a la información constituyó, en sus orígenes, una tarea de muy bajo nivel realizada sobre cintas magnéticas o tarjetas perforadas por grandes sistemas monolíticos. Conforme los sistemas se volvían más complejos mayores niveles de abstracción fueron necesarios. Esto llevó a plantear una clara separación arquitectónica entre el acceso a los datos y el procesamiento de los mismos. Actualmente, la realidad presenta sistemas de información distribuidos, interoperables, con fuentes de datos heterogéneas y con capacidades de almacenamiento masivo. Sin lugar a dudas, la orientación a objetos se ha convertido en el paradigma dominante para el desarrollo de este tipo de sistemas. En este contexto, el mecanismo utilizado para acceder a los datos es de vital importancia, no solo por su impacto en el desempeño final del sistema sino también a los efectos de asegurar atributos deseables de calidad como la mantenibilidad, reusabilidad, escalabilidad, etc.

Dado un sistema orientado a objetos, existen numerosas alternativas a la hora de seleccionar un mecanismo para acceder a los datos (mecanismo de persistencia de ahora en más) y posteriormente manipularlos. Estas alternativas se diferencian en: los enfoques utilizados, los contextos a los que aplican, su grado de usabilidad, portabilidad, desempeño, impacto a nivel arquitectónico, entre otros. En un principio, la amplia gama de posibilidades resulta abrumadora a la hora de seleccionar un mecanismo de persistencia concreto acorde a las necesidades. Quien asuma dicha tarea deberá contar con la información necesaria para justificar su elección. Esta elección no debe estar influenciada por cuestiones de marketing o sesgos personales, sino por una adecuada evaluación de los requerimientos. Por tal motivo, es de suma importancia contar con una visión general, objetiva y bien estructurada del problema.

De acuerdo con lo planteado anteriormente, es el propósito de este trabajo contribuir a la formación de la antedicha visión. Con esto en mente se plantean los siguientes objetivos:

- Realizar un relevamiento de los mecanismos de persistencia existentes para un sistema orientado a objetos, no solo incluyendo los utilizados a nivel industrial sino también aquellos emergentes académicamente.

- Proponer una clasificación completa y estructurada de mecanismos de persistencia. Esta clasificación será independiente de los productos existentes en el mercado a fin

de facilitar la compresión del dominio en cuestión, permitiendo además una forma de agrupar características y propiedades compartidas por todos los productos que implementen determinado mecanismo. Además, estará destinada a la elección por parte de un desarrollador del mecanismo a utilizar.

- Realizar una comparación entre los mecanismos a partir de un conjunto de criterios definidos a tales efectos, con el objetivo de contrastar sus diferencias y similitudes. Esto permitirá facilitar la selección de un mecanismo en determinado contexto.
- Plantear criterios de comparación de productos que implementen cierto mecanismo a los efectos de facilitar la toma de decisiones. Ejemplificar la comparación en base a la implementación de un caso de estudio utilizando diferentes herramientas características de un mecanismo seleccionado.

En cuanto a la organización del presente trabajo, el mismo se divide en cuatro capítulos. El Capítulo 2 realiza el relevamiento de la información referente a mecanismos de persistencia y sus temas relacionados. El Capítulo 3 se centra a nivel de mecanismos. El mismo presenta y analiza clasificaciones existentes para luego realizar una propuesta que intenta mitigar carencias detectadas en las anteriores. Asimismo se definen criterios de comparación a nivel de mecanismos y un conjunto de los mismos son evaluados. El Capítulo 4 presenta un conjunto de criterios de evaluación de productos y posteriormente la evaluación de un subconjunto de ellos tomando tres productos característicos del mecanismo Mapeadores Objeto-Relacional (uno de los más utilizados): Hibernate, JPA/-TopLink y LLBLGen Pro. Dicha evaluación hace énfasis en los criterios de performance y transparencia. Finalmente, el Capítulo 5 resume el trabajo realizado, expone las conclusiones fundamentales del mismo, y realiza una reseña de posibles trabajos futuros. El detalle de ciertos temas se exponen en los apéndices correspondientes.

Capítulo 2

Estado del Arte

2.1. Introducción

El procesamiento de la información ha evolucionado considerablemente en las últimas décadas; desde tiempos en que era realizado por grandes *mainframes* y tanto programas como datos eran almacenados en tarjetas perforadas, a la realidad actual de sistemas distribuidos, interoperables, con fuentes de datos heterogéneas y con capacidades de almacenamiento masivo. Toda aplicación que requiera del procesamiento de datos puede ser dividida a grandes rasgos en dos niveles: datos persistentes y el procesamiento de los mismos. Los mecanismos utilizados para interactuar con los datos son de vital importancia no solo por su impacto en el desempeño final del sistema sino también desde el punto de vista de aspectos tales como mantenibilidad, diseño, reusabilidad y escalabilidad.

Como primer solución destacable para la persistencia de datos surgen las bases de datos. En los comienzos, el acceso a la información constituía una tarea de muy bajo nivel. Con el objetivo de manejar la creciente complejidad de los sistemas, mayores niveles de abstracción fueron necesarios. Es así que surgen interfaces de acceso a datos y lenguajes de consulta estándares. Con el auge de los lenguajes de programación orientados a objetos y la gran aceptación de las bases de datos relacionales, aparecen los denominados mapeadores como mecanismos de traducción entre los paradigmas subyacentes. A la par del avance tecnológico, los procesos de desarrollo de software evolucionaron integrando nuevas herramientas de abstracción como ser lenguajes de modelado. Asimismo la experiencia de la comunidad de desarrolladores dio a conocer buenas soluciones para problemas conocidos, a las cuales se les denominaron patrones de diseño. Estos se beneficiaron de los lenguajes de modelado como herramienta para especificar soluciones. Tomando en cuenta que un sistema a menudo requería la aplicación repetitiva de distintos patrones, comienzan a popularizarse herramientas de generación automática de código. Si bien en la actualidad un conjunto de tecnologías para persistencia lideran el mercado, existen alternativas menos difundidas con enfoques diferentes como ser lenguajes persistentes, orientación a aspectos y prevalencia.

Para una visualización rápida de los hitos a lo largo del tiempo, ver Figura 2.1.

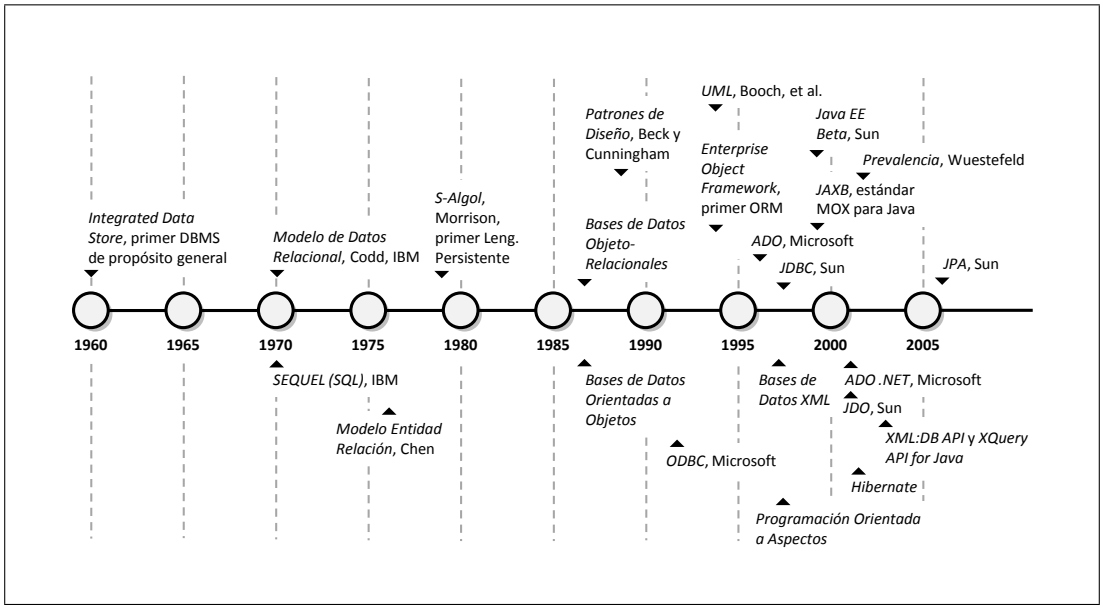


Figura 2.1: Estado del Arte - Hitos a lo largo del tiempo

2.2. Bases de Datos y DBMSs

Con la necesidad de almacenar grandes volúmenes de datos, controlar el acceso concurrente a los mismos y asegurar la consistencia y seguridad de la información surge el concepto de *Database Management System* (DBMS). De acuerdo con [EN06, Capítulo 1] se define DBMS como un paquete o sistema de software diseñado para facilitar la creación y el mantenimiento de bases de datos computarizadas. Asimismo, se entiende *Base de Datos* como una colección de datos relacionados.

El primer DBMS de propósito general, denominado Integrated Data Store, fue diseñado por Charles Bachean en General Electric a principios de la década del 60. El mismo constituyó la base para el Modelo de Datos de Red y fuertemente influenció los sistemas de base de datos en la década en cuestión. A finales de este período, IBM desarrolló el Information Management System (IMS), aun hoy utilizado en grandes instalaciones. El IMS constituyó la base para un método de representación de datos alternativo llamado Modelo de Datos Jerárquico.

Las bases de datos pre-relacionales no contaban con un conjunto de comandos que permitiese trabajar con los datos. Cada base tenía su propio lenguaje o utilizaba programas escritos en COBOL o C para manipular registros. Asimismo, eran virtualmente inflexibles y no permitían ningún cambio estructural sin tener que detener su ejecución y reescribir gran cantidad de código. Esto último funcionó de forma relativamente efectiva hasta fines de los 60 donde las aplicaciones se encontraban basadas estrictamente en procesamientos por lote. A principio de los años 70, el crecimiento de aplicaciones que requerían interacción por parte de los usuarios demandó la necesidad de algo más flexible. Situaciones en que un campo extra era requerido o un número de subcampos excedían la máxima cantidad permitida en determinado archivo se volvieron más y más comunes. Concretamente en 1970 Edgar Codd, en el San Jose Research Laboratory de IBM, propuso un nuevo modelo

de datos que se denominó *Modelo de Datos Relacional* [Cod83]. El mismo demostró ser una línea divisoria en el desarrollo de sistemas de base de datos. Éste produjo el rápido desarrollo de varios DBMSs basados en el mismo (RDBMSs), así como importantes resultados teóricos que ubicaron el campo sobre una base firme. Los sistemas de base de datos maduraron como una disciplina académica, y la popularidad de los DBMSs relacionales se extendió hasta el plano comercial. Sus beneficios fueron ampliamente reconocidos, y el uso de DBMSs para administrar información corporativa se convirtió en una práctica estándar. En 1980, el Modelo Relacional consolidó su posición como el paradigma dominante, y los sistemas de base de datos continuaron ganando aceptación.

A finales de los 80 y principio de los 90, se lograron varios avances en el área de sistemas de base de datos. Se realizó una considerable investigación para obtener lenguajes de consulta más poderosos y modelos de datos más expresivos. Hubo sin lugar a dudas un gran énfasis en el análisis de datos complejos. Varios proveedores como IBM, Oracle e Informix extendieron sus sistemas con la habilidad para almacenar nuevos tipos de datos (p.ej. imágenes) así como con la capacidad de realizar consultas más complejas. Sistemas especializados fueron desarrollados por varias empresas a fin de poder crear Data Warehouses, integrar información de distintas bases, y llevar a cabo análisis especializados de la información. Concretamente en este tiempo emergen los denominados *DBMSs Orientados a Objetos* (OODBMS) [ADM+92] y los *DBMSs Objeto-Relacional* (ORDBMS) [fADFC90].

La primera generación de OODBMSs data del año 1986 y su objetivo común fue el de proveer persistencia para lenguajes orientados a objetos (en particular para lenguajes relacionados con la inteligencia artificial). Las características distintivas de estos sistemas fueron que se trataban de sistemas *standalone*, se basaban en lenguajes propietarios y no hacían uso de plataformas estándar a nivel de la industria. El lanzamiento de Ontos en 1989 por la compañía del mismo nombre marcó el comienzo de la segunda generación en el desarrollo de OODBMSs. Esta generación se basó, a diferencia de la primera, en utilizar una arquitectura Cliente/Servidor y plataformas conocidas como C++, X Window y UNIX. El primer producto de la tercera generación, Itasca, fue lanzado al mercado en agosto de 1990. Si bien los OODBMSs de primera generación pueden ser considerados como extensiones a lenguajes orientados a objetos (típicamente extendían estos lenguajes con características de base de datos) los sistemas de tercera generación pueden ser definidos como DBMSs con características avanzadas (p.ej. soporte de versionado) y lenguajes orientados a objetos para la definición y manipulación de datos.

En lo que a este campo refiere, un aspecto interesante a tener en cuenta es que si bien la orientación a objetos se ha establecido firmemente en el mercado como uno de los paradigmas de desarrollo dominantes, su popularidad no tiene relación con la popularidad de los OODBMSs. En concreto, la adopción de las bases de datos orientadas a objetos ha sido considerablemente limitada. Actualmente, los RDBMSs (y ORDBMSs) dominan el mercado y es muy poco probable que sean remplazados (a menos en el corto plazo) por los OODBMSs. Algunas razones de este hecho se presentan a continuación:

- A pesar de la existencia de estándares para OODBMSs la realidad es que los productos disponibles solo los implementan de manera parcial.
- Existen muchas aplicaciones desarrolladas sobre la tecnología relacional. De acuerdo con ello, la idea de portar dichas aplicaciones a fin de utilizar OODBMSs es

dificultosa y poco realista.

- No es sencillo para los desarrolladores migrar a una nueva tecnología.

Con la misma idea que los OODBMSs, los ORDBMSs propusieron una alternativa para utilizar objetos con un sistema de base de datos. Mientras los primeros tomaron un camino revolucionario, los segundos se basaron en la evolución integrando el concepto de objeto al modelo de datos relacional y extendiendo los existentes RDBMSs con características orientadas a objetos (p.ej. encapsulación, herencia y polimorfismo). Durante toda la década del 90 se desarrolló una intensa investigación y experimentación en relación con estos sistemas y concretamente en 1996, los mayores proveedores de base de datos (Sybase, IBM y Oracle) comenzaron a migrar sus productos relacionales hacia productos objeto-relacionales.

A mediados de los 90, con el auge de Internet y la popularidad de HTML como formato para desplegar información en este medio surge XML. El mismo, fue inicialmente desarrollado como estándar para la Web a fin de permitir que los datos estuviesen claramente separados de los detalles de presentación. Rápidamente se notó que XML cubría la necesidad de una sintaxis flexible para intercambiar información entre aplicaciones. Si bien XML no fue inicialmente desarrollado como un lenguaje para persistir información, al ser adoptado por numerosas organizaciones para llevar a cabo sus procesos de negocios, surgió la necesidad de contar con un mecanismo para almacenar los datos en dicho formato. Alternativas como persistencia de los documentos en el sistema de archivos o en una base relacional presentaron numerosas desventajas. De acuerdo con ello surgen bases de datos capaces de entender la estructura de documentos XML así como de llevar a cabo consultas sobre los mismos de una manera eficiente; las mismas reciben el nombre de Base de Datos XML. Actualmente, se acepta la clasificación de las anteriores en dos categorías, a saber: *Base de Datos XML-Native* y *Base de Datos XML-Enabled* [Bou05]. Estas últimas se caracterizan por presentar un modelo de datos interno distinto a XML (generalmente se utiliza el Modelo Relacional) y un componente de software para la traducción correspondiente. De forma general, este módulo de traducción entre modelos no puede manejar todos los posibles documentos XML. En términos de lo anterior, reserva su funcionalidad para la subclase de documentos que cumplen con determinado esquema derivado a partir de los datos almacenados en la base. Finalmente, a diferencia de las XML-Enabled, las bases de datos XML-Native son aquellas que utilizan el modelo de datos XML directamente. Esto es, utilizan un conjunto de estructuras para almacenar documentos XML, en principio, arbitrarios.

2.3. Interfaces de Acceso a Datos y Lenguajes de Consulta

En los comienzos de las bases de datos, los desarrolladores necesitaban tener un conocimiento íntimo del producto particular que utilizaban. En aquel entonces no existía un mecanismo estándar para acceder a los datos sino que, dependiendo de la implementación concreta, se recorrían estructuras con punteros a fin de obtener los registros deseados. Esto sin lugar a dudas afectaba notablemente tanto la portabilidad y mantenibilidad de la solución así como la productividad del desarrollador. Conforme los sistemas fueron creciendo

en complejidad soluciones más flexibles fueron necesarias.

Durante los años 70, en el Centro de Desarrollo de IBM en San José se desarrolló un sistema de base de datos denominado System R basado en el modelo de datos de Codd. Asimismo, Structured English Query Language (SEQUEL) fue diseñado con el propósito de manipular y obtener la información almacenada en el sistema. El acrónimo SEQUEL fue posteriormente condensado a SQL por razones de derechos reservados. El lenguaje fue adoptado como estándar por ANSI en 1986 e ISO en 1989.

A principio de la década del 90 el Open Group desarrolló un API denominado Call Level Interface para COBOL y C que definía cómo los programas debían enviar sus consultas SQL al DBMS y cómo los datos retornados debían ser manejados. Si bien esta interfaz no tuvo popularidad comercial, la misma sirvió de base para el desarrollo de Open Database Connectivity (ODBC) por Microsoft en 1992. ODBC es una especificación neutral entre bases relacional que ofrece un API procedural para utilizar consultas SQL con la finalidad de acceder información. El desarrollo en cuanto al acceso a datos por parte de la empresa no se detuvo con esta tecnología sino que se tuvo un importante hito en 1996 al lanzarse al mercado ActiveX Data Objects (ADO). Una de las premisas de ADO fue proveer un mecanismo de acceso a datos mucho más simple en relación a lo que ofrecían productos anteriores. Hoy en día, como parte de la plataforma .NET se tiene una evolución de ADO denominada ADO .NET. Su diseño fue modificado específicamente para la Web, haciendo fuerte hincapié en aspectos tales como escalabilidad, independencia y el uso de XML como tecnología base.

Con el objetivo de lograr una interfaz neutral a la plataforma entre bases relacionales y Java, en 1997 Sun Microsystems lanza Java Database Connectivity JDBC. De acuerdo con la compañía, la tecnología JDBC es un API que permite acceder virtualmente cualquier fuente de datos tabular desde el lenguaje de programación Java. Provee conectividad con una gran cantidad de bases SQL, y hoy por hoy, permite acceso a otras fuentes tabulares como lo son hojas de cálculo y archivos planos. Sin lugar a dudas, a lo largo de los años JDBC se ha consolidado como un estándar de la industria para el acceso consistente y uniforme a bases relacionales a través de SQL. Asimismo, provee una base común a partir de la cual nuevas herramientas e interfaces de más alto nivel pueden ser construidas. JDBC brinda también un mecanismo de soporte para ODBC. Concretamente se tiene un puente ODBC que implementa JDBC en términos del API ODBC para C.

En un intento de estandarizar el acceso transparente a bases de datos desde lenguajes orientados a objetos, surge en 1991 el Object Data Management Group (ODMG). El estándar del grupo es anterior a Java, habiendo sido desarrollado cuando el mayor debate dentro la comunidad de desarrollo orientado a objetos tenía lugar: ¿Smalltalk o C++ sería el lenguaje dominante? El debate resultó ser académico tomando en cuenta que Java se convirtió en el estándar *de facto*. Esto último llevó a que ODMG respondiese adaptando sus interfaces C++ y Smalltalk a Java. Sin embargo, el proceso de adaptación resultó problemático y esto trajo como consecuencia que cualquier proveedor que tuviese una implementación que se pareciese a ODMG fuese libre de proclamarse conforme al estándar. La entrada de la Java Community Process (JCP) pareció en principio solucionar los problemas más inmediatos inherentes al proceso ODMG; el requerimiento de una implementación de referencia y una batería de tests para conformidad. En Mayo de 1999, el proceso fue adoptado y la Java Specification Request (JSR 12) fue enviada y aprobada para

el desarrollo contando además con el soporte de importantes proveedores de base de datos, *middleware* y herramientas. El desarrollo del estándar en cuestión, JDO, se llevó a cabo en paralelo con otros dos estándares: la Java Connector Achitecture (JCA) y la Container Managed Persistence (CMP). Los requerimientos de la JCP para una implementación de referencia y una batería de tests de conformidad se volvieron un factor determinante en el tiempo que llevó al lanzamiento final del estándar. Concretamente, en Abril del 2002 se lanza JDO 1.0. Hoy por hoy, y desde el 2004, JDO 2.0 se encuentra en desarrollo bajo el JSR 243.

En término de las bases de datos XML-Native, desde Julio de 2004, dos interfaces para acceso estandarizado están en proceso de desarrollo. Por un lado se tiene el XML:DB API por XML:DB.org que utiliza XPath como su lenguaje de consulta y está siendo extendido actualmente para el soporte de XQuery. Por otro lado, bajo el JSR 225 y basado en JDBC, se tiene el XQuery API for Java (XQJ) qué como su nombre lo indica, utiliza XQuery como lenguaje para consultar la base. Tomando en cuenta que su iniciativa está siendo liderada por IBM y Oracle, la eventual adopción generalizada se muestra probable.

2.4. Mapeadores

Con la creciente popularidad de los lenguajes orientados a objetos en el mundo del desarrollo de aplicaciones empresariales y de las bases de datos relacionales como medio de almacenamiento de datos, un problema presente en la integración de estas tecnologías se convirtió en el centro de atención. El problema en cuestión, denominado *Object-Relational Impedance Mismatch*, es el resultado de las diferencias existentes entre el paradigma de orientación a objetos y el paradigma relacional (p.ej. los conceptos básicos de identidad, clase, herencia y polimorfismo no son soportados de forma nativa por una base relacional) [Amb03, Capítulo 7].

Conforme los desarrolladores se vieron realizando la implementación del mapeo de objetos a datos relacionales una y otra a vez, surgen herramientas para automatizar este proceso. Estas herramientas recibieron el nombre de Mapeadores Objeto-Relacional. Las mismas pretenden encapsular el mecanismo de traducción de objetos a datos relacionales y viceversa. En 1994, el Enterprise Object Framework (EOF) fue introducido al mercado como un pionero en el mapeo objeto-relacional. Dicho *framework* gozó de notable popularidad hasta 1996, año en el cual el producto fue comprado por Apple Computer. En 1999 se libera la versión beta de la plataforma de Java EE. La misma cubría el área de mapeo como parte de la especificación de Enterprise Java Beans (EJBs). Este tipo de herramientas introdujeron una complejidad propia, requiriendo que los desarrolladores respetaran reglas detalladas para su uso. Debido a esto, las aplicaciones tenían que ser desarrolladas teniendo en cuenta las restricciones y requerimientos impuestos por el sistema de mapeo que utilizaban. Esto produjo que el desarrollo de aplicaciones empresariales se realizara mediante una mezcla contradictoria de razonamiento profundo (en cuanto a la lógica involucrada) y tareas repetitivas y triviales. Estas últimas tenían como objetivo cumplir con las restricciones y requerimientos de la herramienta de mapeo y se caracterizaban por consumir gran parte del tiempo de desarrollo.

Lo expuesto anteriormente contribuyó al surgimiento de una nueva tendencia por so-

luciones de mapeo más livianas. De acuerdo con ello, en el 2002 se lanza Hibernate 1.0, un *framework* de código libre para el mapeo objecto-relacional. A diferencia de otros productos, Hibernate se opuso a la generación de código como parte de la solución, permitiendo así que el framework se adecuara al desarrollo ágil de aplicaciones. En la actualidad Hibernate es una de las herramientas de mapeo más popular del mercado. La versión actual (v3.0) incorpora el uso intensivo de *annotations* y es un componente básico en la implementación de JBoss de EJB. Al mismo tiempo plataformas como Java EE están adoptando los principios de simplicidad en los cuales se basa fundamentalmente Hibernate. Con la liberación de Java EE 5 que incluye EJB 3.0 se propone a JPA como el API recomendado por Sun para el mapeo de objeto-relacional. Este API propone una significante reducción en el esfuerzo necesario para especificar el mapeo de una entidad. Tomando en cuenta que JPA funciona fuera de un contenedor EJB, es natural entender que también se trate de la propuesta de Sun para el mapeo de POJOs en Java SE.

En lo que a .NET se refiere una gran cantidad de herramientas de mapeo han emergido sin existir aun ningún producto dominante. Particularmente se puede distinguir NHibernate (un porte a .NET del popular *framework* para Java). En el 2005 Microsoft anunció un nuevo framework de mapeo objeto-relacional, ObjectSpaces, a ser parte del .NET Framework y a ser liberado junto con Visual Studio 2005. El proyecto fue aplazado para ser parte del .NET Framework 3.0 para luego ser cancelado y reemplazado por la tecnología DLinq, un componente del proyecto LINQ. Dicho proyecto pretende revolucionar el acceso a datos mediante un conjunto de extensiones para .NET Framwork que integra un lenguaje de consulta a los diferentes lenguajes de programación del mismo. En este contexto, DLinq presenta un nuevo enfoque al mapeo entre objetos y datos relacionales basado en la integración antes mencionada.

Con la creciente popularidad de XML como lenguaje para el intercambio de datos, se impulsa el desarrollo de otra modalidad de mapeo. La misma refiere al mapeo de objetos a documentos XML. Los Mapeadores Objeto-XML pretenden brindar acceso a los datos contenidos en un documento XML a través de objetos. A diferencia de modelos como DOM y JDOM, los Mapeadores Objeto-XML toman un enfoque centrado en los datos de forma tal que el desarrollador utiliza objetos que están relacionados con el contenido mismo de los documentos. Esto permite el uso de objetos para manipular y navegar los datos al mismo tiempo que encapsular muchas de las reglas del esquema XML en el comportamiento de los objetos. En 1999 comienza el proyecto de desarrollo de Java Architecture for XML Binding (JAXB), el estándar de mapeo objeto-XML para la plataforma Java. Un año más tarde se realiza la primer presentación pública del proyecto Castor, un *framework* de mapeo objeto-XML/objeto-relacional para la plataforma Java que tendrá un importante impacto en el área. A diferencia de otros, Castor es capaz de utilizar la tecnología Reflection para mapear clases previamente existentes (no es necesario utilizar clases generadas específicamente para este propósito). La conferencia JavaOne del 2001 se convierte en el escenario de la liberación de una versión Early Access de JAXB. La misma se valía de la generación de código Java a partir de los DTDs correspondientes a los documentos XML a ser mapeados. El año siguiente Sun anuncia que futuras versiones de JAXB contarían con una arquitectura rediseñada a partir del *feedback* recibido. Entre los cambios propuestos se encontraba la migración a W3C XML Schema y el abandono de la idea de un *framework* único (permitiendo que existan varias implementaciones que cumpliesen con la interfaz). En el 2006 se libera la versión 2.0 del estándar que incluye numerosas mejoras, nuevas

funcionalidades y sobre todo compatibilidad hacia atrás con JAXB 1.0.

Otro *framework* de mapeo con amplia aceptación en la comunidad es TopLink de Oracle. El mismo tiene sus orígenes como un *framework* de mapeo objeto-relacional para Smalltalk migrando en 1996 hacia la plataforma Java. Actualmente con la demanda de mapeadores objeto-XML, TopLink ha incluido soporte para esta funcionalidad. Al igual que Castor, TopLink permite el mapeo de objetos ya existentes al mismo tiempo que implementa JAXB 1.0. En el 2006 se libera la versión 1.0 del *framework* de mapeo Castor lo que significa un gran paso hacia la consolidación de la herramienta y al futuro soporte para JAXB 2.0 y JPA.

2.5. Lenguajes de Modelado

El modelado de los datos ha evolucionado en los últimos años de acuerdo a las necesidades de los diseñadores. El surgimiento de los primeros lenguajes de modelado se da en la década de los 70 a través del *Modelo Entidad-Relación* (MER) propuesto por Peter Chen [Che76]. El principal objetivo del MER es modelar el diseño de las bases de datos relacionales desde una perspectiva más abstracta y general, tratando de encontrar caminos para representar las semánticas de los modelos de datos que los unificaran. El MER no fue el único lenguaje de modelado propuesto, sino que hubieron otros intentos de desarrollar lenguajes de alto nivel para la definición de datos como ser el Conceptual Schema Language (CSL). Ya en los 80, se generalizó el uso del Modelo Entidad-Relación pero varios lenguajes de modelado surgieron con el propósito de satisfacer las nuevas necesidades en lo que respecta al diseño de bases de datos. Se realizan entonces varias extensiones al MER como por ejemplo el desarrollo de herramientas CASE. Las extensiones del MER (p.ej. dando soporte para generalización) tenían origen en el diseño y la programación orientada a objetos que comenzaba a ser de interés.

Al inicio los 90 los estudios referentes al diseño de bases de datos se centraron en la integración y transformación de esquemas, y en las diferentes medidas de calidad para los esquemas conceptuales. La evolución de la programación orientada a objetos, conlleva al desarrollo de lenguajes de modelado orientados a objetos. El desarrollo se produjo de manera independiente, desarrollándose más de cincuenta lenguajes de modelado. Esto no hizo más que limitar la evolución de los mismos. De acuerdo con ello es que en 1994, Booch, Jacobson y Rumbaugh deciden desarrollar un lenguaje de modelado orientado a objetos que pretende unificar la notación y semántica de la gran cantidad de lenguajes de modelado existentes. Dicho lenguaje fue denominado *Unified Modeling Language* (UML) [RJB04]. En 1997, varias compañías se unieron con el objetivo de lanzar una versión de UML que se convertiría en un estándar adoptado por la OMG para el modelado orientado a objetos. Desde finales de los 90 que UML ha ido evolucionando ofreciendo así nuevas funcionalidades de modelado como ser la definición de perfiles UML (permiten extender el lenguaje acorde a las necesidades).

2.6. Patrones de Diseño

Los patrones de diseño proveen soluciones reusables a problemas de diseño conocidos. Un patrón no define código y no es específico a determinado dominio de programación sino que por el contrario brinda una solución genérica y testeada para una clase de problemas de diseño similares. Asimismo, conllevan una terminología común que puede ser utilizada para documentar de forma sencilla y sistemática diseños propios [GHJV95].

El origen de los patrones de diseño se puede situar a fines de los 80 cuando Ward Cunningham y Kent Beck desarrollaron un conjunto de patrones para crear elegantes interfaces de usuario bajo Smalltalk. Alrededor del mismo tiempo, Jim Coplien se encontraba desarrollando un catálogo de patrones específicos para C++ que él denominó *idioms*. Mientras tanto, Erich Gamma, trabajando en su disertación doctoral sobre desarrollo orientado a objetos, reconocía el valor de explícitamente registrar estructuras de diseño recurrentes. Fue entonces que estas personas se reunieron e intensificaron sus discusiones sobre patrones en una serie de *workshops* de OOPSLA organizados por Bruce Andreson en 1991. Dos años más tarde la primera versión de un catálogo de patrones se encontraba en forma de bosquejo. El mismo eventualmente se establecería como base para el primer libro de patrones de diseño "Design Patterns: Elements of Reusable Object-Oriented Software" [GHJV95]. Todas estas actividades fueron influenciadas por los trabajos de Christopher Alexander, un arquitecto y planificador urbano quien por primera vez utilizó el término *patrón* para referirse a diseños recurrentes en el campo de la arquitectura de edificios. En el verano de 1993, un pequeño grupo de entusiastas en el tema de patrones formaron el llamado Hillside Generative Patterns Group y consecuentemente, en 1994, organizaron la primera conferencia sobre patrones que recibió el nombre de Pattern Languages of Programming (PLoP).

Así como los patrones de diseño documentan soluciones a problemas de diseño comunes, los patrones para persistencia tienen un rol similar en el campo en cuestión. Concretamente se presentan a continuación una serie de categorías para patrones de persistencia según lo descrito en [Noc03]. Los *Patrones de Separación* describen estrategias para aislar los componentes de acceso a datos de otras partes de una aplicación. Este aislamiento brinda flexibilidad a la hora de seleccionar un modelo de datos de soporte, así como también cuando se realizan cambios en las distintas formas de acceso a datos para un sistema. Los *Patrones de Recursos* por su parte describen estrategias para administrar los objetos involucrados en el acceso a bases relacionales. En concreto, pretenden solucionar temas de desempeño y semántica que frecuentemente surgen cuando se utiliza el acceso a través de una interfaz como ODBC o JDBC. La categoría de *Patrones de Entrada/Salida* describen soluciones que simplifican operaciones de entrada o salida de datos utilizando traducciones consistentes entre datos relacionales y los objetos de dominio relacionados. Los *Patrones de Cache* presentan soluciones que reducen la frecuencia de las operaciones para acceso a datos almacenando datos comunes un una estructura de cache. Los patrones en esta categoría usualmente se basan en cachear objetos de dominio en vez de datos físicos permitiendo así mayor eficiencia ya que la lógica de negocios trabaja con estos primeros directamente. Finalmente, la última categoría de patrones, *Patrones de Concurrencia*, trata qué sucede cuando múltiples usuarios realizan operaciones concurrentes que implican datos comunes.

2.7. Generadores de Código

La generación automática de código no es un concepto nuevo a pesar de haber tenido poco reconocimiento en el pasado. Los cambios recientes en la forma en que se lleva a cabo el desarrollo de sistemas ha causado interés en el uso de generadores de código como una forma de disminuir el tiempo de desarrollo así como también los recursos humanos necesarios. La complejidad involucrada en el desarrollo de software se ha visto incrementada en los últimos años. Los desarrolladores poseen cada vez menos tiempo para dedicar a la lógica de negocio o los algoritmos que son la razón misma por la cual el software está siendo desarrollado. A pesar de la existencia de patrones de diseño y técnicas de programación comprobadas, programar las otras partes del sistema se ha convertido en una tarea tediosa y consumidora de tiempo. Esta es la motivación fundamental por la cual surgen los generadores de código en la actualidad. La generación de código implica básicamente utilizar herramientas que escriban programas. Con *frameworks* de la actualidad como ser Java EE, .NET y MFC, es cada vez más importante hacer uso de herramientas que asistan en la construcción de aplicaciones. En términos generales se podría decir que, cuanto más complejo sea el *framework* utilizado, más atractiva es la idea de una solución que involucre generadores de código.

Java es uno de los lenguajes más populares para la generación de código. Esto se debe básicamente a la complejidad de los *frameworks* disponibles para la plataforma, y el hecho de que algunos aspectos de diseño del lenguaje lo hacen ideal para la generación. Por estas razones existe una gran variedad de herramientas de generación de código bajo Java. En lo que respecta a .NET la generación de código es un área emergente que ha experimentado un amplio crecimiento en los últimos años. En la actualidad existe una gran variedad de productos como también portales web dedicados únicamente a la evaluación y difusión de los mismos (p.ej. [Cod]).

Dentro de los generadores de código son de interés particular aquellos capaces de generar una capa de acceso a datos. Este es uno de los usos más comunes de los generadores ya que el código de acceso a datos, por ejemplo a una base de datos, suele ser repetitivo y muy susceptivo a errores. En este contexto, los generadores de código pueden ser divididos en dos categorías: aquellos que generan código a partir de un proyecto y su *metadata*, y aquellos que analizan el esquema de la base subyacente a fin realizar la generación correspondiente. Un ejemplo de generador de código es FireStorm/DAO, capaz de generar una capa de persistencia aplicando el patrón DAO (patrón de diseño para el acceso a datos) . Más allá de este producto, en la actualidad existen cientos de generadores de código en el mercado. Los mismos varían en sus habilidades, plataformas y enfoques utilizados. La mayoría de los generadores de código suelen formar parte de paquetes mayores de desarrollo y modelado (generalmente desarrollados para ser aplicados en procesos basados en *Model Driven Development*).

2.8. Técnicas Alternativas

Existen diferentes alternativas para la persistencia de información de un sistema que, no por menos difundidas son menos interesantes. Si bien su objetivo es común a las técni-

cas anteriormente expuestas, los enfoques utilizados son totalmente diferentes. Una de estas alternativas es la Programación Orientada Aspectos, donde la persistencia será considerada un aspecto que puede ser implementado de forma ortogonal a los objetos de un sistema. Otras alternativas son los Lenguajes Persistentes y la técnica de Prevalencia. Las mismas proponen una alternativa a la utilización de DBMSs para el almacenamiento y administración de los datos, haciendo además que la persistencia sea transparente al desarrollador.

2.8.1. Programación Orientada a Aspectos (AOP)

A medida que el paradigma de orientación a objetos fue ganando popularidad entre los desarrolladores, se comienza a considerar los sistemas como grupos de entidades que interactúan entre sí. Esto fue permitiendo la construcción de aplicaciones más complejas en menor tiempo de desarrollo. Sin embargo, existe el problema que al efectuar cambios sobre ciertos requerimientos que involucren a varios objetos, pueda producirse un profundo impacto en los tiempos de desarrollo. Una posible solución al antedicho problema es la aparición de una nueva forma de programación, la *Programación Orientada a Aspectos*, presentada formalmente en el año 1997 [KLM+97]. Su característica fundamental es la posibilidad de separar ciertos requerimientos mediante la definición de abstracciones que encapsulen características que cortan transversalmente a un sistema. Dichas abstracciones pueden ser requerimientos de seguridad, manejo de excepciones, logueo, persistencia, etc. El aspecto de persistencia es de particular interés como técnica para persistir la información de un sistema, existiendo distintas investigaciones sobre el tema. Desde el surgimiento de la programación orientada a aspectos, distintos *frameworks* se han desarrollado con el fin de definir y componer los aspectos de un sistema. Hoy en día se pueden encontrar *frameworks* para Java (p.ej. AspectJ) o para .NET (p.ej. AOSD.net) entre otros.

2.8.2. Lenguajes Persistentes

En un principio, los lenguajes de programación tradicionales requerían el uso de otras herramientas que se encargaran de la manipulación de los datos cuyo ciclo de vida excediera el de la ejecución de la aplicación. En los últimos tiempos, el almacenamiento de los datos ha sido de interés para los diseñadores de los lenguajes de programación. Es así que surgen los *Lenguajes Persistentes* como mecanismo de persistencia, caracterizándose por cumplir con los tres principios de persistencia ortogonal: *ortogonalidad de tipo* (cada objeto, independiente de su tipo, tiene el mismo derecho a ser persistido), *persistencia por alcance* (se persisten sólo aquellos objetos que son alcanzables desde la raíz de persistencia) e *independencia de persistencia* (la introducción de persistencia no puede introducir cambios semánticos del código fuente). Uno de los primeros *Lenguajes Persistentes* desarrollados fue S-algol, basado en los principios de abstracción, ortogonalidad e independencia de tipo de dato. Dicho lenguaje sólo fue utilizado para fines académicos. Posteriormente se realizaron extensiones de S-algol, agregando nuevas funcionalidades, por lo que pasó a llamarse PS-algol. Tomando los principios de PS-algol fue que se desarrollaron numerosos prototipos y extensiones a lenguajes tradicionales para que soporten de forma nativa la persistencia de los datos. Hoy en día existen lenguajes persistentes como

ser PJama en forma de extensión de Java y Barbados para C++.

2.8.3. Prevalencia

A través de los años, los desarrolladores han utilizado diferentes mecanismos de persistencia de objetos. Esto tuvo como principal problema la utilización de distintas formas de almacenar los datos que no eran orientadas a objetos. Esto ha llevado a buscar alternativas para solucionar el antedicho problema. Una de las alternativas es la *Prevalencia de Objetos*, propuesta por Klaus Wuestefeld en el año 2001 [Wue03]. El concepto de prevalencia implica el mantenimiento de la totalidad de las instancias de objetos que componen un sistema en memoria principal. La persistencia se realiza mediante *snapshots* del estado del sistema que se efectúan de forma periódica y que utilizan la técnica de serialización de los objetos en memoria. La primer implementación de prevalencia se desarrolló como un proyecto Java de código abierto, denominada Prevayler. Recién en el 2004 se lanza la versión 2.0 de Prevayler que incluye funcionalidades como ser *rollback* automático ante fallos, volcados XML, y mejoras en la escalabilidad global. Distintas implementaciones se han desarrollado para otros lenguajes de programación. Un ejemplo de ello es Bamboo Prevalence, una implementación para .NET.

2.9. Resumen

Toda aplicación que requiera del procesamiento de datos puede ser dividida a grandes rasgos en dos niveles: datos persistentes y el procesamiento de los mismos. Los mecanismos utilizados para acceder e interactuar con los datos son de vital importancia no solo por su impacto en el desempeño final del sistema sino también desde el punto de vista de aspectos tales como mantenibilidad, diseño, reusabilidad y escalabilidad. Como se pretendió mostrar en este capítulo, las formas disponibles para la persistencia de información es un campo sumamente amplio. El mismo abarca desde las bien conocidas bases de datos, pasando por mapeadores y generadores de código, extendiéndose hacia alternativas menos difundidas y emergentes como la prevalencia. Asimismo se destacan, aunque no permiten la persistencia directamente, tópicos como los lenguajes de modelado y patrones de diseño. Se hace fuerte hincapié en notar la gran variedad de alternativas a la que se enfrenta un desarrollador a la hora de solucionar la persistencia de determinado sistema.

Capítulo 3

Mecanismos de Persistencia

3.1. Introducción

De acuerdo con lo presentado en el Capítulo 2 existen numerosas alternativas a la hora de seleccionar un mecanismo para acceder a los datos y posteriormente manipularlos. Estas alternativas se diferencian en los enfoques utilizados, los contextos a los que aplican, su grado de usabilidad, portabilidad, desempeño, impacto a nivel arquitectónico, entre otros. En un principio, la amplia gama de posibilidades resulta abrumadora a la hora de seleccionar un mecanismo de persistencia concreto acorde a las necesidades. Quien asuma dicha tarea deberá contar con la información necesaria para justificar su elección. Esta elección no debe estar influenciada por cuestiones de *marketing* o sesgos personales, sino por una adecuada evaluación de los requerimientos. Por tal motivo, es de suma importancia contar con una visión general, objetiva y bien estructurada del problema.

A fin de contribuir con esta visión objetiva, se propone una clasificación de los mecanismos de persistencia existentes para sistemas orientados a objetos. Esta clasificación presenta una descripción de cada uno de los mecanismos con sus principales características, los contextos factibles de aplicación, las ventajas y desventajas, las herramientas representativas de cada mecanismo, etc. Además, se presenta un análisis comparativo de los mecanismos, basado en un conjunto de criterios de comparación, lo que permite obtener un mejor entendimiento global de las alternativas disponibles. Cabe notar que si bien existen trabajos en este sentido, las clasificaciones realizadas no son completas, no describen apropiadamente las categorías que definen, o no están orientadas a que un desarrollador las utilice como guía para la selección del mecanismo apropiado. La clasificación propuesta pretende subsanar dichas carencias.

Un punto importante a tener en cuenta en este capítulo es que se busca independizarse de productos particulares y mantenerse a un nivel de abstracción mayor. La idea detrás de ello es permitir simplificar la visión del dominio complejo con el que se trata. De acuerdo con esto se consideran *Mecanismos de Persistencia*. Se define esto último, en el contexto del desarrollo de un sistema orientado a objetos, como una técnica básica que permite resolver la *persistencia* de objetos. Esto implica extender el tiempo de vida de éstos mas allá del tiempo de vida del proceso que los creó. Téngase en cuenta la relación existente

entre un producto particular y un mecanismo. Esto es, un producto aplica de forma general uno o más mecanismos con el objeto de solucionar la persistencia de determinado sistema orientado a objetos, con cierto lenguaje, tecnología o plataforma.

El capítulo está organizado de la siguiente manera. En la Sección 3.2 se presentan y analizan clasificaciones existentes para mecanismos de persistencia. La Sección 3.3 expone la clasificación propuesta y las ventajas que ofrece frente a las existentes. En la Sección 3.4 se definen criterios de comparación a nivel de mecanismos y un conjunto de los mismos son evaluados. Finalmente, la Sección 3.5 resume lo presentado en este capítulo.

3.2. Clasificaciones Existentes

Optar por un mecanismo u otro implica evaluar las características de cada uno de ellos, correlacionándolos con las necesidades del sistema a construir. Por tal motivo, es de suma importancia contar con una visión general, objetiva y bien estructurada del problema. En este sentido, existen algunas clasificaciones para mecanismos de persistencia pero ninguna de ellas ofrece una vista clara y completa del problema. Asimismo, no asisten al desarrollador en la selección de un mecanismo (o un conjunto de de los mismos) basándose en el contexto en que este se encuentre.

3.2.1. Clasificación I: *Software Practice Advancement 2006*

En el contexto de un *workshop* realizado en la conferencia Software Practice Advancement 2006 [SPA06] se elaboró una clasificación de mecanismos de persistencia (Figura 3.1). Dicha clasificación se llevó a cabo con el objetivo de ser parte de una guía a ser utilizada por los desarrolladores al momento de elegir un mecanismo de persistencia. La misma es el resultado de los conocimientos y experiencia de los participantes de dicho evento. Se incluyeron arquitectos, diseñadores y desarrolladores. De acuerdo con ello se supone que la clasificación obtenida es representante de las técnicas más utilizadas en la industria.

Se trata de una clasificación jerárquica de un solo nivel donde se enumeran agrupaciones generales y luego se presentan productos concretos que caracterizan cada grupo. La misma se encuentra orientada (por la naturaleza de los participantes del evento) a su utilización en forma práctica. A continuación se enumeran algunas aclaraciones con respecto a las categorías consideradas (ver Figura 3.1). Se distinguen tres categorías basadas en la misma técnica. A saber, *O/R Mechanisms*, *Old O/R* y *O/R Frameworks*. En cuanto a la categoría denominada *Bespoke*, la misma refiere a aquellos mecanismos desarrollados a medida con el objetivo de solucionar la persistencia de una aplicación especifica (no se trata de soluciones genéricas). La categoría de *Pattern Mechanisms* refiere por su parte a mecanismos que solucionan la persistencia mediante la aplicación de patrones de diseño.

