Facultad de Informática
Universidad de Murcia

PERSISTENCIA DE OBJETOS. JDO, SOLUCIÓN JAVA

Juan Mármol Castillo
Director: Dr. Jesús Joaquín García Molina

Agradecimientos

A mi esposa María José, a mis hijos Juan y María, que durante estos últimos años, han sufrido mis constantes ausencias con compresión y cariño.

A mi hermana Loli y cuñado Paco, que desde siempre tanto me han ayudado.

A mis amigos que no han dejando de alentarme para completar con este trabajo los estudios de informática.

Prólogo

La persistencia es un problema que siempre me ha interesado, por mi trabajo como programador y como estudiante de la orientación a objetos. En mi trayectoria como programador de aplicaciones, primero con lenguajes de cuarta generación, y tiempo después, con lenguajes orientados a objetos, guardar y recuperar datos de ficheros y bases de datos relacionales, consumían buena parte del esfuerzo de desarrollo. Participé en la programación de un conjunto de librerías de objetos que automatizaban la interacción en SQL con la base de datos relacional para formularios y listados, pero la solución no era completamente satisfactoria, no era transparente, los modelos de objetos y datos eran distintos, se podía programar sin un modelo de objetos del dominio. La asistencia a conferencias de Oscar Díaz, Mario Piattini, K.Dittrich, y Elisa Bertino, sobre las bases de datos orientadas a objetos (OODBMS), provocó mi interés por conocer estos sistemas y estudiar la solución al problema de persistir objetos en bases de datos relacionales.

Los planes de estudio, que he tenido ocasión conocer más de cerca, contienen créditos de formación sobre la orientación a objetos, las bases de datos relacionales y JAVA, con diferentes enfoques y amplitud. Materias todas ellas relacionadas con el problema la persistencia orientada objetos, que en cambio, no ocupa un lugar propio en los planes de estudio, pese a ser, en mi opinión, un tema esencial para el trabajo de un ingeniero de informática, dominar las técnicas de cómo transformar los objetos en datos y hacerlos perdurar y viceversa.

Este prólogo precede, un trabajo que, comenzó tres años atrás, cuando aparecían las primeras noticias sobre la propuesta de creación de un nuevo estándar para persistir los objetos Java, el Java Data Objects (JDO), tres años más tarde, el estándar se había convertido en uno de los temas Java más populares, con un aparente creciente respaldo, numerosos trabajos y discusiones abiertas.

La cuestión de la persistencia es amplía y afecta a distintas materias, como son las bases de datos, lenguajes de programación, compiladores, la orientación a objetos y la programación, que dan lugar a diferentes perspectivas y planteamientos posibles. De otro lado, para muchos el problema no es tal, hasta enfrentarse a este reiteradas veces, la persistencia siempre tiene una consideración lateral, ¡sólo es guardar y recuperar datos! Pero mi experiencia, me lleva a pensar que es necesaria una visión global para entender el problema en su extensión y comprender la conveniencia de una solución completa a la persistencia de objetos.

Aquellos que no han necesitado persistir sus objetos, pero vayan a hacerlo, y quienes no encuentran una solución satisfactoria a sus necesidades, pero deban decidir entre comprar o escribir su propia solución; espero encuentren en las páginas siguientes elementos de juicio que sirvan para valorar cual debe ser su decisión.

Introducción

El desarrollo de aplicaciones para sistemas de información empresarial ideados para su utilización en la Red, es un área en continua expansión desde mediados de los noventa. Estas aplicaciones deben guardar y recuperar datos que deben perdurar tras finalizar la ejecución de las mismas, datos que deben persistir, ser persistentes. Estos datos son guardados sobre diferentes sistemas de gestión y manejo de datos como son archivos, bases de datos o sistemas de procesamiento de transacciones.

En mayo del 1995 fue presentado JAVA[49], cuyo nivel de aceptación es cada vez mayor, en el desarrollo de programas dirigidos a la Red. Las aplicaciones Java que manejan datos se enfrentan también con la dificultad de guardar y recuperar datos, que están situados sobre distintos tipos de fuentes de información o sistemas de gestión de datos. Medios estos que están solidamente implantados en las organizaciones y que son el motor de datos de sus sistemas de información. Otro área en auge donde Java esta creciendo es, el mercado de los nuevos aplicativos diseñados para los pequeños dispositivos conectados a la Red, que aparentemente van a cambiar nuestras vidas [21][43][1], y que también necesitan guardar y recuperar datos de estado, pensemos en un moderno asistente personal conectado vía telefonía celular a la Red, con agenda, avisos y alarmas. La persistencia de los objetos es necesaria en todo tipo de entornos, abarca desde los grandes sistemas a dispositivos del tamaño de una tarjeta de crédito.

Las aplicaciones Java que necesitan manejar datos que deben perdurar, plantean un escenario donde los programadores están obligados a utilizar y conocer detalladamente diferentes técnicas de acceso e interfaces de programación para cada uno de los sistemas de gestión de datos empleados. Esto significa conocer como poco:

- Dos lenguajes distintos para plasmar la lógica del negocio: Java y el lenguaje especializado requerido por el sistema de gestión de datos.
- El modo de integrar ambos evitando las resistencia por la falta de correspondencia (Impedance mismatch) de uno y otro lenguaje; excepción hecha de los sistemas de gestión de datos basados en objetos

Esta situación ha impulsado la construcción de servicios para almacenar y recuperar datos, servicios de persistencia, que desempeñen su labor de forma transparente, uniforme e independiente del sistema plataforma de gestión de datos. Esto permitiría facilitar la tarea del programador, siendo más productiva y centrada en lógica del negocio. La demanda de servicios de persistencia transparentes, es y ha sido fuente de gran interés para la investigación con importantes proyectos [33] y para la industria con numerosos productos [52].

La persistencia del estado de los objetos tiene sólidos fundamentos, teóricos y prácticos, con más de una década [1][2]. Las tecnologías de persistencia de objetos han sido aplicadas con éxito a proyectos con requisitos de altas prestaciones en áreas de los negocios de las telecomunicaciones como Iridium [54] y bancos de datos científicos como acelerador de partículas de Standford [46], desde principios de los noventa, pero sin gran repercusión en el mundo del desarrollo de aplicaciones de empresa.

El OMG tiene un servicio estándar de persistencia de estado de los objetos independiente del lenguaje de programación e interoperable, denominado Persistent State Service (PSS), su objetivo es manejar el estado persistente de los objetos CORBA, dentro del

[1] Java, primero llamado Oak, fue inicialmente concebido para unos dispositivos de mano aplicados a la televisión por cable dentro del proyecto Green [i1]

ámbito de la comunicación interna entre servicios CORBA y servicios de datos, la interfaz entre ambos para la persistencia del estado de objetos CORBA; probablemente este estándar sea poco apropiado como interfaz a una base de datos pues considera el soporte de transacciones como opcional[22].

La persistencia con Java es un caso particular del problema de la persistencia con objetos donde, hasta finales del año 2001, había cuatro estándares principales para manejar datos Java persistentes: la serialización, Java Database Connectivity (JDBC), SQLJ y la adaptación para Java del estándar ODMG. La serialización preserva la integridad referencial de los objetos, pero no soporta la concurrencia de acceso por múltiples sesiones de usuario. JDBC necesita que el desarrollador explícitamente maneje la correspondencia entre el estado de los objetos y su proyección sobre relaciones del sistema gestor de bases de datos relacional (RDBMS) en uso y conocer otro lenguaje, SQL. SQLJ ofrece simplificar el código Java, al embeber el código SQL estático en los programas, utiliza JDBC para el SQL dinámico y ofrece soporte al modelo de objetos del SQL:1999. Ninguno de los tres cubre satisfactoriamente la cuestión de la persistencia. El último, la adaptación para Java del ODMG, es hasta la fecha la mejor tecnología disponible para resolver la persistencia en Java en opinión de David Jordan y Douglas Barry [34], expertos de renombre sobre la materia de la persistencia, ideado para persistir sobre RDBMS y OODBMS. De este último estándar existen en el mercado acreditados productos como Java Blend de Sun, Jasmine de Computer Associates, FastObjects de Poet, Objecstore de Progress, Versant, Objectivity y otros.

La especificación para datos de objetos Java, Java Data Objects (JDO) desarrollada bajo el Sun Community Process con la identificación JSR-12, es la nueva propuesta para resolver el problema de la persistencia de objetos Java, sobre cualquier sistema de gestión de datos, tanto en entornos reducidos, como un teléfono Java, como en los grandes sistemas escalables de servidores de aplicaciones. El proceso comenzó en Julio de 1.999 y fue aprobado como estándar el 25 de Marzo de 2002.

Objetivos

Este trabajo es una aproximación al problema de la persistencia de objetos y su solución en Java, con el estándar JDO, que pretende:

i. Mostrar el problema de la persistencia estableciendo un marco de análisis sobre criterios estructurales, de organización, funcionales, tecnológicos y económicos.
ii. Exponer una solución al problema de convertir clases en tablas y objetos en filas almacenadas en una base de datos relacional.
iii. Presentar brevemente la especificación JDO: objetivos, alcance, funcionalidad, situación actual y futura. No ser un manual de referencia de JDO.
iv. Elaborar una comparativa frente a las otras alternativas de persistencia Java, para analizar JDO a luz del marco establecido de requisitos.
v. Presentar unas conclusiones que ayuden a formar la opinión de quien lea este proyecto, ayudando a decidir cuando utilizar JDO.

Para lograr los objetivos marcados ha sido desarrollado el siguiente planteamiento:

1. Estudio de la documentación relativa a distintos aspectos del problema, recabada de diferentes autores mencionados a lo largo texto e incluidos en las referencias y bibliografía.

2. Llevar a cabo un proceso de descubrimiento de los requisitos, analizando distintas situaciones, desde el planteamiento más simple a la práctica más compleja, para poner de manifiesto y fundamentar los requisitos presentados. Diversos programas Java irán ilustrando las situaciones que serán discutidas.
3. Elaborar unas pruebas basadas en sencillos programas Java, para confrontar las tecnologías y los requisitos, empleando algún producto comercial JDO. Una de las pruebas consiste en la modificación una aplicación existente que usa JDBC para medir el impacto en la productividad de JDO frente JDBC.

Temas no considerados

El tema de la persistencia de objetos y el estudio en profundidad de todos los aspectos implicados en la especificación JDO, es muy extenso, para un solo proyecto fin de carrera. Por lo cual, en este trabajo, no son considerados los siguientes puntos:

- Problemática de desmenuzamiento y duración de transacciones. Relación con servicios de transacciones.
- Tratamiento de objetos de gran tamaño: objetos multimedia, estructuras complejas de gran volumen.
- Objetos distribuidos. Objetos agregados de otros dispersos sobre diferentes sistemas.
- Optimización de modelos para acceso. Mejoras en las técnicas de acceso.
- Integración en J2ME, Java 2 Micro Edition, para dispositivos que conectan a la Red.

Organización del proyecto

El presente documento se organiza en cinco capítulos:

- El primer capítulo, responde a las preguntas: ¿Qué es la persistencia de objetos? ¿Cuál son los requisitos a cubrir? Establece los fundamentos, los términos y requisitos que forman el marco de análisis utilizado a lo largo del trabajo.
- El segundo, trata sobre la equivalencia de clases en tablas relacionales, ilustra una solución al problema de falta de correspondencia entre clases y tablas de las bases de datos relacionales
- En tercero de los capítulos, se descubre Java Data Objects, presentando la especificación de una forma breve y resumida, pero útil para los objetivos del proyecto.
- El capítulo número cuatro, es donde se comparan las alternativas a JDO dentro del marco establecido en primer capítulo, se razona sobre le rendimiento y la productividad, mostrando una pruebas simples pero, que ayudan a contrastar diferentes postulados.
- En el último, son expuestas las ventajas e inconvenientes de trabajar con JDO, su situación en panorama económico y posible evolución futura, junto con las respuestas a algunas interrogantes que surgen tras leer los cuatro primeros capítulos..

Tabla de contenidos

Índice de figuras

Índice de listados Java

Índice de tablas

Capítulo 1
PERSISTENCIA DE OBJETOS: Conceptos y Requisitos

Este capítulo presenta el problema de la persistencia de los objetos de las aplicaciones, y las características de una solución para persistir objetos. Se identifica un conjunto de requisitos, que servirá de base para valorar JDO, como solución Java a la persistencia de objetos.