Cabe notar que el origen informal del evento hizo que no exista una definición concreta y concisa de las categorías consideradas. Esto claramente imposibilita comprender el alcance de cada una ellas. De acuerdo con ello la utilidad de la clasificación se ve seriamente restringida. Asimismo considerar un único nivel en la jerarquía no permite exhibir un grado de granularidad adecuado ni agrupar categorías similares. Otras carencias detectadas se

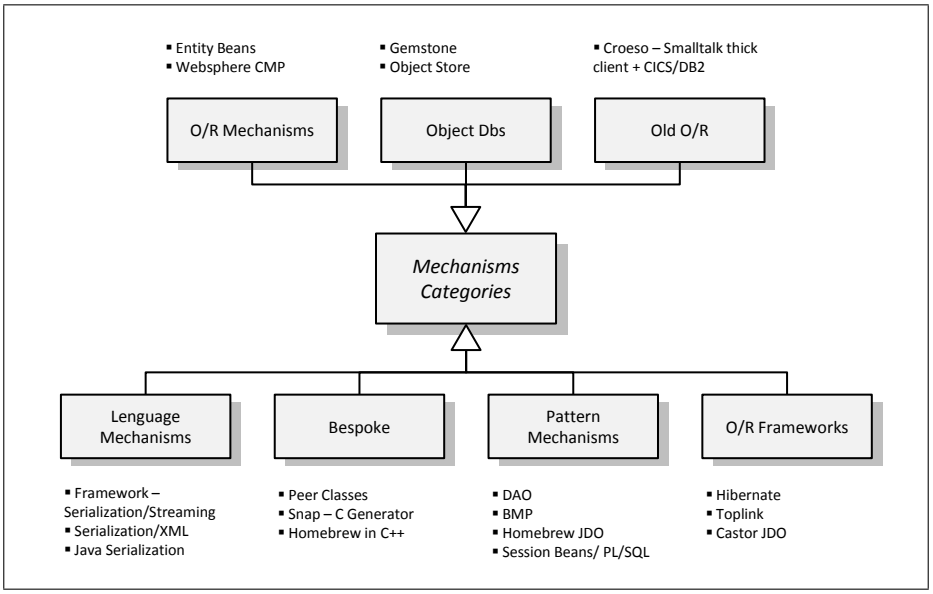


Figura 3.1: Clasificación I - Software Practice Advancement 2006

ven reflejadas en hacer hincapié en productos particulares y no considerar técnicas menos establecidas como por ejemplo la Programación Orientada a Aspectos. Además se proponen categorías con gran similitud cuya distinción no tiene justificación desde el punto de vista del mecanismo (p.ej. las categorías O/R Mechanisms, Old O/R, O/R framework).

3.2.2. Clasificación II: *Survey of Persistence Approaches*

Se trata de una tesis de maestría de titulo "Survey of Persistence Approaches" [Tak05] que realiza una comparación entre diferentes mecanismos de persistencia como un paso hacia la definición de un mecanismo de persistencia ideal y transparente. Para ello se describen los mecanismos más relevantes existentes (Figura 3.2) y se definen criterios de comparación a ser aplicados a productos particulares. Los criterios más importantes considerados son los siguientes: ortogonalidad de tipo, independencia de persistencia, alcance, integridad, soporte de transacciones, reusabilidad, lenguaje de consulta y *performance*.

La *ortogonalidad de tipo* establece que todos los objetos poseen los mismos derechos de ser persistidos independientemente de su tipo. La *independencia de persistencia* se refiere al hecho de que no sea necesario realizar cambios en el código fuente para lograr que un objeto sea persistente. Todo mecanismo requiere de una estrategia para indicar qué objetos han de ser persistidos. Usualmente se cuenta con un objeto denominado *raíz de persistencia* tal que todos aquellos objetos alcanzables desde él son persistentes. Dicho enfoque es denominado *persistencia por alcance*. El propósito del *chequeo de integridad* es proveer medios que garanticen la consistencia de los datos. El *soporte transaccional* provisto por el mecanismo se refiere particularmente a la cobertura de las propiedades ACID. En cuanto al principio de *reusabilidad*, el mismo es definido como la habilidad de ejecutar una aplicación en un contexto persistente o convencional sin la necesidad de modificar el código fuente. El criterio de *lenguaje de consulta* se refiere al mecanismo de

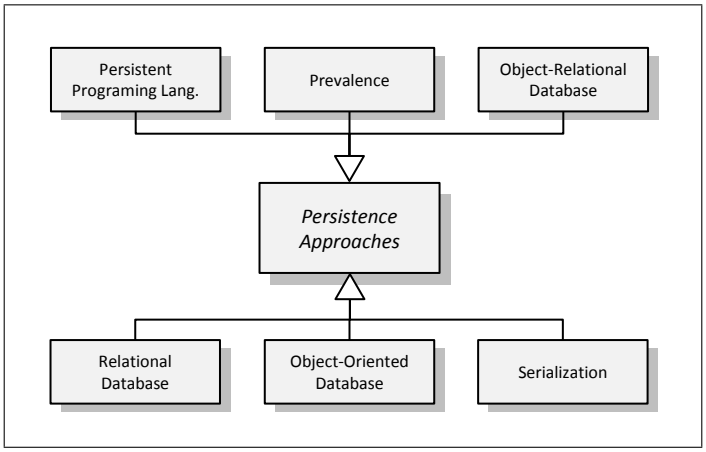


Figura 3.2: Clasificación II - Survey of Persistence Approaches

recuperación de objetos provisto. Las estrategias utilizadas por los diferentes mecanismos para mejorar su *performance* es otro de los criterios considerados.

Con respecto a esta clasificación se puede notar que la misma no está orientada al desarrollador ya que considera exclusivamente categorías guiadas por la técnica de bajo nivel utilizada (bases de datos, serialización, etc.). Se trata de una clasificación jerárquica de un solo nivel que no es completa al no incluir alguno de los mecanismos relevantes (p.ej. Mapeadores Objeto-XML, Objeto-Relacional, etc.). Otro aspecto a destacar es que la comparación se realiza entre productos y no categorías, por lo que los resultados están sujetos a la utilización del producto utilizado en la comparación.

3.2.3. Clasificación III: *Agile Database Techniques*

En [Amb03, Capítulo 6] se describen los mecanismos de persistencia como alternativa al uso de una base de datos relacional (Figura 3.3), mecanismo que considera el más relevante y sobre el cual se basa el contenido del libro. Se incluye, como parte de la definición de las categorías, contextos de aplicación así como también ventajas y desventajas.

La categorización se focaliza en el destino de los datos. Más allá de que es un criterio válido para la clasificación se trata de una partición demasiado general a la hora de utilizarla como guía en la elección de un mecanismo apropiado. Como un ejemplo de lo anterior, si se considera la categoría que tiene a una base de datos relacional como destino de los datos, existen múltiples opciones para el desarrollador que estarían dentro de esta categoría. A modo de ejemplo: mapeadores objeto-relacional, generadores de código, etc.

3.2.4. Clasificación IV - *Approaches to Adding Persistence to Java*

Este artículo [MH96] presenta una clasificación basada en una serie de preguntas centradas en la plataforma Java, cuyas respuestas determinan la categoría a la que pertenece el mecanismo. Dichas preguntas se concentran en tres aspectos: ortogonalidad, transparencia e implementación.

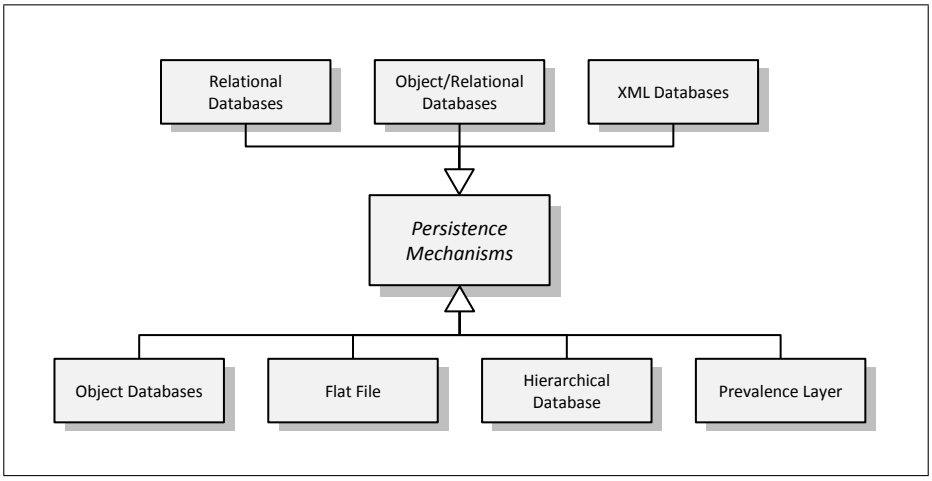


Figura 3.3: Clasificación III - Agile Database Techniques

Por ortogonalidad se entiende la propiedad de un mecanismo de permitir que cualquier instancia de una clase sea potencialmente persistente sin que los programadores estén obligados a indicarlo explícitamente. Esta definición es de referencia ya que rara vez es deseable o posible en su forma más pura. En este sentido, las preguntas apuntan a conocer el nivel de ortogonalidad del mecanismo. Las mismas se exponen a continuación:

- ¿Se trata de persistencia por alcance (a partir de un objeto raíz), o se utiliza algún otro método para designar cuales objetos deben ser persistidos?
- ¿El código (de objeto) es persistente (tanto comportamiento como datos)?
- ¿El estado de ejecución del programa es persistente?
- ¿Cuál es el modelo de transacciones?

El artículo define transparencia como la medida en que un programa que potencialmente requiere la persistencia de objetos luce esencialmente igual a uno que no tiene este requerimiento. Por más que la transparencia suele considerarse solo en términos de código fuente, al tratarse Java no solo de un lenguaje sino también de una maquina virtual estandarizada, otras consideraciones pueden incluirse. Las preguntas planteadas son:

- ¿Es necesario modificar el lenguaje para persistir?
- ¿Es necesario modificar el byte-code generado?

Por implementación se refiere a si es necesario cambiar la propia implementación de la plataforma para persistir, sin que el programador realmente conozca que es así. Las preguntas propuestas en este aspecto son las siguientes:

- ¿Se cambia el compilador de Java?

- ¿Se cambia el intérprete de Java y el ambiente de ejecución?

Lo presentado se trata de un enfoque diferente a los vistos anteriormente. Concretamente, cada tupla posible de respuestas determina una categoría. Debido a que no se define cuales combinaciones de respuestas son válidas (si existe o no un mecanismo para ellas) no es una clasificación clara. Una desventaja fundamental a destacar es que la clasificación se restringe a la plataforma Java. Además, se puede agregar que la misma se concentra únicamente en aspectos de ortogonalidad, transparencia e implementación sin considerar otras características relevantes, como por ejemplo las consideradas en 3.2.2.

3.2.5. Clasificación V: *Integrating Programming Languages & Databases: What's the Problem?*

El artículo [CI05] presenta una clasificación basada en un conjunto de criterios de comparación entre distintas herramientas. Se plantea el problema del *impedance mismatch* que se produce entre lenguajes de programación orientados a objetos y las bases de datos, y se analizan diferentes soluciones al problema definiendo qué criterios utilizar para comparar diferentes herramientas de cada solución.

Los criterios de comparación se dividen en tres categorías: tipado, optimización y reuso. El *tipado* refiere a la diferencia entre los tipos de datos de lenguajes de programación y bases de datos, y cómo se realiza el mapeo de los datos propiamente dichos. La *optimización* refiere a cómo optimiza cada solución el acceso a los datos. Finalmente, el *reuso* implica que las consultas sobre los datos puedan ser parametrizadas, dinámicas y modularizadas.

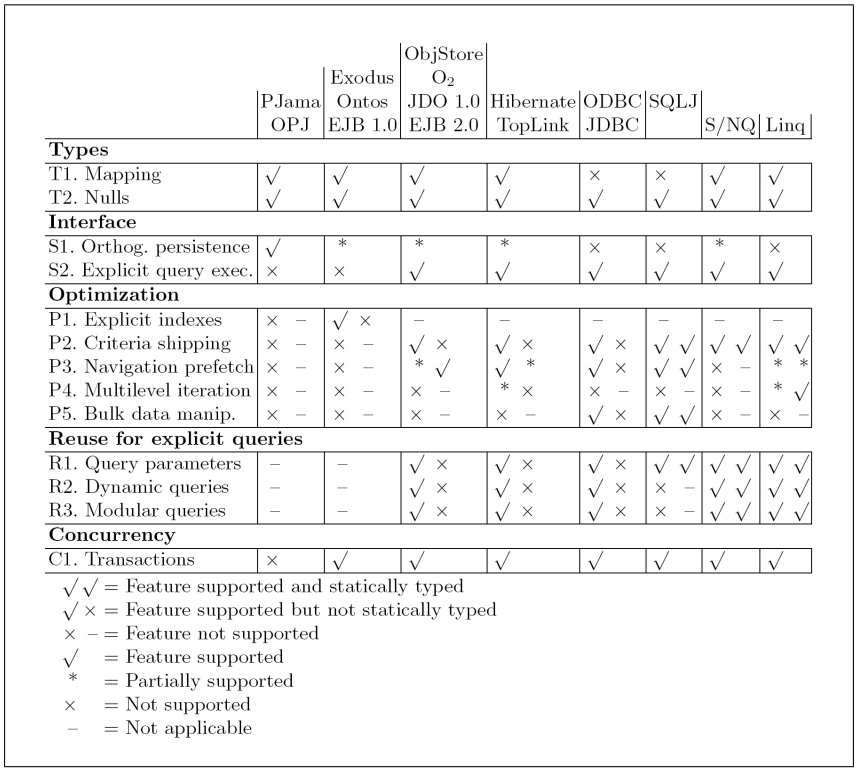
Estos criterios se aplican en soluciones específicas al *impedance mismatch* como ser bases de datos orientadas a objetos, mapeadores objeto-relacional, APIs de acceso a datos, lenguajes de programación con persistencia ortogonal, y lenguajes de consulta embebidos.

Como resultado de la evaluación de los criterios se obtiene una tabla (Figura 3.4) en la que se agrupan herramientas que comparten los mismos valores para determinados criterios de comparación. Esto último lleva a que herramientas que coinciden en los valores de comparación, sean consideradas dentro de la misma categoría. Una desventaja es que la comparación se realiza sobre herramientas particulares, no fundamentando el por qué de su elección.

3.2.6. Criticas a las Clasificaciones Existentes

Las clasificaciones existentes presentan distintas carencias que las hacen inadecuadas para ser utilizadas por un desarrollador con el objetivo de entender la situación actual en cuanto a mecanismos de persistencia. De las clasificaciones anteriores se pueden destacar debilidades en los siguientes aspectos:

- *Completitud*. La falta de completitud de las mismas en cuanto a los mecanismos que éstas consideran.



	PJama OPJ	Exodus Ontos EJB 1.0	ObjStore O_2 JDO 1.0 EJB 2.0	Hibernate TopLink	ODBC JDBC	SQLJ	S/NQ	Linq
Types								
T1. Mapping	√	√	√	√	×	×	√	√
T2. Nulls	√	√	√	√	√	√	√	√
Interface								
S1. Orthog. persistence	√	*	*	*	×	×	*	×
S2. Explicit query exec.	×	×	√	√	√	√	√	√
Optimization								
P1. Explicit indexes	× –	√ ×	–	–	–	–	–	–
P2. Criteria shipping	× –	× –	√ ×	√ ×	√ ×	√ √	√ √	√ √
P3. Navigation prefetch	× –	× –	* √	√ *	√ ×	√ √	× –	* *
P4. Multilevel iteration	× –	× –	× –	* ×	× –	× –	× –	* √
P5. Bulk data manip.	× –	× –	× –	× –	√ ×	√ √	× –	× –
Reuse for explicit queries								
R1. Query parameters	–	–	√ ×	√ ×	√ ×	√ √	√ √	√ √
R2. Dynamic queries	–	–	√ ×	√ ×	√ ×	× –	√ √	√ √
R3. Modular queries	–	–	√ ×	√ ×	√ ×	× –	√ √	√ √
Concurrency								
C1. Transactions	×	√	√	√	√	√	√	√

√ √ = Feature supported and statically typed
√ × = Feature supported but not statically typed
× – = Feature not supported
√ = Feature supported
* = Partially supported
× = Not supported
– = Not applicable

Figura 3.4: Clasificación V: Integrating Programming Languages & Databases

- *Enfoque*. La mayoría de ellas no están orientadas a las opciones que un desarrollador debe considerar.
- *Definición*. En algunos casos las categorías o bien no se definen, o se hace de forma poco clara e incompleta.
- *Granularidad*. Las clasificaciones existentes consideran un solo nivel de categorización por lo que no presentan la granularidad necesaria para tener un panorama descriptivo de la realidad. De acuerdo con ello una categoría puede incluir mecanismos que deberían ser considerados de manera separada.
- *Comparación*. Las comparaciones presentadas se basan en productos particulares. De acuerdo con ello no reflejan las características de los mecanismos en los cuales estos se basan.

3.3. Propuesta de Clasificación

A los efectos de solucionar los problemas que contienen las clasificaciones existentes se desarrolló una nueva clasificación para mecanismos de persistencia [CCM+06]. Para ello se tuvo en cuenta, además de las clasificaciones antes mencionadas, todas las opciones detectadas (como parte del proceso de investigación del estado del arte) al momento de solucionar la persistencia de un sistema. Es así que se decidió incluir nuevas categorías no consideradas y tomar un criterio de definición de categorías basado en lo que se define

como la *perspectiva del desarrollador*. Esta perspectiva refiere a la técnica o técnicas percibidas por el desarrollador al momento de utilizar el mecanismo. A modo de ejemplo, si se considera la utilización de un mapeador objeto-relacional que se encuentra implementado como un generador de código y además el desarrollador no tiene contacto con el código generado, el mismo percibe estar trabajando simplemente con un mapeador. Esto implica que cada categoría definida representa una opción disponible para el desarrollador a la hora de persistir la información de su sistema. Más allá que las categorías estén claramente diferenciadas, en la práctica, un producto particular puede pertenecer a más de una de ellas. Esto se entiende como el hecho que el producto particular hace uso de una o más técnicas concretas (visibles a nivel de la perspectiva del desarrollador). La Figura 3.5 muestra la relación taxonómica de los mecanismos según la clasificación propuesta.

Como se puede observar se trata de una categorización en dos niveles que incluye tanto técnicas establecidas como ser acceso directo a bases de datos relacionales (una subcategoría de acceso directo a bases de datos) como también técnicas más recientes y de creciente popularidad (p.ej. generadores de código y mapeadores objeto-relacional). La existencia de un segundo nivel permite separar técnicas que más allá de compartir una misma técnica base (p.ej. en el caso de los mapeadores) tienen características diferenciadas (el caso de los mapeadores objeto-relacional y objeto-XML). Concretamente, la clasificación propuesta define categorías que contemplan: Mapeadores (mapeo de objetos a otro modelo de datos), Acceso Directo a Base de Datos (interacción directa con una base de datos), Lenguajes Orientados a Objetos (soporte del lenguaje para la persistencia de objetos), Orientación a Aspectos (implementación de persistencia a través de aspectos), Generadores de Código (generación del código para persistencia a través de herramientas) y Prevalencia (persistencia mediante *snapshots* y fuerte uso de memoria principal).

La clasificación no está compuesta solamente por la taxonomía presentada en la Figura 3.5, sino que incluye una descripción detallada de cada uno de los mecanismos presentes (Apéndice A) así como un análisis comparativo de estos (Apéndice B).

3.3.1. Estructura para Descripción de Mecanismos

La motivación para describir cada mecanismo de persistencia es la necesidad de presentar un esquema de solución (en un alto nivel de abstracción) a un problema surgido repetidamente, documentando la experiencia existente en el abordaje de dicho problema. Por dicho motivo, se optó por definir una estructura común para describir cada mecanismo de persistencia. Dicha estructura tiene elementos en común con las utilizadas para describir patrones de diseño en [GHJV95].

En concreto, para la exposición de los distintos mecanismos de persistencia se hará uso de la siguiente estructura:

- *Nombre del Mecanismo*. Nombre propuesto para la identificación del mecanismo en cuestión.
- *Intención*. Breve descripción del objetivo perseguido por el mecanismo.
- *Motivación*. Explicación de la necesidad que dio a lugar al surgimiento del mecanis-

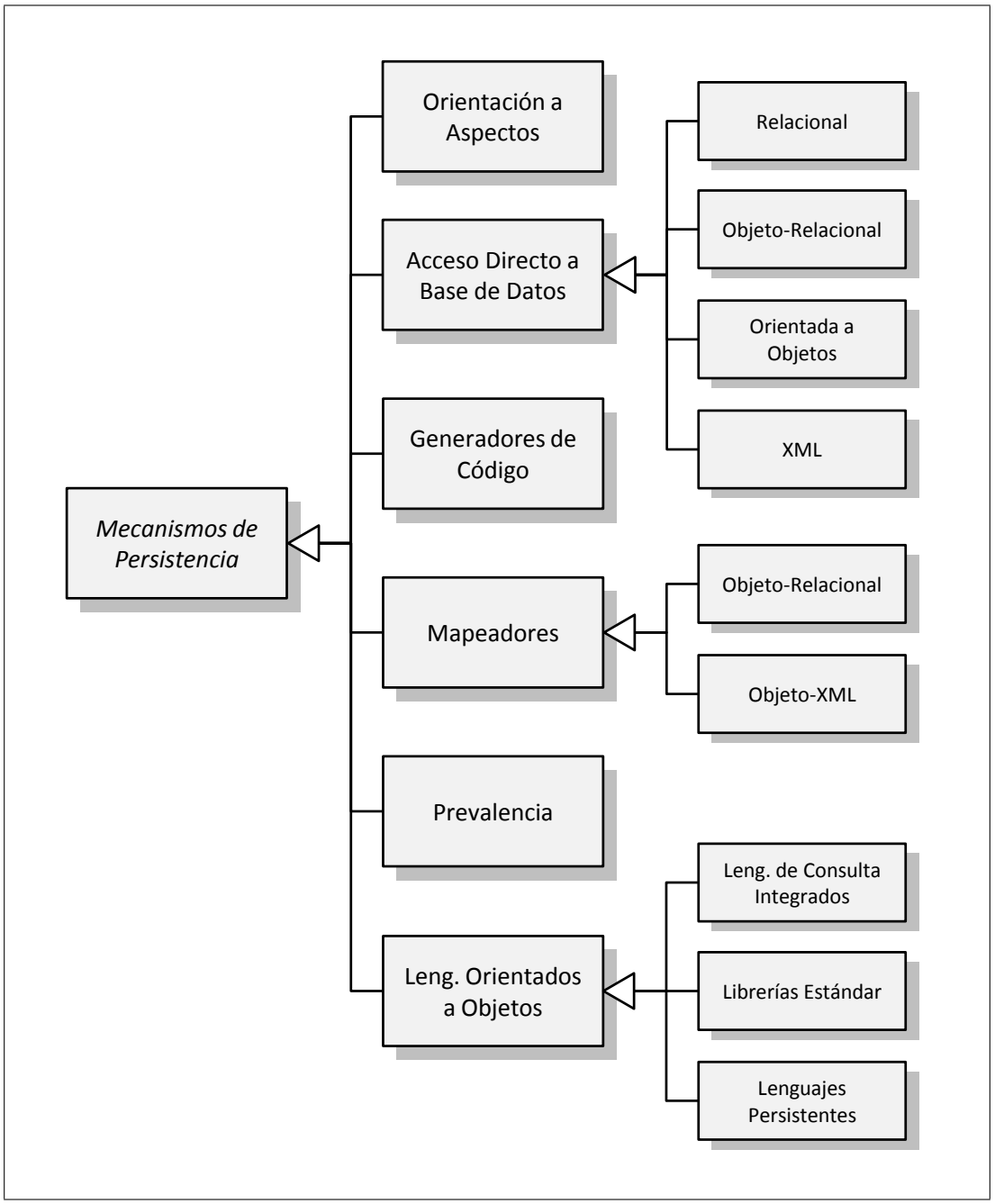


Figura 3.5: Clasificación propuesta para Mecanismos de Persistencia

mo.

- *Descripción.* Presentación de aspectos generales de su funcionamiento, principales características y definición de conceptos involucrados.
- *Contextos de Aplicación.* Descripción de los ambientes en los cuales es recomendable la aplicación del mecanismo. Se tienen en cuenta, entre otras cosas, plataformas, dominio de aplicación y requerimientos de *performance*.
- *Ventajas.* Enumeración de los puntos favorables de la aplicación del mecanismo.
- *Desventajas.* Enumeración de los puntos desfavorables de la aplicación del mecanismo.
- *Mecanismo Base.* En caso de aplicar, se refiere al mecanismo general que se especializa. Por tratarse de una clasificación taxonómica, el mecanismo especializado cumple con la descripción presentada en el mecanismo base.
- *Mecanismos Relacionados.* De existir mecanismos relacionados los mismos son enumerados en esta sección explicando además su relación.
- *Herramientas Representativas.* Se sugieren algunas herramientas representativas del mecanismo tanto a nivel académico como de la industria.

Nótese que la estructura anterior aplica únicamente a los mecanismos concretos (entiéndase como las hojas del árbol de categorías). En cuanto a lo que se denomina mecanismos base (aquellas categorías descendientes directas de la raíz "Mecanismos de Persistencia" que tienen categorías hijas propias) se realiza simplemente una descripción general.

3.3.2. Presentación de Mecanismos Considerados

A continuación se realiza un breve resumen de cada uno de los mecanismos involucrados en la taxonomía. La exposición completa de los mismos se encuentra en el Apéndice A.

Acceso Directo a Base de Datos

El mecanismo en cuestión implica el uso directo de la base de datos para implementar la persistencia del sistema. Esto último refiere al acceso a los datos utilizando por ejemplo una interfaz estandarizada en conjunto con algún lenguaje de consulta soportado directamente por el DBMS. Téngase en cuenta que no existe ningún tipo de *middleware* entre la aplicación y el DBMS utilizado.

- *Acceso Directo a Base de Datos Relacional.* Haciendo uso de una base de datos relacional como dispositivo de almacenamiento, es la intención fundamental proveer un mecanismo de acceso a datos que maneje el modelo relacional.

- *Acceso Directo a Base de Datos Objeto-Relacional.* Haciendo uso de una base de datos objeto-relacional como dispositivo de almacenamiento, es la intención fundamental proveer un mecanismo de acceso a datos que brinde un modelo de datos más rico que el relacional.
- *Acceso Directo a Base de Datos Orientada a Objetos.* Haciendo uso de una base de datos orientada a objetos como dispositivo de almacenamiento, es la intención fundamental proveer un mecanismo de acceso a datos que no implique una diferencia de paradigmas entre el nivel lógico y de persistencia.
- *Acceso Directo a Base de Datos XML.* Haciendo uso de una base de datos XML como dispositivo de almacenamiento, es la intención fundamental proveer un mecanismo de persistencia que permita almacenar los datos de los objetos en documentos XML.

Mapeadores

En esta categoría se incluyen aquellos mecanismos que se basan en la traducción bidireccional entre los datos encapsulados en los objetos de la lógica de un sistema orientado a objetos y una fuente de datos que maneja un paradigma distinto. El mapeador ha de lidiar con los diferentes problemas que se presentan al mapear los datos entre paradigmas. El conjunto de problemas que se derivan de estas diferencias recibe el nombre de *impedance mismatch.*

- *Mapeadores Objeto-Relacional.* La intención de este mecanismo radica en contar con un mecanismo para mapear los objetos de la lógica de un sistema orientado a objetos a una base de datos relacional. Debido a las diferencias entre la representación tabular de información y la encapsulación de los datos en objetos (problema conocido como *impedance mismatch objeto-relacional*) debe considerarse alguna estrategia (manual o automática) para poder resolver estas diferencias e integrar ambas tecnologías.
- *Mapeadores Objeto-XML.* Es la intención de este mecanismo permitir el almacenamiento de los datos contenidos en los objetos de la lógica en forma de documentos XML. El mapeo objeto-XML provee un medio por el cual los datos de negocio pueden ser visualizados en su forma persistida (incluso sin la necesidad de un DBMS), al mismo tiempo de que se facilita el intercambio de dichos datos con otros sistemas. A diferencia de los modelos como DOM y JDOM, los mapeadores objeto-XML considerados toman un enfoque centrado en los datos y no en la estructura del documento XML de forma tal que el desarrollador utiliza objetos de negocio que reflejan el contenido mismo de los documentos.

Generadores de Código

El surgimiento de numerosos *frameworks* que resuelven parte de la problemática del desarrollo de una aplicación empresarial y el creciente uso de patrones ha permitido el desarrollo de aplicaciones empresariales más robustas en menos tiempo. Sin embargo el programador aun se ve obligado a realizar tareas repetitivas. La idea de que el programador

debe concentrarse en el código que representa la lógica de negocio es la principal motivación para el surgimiento de los generadores de código. Concretamente, se trata de herramientas que basándose en *metadata* de un proyecto son capaces de generar código correcto, robusto y que aplica los patrones de diseño pertinentes. Este código generado se encarga de resolver parte de la problemática del sistema, en este caso la persistencia del mismo.

Orientación a Aspectos

La *Programación Orientada a Aspectos* (AOP) es un paradigma que permite definir abstracciones que encapsulen características que involucren a un grupo de componentes funcionales, es decir, que corten transversalmente al sistema (p.ej. seguridad, comunicación, replicación, etc.). En la programación orientada a aspectos, las clases son diseñadas e implementadas de forma separada a los aspectos requiriendo luego una fusión. Es intención de este mecanismo manipular concretamente la persistencia como un aspecto ortogonal a las funcionalidades, desacoplando el código correspondiente al resto del sistema. Otro objetivo de este mecanismo es a su vez permitir modularizar la persistencia para luego poder reutilizar el código generado.

Lenguajes Orientados a Objetos

La categoría de lenguajes orientados a objetos como mecanismo de persistencia implica resolver la persistencia de datos utilizando funcionalidades provistas por el propio lenguaje de programación.

- *Librerías Estándar.* La idea fundamental de este mecanismo yace en utilizar las funcionalidades básicas incluidas en la infraestructura de un lenguaje de programación orientado a objetos a fin de resolver la persistencia de un sistema. Se trata luego de una estrategia simple que evita la inclusión de *frameworks* u otras herramientas ajenas al lenguaje. Tomando como ejemplo el lenguaje de programación Java, se tienen librerías para el manejo de archivos que permiten persistir los datos de una aplicación en forma de texto plano, CSV (Comma Separated Value) o XML. Asimismo se pone a disposición del desarrollador la posibilidad de utilizar técnicas de serialización de objetos.

- *Lenguajes Persistentes.* El objeto de este mecanismo implica resolver de manera transparente la persistencia de datos haciendo uso de funcionalidades provistas por el lenguaje de programación, cumpliendo con los tres principios de persistencia ortogonal (o en su defecto, una relajación de los mismos): ortogonalidad de tipo, persistencia por alcance e independencia de persistencia. En concreto, dichos principios implican respectivamente que: (1) Cada objeto, independiente de su tipo, tiene el mismo derecho a ser persistido. (2) Se persisten solo aquellos objetos que son alcanzables desde la raíz de persistencia. (3) La introducción de persistencia no puede introducir cambios semánticos en el código fuente.

- *Lenguaje de Consulta Integrado.* Existen múltiples interfaces de acceso a datos que son utilizadas desde numerosos lenguajes de programación como método para admi-

nistrar y consultar los datos persistentes de un sistema. El uso de interfaces de este tipo tiene como consecuencia una disminución en la claridad de la solución. Esto debido no solo al agregado en el código fuente de llamadas a operaciones definidas por el API en particular, sino también por la inclusión, en forma ajena al lenguaje de programación, de lenguajes de consulta como ser SQL, OQL, HQL, etc. Tomando en cuenta lo anterior, el mecanismo propone la utilización de un lenguaje de programación que integre como parte del mismo un lenguaje de consulta y administración de datos persistentes.

Prevalencia

La prevalencia de objetos surge como una propuesta diferente a utilizar bases de datos para lograr la persistencia. La mayoría de los programas operan sobre datos en memoria principal, ya que de esa forma logran procesar los datos más rápido que en un escenario donde se trabaja directamente en memoria secundaria. Este mecanismo permite mantener las instancias de objetos que componen a un sistema en memoria principal, añadiendo en forma periódica, la persistencia del estado de los anteriores mediante técnicas de serialización. La persistencia del estado de los objetos se denomina *snapshot* de los objetos en memoria, y representa el último estado consistente de los datos. Asimismo, permite el manejo de transacciones y la capacidad de recuperar un estado consiste del sistema tras la caída del mismo.

3.3.3. Propiedades de la Clasificación

Se realizó una nueva clasificación que pretende subsanar las carencias detectadas en las clasificaciones presentadas en 3.2.

- *Completitud*. La clasificación propuesta pretende ser completa con respecto a los mecanismos detectados. Se puso énfasis no solo en destacar mecanismos conocidos en la industria sino también en incluir mecanismos emergentes (aquellos que por el momento solo existen en ámbitos académicos o no han logrado popularidad).
- *Enfoque*. Se presenta un enfoque diferente, centrado en la vista del desarrollador y que pretende facilitar la comprensión de la clasificación.
- *Definición*. Se propone una estructura bien definida (basada en la definición de patrones de diseño) para la descripción de las categorías.
- *Granularidad*. Se trata de una categorización de dos niveles de profundidad que disminuye el nivel de abstracción en relación al considerado por las clasificaciones existentes. De esta forma se permite acentuar las diferencias relevantes y asimismo agrupar características comunes.

Téngase en cuenta además que fue de sumo interés no basarse en plataformas ni lenguajes particulares al momento de definir las categorías. De esta forma se permitió identificar características a nivel de mecanismos y no de un producto particular.

3.4. Comparación entre Mecanismos

La descripción individual de cada mecanismo puede verse favorecida incluyendo una segunda vista de la clasificación que involucre un análisis comparativo de los mecanismos. Esto le permite al desarrollador obtener un mejor entendimiento global de las alternativas disponibles. Con este objetivo, se consideran criterios que permitan contrastar de manera organizada las diferencias y similitudes entre las distintas alternativas.

3.4.1. Criterios de Comparación

El conjunto de criterios definido está basado fuertemente en los criterios propuestos en [Tak05] (correspondientes a la Clasificación II presentada en 3.2.2).

A continuación se presentan la totalidad de los criterios a considerar así como también una breve definición de los mismos. Téngase en cuenta además la agrupación de criterios de la misma naturaleza.

Incidencia en la Lógica del Sistema

Los criterios de comparación pertenecientes a este grupo miden características que afectan directamente a la lógica del sistema.

- *Transparencia.* Refiere a una medida cuantitativa inversamente proporcional al esfuerzo invertido en tareas, cambios y agregados en cuanto a los recursos del proyecto (incluyendo código fuente y *medatada*), al agregar la cualidad de persistencia a los objetos de negocio de la aplicación.

- *Persistencia por Alcance.* Todo mecanismo requiere de una estrategia para indicar qué objetos han de ser persistidos. Usualmente se cuenta con un objeto denominado *raíz de persistencia* tal que todos aquellos objetos alcanzables desde él son persistentes. Dicho enfoque es denominado persistencia por alcance. De acuerdo con lo anterior, el criterio en cuestión refiere a la utilización o no de esta estrategia por parte del mecanismo.

- *Mecanismo de Consulta.* Refiere al mecanismo de recuperación de objetos provisto.

- *Manejo de Impedance Mismatch.* En el caso de existir *impedance mismatch*, se refiere al nivel en el que el desarrollador debe lidiar con este problema. El dominio de este criterio se define como: *manual* (el desarrollador es el encargado de solucionarlo), *automático-configurado* (el mecanismo asiste de forma automática basándose en *metadata* provista por el desarrollador) y *automático*.

- *Soporte Transaccional.* Indica a qué nivel el mecanismo cubre las propiedades ACID.

Resultado de su Aplicación

El presente grupo de criterios de comparación involucra a aquellos que se refieren a características propias del resultado de la aplicación del mecanismo.

- *Performance.* Refiere al grado general de *performance* obtenido a nivel de la solución a través de la aplicación del mecanismo.
- *Soporte de Evolución del Esquema.* Indica qué facilidades brinda el mecanismo para lidiar con la evolución del esquema de datos subyacente.
- *Escalabilidad.* Refiere a la habilidad de aumentar la capacidad del sistema a medida que los requerimientos aumentan.
- *Mantenibilidad de la Solución.* Medida de cuán mantenible resulta la solución al utilizar el mecanismo.

Incidencia en el Proyecto

Los siguientes criterios se refieren a factores que inciden a nivel de proyecto al utilizar un mecanismo de persistencia.

- *Productividad.* Mide cuán productiva es la aplicación del mecanismo para el desarrollador a fin de solucionar la persistencia del sistema.
- *Tecnologías Dependientes.* Tecnologías relevantes de las cuales el mecanismo depende.
- *Porte del Sistema.* Se refiere a los diferentes tipos de sistemas, desde el punto de vista de su tamaño, para los cuales se recomienda aplicar el mecanismo.

Vista del Mercado

Este grupo de criterios refleja características de un mecanismo de persistencia desde la perspectiva del mercado.

- *Existencia de Productos.* Medida de la variedad de productos existentes en el mercado que utilicen el mecanismo correspondiente.
- *Popularidad en el Mercado.* Refiere al nivel de aceptación general del mecanismo en el mercado.
- *Madurez.* Nivel de madurez del mecanismo.
- *Nivel de Estandarización.* Medida de la existencia de estándares relacionados con el mecanismo en cuestión.

3.4.2. Evaluación de Criterios

En base a los criterios definidos anteriormente se presenta la evaluación de un subconjunto de los mismos para (1) Acceso Directo a Base de Datos Relacional (2) Acceso Directo a Base de Datos Orientada a Objetos (3) Mapeadores Objeto-Relacional y (4) Generadores de Código. Téngase en cuenta que los criterios seleccionados son aquellos que se consideran de mayor interés. Nótese además que la naturaleza de muchos de ellos implicaría la definición de métricas complejas y una experimentación exhaustiva a fin logar una evaluación objetiva. De acuerdo con esto último y considerando el alcance del proyecto, la siguiente comparación resulta de la recolección de información disponible y opiniones de terceros. Se entiende sin embargo, que el resultado de estas evaluaciones son de valor al momento de contrastar los diferentes mecanismos de persistencia.

Para una evaluación más completa de criterios a nivel de mecanismos refiérase al Apéndice B.

Incidencia en la Lógica del Sistema

La evaluación de Transparencia y Mecanismo de Consulta se presenta en los Cuadros 3.1, y 3.2 respectivamente.

Cuadro 3.1: Evaluación de Transparencia

Mecanismo	*Transparencia*
Acceso Directo a BD Relacional	Por definición del mecanismo los datos se acceden directamente desde la lógica orientada a objetos mediante SQL. Esto último implica que no se oculta de ninguna forma la naturaleza relacional de los datos. De acuerdo con esto el nivel de transparencia es nulo.
Acceso Directo a BD Orientada a Objetos	Debido a que la base de datos subyacente trabaja directamente con objetos se puede obtener una solución más transparente en comparación a los demás mecanismos de acceso directo. Nótese igual la posible existencia de consultas embebidas, afectando negativamente la transparencia.

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Transparencia*
Mapeadores Objeto-Relacional	Los mapeadores suelen tener como objetivo la transparencia por lo que su nivel tiende a ser alto. Dicho objetivo se suele lograr mediante el uso de APIs sencillos y la especificación de *metadata* de mapeo necesaria.
Generadores de Código	Depende del Generador.

Cuadro 3.2: Evaluación de Mecanismo de Consulta

Mecanismo	*Mecanismo de Consulta*
Acceso Directo a BD Relacional	Las consultas se realizan mediante SQL.
Acceso Directo a BD Orientada a Objetos	Dependiendo de la interfaz de acceso podrían tenerse distintas formas de consultar la base. De manera general se distinguen lenguajes de consulta como JDOQL u OQL y las llamadas consultas nativas (realizadas de forma programática).
Mapeadores Objeto-Relacional	Suelen proveer lenguajes de consulta propietarios o estandarizados a nivel de objetos (p.ej. OQL) al mismo tiempo que es posible realizar consultas directas a la base de datos relacional. En algunos casos se incluye un API de formulación de consultas.
Generadores de Código	Depende del Generador.

Resultado de su Aplicación

La evaluación de Performance y Mantenibilidad de la Solución se presenta en el Cuadros 3.3 y 3.4 respectivamente.

Cuadro 3.3: Evaluación de Performance

Mecanismo	*Performance*

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Performance*
Acceso Directo a BD Relacional	Muy buen nivel de desempeño debido a que el acceso se realiza de forma directa sin tener que pasar por ningún tipo de middleware. Nótese que se supone la existencia de un controlador y un DBMS eficientes para brindar acceso.
Acceso Directo a BD Orientada a Objetos	Ídem anterior.
Mapeadores Objeto-Relacional	No se comportan bien en procesos de escritura por lotes. La caída en la *performance* en comparación al acceso directo a una base de datos relacional se debe al *overhead* producido por el mapeo entre ambos paradigmas. Suelen comportarse bien en aplicaciones `READ-CENTRIC` o que sigan un ciclo de vida tipo `READ-MODIFY-WRITE`.
Generadores de Código	Depende del Generador.

Cuadro 3.4: Evaluación de Mantenibilidad de la Solución

Mecanismo	*Mantenibilidad de la Solución*
Acceso Directo a BD Relacional	Es complicado diseñar soluciones mantenibles ya que el código necesario para el acceso a datos se entrelaza con el código de la lógica de negocios.
Acceso Directo a BD Orientada a Objetos	Alta mantenibilidad. El enfoque puramente de objetos en la lógica y la ausencia del *impedance mismatch* facilita el mantenimiento.
Mapeadores Objeto-Relacional	Es ciertamente más mantenible que una estrategia de acceso directo a una base de datos. El enfoque puramente de objetos en la lógica facilita el mantenimiento.

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Mantenibilidad de la Solución*
Generadores de Código	Se producen soluciones que por la naturaleza del mecanismo son altamente mantenibles.

Incidencia en el Proyecto

La evaluación de Productividad se presenta en el Cuadro 3.5.

Cuadro 3.5: Evaluación de Productividad

Mecanismo	*Productividad*
Acceso Directo a BD Relacional	Baja.
Acceso Directo a BD Orientada a Objetos	Alta.
Mapeadores Objeto-Relacional	Alta.
Generadores de Código	Alta.

Vista de Mercado

La evaluación de Madurez se presenta en el Cuadro 3.6.

Cuadro 3.6: Evaluación de Madurez

Mecanismo	*Madurez*
Acceso Directo a BD Relacional	Presenta un gran nivel de madurez debido a su temprano surgimiento, su éxito logrado en el correr de los años y la amplia investigación y desarrollo en el campo.
Acceso Directo a BD Orientada a Objetos	Presenta creciente nivel de madurez.
Mapeadores Objeto-Relacional	Moderada, ya que en los últimos seis años se ha investigado y desarrollado mucho en el tema.

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Madurez*
Generadores de Código	Se trata de un campo que no ha cobrado interés sino hasta los últimos años. De acuerdo con ello no se considera un mecanismo maduro.

3.5. Resumen

En base al estudio de clasificaciones existentes de mecanismos de persistencia, se realizó una nueva clasificación que pretende subsanar las carencias detectadas en las mismas. Concretamente se desarrolló una clasificación que pretende ser completa, con mecanismos bien definidos y orientada al desarrollador. Se consideró como principal objetivo lograr una visión de los mecanismos disponibles que aportara al desarrollador en la comprensión global de las alternativas existentes. Además, fue de sumo interés (a diferencia de algunas de las clasificaciones existentes) el mantenerse en un nivel abstracto a la hora de definir las categorías. Esto es, no basarse en plataformas ni lenguajes particulares, permitiendo así identificar las características de un mecanismo y no de un producto particular. Se logró una definición clara y concisa de cada mecanismo utilizando una estructura común que permitiese incluir toda la información que se considera relevante para caracterizarlo. Con respecto a la completitud, se puso énfasis no solo en destacar mecanismos conocidos en la industria sino también en incluir mecanismos emergentes (aquellos que por el momento solo existen en ámbitos académicos o no han logrado popularidad). Para un mejor entendimiento global de las alternativas disponibles se complementó la clasificación realizada con un análisis comparativo de mecanismos. Para ello se definieron y evaluaron un conjunto de criterios que permitieron contrastar de manera organizada las diferencias y similitudes.

Los resultados tanto del análisis de los mecanismos (ventajas, desventajas y contextos de aplicación) como también de las comparaciones realizadas sugieren que no hay un mecanismo *one-size-fits-all* en lo que a persistencia se refiere. La conveniencia de uno u otro depende de las particularidades del proyecto para el cual se utilizará. Mas allá de esto, de todos los mecanismos presentados, solo algunos son aplicables en la realidad de hoy en día dada la inmadurez, el carácter académico o desuso del resto.

Capítulo 4

Productos: Mapeadores Objeto-Relacional

4.1. Introducción

La definición de una clasificación de mecanismos de persistencia clara y completa ayuda a la comprensión del amplio abanico de alternativas existentes. Al mismo tiempo las comparaciones realizadas permiten identificar las principales diferencias entre ellas. Si bien las características de un proyecto de desarrollo de software pueden sugerir un conjunto de mecanismos aplicables, es necesaria la selección de un producto concreto dentro de ese mecanismo. La necesidad de comparaciones y análisis a nivel de producto es real y se puede ver como una continuación lógica del estudio a nivel de mecanismos. Nótese que a este nivel existe un conjunto de características diferentes a las consideradas a nivel de mecanismos que condicionan la conveniencia de un producto para un proyecto en particular. Estas nuevas características no fueron incluidas en las evaluaciones realizadas para mecanismos por no ser aplicables (p.ej. licencia con la que se distribuye el producto, plataformas en las que puede ser utilizado, etc.). Es así que se pretende contextualizar el presente capítulo dentro del marco de productos particulares.

Si bien es de interés contar con evaluaciones y comparaciones cruzadas de productos de diferentes categorías, la vastedad de productos existentes hacen de ésta una tarea muy ardua. Es por ello que se decide restringir este estudio únicamente a tres productos pertenecientes a uno de los mecanismos. De acuerdo con lo anterior se propone el estudio del mecanismo Mapeadores Objeto-Relacional, siendo éste muy utilizado en la actualidad.

Con el objetivo de comparar se definen un conjunto de criterios de comparación para productos. Algunos de éstos son aplicables a cualquier producto y el resto son específicos para Mapeadores Objeto-Relacional. Téngase en cuenta que un subconjunto de estos criterios serán evaluados para los productos Hibernate, JPA/TopLink y LLBLGen Pro. A efectos de llevar a cabo la antedicha evaluación se desarrolla un caso de estudio. El mismo considera la realidad presentada en el artículo "SAD del Subsistema de Reservas del Sistema de Gestión Hotelera" [PV03] que describe la arquitectura de un sistema de gestión de una cadena hotelera. En particular, el caso de estudio se centra en un subsistema del

mismo, el Subsistema de Reservas. Por más detalles refiérase al Apéndice D.

Debido al marco del capítulo y a fin de introducir conceptos y características relevantes, se presenta en la Sección 4.2 la definición completa del mecanismo seleccionado. En la Sección 4.3 se presenta una breve descripción de los productos que incluye las características básicas que los diferencian del resto. La Sección 4.4 define los diferentes conjuntos de criterios de comparación a ser utilizados, seguidos de la evaluación de un subconjunto de ellos para los productos considerados. El capítulo finaliza con un resumen del mismo en la Sección 4.5.

4.2. Mapeadores Objeto-Relacional

De acuerdo con la estructura para descripción de mecanismos de persistencia presentada en 3.3.1, se expone la categoría de Mapeadores Objeto-Relacional.

Intención

Contar con un mecanismo para mapear los objetos de la lógica de un sistema orientado a objetos a una base de datos relacional.

Motivación

Hoy en día muchos desarrolladores trabajan en aplicaciones empresariales que crean, manejan y almacenan información estructurada, la cual es compartida entre muchos usuarios. Tradicionalmente la mayoría de estas aplicaciones involucran el uso de un DBMS basado en SQL. Más aun, una porción de ellas son completamente dependientes del modelo relacional en el propio núcleo de su lógica de negocio.

En los últimos años la programación orientada a objetos, con el surgimiento de lenguajes como Java y C#, ha tomado su lugar en la comunidad de desarrollo de software como el paradigma por excelencia. A pesar de que los desarrolladores se encuentran convencidos de los beneficios que este paradigma ofrece, la mayoría de las empresas están atadas a inversiones de largo plazo en costosos sistemas de bases de datos relacionales. Además, suelen existir datos legados (en bases de datos relacionales) con los cuales los nuevos sistemas orientados a objetos deben interactuar.

Debido a las diferencias entre la representación tabular de información y la encapsulación de los datos en objetos debe considerarse alguna técnica para poder resolver estas diferencias e integrar ambas tecnologías. Tradicionalmente la importancia de este problema ha sido subestimada. Mientras los programadores atribuían la culpa al modelo relacional, los profesionales en datos culpaban al paradigma de objetos. Es así que se motiva el surgimiento de soluciones automatizadas que toman el nombre de Mapeadores Objeto-Relacional.

Es a través de un mecanismo de este tipo que se puede contar tanto con los beneficios de

un lenguaje orientado a objetos como los de una tecnología madura como ser la relacional.

Descripción

El mecanismo en cuestión se basa en el uso de un componente que se encargue del mapeo de objetos de negocio a un modelo relacional y viceversa (Figura 4.1). Al momento de mapear objetos a una base relacional se hace presente un problema que surge de las diferencias existentes entre ambos paradigmas. Este problema es denominado *impedance mismatch objeto-relacional* y se refiere a las dificultades que se presentan al mapear un objeto con identidad, que encapsula datos y comportamiento, a un modelo relacional que solo se concentra en los datos. Más aun, los tipos de datos básicos utilizados por ambos paradigmas son distintos. La solución a este último problema es directa ya que simplemente se mapea un tipo básico de un paradigma a uno del otro. Otro punto a tener en cuenta es que ambos paradigmas se basan en principios diferentes: la orientación a objetos se basa en principios del desarrollo de software mientras que el paradigma relacional se basa en principios matemáticos. De acuerdo con todo lo expuesto anteriormente, resulta lógico que deba tomarse alguna política de solución.

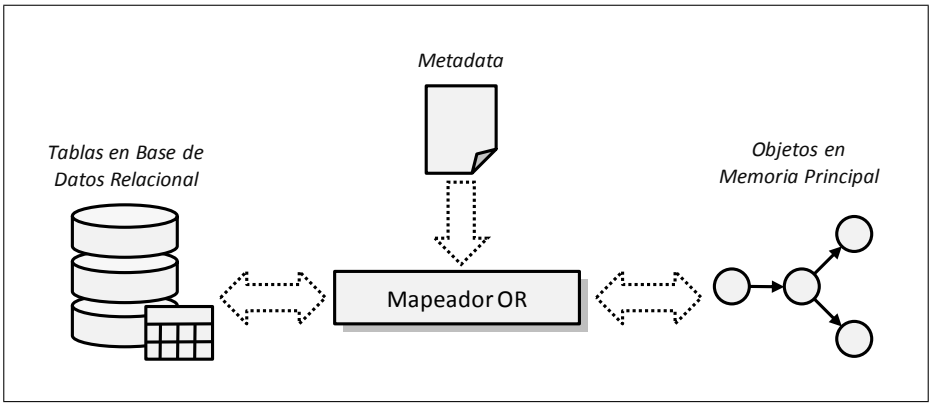


Figura 4.1: Esquema de un Mapeador Objeto-Relacional

Para lograr el mapeo el mapeador se vale de *metadata* de mapeo provista por el desarrollador. La misma especifica el mapeo de: clases a tablas, atributos a columnas, relaciones a claves foráneas, etc. Además de lo anterior, el desarrollador es encargado de seleccionar diferentes estrategias de funcionamiento de forma explícita o por omisión. Mas allá de que la forma en que los objetos son mapeados a la base es particular a la implementación del mecanismo, existen estrategias básicas que son comúnmente aplicadas por los mismos.

Contextos de Aplicación

- Aplicaciones cuyas entidades siguen un ciclo de vida tipo `READ-MODIFY-WRITE`. Este ciclo implica que se obtiene un objeto a partir del mapeo de los datos almacenados en una base de datos relacional, se hacen modificaciones al mismo y luego es escrito nuevamente a la base. Un ejemplo de aplicaciones que responden a este patrón son las aplicaciones Web.

- Aplicaciones `READ-CENTRIC`. Es decir, aquellas que pasan la mayor parte del tiempo realizando operaciones de lectura de información.
- Contextos en el que se deba interactuar con una base de datos relacional desde un ambiente orientado a objetos.

Ventajas

- La utilización de un mecanismo de este tipo permite reducir radicalmente la cantidad de código necesario (manejado por el desarrollador) para lograr la persistencia. De acuerdo con ello se logra un incremento en la productividad.
- La aplicación es más fácil de mantener, por lo menos comparando esta solución a una en la que se accede directamente a la base de datos utilizando un lenguaje de consulta embebido (p.ej. SQL).
- Permite que el desarrollador pueda mantener una mentalidad orientada a objetos al momento de programar.
- La utilización de un cache de objetos, normalmente presente en los diferentes mapeadores, afecta positivamente la performance.

Desventajas

- Requiere agregar y mantener *metadata* sobre las entidades. Esta *metadata* contiene principalmente las particularidades del mapeo de cada entidad.
- Se pierde el control de cómo se lleva a cabo el proceso de persistencia. Esto es, se dejan al mapeador los detalles específicos de cómo las entidades son persistidas.
- Se dificulta el *tuning* de consultas, ya que requiere conocimientos de cómo se lleva a cabo el mapeo (información que no siempre está disponible).
- La herramienta de mapeo utilizada puede impactar de forma negativa en la *performance* del sistema, sobre todo si se trata de una aplicación que se dedique la mayor parte del tiempo a realizar operaciones tipo `WRITE`. De lo anterior se tiene que este mecanismo no es una opción razonable para sistemas con fuertes requerimientos de *performance* (p.ej. sistemas de tiempo real, aplicaciones militares, bolsa de valores, etc.).
- Existen consultas (complejas generalmente) en las que no se puede optar por una solución programada exclusivamente en el lenguaje orientado a objetos utilizado ya que de ser así, se agregaría un importante *overhead* al procesamiento de las mismas. Para resolver lo anterior suele recurrirse a un lenguaje de consulta como ser SQL (teniendo que lidiar con el modelo relacional) o a un lenguaje de consulta provisto por el mapeador (manteniéndose a nivel de objetos).
- A menos que el lenguaje de consulta provisto por la herramienta sea un lenguaje estandarizado (p.ej. SQL, OQL, etc.), el código de consulta queda ligado a la herramienta particular.

- Se requiere de lo que se denomina *shadow information* para poder mantener aquella información de los objetos que va mas allá de los datos de dominio. A modo de ejemplo: una clave primaria que provee la identidad del objeto (no tiene por qué tener ningún significado en el negocio) o un campo que mantenga la versión para asistir en chequeos de integridad.

- El uso de un *cache* de objetos implica mayores requerimientos de memoria y probablemente problemas de consistencia entre los datos cacheados y los persistidos en la base de datos (sobre todo si existe más de una aplicación accediendo a esta).

Mecanismo Base

- Mapeadores (Sección A.2).

Mecanismos Relacionados

- *Acceso Directo a Base de Datos Relacional* (Sección A.1.1). El vínculo yace en el hecho de que un mapeador cumple la tarea de traducir los datos contenidos en objetos a datos en una base de datos relacional, permitiendo de esta forma obviar el *impedance mismatch* existente entre ambos paradigmas. Para lograr este cometido, normalmente el mapeador hace uso del mecanismo de acceso directo a una base de datos de forma transparente para el desarrollador.

- *Mapeadores Objeto-XML* (Sección A.2.2). A pesar de que el destino de los datos es distinto (documentos XML), muchos de los problemas básicos del mapeo se hacen presentes de igual forma (aunque la solución a los mismos sea muy diferente).

Herramientas Representativas

- Hibernate

  - Es una solución de código abierto para el mapeo objeto-relacional bajo la plataforma Java. La misma se adecua al proceso de desarrollo ya sea comenzando desde cero o partiendo de una base existente. Hibernate puede ser utilizado tanto para aplicaciones basadas en Java SE como en Java EE.
  - `http://www.hibernate.org`

- Java EE 5 y EJB 3.0

  - El lanzamiento de EJB 3.0 provee una nueva y simplificada API para el desarrollo de aplicaciones de porte empresarial. Dicha API tiene como objeto disminuir el costo de desarrollo y se trata concretamente de una simplificación de versiones anteriores de la especificación.
  - `http://java.sun.com/javaee/technologies/javaee5.jsp`

- TopLink

  - Paquete de Oracle para el mapeo objeto-relacional bajo Java. Asimismo ofrece además de soporte para JAXB, mapeos objeto-XML y un completo *framework* de consulta que permite expresiones orientadas a objetos, QBE, EJB QL, SQL y procedimientos almacenados.
  - `http://www.oracle.com/technology/products/ias/toplink`

- LLBLGen Pro
  - Se trata de un producto para .NET que permite la generación de la capa de mapeo objeto-relacional para un sistema valiéndose de un esquema de base de datos existente.
  - `http://www.llblgen.com`

4.3. Productos Considerados

La selección de los tres productos particulares se basa fuertemente en los resultados de una encuesta realizada sobre 15 empresas nacionales en cuanto a qué solución de mapeo utilizan en sus desarrollos. Asimismo se tomaron en cuenta los intereses propios del grupo. Concretamente los tres productos seleccionados son: Hibernate y JPA/TopLink Essentials para la plataforma Java y LLBLGen Pro para .NET. El primero, Hibernate, es elegido por ser el producto líder en el mercado a nivel mundial (y nacional como fue indicado por la encuesta antes mencionada) en lo que refiere a mapeadores objeto-relacional. Se propone TopLink como otro de los productos a evaluar debido a: ser un producto de la empresa Oracle que cuenta con una gran trayectoria en el área de persistencia, implementar la especificación JPA de Sun que pretende ser el estándar *de facto* para la utilización de mapeadores objeto-relacional en la plataforma Java, ser la implementación de referencia de dicho estándar y ser el mapeador incluido en el proyecto Glassfish (proyecto código abierto que pretende ofrecer un servidor de aplicaciones Java EE de calidad empresarial). La elección de LLBLGen Pro pretende introducir el factor .NET a la ecuación, siendo este producto el de mayor uso para esta plataforma (según la información recabada en la encuesta). Mas allá de su popularidad LLBLGen Pro es de interés por tratarse no solo de un mapeador sino también de un generador de código, por lo cual presenta particularidades que contrastan con las otras soluciones.

4.3.1. Hibernate 3

Hibernate [Hib] es un mapeador objeto-relacional gratuito y de código libre para la plataforma Java. Uno de sus propósitos fundamentales es de librar al desarrollador de una significativa cantidad de tareas relacionadas con la persistencia de datos. Hibernate se adapta al proceso de desarrollo ya sea creando un nuevo esquema de base de datos o haciendo uso de uno ya existente. No solo se ocupa del mapeo de clases Java a tablas en la base de datos (y de tipos Java a tipos SQL), sino que también ofrece un lenguaje de consulta propio y orientado a objetos (HQL) que reduce el tiempo de desarrollo. De otra forma, dicho tiempo de desarrollo debería ser empleado en realizar el manejo manual de JDBC y SQL. Hibernate se dedica a generar las sentencias SQL correspondientes y evita

que el desarrollador tenga que hacer la conversión manual a objetos. De esta manera la aplicación se mantiene portable a todos los DBMSs basados en SQL.

En vez de la gran cantidad de clases y propiedades de configuración requeridas por otras soluciones de persistencia (como EJBs), Hibernate necesita un único archivo de configuración y un documento de mapeo por cada tipo de objeto a ser persistido. El archivo de configuración puede especificarse como un archivo de propiedades clave-valor o mediante XML. Alternativamente, se podría realizar la configuración de manera programática. En cuanto a los archivos de mapeo (en formato XML), los mismos pueden llegar a ser muy pequeños dejando que la herramienta defina el resto. Opcionalmente se podría proveer más información como por ejemplo un nombre alternativo para determinada columna de cierta tabla.

Al hacer uso de otro *framework* de persistencia la aplicación desarrollada podría quedar fuertemente acoplada al mismo. Hibernate no crea esta dependencia adicional; los objetos persistentes no tienen que heredar de una clase particular de Hibernate u obedecer alguna semántica específica. Asimismo no se requiere ningún tipo especial de contenedor Java EE para funcionar. Hibernate puede ser utilizado en cualquier entorno de aplicación (ya sea en una aplicación *standalone* o en el contexto de un servidor de aplicaciones).

4.3.2. Java Persistence API (JPA) / TopLink Essentials

La versión 5 de la plataforma Java Enterprise Edition (Java EE) [Jav] introduce, como parte de la especificación de EJB 3.0, Java Persistente API (JPA). Se trata de una especificación que simplifica la persistencia de objetos java que no presentan ninguna particularidad necesaria para la persistencia de los mismos (estos reciben el nombre de POJOs - Plain Old Java Objects). JPA pone énfasis en definir la *metadata* mínima necesaria para el mapeo de objetos Java a una base de datos relacional. Su objetivo fundamental es el de convertirse en el estándar por excelencia para la utilización de un mapeador objeto-relacional dentro de la plataforma

El estándar en cuestión es el resultado del trabajo de un grupo de expertos en el área quienes tomaron en consideración ideas de *frameworks* lideres del mercado y otras especificaciones existentes como ser: Hibernate, Oracle TopLink, Java Data Objects (JDO), entre otros. La intención detrás de su creación es que el mismo sea implementado por diferentes proveedores logrando así que diferentes productos ofrezcan una misma interfaz cuya semántica pueda ser asumida por las aplicaciones.

Dos de las características más innovadoras que distinguen a JPA de otras interfaces existentes son: la utilización intensiva de *annotations* para la definición de *metadata* (a pesar de que es posible utilizar XML para este mismo objetivo) y de *configuración por omisión*. Una *annotation* es un tipo especial dentro del lenguaje de programación Java que permite agregar información a diferentes elementos como ser: clases, métodos, atributos y variables. La información contenida en una *annotation* puede ser conservada en tiempo de ejecución, permitiendo a los diferentes *frameworks* interesados consultar esta información a través de una API especializada.

Debido a que JPA es solamente una especificación, para solucionar la persistencia de

un sistema se debe decidir qué implementación de dicha especificación ha de usarse. Las diferentes implementaciones son denominadas (dentro de la especificación) *proveedores de persistencia*. El proveedor de persistencia particular seleccionado es el provisto por Oracle en su producto TopLink Essentials. TopLink Essentials es parte del servidor de aplicaciones Glassfish que pretende ser un servidor de aplicaciones de código abierto y calidad empresarial. Téngase en cuenta que TopLink Essentials es la versión gratuita del producto TopLink de Oracle y que además es la implementación de referencia para la especificación JPA.

Las diferentes propiedades de la solución de persistencia obtenida es el resultado del par JPA (especificación), TopLink Essentials (implementación). Todas las características que se refieran a la interacción con el mapeador y la definición de la *metadata* de mapeo están condicionadas a la especificación JPA. Por otro lado, otras características como ser *performance* y detalles de implementación que no son referidos por la especificación dependen totalmente del proveedor de persistencia seleccionado.

4.3.3. LLBLGen Pro

LLBLGen Pro [LLB] es un mapeador objeto-relacional comercial para la plataforma .NET. La definición del mapeo se realiza en un diseñador que no necesita de un IDE para su ejecución. Una vez definido el mapeo, se genera el código correspondiente a la capa de acceso a datos, sobre la cual se accederá a los objetos persistidos. La utilización LLBLGen Pro apunta a incrementar la productividad del desarrollo de una aplicación, permitiendo al desarrollador concentrarse en la lógica del negocio.

El diseño del mapeo se realiza de forma sencilla, definiendo distintos tipos de entidades y relaciones, a partir de un esquema existente de la base de datos. LLBLGen Pro Soporta la definición de esquemas para varios motores de base de datos como ser SQL Server, Oracle, PostgreSQL y MySQL. En lo que refiere a la generación de código, el mismo podrá ser realizado en los lenguajes de programación C# o VisualBasic .NET. Una vez generado el código, se podrá persistir los objetos de forma simple y se podrán realizar consultas mediante un mecanismo de filtros, sin necesidad de embeber sentencias SQL en el código.

4.4. Comparación entre Productos

4.4.1. Criterios de Comparación

Dentro de los criterios de comparación a utilizar entre productos se distinguen tres grandes grupos. Estos grupos están compuestos por: (1) criterios específicos al mecanismo de persistencia considerado, (2) criterios aplicables a productos de cualquier mecanismo y por último (3) criterios ya considerados a nivel de mecanismos que son evaluados nuevamente a la luz del nuevo contexto (a nivel de productos). A continuación se presentan algunos de los criterios identificados para cada uno de estos grupos, incluyendo una breve definición de aquellos que no hayan sido introducidos en la comparación de mecanismos. Para la lista completa de los mismos refiérase a la Sección E.1.

Criterios a Nivel de Mapeador

En base a los trabajos publicados en [CI05], [Sin06] y [Yal04], se describen un conjunto de criterios de comparación para mapeadores objeto-relacional.

- *Construcción Manual de SQL.* Indica si el desarrollador debe lidiar manualmente, de manera obligatoria, con la construcción de sentencias SQL para acceder y manipular los datos.
- *Consultas Dinámicas.* Explicita si el mapeador en cuestión brinda soporte para la construcción de consultas en tiempo de ejecución.
- *Creación Automática de Esquema.* Se refiere a si el producto provee la funcionalidad de generar de forma automática el esquema de base de datos al que se mapean los objetos persistentes.
- *DBMSs Soportados.* Conjunto de motores de base de datos que son soportadas por el mapeador en cuestión.
- *Lenguaje de Consulta OO.* Indica si el mapeador provee un lenguaje de consulta orientado a objetos.
- *Relaciones Soportadas.* Se explicitan los tipos de relaciones soportadas entre objetos. Las mismas pueden ser 1-1, 1-n o m-n.
- *Soporte para Cascading.* Refiere a si el mapeador permite especificar la propagación de operaciones de persistencia a lo largo de sus asociaciones.
- *Proceso/Generación de Código.* Refiere a si el uso del mapeador requiere en algún momento de generación y/o el procesamiento de código fuente.

Criterios a Nivel de Producto

Esta sección presenta una colección de criterios propios de un producto y que son independientes del mecanismo considerado.

- *Última Versión.* Se refiere a la última versión estable del producto. Este criterio es de interés debido a que la versión es un indicador de la madurez y estabilidad del mismo.
- *Licencia.* La licencia bajo la cual se distribuye el producto.
- *Plataformas.* Lista de las plataformas bajo las cuales el producto puede ser utilizado. Por plataforma se entiende el entorno necesario para la utilización del producto.
- *Curva de Aprendizaje.* Esfuerzo requerido para la comprensión y aprendizaje de la forma de uso del producto. Esta medida es importante en lo que refiere a la capacitación de los desarrolladores en el caso de no tener experiencia con el producto.

Criterios a Nivel de Mecanismo

En este grupo no se incluyen todos los criterios considerados a nivel de mecanismos, sino que solo se consideran aquellos que pueden aportar a la diferenciación de los productos. En particular los criterios de interés son: Performance, Transparencia, Mecanismo de Consulta, Soporte para Evolución de Esquema, Soporte Transaccional y Popularidad.

4.4.2. Evaluación de Criterios

Debido a la naturaleza simple y directa del resultado de la evaluación de muchos de los criterios definidos anteriormente, se presenta a continuación un conjunto de los mismos en forma tabular. Asimismo se expone parte de la evaluación de Performance y Transparencia. Téngase en cuenta que para la evaluación de estos dos últimos criterios se utilizó el caso de estudio presentado en el Apéndice D. La exposición completa de la evaluación de criterios de comparación a nivel de productos se encuentra en la Apéndice E.2.

Cuadro Comparativo de Mapeadores

El Cuadro 4.1 permite contrastar las diferencias de los mapeadores comparados. Se consideran en la misma aquellos criterios que ameritan una representación simple en cuanto a los resultados de su evaluación.

Cuadro 4.1: Evaluación de Criterios para Productos

Criterio	**Hibernate**	**JPA/TopLink**	**LLBLGen Pro**
Nivel de Productos			
Ultima Versión	v3.2.1	v2.0b41	v2.0
Licencia	LGPL	CDDL	Comercial
Plataformas	Múltiples (Java)	Múltiples (Java)	Múltiples (.NET)
Nivel de Mapeadores			
Construcción Manual de SQL	✗	✗	✗
Consultas Dinámicas	✓	✓	✓
Creación Automática de Esquema	✓	✓	✗
DBMSs Soportados	JDBC-compliant	JDBC-compliant	SQL Server, Oracle, PostgreSQL, Firebird, DB2 UDB, MySQL, MS Access
Lenguaje de Consulta OO	✓(HQL)	✓(JPQL)	✗
Proceso/Generación de Código	✗	✗	✓
Relaciones Soportadas	{1-1, 1-n, m-n}	{1-1, 1-n, m-n}	{1-1, 1-n, m-n}
Soporte para *Cascading*	✓	✓	✓

Performance

La evaluación de la performance de un mapeador objeto-relacional es un tema de controversia. Uno de los problemas que se encuentra es la no existencia de un *benchmark*

estándar aceptado por los diferentes proveedores. A pesar de existir intentos de desarrollar un *benchmark* de estas características que permita comparar, de forma objetiva, el desempeño de los diferentes mapeadores, los profundos intereses comerciales detrás de dicho desarrollo han impedido su surgimiento.

Propuesta

Es en este contexto que se elabora un *benchmark* propio para la evaluación de la performance de los productos considerados. Esto se realiza basándose en el modelo de negocio del caso de estudio (considerando al mismo como un sistema de información típico) y en los tests propuestos por los *benchmarks* oo7 y PolePostion.

Cabe destacar que el *benchmark* es una simplificación de ambos y no pretende ser un estándar para la evaluación de mapeadores en sistemas de información. El mismo es de valor simplemente para la evaluación rudimentaria de la performance en el caso particular del caso de estudio.

Resultados del Benchmark Propuesto

La configuración de *hardware* y *software* en la cual se ejecutaron los tests del *benchmark* propuesto se muestra en el Cuadro 4.2.

Cuadro 4.2: Ambiente de Ejecución para el Benchmark Propuesto

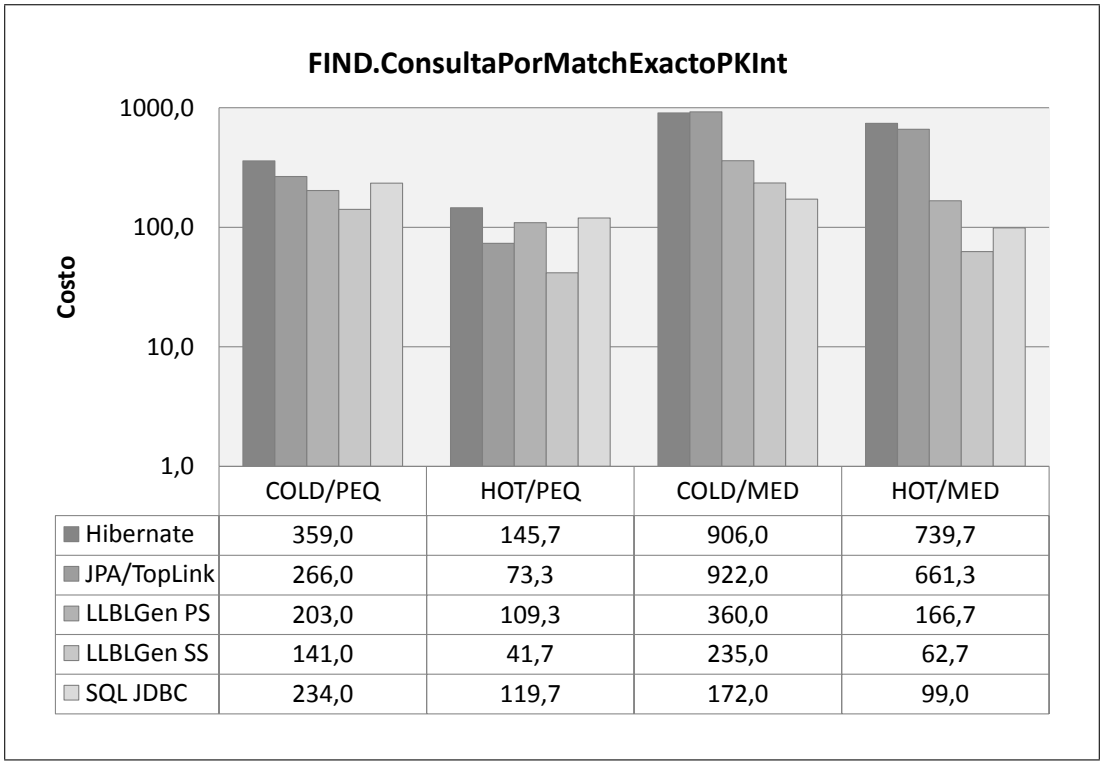
Item	**Detalles**
Hardware	
Procesador	Intel Pentium^{TM} 4 3.0GHz HT
Cache L2	1.0MB
Bus de Datos	800MHz
Memoria Principal	512MB DDR2 400MHz
Disco Duro	80GB 5400RPM
Software	
Sistema Operativo	Windows XP Professional
Java	Java SE Runtime Enviroment 6 Update 1
.NET	Microsoft .NET Framework 2.0
PostgreSQL	PostgreSQL Database Server 8.2
SQL Server	Microsoft SQL Server Express Edition 9.0.1399

En cuanto a la base de datos relacional, destino de los datos mapeados, se propone utilizar la misma para todos los productos de mapeo. En concreto se hace uso de PostgreSQL pretendiendo con ello eliminar diferencias de *performance* cuya causa sea el DBMS (se recuerda que la intención es evaluar mapeadores y no base de datos). Mas allá de esto, en el caso de LLBLGen Pro se hizo una excepción debido a que la herramienta solo permite la utilización de versiones anteriores de PostgreSQL y que además implica la utilización de un controlador desarrollado por la comunidad (Npqsql) que compromete los resultados del *benchmark*. Es así que para la evaluación de LLBLGen Pro se considerará como destino de los datos tanto Microsoft SQL Server Express Edition (abreviado en las gráficas como LLBLGen SS) como PostgresSQL (abreviado LLBLGen PS).

Los resultados del *benchmark* propuesto se presentan a razón de una gráfica por test,

incluyendo en cada una los valores para los diferentes productos. Se indica el tamaño de la base de datos mediante las abreviaciones PEQ indicando que la cantidad de datos existentes es relativamente pequeña y MED para indicar un tamaño mediano de datos. Asimismo se diferencia el método de ejecución de los tests con HOT para ejecuciones en caliente y COLD para ejecuciones en frío. Téngase en cuenta que los costos expresados en las gráficas en escala logarítmica refieren al tiempo de ejecución de los tests en milisegundos. Los tests son identificados por una clave compuesta por: el identificador del grupo (definido en E.2.4) y el nombre del test dentro del grupo. Como dato complementario se agrega a cada gráfica los valores correspondientes a la implementación de la persistencia basada en JDBC. A continuación se exponen, a modo de ejemplo, los resultados para algunos de los tests. La totalidad de los mismos se encuentra en el Apéndice E.

La Figura 4.2 presenta los resultados para el test *FIND.ConsultaPorMatchExactoPKInt*. El mismo recupera 40 objetos de tipo `Reserva` (seleccionados de forma aleatoria) valiéndose de las claves persistentes de los mismos, las cuales son de tipo `int`.



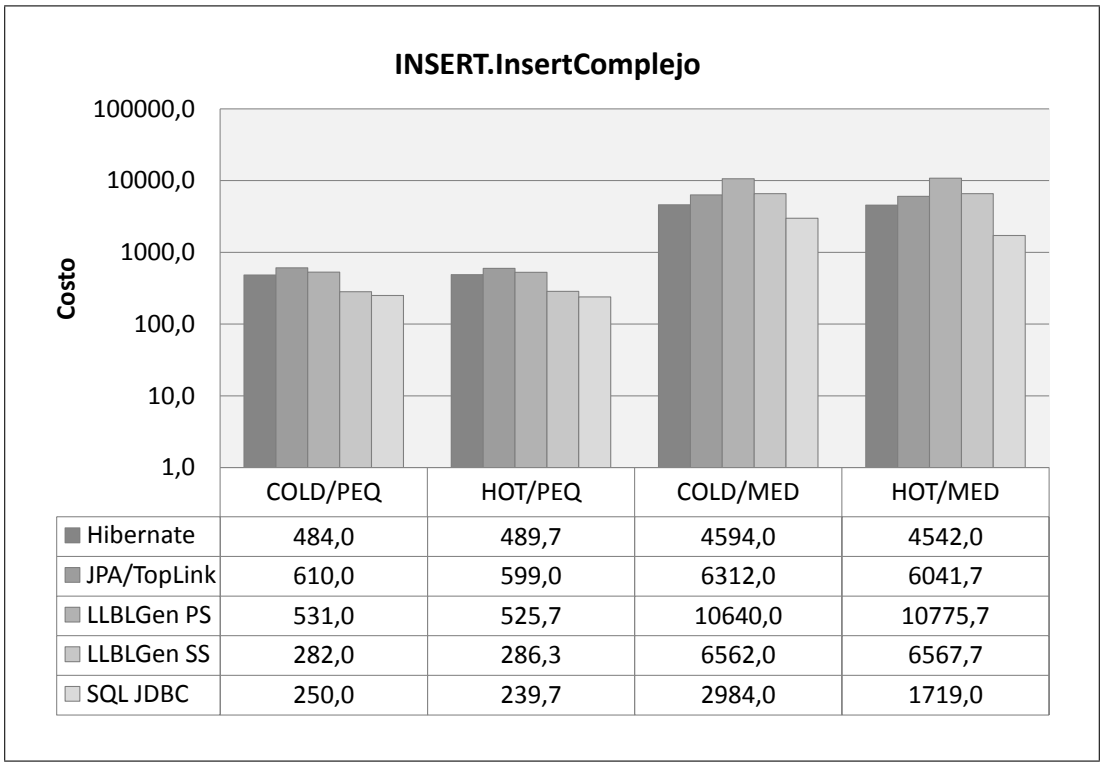
	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	359,0	145,7	906,0	739,7
JPA/TopLink	266,0	73,3	922,0	661,3
LLBLGen PS	203,0	109,3	360,0	166,7
LLBLGen SS	141,0	41,7	235,0	62,7
SQL JDBC	234,0	119,7	172,0	99,0

Figura 4.2: Test de Performance tipo FIND

En la Figura 4.3 se pueden observar los resultados del test *INSERT.InsertComplejo*. En el se realiza la inserción de 20 nuevos hoteles, lo que implica la creación de las habitaciones y recepciones de cada uno de ellos.

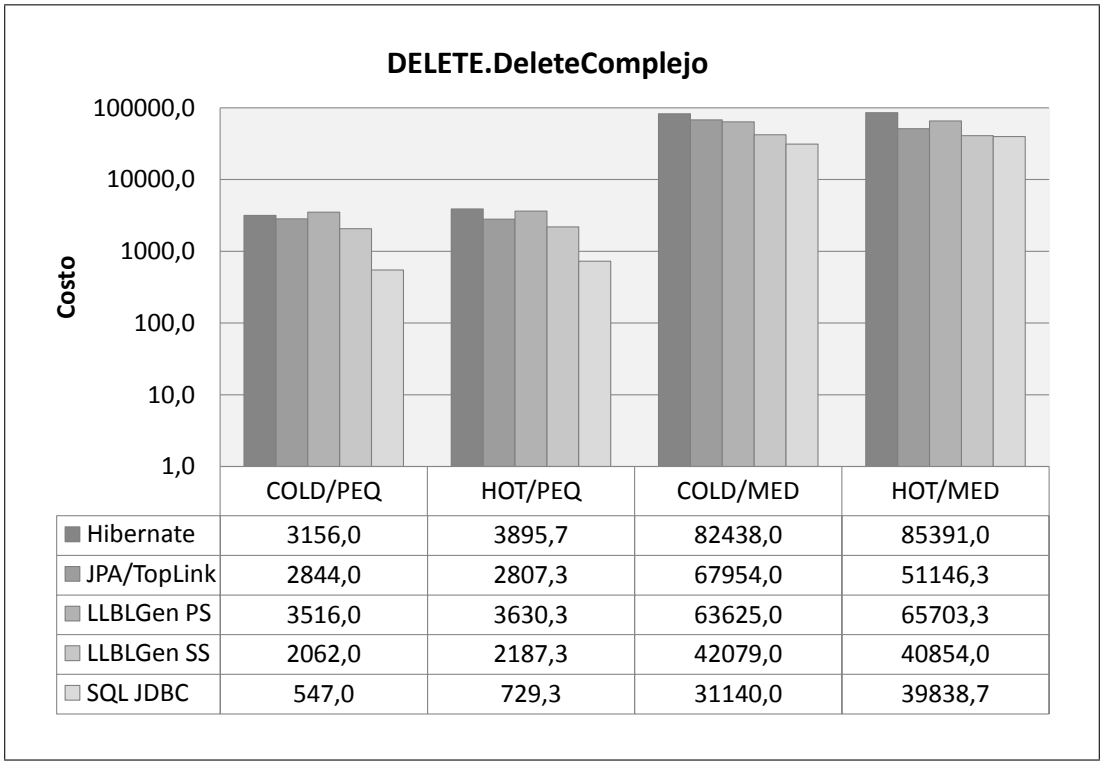
Por último la Figura 4.4 expone los resultados para el test *DELETE.DeleteComplejo*. Este test realiza la eliminación de 120 entidades de tipo `Hotel`, incluyendo la eliminación en cascada de las entidades de tipo `Habitacion` y `Recepcion` relacionadas.

De los resultados obtenidos se puede concluir lo siguiente. Es claro que al considerar la



	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	484,0	489,7	4594,0	4542,0
JPA/TopLink	610,0	599,0	6312,0	6041,7
LLBLGen PS	531,0	525,7	10640,0	10775,7
LLBLGen SS	282,0	286,3	6562,0	6567,7
SQL JDBC	250,0	239,7	2984,0	1719,0

Figura 4.3: Test de Performance tipo INSERT



	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	3156,0	3895,7	82438,0	85391,0
JPA/TopLink	2844,0	2807,3	67954,0	51146,3
LLBLGen PS	3516,0	3630,3	63625,0	65703,3
LLBLGen SS	2062,0	2187,3	42079,0	40854,0
SQL JDBC	547,0	729,3	31140,0	39838,7

Figura 4.4: Test de Performance tipo DELETE

totalidad de los tests no hay un mapeador que sea más performante en todas las categorías. Es por esto que las conclusiones que se pueden deducir de los resultados se refieren a un subconjunto de tests. Se identifica una tendencia de Hibernate a ser más lento con respecto a la solución de JPA/TopLink, sin ser así en los grupos de tests FIND e INSERT donde los resultados de ambos son muy similares. Al mismo tiempo se tiene una clara ventaja en performance en la utilización de LLBLGen Pro con respecto a los demás mapeadores en lo que se refiere a los grupos de tests QUERIES y DELETE. Nótese las diferencias en tiempo de ejecución de los diferentes tests para LLBLGen Pro sobre PostreSQL (utilizando el controlador Npsql) y LLBLGen Pro sobre Microsoft SQL Server Express Edition, lo que ratifica la necesidad de considerar ambas combinaciones como soluciones diferentes en lo que a performance se refiere.

Un resultado a destacar del desarrollo de los tests para la plataforma Java es que, a menos que se cuente con amplios conocimientos en el uso de JDBC, la utilización de un mapador objeto-relacional es recomendable. Lo anterior se ve fundamentado en el hecho de que los mapeadores consideran estrategias de optimización del uso de JDBC que no siempre están al alcance inmediato del desarrollador. Ademas de esto los mapeadores integran tecnologías de *caching* y similares que permiten mejorar la performance de recuperación y actualización de datos. Es así que para el desarrollo con JDBC fue necesario valerse de varias optimizaciones particulares al caso a fin de lograr resultados mejores o comparables a los obtenidos con mapeadores.

Transparencia

Dada la definición de transparencia presentada en 3.4.1 surge el problema de cómo evaluarla. En concreto debe buscarse una forma de cuantificar el esfuerzo invertido al utilizar determinada herramienta. Debido a la complejidad en la evaluación de dicho criterio, y considerando además el alcance del proyecto, se propone la medición de transparencia en términos de líneas totales de código (incluyendo código fuente y archivos de configuración). En general se supone dicha medición como un indicador aceptable. Dependiendo de la herramienta en particular se desglosa la medida anterior según resulte de interés.

En la Figura 4.5 se muestran las líneas totales de código para la implementación del caso de estudio sin persistencia en comparación a las líneas de código totales una vez agregada esta cualidad. Se discrimina además según la plataforma. Nótese que como dato interesante para la plataforma Java, se agrega el valor de líneas de código totales para el caso de la resolución de la persistencia mediante SQL (JDBC) teniendo que lidiar directamente con el *impedance mismatch*.

Para el caso de Hibernate se exponen la cantidad de líneas de código fuente en contraste a las líneas de código de configuración y mapeo. En cuanto a JPA/TopLink, se presenta una división de las líneas totales de código en: líneas de código fuente (sin considerar *annotations*), de configuración y *annotations*. Finalmente en lo que refiere a LLBLGen Pro se realiza un desglose en líneas de código fuente generados de forma automática en contraste con las líneas de código bajo el control del desarrollador. Para todos los casos se distingue dentro del código fuente las líneas referentes a test unitarios y las correspondientes a la lógica de la aplicación. Refiérase al Cuadro 4.3 para visualizar los

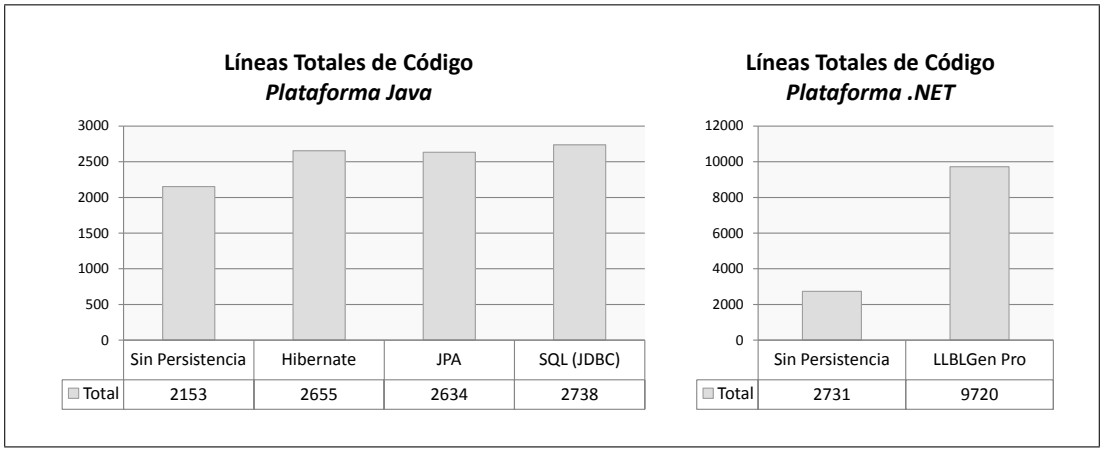


Figura 4.5: Líneas Totales de Código Java y .NET

valores obtenidos.

Cuadro 4.3: Desglose de Líneas de Código

Tipo	**Cantidad**
Hibernate	
Código Fuente Aplicación	1824
Código Fuente Tests	649
Configuración	23
Mapeo	159
Total	**2655**
JPA/TopLink	
Código Fuente Aplicación	1874
Código Fuente Tests	672
Configuración	42
Annotations	46
Total	**2634**
LLBLGen Pro	
Código Fuente Generado	7264
Código Fuente Desarrollador Aplicación	1546
Código Fuente Desarrollador Tests	910
Total	**9720**

Téngase en cuenta que las líneas de código generadas de forma automática para el caso de LLBLGen Pro no afectan directamente la transparencia de la solución y son presentadas como dato complementario. A fin de medir la transparencia de mejor forma, debería considerarse a parte de las líneas de código controladas por el desarrollador, el esfuerzo invertido en los pasos anteriores a la generación. Esto último incluye el diseño del esquema de base de datos y el diseño del mapeo objeto-relacional en la herramienta gráfica provista por el producto. Realizar esto se considera fuera del alcance del proyecto.

De los tres productos evaluados, se visualiza que LLBLGen Pro necesita de generación de código una vez definido el mapeo. Además, es necesario conocer el código generado ya

sea para modificar las clases generadas o bien utilizar sus métodos. Esto afecta negativamente en la transparencia, en comparación con los demás productos.

4.5. Resumen

En este capítulo se presentaron los mapeadores objeto-relacional Hibernate y JPA para la plataforma Java y LLBLGen Pro para la plataforma .NET. Se tuvo como objetivo principal realizar comparaciones entre los distintos mapeadores. De acuerdo con ello fue necesario definir previamente un conjunto de criterios de comparación para poder evaluar y comparar los productos considerados. Los mismos se agruparon en: criterios a nivel de mecanismos evaluados en el contexto de nivel de productos, criterios aplicables a productos de cualquier mecanismo y criterios específicos al mecanismo de persistencia considerado.

Con los criterios de comparación definidos, agrupados y en el contexto de un mismo caso de estudio, se procedió a evaluar algunos de ellos para los mapeadores objeto-relacional considerados. Como resultado se presentó en formato tabular la evaluación de varios criterios simples que permiten apreciar las diferencias y similitudes de los distintos productos. Se destaca además, la evaluación de los criterios de Transparencia y Performance. En lo referente a Performance se elaboró un *benchmark* tomando en cuenta el caso de estudio y *benchmarks* existentes. En cuanto a la evaluación de Transparencia, la misma se midió en términos de líneas totales de código comparando contra la implementación sin persistencia de la misma realidad.

Los criterios definidos permiten la comparación de las características más interesantes a nivel de: mapeadores, productos y mecanismos. Téngase en cuenta que los resultados de las comparaciones realizadas no destacan a uno de los productos como superior. Sin embargo provee al desarollador de información detallada en cuanto a las diferencias existentes en los aspectos considerados. El producto más adecuado para un proyecto en particular dependerá de las características condicionantes del mismo.

Capítulo 5

Resumen, Conclusiones y Trabajo Futuro

Este trabajo estuvo motivado por la necesidad de contar con una visión general, objetiva y bien estructurada que facilite la selección del mecanismo de persistencia acorde a las necesidades de un sistema orientado a objetos.

Primeramente se realizó un relevamiento que incluyó un estudio de los mecanismos existentes, tanto aquellos utilizados a nivel industrial como también aquellos emergentes académicamente o que no han logrado popularidad, así como un registro de su evolución histórica. Asimismo se buscaron clasificaciones de mecanismos de persistencia que pudieran contribuir a la visión objetiva de la realidad.