En primera parte del capítulo, para conocer más de cerca el problema de la persistencia, son expuestos una serie de conceptos y términos fundamentales, empleados a lo largo de este documento. En la segunda parte de este capítulo, se intenta desmenuzar el problema tratando descubrir los requisitos que se deberían satisfacer, estudiando distintas situaciones donde deben ser guardados y recuperados objetos considerando diferentes planteamientos y una complejidad creciente.

1.1. Conceptos sobre persistencia de objetos

A continuación, se introducen una serie conceptos y definiciones, de forma clara y sencilla, tratando de evitar distraer la atención del tema, o promover guerras semánticas sobre el significado de los términos empleados y sus acepciones. Nuevas nociones que vierten luz sobre el tema que nos interesa, y al mismo tiempo, sirven de base para formar un criterio propio.

1.1.1. Persistencia

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

La primera, más antigua, dice así: *«Es la capacidad del programador para conseguir que sus datos sobrevivan a la ejecución del proceso que los creo, de forma que puedan ser reutilizados en otro proceso. Cada objeto, independiente de su tipo, debería poder llegar a ser persistente sin traducción explicita. También, debería ser implícito que el usuario no tuviera que mover o copiar los datos expresamente para ser persistentes»* [2].

Esta definición nos recuerda que es tarea del programador, determinar cuando y cómo una instancia pasa a ser persistente o deja de serlo, o cuando, debe ser nuevamente reconstruida; asimismo, que la transformación de un objeto en su imagen persistente y viceversa, debe ser transparente para el programador, sin su intervención; y que todos los tipos, clases, deberían tener la posibilidad de que sus instancias perduren.

La otra definición dice así: Persistencia es *«la capacidad de un lenguaje de programación o entorno de desarrollo de programación para, almacenar y recuperar el estado de los objetos de forma que sobrevivan a los procesos que los manipulan»[19]*

Esta definición indica que el programador no debería preocuparse por el mecanismo interno que hace un objeto ser persistente, sea este mecanismo soportado por el propio lenguaje de programación usado, o por utilidades de programación para la persistencia, como librerías, framework o compiladores.

En definitiva, el programador debería disponer de algún medio para poder convertir el estado de un objeto, a una representación adecuada sobre un soporte de información, que permitirá con posterioridad revivir o reconstruir el objeto, logrando que como programadores, no debamos preocuparnos de cómo esta operación es llevada a cabo.

1.1.2. Instancia Persistente y Transitoria

Una instancia persistente es aquella cuyos datos perduran a la ejecución del proceso que materializó la instancia. Una instancia transitoria o temporal, es toda instancia cuyos datos desaparecen cuando finalizan los procesos que la manipulan. En ambos casos, las instancias en sí, como estructuras de datos residentes en memoria, desaparecen al finalizar los procesos que las crearon.

Veámoslo con un sencillo ejemplo, imaginemos la ejecución de un programa que solicita introducir nuestro nombre que será usado repetidas veces en distintas operaciones. Si este dato es recogido en una instancia transitoria, cuando finalice el programa y lo volvamos a ejecutar, deberemos nuevamente introducir el dato; pero si está asociado a una instancia persistente, el dato introducido podría ser recuperado y mostrado en sucesivas ejecuciones del programa.

1.1.3. Servicio de persistencia de objetos

El concepto de servicio de persistencia es tema vigente de debate e investigación y desarrollo, en los mundos, académico y de la industria. Acotar con nitidez qué es un servicio de persistencia, es una cuestión abierta, porque la separación entre este concepto y el de base de datos es difusa. Aquí, asumiendo que el destino final de los datos serán principalmente los sistemas de gestión de datos de la empresa, es adoptada la siguiente acepción:

> Servicio de persistencia es *un sistema o mecanismo programado para posibilitar una interfaz única para el almacenamiento, recuperación, actualización y eliminación del estado de los objetos que pueden ser persistentes en uno o más sistemas gestores de datos.*

La definición hecha, considera que el sistema gestor de datos, puede ser un sistema RDBMS, un sistema OODBMS, o cualquiera otro sistema; que el estado podría estar repartido sobre varios sistemas, incluso de distinto tipo; y lo más importante, que un servicio de persistencia de objetos aporta los elementos necesarios para efectuar la modificación y la eliminación de los objetos persistentes, además del volcado y recuperación del estado en los sistemas gestores de datos. Y todo ello, debería ser efectuado de acuerdo a la definición hecha más atrás de persistencia, sin necesidad de traducción explicita por parte del programador. En todos los casos, sea cual sea el tipo de gestor datos, los servicios de persistencia de objetos, facilitan la ilusión de trabajar con un sistema de bases de datos de objetos integrado con el lenguaje de programación, ocultando las diferencias entre el modelo de objetos del lenguaje y el modelo de datos del sistema empleado como base de datos. A pesar de lo dicho, un servicio de persistencia no es un sistema de gestión de bases de datos orientado a objetos. El servicio de persistencia es un componente esencial de todo sistema gestor de bases de datos de objetos (ODBMS), que resuelve otros aspectos, además de la persistencia [2]. Más adelante, el concepto de servicio de persistencia será matizado, a través de los requisitos que debe cubrir, entonces será cuando la idea de servicio quedará más nítida, frente a la idea de que un servicio de persistencia podría ser solo un interfaz vacío de funcionalidad para acceder a bases de datos.

De las alternativas estándar para hacer persistir los objetos en Java, comentadas en la introducción (la serialización, JDBC, SQLJ, ODMG 3.0 y JDO), solo ODMG 3.0 y JDO pueden tener la consideración de servicio de persistencia de objetos Java. También es posible proponer utilizar un servicio de persistencia del OMG, PSS 2.0, para persistir objetos Java, pero este estándar, considera opcionales las transacciones y la persistencia transparente [22], que son funciones necesarias para ofrecer un servicio de persistencia eficaz, conforme a las definiciones vistas de persistencia. JDBC, SQLJ requieren que el programador defina, e implemente las operaciones de persistencia teniendo en cuenta cómo convertir los objetos en tuplas. Ambos pueden ser utilizados en la construcción de servicios de persistencia para bases de datos relacionales. La serialización es el servicio de persistencia más básico, solo ofrece servicios para guardar y recuperar el estado de un objeto sobre ficheros y flujos de entrada salida.

A partir del diagrama de clases UML de la Figura 1, veamos cómo guardar un objeto `Cliente` en Java, con cada uno los estándares mencionados de Serialización, JDBC, SQLJ, ODMG 3.0 y JDO.

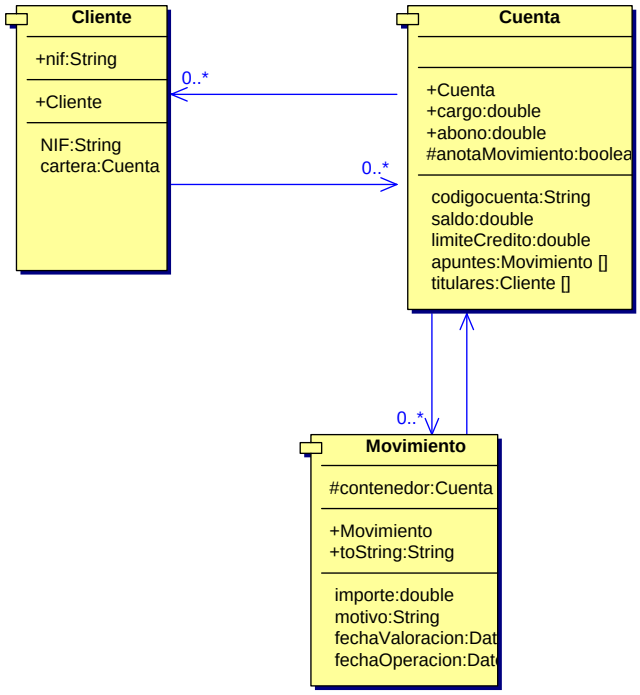


Figura 1 Diagrama de Clases del Banco

Serialización

```
private void persisteMetodoSerialize
                        (Cliente cli) throws Exception {
  try {
    ObjectOutputStream salida = new
                        ObjectOutputStream(new FileOutputStream(
                                    "nuevo_cliente"));
    salida.writeObject(cli);
    salida.close();
  }
  catch (Exception e) {
    e.printStackTrace();
  }
}
```

Listado 1 Serialización de un objeto cliente

Con la serialización el objeto debe ser del tipo `Serializable`, esto es, la clase `cliente` debe implementar los métodos de la interfaz `Serializable` para conseguir la persistencia con Serialización.

En **JDBC**

```
private void persisteMetodoJDBC(Cliente cli) throws Exception {
  Connection con;
  Driver driver = null;
  PreparedStatement pstmt;
  try {
      //…… La connexion ya fue establecida
    pstmt = con.prepareStatement(
        "INSERT INTO Cliente " +
        "(Nif ) VALUES (? )");
    pstmt.setString(1, cli.nif);
    pstmt.execute();
    /*guardar los objetos que dependen del cliente*/
    // Cuentas de la cartera del cliente
    pstmt = con.prepareStatement(
        "INSERT INTO CuentasCliente " +
        "(codigocuenta, saldo, limiteCredito, nif) VALUES (?,?,?,?) ");

    int last = cli.getCartera().length;
    for (int i = 0; i < last; ++i) {
      Cuenta cu = (cli.getCartera())[i];
      pstmt.setString(1, cu.getCodigocuenta());
      pstmt.setDouble(2, cu.getSaldo());
      pstmt.setDouble(3, cu.getLimiteCredito());
      pstmt.setString(4, cli.nif);
      pstmt.execute();
      //Para cada cuenta guardar los apuntes y los titulares
      int mx = cu.getTitulares().length;
      for (int j = 0; j < mx; ++j) {
        // ... grabar titulares
        ..........
      }
      for (int j = 0; j < mx; ++j) {
        // ... grabar apuntes
        ............
      }
    }
    pstmt.close();
    con.commit();
  }
  catch (Exception e) {
    e.printStackTrace();
  }

}
```

Listado 2 Grabar un objeto cliente con JDBC

En listado anterior, el programa descompone el objeto `Cliente` sobre varias tablas, ejecutando una inserción en la tabla Cliente, y grabando cada objeto de las colecciones cartera, titulares y apuntes, en sendas tablas con claves ajenas a la tabla Cliente, mediante el atributo `nif`. El listado muestra numerosos detalles, que este momento, es mejor no profundizar, para no desviar la atención de la idea, principal, que es, tener que convertir en filas, los objetos y sus atributos de forma expresa con el esfuerzo del programador.

En **SQLJ**

```
private void persisteMetodoSQLJ(Cliente cli) {

  try {
    /* conexión
        ....
        */
    #sql {INSERT INTO Cliente(Nif) values( :cli.nif)};

    for (int i = 0; i < last; ++i) {
      Cuenta cu = (cli.getCartera())[i];
      #sql {
          INSERT INTO CuentasCliente
          (codigocuenta, saldo, limiteCredito, nif) VALUES
          ( :(cu.getCodigocuenta()), :(cu.getSaldo()),
           :(cu.getLimiteCredito()),:(cli.nif))};

      //Para cada cuenta guardar los apuntes y los titulares
      int mx = cu.getTitulares().length;
      for (int j = 0; j < mx; ++j) {
        // ... grabar titulares
        ..........
      }
      for (int j = 0; j < mx; ++j) {
        // ... grabar apuntes
        ............
      }
    }
    ........
  }
  catch (SQLException ex) { //Catch SQLExceptions.
     ........
```

Listado 3 Grabar un objeto cliente con SQLJ

En esencia el código JDBC y SQLJ siguen el mismo patrón para realizar la persistencia, la diferencia esencial entre JDBC y SQLJ es sintáctica, SQLJ es más simple y conciso para expresar esta operación, al embeber el código SQL mezclado directamente con las variables Java. SQLJ y JDBC exigen al programador la traducción explícita de los objetos en su representación sobre una base de datos relacional, convertir los objetos en filas sobre las relaciones definidas en el RDBMS, esto significa entonces, que SQLJ y JBDC, no son servicios de persistencia, en el sentido de las definiciones vistas de persistencia.