En base al relevamiento se dedujo que no existe una clasificación que cumpla con las características deseadas. Es por ello que se propuso una clasificación que intenta subsanar las carencias detectadas. La clasificación pretende contar con mecanismos claramente definidos en base a una estructura común, e independiente de toda tecnología. Esto aporta mayor comprensión global de las alternativas existentes desde el punto de vista del desarrollador que necesite utilizar cierto mecanismo.

Para un mejor entendimiento de las alternativas disponibles se complementó la clasificación con un análisis comparativo entre los mecanismos. Para ello se definieron y evaluaron un conjunto de criterios que permitieron contrastar de manera organizada las diferencias y similitudes. Los criterios involucran, a grandes rasgos, la incidencia de la aplicación del mecanismo en la lógica del sistema, los resultados obtenidos tras su aplicación, la incidencia desde la perspectiva del proyecto y consideraciones sobre la visión que el mercado tiene del mecanismo. Las evaluaciones de los mismos se basaron fuertemente en la experiencia y resultados obtenidos a partir de un caso de estudio considerado como representativo para el objetivo en cuestión.

Los resultados obtenidos del análisis y comparación de los mecanismos sugieren que no hay un mecanismo *one-size-fits-all* en lo que a persistencia se refiere. La conveniencia de uno u otro depende de las particularidades del proyecto para el cual se utilizará. Más allá de esto, de todos los mecanismos presentados, solo algunos son aplicables en la realidad

de hoy en día dada la inmadurez, el carácter académico o desuso del resto.

A pesar de no presentar un proceso sistemático de decisión que permita seleccionar el mecanismo a utilizar dado el contexto, se provee al desarrollador con la información necesaria para realizar la tarea de elección. Para esto, son de particular importancia los campos Contexto de Aplicación, Ventajas y Desventajas incluidos en la descripción de mecanismos de la clasificación propuesta. Asimismo, las comparaciones entre los distintos mecanismos ayudan en este cometido.

La necesidad de comparaciones y análisis a nivel de producto se puede ver como una continuación lógica del estudio a nivel de mecanismos. A este nivel existe un conjunto de características diferentes a las consideradas a nivel de mecanismo que condicionan la conveniencia de un producto para un proyecto en particular. Por tal motivo se plantearon criterios de comparación de productos los cuales fueron agrupados en: criterios generales anteriormente aplicados a mecanismos pero que pueden conducir a diferencias entre productos, criterios aplicables a productos de cualquier mecanismo y criterios específicos al mecanismo de persistencia considerado.

Finalmente, se ejemplificó la comparación entre productos tomando tres productos ampliamente utilizados en la industria (Hibernate, JPA/TopLink y LLBLGen Pro) que corresponden a implementaciones del mecanismo Mapeadores Objeto-Relacional. Para dicha comparación se implementó un caso de estudio utilizando los tres productos, haciendo énfasis en la evaluación de los criterios de Performance y Transparencia.

Los resultados de las comparaciones realizadas no destacan a uno de los productos como superior. Sin embargo provee al desarrollador de información detallada en cuanto a las diferencias existentes en los aspectos considerados.

Un resultado a destacar, obtenido a partir de la evaluación de performance para la plataforma Java, es que a menos que se cuente con amplios conocimientos en el uso de JDBC, la utilización de un mapeador objeto-relacional es recomendable. Lo anterior se ve fundamentado en el hecho de que los mapeadores consideran estrategias de optimización del uso de JDBC que no siempre están al alcance inmediato del desarrollador.

Al analizar el contenido del proyecto surgen cuatro aspectos a ser desarrollados como trabajo futuro: actualización, generación de métricas, flexibilización de las comparaciones y automatización de la clasificación, los que se describen a continuación.

La naturaleza cambiante de la realidad de los mecanismos de persistencia y temas relacionados conlleva a la necesidad de una constante actualización de las alternativas disponibles. De acuerdo a ello se debe adaptar la clasificación propuesta a la luz de nuevos mecanismos. Asimismo, es de interés la validación de las ventajas, desventajas y contextos de aplicación de los diferentes mecanismos definidos en la clasificación propuesta.

Con respecto a la evaluación de los criterios de comparación a nivel de mecanismo, la naturaleza de muchos de ellos implica la definición de métricas complejas y experimentación exhaustiva a fin lograr una evaluación objetiva. De igual forma a lo realizado para Mapeadores Objeto-Relacional, se propone la comparación de productos pertenecientes a otros mecanismos. Además es de interés la realización de comparaciones cruzadas entre productos de diferentes mecanismos.

La experimentación exhaustiva no necesariamente produce, como se ha visto, resultados determinantes para la elección del mecanismo o producto. Por el contrario, puede resultar contraproducente ya que el costo puede resultar muy grande en relación al beneficio obtenido. Por tal motivo, es de interés buscar mecanismos de flexibilización para la realización de las comparaciones, por ejemplo, que apunten a determinar la existencia o no de diferencias sustanciales con respecto a determinado criterio en lugar de una medida absoluta del mismo.

Por último, sería conveniente estudiar la forma de automatizar la toma de decisiones en base al conjunto de criterios seleccionados. Para ello podría definirse una ontología que involucre las características del proyecto para el cual se seleccionará el mecanismo, los criterios de comparación y un conjunto de productos a evaluar. La ontología permitiría seleccionar el/los mecanismos apropiados según las necesidades del proyecto.

Apéndice A

Descripción de Mecanismos de Persistencia

De acuerdo con la clasificación expuesta en 3.3 se presentan en detalle cada una de las categorías de la misma. La motivación para describir cada mecanismo de persistencia es la necesidad de presentar un esquema de solución (en un alto nivel de abstracción) a un problema surgido repetidamente, documentando la experiencia existente en el abordaje de dicho problema. Por dicho motivo, se optó por definir una estructura común para describir cada mecanismo de persistencia.

En concreto, para la exposición de los distintos mecanismos de persistencia se hará uso de la siguiente estructura:

- *Nombre del Mecanismo*. Nombre propuesto para la identificación del mecanismo en cuestión.
- *Intención*. Breve descripción del objetivo perseguido por el mecanismo.
- *Motivación*. Explicación de la necesidad que dio a lugar al surgimiento del mecanismo.
- *Descripción*. Presentación de aspectos generales de su funcionamiento, principales características y definición de conceptos involucrados.
- *Contextos de Aplicación*. Descripción de los ambientes en los cuales es recomendable la aplicación del mecanismo. Se tienen en cuenta, entre otras cosas, plataformas, dominio de aplicación y requerimientos de *performance*.
- *Ventajas*. Enumeración de los puntos favorables de la aplicación del mecanismo.
- *Desventajas*. Enumeración de los puntos desfavorables de la aplicación del mecanismo.
- *Mecanismo Base*. En caso de aplicar, se refiere al mecanismo general que se especializa. Por tratarse de una clasificación taxonómica, el mecanismo especializado cumple con la descripción presentada en el mecanismo base.

- *Mecanismos Relacionados.* De existir mecanismos relacionados los mismos son enumerados en esta sección explicando además su relación.

- *Herramientas Representativas.* Se sugieren algunas herramientas representativas del mecanismo tanto a nivel académico como de la industria.

Nótese que la estructura anterior aplica únicamente a los mecanismos concretos (entiéndase como las hojas del árbol de categorías). En cuanto a lo que se denomina mecanismos base (aquellas categorías descendientes directas de la raíz "Mecanismos de Persistencia" que tienen categorías hijas propias) se realiza simplemente una descripción general.

Téngase en cuenta que la naturaleza de los puntos Ventajas, Desventajas y Contexto de Aplicación requieren de investigación y experimentación exhaustiva a fin logar una evaluación objetiva. De acuerdo con esto último y considerando el alcance del proyecto, estos campos resultan de la recolección de información disponible y opiniones de terceros. Se entiende sin embargo, que el resultado de estas evaluaciones es de valor al momento de contrastar los diferentes mecanismos de persistencia.

A.1. Acceso Directo a Base de Datos

El mecanismo en cuestión implica el uso directo de la base de datos para implementar la persistencia del sistema. Esto último refiere al acceso a los datos utilizando una interfaz estandarizada en conjunto con algún lenguaje de consulta soportado directamente por el DBMS. Téngase en cuenta que no existe ningún tipo de *middleware* entre la aplicación y el DBMS utilizado.

A.1.1. Acceso Directo a Base de Datos Relacional

Intención

Proveer un mecanismo de acceso a los datos que sea eficiente haciendo uso de una base de datos relacional como dispositivo de almacenamiento.

Motivación

La necesidad de construir sistemas de mayor porte y complejidad llevó a que se requiriesen modelos de datos más flexibles. Si bien los modelos jerárquico y de red permitieron hacer más efectivo el uso de los nuevos dispositivos para almacenamiento con acceso directo, no se ajustaron a las necesidades conforme el tiempo avanzó. Tomando una perspectiva con mayor soporte teórico y demostrando ser una línea divisoria en el desarrollo de sistemas de base de datos, surge el Modelo Relacional y luego las Bases de Datos Relacionales.

Descripción

Se considera una Base de Datos Relacional a aquel tipo de base de datos cuyo modelo de datos subyacente es el relacional. La estructura fundamental de dicho modelo se denomina *relación*. La misma puede ser interpretada como una tabla bidimensional constituida por filas (tuplas) y columnas (atributos). Concretamente, las relaciones se utilizan para representar los datos de interés según el caso particular. En términos del mecanismo en cuestión, el mismo implica el uso directo de un base de datos relacional para implementar la persistencia del sistema.

Contextos de Aplicación

- Puede ser de utilidad para un sistema pequeño en el cual no se justifica un diseño que permita altos niveles de mantenibilidad.
- Cualquier contexto en que se requiera un considerable desempeño en acceder a los datos se puede ver beneficiado al aplicar el presente mecanismo. El no contar con la presencia de capas de abstracción por encima de la base (más allá de por ejemplo una interfaz de acceso) permite que la interacción con la misma se lleve a cabo con una mayor eficiencia.

Ventajas

- Muy buen nivel de desempeño debido a que el acceso se realiza de forma directa sin tener que pasar por ningún tipo de *middleware*. Nótese que se supone la existencia de un controlador y un DBMS eficientes para brindar acceso.
- El uso de interfaces estandarizadas para el acceso a datos permite cambiar la base subyacente sin impactos significativos en el diseño e implementación del sistema.
- El campo de las base de datos relacionales se ve sustentado por sólidas teorías y la experiencia que se ha ido ganando a lo largo de los años.

Desventajas

- Resulta más complicado diseñar soluciones mantenibles ya que el código necesario para el acceso a datos se entrelaza con el código de la lógica de negocios.
- Existencia de diferencia de paradigmas (*impedance mismatch* objeto-relacional) entre el nivel lógico y de persistencia.
- De acuerdo con la definición de tipos del Modelo Relacional, no se tiene un soporte adecuado para estructuras más complejas como ser arreglos, conjuntos, etc.

Mecanismo Base

- Acceso Directo a Base de Datos (Sección A.1).

Mecanismos Relacionados

- No tiene.

Herramientas Representativas

- Informix
  - Familia de manejadores de base de datos de IBM. Dentro de los productos ofrecidos se destaca Informix Dynamic Server (IDS), un *data server* de bajo mantenimiento y muy buen rendimiento con capacidad de procesar transacciones en línea.
  - `http://www-306.ibm.com/software/data/informix`
- Oracle Database
  - Manejador de base de datos comercial para múltiples plataformas ofrecido por Oracle Corporation. Se destaca por su diseño optimizado para trabajar de manera paralela junto con otros manejadores de su misma naturaleza. Existen diferentes versiones que se ajustan a distintos tipos de necesidades.
  - `http://www.oracle.com/database`
- PostgreSQL
  - Base de datos gratuita que soporta, entre otras cosas, procedimientos almacenados (internos y externos), índices, disparadores, concurrencia y una amplia gama de tipos de datos. Asimismo permite la herencia entre tablas de forma tal que una tabla padre puede compartir sus características con sus tablas hijas.
  - `http://www.postgresql.org`

A.1.2. Acceso Directo a Base de Datos Objeto-Relacional

Intención

Proveer un mecanismo de acceso a datos eficiente y que permita la utilización de tipos complejos de manera nativa haciendo uso de una base de datos objeto-relacional como dispositivo de almacenamiento.

Motivación

Conforme muchos sistemas de base de datos comenzaron a incluir manejos de estructuras complejas como puntos geográficos, texto, audio, etc. rápidamente se notaron las grandes limitaciones que presentaba el modelo relacional en estos aspectos. Asimismo, la naturaleza plana de las tablas no brindaba buen soporte para estructuras anidadas como ser arreglos o conjuntos. Tomando en cuenta lo anterior y añadiendo la flexibilidad del mundo orientado a objetos se propuso integrar las mejores características del modelo relacional con este último.

Descripción

Una Base de Datos Objeto-Relacional sintetiza las características de una base relacional con las mejores ideas de una base orientada a objetos. Si bien se reutiliza el modelo relacional, el modelo de datos subyacente se expande para proveer nuevos tipos de datos y funciones que pueden ser implementadas utilizando lenguajes de propósito general como C y Java. Asimismo, características orientadas a objetos como herencia y polimorfismo pasan a formar parte del modelo objeto relacional. Todos los accesos a datos en una base de este tipo se llevan a cabo con sentencias SQL. En términos del mecanismo en cuestión, el mismo implica el uso directo de una base de datos objeto-relacional para implementar la persistencia del sistema.

Contextos de Aplicación

- Puede ser de utilidad para un sistema pequeño en el cual no se justifica un diseño que permita altos niveles de mantenibilidad.
- Cualquier contexto en que se requiera un considerable desempeño en acceder a los datos se puede ver beneficiado al aplicar el presente mecanismo. El no contar con la presencia de capas de abstracción por encima de la base (más allá de por ejemplo una interfaz de acceso) permite que la interacción con la misma se lleve a cabo con una mayor eficiencia.
- Casos en los cuales el uso del modelo relacional sería la opción adecuada si se tuviera más flexibilidad a la hora de la definición de nuevos tipos de datos.

Ventajas

- Muy buen nivel de desempeño debido a que el acceso se realiza de forma directa sin tener que pasar por ningún tipo de *middleware*. Nótese que se supone la existencia de un controlador y un DBMS eficientes para brindar acceso.
- El uso de interfaces estandarizadas para el acceso a datos permite cambiar la base subyacente sin impactos significativos en el diseño e implementación del sistema.
- El campo de base de datos relacionales se ve sustentado por sólidas teorías y la experiencia que se ha ido ganando a lo largo de los años. Todo esto es sin lugar a dudas trasladable al campo de las bases de datos objeto-relacionales.
- Gran flexibilidad a la hora de tratar con tipos de datos complejos.

Desventajas

- Es más complicado diseñar soluciones mantenibles ya que el código necesario para el acceso a datos se entrelaza con el código de la lógica de negocios.
- Existencia de diferencia de paradigmas entre el nivel lógico y de persistencia.

- En relación con otros tipos de base de datos, las objeto-relacional son más complicadas desde el punto de vista de su utilización.

Mecanismo Base

- Acceso Directo a Base de Datos (Sección A.1).

Mecanismos Relacionados

- No tiene.

Herramientas Representativas

- Informix
  - Familia de manejadores de base de datos de IBM. Dentro de los productos ofrecidos se destaca Informix Dynamic Server (IDS), un *data server* de bajo mantenimiento y muy buen rendimiento con capacidad de procesar transacciones en línea.
  - `http://www-306.ibm.com/software/data/informix`
- Oracle Database
  - Manejador de base de datos comercial para múltiples plataformas ofrecido por Oracle Corporation. Se destaca por su diseño optimizado para trabajar de manera paralela junto con otros manejadores de su misma naturaleza. Existen diferentes versiones que se ajustan a distintos tipos de necesidades.
  - `http://www.oracle.com/database`
- PostgreSQL
  - Base de datos gratuita que soporta, entre otras cosas, procedimientos almacenados (internos y externos), índices, disparadores, concurrencia y una amplia gama de tipos de datos. Asimismo permite la herencia entre tablas de forma tal que una tabla padre puede compartir sus características con sus tablas hijas.
  - `http://www.postgresql.org`

A.1.3. Acceso Directo a Base de Datos Orientada a Objetos

Intención

Proveer un mecanismo de acceso a los datos eficiente y que no implique una diferencia de paradigmas entre el nivel lógico y de persistencia, haciendo uso de una base de datos orientada a objetos como dispositivo de almacenamiento.

Motivación

Contar con una lógica orientada a objetos y, por ejemplo, una base de datos relacional para persistir la información relevante presenta ciertos inconvenientes. En concreto, debido a la diferencia entre los modelo de datos subyacentes, existen problemas al intentar corresponder el concepto de clave con el de identidad de objeto, el de clave foránea con asociación, al intentar representar la herencia a nivel relacional, etc. Toda esta problemática recibe el nombre de *impendance mismatch objeto-relacional*. Una solución posible podría ser contar con algún tipo de *middleware* que brindara la abstracción necesaria de forma tal que siempre se trabajara a nivel de objetos. Si bien esta idea es la que fundamenta a los mapeadores objeto-relacional, otra alternativa podría ser evitar que el *impendance mismatch* se haga presente. De acuerdo con ello se contaría con una lógica orientada a objetos y un dispositivo para la persistencia también orientado objetos.

Descripción

Una Base de Datos Orientada a Objetos refiere a una base de datos cuyo modelo de datos subyacente es orientado a objetos. La idea fundamental de dicho modelo radica en contar con una colección de entidades individuales llamadas objetos. Los mismos, interaccionan mediante el envío y recepción de mensajes. Cada objeto puede ser interpretado como una pieza del sistema global que cuenta con determinadas responsabilidades bien definidas. En términos del mecanismo en cuestión, el mismo implica el uso directo de un base de datos orientada a objetos para implementar la persistencia del sistema.

Contextos de Aplicación

- Aplicaciones que de forma general recuperan pocos objetos altamente complejos y trabajan sobre los mismos por largos períodos de tiempo.
- Sistemas con relaciones de datos complejas. Más concretamente, casos en que se presentan algunas de las siguientes características: (1) falta de identificación única y natural de entidades, (2) gran cantidad de relaciones m-n, y (3) acceso a datos realizado mayoritariamente mediante navegación en el grafo de objetos.
- Aplicaciones con DBMS embebido (p.ej. aplicaciones para dispositivos móviles).
- Sistemas cuyo dominio es, por su naturaleza, cambiante a lo largo del tiempo.

Ventajas

- Muy buen nivel de desempeño en relación con otros mecanismos que utilicen una Base de Datos Orientadas a Objetos como dispositivo de almacenamiento ya que el acceso se realiza de forma directa sin tener que pasar por ningún tipo de *middleware*. Nótese que se supone la existencia de un controlador y un DBMS eficientes para brindar acceso.

- El uso de interfaces estandarizadas para el acceso a datos permite cambiar la base subyacente sin impactos significativos en el diseño e implementación del sistema.
- El modelo de datos orientado a objetos provee características de modelado sumamente ricas en contraste con otros modelos de datos.
- No existen diferencias de paradigmas entre el nivel lógico y de persistencia del sistema.
- Brinda un lenguaje de consulta más expresivo que SQL.
- Muy buen soporte para la evolución de esquema.

Desventajas

- Modificar el esquema subyacente de una base de datos orientada a objetos, ya sea creando, actualizando o modificando una clase persistente, típicamente implica que cambios deban ser realizados en otras clases de la aplicación. Esto último lleva de manera general a que cambiar el esquema implique una recompilación del sistema.
- Se pierde la interoperabilidad con sistemas relacionales. Hay que tener en cuenta que estos últimos son los más difundidos.
- La mayoría de los OODBMSs no brindan el soporte para la definición de vistas ni mecanismos de seguridad adecuados.

Mecanismo Base

- Acceso Directo a Base de Datos (Sección A.1).

Mecanismos Relacionados

- No tiene.

Herramientas Representativas

- db4o
  - Base de datos orientada a objetos de código abierto disponible tanto para la plataforma .NET como Java. Dentro de sus características principales se destacan la posibilidad de embeberla en el sistema que se esté desarrollando y el poder sincronizar objetos bidireccionalmente con cualquier base relacional.
  - `http://www.db4objects.com`
- Objetivity/DB

  - OODBMS comercial producido por Objectivity, Inc. Permite a las aplicaciones construir objetos persistentes tanto en C#, C++, Java, Python como Smalltalk sin la necesidad de mapear los datos a filas y columnas de una tabla. Soporta además SQL/ODBC y XML.
  - `http://www.objectivity.com`

- Versant Object Database

  - Se trata de una base de datos orientada a objetos, comercial, desarrollada por Versant. Provee persistencia transparente de objetos tanto para C++ como Java, soportando en este último caso los estándares JDO y Java EE. Asimismo la base puede ser totalmente embebida en el sistema desarrollado.
  - `http://www.versant.com/en_US/products/objectdatabase`

A.1.4. Acceso Directo a Base de Datos XML

Intención

Proveer un mecanismo de persistencia que, basándose en una base de datos XML, permita almacenar los datos de los objetos en documentos XML.

Motivación

XML fue inicialmente desarrollado como un estándar de la Web que permitiese que los datos estuviesen claramente separados de los detalles de presentación. Rápidamente se notó que XML cubría la necesidad de una sintaxis flexible para intercambiar información entre aplicaciones. Alternativas como persistencia de los documentos en el sistema de archivos o en una base relacional presentaron numerosas desventajas. De acuerdo con ello surgen bases de datos capaces de entender la estructura de documentos XML así como de llevar a cabo consultas sobre los mismos de una manera eficiente; las mismas reciben el nombre de Base de Datos XML.

Descripción

El Extensible Markup Language (XML) es un lenguaje de etiquetado de propósito general que permite crear lenguajes de *markup* para propósitos específicos. Un lenguaje de este tipo combina texto e información extra sobre este último. Dicha información, por ejemplo sobre estructura o presentación, se expresa utilizando *markup* entrelazado con el texto primario.

Actualmente, se acepta la clasificación de las Bases de Datos XML en dos categorías, a saber: Base de Datos XML-Native y Base de Datos XML-Enabled. Estás últimas se caracterizan por presentar un modelo de datos interno distinto a XML (generalmente se utiliza el modelo relacional) y un componente de software para la traducción correspondiente. De forma general, este módulo de traducción entre modelos no puede manejar

todos los posibles documentos XML. Por el contrario, reserva su funcionalidad para la subclase de documentos que cumplen con determinado esquema derivado a partir de los datos almacenados en la base. Finalmente, a diferencia de las XML-Enabled, las Base de Datos XML-Native son aquellas que utilizan el modelo de datos XML directamente. Esto es, utilizan un conjunto de estructuras para almacenar documentos XML arbitrarios.

En términos del mecanismo en cuestión, el mismo implica el uso directo de un Base de Datos XML para implementar la persistencia del sistema.

Contextos de Aplicación

- Aplicaciones centradas en XML, ya sea en el procesamiento del mismo o que lo utilicen como medio de transporte de datos.
- Sistemas en los cuales los datos manejados requieren estructuras más flexibles en comparación con una solución relacional.

Ventajas

- Muy buen nivel de desempeño en relación con otros mecanismos que utilicen una Base de Datos XML como dispositivo de almacenamiento ya que el acceso se realiza de forma directa sin tener que pasar por ningún tipo de *middleware*. Nótese que se supone la existencia de un controlador y un DBMS eficientes para brindar acceso.
- El uso de interfaces estandarizadas para el acceso a datos permite cambiar la base subyacente sin impactos significativos en el diseño e implementación del sistema.
- Si el sistema a construir se basa fuertemente en el almacenamiento, obtención y manipulación de documentos XML, una base de datos XML se encuentra especialmente optimizada para este tipo de operaciones.
- El uso de XML permite una gran flexibilidad a la hora representar los datos en contrapartida con otros modelos de datos (p.ej. el modelo relacional).

Desventajas

- Es más complicado diseñar soluciones mantenibles ya que el código necesario para el acceso a datos se entrelaza con el código de la lógica de negocios.
- Existencia de diferencia de paradigmas entre el nivel lógico y de persistencia.
- La utilización de XML como modelo de datos de soporte implica, considerando por ejemplo el *parsing* e interpretación del XML, un aumento claro en el tiempo de procesamiento. Téngase en cuenta que esto último impacta fundamentalmente si se utiliza una base de datos XML Nativa.

Mecanismo Base

- Acceso Directo a Base de Datos (Sección A.1).

Mecanismos Relacionados

- No tiene.

Herramientas

- XML-Enabled
  - Informix
    - Informix soporta XML mediante su Object Translator y Web DataBalde. El primero genera código que brinda a los objetos la capacidad de transferir sus datos desde y hacia la base de datos. El segundo, es una aplicación que genera documentos XML que contienen código SQL embebido.
    - `http://www.ibm.com/software/data/informix`
  - Oracle Database
    - La base de datos de Oracle provee soporte para el tipo de datos XML, SQL/XML, XQuery, XSLT, DOM, índices XML, el XML DB Repository y XML Schemas. Las columnas XML pueden ser almacenadas mediante un mecanismo objeto-relacional o simplemente a través de un `CLOB`.
    - `http://www.oracle.com/database`
  - PostgreSQL
    - Provee funciones para ejecutar consultas XPath y procesos XSLT sobre XML. Asimismo permite chequear que determinado valor XML esté bien formado. Los mismo pueden ser almacenados en columnas o pasados como entrada de tipo caracter (p.ej. `CHAR`, `VARCHAR`, etc.)
    - `http://www.postgresql.org`
- XML-Native
  - Berkeley DB XML
    - Base de datos XML-Native, de código libre, construida por encima de Berkeley DB agregando un *parser* XML, índices XML y el lenguaje de consulta XQuery. De Berkely DB hereda entre otras cosas el motor de almacenamiento, el soporte para transacciones y la recuperación automática.
    - `http://www.oracle.com/database/berkeley-db/xml`
  - X-Hive/DB
    - Base de datos XML-Native, comercial, que soporta las tecnologías XQuery, XPath, XML Schemas, DOM Level 3, XSL y XSL-FO. Asimismo brinda transacciones, control de acceso tanto a nivel de usuario como de grupo, replicación y balance de carga a través de múltiples servidores.
    - `http://www.x-hive.com/products/db`

- XQuantum XML Database Server
  - Base de datos XML-Native, de licencia comercial, desarrollada sobre un almacén de datos propietario. Soporta un subconjunto de XQuery y XSLT. Se caracteriza por optimizar las consultas mediante un algoritmo basado en costos calculados a través de estadísticas sobre los datos.
  - `http://www.cogneticsystems.com/server.html`

A.2. Mapeadores

En esta categoría se incluyen aquellos mecanismos que se basan en la traducción bidireccional entre los datos encapsulados en los objetos de la lógica de un sistema orientado a objetos y una fuente de datos que maneja un paradigma distinto. El mapeador ha de lidiar con los diferentes problemas que se presentan al mapear los datos entre paradigmas. El conjunto de problemas que se derivan de estas diferencias recibe el nombre de *impedance mismatch*.

A.2.1. Mapeadores Objeto-Relacional

Intención

Contar con un mecanismo para mapear los objetos de la lógica de un sistema orientado a objetos a una base de datos relacional.

Motivación

Hoy en día muchos desarrolladores trabajan en aplicaciones empresariales que crean, manejan y almacenan información estructurada que además es compartida entre muchos usuarios. Tradicionalmente la mayoría de estas aplicaciones involucran el uso de un DBMS basado en SQL. Más aun, una porción de ellas son completamente dependientes del modelo relacional en el propio núcleo de su lógica de negocio.

En los últimos años la programación orientada a objetos, con el surgimiento de lenguajes como Java y C#, ha tomado su lugar en la comunidad de desarrollo de software como el paradigma por excelencia. A pesar de que los desarrolladores se encuentran convencidos de los beneficios que este paradigma ofrece, la mayoría de las empresas están atadas a inversiones de largo plazo en caros sistemas de bases de datos relacionales. Además de esto, suelen existir datos legados (en bases de datos relacionales) con los cuales los nuevos sistemas orientados a objetos deben interactuar.

Debido a las diferencias entre la representación tabular de información y la encapsulación de los datos en objetos debe considerarse alguna técnica para poder resolver estas diferencias e integrar ambas tecnologías. Tradicionalmente la importancia de este problema ha sido subestimada. Mientras los programadores atribuían la culpa al modelo

relacional, los profesionales en datos culpaban al paradigma de objetos. Es así que se motiva el surgimiento de soluciones automatizadas que tomaron el nombre de Mapeadores Objeto-Relacional.

Es a través de un mecanismo de este tipo que se puede contar tanto con los beneficios de un lenguaje orientado a objetos como los de una tecnología madura como ser la relacional.

Descripción

El mecanismo en cuestión se basa en el uso de una entidad que se encargue del mapeo de objetos de negocio a un modelo relacional y viceversa. Al momento de mapear objetos a una base relacional se hace presente un problema que surge de las diferencias existentes entre ambos paradigmas. Este problema es denominado *impedance mismatch objeto-relacional* y se refiere a las dificultades que se presentan al mapear un objeto con identidad, que encapsula datos, comportamiento y que se relaciona con otros, a un modelo relacional que solo se concentra en los datos. Más aun, los tipos de datos básicos utilizados por ambos paradigmas son distintos. La solución a este último problema es directa ya que simplemente se mapea un tipo básico de un paradigma a uno del otro. Otro punto a tener en cuenta es que ambos paradigmas se basan en principios diferentes: la orientación a objetos se basa en principios del desarrollo de software mientras que el paradigma relacional se basa en principios matemáticos.

Para lidiar con la problemática antes expuesta, un mapeador se vale de *metadata* de mapeo provista por el desarrollador. La misma especifica el mapeo de: clases a tablas, atributos a columnas, relaciones a claves foráneas, etc. Además de proveer dicha *metadata*, el desarrollador es encargado de seleccionar diferentes estrategias de funcionamiento de forma explícita o por omisión. Mas allá de que la forma en que los objetos son mapeados a la base es particular a la implementación del mecanismo, existen estrategias básicas que son comúnmente aplicadas por los mismos.

Contextos de Aplicación

- Aplicaciones cuyas entidades siguen un ciclo de vida tipo `READ-MODIFY-WRITE`. Este ciclo implica que se obtiene un objeto a partir del mapeo de los datos almacenados en una base de datos relacional, se hacen modificaciones al mismo y luego es escrito nuevamente a la base. Un ejemplo de aplicaciones que responden a este patrón son las aplicaciones Web.

- Aplicaciones `READ-CENTRIC`. Es decir, aquellas que pasan la mayor parte del tiempo realizando operaciones de lectura de información.

- En el caso que se deba interactuar con una base de datos relacional desde un ambiente orientado a objetos.

Ventajas

- La utilización de un mecanismo de este tipo permite reducir radicalmente la cantidad de código necesario (manejado por el desarrollador) para lograr la persistencia. De acuerdo con ello se logra un incremento en la productividad.
- La aplicación es más fácil de mantener, por lo menos comparando esta solución a una en la que se accede directamente a la base de datos utilizando un lenguaje de consulta embebido (p.ej. SQL).
- Permite que el desarrollador pueda mantener una mentalidad orientada a objetos al momento de programar.
- La utilización de un cache de objetos, normalmente presente en los diferentes mapeadores, afecta positivamente la performance.

Desventajas

- Requiere agregar y mantener *metadata* sobre las entidades. Esta *metadata* contiene principalmente las particularidades del mapeo de cada entidad.
- Se pierde el control de cómo se lleva a cabo el proceso de persistencia. Esto es, se dejan al mapeador los detalles específicos de cómo las entidades son persistidas.
- Se dificulta el *tuning* de consultas, ya que requiere conocimientos de cómo se lleva a cabo el mapeo (información que no siempre está disponible).
- La herramienta de mapeo utilizada puede impactar de forma negativa en la *performance* del sistema, sobre todo si se trata de una aplicación que se dedique la mayor parte del tiempo a realizar operaciones tipo `WRITE`. De lo anterior se tiene que este mecanismo no es una opción razonable para sistemas con fuertes requerimientos de *performance* (p.ej. sistemas de tiempo real, aplicaciones militares, bolsa de valores, etc.).
- Existen consultas (complejas generalmente) en las que no se puede optar por una solución programada exclusivamente en el lenguaje orientado a objetos utilizado ya que de ser así, se agregaría un importante overhead al procesamiento de las mismas. Para resolver lo anterior suele recurrirse a un lenguaje de consulta como ser SQL (teniendo que lidiar con el modelo relacional) o a un lenguaje de consulta provisto por el mapeador (manteniéndose a nivel de objetos).
- A menos que el lenguaje de consulta provisto por la herramienta sea un lenguaje estandarizado (p.ej. SQL, OQL, etc.), el código de consulta queda ligado a la herramienta particular.
- Se requiere de lo que se denomina *shadow information* para poder mantener aquella información de los objetos que va mas allá de los datos de dominio. A modo de ejemplo: una clave primaria que provee la identidad del objeto (no tiene porque tener ningún significado en el negocio) o un campo que mantenga la versión para asistir en chequeos de integridad.

- El uso de un cache de objetos implica mayores requerimientos de memoria y probablemente problemas de consistencia entre los datos cacheados y los persistidos en la base de datos (sobre todo si existe más de una aplicación accediendo a esta).

Mecanismo Base

- Mapeadores (Sección A.2).

Mecanismos Relacionados

- *Acceso Directo a Base de Datos Relacional* (Sección A.1.1). El vínculo yace en el hecho de que un mapeador cumple la tarea de traducir los datos contenidos en objetos a datos en una base de datos relacional, permitiendo de esta forma obviar el *impedance mismatch* existente entre ambos paradigmas. Para lograr este cometido, normalmente el mapeador hace uso del mecanismo de acceso directo a una base de datos de forma transparente para el desarrollador.
- *Mapeadores Objeto-XML* (Sección A.2.2). A pesar de que el destino de los datos es distinto (documentos XML), muchos de los problemas básicos del mapeo se hacen presentes de igual forma (aunque la solución a los mismos sea muy diferente).

Herramientas Representativas

- Hibernate
  - Es una solución de código abierto para el mapeo objeto-relacional bajo la plataforma Java. La misma se adecua al proceso de desarrollo ya sea comenzando desde cero o partiendo de una base existente. Hibernate puede ser utilizado tanto para aplicaciones basadas en Java SE como en Java EE.
  - `http://www.hibernate.org`
- Java EE 5 y EJB 3.0
  - El lanzamiento de EJB 3.0 provee una nueva y simplificada API para el desarrollo de aplicaciones de porte empresarial. Dicha API tiene como objeto disminuir el costo de desarrollo y se trata concretamente de una simplificación de versiones anteriores de la especificación.
  - `http://java.sun.com/javaee/technologies/javaee5.jsp`
- TopLink
  - Paquete de Oracle para el mapeo objeto-relacional bajo Java. Asimismo ofrece, además de soporte para JAXB, mapeos objeto-XML y un rico *framework* de consulta que permite expresiones orientadas a objetos, QBE, EJB QL, SQL y procedimientos almacenados.
  - `http://www.oracle.com/technology/products/ias/toplink`

- LLBLGen Pro
  - Se trata de un producto para .NET que permite la generación de la capa de mapeo objeto-relacional para un sistema valiéndose de un esquema de base de datos existente.
  - `http://www.llblgen.com`

A.2.2. Mapeadores Objeto-XML

Intención

Contar con mecanismos de persistencia que almacenen los datos contenidos en los objetos de la lógica en forma de documentos XML.

Motivación

Con la creciente popularidad de XML como formato para el intercambio de datos, surgen nuevas modalidades de mapeo de objetos. La popularidad de los lenguajes de programación orientados a objetos y de documentos XML para intercambiar información entre sistemas motivó el surgimiento de mecanismos de persistencia que integraran estas tecnologías. El mapeo objeto-XML provee un medio por el cual los datos de negocio pueden ser visualizados en su forma persistida (no se trata de un formato críptico para el ojo humano), al mismo tiempo de que se facilita el intercambio de dichos datos con otros sistemas. La motivación principal de los mapeadores objeto-XML es simplificar la lógica del negocio al no tener que lidiar directamente con la estructura del documento XML y permitir que la misma se dedique simplemente a la manipulación de objetos (los cuales son mapeados desde y hacia los documentos XML correspondientes).

Descripción

La categoría en cuestión comprende aquellos mecanismos que se basan en el mapeo de objetos a un modelo de datos XML. Se trata del mapeo de los datos encapsulados en objetos a un documento XML y viceversa. Este proceso de conversión también se conoce bajo el nombre de *XML Marshalling* o *XML Serialization*. En el campo en cuestión, un *marshaller* es el responsable de serializar un objeto (o grafo de objetos) a XML. De manera similar, un *unmarshaller* deserializa el XML a un objeto (o grafo de objetos).

Los mapeadores objeto-XML brindan acceso a los datos contenidos en un documento XML a través de objetos. A diferencia de los modelos como DOM y JDOM, los mapeadores objeto-XML considerados toman un enfoque centrado en los datos y no en la estructura del documento XML de forma tal que el desarrollador utiliza objetos de negocio que reflejan el contenido mismo de los documentos. Este enfoque se puede resumir diciendo que los procesos de manipulación de los datos del sistema son *business-driven* y no *XML-driven*. En este proceso, al igual que en el mapeo objeto-relacional, se hace presente el problema denominado *impedance mismatch*. En el caso particular de los mapeadores objeto-XML

se refiere a las complicaciones que se hacen presentes en el proceso de mapeo entre el paradigma de objetos y el modelo XML (*impedance mismatch objeto-XML*).

Existen varias estrategias a la hora de implementar el mapeo objeto-XML. Dos de las más populares son la generación de código fuente a ser utilizado para acceder a los datos de negocio contenidos en los documentos XML y la especificación de *metadata* de mapeo entre objetos y documentos XML. En el caso de que se esté trabajando con documentos con una estructura estable, la generación de código probablemente sea la mejor opción. En cambio si se está trabajando con clases existentes o si se desea utilizar una estructura de clases que refleje el uso de los datos según la lógica de negocio y no la estructura del documento XML, la especificación de un mapeo a través de un archivo de *metadata* es la opción correcta.

Podemos dividir a los mapeadores objeto-XML en dos categorías: de tiempo de diseño y de tiempo de ejecución. Las dos estrategias mencionadas en el párrafo anterior se encuentran contempladas dentro de la primera categoría. En lo que respecta a la segunda categoría, la misma se caracteriza por no requerir ningún tipo de configuración de mapeo o generación de código. Los mapeadores que están considerados dentro de la misma pueden ser utilizados directamente para serializar y deserializar objetos. A pesar de su fácil uso, generalmente no se tiene ningún control de cómo los objetos son mapeados a XML. Muchos de los productos dentro de esta categoría se valen de tecnologías como Reflection para descubrir las propiedades de los objetos y utilizan los nombres de las mismas para definir la estructura del documento XML correspondiente.

Contextos de Aplicación

- *Archivos de Configuración.* Debido a que casi la totalidad de los archivos de configuración de aplicaciones actuales eligen XML como formato, es de especial interés para la lectura de estos datos la utilización de mapeadores objeto-XML. El uso de este mecanismo evita la necesidad de utilizar interfaces como SAX o DOM para lograr acceder a los datos de configuración. De esta forma se requiere, en la mayoría de los casos, un esfuerzo mínimo por parte del desarrollador.

- *Persistencia XML.* En el caso de que se desee utilizar una base de datos XML para persistir los datos, es necesario presentar los datos a la interfaz pertinente en forma de documentos XML. Debido a esto, el uso de un mapeador objeto-XML es de gran utilidad en el caso de que la fuente de datos sea una base de datos XML. De esta forma, la lógica del negocio solo debe tratar con los objetos quienes posteriormente son mapeados a XML (por el mapeador) y persistidos en una base de datos XML.

- *Presentación de datos.* Si existe particular interés en la representación gráfica de los datos persistidos, la utilización de un mapeador objeto-XML es aconsejable. Existen varias tecnologías que permiten transformar los datos contenidos en un documento XML a una representación gráfica de los mismos (p.ej. XSLT, Formating Objects, etc.). De la misma forma existen editores visuales de documentos XML que permiten la edición de los datos de manera intuitiva.

- *Intercambio de datos.* La utilización de XML como formato para el intercambio de información entre sistemas heterogéneos es un escenario donde los mapeadores

objeto-XML son de interés. Los datos en forma de XML recibidos por un sistema pueden ser mapeados a objetos para ser procesados. Luego, el resultado de este proceso (en forma de otro objeto) puede ser mapeado nuevamente a XML para ser devuelto como respuesta a otro sistema con el que se interopera.

Ventajas

- *Documentos fácilmente legibles y auto-descriptivos.* Si se cuenta con un buen diseño de documento es razonablemente simple comprender el tipo de información que este contiene.
- *Interoperabilidad.* No hay nada acerca de XML que lo haga dependiente de una plataforma o tecnología particular. En adición a esta independencia, lo único requerido para poder editar un documento es un editor de texto. Al mismo tiempo existe una gran variedad de *parsers* disponibles para las diferentes plataformas.
- *Tolerancia de expansión de la estructura.* Siempre y cuando el documento XML siga siendo sintácticamente correcto es posible agregar nuevos elementos a la estructura sin afectar la compatibilidad con versiones anteriores de aplicaciones que utilicen estos documentos.

Desventajas

- *Estructura física del documento XML.* La mayoría de los productos dentro de esta categoría no preservan construcciones propias del modelo XML, tales como secciones `CDATA` y referencias a entidades. La razón de esto es que tales construcciones no tienen correspondencia con los lenguajes de programación orientados a objetos.
- *Comentarios e instrucciones de procesamiento.* En la mayoría de los casos no se preservan comentarios o instrucciones de procesamiento debido a que el mapeo de dichos elementos al paradigma de objetos no es trivial (sobre todo debido a que los comentarios pueden ocurrir en cualquier parte de un documento XML).
- *Soporte incompleto de esquema.* Los mecanismos de mapeo objeto-XML suelen soportar solo un subconjunto de XML Schemas. A modo de ejemplo: wild cards, restricciones complejas, etc. no suelen ser soportados.
- *Limitaciones con fragmentos de documentos.* En la mayoría de los casos no es posible operar sobre fragmentos de documentos. Es decir, no se puede trabajar de forma concurrente en un mismo documento a pesar de que los agentes interesados estén trabajando en secciones diferentes del mismo. Es por eso que los interesados en modificar una pequeña porción del documento deben hacerlo en forma serial o implementar soluciones propias al problema.

Mecanismo Base

- Mapeadores (Sección A.2)

Mecanismos Relacionados

- *Generadores de Código* (Sección A.3). Esto se ve fundamentado en el hecho que muchas de las implementaciones de mapeadores objeto-XML se valen de la generación de clases capaces de ser serializadas a un documento XML.

Herramientas Representativas

- Castor
  - Es un herramienta de código abierto para la plataforma Java que permite tanto la conversión objeto-relacional (Castor JDO) como objeto-XML (Castor XML).
  - `http://www.castor.org`
- Java Architecture for XML Binding (JAXB)
  - JAXB permite a los desarrolladores Java crear y editar documentos XML mediante objetos. Quienes utilicen JAXB, simplemente requieren de dos pasos para el análisis de un documento XML (el cual requiere un esquema). El primero es el proceso de *binding*, en el que se generaran las clases que representan los elementos y las relaciones del esquema. El segundo es el proceso de *unmarshalling*, en el que se construye el árbol de objetos que representa los datos contenidos en el documento XML.
  - `http://java.sun.com/webservices/jaxb`
- TopLink
  - Paquete de Oracle para el mapeo objeto-relacional bajo Java. Asimismo ofrece, además de soporte para JAXB, mapeos objeto-XML y un rico *framework* de consulta que permite expresiones orientadas a objetos, QBE, EJB QL, SQL y procedimientos almacenados.
  - `http://www.oracle.com/technology/products/ias/toplink`

A.3. Generadores de Código

Intención

Concentrarse en la especificación abstracta y de alto nivel de un sistema, delegando su construcción a una herramienta (semi) automática.

Motivación

El surgimiento de numerosos *frameworks* que solucionan parte de la problemática del desarrollo de una aplicación empresarial y el creciente uso de patrones de diseño ha tenido

como consecuencia el desarrollo de aplicaciones empresariales más robustas en menos tiempo. Sin embargo el programador aun se ve obligado a realizar tareas repetitivas viéndose afectada así la productividad de estos. Un ejemplo de lo anterior se puede encontrar en la plataforma Java EE, donde gran parte del tiempo del programador se pierde en producir el código involucrado en la definición de EJBs (este problema ha sido atacado en la nueva versión de la plataforma). La idea de que el programador debe concentrarse en el código que representa la lógica de negocio es la principal motivación para el surgimiento de los generadores de código.

Descripción

Un generador de código es una herramienta que basándose en *metadata* de un proyecto es capaz de generar código correcto, robusto y que aplica los patrones de diseño pertinentes. Este código generado se encarga de resolver parte de la problemática del sistema.

Los generadores de código pueden lograr que la evolución del proyecto sea muy ágil. La afirmación anterior se basa en que un cambio en la implementación detrás del código generado se reduce a configurar el generador de forma diferente (p.ej. utilizando una plantilla de generación diferente) y luego proceder a generar el código nuevamente. El incremento en la productividad que se puede lograr utilizando generadores es considerable, llegando a ser de varios ordenes de magnitud. A diferencia de pequeñas herramientas de generación de código incluidas en los IDEs populares, los generadores de código que se consideran en esta categoría son capaces de generar capas del sistema en su totalidad y en algunos casos aplicaciones completas. Una aplicación particular de los generadores de código es la generación de capas de acceso a datos o de código necesario para lograr la persistencia a través de *frameworks* u otra técnica. Particularmente se puede hablar de la generación de una capa de acceso a datos que responde al patrón Data Access Object (DAO), que se genera tomando como *metadata* la definición del esquema de base de datos. Otra aplicación común es la generación del esquema de base de datos a partir de la definición de las clases del negocio y *metadata* de mapeo.

A modo de resumen, mas allá de que se utilice una herramienta off-the-shelf o propia para la generación de código con el objetivo de resolver la persistencia de un sistema, la primicia detrás de ambas alternativas es evitar la parte repetitiva y predecible del desarrollo.

Contextos de Aplicación

Este tipo de mecanismo es especialmente aplicable en situaciones donde la solución buscada resulta de la repetición de numerosas y similares iteraciones. Un ejemplo de estos escenarios es la generación de una capa de acceso a una base de datos relacional, donde la *metadata* se encuentra en el esquema de la base de datos. El patrón utilizado para acceder a las diferentes tablas es el mismo. Al tratarse de una tarea repetitiva, un generador de código es la herramienta idónea. Bajo estas condiciones, escribir el generador de código (o configurar uno) que genere la capa probablemente sea menos costoso que codificar la capa entera. Sin mencionar que al momento de tener que actualizar el código debido a cambios

en el esquema, si se cuenta con un generador, solo es necesario ejecutarlo nuevamente.

Ventajas

- En el caso de que los requerimientos sobre el código generado cambien, solo es necesario modificar el generador (o configurarlo apropiadamente) para luego ejecutarlo y obtener el código fuente que cumpla con los nuevos requerimientos.
- El código generado suele ser estable y libre de errores. Esto debido a que la depuración del código generado se hizo en una etapa anterior al proyecto, o sea en el propio desarrollo del generador.
- La generación es extremadamente rápida permitiendo así que el código generado esté al día con los cambios en la *metadata* del proyecto.
- La generación es personalizable. En el caso de que se disponga del código fuente del generador o que el mismo soporte plantillas de generación, el código generado puede ser altamente personalizado.
- Los desarrolladores pueden concentrarse en las áreas del sistema que involucran la lógica de negocio y que requieren de razonamiento intensivo. Esto claramente tiene un impacto positivo en la moral y por lo tanto en la productividad.
- Grandes volúmenes de código escrito a mano tiende a presentar inconsistencias ya que los programadores suelen encontrar mejores formas de hacer lo mismo a medida que el proyecto avanza. La generación de código crea una base de código consistente de forma instantánea, lo que incrementa la calidad del producto.
- Aun en el mejor sistema escrito a mano, el cambiar el nombre de una entidad (por ejemplo el nombre de una tabla) requiere cambios a lo largo del sistema (p.ej. capa de objetos, documentación y casos de prueba). Un enfoque de generación de código permite que este nombre se cambie en un solo lugar y que luego se regenere las partes involucradas para que utilicen el nuevo nombre.

Desventajas

- Se debe invertir tiempo en realizar una de las siguientes tareas: desarrollar un generador de código propio, seleccionar un generador ya desarrollado para el propósito específico de solucionar la persistencia o construir una plantilla de generación con la cual se pueda configurar un generador de propósito general.
- Siempre existirá código que de debe ser escrito a mano.
- Los generadores de capas de acceso a datos no suelen comportarse bien con bases de datos de características de diseño especiales (se debe seguir una estructura típica).

Mecanismo Base

- No tiene.

Mecanismos Relacionados

- *Acceso Directo a Base de Datos Relacional* (Sección A.1.1). Existen casos en que los generadores tienen como salida de su proceso de generación código fuente y archivos de *metadata* necesarios para poder utilizar este mecanismo.
- *Mapeadores Objeto-Relacional* (Sección A.2.1). Existen casos en que los generadores tienen como salida de su proceso de generación código fuente y archivos de *metadata* necesarios para poder utilizar este mecanismo.
- *Mapeadores Objeto-XML* (Sección A.2.2). Existen casos en que los generadores tienen como salida de su proceso de generación código fuente y archivos de *metadata* necesarios para poder utilizar este mecanismo.

Herramientas Representativas

- FireStorm/DAO
  - Es un generador de código para Java capaz de importar esquemas de base de datos existentes y a partir de ellos generar la capa de persistencia para determinado sistema. La misma puede estar basada en cualquiera de las siguientes tecnologías: JDBC, JDO, EJB o Hibernate.
  - `http://www.codefutures.com/products/firestorm`
- JAG
  - JAG es una herramienta que permite crear aplicaciones Java EE completas y funcionales. La arquitectura del sistema a generar puede controlarse mediante plantillas predefinidas o creadas manualmente. Las aplicaciones generadas están orientadas a ser utilizadas como base en el proceso de desarrollo.
  - `http://jag.sourceforge.net`
- LLBLGen Pro
  - Se trata de un producto para .NET que permite la generación de la capa de acceso a datos para un sistema valiéndose de un esquema de base de datos existente. El código generado se provee como un proyecto Visual Studio .NET que puede ser agregado a la solución existente o compilado de manera separada.
  - `http://www.llblgen.com`
- MyGeneration
  - MyGeneration es una solución para .NET que permite generar capas de mapeo objeto-relacional o archivos de mapeo para herramientas como Gentle.NET, Opf3 o NHibernate entre otros. La generación se realiza a partir de plantillas que pueden ser escritas en C#, VB.NET, JScript o VBScript.
  - `http://www.mygenerationsoftware.com`

A.4. Orientación a Aspectos

Intención

La idea fundamental es manipular la persistencia como un aspecto ortogonal a las funcionalidades, desacoplando así el código correspondiente al resto del sistema.

Motivación

La *Programación Orientada a Aspectos* (AOP) es un paradigma que permite definir abstracciones que encapsulen características que involucren a un grupo de componentes funcionales, es decir, que corten transversalmente al sistema. A cada una de estas abstracciones definidas se les denomina *aspecto*. A modo de ejemplo se puede encontrar en el desarrollo de un sistema aspectos de seguridad, manejo de excepciones, logueo, comunicación, replicación e incluso persistencia. En la programación orientada a aspectos, las clases son diseñadas e implementadas de forma separada a los aspectos, requiriendo luego una fusión mediante el denominado *aspect weaver*. Este último une las clases y los aspectos mediante puntos de juntura que establecen la relación entre los mismos. Ejemplos de puntos de juntura son llamadas a procedimientos, ejecución de métodos, constructores, referencias a atributos, entre otros.

Descripción

De acuerdo con el paradigma de programación orientada a aspectos, se podría definir la persistencia como un aspecto. En el mismo se almacenarían y obtendrían los datos desde una unidad de almacenamiento secundario. Se ve que este requerimiento corta transversalmente un sistema. Uno de los objetivos de definir el aspecto persistencia es permitir que una aplicación pueda ser diseñada e implementada sin considerar los requerimientos de persistencia, debido a que la misma será desarrollada de forma ortogonal a la implementación de la lógica del sistema, para luego ser fusionada en una etapa posterior. Otro objetivo de este mecanismo es a su vez permitir modularizar la persistencia para luego poder reutilizar el código generado.

Contextos de Aplicación

Por tratarse de una tecnología emergente, se han realizado distintos prototipos y casos de estudio a nivel académico a fin de lograr la aspectización de la persistencia. Se requerirá de la evolución de la tecnología así como también de la profundización en el área por intermedio de diferentes proyectos para poder determinar: la independencia entre el aspecto persistencia y las clases de una aplicación, la reusabilidad del aspecto frente a diferentes contextos de aplicación y los efectos de la *performance* comparados con otros mecanismos de persistencia.

Ventajas

- *Separación de la persistencia*. Se mantiene todo lo referente a la persistencia concentrado en determinada ubicación en contrapartida a tener el código que representa la persistencia de los objetos dispersos en distintos componentes. Esto incrementa la modularidad del sistema.
- *Mantenibilidad*. Los sistemas que utilizan la persistencia como un aspecto facilitan el mantenimiento, entendimiento y evolución del mismo al tener localizado dónde realizar los cambios que afecten la persistencia de los objetos.

Desventajas

- Requiere del conocimiento de un nuevo paradigma con su teoría, reglas y construcciones.
- No existen ejemplos prácticos que permitan demostrar que la utilización de AOP para la persistencia logre modularizarla y que dicho aspecto pueda ser reutilizado en otras aplicaciones.
- Es aún una tecnología emergente, lo cual no lo hace un mecanismo aplicable más allá de los límites académicos.

Mecanismo Base

- No tiene.

Mecanismos Relacionados

- *Generadores de Código* Sección A.3). El denominado *aspect weaver* es en si un generador de código, que toma la implementación del aspecto de persistencia como entrada, así como la implementación de los distintos componentes del sistema, y genera el código necesario para la persistencia de la aplicación. Esta fusión es lograda mediante la definición de puntos de juntura que relacionan las clases con los aspectos.

Herramientas Representativas

- AspectJ
  - Se trata de una extensión para el lenguaje Java que permite al programador hacer uso del paradigma de orientación a aspectos en sus proyectos. AspectJ se ha convertido en el estándar de facto para AOP mediante el énfasis en su simplicidad y usabilidad a nivel de los desarrolladores.
  - `http://www.eclipse.org/aspectj`
- AspectC++

- De manera similar a AspectJ, se trata de una extensión para C y C++ que permite utilizar la orientación a aspectos en éstos lenguajes. AspectC++ se basa en la traducción código a código, convirtiendo el código fuente AspectC++ en código fuente C++.
- `http://www.aspectc.org`

A.5. Lenguajes Orientados a Objetos

Descripción

La categoría de lenguajes orientados a objetos como mecanismo de persistencia implica resolver la persistencia de datos utilizando funcionalidades provistas por el propio lenguaje de programación.

A.5.1. Librerías Estándar

Intención

Utilizar las funcionalidades básica incluidas en al infraestructura de un lenguaje de programación orientado a objetos para solucionar la persistencia de un sistema.

Motivación

El principal motivo del uso del mecanismo de librerías estándar es la utilización de las capacidades de persistencia ofrecidas por el lenguaje de programación utilizado, evitando la inclusión de *frameworks* u otras herramientas ajenas al mismo.

Descripción

El uso de librerías que provee el lenguaje de programación orientado a objetos, nos permite resolver la persistencia de una aplicación. Tomando como ejemplo el lenguaje de programación Java, se tienen librerías para el manejo de archivos que permiten persistir los datos de una aplicación en archivos de texto plano, CSV (Comma Separated Value) o XML. También permite la utilización de técnicas de serialización de objetos. En dicha técnica se almacena el estado de un objeto y el grafo de objetos que él mismo referencia, a un *stream* de salida. Las relaciones entre los objetos se mantienen de manera que el grafo completo se puede reconstruir más adelante. De esta forma se tiene la habilidad de leer y escribir objetos desde y hacia *streams* de bytes mediante la implementación de la interfaz `Serializable`.

Contextos de Aplicación

- Puede ser de utilidad para aplicaciones de porte pequeño.
- Aplicable a la persistencia de información de configuración.

Ventajas

- Es un mecanismo simple de persistencia.
- Se evitan las complejidades de instalación, configuración y utilización asociadas a otros mecanismos (p.ej. mapeadores)

Desventajas

- No soporta transacciones.
- No se dispone de mecanismos de formulación de consultas sobre los datos.

Mecanismo base

- Lenguajes Orientados a Objetos (Sección A.5).

Mecanismos Relacionados

- No tiene.

Herramientas Representativas

- General
  - Utilización de cualquier librería estándar para el manejo de archivos a fin persistir en formatos propios.
- Java
  - Mecanismo de serialización de objetos en Java. Incluido en el paquete `java.io` como parte del Java Development Kit (JDK).
  - Mecanismo de manejo de archivos de propiedades en Java.
  - `http://java.sun.com`
- .NET
  - .NET provee el mecanismo de serialización de objetos, como parte del .NET framework
  - `http://www.microsoft.com/net`

A.5.2. Lenguajes Persistentes

Intención

Resolver de manera transparente la persistencia de datos utilizando funcionalidades provistas por el lenguaje de programación, cumpliendo con los tres principios de persistencia ortogonal (o una relajación de ellos).

Motivación

La idea de realizar un lenguaje de programación persistente surge a principios de los años ochenta, cuando comienza a desarrollarse PS-Algol, que es lanzado recién a mediados de la misma década. Uno de los motivos principales que llevó a desarrollar dicho lenguaje, fue la gran cantidad de tiempo que se dedicaba a la transferencia de datos desde y hacia el manejador de base de datos (DBMS). Esto no sólo requería cerca de la tercera parte en el tiempo de programación de una aplicación, sino que agregaba complejidad al tener que lidiar con el mapeo a las estructuras de la base de datos. Otro de los motivos fue la perdía de la protección de tipo que el lenguaje proveía.

Descripción

La utilización de lenguajes persistentes como mecanismo de persistencia implica que el lenguaje de programación cumpla con los tres principios de persistencia ortogonal:

- *ortogonalidad de tipo*: cada objeto, independiente de su tipo, tiene el mismo derecho a ser persistido.
- *persistencia por alcance*: se persisten sólo aquellos objetos que son alcanzables desde la raíz de persistencia.
- *independencia de persistencia*: la introducción de persistencia no puede introducir cambios semánticos del código fuente.

Estos tres principios tienen como propósito principal maximizar la productividad del desarrollador. Permite trabajar sin considerar mapeos entre representaciones de datos activas y persistentes. Tampoco se tiene que realizar explícitamente la transferencia de datos entre las aplicaciones y los sistemas de almacenamiento. Desde el punto de vista del desarrollador, la persistencia se realiza de forma totalmente transparente y de manera automática. Por lo tanto, no es necesario considerar procedimientos específicos para guardar y recuperar los objetos persistentes del sistema. Sin embargo, esta transparencia se ve afectada al agregar soporte de transacciones o al evolucionar el esquema. Con motivo de obtener sistemas más eficientes, se han relajado los tres principios descriptos anteriormente, desarrollándose lenguajes de programación que cumplen únicamente con el principio de ortogonalidad de tipo (p.ej. Barbados). Además de cumplir solo con el primer principio, suelen existir restricciones con respecto a qué objetos pueden ser referenciados desde otros objetos. Estos lenguajes suelen utilizar persistencia basada en contenedores, siendo estos

objetos especializados que funcionan como medio de acceso a los objetos persistentes. De esta forma se tiene que todo los objetos persistentes del sistema están diferenciados del resto por estar contenidos en uno de estos contenedores o por ser referenciado por uno que si lo esta. El motivo de desarrollar lenguajes que sólo cumplan con la ortogonalidad de tipo es, además de ser más eficientes, otorgar mayor seguridad y organización sobre los datos, obtener mayor distribución y favorecer a la evolución del esquema.

Algunos de los problemas enfrentados por este tipo de lenguajes son: (1) resolver problemas de cómo agrupar los objetos para una mejor accesibilidad y (2) cómo proteger los objetos en la memoria. Además, se deben resolver ambos problemas minimizando el impacto sobre el lenguaje de programación.

Contextos de Aplicación

Los lenguajes persistentes no han madurado como para salir del área académica. En un principio se desarrollaron diferentes prototipos, como extensiones de PS-Algol, o tomando sus ideas, con el objetivo de adaptarlo a la tecnología de objetos. Pero al querer respetar la ortogonalidad de tipo, la persistencia por alcance y la independencia de persistencia, se dejaron de lado otros aspectos como por ejemplo la *performance*. Todo esto hace que el mecanismo no sea aplicable fuera del ambiente académico.

Ventajas

- Evita la realización del mapeo entre la base de datos y la aplicación, reduciendo el tiempo de desarrollo y la complejidad del mismo.
- El cumplimiento de los tres principios de persistencia ortogonal favorece a aspectos como ser transparencia y productividad.
- Al agregar persistencia a un lenguaje de programación, no requiere el aprendizaje de nuevas tecnologías o herramientas para persistir los datos, incrementando la productividad.

Desventajas

- La persistencia ortogonal posee como principal desventaja el notorio decremento de *performance* de una aplicación.

Mecanismo base

- Lenguajes Orientados a Objetos (Sección A.5).

Mecanismos Relacionados

- *Prevalencia* (Sección A.6). Ambos enfoques implican altos niveles de transparencia para el desarrollador, sacrificando aspectos como *performance* y escalabilidad.

Herramientas Representativas

- PJama
    - Es un prototipo experimental que implementa *Orhtogonal Persistence for the Java platform* (OPJ). Por su parte, OPJ es un enfoque que tiene como objetivo persistir los objetos de aplicación a lo largo de las diferentes ejecuciones del sistema utilizado el mínimo esfuerzo posible de parte de los desarrolladores.
    - `http://research.sun.com/forest/opj.main.html`

A.5.3. Lenguaje de Consulta Integrado

Intención

Contar con un lenguaje de programación que integre como parte del mismo un lenguaje de consulta y administración de datos persistentes.

Motivación

Existen múltiples interfaces de acceso a datos que son utilizadas desde numerosos lenguajes de programación como método para administrar y consultar los datos persistentes de un sistema. Estas interfaces pueden ser tanto interfaces de acceso a bases de datos, mapeadores u a otro tipo de componentes que permitan resolver las necesidades de persistencia del sistema. Las mismas permiten controlar el ciclo de vida de los datos y ejecutar consultas sobre los mismos, ya sea mediante un lenguaje de consulta estándar, propietario o mediante la utilización de API de creación de consultas a través de objetos (por ejemplo el provisto por Hibernate en su Hibernate Criteria Query API).

El uso de interfaces de este tipo tiene como consecuencia una disminución en la claridad de la solución. Esto debido no solo al agregado en el código fuente de llamadas a operaciones definidas por el API en particular, sino también por la inclusión, en forma ajena al lenguaje de programación, de lenguajes de consulta como ser SQL, OQL, HQL, etc. Debido a que estos lenguajes de consulta no son parte del lenguaje de programación, el compilador no es capaz de realizar chequeos de correctitud de las construcciones de los mismos. Siendo necesario esperar a la ejecución del código para obtener *feedback* de los posibles errores sintácticos, de tipo, etc.

Es con esto en mente que surge la idea de contar con un lenguaje de programación que integre de forma nativa construcciones que permitan definir y ejecutar consultas sobre una fuente de datos. Permitiendo de esta forma uniformidad en el código, la realización de chequeos sintácticos y de tipo en tiempo de compilación como también la asistencia por parte de las herramientas de desarrollo (p.ej. autocompletación de código).

Descripción

La categoría en cuestión se refiere a la utilización de un lenguaje de programación que integre de forma nativa construcciones que permitan administrar y consultar los datos persistidos.

La mayoría de estos lenguajes se basan en la integración de un lenguaje de programación orientado a objetos (lenguaje *host*) y un lenguaje de consultas (lenguaje *guest*) ya populares. Esta integración tiene dos posibles combinaciones. La primera toma foco en un lenguaje de consultas ya existente y extiende el mismo con construcciones de un lenguaje de programación, agregando (entre otras cosas) la posibilidad de definición de funciones y estructuras de control necesarias. La segunda toma el camino opuesto, centrándose en un lenguaje de programación y agregando a este las construcciones necesarias para la creación de consultas. Nótese que para la categoría en cuestión solo considerará el segundo tipo de combinación ya que se considerara a la primera combinación como un caso particular de la utilización de otro mecanismo ya definido (como ser Acceso Directo a Bases de Datos). Considerando que el lenguaje integrado resulta de la extensión de un lenguaje de programación popular, existen por lo menos dos posibles estrategias de implementación del mismo. La primera (y mas directa) implica la inclusión de las nuevas construcciones en la gramática considerada por el compilador del lenguaje de programación (como es el caso de LINQ de Microsoft). Una segunda estrategia implica la implementación de un traductor del lenguaje integrado al lenguaje de programación host donde las construcciones de consultas agregadas al lenguaje son traducidas a llamadas a APIs de administración y manejo de datos (como es el caso de SQLJ de Oracle).

Contextos de Aplicación

- En el caso de contar con la libertad de elegir el lenguaje de programación orientado a objetos a utilizar en el proyecto.
- Si no se depende de herramientas CASE especificas al lenguaje host.

Ventajas

- Al reemplazar las llamadas a métodos de interfaces de acceso en el código fuente por construcciones propias del lenguaje se logra resolver la misma problemática en menos líneas de código. Al mismo tiempo, el código fuente es más fácil de comprender. Esto último es consecuencia de evitar el overhead (en código) generado por la invocación de métodos, pasaje de parámetros y manejo de excepciones de una interfaz de acceso. Las llamadas a operaciones son reemplazadas por construcciones del lenguaje integrado. Construcciones con semántica específica para la problemática de persistencia de datos y consulta de los mismos.
- Es posible realizar chequeos sintácticos y semánticos en tiempo de compilación valiéndose de la información de la fuente de datos a ser utilizada (por ejemplo valiéndose de una conexión a una base de datos relacional).

- Es posible realizar chequeos de tipos en tiempo de compilación del resultado de las consulta, así como también de los parámetros que son pasados a las mismas.

- De la misma forma que el compilador puede realizar chequeos en tiempo de compilación las herramientas de desarrollo pueden asistir al desarrollador por ejemplo a través de autocompletación de código.

- Debido a que muchos de los chequeos, son realizados en tiempo de compilación (en contrapartida a la utilización de un interfaz de acceso a datos), la utilización de un lenguaje integrado de consulta es potencialmente más performante.

- El desarrollador no necesita tener conocimientos tanto de un lenguaje de programación como también de uno de consulta. Solo es necesario concentrarse en aprender el lenguaje integrado (que potencialmente sea derivado de lenguajes de programación y consultas conocidos).

Desventajas

- Es posible que dependiendo del caso, al tratar de producir soluciones más simples para las tareas de administración y consulta de datos, se oculten (a los ojos del desarrollador) las actividades que realmente se están llevando acabo. De esta forma el desarrollador puede, por ejemplo, no percatarse del flujo de datos que esta siendo realmente transmitido desde el servidor de datos.

- El lenguaje de consulta integrado puede restringir las fuentes de datos con las cuales se puede interactuar. Por ejemplo SQLJ se reduce a proveer una forma de interacción con bases de datos relacionales a diferencia de LINQ que permite el uso de fuentes relacionales, XML y objetos en memoria.

- Al extender el lenguaje host, muchos de los editores y herramientas CASE para el mismo dejan de funcionar. Es necesario considerar herramientas específicas para el nuevo lenguaje extendido.

Mecanismo base

- Lenguajes Orientados a Objetos (Sección A.5).

Mecanismos Relacionados

- *Acceso Directo a Bases de Datos* (Sec A.1). La relación entre los lenguajes de consulta integrados y el acceso directo a bases de datos, es que la mayoría de estos lenguajes integrados se presentan como una alternativa más conveniente a la utilización del acceso directo. Al mismo tiempo las construcciones agregadas al lenguaje de programación host son derivadas de lenguajes de consultas populares como ser SQL, OQL, etc.

- *Mapeadores* (Sección A.2). En el caso particular de LINQ el lenguaje integrado de consulta trabaja a nivel de objetos, sin importar la fuente de datos (datos relacionales, XML, objetos), es por ello que el compilador puede ser visto como un mapeador de objetos a estas fuentes de datos.

Herramientas Representativas

- LINQ
  - Language Integrated Query (LINQ) es proyecto de Microsoft que extiende los distintos lenguajes .NET con una sintaxis para realizar consultas de propósito general. Dicha facilidad aplica a fuentes de datos como: datos relacionales, objetos o XML.
  - `http://msdn.microsoft.com/data/ref/linq`
- SQLJ
  - Se trata de un estándar para embeber sentencias SQL en programas Java que, a diferencia de JDBC, no es un API sino una extensión al lenguaje. Al utilizar SQLJ, antes de que las aplicaciones puedan ser compiladas, las mismas deben ser traducidas por un preprocesador denominado SQLJ Translator.
  - `http://www.oracle.com/technology/tech/java/sqlj_jdbc`

A.6. Prevalencia

Intención

Resolver la persistencia de objetos de forma tal que todos los objetos persistentes se encuentran en memoria principal.

Motivación

La mayoría de los programas operan sobre datos en memoria principal, ya que de esa forma logran procesar los datos más rápido que en un escenario donde se trabaja directamente en memoria secundaria. Al estar todos los objetos persistentes en memoria principal se aumenta considerablemente la *performance* del sistema. La estrategia de lograr la persistencia utilizando la memoria principal enfrenta dos grandes problemas, (1) el tamaño tradicionalmente chico de la memoria principal y (2) la volatilidad de la misma. La prevalencia de objetos surge como una propuesta diferente a utilizar bases de datos para lograr la persistencia.

Descripción

La técnica de prevalencia consiste en el mantenimiento de las instancias de objetos que componen a un sistema en memoria principal. Además, de forma periódica se persiste el estado de todos los objetos en memoria mediante técnicas de serialización, acompañadas del logueo de las transacciones realizadas. La persistencia del estado de los objetos se denomina *snapshot* de los objetos en memoria, y representará el último estado consistente de los datos.

El proceso de la toma de *snapshots* puede ser implementado de forma eficiente evitando el bloqueo del procesamiento de transacciones. Una posibilidad para ello consiste en mantener una réplica del sistema, de manera que todas las transacciones sean ejecutadas sobre el sistema original y la réplica del mismo en forma paralela. Al momento de realizarse la persistencia mediante serialización de objetos, la réplica es bloqueada para tomar un *snapshot*, mientras que el sistema original puede continuar funcionando con total normalidad.

Contextos de Aplicación

- Se aplica en sistemas donde sea factible mantener todos los objetos que componen el sistema en memoria principal.
- También es de utilidad en sistemas de tiempo real donde la *performance* es un requerimiento crítico. Esto se debe a que el manejo en memoria de los objetos mejora los tiempos de ejecución.

Ventajas

- Transparencia de la persistencia en el código de aplicación, permitiendo reducir el tamaño del código y su complejidad.
- Mejora de la *performance* de consultas y actualizaciones debido a que todos los datos son almacenados en memoria principal.
- La principal ventaja por sobre la utilización de los sistemas de serialización normalmente incluidos en los lenguajes orientados a objetos es la posibilidad de manejar transacciones y ser capaz de recuperar un estado consistente del sistema tras la caída del mismo (gracias a los *snapshots* y log de transacciones).
- Las consultas son simplemente procesos en memoria que acceden directamente a los objetos desde el lenguaje de programación considerado. No hay necesidad de incorporar un lenguaje de consulta.
- Mediante el uso de replicas del sistema (potencialmente remotas) se puede aumentar la disponibilidad del sistema, ya que al caerse el sistema original se puede pasar a utilizar cualquiera de las replicas disponibles.

Desventajas

- La cantidad de objetos que el sistema puede manejar está limitado por el tamaño de la memoria principal.
- El cumplimiento de las propiedades ACID del sistema quedan en manos del programador. Concretamente para cumplir con la propiedad de aislamiento, se debe recurrir a técnicas de manejo de concurrencia provistas por el lenguaje de programación.
- Requiere que el comportamiento de los objetos de negocio sea determinista. Esta propiedad es necesaria al momento de recuperar un estado consistente debido a una caída del sistema.
- Depende de que el lenguaje utilizado provea técnicas de serialización de objetos.
- Si se utiliza la estrategia de mantener varias replicas del sistema, la memoria principal requerida aumenta de forma lineal a la cantidad de replicas.

Mecanismo base

- No tiene.

Mecanismos Relacionados

- *Lenguajes Persistentes* (Sección A.5.2). Desde el punto de vista del desarrollador, la persistencia de los objetos de una aplicación se realiza de forma transparente, sin la necesidad de un manejador de base de datos (DBMS), que resuelva dicha persistencia. Si bien estas categorías son diferentes ninguna de las dos plantea la necesidad de un DBMS para persistir los datos.
- *Librerías Estándar* (Sección A.5.1). La prevalencia puede verse como una extensión a la utilización de librerías estándar. Más concretamente como una extensión del sistema de serialización de objetos.

Herramientas Representativas

- Prevayler
  - Prevalayer es una capa para la prevalencia de objetos escrita en Java que permite persistir POJOs de manera transparente. La herramienta, siendo además de código abierto, es la primera dentro de su naturaleza en estar disponible de manera pública.
  - `http://sourceforge.net/projects/prevayler`
- Bamboo Prevalence
  - Se trata de de una herramienta de código abierto que, al igual que Prevalayer, ofrece una capa para la prevalencia de objetos, en este caso, para la plataforma .NET.

- `http://sourceforge.net/projects/bbooprevalence`

Apéndice B

Evaluación de Criterios para Mecanismos de Persistencia

En base a los criterios definidos en 3.4.1 se presenta la evaluación de un subconjunto de los mismos. Téngase en cuenta que los criterios seleccionados son aquellos que se consideran de mayor interés. Concretamente fueron seleccionados: Transparencia, Persistencia por Alcance, Mecanismo de Consulta, Performance, Mantenibilidad de la Solución, Productividad y Madurez. Dejando fuera de la evaluación los siguientes criterios: Soporte Transaccional, Manejo de Impedance Mismatch, Soporte para Evolución de Esquema, Escalabilidad, Tecnologías Dependientes, Porte del Sistema, Existencia de Productos, Popularidad en el Mercado y Nivel de Estandarización. Nótese además que la naturaleza de muchos de ellos implicaría la definición de métricas complejas y una experimentación exhaustiva a fin logar una evaluación objetiva. De acuerdo con esto último y considerando el alcance del proyecto, la siguiente comparación resulta de la recolección de información disponible y opiniones de terceros. Se entiende sin embargo, que el resultado de estas evaluaciones son de valor al momento de contrastar los diferentes mecanismos de persistencia.

Incidencia en la Lógica del Sistema

La evaluación de Transparencia, Persistencia por Alcance y Mecanismo de Consulta se presenta en los Cuadros B.1, B.2 y B.3 respectivamente.

Cuadro B.1: Evaluación de Transparencia

Mecanismo	*Transparencia*

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Transparencia*
Acceso Directo a BD Relacional	Por definición del mecanismo los datos se acceden directamente desde la lógica orientada a objetos mediante SQL. Esto último implica que no se oculta de ninguna forma la naturaleza relacional de los datos. De acuerdo con esto el nivel de transparencia es nulo.
Acceso Directo a BD Objeto-Relacional	Por definición del mecanismo los datos se acceden directamente desde la lógica orientada a objetos mediante SQL. Esto último implica que no se oculta de ninguna forma la naturaleza relacional de los datos. De acuerdo con esto el nivel de transparencia es bajo.
Acceso Directo a BD Orientada a Objetos	Debido a que la base de datos subyacente trabaja directamente con objetos se puede obtener una solución mas transparente en comparación a los demás mecanismos de acceso directo. Nótese igual la posible existencia de consultas embebidas.
Acceso Directo a BD XML	Por definición del mecanismo los datos se acceden directamente desde la lógica teniendo que lidiar con la estructura de los documentos XML involucrados. De acuerdo con ello, el nivel de transparencia es bajo.
Mapeadores Objeto-Relacional	Los mapeadores suelen tener como objetivo la transparencia, por lo que su nivel tiende a ser alto. Dicho objetivo se suele lograr mediante el uso de APIs sencillos y la especificación de *metadata* de mapeo necesaria.
Mapeadores Objeto-XML	Los mapeadores suelen tener como objetivo la transparencia, por lo que su nivel tiende a ser alto. Dicho objetivo se suele lograr mediante el uso de APIs sencillos y la especificación de *metadata* de mapeo necesaria.

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Transparencia*
Generadores de Código	Depende del Generador.
Orientación a Aspectos	Implica la utilización de puntos de unión embebidos en el código fuente. Esto impacta negativamente en la transparencia.
Librerías Estándar	Depende de la Librería.
Lenguajes Persistentes	La transparencia es uno de los objetivos de los lenguajes persistentes. La otrogonalidad de tipo contribuye a esto último.
Lenguaje de Consulta Integrado	El impacto en la transparencia del código fuente es discutible debido a que, si bien son necesarios cambios en el mismo, los cambios implican el agregado de construcciones propias del lenguaje y no el uso de *frameworks* o librerías dedicadas a la persistencia. Mas allá de esta sutileza no se trata de una solución transparente en el sentido de que el desarrollador debe lidiar con particularidades del paradigma utilizado como destino de los datos.
Prevalencia	La transparencia es uno de los objetivos de prevalencia. La misma se ve afectada por el requerimiento de que todo objeto a ser persistido debe ser serializable.

Cuadro B.2: Evaluación de Persistencia por Alcance

Mecanismo	*Persistencia por Alcance*
Acceso Directo a BD Relacional	Los objetos a ser persistidos deben explicitarse. No existe la persistencia por alcance.
Acceso Directo a BD Objeto-Relacional	Los objetos a ser persistidos deben explicitarse. No existe la persistencia por alcance.
Acceso Directo a BD Orientada a Objetos	Aunque podría llegar a depender por ejemplo de la interfaz de acceso utilizada, se tiene que existe la persistencia por alcance.

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Persistencia por Alcance*
Acceso Directo a BD XML	Los objetos a ser persistidos deben explicitarse en formato XML. No existe la persistencia por alcance.
Mapeadores Objeto-Relacional	Implementan persistencia por alcance persistiendo todos aquellos objetos referenciados por las raíces de persistencia. Teniendo la posibilidad de indicar que ciertos objetos referenciados no deben ser persistidos.
Mapeadores Objeto-XML	Implementan persistencia por alcance persistiendo todos aquellos objetos referenciados por las raíces de persistencia. Teniendo la posibilidad de indicar que ciertos objetos referenciados no deben ser persistidos.
Generadores de Código	Depende del Generador.
Orientación a Aspectos	No Aplica.
Librerías Estándar	Depende de la Librería.
Lenguajes Persistentes	Presenta persistencia por alcance. Es uno de los tres principios que debe cumplir según la definición del mecanismo.
Lenguaje de Consulta Integrado	Depende del caso particular.
Prevalencia	Dado que la técnica persiste todas las clases alcanzables desde el sistema prevalente (contenedor de objetos), presenta persistencia por alcance.

Cuadro B.3: Evaluación de Mecanismo de Consulta

Mecanismo	*Mecanismo de Consulta*
Acceso Directo a BD Relacional	Las consultas se realizan mediante SQL.
Acceso Directo a BD Objeto-Relacional	Las consultas se realizan fundamentalmente mediante SQL3.
Acceso Directo a BD Orientada a Objetos	Dependiendo de la interfaz de acceso podrían tenerse distintas formas de consultar la base. De manera general se distinguen lenguajes de consulta como JDOQL u OQL y las llamadas consultas nativas (realizadas de forma programática).

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Mecanismo de Consulta*
Acceso Directo a BD XML	De manera general, las consultas se realizan mediante XQuery o XPath (un subconjunto del anterior).
Mapeadores Objeto-Relacional	Suelen proveer lenguajes de consulta propietarios o estandarizados a nivel de objetos (p.ej. OQL) al mismo tiempo que es posible realizar consultas directas a la base de datos relacional. En algunos casos se incluye un api de formulación de consultas.
Mapeadores Objeto-XML	Existen lenguajes de consulta como ser XQuery, XML-QL o XPath que se pueden utilizar para consultar los documentos XML de forma directa. Algunos mapeadores proveen un lenguaje de consulta a nivel de objetos.
Generadores de Código	Depende del Generador.
Orientación a Aspectos	No Aplica.
Librerías Estándar	Depende de la Librería.
Lenguajes Persistentes	Dada la naturaleza del lenguaje, el mecanismo de consulta se encuentra implícito. Para recuperar información contenida en el sistema, simplemente se accede a los objetos a través de la raíz de persistencia.
Lenguaje de Consulta Integrado	Utilización del lenguaje de consulta integrado al lenguaje de programación.
Prevalencia	No se dispone de un mecanismo de consulta. Al estar todos los objetos en memoria principal, el acceso a los mismos es directo.

Resultado de su Aplicación

La evaluación de Performance y Mantenibilidad de la Solución se presenta en el Cuadros B.4 y B.5 respectivamente.

Cuadro B.4: Evaluación de Performance

Mecanismo	*Performance*

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Performance*
Acceso Directo a BD Relacional	Muy buen nivel de desempeño debido a que el acceso se realiza de forma directa sin tener que pasar por ningún tipo de *middleware*. Nótese que se supone la existencia de un controlador y un DBMS eficientes para brindar acceso.
Acceso Directo a BD Objeto-Relacional	Muy buen nivel de desempeño debido a que el acceso se realiza de forma directa sin tener que pasar por ningún tipo de *middleware*. Nótese que se supone la existencia de un controlador y un DBMS eficientes para brindar acceso.
Acceso Directo a BD Orientada a Objetos	Muy buen nivel de desempeño debido a que el acceso se realiza de forma directa sin tener que pasar por ningún tipo de *middleware*. Nótese que se supone la existencia de un controlador y un DBMS eficientes para brindar acceso.
Acceso Directo a BD XML	Buen nivel de desempeño debido a que el acceso se realiza de forma directa sin tener que pasar por ningún tipo de *middleware*. Tenganse en cuenta que el mismo puede variar dependiendo se trate de una base de datos XML-native o XML-enabled.
Mapeadores Objeto-Relacional	No se comportan bien en procesos de escritura por lotes. La caída en la *performance* en comparación al acceso directo a una base de datos relacional se debe al *overhead* producido por el mapeo entre ambos paradigmas. Suelen comportarse bien en aplicaciones `READ-CENTRIC` o que sigan un ciclo de vida tipo `READ-MODIFY-WRITE`.
Mapeadores Objeto-XML	La *performance* depende del *parser* de XML utilizado como también de la implementación particular del mapeo.
Generadores de Código	Depende del Generador.
Orientación a Aspectos	No Aplica.
Librerías Estándar	Depende de la Librería.
Lenguajes Persistentes	El decremento en la *performance* a causa de cumplir con los tres principios de persistencia ortogonal es una de las principales desventajas de los lenguajes persistentes.

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Performance*
Lenguaje de Consulta Integrado	La *performance* se ve influenciada por el tipo de lenguaje de consulta integrado y si existe o no algún tipo de mapeo realizado de forma automática entre el modelo de objetos y el paradigma de destino de los datos.
Prevalencia	Alta performance debido a que los cambios de estado de los objetos se realizan en memoria principal.

Cuadro B.5: Evaluación de Mantenibilidad de la Solución

Mecanismo	*Mantenibilidad de la Solución*
Acceso Directo a BD Relacional	Es complicado diseñar soluciones mantenibles ya que el código necesario para el acceso a datos se entrelaza con el código de la lógica de negocios.
Acceso Directo a BD Objeto-Relacional	Es complicado diseñar soluciones mantenibles ya que el código necesario para el acceso a datos se entrelaza con el código de la lógica de negocios.
Acceso Directo a BD Orientada a Objetos	Alta mantenibilidad. El enfoque puramente de objetos en la lógica y la ausencia del *impedance mismatch* facilita el mantenimiento.
Acceso Directo a BD XML	Es complicado diseñar soluciones mantenibles ya que el código necesario para el acceso a datos se entrelaza con el código de la lógica de negocios.
Mapeadores Objeto-Relacional	Es ciertamente más mantenible que una estrategia de acceso directo a una base de datos. El enfoque puramente de objetos en la lógica facilita el mantenimiento.
Mapeadores Objeto-XML	Es ciertamente más mantenible que una estrategia de acceso directo a una base de datos. El enfoque puramente de objetos en la lógica facilita el mantenimiento.

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Mantenibilidad de la Solución*
Generadores de Código	Se producen soluciones que por la naturaleza del mecanismo son altamente mantenibles.
Orientación a Aspectos	Se facilita el mantenimiento al tener localizado dónde realizar los cambios relativos a la persistencia.
Librerías Estándar	Depende de la Librería.
Lenguajes Persistentes	La utilización del mecanismo lleva a soluciones altamente mantenibles.
Lenguaje de Consulta Integrado	En principio supone que la mantenibildad de una solución de este tipo es mayor a una realizada utilizando el mismo lenguaje de consulta a través de algún API. Esto debido a que la integración del lenguaje de consulta al lenguaje de programación habilita a los IDEs a proveer asistencia adecuada al desarrollador.
Prevalencia	Las aplicaciones son mantenibles, por la naturaleza del mecanismo.

Incidencia en el Proyecto

La evaluación de Productividad se presenta en el Cuadro B.6.

Cuadro B.6: Evaluación de Productividad

Mecanismo	*Productividad*
Acceso Directo a BD Relacional	Baja.
Acceso Directo a BD Objeto-Relacional	Baja.
Acceso Directo a BD Orientada a Objetos	Alta.
Acceso Directo a BD XML	Baja.
Mapeadores Objeto-Relacional	Alta.

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Productividad*
Mapeadores Objeto-XML	Alta.
Generadores de Código	Alta.
Orientación a Aspectos	Depende.
Librerías Estándar	Depende de la Librería.
Lenguajes Persistentes	Alta.
Lenguaje de Consulta Integrado	Alta/Media dependiendo del caso.
Prevalencia	Alta.

Vista de Mercado

La evaluación de Madurez se presenta en el Cuadro B.7.

Cuadro B.7: Evaluación de Madurez

Mecanismo	*Madurez*
Acceso Directo a BD Relacional	Presenta un gran nivel de madurez debido a su temprano surgimiento, su éxito logrado en el correr de los años y la amplia investigación y desarrollo en el campo.
Acceso Directo a BD Objeto-Relacional	Presenta un buen nivel de madurez debido a su temprano surgimiento, su éxito logrado en el correr de los años y la amplia investigación y desarrollo en el campo.
Acceso Directo a BD Orientada a Objetos	Presenta creciente nivel de madurez.
Acceso Directo a BD XML	Se considera que XML, y por ende, las tecnologías que dependen de él no han alcanzado un nivel de madurez considerable. Esto se debe fundamentalmente a su reciente surgimiento.

Continúa en la página siguiente...

Continúa de la página anterior...

Mecanismo	*Madurez*
Mapeadores Objeto-Relacional	Moderada, ya que en los últimos seis años se ha investigado y desarrollado mucho en el tema.
Mapeadores Objeto-XML	Poca. A pesar de esto es un área en la cual existe mucho interés actualmente, habiendo surgido una serie de herramientas en los últimos años que continúan su evolución.
Generadores de Código	Se trata de un campo que no ha cobrado interés sino hasta los últimos años. De acuerdo con ello no se considera un mecanismo maduro.
Orientación a Aspectos	Es una tecnología emergente en desarrollo.
Librerías Estándar	Generalmente su nivel de madurez depende de la madurez del lenguaje involucrado.
Lenguajes Persistentes	No presenta madurez alguna, dado que no ha salido de los límites académicos y teóricos.
Lenguaje de Consulta Integrado	Es una tecnología emergente en desarrollo.
Prevalencia	Al ser un mecanismo emergente, no se considera maduro.

Apéndice C

Mapeadores Objeto-Relacional Considerados

C.1. Hibernate 3

C.1.1. Introducción

Hibernate es un mapeador objeto-relacional gratuito y de código libre para la plataforma Java. Uno de sus propósitos fundamentales es de librar al desarrollador de una significativa cantidad de tareas relacionadas con la persistencia de datos. Hibernate se adapta al proceso de desarrollo ya sea creando un nuevo esquema de base de datos o haciendo uso de uno ya existente. No solo se ocupa del mapeo de clases Java a tablas en la base de datos (y de tipos Java a tipos SQL), sino que también ofrece un lenguaje de consulta propio y orientado a objetos (HQL) que reduce el tiempo de desarrollo. De otra forma, dicho tiempo de desarrollo debería ser empleado en realizar el manejo manual de JDBC y SQL. Hibernate se dedica a generar las sentencias SQL correspondientes y evita que el desarrollador tenga que hacer la conversión manual a objetos. De esta manera la aplicación se mantiene portable a todos los DBMSs basados en SQL.

En vez de la gran cantidad de clases y propiedades de configuración requeridas por otras soluciones de persistencia (como EJBs), Hibernate necesita un único archivo de configuración y un documento de mapeo por cada tipo de objeto a ser persistido. El archivo de configuración puede especificarse como un archivo de propiedades clave-valor o mediante XML con un archivo que comúnmente se nombra `hibernate.cfg.xml` (se supondrá esta opción de aquí en más). Alternativamente, se podría realizar la configuración de manera programática. En cuanto a los archivos de mapeo (en formato XML), los mismos pueden llegar a ser muy pequeños dejando que la herramienta defina el resto. Opcionalmente se podría proveer más información como por ejemplo un nombre alternativo para determinada columna de cierta tabla.

Al hacer uso de otro *framework* de persistencia la aplicación desarrollada puede quedar fuertemente acoplada al mismo. Hibernate no crea esta dependencia adicional; los objetos persistentes no tienen que heredar de una clase particular de Hibernate u obedecer alguna

semántica específica. Asimismo no se requiere ningún tipo especial de contenedor Java EE para funcionar. Hibernate puede ser utilizado en cualquier entorno de aplicación (ya sea en una aplicación *standalone* o en el contexto de un servidor de aplicaciones).

C.1.2. Configuración

Antes de proceder con la configuración de Hibernate es necesario decidir cómo el servicio de persistencia obtendrá las conexiones a la base de datos. Concretamente, los opciones pueden ser utilizar las conexiones JDBC provistas por Hibernate o las ofrecidas por un JNDI `DataSource`. Un tercer método, conexiones JDBC provistas por el usuario, se encuentra disponible pero es rara vez utilizado. La primer alternativa es la que se utilizará de aquí en mas y típicamente suele ser la elección para ambientes que no cuentan con un servidor de aplicaciones (p.ej. una aplicación *standalone*).

En el listado C.1 se muestra un ejemplo del archivo de configuración principal para Hibernate. Para hacer uso de las conexiones provistas por el *framework* se necesitan especificar las siguientes cinco propiedades: `connection.driver_class` (driver JDBC para la base de datos utilizada), `connection.url` (dirección JDBC para la base), `connection.username` (nombre de usuario), `connection.password` (contraseña) y `dialect` (nombre del dialecto SQL particular a utilizar). Por otro lado, Hibernate necesita saber las ubicaciones y nombres de los archivos de mapeo que describen las clases persistentes. El elemento `mapping` se utiliza para este propósito.