De otra parte, los OODBMS ofrecen servicios de persistencia, que adoptan una o más de las siguientes aproximaciones de cómo un objeto es hecho persistente [7]:

1. *Por tipo*: Un objeto puede llegar a ser persistente cuando es creado de algún tipo (clase) dotado de la capacidad de persistir o de un subtipo (clase descendiente) de estos. Los tipos persistentes son distintos de los tipos cuyas instancias son transitorias. El tipo podría ser identificado como persistente en su declaración, o por herencia de alguno de los tipos predeterminados por el sistema. De forma parecida, la serialización obliga que las clases implementen la interfaz `Serializable`.

2. *Por invocación explicita:* el programador invoca un método que provoca que un objeto pase a ser persistente. La llamada al método puede ser en la creación del objeto o en cualquier instante, según implementación.

3. Por referencia: un objeto es hecho persistente al ser referenciado por otro que es persistente. Añadir un objeto a una colección persistente, la asignación de un objeto a un atributo de otro persistente o la asignación a cierto tipo de referencias, provocan que un objeto transitorio pase a ser persistente.

En el siguiente listado aparecen resaltados ejemplos que ilustran los tres modos anteriores utilizando dos OODBMS: ObjectStore con sintaxis ODMG 3 .0 y con JDO.

ODMG con OODMS ObjectStore

```
private void persisteMetodoODMGModos(Cliente cli) {

  Database db = Database.open("OODBMSBANCO", ......);
  Transaction tr = new Transaction();
  try {
    tr.begin();
    ListOfObject listaClientes =
                      (ListOfObject) db.lookup("Clientes");
    if (listaClientes == null) {
       // Por Tipo persistente;
      listaClientes = new ListOfObject(db);
      db.bind(listClientes, "Clientes");
    }
    // por Referencia
    listaClientes.add(Cli);
    tr.commit();
    db.close();
  }
  catch (ODMGException e) {
    //.....
```

Por tipo

Por Referencia

Persistir con JDO

```
private void persisteMetodoJDOModos(Cliente cli) {
  PersistenceManagerFactory pmf =
      JDOHelper.getPersistenceManagerFactory(properties);
  PersistenceManager pm = pmf.getPersistenceManager();
  / ....
      Transaction tr = pm.currentTransaction();
  try {
    tr.begin();
    todosLosClientes = pm.getExtent(LisOfObject.class, true);
    // ....
    if (todosLosClientes == null) {
      //por invocación
      pm.makePersistent(listaClientes);
    .......
    }
    //....
    // por Referencia
    listaClientes.add(Cli);
    tr.commit();
    pm.close();
  }
  catch (JDOException e) {
    //.....
```

Por Invocación

Por referencia

Listado 4 Modos de persistir con un OODMBS

Vemos en el anterior listado, que el código en ambos casos presenta gran similitud, no hay código para convertir los objetos a un formato apropiado para la base de datos, no es necesaria una traducción explicita para lograr persistir los objetos. La persistencia parece menos complicada con la serialización, o con las OODBMS, sean o no conformes al estándar ODMG 3.0, que con JDBC o SQLJ. La persistencia afecta a los programas Java en mayor medida con JDBC y SQLJ, que con la funcionalidad aportada por servicios de persistencia similares a los de los ejemplos anteriores, en estos últimos, el código java es

más independiente de la persistencia, más en consonancia con la primera de las definiciones dadas de persistencia.

1.1.4. Persistencia ortogonal

Dos características serán ortogonales, si el uso de una no afecta a la otra, esto es, son independientes entre sí. Programas y persistencia serán ortogonales, si la forma en la que los objetos son manipulados por estos programas, es independiente de la utilización de la persistencia, que los mismos mecanismos operaran tanto sobre objetos persistentes como sobre objetos transitorios, ambas categorías serían tratadas de la misma manera con independencia de su característica de persistencia. Ser persistente debería ser una característica intrínseca del objeto, soportada por la infraestructura del entorno de programación y persistencia. La persistencia de un objeto debe ser ortogonal al uso, tipo e identificación. Esto es, cualquier objeto debería poder existir el tiempo que sea preciso, ser manipulado sin tener en cuenta, si la duración de su vida, supera al proceso que lo creo, y su identificación no estar vinculada al sistema de tipos, como la posibilidad dar nombres a los objetos [2]

Veamos como el denso párrafo anterior, afecta a la interacción con la base de datos, al código de nuestros programas y que beneficios aporta. A la hora de plasmar el uso de la persistencia en nuestros programas, una persistencia ortogonal ideal, llevaría a no tener que modificar el código de nuestras clases, salvo aquellas donde debamos introducir las operaciones que provocan la persistencia para cualquier objeto que sea duradero. Los beneficios que aporta la persistencia ortogonal son importantes: mayores cotas de facilidad de mantenimiento, corrección, continuidad del código y productividad de desarrollo. Se consigue:

1. Menos código. Una semántica para expresar las operaciones de persistencia más simple de usar y entender. Evita la duplicidad de código uno preparado para instancias transitorias y otro para instancias persistentes.
2. Evitar la traducción explicita entre el estado de los objetos y su representación en base de datos, que redunda en mayor facilidad de mantenimiento y menos código también.
3. Facilitar la integridad y permitir que actúe el sistema de tipos subyacente, que automáticamente podría verificar la consistencia y la correspondencia de tipos, entre estados en base de datos y objetos en programa, la integridad no sería responsabilidad del programador.

Todo lo visto en este apartado apunta la conveniencia de usar persistencia ortogonal. En la práctica conseguir persistencia ortogonal completa no es fácil, habitualmente encontraremos limitaciones. Habrá clases de objetos que no son soportadas por los servicios de persistencia, bien por compromisos de diseño, como la dificultad de implementación; bien porque cabe pensar que determinados objetos no tienen sentido fuera del contexto de ejecución concreto de un proceso, como por ejemplo un puerto de comunicaciones para IP. No obstante, en el proyecto Pjama, desarrollado en la Universidad de Glasgow se da soporte a tipos de objetos propios del contexto de ejecución [33]. Para lograr una persistencia lo más ortogonal posible, los servicios de persistencia adoptan distintos enfoques, unos, necesitan que el código de la clases sea modificado, en otros, debemos incluir meta información en forma de marcas, comentarios, ficheros,... para un preprocesador de código fuente u objeto, que realiza los cambios necesarios para conseguir persistencia, y en otros, toda la información necesaria es obtenida en tiempo de ejecución,

presentando un entorno de ejecución extendido con las capacidades de persistencia, donde la funcionalidad consiste en un conjunto de interfaces de programación.

1.1.5. Cierre de persistencia

Los objetos suelen referenciar a otros objetos, estos a su vez a otros, y así puede continuar sucesivamente. Cada objeto puede tener un gran número de objetos dependientes de manera directa e indirecta. Esta relación de dependencias es parte integrante del estado de cada objeto. Cuando el estado de un objeto es salvado o recuperado, sus dependencias también deberían ser guardadas o recuperadas [19]. De otro modo, cuando el objeto fuese recuperado, llegaría a estar incompleto, sería inconsistente con respecto a como fue guardado. En la Figura 1, si una cuenta tiene movimientos, estos deberán persistir cuando la cuenta sea guardada, y viceversa.

Un mecanismo de persistencia que posibilita la persistencia automática de las dependencias de un objeto, que deban persistir, se dice que admite el cierre de persistencia. Cuando el estado de un objeto es almacenado, los estados de los objetos dependientes que tengan que ser persistentes, son también almacenados, y así, sucesivamente. En otro sentido, en la recuperación del estado de un objeto, los estados de los objetos dependientes son recuperados. El cierre de persistencia determina el conjunto de referencias necesario, que ayuda a conseguir la consistencia entre el estado del objeto en el instante de guardar y estado resultante de su recuperación.

1.1.6. Persistencia por alcance

La persistencia por alcance o persistencia en profundidad, es el proceso de convertir automáticamente en persistente, todo objeto referenciado directa o indirectamente por un objeto persistente, los objetos del cierre de persistencia de un objeto son hechos persistentes [34] [35]. Es la aplicación recurrente de la estrategia de persistencia por referencia, vista más atrás.

1.1.7. Transparencia de datos

Cuando un sistema o entorno de programación ofrece transparencia de datos, el conjunto de las clases persistentes y el esquema de la base de datos es uno, las clases definen de hecho el esquema en la base de datos [34] [2] [10]. Los estados almacenados en la base de datos, son manejados con el lenguaje de programación elegido, no es necesario otro. Los objetos son recuperados de la base de datos automáticamente, cuando las referencias a estos son accedidas. También, las modificaciones del estado de objetos persistentes son reflejadas en la base de datos automáticamente. Los estados de los objetos son recuperados y actualizados de forma transparente; no hay cambios en la semántica de referencia o de asignación en la aplicación. Objetos transitorios y persistentes son manipulados de igual forma. Las operaciones propias de la persistencia son efectuadas sin la intervención directa del programador, con más código. La frontera entre el lenguaje de programación y los servicios de datos desaparece a los ojos del programador, evitando la falta de correspondencia (impedance mismatch) entre la base de datos y lenguaje de programación [1].

Veámoslo con un ejemplo que ilustra el concepto, imaginemos una rutina que recorre y actualiza los elementos de una colección. Cuando esta colección es recorrida, el programa no se preocupa de que el siguiente objeto sea recuperado desde la base de datos, si el objeto no esta en memoria, el servicio de persistencia lo obtiene desde la base de datos

para el proceso. La rutina siguiente tal cual está, puede ser aplicada sobre una colección que contenga algunos elementos persistentes o solo temporales, debe ser indiferente.

```
private void rutinaConTransparenciaDatos(Collection cllCuentas){
  final double xlc = 1.10;
  Iterator it = cllCuentas.iterator();
  while (it.hasNext()){
    Cuenta cu = (Cuenta) it.next();
    cu.setLimiteCredito(cu.getLimiteCredito()*xlc);
```

Listado 5 Ejemplo rutina con transparencia de datos.

No debemos confundir transparencia con persistencia ortogonal, la primera es una consecuencia de la segunda. Podemos encontrar transparencia de datos sin disponer de persistencia ortogonal total, es el caso de que determinadas clases no puedan persistir, esto concretamente supone que la solución no sería ortogonal, independiente, respecto el tipo.

1.1.8. Falta de correspondencia entre clases y datos

Cuando trabajamos con sistemas gestores de datos, como bases de datos relacionales, ficheros, bases de datos documentales XML, etc., cuyo modelo datos no tiene una equivalencia directa con el modelo de objetos del lenguaje de programación usado, hay una falta de correspondencia (impedance mismatch) entre la base de datos y lenguaje de programación, es necesario establecer un modelo de correspondencia, que defina como un clase se convierte en datos sobre el modelo ofrecido por sistema que albergará el estado de los objetos. A esta equivalencia entre la clase y los datos, se denomina aquí *correspondencia clase-datos*(object mapping). El caso particular de la correspondencia entre clases y tablas en un RDBMS, es la *correspondencia objeto-registros*. Utilizaremos el término mapear, para señalar al proceso de definición de la correspondencia entre las clases de los objetos persistentes y los modelos de datos, el proceso puede implicar cambios en ambos lados de la correspondencia, en el modelo de clases creado o modificado para asumir ciertos modelos de datos, y al contrario, el modelo de datos puede ser diseñado o cambiado, para permitir una correspondencia más eficiente y eficaz con ciertos modelos de clases.

Hemos visto una breve relación de los conceptos más relevantes concernientes a la cuestión de la persistencia. Las definiciones de persistencia vistas nos invitan a emplear una persistencia que no cambie la forma de trabajar con el lenguaje de programación, que actué de forma transparente y consistente, donde el modelo de clases y el modelo de datos son la misma cosa, y que sea una persistencia ortogonal. Los beneficios serán un código con menos líneas, más fácil de mantener, más correcto y productivo.

1.2. Requisitos para un servicio de persistencia

Intentaremos ahora establecer cuales son las funcionalidades a soportar por un servicio de persistencia, teniendo en cuenta los aspectos técnicos, los organizativos y económicos. Dependiendo del enfoque que sea considerado, estas funcionalidades cambian, tendrán mayor o menor importancia, y algunas de ellas serán incompatibles. Si el punto de vista escogido es el desarrollo de programas rápido, nos interesará un servicio simple, transparente y sencillo de usar, que cubra los requisitos esenciales; si nuestro interés es conseguir un elevado número de transacciones, o manejar grandes volúmenes de información de bases de datos relaciones, la correspondencia entre clases y relaciones, el rendimiento, la jerarquía de memoria (caches) y la gestión del almacenamiento son las cuestiones críticas; en otro área, el diseñador de componentes, muy posiblemente, estaría

dispuesto a penalizar el rendimiento, si con ello consigue una mejor integración con el marco (framework) de componentes, o una mayor uniformidad en el tratamiento de distintas fuentes de datos; un gestor o directivo le importa más la tasa de retorno de la inversión, la productividad, la disponibilidad de un mayor número de proveedores, de profesionales y menores costes de formación, etc. Como podemos imaginar, establecer una solución valida para todos los enfoques es tarea difícil.

Los requisitos serán considerados sobre tres dimensiones: tecnológica, organizacional y económica. La dimensión tecnológica recoge los mecanismos que debería cubrir un servicio de persistencia. Respecto a la organización nos interesa conocer el impacto provocado por la adopción de un servicio de persistencia como es JDO. El factor económico es esencial, pues toda tecnología no puede sustentarse o evolucionar sin el respaldo de un mercado sólido [31].

Para acometer la tarea, nos hemos fijado en los trabajos incluidos en las bibliografía de autores reconocidos como S. W. Ambler, M. Atkinson, K. Dittrich, Craig Larman, Arthur M. Keller, Bertino, C. Szypersky, T. Reenskaug, G. Booch, J. Rumbaugh, Ivar Jacobson, Claus P. Priese y Wolfang Keller, Robert Orfali, David Jordan, cuyas referencias aparecen a más adelante en el texto.

1.2.1. Dimensión tecnológica. Requisitos estructurales y funcionales

Queremos determinar qué mecanismos debe proporcionar un buen sistema de persistencia que asuma los enfoques antes expuestos. Trataremos de conseguir especificar las funcionalidades de un servicio de persistencia con transparencia de datos, capacidad para guardar y recuperar concurrentemente desde diferentes sistemas gestores de datos, que sea ortogonal en la mayor medida posible, robusto, consistente, eficaz, fácil de usar y extensible. Cada funcionalidad va a ser considerada respecto a su influencia en el rendimiento, mantenimiento y coste de la solución. En primer lugar son consideradas las cualidades estructurales que debería tener un servicio de persistencia. Para seguidamente, introducir la funcionalidad que se espera acometa un servicio de persistencia, tal y como ha sido descrito antes.

La ilustración de ejemplos y situaciones, es utilizada para poner de manifiesto los requisitos para obtener la funcionalidad esperada. Ejemplos que cubrirán unos requisitos y servirán para mostrar otros. Escenarios cuya descripción permitirá presentar distintas cuestiones a partir de las cuales será expuesta la funcionalidad.

Como no todos los requisitos técnicos son cubiertos en la práctica, las prestaciones del servicio serán clasificadas en esenciales u opcionales de acuerdo al nivel de exigencia de su soporte. Exigencia determinada, en primer lugar, a la necesidad o conveniencia técnica según la opinión de distintos autores y, en segundo termino, respecto del coste y beneficio aportado al rendimiento y la mantenibilidad. Habrá también requisitos que son soportados directamente por los mecanismos subyacentes, y que por tanto, no necesitan ser plasmados a nivel del servicio de persistencia, y otros que son el reflejo de los presentes en los mecanismos de persistencia utilizados.

El documento de Scott W. Ambler "Design of a Robust Persistence Layer For Relational Databases"[32], es el fundamento y guión inicial de los siguientes apartados. Base que es ampliada para considerar un punto de vista más general, incorporando aportaciones de los autores mencionados.

1.2.1.1. Estructurales

Los requisitos estructurales, son las cualidades y capacidades elegidas como referente para la construcción de la funcionalidad. El aspecto estructural determina en buena medida como es conseguida la funcionalidad, cual será la continuidad, la facilidad de uso, la capacidad de integración, la eficacia y la eficiencia, el coste,... factores todos ellos que determinan la calidad de las soluciones implementadas.

¿Cuáles son las propiedades estructurales que debe tener todo servicio de persistencia? ¿Cuál es la estructura más indicada? ¿Cuáles son las bases sobre la que construir la funcionalidad?

Pretender contestar directamente de la forma adecuada es una osadía. El planteamiento aquí, es asumir las pautas y recomendaciones marcadas por quienes son considerados verdaderas autoridades de la disciplina del diseño de programas, primordialmente, dentro de la construcción de programas orientados a objetos. Pero transcribir aquí los criterios, principios y reglas de carácter general ampliamente aceptados, no es suficiente, al menos, los principales aspectos deben ser concretados. Evidentemente, que un servicio de persistencia debería ser diseñado de una forma disciplinada, rigurosa, consistente, correcta, robusta, eficaz, extensible, reutilizable, eficiente, transportable, funcional, oportuna, compatible y manejable [19].

Pasemos ahora a intentar concretar algunas de las cualidades o propiedades, que deben satisfacer un buen servicio de persistencia.

Simplicidad y sencillez

Un requisito primordial para lograr una buena arquitectura es intentar mantener la simplicidad y la sencillez de la solución, evitar los artificios innecesarios. No significa esto, que la solución no deba cubrir la complejidad de la tarea encomendada, pero el planteamiento, diseño, implementación y documentación no deben parecer cuencos de espaguetis. Debería ser tomada en consideración la siguiente máxima de Hoare:

> *"Hay dos formas de construir un diseño de software:*
> *Una forma es hacerlo tan simple que obviamente no hay deficiencias*
> *Y la otra forma es hacerlo tan complicado que no hay deficiencias obvias."*
> *—C.A.R. Hoare*

Fiabilidad

Un servicio de persistencia debería mostrar una alta probabilidad de funcionar correctamente bajo toda circunstancia. La fiabilidad o confiabilidad es uno de los fundamentos básicos sobre los que se sustenta toda solución estructurada. La confiabilidad va a depender, en gran medida, de las propiedades estructurales como el intercambio de datos, la señalización y tratamiento de condiciones de error, la gestión de excepciones, iniciación de variables, etcétera. Todo debe ser cuidadosamente pensado y llevado a cabo.

Un servicio de persistencia debe ser fiable, funcionar correctamente evitando resultados no esperados o imprevisibles. La robustez es uno de los objetivos iniciales marcados, que debe ser conseguido gracias a un diseño que contemple el tratamiento y reacción ante situaciones anormales. Más adelante veremos que funcionalidad se requiere.

Modularidad

La solución a un problema complejo pasa por establecer una arquitectura modular, que estructura el problema en distintas partes, cuyas soluciones contribuyen a la solución

completa del problema inicial. Esto es, la solución a un problema complejo, pasa por la creación de módulos que ensamblados definen el sistema que plasma la solución.

Una buena modularidad facilita la consecución de tres propiedades esenciales a toda solución de programación: la ocultación de la información, la facilidad de mantenimiento y el desarrollo en paralelo de las partes. Debemos recordar que toda construcción orientada objetos debe seguir una arquitectura guiada por los tipos de los objetos que manipula [7]. Asimismo, que la separación de conceptos frente a su coste de realización, ayuda a concretar la división en módulos de la funcionalidad 0.

Algunas de las propuestas revisadas [41], [42],[14],[6],[24], [32] y [4], difieren en los componentes concretos que forman la arquitectura pero desde un punto de vista más general coinciden en:

- Estructurar las responsabilidades en capas de servicios.
- La distinción clara de dos capas una objetual y otra de gestión del almacenamiento.

Figura 2 Puzzle de la persistencia

La estructura general en capas, facilita la organización de los servicios en distintos niveles, cada uno de los cuales constituyen el soporte para los servicios del nivel superior y es sustentado, a su vez, por los servicios que necesita del nivel inferior, consiguiendo una alta cohesión, bajo acoplamiento y facilidad para la encapsulación. Ejemplos clásicos son las pila de servicios para red OSI y TCP/IP. Para estudiar más detenidamente esta forma de estructurar los servicios de persistencia, para el caso particular de las bases de datos relacionales ver [12] [6] [32].

La separación en una capa de objetos y otra para administrar el almacenamiento, contribuye a la separación de los conceptos de la orientación objetos, de aquellos relativos a la programación de bases de datos, sistemas de archivos o servicios de empresa. Cada una estas áreas está representada por abstracciones bien conocidas, con patrones de solución bien implantados dentro sus dominios de aplicación. De esta forma, podrán ser aplicadas las mejores soluciones a cada abstracción. Entre ambas capas, se sitúan uno o más niveles de servicios que cubren parte de la funcionalidad requerida y la falta de correspondencia, entre los mundos de los mecanismos de persistencia y el mundo de los objetos. El coste de separar en capas un servicio de persistencia debe ser recuperado por un aumento en la mantenibilidad y una más fácil puesta a punto del rendimiento [32] .

Encapsulación

Scott W. Ambler [32] propone que idealmente solo se deberían mandar los mensajes de *salvar*, *borrar* y *recuperar* al objeto en cuestión ocultándose todo detalle sobre el

mecanismo concreto de persistencia. En [42] de Wolfgang Keller podemos encontrar también, la justificación sobre la conveniencia de que un servicio de persistencia debe encapsular los detalles de los mecanismos de persistencia subyacentes.

Así pues, alcanzar la encapsulación de los mecanismos de persistencia es un requisito esencial. La estructuración en capas facilita lograr el requisito de la ocultación de la información.

Diferentes Estilos de Aplicación

Veamos a un ejemplo que servirá como ilustración de nuevos requisitos, una secuencia simple de lo que sucede cuando vamos a pagar la compra del supermercado con una tarjeta de pago electrónico...

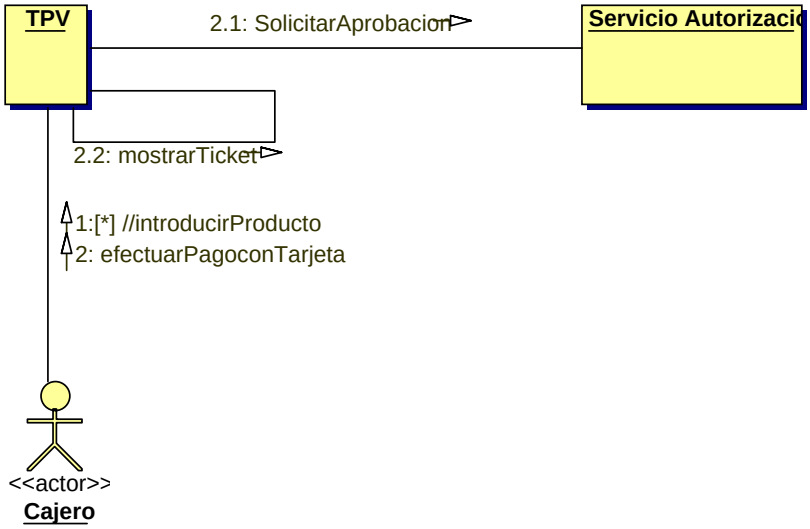


Figura 3 Diagrama de Colaboración Venta TPV

La secuencia muestra la interacción entre tres objetos: el cajero, un tpv y un servicio de autorización bancaria. Un cajero debidamente acreditado ante el TPV que realiza las transacciones de anotar los productos que nos llevaremos a casa, la operación del pago con tarjeta y finalizar la venta con la entrega del ticket, etc. El TPV, que instancia el sistema de ventas, tiene por misión registrar la salida de artículos, los registros de actividad sobre operaciones del cajero, preparar la cuenta y acceder al servicio de autorizaciones de pago con tarjeta. El tercero, el servicio de autorización de pago, accede a los sistemas de banca electrónica para conceder o denegar la operación de pago.