C.1.3. Especificación de Mapeos

Los documentos de mapeo son utilizados fundamentalmente para brindar a Hibernate la información necesaria para persistir objetos en una base de datos relacional. En el Listado C.2 se muestra un ejemplo concreto para la clase llamada `Reserva`.

El elemento `id` se utiliza para describir la clave primaria de la clase persistente así como también el método de generación para la misma. Toda clase persistente debe contar con un elemento `id` que declare la clave primaria para la tabla relacional. Los elementos `property` permiten indicar, como su nombre lo indica, qué propiedades se persisten. Asimismo, las asociaciones pueden ser mapeadas, según aplique el caso, mediante alguna de las siguientes construcciones: `one-to-one`, `many-to-many`, `one-to-many` o `many-to-one`. Un característica muy útil de Hibernate son las denominadas *cascadas* también configuradas a nivel del documento de mapeo. Las mismas permiten propagar ciertas operaciones sobre una tabla (p.ej. un `delete` o `save`) a sus tablas asociadas.

La herencia es un concepto fundamental de los lenguajes orientados a objetos. Persistir una jerarquía particular puede ser complicado ya que cada caso particular presenta sus propios requerimientos. A modo de solucionar los problemas en este aspecto, Hibernate provee diferentes estrategias de persistencia para herencias. Concretamente permite al desarrollador utilizar (en creciente orden de complejidad) una única tabla por jerarquía de clases, una tabla particular por subclase o una tabla por clase concreta. En el Listado C.3 se muestra un ejemplo de la primer opción. Específicamente se hace uso del elemento `discriminator` que permite indicar qué columna se utiliza para identificar el tipo con-

Listado C.1: Ejemplo de archivo de configuración hibernate.cfg.xml.

```
<?xml version="1.0"?>
<!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="connection.username">postgres</property>
      <property name="connection.password">postgres</property>
      <property name="connection.url">
         jdbc:postgresql://localhost:5432/sgh_hibernate
      </property>
      <property name="connection.driver_class">
         org.postgresql.Driver
      </property>
      <property name="dialect">
         org.hibernate.dialect.PostgreSQLDialect
      </property>
      <mapping resource="Cliente.hbm.xml"/>
      <mapping resource="Habitacion.hbm.xml"/>
      <mapping resource="Hotel.hbm.xml"/>
      <mapping resource="Recepcion.hbm.xml"/>
      <mapping resource="TipoHabitacion.hbm.xml"/>
      <mapping resource="Huesped.hbm.xml"/>
      <mapping resource="Reserva.hbm.xml"/>
   </session-factory>
</hibernate-configuration>
```

Listado C.2: Ejemplo de documento de mapeo para la clase Reserva.

```
<?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="edu.fing.uy.proygrado06.sgh.negocio.reservas.
    entidades">
    <class name="Reserva" table="reservas">
        <id name="reservaId" type="integer" unsaved-value="null" access="
            field">
            <generator class="increment"/>
        </id>
        <property name="checkIn" type="date" access="field"/>
        <property name="checkOut" type="date" access="field"/>
        <property name="estado" access="field">
            <type name="edu.fing.uy.proygrado06.sgh.utils.EnumUserType">
                <param name="enumClassName">
                    edu.fing.uy.proygrado06.sgh.negocio.reservas.
                        constantes.EstadoReserva
                </param>
            </type>
        </property>
        <property name="hotelId" type="string" access="field"/>
        <property name="habitacion" type="string" access="field"/>
        <property name="tipoHab" type="string" access="field"/>
        <property name="clienteId" type="integer" access="field"/>
        <list name="huespedes" access="field" cascade="all">
            <key column="reserva_id"/>
            <index column="indice" />
            <one-to-many class="edu.fing.uy.proygrado06.sgh.negocio.
                reservas.entidades.Huesped" />
        </list>
    </class>
</hibernate-mapping>
```

creto de instancia. Téngase en cuenta que en este caso, las columnas que no apliquen a determinada instancia serán nulas.

Listado C.3: Ejemplo de mapeo de herencia utilizando una tabla por jerarquía.

```
...
   <class name="Cliente" table="clientes" discriminator-value="CLIENTE">
      <id name="codigo" type="integer" unsaved-value="null" access="field"
          >
         <generator class="increment"/>
      </id>
      <discriminator column="tipo" type="string"/>
      <property name="nombre" type="string" access="field"/>
      ...
      <property name="fax" type="string" access="field"/>
      <subclass name="AgenciaDeViajes" discriminator-value="
          AGENCIA_DE_VIAJES">
         <property name="ruc" type="string" access="field"/>
      </subclass>
   </class>
...
```

C.1.4. API de Hibernate

La interface `SessionFactory` permite obtener instancias de la clase `Session`, que representa una conexión con la base de datos. Las instancias de `SessionFactory` son *thread-safe* y típicamente se comparten a lo largo de una aplicación. Por el otro lado, las de `Session` no cumplen con ésta característica y deberían ser utilizadas únicamente para una transacción. La clase `Configuration` se encarga de lanzar la porción *runtime* de Hibernate. La misma se utiliza para cargar los archivos de mapeo y crear una `SessionFactory` para estos últimos.

Persistir un objeto por primera vez con Hibernate es tan simple como invocar el método `save(...)` de la interface `Session`. En caso de necesitar actualizar uno ya existente se debe recurrir a `update(...)`. Para simplificar la tarea al desarrollador se provee el método `saveOrUpdate(...)` que determina la operación correcta a ejecutar dependiendo del caso. Esto último se determina haciendo uso del atributo `unsaved-value` del elemento `id`, especificado en el documento de mapeo de la clase correspondiente. Tomando en cuenta el listado C.2 y suponiendo se tenga una instancia de `Reserva`: Hibernate asume que el objeto no ha sido persistido (se utiliza `save`) si el valor de `codigo` es `null` y asume que el objeto ya está en la base de datos (se utiliza `update`) si el valor del atributo es distinto de `null`. Se muestra en el Listado C.4 un ejemplo típico para la persistencia de un objeto.

En lo que respecta a la recuperación de un objeto persistido existen distintos mecanismos. En caso de contar con el identificador del objeto que se desea obtener puede hacerse uso del método `load(...)` de la sesión que se esté utilizando. Si no se cuenta con dicho identificador debe recurrirse al lenguaje de consulta provisto por Hibernate. El *Hibernate Query Language* (HQL) es un lenguaje de consulta similar a SQL pero que a diferencia

Listado C.4: Ejemplo típico para la persistencia de un objeto.

```
Configuration cfg = new Configuration();
SessionFactory factory = cfg.buildSessionFactory();

Reserva reserva = new Reserva();
...

Session sesion = factory.openSession();
sesion.saveOrUpdate(reserva);
sesion.flush();
sesion.close();
```

de éste es orientado a objetos. En lo que respecta al API propiamente dicho, un objeto `Session` permite crear objetos `Query` a fin de recuperar las entidades requeridas. Se destacan a continuación algunas de las características más importantes de HQL:

- Habilidad de aplicar restricciones a propiedades de objetos asociados ya sea por referencia o contenidos en colecciones.
- Capacidad de recuperar solo las propiedades de una entidad o entidades.
- Posibilidad de ordenar y paginar los resultados de una consulta.
- Agregación con `group by`, `having` y funciones agregadas como `sum`, `min` y `max`.
- *Outer Joins* al recuperar múltiples objetos por fila.
- Habilidad de ejecutar funciones SQL definidas por el usuario.
- Soporte para subconsultas.

C.2. Java Persistence API (JPA) / TopLink Essentials

C.2.1. Introducción

Con el lanzamiento de la versión 5 de la plataforma Java Enterprise Edition (Java EE) se introducen una serie de cambios y mejoras orientadas a disminuir la complejidad asociada al desarrollo. Esto puede verse reflejado en una considerable disminución de la cantidad de código necesaria para llevar adelante determinado proyecto. A pesar de los cambios realizados a la plataforma, el potencial de la misma en cuanto a las funcionalidades ofrecidas no se vio afectado.

Una de las mejoras más interesantes refiere al agregado (como parte de la especificación de EJB 3.0) de Java Persistente API (JPA), especificación que simplifica la persistencia de Plain Old Java Objects (POJOs) al mismo tiempo que agrega nuevas funcionalidades. JPA pone énfasis en definir la *metadata* mínima necesaria para el mapeo de objetos Java a una base de datos relacional. Su objetivo fundamental es el de convertirse en el estándar por

excelencia para la utilización de un mapeador objeto-relacional dentro de la plataforma Java. De acuerdo a la filosofía de la plataforma esta especificación presta especial atención a la simplificación de la tarea de solucionar la persistencia de un sistema.

El estándar en cuestión es el resultado del trabajo de un grupo de expertos en el área quienes tomaron en consideración ideas de *frameworks* líderes del mercado y otras especificaciones existentes como ser: Hibernate, Oracle TopLink, Java Data Objects (JDO), entre otros. La intención detrás de su creación es que el mismo sea implementado por diferentes proveedores logrando así que diferentes productos ofrezcan una misma interfaz cuya semántica pueda ser asumida por las aplicaciones. Cabe destacar que la interfaz en cuestión se encuentra diseñada para ser utilizada tanto en un ambiente Java EE como Java SE.

Debido a que JPA es solamente una especificación, para solucionar la persistencia de un sistema se debe decidir qué implementación de dicha especificación ha de usarse. Las diferentes implementaciones son denominadas (dentro de la especificación) *proveedores de persistencia*. El proveedor de persistencia particular seleccionado es el provisto por Oracle en su producto TopLink Essentials. TopLink Essentials es parte del servidor de aplicaciones Glassfish que pretende ser un servidor de aplicaciones de código abierto y calidad empresarial. Téngase en cuenta que TopLink Essentials es la versión gratuita del producto TopLink de Oracle y que además es la implementación de referencia para la especificación JPA.

Las diferentes propiedades del mapeador a utilizar son el resultado del par JPA (especificación), TopLink Essentials (implementación). Todas las características que se refieran a la interacción con el mapeador y la definición de la *metadata* de mapeo están condicionadas a la especificación JPA. Por otro lado, otras características (p.ej. *performance*) y detalles de implementación que no son referidos por la especificación dependen totalmente del proveedor de persistencia seleccionado.

C.2.2. Características Generales

A continuación se presentan algunos conceptos y características de JPA.

Persistencia de POJOs

Uno de los aspectos más importantes de JPA es que los objetos a ser persistidos son POJOs, lo que quiere decir que no es necesario tener ninguna consideración especial sobre los mismos (como ser extender cierta clase, implementar cierta interfaz, implementar determinados métodos, etc.). Debido a que el mapeo es totalmente *metadata-driven*, cualquier objeto puede ser persistido sin tener que cambiar ninguna línea de código de la clase a la que pertenece (aun así es necesario valerse de algún método para especificar la *metadata* de mapeo).

Persistencia No-Intrusiva

Se categoriza a JPA como persistencia *no-intrusiva* debido en parte a que, si consideramos la arquitectura de capas que compone un sistema de información clásico, el API de persistencia se encuentra en una capa diferente a la de los objetos de la lógica de negocio que son persistidos. La capa de persistencia, compuesta en este caso por JPA, es llamada por la lógica de negocio la cual pasa los objetos de negocio como parámetros e instruye a la capa de persistencia sobre como operar sobre los mismos. Por lo tanto, mas allá de que la lógica de negocio debe estar consciente de la utilización de JPA, los objetos persistentes no lo están. Es por todo esto que JPA se considera un enfoque *no-intrusivo*.

Consulta de Objetos

Un mapeador objeto-relacional potente debería ofrecer la posibilidad de realizar consultas sobre los objetos persistidos y sus relaciones sin necesidad de recurrir a consultas SQL sobre la base de datos a la cual los objetos son mapeados. Es así que JPA provee un lenguaje de consultas propio (a nivel de objetos) denominado JP-QL (Java Persistence Query Language. Este lenguaje esta basado en EJB-QL y SQL pero no esta atado al modelo relacional (ya que trabaja dentro del paradigma de objetos). Las consultas en JP-QL, se basan en el estado de los objetos persistentes y no en las columnas en las cuales la entidad es persistida. Es por ello que la construcción de una consulta no requiere conocimiento de la *metadata* de mapeo (solo se necesita conocer el modelo de objetos) y normalmente devuelve su resultado en forma de objetos.

Una consulta puede estar definida de forma estática (en la *metadata*) o ser creada de forma dinámica (en tiempo de ejecución). Las consultas JP-QL representan una poderosa abstracción que permite realizar consultas sobre el modelo del dominio y no sobre tablas de la base de datos.Es también posible bajar al nivel relacional y realizar consultas en SQL (de ser necesario).

Simplicidad de Configuración

La especificación de la *metadata* de mapeo se realiza a través de anotaciones de Java 5 [KS06, Pág. 20], XML, o una combinación de ambos métodos. El primer método se basa en una novedad introducida en Java 5 que permite agregar *metadata* en el propio código fuente valiéndose de construcciones propias del lenguaje. Las anotaciones ofrecen una gran simplicidad y facilidad de uso, disminuyendo considerablemente la curva de aprendizaje. El segundo método se asemeja a las técnicas de definición de *metadata* comúnmente utilizadas por los *frameworks* de mapeo objeto-relacional populares (p.ej. Hibernate). Más allá de la facilidad del método de especificación de la *metadata* se destaca el amplio uso de *configuraciones por omisión*[1] que permite obviar mucha de la información de mapeo. Las configuraciones por omisión permiten que la cantidad de *metadata* que debe ser especificada por el desarrollador sea mínima.

[1]Valores predeterminados para los deferentes parámetros de configuración del mapeo objeto-relacional

C.2.3. Utilización de JPA

Esta sección presenta conceptos y consideraciones a la hora de aplicar JPA para solucionar las necesidades de persistencia de un sistema.

Entidades

Los objetos java que sean instancias de clases a las cuales se les a agregado la *metadata* necesaria para el mapeo a una base de datos relacional, son denominados por JPA como *entidades*. Estos objetos llamados entidades tienen ciertas características:

- *Persistibles*. La principal característica de las entidades es que son persistibles, lo que quiere decir que estos objetos pueden ser persistidos a una base de datos relacional para ser recuperados en el futuro (extendiendo su ciclo de vida mas allá al fin del proceso que los creo). El control de la persistencia de una entidad esta totalmente en las manos de la aplicación, esto quiere decir que una ves instanciada una entidad, esta no es persistida automáticamente. La persistencia de una entidad debe ser iniciada explícitamente por la aplicación mediante una llamada al API.
- *Identidad*. Cuando una entidad de encuentra en su forma persistente, la misma debe ser capaz de mantener su identidad (al igual que en su forma de objeto). Para esto es que se considera una identidad persistente llamada también identificador que identifica a una entidad en particular y la diferencia del resto de las entidades del mismo tipo. Es posible delegar la responsabilidad de manejar esta identidad persistente al proveedor de persistencia. Esto se logra indicando una estrategia de generación de identificadores (Listados C.5 y C.6)
- *Transaccionabilidad*. Las entidades son normalmente creadas, actualizadas y eliminadas dentro de una transacción, esto es requerido para que los cambios tengan impacto en la base de datos. El manejo de las transacciones, en un contexto de Java SE, deben ser realizados de forma explicita por la lógica de negocio. Los cambios incluidos en una transacción son exitosos o fallan de forma atómica. Sin embargo en memoria las entidades pueden cambiar sin que estos cambios sean persistidos. En caso de un *rollback* o una falla de la transacción, las entidades en memoria pueden quedar en un estado inconsistente y no deben ser utilizadas por la aplicación. Las entidades en memoria son simples objetos java, por lo que obedecen a todas las reglas y restricciones establecidas por la JVM (Java Virtual Machine) (Listado C.7).

Metadata

Cada entidad tiene asociada cierta cantidad de *metadata* que le permite a la capa de persistencia reconocer e interpretar la estructura de la entidad y de esta forma permitir el mapeo de la misma desde y hacia la fuente de datos relacional. Esta *metadata* incluye información de los atributos persistentes de la entidad, las relaciones en que participa y de que forma lo hace, entre otras cosas. La *metadata* requerida por JPA para la definición de

Listado C.5: Ejemplo de la definición de un identificador que es asignado de forma manual en tiempo de ejecución.

```
@Entity
...
public class Hotel {

   @Id
   private String hotelId;
   ...
}
```

Listado C.6: Ejemplo de definición de un Identificador con una estrategia de generación automática basada en la utilización de una tabla de generación.

```
@Entity
...
public class Reserva {

   @TableGenerator(name = "RESERVA_GEN")
   @Id
   @GeneratedValue(generator = "RESERVA_GEN")
   private int reservaId;
   ...
}
```

Listado C.7: Ejemplo de manejo de una transacción en un proceso de negocio.

```
public void tomarReserva(int id, Set<DatosHuesped> sdh)
   throws FaltanDatos, ReservaNoExiste, EstadoReservaIncorrecto,
   FaltanDatosHuesped {

   EntityManager em = PersistenciaUtil.getCurrentEM();
   try {
      em.getTransaction().begin();

      FachadaNegocio.getInstance().getIManejadorReservas().tomarReserva(id
          , sdh);

      em.getTransaction().commit();

   } finally {
      if (em.getTransaction().isActive()) {
         em.getTransaction().rollback();
      }
      PersistenciaUtil.closeCurrentEM();
   }
}
```

una entidad es mínima, pudiendo extenderse especificando cada detalle (de ser necesario) como ser: configuraciones de comportamientos específicos a la entidad (p.ej. definir una atributo como *lazy*, indicar que una operación sobre la entidad debe ser propagada a las entidades de una de sus relaciones, etc.) y particularidades del mapeo del estado de la misma (p.ej. nombre de tablas, columnas,etc). La cantidad de *metadata* dependerá de los requerimientos de la aplicación y la complejidad del dominio. Como se mencionó antes, esta *metadata* puede ser especificada en dos formas: haciendo uso de anotaciones o XML (Listado C.8).

Configuración por omisión

La configuración por omisión es uno de los grandes pilares de JPA y se refiere a que la especificación define valores por defecto para todas aquellas propiedades de mapeo y configuración para las cuales haya un valor obvio, ya sea por ser el valor mas utilizado en el caso general o por ser posible derivar un valor válido para esta propiedad (valor posiblemente derivado del modelo de dominio). Esta característica de JPA reduce considerablemente la cantidad de *metadata* que debe ser especificada facilitando el desarrollo. Mas allá de esta cualidad positiva de la configuración por defecto existe la desventaja de que al no ser necesario la especificación explícita de mucha de la *metadata* el desarrollador puede desconocer las opciones por defecto que están siendo aplicadas. Esto puede traducirse en problemas de *performance* o la ilusión del desarrollador de no poder personalizar cierto comportamiento particular. Como conclusión se puede decir que la configuración por omisión es una herramienta muy poderosa, al mismo tiempo que es peligrosa. Para evitar problemas el desarrollador debería estar consciente en todo momento de los valores por defecto que están siendo aplicados en cada caso que estos se omitan.

C.3. LLBLGen Pro

C.3.1. Introducción

LLBLGen Pro es un mapeador objeto-relacional (ORM) que utiliza generación de código para producir la capa de acceso a datos para la plataforma .NET. Se trata de una herramienta no gratuita, que no necesita de un IDE para su ejecución. La utilización de dicho mapeador apunta a incrementar la productividad del desarrollo de una aplicación, permitiendo al desarrollador concentrarse en la lógica del negocio. Soporta varios motores de base de datos como ser SqlServer, Oracle, PostgreSql y MySql.

Para poder utilizar LLBLGen Pro, se debe primero diseñar y crear el esquema de la base de datos, sobre el cual trabajará el mapeador. Una vez configurado el mapeo, se podrá generar el código en C# o VisualBasic .NET, correspondiente a la capa de acceso a datos. También generará plantillas sobre las clases de la lógica del negocio que serán persistidas. Toda clase generada podrá ser modificada en regiones particulares. Esto permitirá ante cambios en la definición del mapeo y regeneración de código, mantener el código agregado en las antedichas regiones.

Listado C.8: Ejemplo de *metadata* de mapeo de una entidad en JPA.

```
@Entity
@NamedQueries( {
            @NamedQuery(name = "Reserva.reservasEnHotelEnEstado",
            query = "SELECT  r FROM Reserva r "
                        + "WHERE r.hotelId=:hotelId AND r.estado=:estado")
                            ,
            ...

})
public class Reserva {

    @TableGenerator(name = "RESERVA_GEN")
    @Id
    @GeneratedValue(generator = "RESERVA_GEN")
    private int reservaId;

    @Temporal(TemporalType.DATE)
    private Date checkIn;

    @Temporal(TemporalType.DATE)
    private Date checkOut;

    @Enumerated(EnumType.STRING)
    private EstadoReserva estado;

    private String hotelId;

    private String habitacion;

    private String tipoHab;

    private int clienteId;

    @OneToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE,
    CascadeType.REMOVE })
    @OrderBy("nombre ASC")
    private List<Huesped> huespedes;

    public Reserva() {
        huespedes = new ArrayList<Huesped>();
        estado = EstadoReserva.Pendiente;
    }

    ...
}
```

C.3.2. Utilización de LLBLGen Pro

En esta sección se presentan los conceptos que se deben considerar para la correcta definición del mapeo objeto-relacional y posterior utilización del código generado.

Conceptos

- *Entidades.* Para cada tabla de la base de datos del proyecto al cual se le realizará el mapeo, se define una entidad específica que representa dicha tabla. Cada instancia de la entidad definida, se corresponde con una fila de la tabla relacionada, y las propiedades que en la entidad se definen, serán las columnas de la tabla.
- *Colecciones.* Para cada entidad que se define, LLBLGen Pro crea también una entidad colección, que es un contenedor de las instancias de la entidad a la que se relaciona. La utilidad de esta clase de entidades es que permite obtener un conjunto de entidades a partir de ciertos filtros que se aplican sobre todos las instancias de una entidad.
- *Relaciones.* Para cada entidad definida, se pueden definir las relaciones con otras entidades, que permitirán luego navegar por las entidades relacionadas. Dependiendo del tipo de relación que se defina, se podrá acceder a una entidad simple o a una entidad colección.
- *Vistas Tipadas.* LLBLGen Pro permite definir vistas de la base de datos como entidades en el proyecto. Estas entidades heredan de la clase DataTable, y permiten definir el contenido de cada columna en una vista.
- *Listas Tipadas.* Esta clase permite definir entidades que no se corresponden de forma exacta con una tabla de la base de datos. De esta forma será posible definir listas que contengan un subconjunto de columnas de una o más tablas, accediendo a la información necesaria, sin la necesidad de obtener todas las columnas de las tablas a considerar.
- *Entidades invocadoras de Store Procedures.* Para facilitar el acceso a los Store Procedures definidos en la base de datos, LLBLGen Pro permite definir entidades desde las cuales podremos invocarlos.

Uso del Diseñador

El diseñador de LLBLGen Pro es una aplicación .NET que no necesita ejecutarse en el contexto de un IDE. El mismo permite la definición de los tipos de entidades definidas anteriormente. Debido a que LLBLGen Pro no define el esquema de la base de datos sino que trabaja sobre él, previamente a la utilización del mapeador, se debe crear el esquema de la base de datos.

Por lo tanto, para crear un nuevo proyecto, se debe seleccionar el esquema de la base de datos a mapear. Una vez creado el proyecto, se podrá acceder a una vista de todas las tablas del esquema de la base de datos aún no mapeadas. Sobre cada tabla a mapear se

debe definir una entidad y configurar las propiedades de los atributos como ser su tipo, si admite nulos, si pertenece a la clave primaria, entre otras.

Para cada entidad , se definirán las relaciones con otras entidades, seleccionando la entidad con la que se relacionará, el tipo de relación (1:1, 1:n, n:1, n:m), y por que atributo se relacionan.

Generación de Código

Una vez definido el mapeo, se deberá generar el código correspondiente a la capa de acceso a datos, y a las entidades de la lógica del negocio. Para ello, LLBLGen Pro permite generar el código en los lenguajes de programación C# y VisualBasic.NET, para distintas versiones del .NET Framework.

Antes de ejecutar el generador de código, se debe seleccionar el grupo de plantillas que generará la herramienta, los cuales podrán ser SelfServicing o Adapter. Esto determinará la forma en que se genera el código y condicionará le uso del código generado para la manipulación de datos. Dependiendo del proyecto y la forma en que se quiera manejar la persistencia, es que se decidirá por uno u otra plantilla.

Si se utiliza *SelfServicing*, toda la lógica que interactúa con el almacenamiento persistente se encontrará en las entidades, vistas y listas tipadas. Esto significa que cada objeto sabe como persistirse. La utilización de esta plantilla tiene como ventaja que no se necesitan de objetos intermediarios para acceder a los datos de la base y como desventaja que se puede persistir directamente desde un componente gráfico, y para evitarlo se deberá agregar lógica extra a las entidades generadas.

Si se utiliza *Adapter*, toda la lógica de la persistencia se realiza en un objeto adaptador llamado *DataAccessAdapter*. Esta forma de generación de código, está basado en el paradigma de considerar la persistencia como un servicio. Si bien esto permite más control y modularización de la persistencia, requiere más líneas de código para guardar y obtener información de la base datos.

Utilización del Código Generado

Una vez generado el código, el mismo se puede agregar a un proyecto .NET existente o bien ser un proyecto nuevo. Cada entidad generada tendrá una región en la que permitirá agregar código de forma que no sea eliminado al regenerar el código desde el diseñador. Dependiendo de la plantilla de generación elegida, SelfServicing o Adapter, cambiará la forma de persistir un objeto y la forma de acceder a los datos. A continuación se muestra como persistir un objeto en modo Self Servicing.

LLBLGen Pro provee un mecanismo de filtros para acceder a los datos de la base de datos sin necesidad de embeber las consultas SQL en el código generado. En el caso que la información a obtener se encuentre en más de una tabla de la base de datos, se pueden relacionar entidades y utilizar filtros más complejos sobre las entidades participantes en la consulta. A continuación se muestra un ejemplo de cómo obtener utilizando SelfServicing,

Listado C.9: Ejemplo de creación de una entidad en LLBLGen Pro.

```
ClienteEntity oCliente1 = new ClienteEntity();
oCliente1.Codigo = 3;
oCliente1.Nombre = "Pablo Soma";
oCliente1.Usuario = "psoma";
oCliente1.Password = "psoma";
oCliente1.Tel = "094949425";
oCliente1.Email = "psoma@mail.com";
oCliente1.Fax = "";
oCliente3.Save();
```

información sobre varias entidades relacionadas, dependiendo del filtro aplicado.

Listado C.10: Ejemplo de filtros en LLBLGen Pro.

```
HotelCollection hoteles = new HotelCollection();
RelationCollection relationsToUse = new RelationCollection();
relationsToUse.Add(HotelEntity.Relations.RecepcionEntityUsingHotelId);

IPredicateExpression selectFilter = new PredicateExpression(
    RecepcionFields.Ipaddress == ipRecepcion);
hoteles.GetMulti(selectFilter, relationsToUse);
```

Apéndice D

Caso de Estudio

Con el objetivo de evaluar los distintos criterios de comparación de productos de persistencia en un sistema orientado a objetos, es que se propone el caso de estudio detallado a continuación.

El caso de estudio considera la realidad presentada en el articulo "SAD del Subsistema de Reservas del Sistema de Gestión Hotelera" [PV03] que describe la arquitectura de un sistema de gestión de una cadena hotelera (Figura D.1). En particular, el caso de estudio se centra en un subsistema del mismo, el Subsistema de Reservas. Por mas detalles sobre la realidad y el diseño del sistema, referirse a [PV03].

A efectos del objetivo antes mencionado, y con la intención de facilitar la implementación y concentrarse en la lógica de negocio, se ignoran una serie de requerimientos planteados en el articulo. En primer lugar no se implementara una interfaz gráfica para acceder al sistema, ya que no es necesaria en el contexto del caso de estudio. Asimismo, se ignoraran una serie de restricciones que se imponen en cuanto a: licenciamiento, formas de pago, registro impositivo, estándares, tecnologías, interacción con sistemas existentes (descriptos en el punto 4 del artículo). Finalmente se descartan los requerimientos no funcionales (punto 5 del artículo).

Siendo el objetivo la evaluación de diferentes productos de persistencia de objetos, se realizo en primera instancia, una implementación del sistema que no considerara el aspecto de persistencia. Dicha implementación será luego extendida y modificada según lo requerido por cada producto en particular. De esta forma será posible identificar las particularidades y requerimientos de cada uno de ellos al momento de ser utilizados. Cabe destacar que ésta implementación básica (sin persistencia) se realizará tanto para la plataforma Java SE como también para el lenguaje C# de la plataforma .NET.

En un principio, las implementaciones básicas en Java y .NET fueron extendidas para utilizar mapeadores objeto-relacional populares o de interés para cada una de estas plataformas.

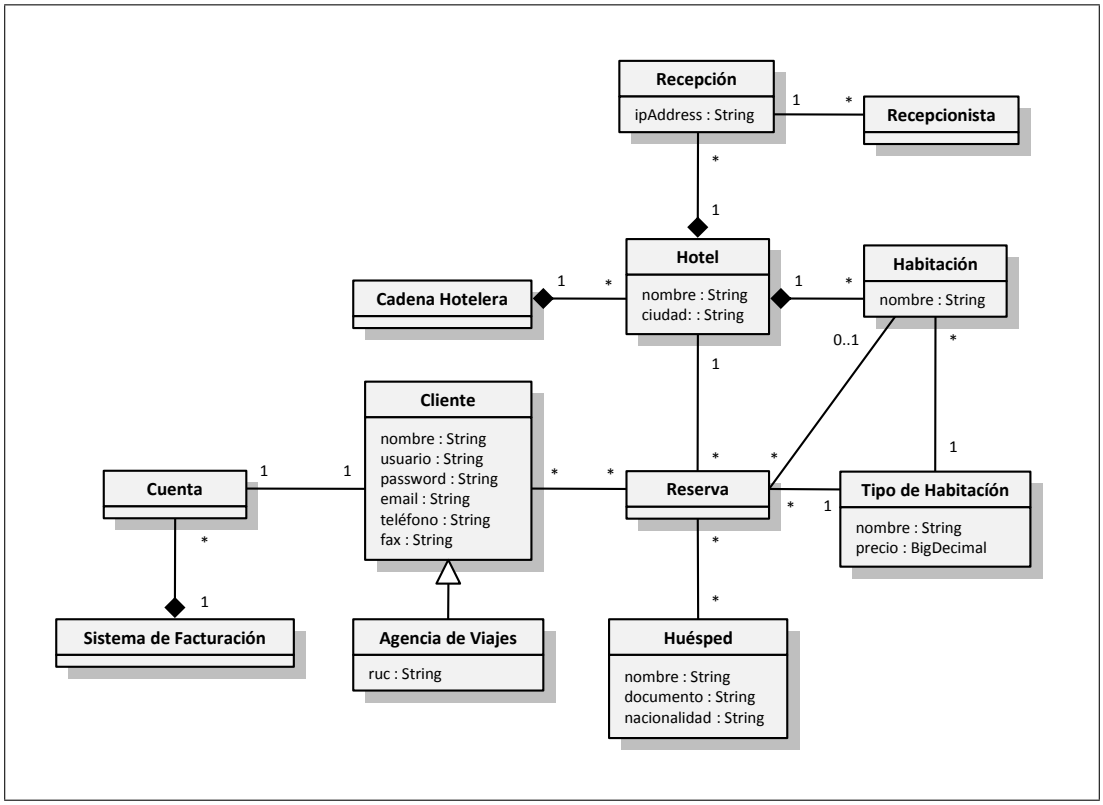


Figura D.1: Modelo conceptual de la cadena hotelera

D.1. Consideraciones de Implementación

A la hora de adaptar la lógica de negocio de la implementación básica para la utilización de un mapeador objeto-relacional, se divisan dos enfoques diferentes en cuanto a la implementación de los procesos de negocio. El primero se basa en trabajar principalmente a nivel de objetos a la hora de implementar los procesos del sistema. Esto implica realizar recorridas en colecciones y utilizaciones de otras estructuras de datos como *hashs* y similares para la resolución de la problemática. Un segundo enfoque considera la creación de consultas, en el lenguaje de consultas provisto por el producto, para la recuperación y actualización de los objetos persistidos a ser utilizados. Estas dos alternativas representan diferentes balances en cuanto a transparencia vs. *performance*. Concretamente, la primera se caracteriza por ofrecer mayores niveles de transparencia, ya que el programador no debe lidiar con un lenguaje de consultas. El programador implementa la lógica sin variar de forma significativa la implementación sin persistencia de la cual se partió. Esto condiciona la *performance* obtenida a las capacidades del producto considerado, para manejar este tipo de enfoque. De forma general se tiene que el desempeño se ve afectado negativamente. La segunda alternativa soluciona este problema mediante la utilización del lenguaje de consultas provisto. Esto permite que el mapeador ejecute de la consulta de forma mas eficiente (valiéndose de heurísticas y optimizaciones pertinentes). Sin embargo esta solución impacta negativamente en el nivel de transparencia del código fuente.

En la practica una solución suele utilizar una combinación de ambos enfoques. Consi-

derando el primero para la mayoría de las operaciones de tipo ABM (Alta Baja y Modificación) y el segundo en resolución de consultas complejas como también en operaciones con altos requerimientos de *performance*. En lo que respecta al caso de estudio se opto por utilizar esta combinación de estrategias.

Apéndice E

Comparación entre Productos

E.1. Criterios de Comparación

Dentro de los criterios de comparación a utilizar entre productos se distinguen tres grandes grupos. Estos grupos están compuestos por:(1) criterios específicos al mecanismo de persistencia considerado, (2) criterios aplicables a productos de cualquier mecanismo y por último (3) criterios ya considerados a nivel de mecanismos que son evaluados nuevamente a la luz del nuevo contexto (a nivel de productos). A continuación se presentan los criterios identificados para cada uno de estos grupos, incluyendo una breve definición de aquellos que no hayan sido introducidos en la comparación de mecanismos.

Criterios a Nivel de Mapeador

En base a los trabajos publicados en [CI05], [Sin06] y [Yal04], se describen un conjunto de criterios de comparación para mapeadores objeto-relacional.

- *Arquitectura de Cache*. En caso de que el mapeador provea *caching* de objetos se indica la arquitectura involucrada. Normalmente se utiliza una arquitectura compuesta por dos niveles de caches, uno compartido por todas las sesiones dentro de la aplicación (*cache* de segundo nivel) y uno particular a la sesión actual (cache de primer nivel).
- *Bulk Data Manipulation.* Se refiere a la aplicación de un patrón de diseño conocido también por el nombre de *Bundled Write*. Este patrón implica que el mapeador envíe conjuntos de sentencias SQL en una sola operación, afectando positivamente la *performance* del sistema.
- *Claves Primarias Compuestas.* Indica si se da soporte para la definición de claves primarias formadas por más de un atributo.
- *Construcción* `GROUP BY`. Refiere a si el lenguaje de consulta que brinda el mapeador soporta la construcción GROUP BY.

- *Construcción Manual de SQL.* Indica si el desarrollador debe lidiar manualmente, de manera obligatoria, con la construcción de sentencias SQL para acceder y manipular los datos.
- *Constructor por Defecto.* Refiere a si una entidad debe contar necesariamente con un constructor por defecto.
- *Consultas Dinámicas.* Explicita si el mapeador en cuestión brinda soporte para la construcción de consultas en tiempo de ejecución.
- *Consultas Nombradas.* Esto implica la posibilidad de que una consulta que es invocada de forma frecuente pueda permanecer en el mapeador, pre-procesada, de tal forma que la ejecución consecutiva de la misma sea más performante.
- *Consultas Polimórficas.* Refiere a si se soporta el concepto de polimorfismo a nivel de las consultas. Esto es, las instancias retornadas por una consulta incluyen todas las instancias de las subclases que satisfagan las condiciones requeridas.
- *Creación Automática de Esquema.* Se refiere a si el producto provee la funcionalidad de generar de forma automática el esquema de base de datos al que se mapean los objetos persistentes.
- *DBMSs Soportados.* Conjunto de las base de datos que son soportadas por el mapeador en cuestión.
- *Estrategias de Fetching.* Establece las estrategias provistas por el mapeador que permiten especificar cómo se obtienen los atributos y asociaciones de determinado objeto. Típicamente, las opciones disponibles son: *eager fetching* y *lazy fetching*. Al utilizar la primera opción sobre un atributo o asociación, el recuperar el objeto al que pertenece, implica su recuperación inmediata. De manera opuesta, utilizar la estrategia *lazy* implica que se retrasa la recuperación de los datos asociados hasta que los mismos son explícitamente requeridos.
- *Estrategias para Herencia.* Conjunto de estrategias de mapeo de herencia (en caso de aplicar) provistas por el mapeador.
- *Funciones Agregadas.* Indica las funciones agregadas soportadas por el lenguaje de consulta provisto por el mapeador (p.ej. `count`, `avg`, etc.).
- *Generación Automática de ID.* Indica si se brinda soporte para la generación automática de identificadores de entidades.
- *Lenguaje de Consulta OO.* Indica si el mapeador en cuestión provee un lenguaje de consulta orientado a objetos.
- *Mapeo de 1 Clase a N Tablas.* Refiere a si una clase puede mapearse a múltiples tablas. Esto es, el estado persistido de un objeto está disperso a lo largo de un conjunto de tablas.
- *Mapeo de N Clases a 1 Tabla.* Indica si múltiples clases pueden ser mapeadas a una única tabla. Esto se refiere a la posibilidad de que varias entidades compartan un mismo destino para uno de sus atributos (una misma columna de determinada tabla).

- *Objetos Asociados con Outer Joins.* Indica si los objetos asociados a otro objeto pueden ser obtenidos mediante el uso de *outer joins*.
- *Ortogonalidad de Tipo.* Refiere a la posibilidad de persistir objetos de clases arbitrarias sin requerir ninguna superclase o interface específica.
- *Persistencia de EJBs.* Refiere al soporte para persistencia de tanto *Session Beans* como de *Entity Beans* con *Bean Managed Persistence* (BMP).
- *Relaciones Soportadas.* Se explicitan los tipos de relaciones soportadas entre objetos. Las mismas pueden ser 1-1, 1-n o m-n.
- *Soporte de Annotations.* Indica si es posible especificar la *metadata* de mapeo de una clase utilizando únicamente *annotations*.
- *Soporte para Cascading.* Refiere a si el mapeador permite especificar la propagación de operaciones de persistencia a lo largo de sus asociaciones.
- *Soporte para Detached Objects.* Se entiende por *detached objects* aquellas entidades obtenidas en el contexto de una sesión y utilizadas posteriormente al cierre de esta última. Este criterio refiere a la posibilidad de incluir en una nueva sesión estos tipos de objetos (considerando posibles modificaciones realizadas) y que vuelvan a ser tratados como una entidad persistente normal.
- *Soporte para Optimistic Locking.* Indica si el mapeador en cuestión soporta *locking* optimista. Esta estrategia detecta el caso en que una transacción obtenga ciertos datos, los modifique en memoria, y antes de actualizar la base de datos, otra transacción ya lo haya hecho. En este caso se produce una excepción. La responsabilidad de tratar con la misma recae en el desarrollador.
- *Soporte para Paginación.* Refiere a si el mapeador utilizado permite recuperar un rango de *entidades* dentro del resultado total de una consulta. De esta forma es posible desplegar la información de la misma de manera parcial y a demanda.
- *Tablas Extra en la Base de Datos.* Indica si se necesitan tablas extras en la base de datos para mantener *locks*, *metadata*, etc. en *algún* escenario.
- *Tipos Personalizados.* Hace referencia a si el desarrollador puede definir tipos personalizados (no entidades) de forma tal que un atributo de este tipo pueda ser mapeado a una o más columnas de la tabla asociada a la entidad contenedora.
- *Uniquing.* Refiere a si, en el contexto de una misma sesión, dos objetos retornados por dos consultas cualesquiera que representan el mismo objeto persistente tienen la misma identidad.
- *Persistencia con Campos Privados.* Indica si se soporta la persistencia de propiedades a través de campos declarados como privados a la clase.
- *Persistencia con `get/set`.* Indica si se soporta la persistencia de propiedades a través de métodos get/set.
- *Proceso/Generación de Código.* Refiere a si el uso del mapeador requiere en algún momento de generación y/o el procesamiento de código fuente.

Criterios a Nivel de Producto

Esta sección presenta una colección de criterios propios de un producto y que en son independientes del mecanismo considerado.

- *Última Versión.* Se refiere a la última versión estable del producto. Este criterio es de interés debido a que la versión es un indicador de la madurez y estabilidad del mismo.
- *Fecha de Lanzamiento.* Corresponde a la fecha de lanzamiento de la última versión estable del producto. La misma tiende a reflejar la actividad en el desarrollo.
- *Licencia.* La licencia bajo la cual se distribuye el producto.
- *Plataformas.* Lista de las plataformas bajo las cuales el producto puede ser utilizado. Por plataforma se entiende el entorno necesario para la utilización del producto.
- *Documentación Disponible.* Listado de la documentación más relevante disponible en relación con el producto. Es claro que la disponibilidad de documentación es un factor muy importante para la comprensión y aprendizaje del mismo.
- *Consumo de Recursos.* Mide el consumo de recurso del producto.
- *Curva de Aprendizaje.* Esfuerzo requerido para la comprensión y aprendizaje de la forma de uso del producto. Esta medida es importante en lo que refiere a la capacitación de los desarrolladores en el caso de no tener experiencia con el producto.
- *Herramientas Disponibles.* Lista de las herramientas de desarrollo más relevantes que asistan en la utilización del producto. La existencia y la calidad de las mismas pueden afectar de manera directa la productividad.

Criterios a Nivel de Mecanismo

En este grupo no se incluyen todos los criterios considerados a nivel de mecanismos, sino que sólo se consideran aquellos que pueden aportar a la diferenciación de los productos. En particular los criterios que son de interés son: Performance, Transparencia, Mecanismo de Consulta, Soporte para Evolución de Esquema, Soporte Transaccional y Popularidad.

E.2. Evaluación de Criterios

E.2.1. Cuadro Comparativo de Mapeadores

Debido a la naturaleza simple y directa del resultado de la evaluación de muchos de los criterios definidos anteriormente, se presenta a continuación un conjunto de los mismos en forma tabular. Concretamente, el Cuadro E.1 permite contrastar las diferencias de los mapeadores comparados. Se consideran en la misma aquellos criterios que ameritan una representación simple en cuanto a los resultados de su evaluación.

Cuadro E.1: Evaluación de Criterios para Productos

Criterio	**Hibernate**	**JPA/TopLink**	**LLBLGen Pro**
	Nivel de Productos		
Ultima Versión	v3.2.1	v2.0b41	v2.0
Fecha de Lanzamiento	16/11/06	11/05/06	02/07/06
Licencia	LGPL	CDDL	Comercial
Plataformas	Múltiples (Java)	Múltiples (Java)	Múltiples (.NET)
	Nivel de Mapeadores		
Arquitectura de *Cache*	2 Niveles	2 Niveles	2 Niveles
Bulk Data Manipulation	✓	✓	✓
Consultas Nombradas	✓	✓	✓
Claves Primarias Compuestas	✓	✓	✓
Construcción Manual de SQL	✗	✗	✗
Construcción `GROUP BY`	✓	✓	✗
Constructor por Defecto	✓	✓	✗
Consultas Dinámicas	✓	✓	✓
Consultas Polimórficas	✓	✓	✓
Creación Automática de Esquema	✓	✓	✗
DBMSs Soportados	JDBC-compliant	JDBC-compliant	SQL Server, Oracle, PostgreSQL, Firebird, DB2 UDB, MySQL, MS Access
Estrategias de *Fetching*	{eager, lazy}	{eager, lazy[1]}	{eager, lazy}
Estrategias para Herencia	{table-per-class, table-per-hierarchy, table-per-subclass}	{table-per-class, table-per-hierarchy, table-per-subclass}	{table-per-class, table-per-hierarchy, table-per-subclass}
Funciones Agregadas	{avg, count, max, min, sum}	{avg, count, max, min, sum}	{avg, count, max, min, sum}
Generación Automática de ID	✓	✓	✓
Lenguaje de Consulta OO	✓(HQL)	✓(JPQL)	✗
Mapeo de 1 Clase a N Tablas	✓	✓	✗
Mapeo de N Clases a 1 Tabla	✓[2]	✓[2]	✓
Objetos Asociados con *Outer Joins*	✓	✓	✓
Ortogonalidad de Tipo	✓	✓	✓
Persistencia de EJBs	✓	✓	✗
Persistencia con Campos Privados	✓	✓	✓
Persistencia con `get/set`	✓	✓	✗
Proceso/Generación de Código	✗	✗	✓
Relaciones Soportadas	{1-1, 1-n, m-n}	{1-1, 1-n, m-n}	{1-1, 1-n, m-n}
Soporte de *Annotations*	✓	✓	✗
Soporte para *Cascading*	✓	✓	✓
Soporte para *Detached Objects*	✓	✓	✓
Soporte para *Optimistic Locking*	✓	✓	✓
Soporte para Paginación	✓	✓	✓
Tablas Extra en la Base de Datos	✓	✓	✓
Tipos Personalizados	✓	✓	✓
Uniquing	✓	✓	✓

[1] Depende de si el proveedor de persistencia utilizado lo soporta.
[2] Solo una clase puede modificar los datos, el resto se restringe a lectura.