Evidentemente, tanto el TPV como los Servicios de Autorización, necesitan registrar los datos procesados, podemos imaginar que la plataformas tecnológicas y los mecanismos de persistencia serán muy distintos. Un TPV podría ser desde un terminal clásico a un ordenador personal adaptado a tal propósito con bajas prestaciones de cómputo. Los servicios de autorización son servicios críticos que requieren de enormes prestaciones para atender un elevado número de transacciones por minuto, típicamente soportados sobre grandes equipos centrales.

Sobre la base de este ejemplo, y echando un vistazo a la literatura sobre las bases de datos, podemos convenir varios tipos de sistemas de información, con diferentes necesidades de mecanismos para guardar, recuperar y concurrir sobre los datos. Aquí, serán definidas dos categorías. De un lado, las aplicaciones de gestión con diferentes necesidades de prestaciones y rendimiento, cubiertas por sistemas basados en archivos y en gestores relacionales. En otro, las aplicaciones de CAD, CASE, GIS, OLAP,... donde los

sistemas gestores de bases de datos relacionales clásicos no han cubierto sus requisitos de persistencia en términos de rendimiento, mantenimiento y coste [41] [42] [32] [7] [28] [25] [8].

Es requisito esencial soportar, las aplicaciones de la primera categoría, las aplicaciones de los negocios, que representan la mayoría de las aplicaciones en uso. La otra categoría es considerada como un requisito opcional.

Diferentes arquitecturas

Las funcionalidades de una aplicación deben ser organizadas siguiendo algún modelo de arquitectura, que, cuando la aplicación es ejecutada, determina la interacción entre sus funcionalidades y donde son desplegadas en el entorno de ejecución. En la actualidad coexisten distintas arquitecturas de aplicaciones que han sido ampliamente aceptadas, y otras nuevas entran en escena como *Model Driven Architecture* (MDA) o los modelos basados en .NET. Todas ellas pueden ser clasificadas de forma genérica en arquitecturas monolíticas, las cliente/servidor en dos o más capas y las distribuidas de objetos. En todas tendremos la necesidad de la persistencia de datos. Arquitecturas diferentes, con necesidades de persistencia distintas, que conducen a que es necesario, soportar diferentes arquitecturas. Las opiniones de los trabajos vistos en este trabajo de S. Ambler [32], T.Reenskaug [25] , C. Priese [24], y B.Meyer [19], recogen esta necesidad de diferentes arquitecturas

Es necesario que un buen servicio de persistencia ofrezca soporte a los modelos clásicos de arquitectura monolítico, cliente/servidor en dos o más capas y distribuidas de objetos.

Esto supone para el diseñador de servicios de persistencia seguir ciertas especificaciones que garanticen la integración del servicio de persistencia en las arquitecturas objetivo. Soportar diferentes arquitecturas implica un mayor coste de desarrollo, por lo que en la práctica la mayor parte de los productos eligen un modelo de arquitectura con vistas a satisfacer un determinado sector del mercado, y delegan en el desarrollador la posible adaptación al modelo de arquitectura escogida, lo cual no siempre es posible. Quizás fuera más adecuado expresar este requisito en forma negativa como:

Es esencial que un servicio de persistencia no deba imponer restricciones estructurales que impidan su inclusión y/o adaptación a diferentes arquitecturas.

Un ejemplo de los inconvenientes de no asumir el requisito expuesto, es el uso de código no reentrante que evitaría mantener en ejecución más de una imagen de un mismo ejecutable o librería. Hace ya unos años, programando aplicaciones de gestión en un entorno gráfico muy conocido, las librerías proporcionadas para acceder a una base de datos archíconocida, no permitían aprovechar la capacidad de ejecución concurrente en hebras del entorno gráfico, la ejecución de consultas con múltiples registros no podía ser solapada con la presentación y edición en distintas aplicaciones en el mismo sistema, peor aún, lanzada una consulta esta no podía ser interrumpida, a riesgo de hundir todo el sistema. Esta limitación forzaba el diseño de la aplicación que no era escalable frente al número de registros o la carga.

Extensibilidad

Al igual que deberíamos ser capaces de añadir nuevas clases a las aplicaciones, deberíamos ser capaces de sustituir los mecanismos de persistencia empleados. También tendría que ser posible incorporar nuevos mecanismos. De hecho con una modularidad adecuada debería ser posible extender el servicio de persistencia con nuevos mecanismos. Que un servicio de persistencia sea extensible o no, es una decisión de diseño motivo de

controversia. De un lado los programadores de los servicios de persistencia querrán disponer de la posibilidad adaptar el servicio a sus necesidades dado el caso; de otro los fabricantes, celosos guardianes del código de sus implementaciones. La orientación objetos permite satisfacer a ambas partes. Debemos tener en cuenta que los programas evolucionan, incorporando o modificando su funcionalidad, en este sentido sería preciso dotar de extensibilidad también a los servicios de persistencia. *La extensibilidad es una cualidad deseable de un buen sistema de persistencia.*

Facilidad de uso

La facilidad de manejo es otra de las propiedades que se espera tenga todo servicio de persistencia. En los apartados anteriores queda de manifiesto la conveniencia, de que un servicio de persistencia debe presentar sus servicios de forma clara, concisa, oportuna y manejable. Ser fácil de usar contribuye a conseguir la economía de los desarrollos, mayor productividad y menor coste en horas/hombre. Esta cualidad garantiza la aceptación y el éxito de implantación de un servicio de persistencia. Cosas tales como la nomenclatura consistente de las características, la flexibilidad y expresividad en la definición de las operaciones, la asignación de valores por defecto, el número de servicios disponible, la compatibilidad con otros tipos de objetos, el empleo de asistentes para automatizar las principales tareas y el empaquetamiento y distribución, son elementos que ayudan a conseguir un servicio más fácil de usar. La facilidad de uso debería repercutir en:

- Un menor esfuerzo de diseño.
- Menor coste y esfuerzo de producción del código.
- Facilidad para la prueba.
- Mejor verificabilidad.
- Mejor mantenibilidad.
- Facilidad de instalación y configuración.

Todos estos factores deberán ser tenidos en cuenta, aún cuando su medida no siempre es fácil. El sentido común será la mejor herramienta para decidir si es sencillo o complicado el uso de un servicio de persistencia. La facilidad de uso puedo llegar a ser un factor crítico determinante en el éxito de un buen servicio de persistencia. Un servicio puede ser el más eficaz y eficiente, el más barato pero si no es fácil de usar, tendrá severas dificultades alcanzar ser aceptado e implantado en las organizaciones.

Escalabilidad y rendimiento

La arquitectura de una solución, afecta a su capacidad para acometer aumentos importantes en el volumen de datos o peticiones de servicio, con un rendimiento y productividad acordes. La escalabilidad es uno de los objetivos iniciales que no es alcanzable, sin un diseño preparado frente a grandes variaciones en la carga. La escalabilidad es una cualidad necesaria para un servicio de persistencia, porque al ser un intermediario entre los objetos y los servicios de datos, se convierte en un factor crítico que puede llegar a ser el cuello de botella de todo un sistema.

Se espera que un servicio de persistencia se adapte a las fluctuaciones en el volumen de datos y peticiones ofreciendo el servicio más eficaz en cada caso. Esto es, un servicio de persistencia debería ser escalable.

Para mejorar el rendimiento, el acceso a los objetos debe ser optimizado y también el acceso a los mecanismos de persistencia, para lo cual, se debe maximizar el rendimiento

de los mecanismos y minimizar el número de peticiones a los mismos [41]. Esto lleva a introducir en la arquitectura del servicio elementos como:

- Caches de datos y objetos.
- Encolamiento de operaciones.
- Procesamiento asíncrono de comunicaciones y operaciones con persistencia.
- Mecanismos para optimizar la concurrencia.

Cada uno de estos elementos, le corresponderá una funcionalidad que no corresponde con el puro objetivo de guardar y recuperar datos y objetos, pero sin los cuáles la solución al problema de la persistencia no sería de utilidad. Las propuestas de [41] [32] incorporan la necesidad de optimizar el rendimiento de los servicios de persistencia.

Modelos de implementación

La implementación de un servicio de persistencia puede ser efectuada de distintas formas, cada una de las cuales presenta sus ventajas e inconvenientes, tiene sus partidarios y sus disidentes. Encontraremos en el mercado, un abanico de realizaciones, que van desde las clásicas colecciones de clases empaquetadas como fuentes o binarios, a sofisticados framework, dotados con programas asistente para cada actividad del proceso de creación, mantenimiento y ejecución.

Cada una de las opciones que podremos encontrar, seguramente, será la más adecuada a cada caso, en función de los requisitos, la cultura de desarrollo y las disponibilidades presupuestarias. A la hora de escoger, deberemos tener muy presente la flexibilidad para poder amoldar la implementación del servicio de persistencia a nuestras necesidades.

Idealmente, la persistencia ortogonal podría ser una característica del lenguaje de programación, su sintaxis y semántica serían extendidas para incluirla, incluyendo al efecto, nuevas instrucciones y modificadores en declaración de los tipos, atributos y objetos. Pero, el mercado parece que tiende a las opciones enlatadas, siguiendo los modelos clásicos de colecciones de servicios y framework. Todas las realizaciones, que encontraremos, presentan en común dos aspectos principales:

- La utilización de meta datos. Meta información acerca de qué será persistente y cual será la correspondencia clase-tipo. La meta información estará incluida en el código fuente de la clases o en repositorios a parte de las fuentes, incluidos los utilizados para la persistencia. Los repositorios son ficheros comunes de texto, ficheros XML e incluso sofisticados modelos de objetos-datos, almacenados en bases de datos.
- La inclusión automática de código en las clases cuyos objetos serán persistentes, y en los métodos de las clases cuyas instancias conducen las operaciones de persistencia. El código añadido podrá ser en el código fuente, en el código ejecutable, o una mezcla de ambos.

Pasemos ahora a estudiar los aspectos funcionales.

1.2.1.2. Funcionalidad

Guardar y recuperar objetos

Este es el requisito esencial por excelencia. Partimos de un sencillo ejemplo para ayudar a descubrir nuevas funcionalidades requeridas. El ejemplo es la continuación de aquel visto donde se guardaba un cliente.

Sin otra restricción de partida, lo más inmediato es aprovechar lo que tenemos más a mano para hacer persistir los objetos, usar la interfaz `Serializable` que recoge sólo la semántica de persistencia y que no obliga, en principio, a implementar método alguno. La clase, cuyas instancias deseamos hacer persistentes, tiene que ser declarada como `Serializable`. Para guardar un objeto debemos usar el método `writeObject` sobre un objeto *`OutputStream`*, y para recuperar usar el método `readObject` aplicado a un objeto `InputStream`.

```
package casosdeuso;

import java.io.*;
import banco.*;

public class TestRWSerialize
{
  public static void main(String[] args)
  {
    TestRWSerialize ts = new TestRWSerialize();
    Cliente cliAGuardar = new Cliente();
    Cliente cliRecuperado = null;

    try {
      final String idCli = "A-30000";
      cliAGuardar.setNIF(idCli);
      ts.persisteMetodoSerialize(cliAGuardar, idCli);
      cliRecuperado = ts.recuperaMetodoSerialize(idCli);

System.out.println("Nif cliente guardado " +
                         cliAGuardar.getNIF());
System.out.println("Nif cliente recupeado " +
                         cliRecuperado.getNIF());

      if (cliRecuperado.getNIF().equals(cliAGuardar.getNIF())) {
        System.out.println("Objetos con los mismos contenidos");
      }
      if (!cliRecuperado.equals(cliAGuardar)) {
        System.out.println(
            "Objetos internamente distintos, Identidad diferente");
      }
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }

  private void persisteMetodoSerialize(Cliente cli, String idExterno)
      throws Exception
  {
    try {
      ObjectOutputStream salida = new ObjectOutputStream(new
                                  FileOutputStream(idExterno));
      salida.writeObject(cli);
      salida.close();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
…
  }
}
```

Identificador
Conecta objeto y estado guardado

Instancias con igual contenido pero distintas.

```
Ejecución
Nif cliente guardado A-30000
```

```
Nif cliente guardado A-30000
Objetos con los mismos contenidos
Objetos internamente distintos, Identidad diferente
```

Listado 6 Ejemplo serialización en Java

El código anterior guarda y recupera un objeto de la clase `Cliente`, que fue declarada `Serializable`, objeto que es hecho persistente con el método `persisteMetodoSerialize`, que escribe el objeto sobre un archivo donde queda almacenado el estado del objeto persistente. Seguidamente, el objeto es recuperado con el método recuperaMetodoSerialize desde el mismo archivo donde fue guardado. La salida estándar muestra como ambos objetos son distintos aunque con los mismos contenidos, estados iguales.

Revisando el código del listado anterior se plantean distintas cuestiones:

- La necesidad de unos identificadores, que conectan los estados almacenados y los objetos en ejecución. En listado anterior era el nombre del archivo donde es almacenado el objeto.
- El objeto recuperado no es el mismo en sentido estricto, tiene el mismo estado, el mismo contenido, es otro objeto con los mismos contenidos. Los objetos iguales, pero no idénticos
- Pueden producirse excepciones al interactuar con el mecanismo de persistencia que deben ser manejadas.
- No basta solo con guardar y recuperar. ¿Qué sucede cuando un objeto persistente es modificado o eliminado? El estado almacenado debe reflejar los cambios producidos. El archivo en ejemplo anterior debería ser actualizado o suprimido, según el caso.

Operaciones básicas de actualización del estado

Guardar y recuperar el estado de los objetos, significa que todo objeto persistente necesitará, que su estado sea creado, leído, actualizado y borrado del medio, soporte de información o sistema donde sea depositado. *Es esencial que un sistema de persistencia facilite las operaciones básicas de creación, lectura, actualización y borrado de los estados de los objetos persistentes* [32] [4].

Identidad de los objetos persistentes

Necesitamos asociar al menos un identificador a cada objeto persistente, que sirve para localizar, de forma inequívoca, la imagen del objeto guardado en los repositorios de datos. Cada identificador establece una correspondencia entre el objeto persistente y su estado almacenado, una referencia con la que es posible efectuar las operaciones de actualización para reflejar los cambios efectuados en los objetos durante la ejecución; en base a estos identificadores es posible asignar una identidad para los objetos persistentes. La elección del sistema de identidad empleado para los objetos persistentes, es uno de los elementos clave de la definición de la correspondencia clase-objetos.

Podemos encontrar modelos de persistencia, en los que la identificación es explicita, es obligatorio añadir o designar un o más atributos de la clase, como la identidad del estado depositado en el sistema gestor de datos; en otros, esto es una opción, pues disponen de la capacidad de generar automáticamente los identificadores. La elección de que técnicas emplear depende de los entornos de persistencia, en conjunción de los gestores destino, ya sean bases de datos relacionales u otros. Hay OODBMS que permiten asignar a un atributo, o varios, la característica de unicidad, que sirve como identidad, pero

distinta además de la de identidad asignada por el sistema, lo que es una clave primaria en SGBDR. Las principales bases de datos relacionales, disponen también de una identidad generada automáticamente para identificar a los registros, que es independiente y distinta de la designación de clave primaria, los identificadores de fila, ambos sirven para establecer la identidad de una fila de datos. Los sistemas de archivos también presentan modelos de identificación que establecen como nombrar de modo singular y localizar los archivos que manejan.

Los identificadores podrán tener o no significado para el usuario, pero es útil poder dar un nombre a los objetos persistentes, mediante el cual buscar y localizar los objetos almacenados en los sistemas gestores de datos, nombrar los objetos es para nosotros un modo natural de identificar. Estándares como ODMG 3.0, CORBA naming services y Java JNI, ofrecen esta posibilidad aunque con perspectivas distintas.

Es un requisito esencial que todo servicio de persistencia soporte identificadores de objetos utilizados para guardar, y acceder posteriormente, a los estados de los objetos persistentes. Opcionalmente debería ser posible emplear nombres, identificadores asignados por el programador o usuario, para la localización de objetos persistentes, para su identificación.

Hemos visto que los mecanismos de persistencia ofrecen fórmulas diferentes, para establecer la identidad de los datos, los servicios de persistencia deben aprovechar la capacidad de los mecanismos para establecer y asegurar la identidad. En concreto, con los sistemas relacionales, *es preciso facilitar la integración de claves primarias e identidad de los objetos persistentes*.

La posibilidad de utilizar identificadores asignados por los usuarios o programadores, lleva a que se debería ofrecer a las aplicaciones un sistema de identidad o claves, independiente de los mecanismos de persistencia soportados, esto aporta flexibilidad frente a cambios del mecanismo empleado. U*n servicio de persistencia debería ofrecer los medios para definir y manejar modelos de identidad basados en*:

- *La identidad asignada por los mecanismos de persistencia.*
- *Identidad asignada mediante la aplicación bien con nombres* [2], *bien con claves primarias [41].*

Los trabajos de G.Booch, J. Rumbaugh, A. Keller, y S. Ambler enseñan distintos modelos de soporte de las claves primarias para sistemas relacionales. En [7] y [4] podemos llegar a conocer algunos de los modelos usados por los sistemas basados en objetos.

Volvamos sobre el ejemplo anterior. Sabemos que `Serializable`, asegura una persistencia por alcance, todos los objetos relacionados directamente o indirectamente, con el objeto a almacenar o recuperar son automáticamente procesados. Así pues, bastará con designar el identificador del objeto singular raíz del cierre de persistencia, para alcanzar el objetivo de guardar y recuperar objetos. La serialización empaqueta el cierre de persistencia de un objeto en un solo depósito, todo contenido en una unidad que no permite acceder a los objetos contenidos por separado. Si deseamos acceder por separado a los objetos contenidos, cada uno de los objetos, debe ser guardado de forma independiente, asignarle una identidad persistente, y de alguna forma mantener el vínculo de las referencias internas existentes entre estos objetos y los correspondientes identificadores asignados. En definitiva, la equivalencia entre referencias internas en ejecución y la identidad persistente, debe ser manejada por el servicio de persistencia, de forma que se mantenga la integridad de las relaciones entre objetos persistentes. Los estados almacenados están relacionados entre sí, como los objetos a los que corresponden, el cierre de persistencia queda también almacenado como una relación entre estados de los objetos.

La traducción de referencias en tiempo de ejecución a la identidad persistente, y viceversa, debe ser efectuada de forma consistente por el servicio de persistencia.

Correspondencia clases-datos

Al guardar y recuperar los objetos, existe una falta de correspondencia entre tipos de los objetos y los tipos de los datos en los almacenes de persistencia, excepción hecha de los gestores de datos objetuales, que debe ser resulta al hacer persistir los objetos en un servicio de datos, para ello hay identificar qué clases serán persistentes, esto es, definir la correspondencia de clases con las entidades de los servicios de datos. La correspondencia permite al servicio de persistencia llevar a cabo intercambio entre ambos sistemas de tipos con integridad. [1] [2]

Un servicio de persistencia debería proporcionar los medios para expresar y establecer la correspondencia entre el modelo de clases y los esquemas de los servicios de datos. Las diferencias entre los sistemas de tipos del lenguaje y los mecanismos de persistencia, exigen la definición de las de equivalencias entre estos, tanto para los tipos básicos o por defecto, como para los tipos complejos como son las estructuras de datos.

Un servicio de persistencia con transparencia de datos debería producir la conversión automática entre los tipos o clases y sus correspondientes tipos, que definen su estado persistente en los depósitos de datos. La conversión debería ser efectuada sin necesidad de añadir código adicional, sin la intervención del programador.

La correspondencia debe cubrir el caso de las clases son definidas a partir de otras de las que dependen, resultando así que se definen objetos complejos, objetos construidos con otros objetos, además de los tipos básicos como enteros, carácter y demás. *Un servicio de persistencia debe soportar objetos complejos* [2] .

Entre los objetos complejos están el conjunto, la tupla y la lista que forman un conjunto de constructores básicos esenciales. Los conjuntos son críticos pues son la forma natural de representar colecciones de objetos del mundo real, las tuplas son la forma natural representar las propiedades de un concepto u objeto y las listas o series (array) capturan el orden [2]. Un *servicio de persistencia debería proporcionar soporte a los tipos estructurados que implementan el conjunto, tupla y lista. Generalizando, un servicio de persistencia debería soportar colecciones persistentes.*

Las relaciones entre objetos también tienen que ser convertidas sobre los datos, así que la herencia, la agregación, la composición, la simple asociación y el polimorfismo deben ser representados con datos usando los elementos proporcionados por los mecanismos de persistencia para perdurar de forma eficaz y eficiente, la tarea no siempre es fácil. Baste de momento reconocer que *un servicio de persistencia de facilitar los medios para expresar y plasmar las relaciones entre los objetos en términos de los mecanismos de persistencia que permite*.

En particular, el mapeo con las bases de datos relacionales es objeto de numerosos trabajos y esfuerzos, entre otros mencionar las propuestas de J. Rumbaugh [30], C. Larman [18], A. Keller [41], W. Keller [42], S. Ambler [32], o R.G.G.Cattell [7] consideran la problemática del mapeo sobre bases de datos relacionales.

La correspondencia clases-datos, *la información sobre el mapeo, debe ser accesible en tiempo de ejecución para asegurar la integridad* con el conocimiento que se obtiene de la correspondencia.

Ortogonalidad

En las definiciones y conceptos presentados sobre persistencia, y en la introducción de esta sección, conducen a considerar la conveniencia de alcanzar la mayor independencia posible. Desde el punto de vista de programador, esto implica conseguir:

- No necesitar modificar el código fuente de las clases y métodos.
- Independencia de uso, los métodos manipulan objetos persistentes y transitorios usando el mismo código.
- Transparencia de datos, esquema es único.
- Poder utilizar claves primarias, asignar nombres o identificadores para denominar, localizar y recuperar objetos persistentes.

En ninguno de los lenguajes de programación más habituales como C, C++, VBasic, Delphi, Eiffel, es posible conseguir una persistencia ortogonal total. De alguna forma el código fuente o el código objeto, debe ser modificado, además de limitar los tipos de datos que pueden persistir. En Smalltalk, encontramos algún entorno como GemStone, donde la maquina virtual Smalltalk es extendida con ciertas extensiones sintácticas para señalar atributos transitorios y claves primarias, etc.; y con la modificación del papel de algunos objetos, que se convierten en el medio para alcanzar persistencia, como las variables globales, o las colecciones persistentes, consiguiendo así una solución sin fisuras entre lenguaje y base de datos de objetos, dotada una ortogonalidad importante

En Java, hay tipos de objetos que no pueden ser persistentes con la técnica descrita de la serialización, como por ejemplo, las instancias de `java.lang.threads`. Ciertamente, la mayor parte de las aplicaciones no necesitan guardar y recuperar un thread o una variable semáforo, son objetos que no tienen sentido fuera del proceso que los usa. Hay trabajos como el proyecto Pjama [33] que pretenden alcanzar una ortogonalidad completa. El estándar ODMG 3.0, tampoco especifica una persistencia completamente ortogonal, como la definición mencionada al principio. Revisando algunos de los productos comerciales a los que se ha tenido acceso y alguna propuesta como UFO [24], vemos que en Java es necesario heredar o implementar determinados interfaces y patrones de código. Uno de los productos de Versant, Enjin, consigue evitar modificar el fuente implementando un procesador de código ejecutable que añade la capacidad de persistencia al código objeto ejecutable, un sencillo, y fácil de usar archivo de configuración, es utilizado para indicar que va a ser persistente.

La independencia completa de la persistencia frente a Java, persistencia ortogonal al tipo, uso e identificación, es costosa y compleja de implementar, podría a llegar a necesitar del cambio en la especificación de Java como lenguaje. Esto podría suponer un gran revés a la amplia aceptación e implantación de Java. Pero sin elevada independencia no hay transparencia de datos, que es uno de los objetivos a conseguir.

Adaptar el código y asumir alguna limitación, es la fórmula habitual para obtener persistencia bastante independiente respecto de los lenguajes de los programas y de los sistemas de persistencia. Que pretendamos conseguir persistencia ortogonal, no debe significar adoptar una postura estricta y purista, sino de compromiso de diseño; ninguna base de datos relacional, cumplía todos las reglas de Codd [8] y no por ello se abandonó su uso. En resumidas cuentas, *un servicio de persistencia debe ser lo suficientemente ortogonal como para conseguir:*

1. *No necesitar modificar el código fuente.*
2. *Independencia de uso.*

3. *Transparencia de datos.*
4. *Utilizar claves primarias para la identificación y localización de objetos*

Estos aspectos conducen a nuevos requisitos, que serán tratados en apartados posteriores, pero antes continuemos analizando necesidades más básicas.