E.2.2. Herramientas

- *Hibernate.* Se destacan las denominadas "Hibernate Tools", un conjunto de herramientas para Hibernate 3 implementadas como una serie de *plugins* para Eclipse (`http://tools.hibernate.org`). Dentro de sus características fundamentales se tiene un editor para mapeos y un módulo de ingeniería reversa capaz de generar, entre otras cosas, clases de dominio, archivos de mapeo y documentación. En lo que respecta al soporte para Hibernate en NetBeans se encuentra disponible el *plugin* "NBXDoclet" (`http://nbxdoclet.sourceforge.net`). De acuerdo con sus características se destaca la posibilidad de generar archivos de mapeo, crear el esquema de base de datos a partir de estos y un asistente para producir consultas HQL. Cabe destacar que ambas herramientas se distribuyen de forma gratuita.

- *JPA.* En lo que a Eclipse se refiere existe un proyecto denominado "Dali JPA Tools" (`http://www.eclipse.org/dali`) que apunta a ofrecer una experiencia integrada para el desarrollo de aplicaciones JPA. Este proyecto lleva a cabo el desarrollo de un *plugin* para el IDE mencionado. Cabe notar que actualmente, el mismo se encuentra en una etapa temprana de desarrollo. Por otro lado, NetBeans en su versión 5.5, ofrece soporte para el desarrollo de aplicaciones JPA mediante el modulo denominado Enterprise Pack 5.5. Téngase en cuenta que trata de una herramienta más completa y madura que el *plugin* disponible para Eclipse. Por más información acerca de las diferencias entre ambas opciones referirse a `http://blogs.sun.com/klingo/entry/jpa_netbeans_5_5_vs`.

- *LLBLGen Pro.* El diseñador que provee el mapeador es excluyente para la realización del mapeo y posterior generación de código. Luego, es de utilidad contar con un IDE para continuar con el desarrollo de la aplicación, utilizando el código generado. LLBLGen Pro permite crear un proyecto Visual Studio .NET o bien integrar lo generado a uno existente.

E.2.3. Documentación

- *Hibernate.* Este producto consta de una amplia documentación que se extiende desde una gran variedad tutoriales en línea a completos libros que ofrecen todo lo necesario para capacitarse de manera adecuada. De acuerdo con esta última categoría se destacan "Hibernate in Action" [BK04] e "Hibernate Quickly" [PH05], ambos de la misma editorial. El primero puede considerarse el libro por excelencia para Hibernate donde se encuentran todos los detalles necesarios a fin de dominar la herramienta. El segundo libro expone las características más relevantes del mapeador y permite de manera rápida comenzar a utilizarlo. Cabe notar que en el sitio oficial de Hibernate pone a disposición, de forma gratuita, una amplia gama de documentación que abarca todos sus aspectos de interés.

- *JPA.* De la documentación disponible para JPA, claramente resulta de interés el documento de especificación de dicho estándar, parte de la especificación de EJB 3.0 [DK06]. Asimismo, se destaca el libro "Pro EJB 3: Java Persistence API" [KS06]. Debido a que el producto considerado en este caso es una combinación de estándar (JPA) y producto (TopLink Essentials), también son relevantes los documentos y

tutoriales referentes al uso particular del proveedor de persistencia de TopLink para JPA. Es así que se puede mencionar un documento en línea que especifica la forma de configurar las extensiones a JPA definidas por TopLink Essentials [Ext].

- *LLBLGen Pro.* Se destacan dos documentos que son indispensables para el correcto uso del mapeador. La documentación de usuario y el manual de referencia, ambos incluidos en el producto. La documentación de usuario asiste tanto en la utilización del diseñador como también en el posterior uso de código generado. El manual de referencia ofrece un completo API sobre las clases provistas por el mapeador y los manejadores de las bases de datos soportadas. Como documentación complementaria, en caso de utilizar C# como lenguaje de programación, es de utilidad el libro "Rapid C# Windows Development: Visual Studio 2005, SQL Server 2005, and LLBLGen Pro" [Cha06], aunque este libro no es para la última versión del mapeador. Se recomiendan también completos ejemplos provistos en el sitio web del mapeador implementados tanto en C# como en Visual Basic .NET, que muestran la aplicación de LLBLGen Pro para distintos casos.

E.2.4. Performance

La evaluación de la *performance* de un mapeador objeto-relacional es un tema de controversia actual. Uno de los problemas que se encuentra es la no existencia de un *benchmark* estándar aceptado por los diferentes proveedores. A pesar de existir intentos de desarrollar un *benchmark* de estas características que permita comparar, de forma objetiva, el desempeño de los diferentes mapeadores, los profundos intereses comerciales detrás de dicho desarrollo han impedido su surgimiento.

Benchmarks Existentes

Uno de los *benchmark* más referenciados en el área académica recibe el nombre de *oo7*. El mismo data del año 1993 y es presentado en [CDN93]. Surge como una serie de tests que realizaban diferentes tipos de operaciones de persistencia (agregar, borrar, actualizar y consultar) orientados a la evaluación de la *performance* de una base de datos orientada a objetos. El modelo de dominio en el que se basa el *benchmark*, a pesar de no modelar ninguna aplicación específica, está fuertemente influenciado por lo que podría ser un modelo de dominio de una aplicación tipo CAD (Computer Aided Design), CAM (Computer Aided Manufacturing) o CASE (Computer Aided Software Engineering). Estos tipos de aplicaciones suelen presentar modelos de dominio con herencias de clases muy profundas y múltiples relaciones entre clases. Mas allá de que éstas ultimas acotaciones parecerían indicar que el *benchmark* no es el más indicado para la evaluación de un mapeador objeto-relacional, el mismo se ha convertido en uno de los más populares (sobre todo a nivel académico). Esto se debe a que la mayoría de los tests están ideados para atacar problemas presentes tanto en la implementación de una base de datos orientada a objetos como en la implementación de un mapeador objeto relacional.

Otro *benchmark* destacable tiene sus orígenes en la empresa db4objects y lleva el nombre de PolePosition [Pol]. Se trata de un proyecto de código abierto orientado a la

evaluación de la *performance* de motores de bases de datos y mapeadores objeto-relacional para la plataforma Java. La arquitectura extensible del mismo permite la utilización de un controlador particular para cada producto de persistencia a evaluar. En cuanto al conjunto de tests incluidos en el *benchmark* los mismos siguen una analogía a la realidad de carreras de formula uno. De acuerdo con ello, cada test recibe el nombre de *circuito*.

Problemas con los Benchmarks

Algunos de los problemas más importantes que obstaculizan el surgimiento de un *benchmark* estándar se mencionan a continuación. El primero de ellos es su origen, esto es, muchos de los *benchmarks* propuestos actualmente surgen en empresas particulares que a su vez ofrecen productos de mapeo objeto-relacional. En términos de lo anterior la objetividad de los mismos resulta cuestionable. Por último, otro problema que los *benchmark* propuestos deben enfrentar es el *tunning* realizado por los diferentes proveedores en sus productos. Esto no se lleva a cabo a fin de obtener un mejor producto sino para simplemente obtener mejores resultados en determinado *benchmark* (lo que no implican mejoras reales en la *performance*). Es por estas razones que Gavin King, fundador de Hiberante (producto líder del mercado), opina que los *benchmarks* de *performance* para mapeadores objeto-relacional son más utilizados para engañar que para comparar.

Propuesta

Es en este contexto que se plantea elaborar un *benchmark* propio para la evaluación de la *performance* de los productos considerados por este documento. Esto se realiza basándose en el modelo de negocio del caso de estudio (considerando al mismo como un sistema de información típico) y en los tests propuestos por los *benchmarks* oo7 y PolePostion.

Cabe destacar que el *benchmark* es una simplificación de ambos, no pretende por ningún motivo ser un estándar para la evaluación de mapeadores en sistemas de información y es de valor simplemente para la evaluación rudimentaria de la *performance* en el caso particular del caso de estudio.

Algunos de los conceptos presentados en oo7 y que son aplicados en el *benchmark* propuesto refieren al método de ejecución de los tests y reciben el nombre de *ejecuciones en frío* y *ejecuciones en caliente*. Una ejecución en frío de un test implica que los diferentes caches que forman parte de la arquitectura de caches presentes en el mapeador objeto-relacional, están vacíos (por lo menos en lo que respecta a las entidades involucradas en el test). Por otro lado, una ejecución en caliente implica la ejecución consecutiva del mismo test, la primer ejecución es (como se definió anteriormente) llevada a cabo en frío mientras que las siguientes ejecuciones ya cuentan con los datos almacenados en el diferentes caches. De esta forma se realiza el promedio de la ejecuciones posteriores a la primera y se considera este tiempo como el tiempo de ejecución en caliente. La ejecución en caliente de tests no sólo afecta la *performance* de aquellas operaciones que hacen uso del cache de objetos, ya que al ejecutar por segunda vez un mismo test se eliminan demoras relacionadas con el cargado en memoria y preparación de estructuras de datos necesarias

que son mantenidas para las ejecuciones siguientes. Otro concepto aplicado que está presente en oo7 y PolePosition es la parametrización de los datos de prueba en cuanto a la cantidad de entidades existentes, en el medio persistente, al momento de la ejecución de los tests. Es así que al igual que oo7 se consideran conjuntos de datos de prueba pequeños, medianos y grandes. Por último, al igual que lo que proponen ambos *benchmarks* antes mencionados, lo propuesto no ofrece como resultado un único número que indique el nivel de *performance* del producto, sino que un conjunto de estos que se refieren a los tiempos de ejecución de los diferentes test realizados.

A continuación se presenta un resumen de los diferentes grupos de tests que se consideran en el *benchmark* propuesto:

- *Búsquedas exactas (FIND).* Este conjunto de tests se centra en la evaluación de una operación básica de un mapeador que permite la recuperación de un objeto en particular basándose en el identificador persistente del mismo o en el valor de alguno de sus atributos que lo identifique. En dicho conjunto se incluyen cuatro test que varían en cuanto al tipo del identificador del objeto (numérico o string) y al hecho de existir o no un índice en la base de datos para la columna que guarda dicho identificador.
- *Consultas (QUERIES).* Se consideran consultas de diferente índole, incluyendo :consultas que requieren el *join* de tablas a nivel de la base de datos para su implementacion y consultas que devuelven el 1 %, 10 % y el 100 % de la totalidad de las instancias de una clase particular.
- *Inserciones (INSERT).* Considera el tiempo requerido para la persistencia de nuevas entidades. Este conjunto de tests toma en cuenta la inserción de entidades simples, sin relaciones y con atributos de tipos básicos, así como también la inserción de entidades complejas que participan en múltiples relaciones.
- *Actualizaciones (UPDATE).* Se mide el tiempo de ejecución de tests que realizan distintos tipos de actualizaciones, tanto sobre entidades simples, como también sobre entidades complejas. Las actualizaciones incluyen modificaciones sobre valores de atributos de tipos básicos, la eliminación de enlaces existentes entre entidades y el agregado de nuevos enlaces en diferentes relaciones.
- *Eliminaciones (DELETE).* En este conjunto de tests se evalúa la *performance* de los métodos de eliminación de entidades, tanto simples como complejas.

Resultados del Benchmark Propuesto

La configuración de *hardware* y *software* en la cual se ejecutaron los *tests* del *benchmark* propuesto se muestra en el Cuadro E.2.

En cuanto a la base de datos relacional, destino de los datos mapeados, se propone utilizar la misma para todos los productos de mapeo. En concreto se hace uso de PostgreSQL pretendiendo con ello eliminar diferencias de *performance* cuya causa sea el DBMS (recordemos que la intención es evaluar mapeadores y no base de datos). Mas allá de esto, en el

Cuadro E.2: Ambiente de Ejecución para el Benchmark Propuesto

Item	**Detalles**
	Hardware
Procesador	Intel Pentium™ 4 3.0GHz HT
Cache L2	1.0MB
Bus de Datos	800MHz
Memoria Principal	512MB DDR2 400MHz
Disco Duro	80GB 5400RPM
	Software
Sistema Operativo	Windows XP Professional
Java	Java SE Runtime Enviroment 6 Update 1
.NET	Microsoft .NET Framework 2.0
PostgreSQL	PostgreSQL Database Server 8.2
SQL Server	Microsoft SQL Server Express Edition 9.0.1399

caso de LLBLGen Pro se hizo una excepción debido a que la herramienta solo permite la utilización de versiones anteriores de PostgreSQL y que además implica la utilización de un controlador desarrollado por la comunidad (Npqsql) que compromete los resultados del *benchmark*. Es así que para la evaluación de LLBLGen Pro se considerará como destino de los datos tanto Microsoft SQL Server Express Edition (abreviado en las gráficas como LLBLGen SS) como PostgresSQL (abreviado LLBLGen PS).

Los resultados del *benchmark* propuesto se presentan a razón de una gráfica por test, incluyendo en cada una los valores para los diferentes productos. Se indica el tamaño de la base de datos mediante las abreviaciones PEQ indicando que la cantidad de datos existentes es relativamente pequeña y MED para indicar un tamaño mediano de datos. Asimismo se diferencia el método de ejecución de los tests con HOT para ejecuciones en caliente y COLD para ejecuciones en frío. Téngase en cuenta que los costos expresados en las gráficas en escala logarítmica refieren al tiempo de ejecución de los tests en milisegundos. Los tests son identificados por una clave compuesta por: el identificador del grupo (definido en E.2.4) y el nombre del test dentro del grupo. Como dato complementario se agrega a cada gráfica los valores correspondientes a la implementación de la persistencia basada en JDBC.

FIND

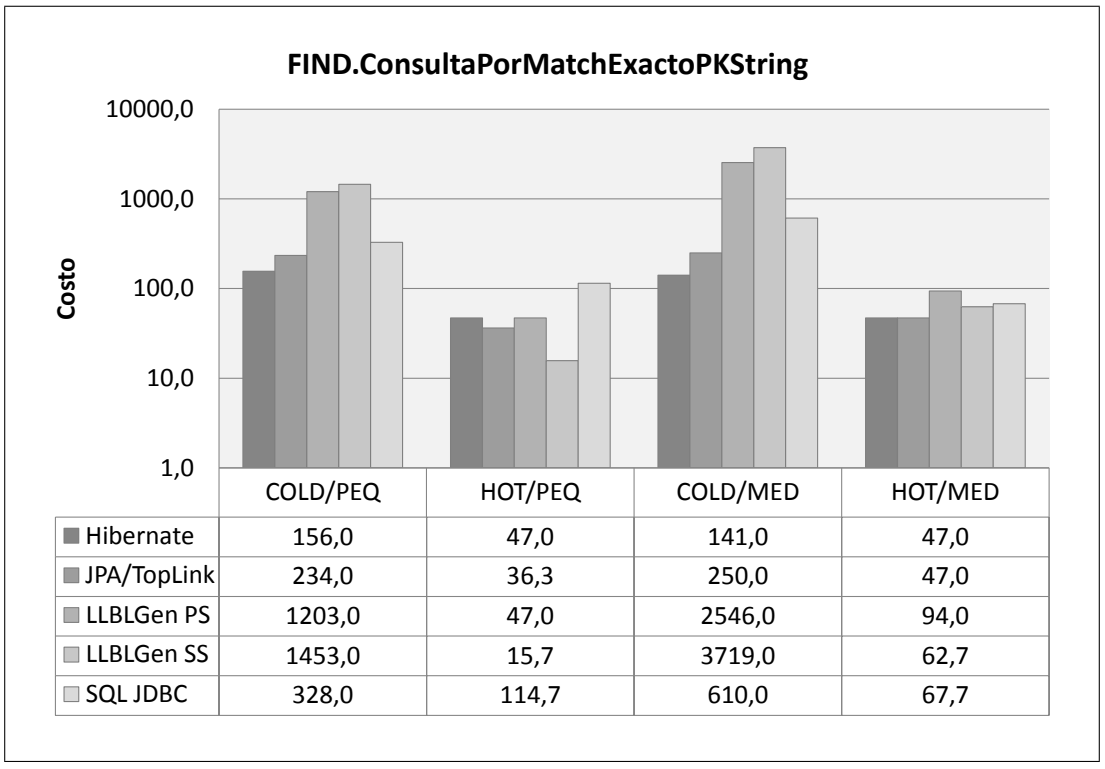
Este conjunto de tests pretende evaluar la performance de una operación básica de un mapeador que consiste en recuperar un objeto desde la base de datos. Esta compuesto por cuatro tests los cuales se resumen a continuación:

- *FIND.ConsultaPorMatchExactoPKString*. Recupera 40 objetos de tipo `Hotel` (seleccionados de forma aleatoria) valiendose de las claves persistentes de los mismos, las cuales son de tipo `String`. Véase la Figura E.1.

- *FIND.ConsultaPorMatchExactoPKInt*. Recupera 40 objetos de tipo `Reserva` (seleccionados de forma aleatoria) valiendose de las claves persistentes de los mismos, las cuales son de tipo `int`. Véase la Figura E.2.

- *FIND.ConsultaPorMatchExactoNoIndexadoString.* Recupera 40 objetos de tipo `Huesped` (seleccionados de forma aleatoria) valiendose del valor del atributo `documento`. Este, a pesar de no ser su clave persistente, identifica al `Huesped`. El no ser la clave primaria resulta en la mayoría de los mapeadores en que no exista un indice para la columna en la cual se mapea el atributo antes mencionado. Nótese que el atributo es de tipo `String`. Véase la Figura E.3.

- *FIND.ConsultaPorMatchExactoNoIndexadoInt.* Recupera 40 objetos de tipo `Reserva` (seleccionados de forma aleatoria) valiéndose del valor del atributo `clienteId`. Al igual que en el test anterior, el atributo seleccionado para la recuperación de los objetos no es su clave persistente. Debido a esto probablemente no exista un indice a nivel de la base de datos para la columna correspondiente. Nótese que el atributo `clienteId` es de tipo `int`. Véase la Figura E.4.



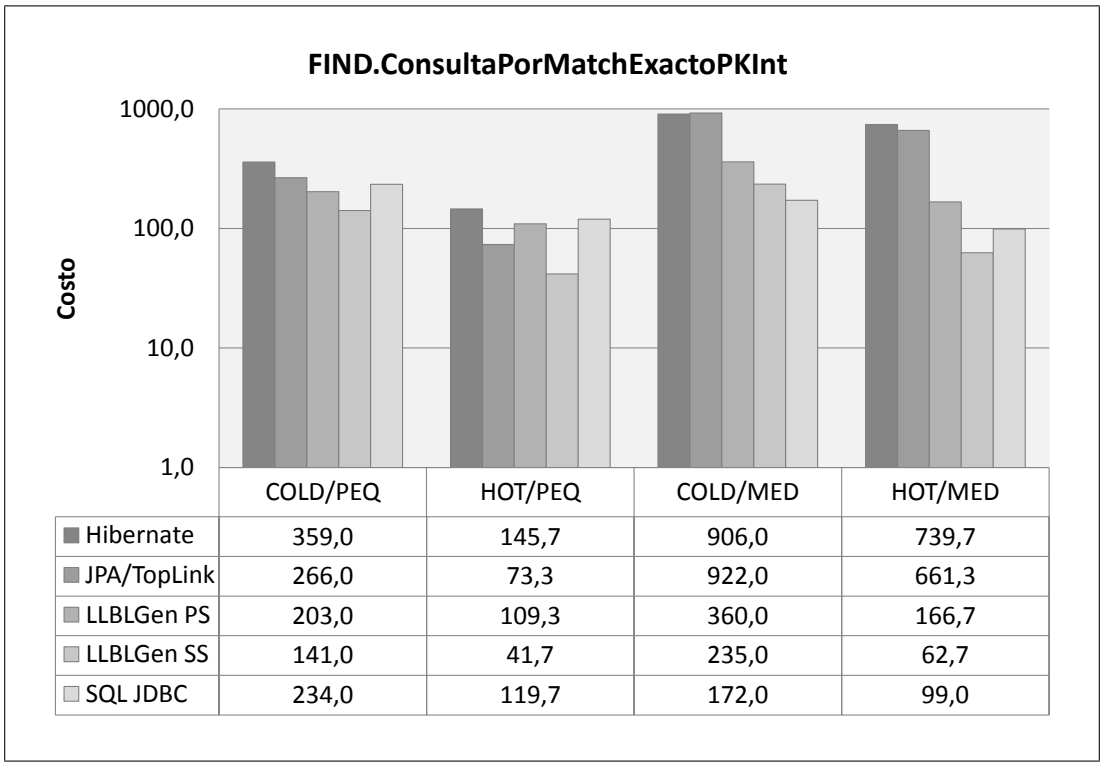
	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	156,0	47,0	141,0	47,0
JPA/TopLink	234,0	36,3	250,0	47,0
LLBLGen PS	1203,0	47,0	2546,0	94,0
LLBLGen SS	1453,0	15,7	3719,0	62,7
SQL JDBC	328,0	114,7	610,0	67,7

Figura E.1: Test de Performance tipo FIND

QUERIES

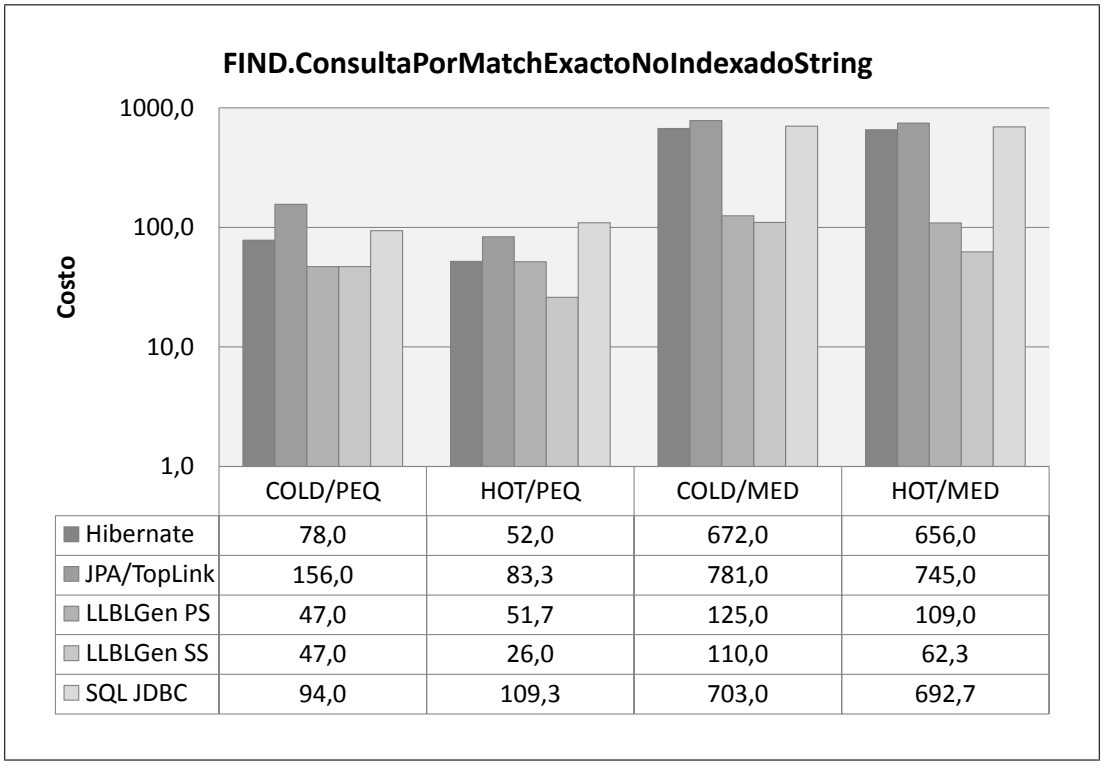
Este conjunto de tests cuenta con cuatro de estos. Los mismos exploran las capacidades de recuperación de objetos a traves del mecanismo de consultas provisto por el mapeador. Estos se resumen a continuacion:

- *QUERIES.UnoPorCientoEnFechas.* Obtiene el uno por ciento de la totalidad de entidades de tipo `Reserva` basándose en el valor del campo `checkIn` de tipo fecha. Véase la Figura E.5.



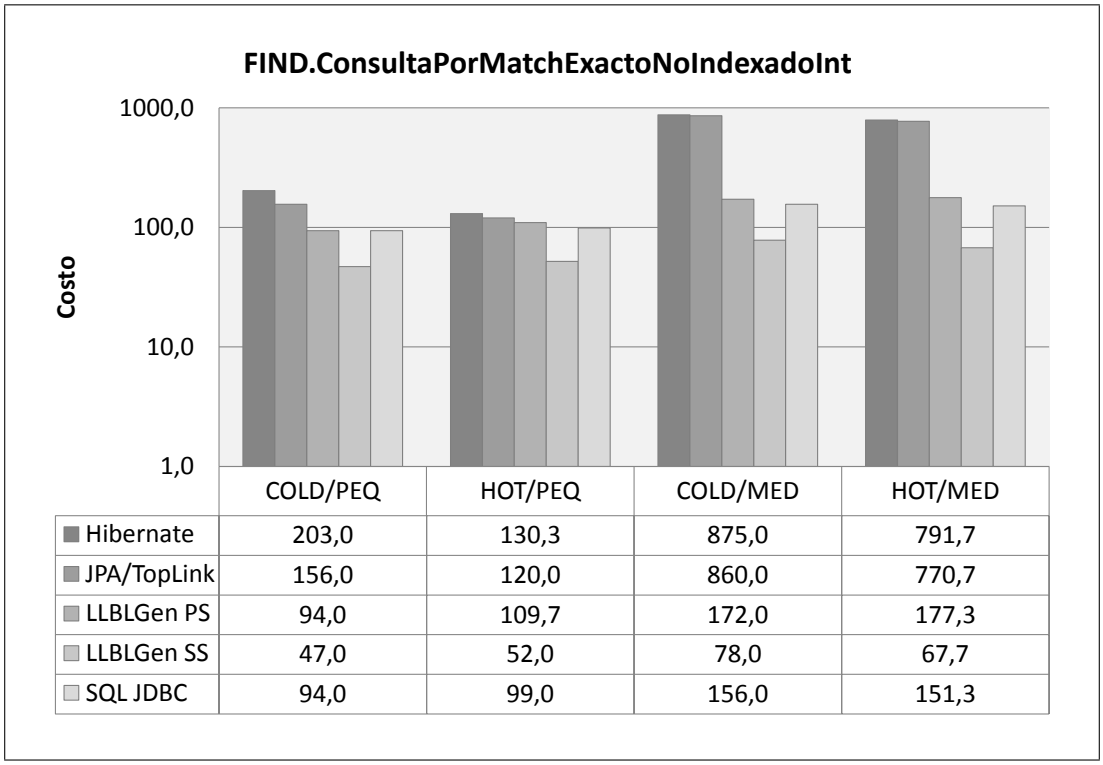
	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	359,0	145,7	906,0	739,7
JPA/TopLink	266,0	73,3	922,0	661,3
LLBLGen PS	203,0	109,3	360,0	166,7
LLBLGen SS	141,0	41,7	235,0	62,7
SQL JDBC	234,0	119,7	172,0	99,0

Figura E.2: Test de Performance tipo FIND



	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	78,0	52,0	672,0	656,0
JPA/TopLink	156,0	83,3	781,0	745,0
LLBLGen PS	47,0	51,7	125,0	109,0
LLBLGen SS	47,0	26,0	110,0	62,3
SQL JDBC	94,0	109,3	703,0	692,7

Figura E.3: Test de Performance tipo FIND



	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	203,0	130,3	875,0	791,7
JPA/TopLink	156,0	120,0	860,0	770,7
LLBLGen PS	94,0	109,7	172,0	177,3
LLBLGen SS	47,0	52,0	78,0	67,7
SQL JDBC	94,0	99,0	156,0	151,3

Figura E.4: Test de Performance tipo FIND

- *QUERIES.DiezPorCientoEnFechas*. Obtiene el diez por ciento de la totalidad de las entidades de tipo `Reserva` basándose en el valor del campo `checkIn` de tipo fecha. Véase la Figura E.6.

- *QUERIES.TodasLasFechas*. Obtiene la totalidad de las entidades de tipo `Reserva`. Véase la Figura E.7.

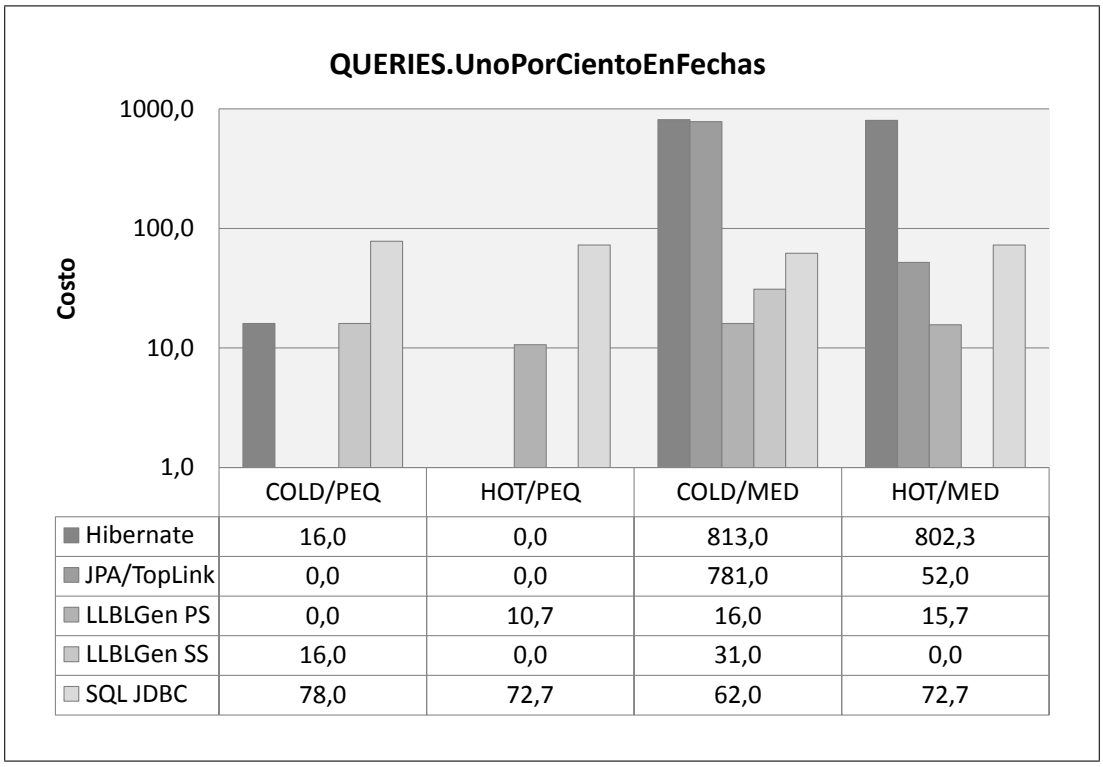
- *QUERIES.Join*. Realiza el join de las entidades de tipo `Reserva` y `Cliente` basandose en el valor de los atributos `clientId` y `codigo` respectivamente. Véase la Figura E.8.

INSERT

Se trata de un conjunto de tests que prueban la inserción de nuevos objetos y su consecuente mapeo a la base de datos relacional. Estos se resumen a continuación:

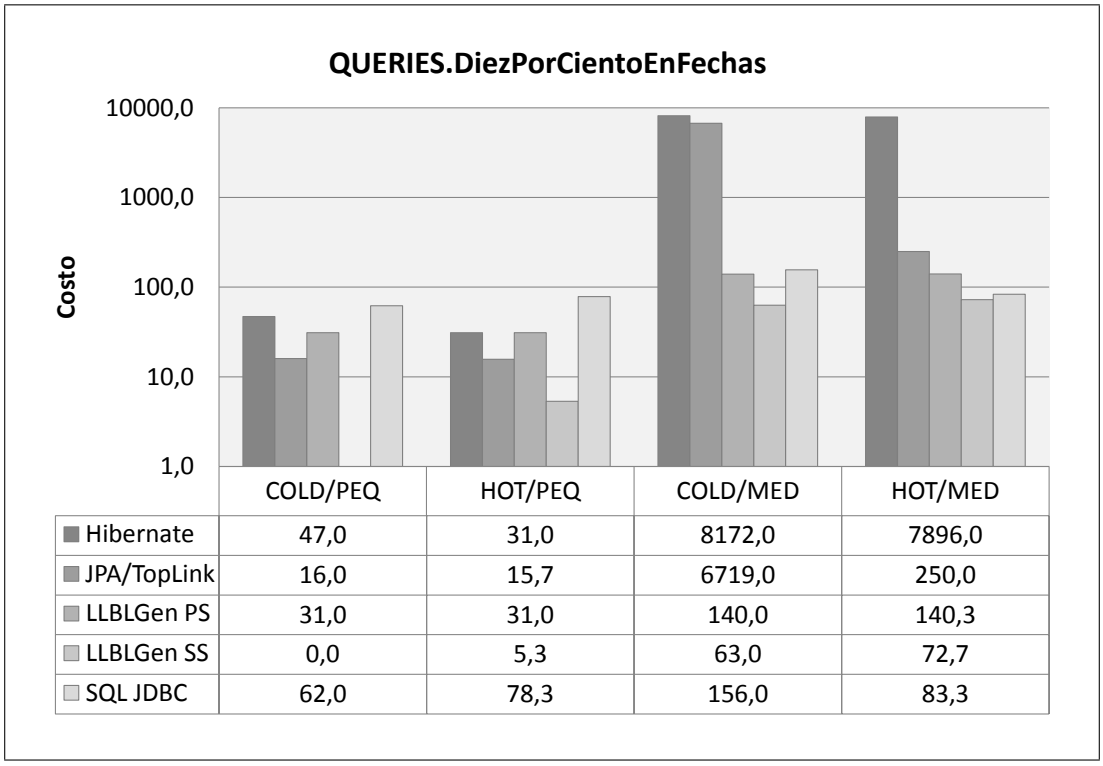
- *INSERT.InsertSimple*. Realiza la inserción de 20 nuevas reservas. Estas son consideradas como inserciones simples debido a que no involucran la creación de *links* con otros objetos persistentes existentes o nuevos. Se reduce al mapeo de nuevas entidades de tipo `Reserva` a la tabla correspondiente. Véase la Figura E.9.

- *INSERT.InsertComplejo*. Realiza la inserción de 20 nuevos hoteles, lo que implica la creación de las habitaciones y recepciones de cada uno de ellos. Véase la Figura E.10.



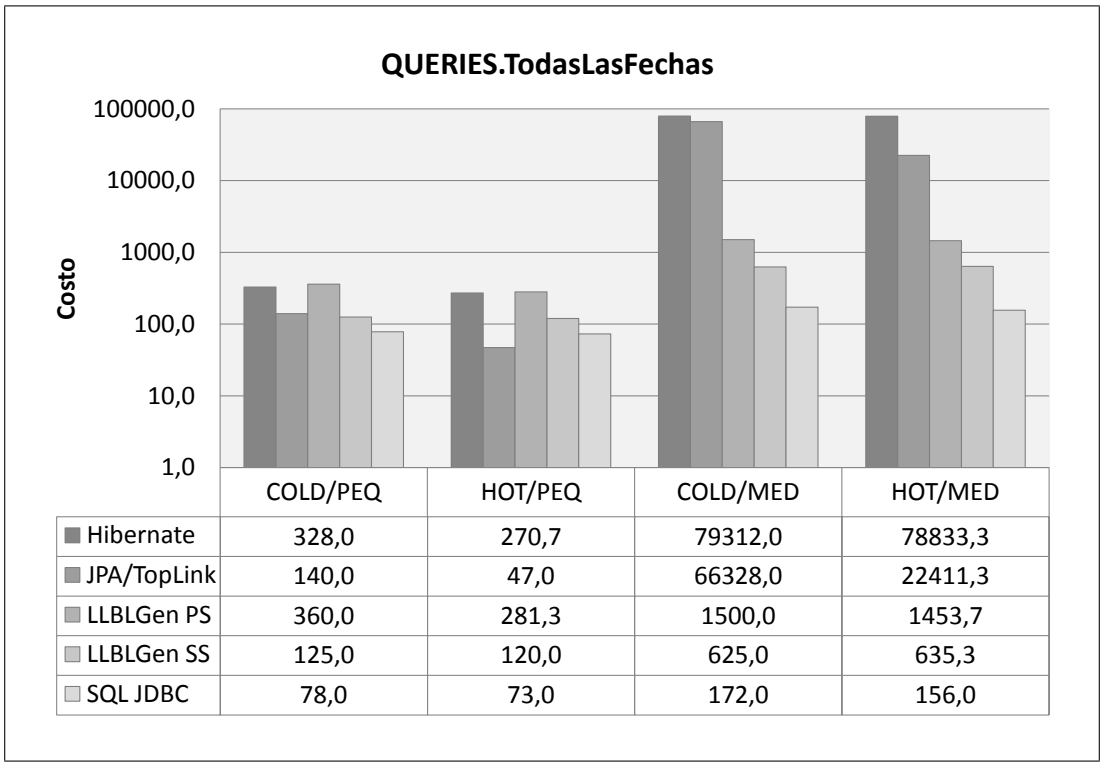
	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	16,0	0,0	813,0	802,3
JPA/TopLink	0,0	0,0	781,0	52,0
LLBLGen PS	0,0	10,7	16,0	15,7
LLBLGen SS	16,0	0,0	31,0	0,0
SQL JDBC	78,0	72,7	62,0	72,7

Figura E.5: Test de Performance tipo QUERIES



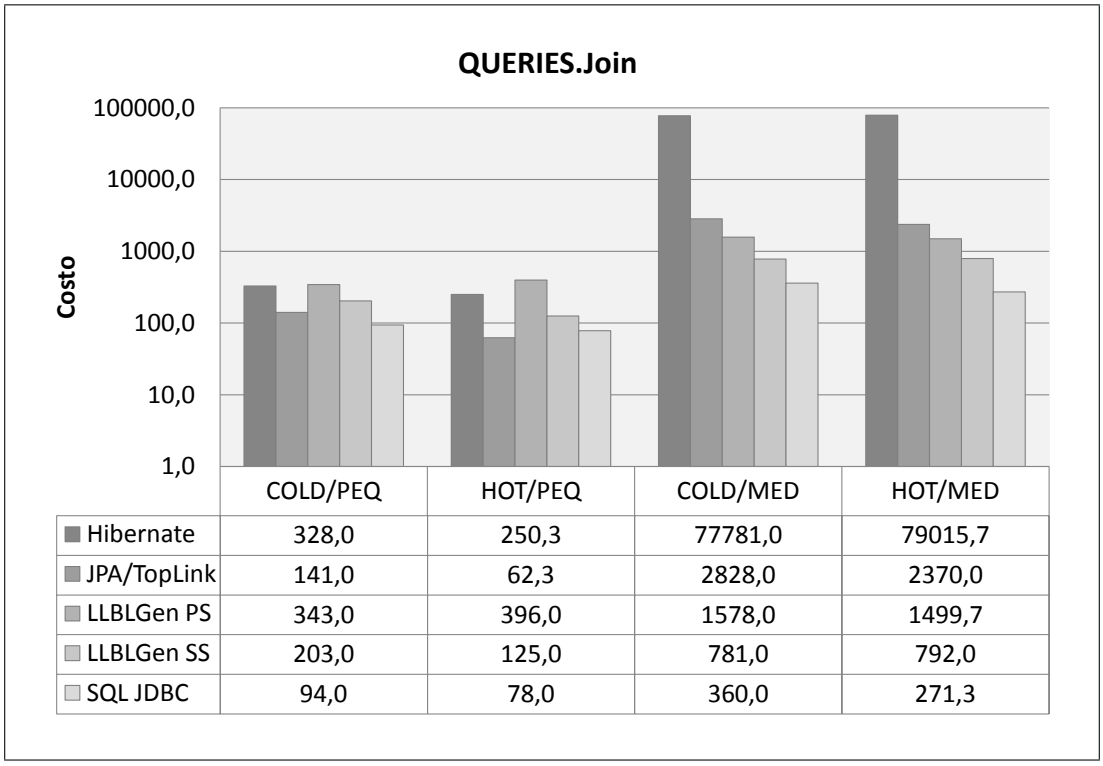
	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	47,0	31,0	8172,0	7896,0
JPA/TopLink	16,0	15,7	6719,0	250,0
LLBLGen PS	31,0	31,0	140,0	140,3
LLBLGen SS	0,0	5,3	63,0	72,7
SQL JDBC	62,0	78,3	156,0	83,3

Figura E.6: Test de Performance tipo QUERIES



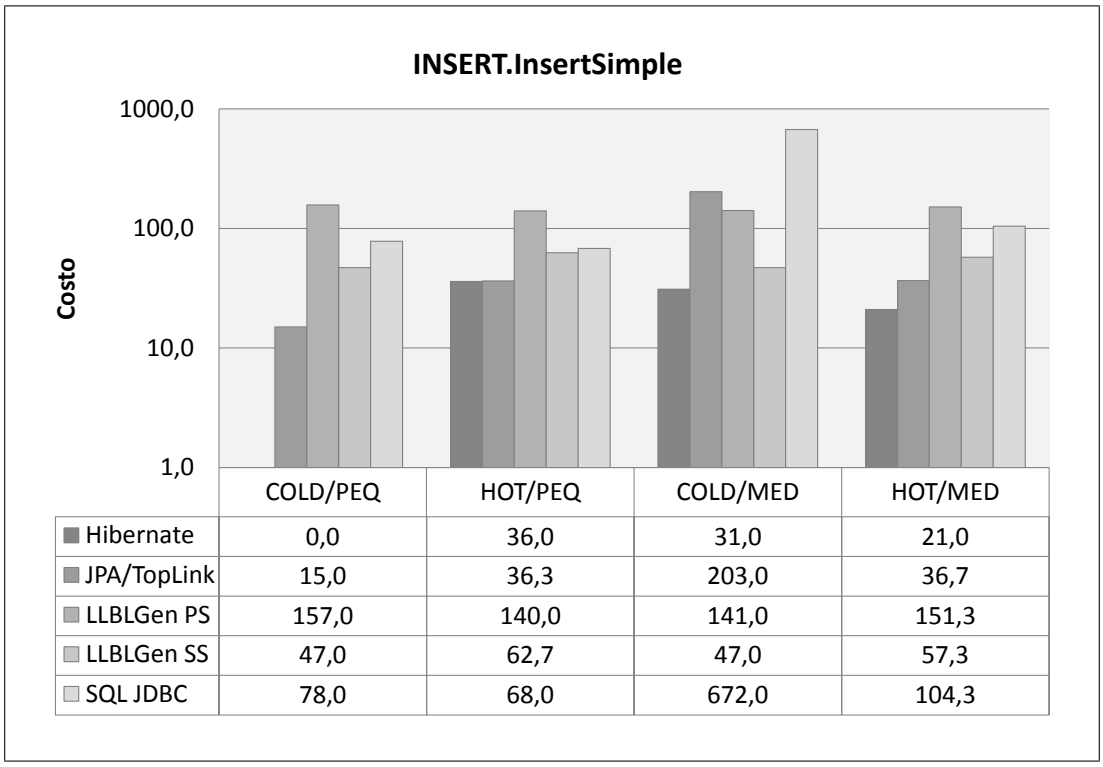
	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	328,0	270,7	79312,0	78833,3
JPA/TopLink	140,0	47,0	66328,0	22411,3
LLBLGen PS	360,0	281,3	1500,0	1453,7
LLBLGen SS	125,0	120,0	625,0	635,3
SQL JDBC	78,0	73,0	172,0	156,0

Figura E.7: Test de Performance tipo QUERIES



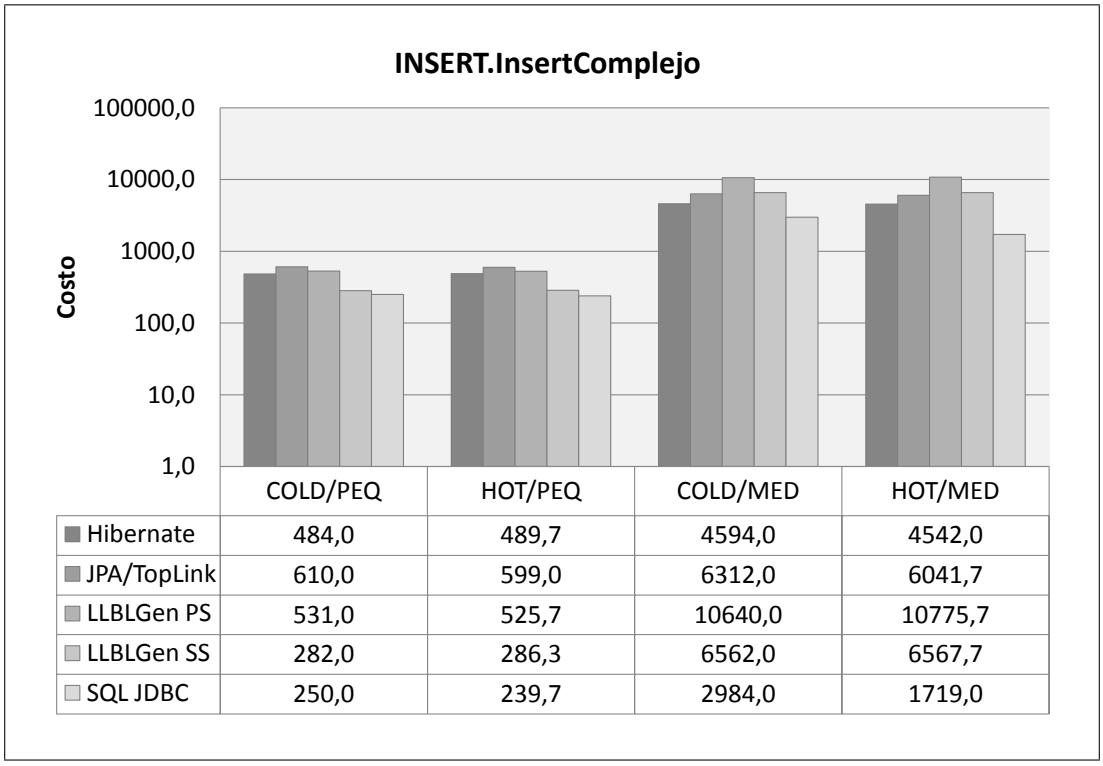
	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	328,0	250,3	77781,0	79015,7
JPA/TopLink	141,0	62,3	2828,0	2370,0
LLBLGen PS	343,0	396,0	1578,0	1499,7
LLBLGen SS	203,0	125,0	781,0	792,0
SQL JDBC	94,0	78,0	360,0	271,3

Figura E.8: Test de Performance tipo QUERIES



	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	0,0	36,0	31,0	21,0
JPA/TopLink	15,0	36,3	203,0	36,7
LLBLGen PS	157,0	140,0	141,0	151,3
LLBLGen SS	47,0	62,7	47,0	57,3
SQL JDBC	78,0	68,0	672,0	104,3

Figura E.9: Test de Performance tipo INSERT



	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	484,0	489,7	4594,0	4542,0
JPA/TopLink	610,0	599,0	6312,0	6041,7
LLBLGen PS	531,0	525,7	10640,0	10775,7
LLBLGen SS	282,0	286,3	6562,0	6567,7
SQL JDBC	250,0	239,7	2984,0	1719,0

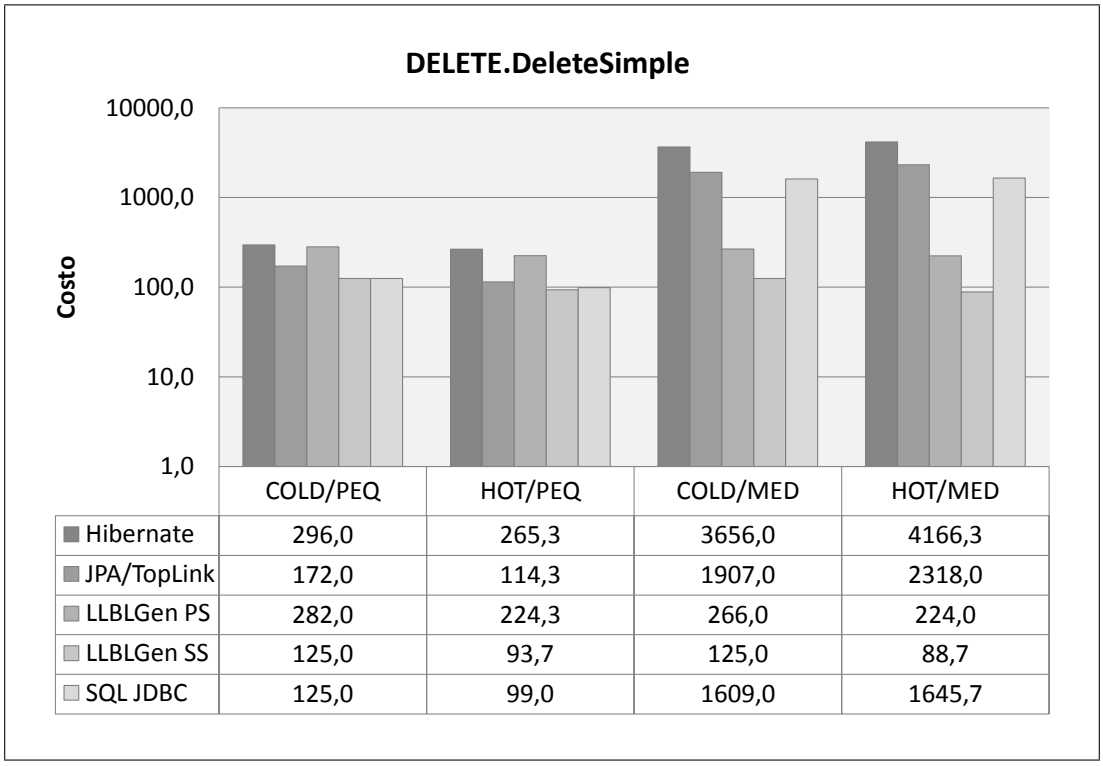
Figura E.10: Test de Performance tipo INSERT

DELETE

Este conjunto de tests se enfoca en la performance de otra operación básica de un mapeador, la eliminación de entidades persistidas. Los tests que lo componen son:

- *DELETE.DeleteSimple*. Este test realiza la eliminación de 120 entidades de tipo `Reserva`. Estas eliminaciones se consideran simples debido a que no implica la eliminación en cascada de entidades relacionadas (esto debido a que las reservas eliminadas en el tests no cuentan con huespedes registrados). Véase la Figura E.11.