Concurrencia. Compartir los datos

La técnica serialización no es suficiente para la mayoría de las aplicaciones que necesitan compartir información además de hacer perdurar sus datos. Veámoslo con otro escenario.

La situación es típica en el mundo informático de la Banca, es el caso de uso de la actualización del saldo de una cuenta bancaria, mediante aplicaciones como las de banca electrónica por la Red y el comercio electrónico.

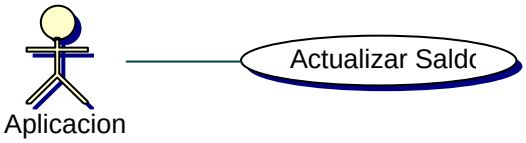


Figura 4 Caso de uso Actualización Saldo

La información debe ser compartida, esta es una obligación de todo servidor de base datos que tiene su reflejo en los servicios de persistencia que usan de los servicios de datos. [8] [2].

Un objeto persistente tiene dos aspectos: uno en ejecución y otro su estado almacenado. Como consecuencia de esta dualidad, la gestión del acceso concurrente a objetos debe cubrir ambos aspectos, el objeto como entidad en ejecución y su estado almacenado en los servicios de datos. De otro modo llegaremos a situaciones inconsistentes. Por ejemplo, los movimientos de una cuenta suman un saldo, pero la cuenta muestra otro distinto.

Java aporta mecanismos para la sincronización a distintos niveles: de clase, objeto, método y bloque de sentencias, con cláusulas y primitivas que controlan el acceso concurrente. En otros lenguajes es necesaria la implementación de las técnicas de sincronización. El siguiente listado la cláusula *synchronized* indica que el método no será ejecutado al mismo tiempo por dos procesos, el acceso al método queda bloqueado mientras su ejecución no ha sido completada.

```
  /**
       * Decrementa el saldo de cuenta anotando los detalles de la
operación
   * @param cantidad importe del cargo > 0
   * @param motivo de cargo
   * @param fechaValor del cargo
   * @return true si fue realizado el cargo, false en otro caso
   */
  public boolean cargo(double cantidad, String motivo, Date fechaValor)
  {
    synchronized (this) {
      double sld = saldo;
      sld -= cantidad;
      if ( (sld >= saldo) || ( -sld > limiteCredito)) {
```

```
        return false;
      }
      anotaMovimiento( -cantidad, motivo, fechaValor);
      saldo = sld;
      debe += cantidad;
      return true;
    }
}
```

Listado 7 Método cargo en cuenta

Si volvemos sobre el Listado 6, vemos que el estado grabado en el archivo podría ser modificado desde otro proceso ejecutado desde otra máquina provocando la inconsistencia entre el objeto y su estado guardado. *Un servicio de persistencia de objetos debe resolver el acceso concurrente a los objetos persistentes teniendo en cuenta también el acceso concurrente a su estado almacenado de forma consistente.*

Cada proceso puede acceder al sistema gestor de datos, deposito de los estados de los objetos, para recuperar un mismo estado. ¿Pero qué sucede cuando un objeto es modificado y su estado es accedido entre tanto por otro proceso? Es necesario emplear alguna estrategia de bloqueo que garantice la consistencia. Los sistemas gestores de datos proporcionan la capacidad de compartir y concurrir a los estados guardados. Las aplicaciones utilizan las interfaces de programación de los servicios de datos para la concurrencia, basadas en las estrategias habituales de control de acceso pesimista y optimista, en el primero los datos son bloqueados para su modificación hasta finalizar el proceso que solicita la modificación, y el control optimista, no se produce el bloqueo ante nuevas modificaciones, pero puede ser analizada la consistencia por la presencia de marcas de tiempo que permiten decidir sobre el efecto de las modificaciones. En ambos modelos de bloqueo pueden ser efectuados de forma automática o explicita, esto es, el acceso a los datos desencadena las acciones de control de concurrencia o bien, la invocación de primitivas de bloqueo. La principal diferencia entre una estrategia y otra es la mayor capacidad de procesamiento del control de concurrencia optimista. Pocos gestores de datos ofrecen ambas estrategias, incluso los hay que no ofrecen ninguna. *Un servicio de persistencia debe aprovechar las estrategias de control de concurrencia ofrecidas por los gestores de datos con los que interactúa, y en ausencia de estas, implantar el bloqueo pesimista y opcionalmente, el bloqueo optimista.*

El problema de la concurrencia es por sí solo un tema muy amplio fuera del alcance de este texto. En [17] se presenta de forma escueta y sencilla una introducción a la problemática de la concurrencia con bases de datos.

Transacciones

El soporte de transacciones es un requisito esencial. Al menos deben ser soportadas las funcionalidades de consolidar y cancelar las modificaciones. Que estrategia de transacciones sea utilizada automática o explicita, la elección de los distintas realizaciones: bloqueos, anidamiento,..., son características, que responden a criterios de diseño y necesidad. Habrá sistemas que soporten o necesiten de anidamiento de transacciones, transacciones de larga duración, bloqueo automático o explicito, etc. Igualmente su implantación dependerá del soporte dado por sistema gestor de datos final.

Con la serialización no hay transacciones, las modificaciones son consolidas en los estados almacenados, volviendo a ejecutar la operación de escribir el objeto. Trabajando con ODMG 3.0, están disponibles las operaciones de transacción de iniciar, consolidar y deshacer las modificaciones efectuadas. Trabajando con sistemas de bases de datos SQL, las operaciones de consolidar y deshacer transacciones forman parte del repertorio de instrucciones SQL de la mayoría de sistemas gestores, el inicio de una transacción comienza

con la ejecución de una orden SQL, así JDBC y SQLJ ofrecen la operación con transacciones. En 0 ilustra el acceso concurrente a objetos en Java y las operaciones básicas con JDBC.

Al margen del aspecto práctico de la consistencia, desde el punto de vista del desarrollo de aplicaciones para sistemas de información de empresa otro argumento más que apoya el requisito de disponer de transacciones, es la conveniencia de que las aplicaciones ofrezcan la capacidad de probar y deshacer modificaciones [25], permitir a los usuarios jugar con las aplicaciones sin consecuencias.

Integración de diferentes tipos de mecanismos de persistencia

Un gran número de aplicaciones, como la de nuestro ejemplo de Banca, deben integrar datos procedentes de bases de datos relacionales. En otras, es un requisito que el soporte persistente de los estados de los objetos, sea una base datos relacional. Por otro lado, las bases de datos relacionales de distintos fabricantes utilizan mecanismos de persistencia muy dispares que presentan limitaciones e incompatibilidades. A esto, debemos sumar que las organizaciones utilizan otros servicios de datos no relacionales, con sus propios mecanismos persistencia. Sin embargo, encontramos aplicaciones que exigen integrar datos procedentes de distintos servicios de datos: base datos relacionales, documentales, sistemas de archivos, Web... Los trabajos relacionados de C. Priese [12], C. Larman [18], S. Ambler [32], consideran necesaria la integración de mecanismos de persistencia distintos. *El soporte de los mecanismos de persistencia para las principales bases de datos relacionales del mercado es esencial.*

*También sería interesante que sean soportados otros mecanismos de persistencia bajo una misma interfaz de programación. M*anejar un único lenguaje aún con sistemas diferentes, una interfaz común, permitiría unificar bajo un *mismo protocolo el acceso a servicios de datos diferentes*. Esto significa mayor productividad, facilidad de integración e interoperatividad.

Veamos como ilustración de esta necesidad, sobre un simplificado caso de uso del cierre diario de un terminal punto de venta de supermercados. Imaginemos una cadena de supermercados con distintos locales dispersos, donde los terminales TPV operan de forma autónoma durante la jornada, al final del día sus datos son transferidos y procesados a otros sistemas información centrales de la corporación.

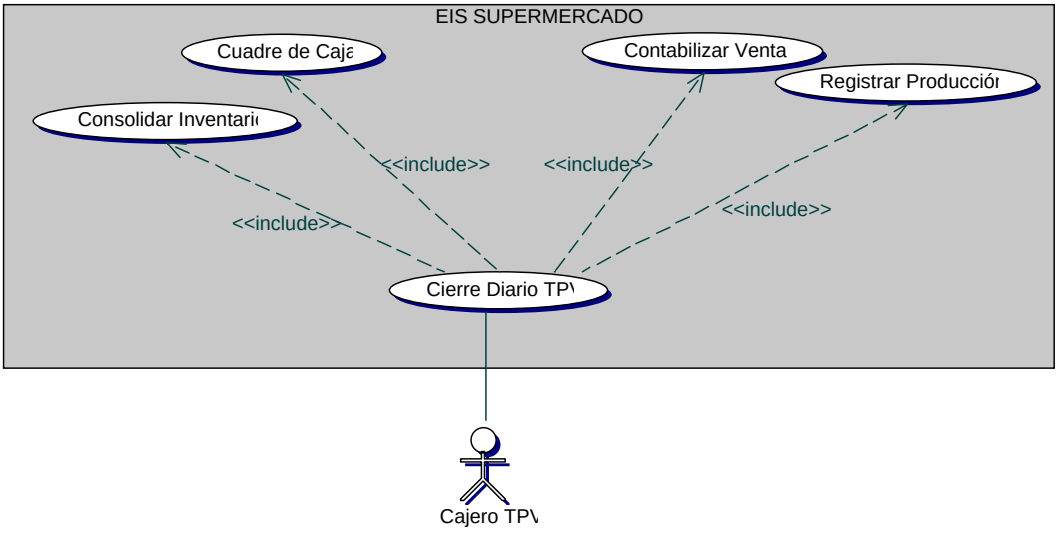


Figura 5 Caso de uso Cierre TPV

Podemos imaginar que los datos locales a transferir se encuentran en archivos o en una base de datos local en el TPV, desde ahí, los datos contables migran a un ERP basado en SAP/R3 o CICS con su propia interfaz de datos: datos sobre los pagos con tarjetas pendientes de compensar con las entidades financieras, cifras de ventas detallada por artículo, flujos de caja,...; los registros sobre las operaciones manuales efectuadas por operador: tiempos de apertura del cajón monedero, tasas errores de lectores ópticos, movimientos de la cinta transportadora, periodos de inactividad, incidencias,... van a parar una base de datos multidimensional de control de producción; y por último, las salidas de artículos son consolidadas contra los inventarios de los almacenes, que podrían estar sobre una base de datos relacional.

Distribución sobre distintos mecanismos. Conexiones Múltiples

Utilizar datos de fuentes distintas, en su emplazamiento, tipo e incluso fabricante, significa efectuar conexiones a cada uno de los servicios de datos implicados. Cada conexión define un contexto o canal sobre el que operar en los datos. Un objeto que maneja datos integrados desde varias fuentes, tendrá que acceder a cada uno de los servicios de datos, donde persisten los datos separados, con los mecanismos de persistencia propios de cada servicio, esto es, emplear canales adecuados para cada servicio. Posiblemente, el acceso deba ser simultáneo, como en el anterior caso de uso del cierre diario de TPV, que necesita de una conexión al soporte local y otra con el destino de los datos. En [14] recoge expresamente esta situación, como común a la mayor parte de los sistemas de información de las empresas. Evidentemente, existen numerosos ejemplos donde es necesaria la comunicación simultánea. También los requisitos estructurales, expuestos más atrás, justifican la necesidad de conexiones múltiples y simultaneas. *Es esencial que poder establecer múltiples conexiones simultaneas a cada uno de los mecanismos de persistencia soportados por el servicio de persistencia.*

Integridad

Un servicio de persistencia es un mediador entre dos sistemas o mundos, el de los gestores de datos y el de los objetos en ejecución. En su papel mediador debe permitir compartir y transferir datos entre los objetos y los servicios de datos. Compartir y transferir datos implica la responsabilidad de asegurar la exactitud de los datos manejados, proteger los datos contra modificaciones y el acceso no correctos [19] [8]. Esta responsabilidad es preservar la integridad de los datos, que es un requisito previo para lograr la transparencia de datos.