- *DELETE.DeleteComplejo*. Este test realiza la eliminación de 120 entidades de tipo `Hotel`, incluyendo la eliminación en cascada de las entidades de tipo `Habitacion` y `Recepcion` relacionadas. Véase la Figura E.12.



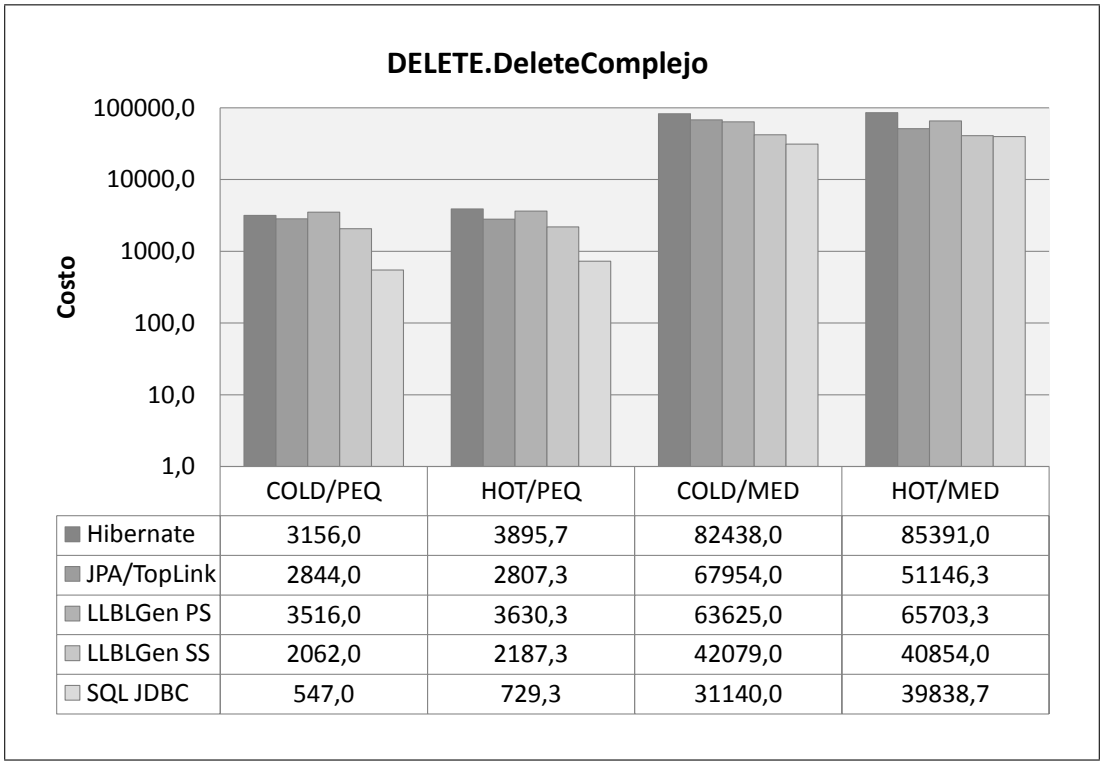
	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	296,0	265,3	3656,0	4166,3
JPA/TopLink	172,0	114,3	1907,0	2318,0
LLBLGen PS	282,0	224,3	266,0	224,0
LLBLGen SS	125,0	93,7	125,0	88,7
SQL JDBC	125,0	99,0	1609,0	1645,7

Figura E.11: Test de Performance tipo DELETE

UPDATE

Se trata de un conjunto de tests que pretenden evaluar la performance de la actualización de entidades existentes. Se consideran los siguientes tests:

- *UPDATE.UpdateSimple*. Actualiza 120 entidades de tipo `Reserva` modificando dos de sus atributos. Se trata de una actualización simple debido a que no involucra la modificación de relaciones con otras entidades. Véase la Figura E.13.



	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	3156,0	3895,7	82438,0	85391,0
JPA/TopLink	2844,0	2807,3	67954,0	51146,3
LLBLGen PS	3516,0	3630,3	63625,0	65703,3
LLBLGen SS	2062,0	2187,3	42079,0	40854,0
SQL JDBC	547,0	729,3	31140,0	39838,7

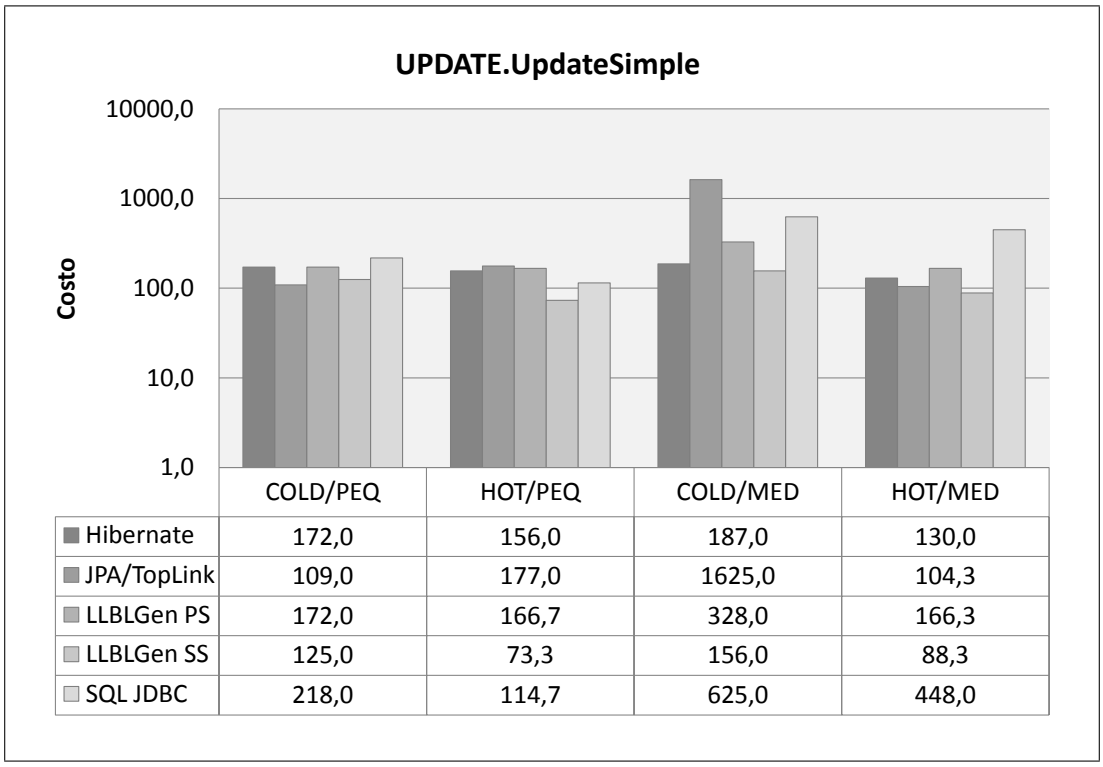
Figura E.12: Test de Performance tipo DELETE

- *UPDATE.UpdateComplejo*. Actualiza 120 entidades de tipo `Hotel`. La actualización incluye la creación y eliminación de habitaciones y recepciones para cada hotel como también la creación y eliminación de links entre las entidades. Véase la Figura E.14.

Análisis de Resultados

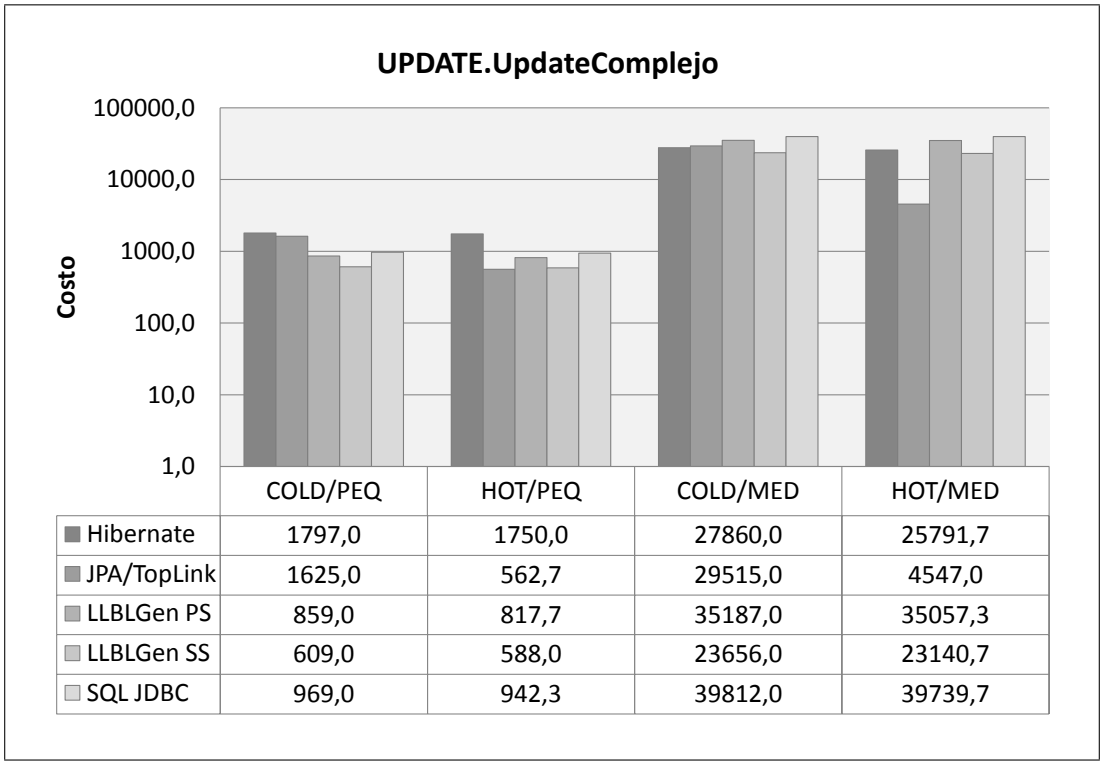
De los resultados obtenidos se puede concluir lo siguiente. Es claro que al considerar la totalidad de los tests no hay un mapeador que sea mas performante en todas las categorías. Es por esto que las conclusiones que se pueden deducir de los resultados se refieren a un subconjunto de tests. Se identifica una tendencia de Hibernate a ser mas lento con respecto a la solución de JPA/TopLink, sin ser así en los grupos de tests FIND e INSERT donde los resultados de ambos son muy similares. Al mismo tiempo se tiene una clara ventaja en performance en la utilización de LLBLGen Pro con respecto a los demás mapeadores en lo que se refiere a los grupos de tests QUERIES y DELETE. Nótese las diferencias en tiempo de ejecución de los diferentes tests para LLBLGen Pro sobre PostreSQL (utilizando el controlador Npgsql) y LLBLGen Pro sobre Microsoft SQL Server Express Edition, lo que ratifica la necesidad de considerar ambas combinaciones como soluciones diferentes en lo que a performance se refiere.

Un resultado a destacar del desarrollo de los tests para la plataforma Java es que, a menos que se cuente con amplios conocimientos en el uso de JDBC, la utilización de un mapador objeto-relacional es recomendable. Lo anterior se ve fundamentado en el hecho de que los mapeadores consideran estrategias de optimización del uso de JDBC que no siempre están al alcance inmediato del desarrollador. Ademas de esto los mapeadores integran



	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	172,0	156,0	187,0	130,0
JPA/TopLink	109,0	177,0	1625,0	104,3
LLBLGen PS	172,0	166,7	328,0	166,3
LLBLGen SS	125,0	73,3	156,0	88,3
SQL JDBC	218,0	114,7	625,0	448,0

Figura E.13: Test de Performance tipo UPDATE



	COLD/PEQ	HOT/PEQ	COLD/MED	HOT/MED
Hibernate	1797,0	1750,0	27860,0	25791,7
JPA/TopLink	1625,0	562,7	29515,0	4547,0
LLBLGen PS	859,0	817,7	35187,0	35057,3
LLBLGen SS	609,0	588,0	23656,0	23140,7
SQL JDBC	969,0	942,3	39812,0	39739,7

Figura E.14: Test de Performance tipo UPDATE

tecnologías de *caching* y similares que permiten mejorar la performance de recuperación y actualización de datos. Es asi que para el desarrollo con JDBC fue necesario valerse de varias optimizaciones particulares al caso a fin de lograr resultados mejores o comparables a los obtenidos con mapeadores.

E.2.5. Transparencia

Dada la definición de transparencia presentada en 3.4.1 surge el problema de cómo evaluarla. En concreto debe buscarse una forma de cuantificar el esfuerzo invertido al utilizar determinada herramienta. Debido a la complejidad en la evaluación de dicho criterio, y considerando además el alcance del proyecto, se propone la medición de transparencia en términos de lineas totales de código (incluyendo código fuente y archivos de configuración). En general se supone dicha medición como un indicador aceptable. Dependiendo de la herramienta en particular se desglosa la medida anterior según resulte de interés.

En la Figura E.15 se muestran las lineas totales de código para implementación del caso de estudio sin persistencia en comparación a las lineas de código totales una vez agregada esta cualidad. Se discrimina además según la plataforma. Nótese que como dato interesante para la plataforma Java, se agrega el valor de líneas de código totales para el caso de la resolución de la persistencia mediante SQL (JDBC).

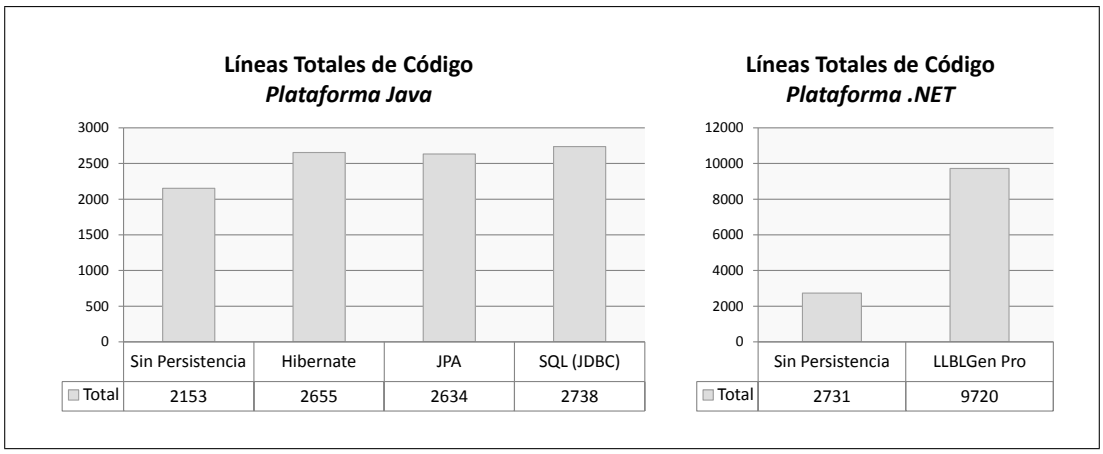


Figura E.15: Líneas Totales de Código Java y .NET

Para el caso de Hibernate se expone la cantidad de lineas de código fuente en contraste a las lineas de código de configuración y mapeo. En cuanto a JPA/TopLink, se presenta una división de las lineas totales de código en: lineas de código fuente (sin considerar *annotations*), de configuración y *annotations*. Finalmente en lo que refiere a LLBLGen Pro se realiza un desglose en lineas de código fuente generados de forma automática en contraste con las lineas de código bajo el control del desarrollador. Para todos los casos se distingue dentro del código fuente las líneas referentes a test unitarios y las correspondientes a la lógica de la aplicación. Refiérase al Cuadro E.3 para visualizar los valores obtenidos.

Téngase en cuenta que las lineas de código generadas de forma automática para el caso de LLBLGen Pro no afectan directamente la transparencia de la solución y son

Cuadro E.3: Desglose de Líneas de Código

Tipo	**Cantidad**
Hibernate	
Código Fuente Aplicación	1824
Código Fuente Tests	649
Configuración	23
Mapeo	159
Total	**2655**
JPA/TopLink	
Código Fuente Aplicación	1874
Código Fuente Tests	672
Configuración	42
Annotations	46
Total	**2634**
LLBLGen Pro	
Código Fuente Generado	7264
Código Fuente Desarrollador Aplicación	1546
Código Fuente Desarrollador Tests	910
Total	**9720**

presentadas como dato complementario. A fin de medir la transparencia de mejor forma, debería considerarse a parte de las lineas de código controladas por el desarrollador, el esfuerzo invertido en los pasos anteriores a la generación. Esto último incluye el diseño del esquema de base de datos y el diseño del mapeo objeto-relacional en la herramienta gráfica provista por el producto. Realizar esto se considera fuera del alcance del proyecto.

De los tres productos evaluados, se visualiza que LLBLGen Pro necesita de generación de código una vez definido el mapeo. Además, es necesario conocer el código generado ya sea para modificar las clases generadas o bien utilizar sus métodos. Esto afecta negativamente en la transparencia, en comparación con los demás productos.

Apéndice F

Gestión del Proyecto

En este anexo se detalla la planificación, coordinación y forma de trabajo realizada en el proyecto, mostrando las diferentes etapas que componen al proyecto y sus actividades principales.

El grupo de trabajo estuvo compuesto por tres estudiantes y la forma de trabajo consistió en la planificación y asignación de distintas tareas a cada miembro del grupo, estableciendo plazos de entrega de las mismas. Para los documentos, código y bibliografía, se utilizó un repositorio común de información, donde se almacenaron las distintas versiones de los documentos y código realizado. El seguimiento de las tareas planificadas, así como también los resultados obtenidos, fue coordinado por intermedio de reuniones quincenales con los tutores.

Con respecto a la planificación de actividades, cabe resaltar que no fue realizada antes de comenzar el proyecto, sino que se ha ido definiendo durante el transcurso del mismo. El proyecto se dividió en etapas, y cada etapa estuvo compuesta por distintas actividades. En cada una de las etapas definidas, se determinó la documentación a realizar y el tiempo estimado para su realización. El proyecto se enmarca en el período Marzo 2006 y Junio 2007, y se compone de las siguientes etapas:

F.1. Estado del Arte

Consiste en la búsqueda en amplitud de los temas y tecnologías relacionadas a Mecanismos de Persistencia, destacándose las siguientes actividades:

- Búsqueda de temas relacionados a Mecanismos de Persistencia
- Organización de la bibliografía
- Investigación de clasificaciones existentes

F.2. Propuesta de Clasificación de Mecanismos de Persistencia

En base a las clasificaciones existentes detectadas en la etapa de estado del arte, se propone una nueva clasificación de los mecanismos de persistencia. Se lista a continuación las actividades que componen la presente etapa:

- Definición de la clasificación
- Descripción de cada mecanismo perteneciente a la clasificación
- Elaboración de artículo sobre dicha propuesta
- Presentación del artículo en TACTiCS

F.3. Análisis y Definición de Criterios de Comparación

En esta etapa se definen y evalúan criterios de comparación entre los Mecanismos de Persistencia. Al final de la etapa se presentó el avance del proyecto, destacándose las siguientes actividades:

- Definición de criterios de comparación entre mecanismos
- Evaluación de los criterios definidos.
- Elaboración de presentación de Avance del Proyecto

F.4. Diseño y Desarrollo de Caso de Estudio

Con el objetivo de evaluar los criterios de comparación definidos para distintos productos de un mecanismo de persistencia en particular, se desarrolla un caso de estudio sobre un sistema de reservas de hoteles. Las actividades en este punto son:

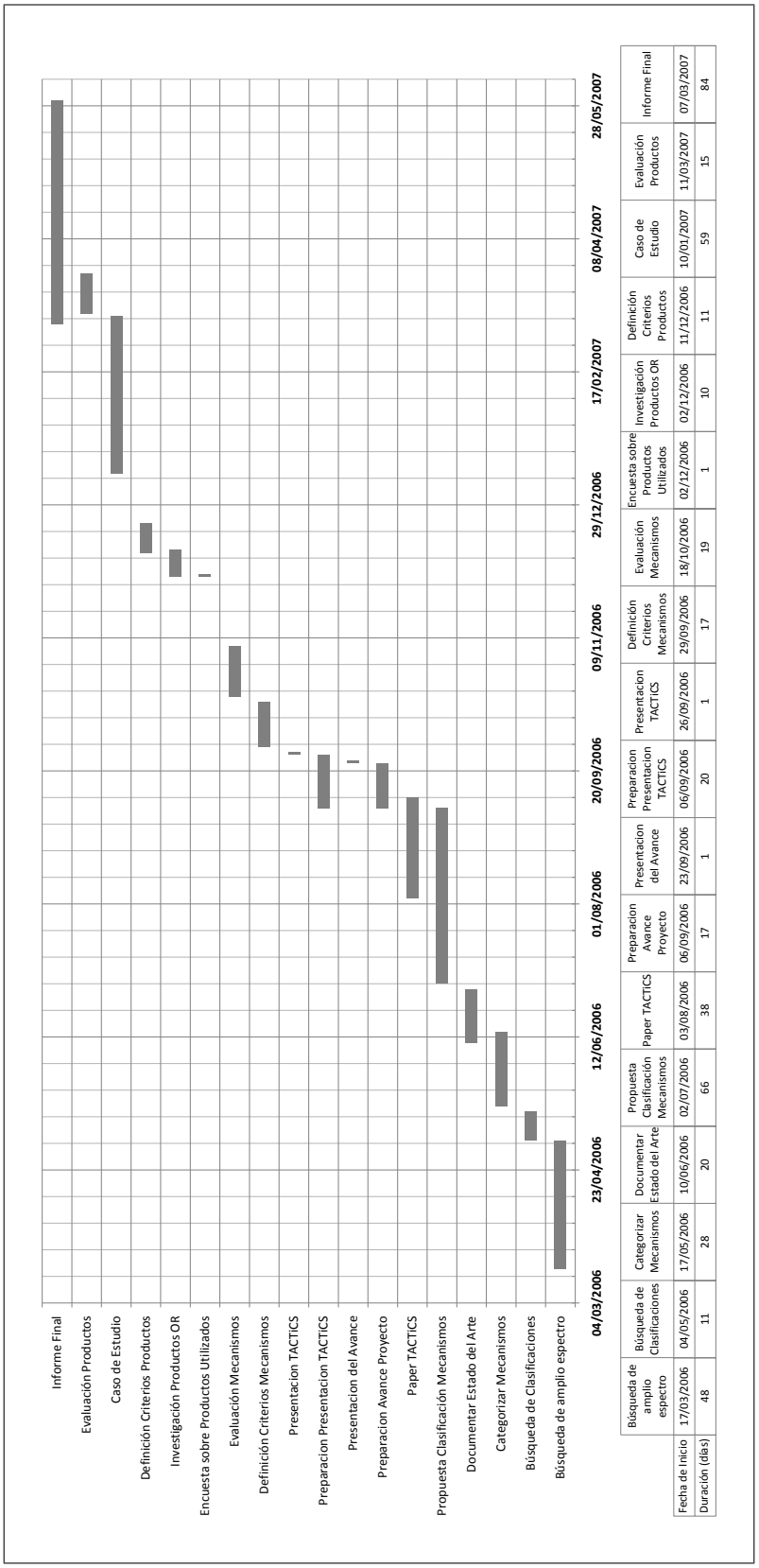
- Encuesta sobre productos utilizados
- Investigación sobre productos del mecanismo Mapeadores Objeto-Relacional
- Definición de criterios de comparación entre productos
- Selección del caso de estudio y determinación de alcance del mismo
- Desarrollo del caso de estudio para los productos seleccionados
- Evaluación de criterios de comparación.

F.5. Informe Final

Informe final del proyecto que intenta resumir y mostrar los resultados de las etapas planificadas.

F.6. Cronograma

En la Figura F.1 se muestra el cronograma de actividades del proyecto.



	Búsqueda de amplio espectro	Búsqueda de Clasificaciones	Categorizar Mecanismos	Documentar Estado del Arte	Propuesta Clasificación Mecanismos	Paper TACTICS	Preparacion Avance Proyecto	Presentacion del Avance	Preparacion Presentacion TACTICS	Presentacion TACTICS	Definición Criterios Mecanismos	Evaluación Mecanismos	Encuesta sobre Productos Utilizados	Investigación Productos OR	Definición Criterios Productos	Caso de Estudio	Evaluación Productos	Informe Final
Fecha de Inicio	17/03/2006	04/05/2006	17/05/2006	10/06/2006	02/07/2006	03/08/2006	06/09/2006	23/09/2006	06/09/2006	26/09/2006	29/09/2006	18/10/2006	02/12/2006	02/12/2006	11/12/2006	10/01/2007	11/03/2007	07/03/2007
Duración (días)	48	11	28	20	66	38	17	1	20	1	17	19	1	10	11	59	15	84

Figura F.1: Cronograma de Actividades

Bibliografía

[ADM+92] Malcolm Atkinson, David DeWitt, David Maier, François Bancilhon, Klaus Dittrich, y Stanley Zdonik. *The object-oriented database system manifesto.* páginas 1–20, 1992.

[Amb03] Scott Ambler. *Agile Database Techniques: Effective Strategies for the Agile Software Developer (Wiley Application Development).* Wiley, Octubre 2003.

[BK04] Christian Bauer y Gavin King. *Hibernate in Action (In Action series).* Manning Publications, 1era edición, Agosto 2004.

[Bou05] Ronald Bourret. *XML and Databases.* Septiembre 2005. `http://www.rpbourret.com/xml/XMLAndDatabases.htm`.

[CCM+06] Daniel Calegari, Jorge Corral, Pablo Miranda, Joaquín Prudenza, y Andrés Segurola. *Arquitectura de Acceso a Datos en Sistemas Orientados a Objetos: Clasificación y Descripción de Mecanismos de Persistencia. TCS Iberoamerica Technical Architect's Conference (TACTiCS)*, Agosto 2006.

[CDN93] Michael J. Carey, David J. DeWitt, y Jeffrey F. Naughton. *The OO7 Benchmark. SIGMOD Record (ACM Special Interest Group on Management of Data)*22():12–21, 1993.

[Cha06] Joseph Chancellor. *Rapid C# Windows Development: Visual Studio 2005, SQL Server 2005, and LLBLGen Pro.* Lulu Press, 1era edición, 2006.

[Che76] Peter Pin-Shan Chen. *The entity-relationship model—toward a unified view of data. ACM Trans. Database Syst.*1():9–36, 1976.

[CI05] William R. Cook y Ali H. Ibrahim. *Integrating Programming Languages & Databases: What's the Problem?* Octubre 2005.

[Cod] *Code Generation Network.* `http://www.codegeneration.net`.

[Cod83] E. F. Codd. *A relational model of data for large shared data banks. Commun. ACM*26():64–69, 1983.

[DK06] Linda DeMichiel y Michael Keith. *JSR 220: Enterprise JavaBeans^{TM}, Versión 3.0 Java Persistence API.* Sun Microsystems, 2006.

[EN06] Ramez Elmasri y Shamkant B. Navathe. *Fundamentals of Database Systems.* Addison Wesley, 5ta edición, Marzo 2006.

[Ext] *Using TopLink JPA Extensions.* `http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-extensions.html`.

[fADFC90] The Committee for Advanced DBMS Function Corporate. *Third-generation database system manifesto. SIGMOD Rec.*19():31–44, 1990.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series).* Addison-Wesley Professional, 1era edición, Enero 1995.

[Hib] *Hibernate.* `http://www.hibernate.org`.

[Jav] *Java Enterprise Edition (Java EE).* `http://java.sun.com/javaee`.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, y John Irwin. *Aspect-Oriented Programming.* En Mehmet Akşit y Satoshi Matsuoka, editores, *Proceedings European Conference on Object-Oriented Programming*, volumen 1241, páginas 220–242. Springer-Verlag, Berlin, Heidelberg, y New York, 1997.

[KS06] Mike Keith y Merrick Schincariol. *Pro EJB 3: Java Persistence API (Pro).* Apress, Mayo 2006.

[LLB] *LLBLGen Pro.* `http://www.llblgenpro.com`.

[MH96] J. Eliot B. Moss y Antony L. Hosking. *Approaches to Adding Persistence to Java First International Workshop on Persistence and Java, Drymen, Scotland.* Septiembre 1996.

[Noc03] Clifton Nock. *Data Access Patterns: Database Interactions in Object-Oriented Applications.* Addison-Wesley Professional, 1era edición, Septiembre 2003.

[PH05] Patrick Peak y Nick Heudecker. *Hibernate Quickly.* Manning Publications, 1era edición, Agosto 2005.

[Pol] *PolePosition.* `http://www.polepos.org`.

[PV03] Daniel Perovich y Andrés Vignaga. *SAD del Subsistema de Reservas del Sistema de Gestión Hotelera.* InCo-PEDECIBA, 2003.

[RJB04] James Rumbaugh, Ivar Jacobson, y Grady Booch. *Unified Modeling Language Reference Manual, The (The Addison-Wesley Object Technology Series).* Addison-Wesley Professional, 2da edición, Julio 2004.

[Sin06] Arti Singhal. *A Comparative Study of Data Access Mechanisms for Distributed Computing in J2EE Platform.* Julio 2006.

[SPA06] SPA. *Software Practice Advancement 2006 (SPA 2006).* En *Guidance On The Choices For Persistence*, 2006. `http://www.spaconference.org/cgibin/wiki.pl/?GuidanceOnTheChoicesForPersistence`.

[Tak05] Shinji Takasaka. *Survey of Persistence Approaches.* Tesis de Maestría, Royal Institute of Technology/Stockholm University en colaboración con Swiss Federal Institute of Technology Zurich, Diciembre 2005.

[Wue03] Klaus Wuestefeld. *Do you still use a database?* En *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, páginas 101–101, New York, NY, USA, 2003. ACM Press.

[Yal04] Sunay Yaldiz. *Evaluation of Web Application Development Frameworks and Object-Relational Mappers: A Case Study.* Febrero 2004.

Glosario

A

Annotation Tipo de metadata a nivel del código fuente que se encuentra disponible en tiempo de ejecución para el desarrollador.

ANSI *American National Standards Institute.* Organización sin fines de lucro que supervisa el desarrollo de estándares para productos, servicios, procesos y sistemas en los Estados Unidos.

API *Application Programming Interface.* Conjunto de funciones y procedimientos (o métodos si se refiere a programación orientada a objetos) que ofrece cierta librería para que sean utilizados por otro software como una capa de abstracción.

Arquitectura Cliente/Servidor Arquitectura de software que se basa en la separación entre cliente y servidor. Cada instancia de cliente envía pedidos a uno o más servidores. De esta forma, los servidores aceptan estas peticiones, las procesan, y retornan la información solicitada a los clientes.

B

Benchmark Técnica utilizada para medir el rendimiento de un sistema o componente de un sistema.

C

CLOB *Character Large Object.* Se trata de un tipo de datos utilizado por DBMSs que representa una extensa colección de caracteres que normalmente excede los 4000 caracteres en longitud.

Consultas Dinámicas Consultas que son creadas en tiempo de ejecución sin previa definición.

Consultas Modularizadas Consultas cuya definición se extiende a varios módulos. El mecanismo de modularización disponible es propio del mecanismo de consultas considerado.

Consultas Parametrizadas Tipo de consultas provisto por algunos mecanismos de consulta que permite la definición de parámetros a ser seteados en tiempo de ejecución.

CSV *Comma Separated Values.* Formato de archivo que almacena datos de forma tabular utilizando el caracter *coma* como separador de valores.

D

DAO *Data Access Object.* Patrón de Diseño que permite abstraer y encapsular todos los accesos a la fuente de datos con la que se esté tratando.

Data Warehouse Colección de datos orientada a un dominio, integrada, no volátil y que varía en el tiempo. Su propósito es el de ayudar a una empresa u organización en la toma de decisiones.

Datos Legados Información en la cual una organización pudo haber invertido una cantidad significante de recursos y que, a lo largo del tiempo, ha retenido su importancia. Se caracteriza por haber sido creada o almacenada mediante el uso de software y/o hardware que hoy en día se considera obsoleto.

DOM *Document Object Model.* API para acceder, añadir y cambiar dinámicamente contenido estructurado en documentos XML.

DTD *Document Type Definition.* Lenguaje para la especificación de esquemas XML. De acuerdo con esto, DTD permite definir una serie de reglas que determinados documentos XML deben cumplir a fin de ser considerados como válidos.

E

EJB *Enterprise JavaBeans.* API que forma parte del estándar para la construcción de aplicaciones empresariales Java EE. Su especificación detalla cómo los servidores de aplicaciones proveen objetos desde el lado del servidor. Dichos objetos son precisamente los EJBs.

EJB-QL *Enterprise JavaBeans Query Language* Lenguaje de consulta que permite definir métodos *finder* para Entity Beans con CMP. La definición utiliza un lenguaje basado en SQL que permite efectuar búsquedas sobre los atributos persistentes de un Enterprise JavaBeans y sus beans asociados.

Ejecución en Caliente Método de ejecución de tests de performance utilizado en *benchmarks* de bases de datos, mapeadores, etc. Una ejecución en caliente implica la ejecución consecutiva del mismo test, de esta forma las ejecuciones posteriores a la primera cuentan con los datos almacenados en el diferentes caches presentes en la herramienta. Esto último afecta los resultados de los tests de performance y permite obtener el valor de los mismos en régimen.

Ejecución en Frio Método de ejecución de tests de performance utilizado en benchmarks de bases de datos, mapeadores, etc. Una ejecución en frío de un test implica que

los diferentes caches que forman parte de la arquitectura de caches presentes en la herramienta, están vacíos.

H

HQL *Hibernate Query Language.* Lenguaje de consulta orientado a objetos provisto por la herramienta Hibernate.

I

ISO *International Organization for Standardization.* Organización internacional no gubernamental, compuesta por representantes de los organismos de normalización nacionales, que produce normas internacionales industriales y comerciales.

J

Java EE *Java Enterprise Edition.* Plataforma de programación para desarrollar y ejecutar software de aplicaciones en el lenguaje de programación Java con arquitectura de varios niveles, distribuida, basándose ampliamente en componentes de software modulares que ejecutan sobre un servidor de aplicaciones.

Java SE *Java Standard Edition.* Plataforma para desarrollar y ejecutar aplicaciones de uso general en el lenguaje de programación Java.

JAXB *Java Architecture for XML Binding.* Permite a los desarrolladores Java crear y editar documentos XML mediante objetos. Quienes utilicen JAXB, simplemente requieren de dos pasos para el análisis de un documento XML (el cual requiere un esquema). El primero es el proceso de *binding*, en el que se generaran las clases que representan los elementos y las relaciones del esquema. El segundo es el proceso de *unmarshalling*, en el que se construye el árbol de objetos que representa los datos contenidos en el documento XML.

JCP *Java Community Process.* Proceso formalizado el cual permite a las partes interesadas involucrarse en la definición de futuras versiones y características de la plataforma Java.

JDBC *Java Database Connectivity.* API que permite la ejecución de operaciones sobre bases de datos desde el lenguaje de programación Java independientemente del sistema donde se ejecute o de la base de datos a la cual se accede.

JDOM Modelo de objetos abierto para la plataforma Java que permite la lectura, manipulación y escritura de documentos XML de forma simple y eficiente. Se integra con DOM y SAX, soporta XPath y XSLT. Asimismo se basa en parsers externos para construir las representaciones en memoria de los documentos XML.

JDOQL *Java Data Objects Query Language.* Lenguaje de consulta a nivel de objetos para la plataforma Java provisto por la interfaz JDO.

JNDI *Java Naming and Directory Interface.* API para servicios de directorio. Esto permite a los clientes descubrir y buscar objetos y datos a través de un nombre definido.

JP-QL *Java Persistence Query Language.* Lenguaje de consulta sucesor de EJB-QL.

JPA *Java Persistence API.* Se trata de un *framework* para la plataforma Java que permite el mapeo de objetos a una base de datos relacional.

JSR *Java Specification Request.* Documento formal que describe las especificaciones y tecnologías propuestas para que sean añadidas a la plataforma Java.

L

LINQ *Language Integrated Query.* Proyecto de Microsoft que pretende revolucionar el acceso a datos mediante un conjunto de extensiones para .NET Framwork que integra un lenguaje de consulta a los diferentes lenguajes de programación del mismo.

M

Mecanismo de Persistencia Se define Mecanismo de Persistencia, en el contexto del desarrollo de un sistema orientado a objetos, como una técnica básica que permite resolver la *persistencia* de objetos.

MER *Modelo Entidad Relación.* Uno de los varios modelos conceptuales existentes para el diseño de bases de datos. Los elementos esenciales del modelo son las entidades, los atributos y las relaciones entre las entidades.

Metadata Metadata es información sobre un conjunto particular de datos.

MFC *Microsoft Foundation Classes.* Librería de Microsoft, introducida en 1992, que abstrae porciones del API de Windows en clases C++.

Middleware Componente de software que permite la conexión entre dos o más elementos de forma tal que los mismos puedan intercambiar información.

Model Driven Development Técnica de desarrollo e integración de sistemas que se basa en el uso de modelos para dirigir el curso de entendimiento, diseño, construcción, deployment, operación, mantenimiento y modificación de los mismos.

Modelo de Datos Un modelo de datos es un sistema formal y abstracto que permite describir los datos de acuerdo con reglas y convenios predefinidos.

Modelo de Datos de Red Modelo de datos que representa la información mediante un grafo de registros.

Modelo de Datos Jerárquico Modelo de datos que representa la información de forma similar a un árbol mediante relaciones padre/hijo entre registros.

O

off-the-shelf Término utilizado para referenciar software, hardware o productos tecnológicos en general que están disponibles en el mercado para ser comprados, alquilados o ser utilizados de forma gratuita por el público en general.

OMG *Object Management Group.* Consorcio sin fines de lucro dedicado al cuidado y el establecimiento de diversos estándares de tecnologías orientadas a objetos (p.ej. UML).

one-size-fits-all Término utilizado para indicar que el sujeto en cuestión aplica en todos los contextos.

Ontología Término que hace referencia al intento de formular un exhaustivo y riguroso esquema conceptual dentro de un dominio dado. De acuerdo con ello, se puede utilizar una ontología para una variedad de propósitos que incluyen el razonamiento inductivo, la clasificación, y una variedad de técnicas de resolución de problemas.

OQL *Object Query Language.* Lenguaje de consulta estándar para bases de datos orientadas a objetos modelado a partir de SQL.

P

POJO *Plain Old Java Object.* En el contexto del trabajo, término utilizado para referirse a objetos Java que no necesitan de ninguna particularidad específica a fin de poder ser persistidos.

Propiedades ACID *Atomicity, Consistency, Isolation, Durability.* Conjunto de propiedades que garantizan que las transacciones de una base de datos sean procesadas de forma confiable.

Q

QBE *Query by Example.* Familia de lenguajes de consulta para base de datos que implementan las ideas del cálculo relacional de dominio.

R

READ-CENTRIC Término utilizado para referirse a aquellas aplicaciones que pasan la mayor parte del tiempo realizando operaciones de lectura de datos persistidos.

Reflection Tecnología disponible en lenguajes como Java y C# que permite, en tiempo de ejecución, descubrir información sobre campos, métodos y constructores de las clases cargadas. Asimismo permite efectuar la invocación dinámica de estos dos últimos.

S

SAX *Simple API for XML.* Se trata de un protocolo de acceso serial a un documento XML basado en eventos. Un parser que implementa SAX genera un evento cada vez que se abre o se cierra un elemento XML, o bien cuando se produce algún error.

Serialización Proceso por el cual se puede codificar un objeto en un medio de almacenamiento (p.ej. un archivo o un buffer en memoria).

Servidor de Aplicaciones Servidor en una red de computadores que ejecuta ciertas aplicaciones y que brinda a las mismas un conjunto de servicios bien definidos.

Sistema de Información Conjunto de funciones o componentes interrelacionados encargados de obtener, procesar, almacenar y distribuir información a fin de asistir en la toma de decisiones y el control en una organización.

Snapshot Término utilizado para referirse a un conjunto de elementos que permiten recuperar sin ambigüedad cierto estado anterior de determinado sistema.

T

Taxonomía Clasificación que ordena, jerarquiza y nombra categorías.

Tuning Proceso de optimización de un programa o sistema para un ambiente en particular.

X

XML-QL Lenguaje de consulta para documentos XML propuesto por la W3C. El mismo puede expresar consultas que extraen porciones de datos de un documento XML, así como también transformaciones que pueden mapear datos entre DTDs diferentes o integrar datos de diferentes fuentes.

XML Schema Lenguaje para la especificación de esquemas XML. De acuerdo con esto, XML Schema permite definir una serie de reglas que determinados documentos XML deben cumplir a fin de ser considerados como válidos.

XPath Lenguaje que permite seleccionar porciones de un documento XML o computar valores (p.ej. numéricos, booleanos, etc.) basándose en el contenido del mismo.

XQuery Lenguaje de consulta (con algunas características de los lenguajes de programación) diseñado para consultar colecciones de datos XML.

XSL *Extensible Stylesheet Language.* Familia de lenguajes de transformación basados en XML que permiten describir cómo documentos XML deben ser formateados o transformados.

XSL-FO *XSL Formatting Objects.* Perteneciente a la familia de lenguajes XSL, permite realizar el formateo visual de documentos XML. Comúnmente se utiliza para producir, a partir de documentos XML, documentos PDF.

XSLT *XSL Transformations.* Perteneciente a la familia de lenguajes XSL, permite realizar transformaciones de documentos XML. XSLT está diseñado para transformar documentos XML en otros documentos XML.