Es necesario asegurar la integridad de los objetos, sus datos, sus referencias a otros objetos, y la integridad de las transacciones. Cabría pensar que la integridad queda garantizada, por cada extremo de la comunicación para la persistencia junto con transacciones; de un lado, en los objetos, el lenguaje de programación con su sistema de tipos y gestión de excepciones, y del otro, los gestores o servicios de datos que sustentan la integridad datos y la integridad referencial cuando son bases de datos; y cubriendo ambos lados, las transacciones integrando las operaciones de cada lado como una sola y atómica. El código que sigue revela varios aspectos de la integridad que no pueden ser cubiertos ni por un sistema de tipos, ni por un gestor de datos, sino que deben ser cubiertos con la colaboración del servicio de persistencia.

La agregación, composición, la herencia, el polimorfismo y la ligadura tardía, la evolución de los esquemas de clases y bases de datos, y la dificultad de la falta de correspondencia entre los sistemas tipos de los lenguajes y los sistemas de gestión de datos afectan a la integridad, cuestión que debe ser contemplada en el diseño de un servicio de persistencia.

Veamos el siguiente código:

```
public void setCodigocuenta(String unCodigocuenta)
{
  codigocuenta = unCodigocuenta;
}

private Cliente recuperaMetodoSerialize(String idCli)
    throws Exception
{
  try {
    ObjectInputStream entrada = new ObjectInputStream(new FileInputStream(
        idCli));
    Cliente cliRecuperado = (Cliente) entrada. readObject();

    entrada.close();
    return cliRecuperado;
  }
  catch (Exception e) {
  ……..
 }
```

La asignación debe cambiar la BD si el objeto es persistente

Correspondencia BD y tipos programa.

Listado 8 Aspectos de la integridad

El listado anterior presenta tres cuestiones esenciales a resolver con la intervención del servicio de persistencia:

- ***Sincronización***, al modificar el estado de un objeto persistente, el estado depositado en el medio de persistencia, debería ser actualizado para reflejar el cambio.
- ***Correspondencia entre los modelos de clases-datos***, es necesario, conocer la correspondencia entre el estado de un objeto persistente y su representación en el medio persistente, salvo que ambos utilicen el mismo sistema de tipos, para poder efectuar la sincronización además de la grabar los nuevos objetos.
- ***Consistencia o congruencia de tipos,*** la recuperación del estado de objeto exige que los tipos de datos del estado almacenado y recuperado sean consistentes con el tipo declarado del objeto persistente.

Estudiemos estas cuestiones para descubrir nuevos requerimientos.

Sincronización

En el ejemplo del Banco es necesario actualizar los datos almacenados, la asignación del atributo "codigocuenta" o del "saldo", implica que su estado persistente debería ser sincronizado con su estado en ejecución. Las transacciones, las operaciones básicas y las conexiones proporcionan la funcionalidad básica. El código más atrás del método "setCodigocuenta", con persistencia completamente transparente, provocaría la actualización del estado persistente de forma acorde al control transaccional utilizado. En el siguiente presenta el extremo opuesto donde el programador asume la responsabilidad.

Listado 9 asignación persistente JDBC del Código de Cuenta

```
private booolean JDBCsetCodigoCuenta(String unCodigocuenta)
    throws Exception
{
  try {
    String antCodC = this.codigocuenta;
    this.codigocuenta = unCodigocuenta;
    pstmt = con.preparedStatement(
        "UPDATE  Cuentas SET  codigocuenta = ? " +
        "WHERE pkcuenta =?");
    pstmt.setString(1, this.codigocuenta);
```

Correspondencia objeto registro

Señala desde donde Copiar los datos

```
      pstmt.setString(2, this.pkcuenta);
      pstmt.executeUpdate();//transaccion iniciada con ejecución SQL
      pstm.close();
      con.commit();
      return true;
    }
    catch (Exception e) {
      System.err.println
          ("Excepcion. Restaurar valores ");
      System.err.println(e);
      this.codigocuenta = antCodC;
      con.rollback();
      return false;
    }
```

En este código anterior, el programador se preocupa de salvaguardar los datos en variables, para el caso de que se produzca un error, que obligue a deshacer la modificación, cancelando la transacción para que los cambios en la base de datos no surtan efecto; si todo va bien, una vez copiado el nuevo valor desde el atributo `codigocuenta` al contexto de la sentencia SQL preparada, los cambios son consolidados en la base de datos mediante la invocación de la operación de consolidar `commit`, mediante la conexión a la base de datos. El programador debe conocer, cual es la tabla donde persiste la cuenta, la correspondencia objeto-registro; se presupone una variable de clase, o instancia, que representa una conexión abierta ("con") sobre la que son efectuadas las operaciones de bases de datos, un empleo de transacciones pesimista implícito, el orden de las sentencias es crucial,...¿Pero que ocurría si el atributo `codigocuenta` fuera público y un método asignará su valor sin mediar operación contra la base de datos? Provocaría la falta de integridad ¿Cuál sería el valor correcto, el asignado en la aplicación o el residente en la base de datos?

La necesidad de mantener actualizado el estado persistente de los objetos, es tan importante que encontramos numerosos ejemplos donde las clases del dominio son utilizadas sólo para encapsular el código de acceso y actualización a los datos.

Al sincronizar, estamos moviendo o copiando el contenido de los objetos a su representación persistente. *Considerando el objetivo de transparencia de datos. Un servicio de persistencia debería actualizar automáticamente los estados almacenados de los objetos persistentes. Actualizar los estados de los objetos modificados, ocultando las instrucciones dependientes del mecanismo de persistencia utilizado, el lenguaje empleado con el mecanismo.* La sincronización o actualización automática, del estado persistente de los objetos, requiere saber en ejecución cual es la situación del objeto, si es nuevo, o si fue guardado y modificado, pero no ha sido aun consolidado en base de datos, ... conocer el estado actual del ciclo de vida de los objetos persistentes, información que es independiente del objeto, pero sin la cual no es posible automatizar la tarea. Esto afecta también a las relaciones entre los objetos.

Un servicio de persistencia de objetos, debe manejar las relaciones entre objetos de agregación, asociación, composición y herencia de manera que se preserve la integridad, ofrecer los mecanismos para conseguir la integridad de las referencias entre objetos de forma automática. Esto significa que las operaciones borrado, actualización, copia o control de concurrencia (bloqueos) deben proceder, según como están relacionados los objetos, por ejemplo un objeto compuesto, al ser bloqueado para modificación, serán bloqueados sus objetos agregados, al ser eliminado el objeto compuesto deben ser eliminados los agregados; las relaciones entre los objetos deben ser consistentes, no puede ser eliminado un objeto referenciado por otro.

Si queremos alcanzar la mayor ortogonalidad posible, es necesario crear el cierre de persistencia de cada objeto persistente automáticamente, dicho de otra forma, *un servicio de persistencia debería ofrecer persistencia por alcance*. La opinión y trabajos de autores como B.

Meyer, K. Priese, y D. Jordan y M. Atkinson ponen de manifiesto la conveniencia de implementar la persistencia por alcance.

El estándar ODMG 3.0 exige la transparencia de datos y la persistencia por alcance. Los estándares SQLJ, JDBC no consideran la transparencia de datos, ni la persistencia por alcance. Encontramos productos OODMBS que no soportan transparencia de datos, y que requieren la intervención del programador marcando los objetos, como modificados para actualizar y debiendo definir el protocolo de transferencia o copia al estado guardado, como sucede con Objectivity; y en el otro extremo, FastObjects o Versant productos estos que soportan transparencia y persistencia por alcance del estado directamente.

Compatibilidad de tipos

Más atrás en el Listado 8 ¿Cómo es garantizada la compatibilidad de tipos entre el objeto del proceso, `cliRecuperado` y el estado recuperado desde el objeto `entrada`? En el ejemplo se presupone que el tipo que será recuperado es compatible con la conversión forzada de tipos que aparece. Si algún proceso sustituye el archivo asociado al objeto `entrada` con un tipo incompatible, adiós exactitud de los datos, no es posible mantener la integridad. Cuando un objeto es guardado al recuperarlo es necesario:

- Conocer el tipo de datos del objeto, su clase, y también el tipo de dato de cada objeto del cierre de persistencia, cuando estos sean extraídos.
- Los tipos utilizados para reinstanciar a los objetos, deberán estar disponibles para ser usados.

La consistencia entre los tipos guardados y actuales de los objetos, y la disponibilidad de las clases en ejecución, conlleva no pocos quebraderos de cabeza en tiempo de ejecución. Tantos como para que un lenguaje, Eiffel, paradigma de los lenguajes orientados a objetos, justifique la inclusión de la instrucción de intento de asignación, una operación de conversión de tipos similar a las presentes en Java y C++ [19].

Para preservar la consistencia de los tipos entre objetos y datos, l*a información sobre el tipo de cada objeto persistente debería ser persistente también junto al estado guardado de cada objeto,* de esta forma es posible conocer en tiempo de ejecución el tipo que será preciso antes de usarlo, facilitando el asegurar la integridad en la correspondencia entre clase y datos.

Las disponibilidad de los tipos esta supeditada a la evolución del esquema de clases y al despliegue de las mismas, temas que dependen del proceso de producción de programas que debe incluir las tareas y procesos necesarios para garantizar la presencia de las clases adecuadas en tiempo de ejecución.

En ningún caso, las situaciones de error provocadas por no estar disponible un tipo o por incongruencia deben provocar la finalización abrupta de la aplicación, el programador debe tener la oportunidad de poder manejar las situaciones de error por inconsistencia o falta de integridad.

Las relaciones entre los objetos, de composición, agregación y herencia necesitan mantener la integridad de las referencias, la solución no es única, en una aproximación las relaciones pueden ser contenidas como parte integrante del estado de un objeto, en otras, las relaciones se convierten en objetos independientes con estado propio.

Cuando un objeto es contenido dentro de otro (composición), forma parte del estado del objeto que lo contiene, y como tal debe ser gestionado por el servicio de persistencia. ¿El objeto contenido tendrá identidad propia en el depósito de los datos?¿Podrá ser obtenido directamente por una consulta, o habrá que navegar por las referencias? Acceder y modificar el estado en base de datos de un objeto contenido como

privado, sería un quebrantamiento de la integridad. La agregación o asociación en la que objetos están relacionados, pero tienen vidas independientes, es la relación donde un objeto es referenciado por otro, presenta el problema de la validez de la referencias, un objeto referenciado puede haber sido eliminado, provocando un error cuando otro objeto pretende recuperarlo.

¿Qué hacer cuando un objeto es eliminado y existen referencias dirigidas a este? Son posibles distintas técnicas, por ejemplo, levantar una excepción al intentar suprimir un objeto referenciado por otros, notificar a los objetos contenedores para actualizar sus referencias, eliminar cada referencia pero no el objeto salvo la última, detener la eliminación o simplemente eliminar el objeto sin más, accesos posteriores provocarían una excepción de objeto inexistente. ¿Qué hacer cuando al recuperar un objeto por medio de una referencia, su clase ha cambiado? Cambiar el objeto en ese instante, forzar el tipo al anterior o levantar una excepción.

El manejo de las situaciones descritas demuestran una complejidad y esfuerzo de programación importantes para conseguir mantener la integridad de las relaciones de agregación y composición entre objetos, que van más allá de conservar el cierre de persistencia.

Cada estándar considera la integridad referencial de distinta forma. La serialización solo considera la posibilidad de guardar conjuntamente el objeto con su cierre de persistencia, aún cuando los objetos del cierre de persistencia deberían poder ser accedidos de forma independiente unos de otros. ODMG 3.0, maneja expresamente la herencia y agregación con los atributos, para la composición y las relaciones para la asociación, las relaciones pueden estar dirigidas y tener inversa, levantar excepciones o provocar la eliminación en cascada de las dependencias aunque no exige la persistencia por alcance. En la persistencia con objetos sobre bases de datos relacionales para dar soporte adecuado a la integridad, se hace necesaria la incorporación de disparadores de bases o restricciones de integridad de datos que garanticen la integridad de los objetos compuestos de objetos, JDBC y SQLJ no contempla la integridad de las referencias, debiendo ser el programador el que plasme la integridad referencial.

La herencia plantea dificultades para la persistencia relativas a la ocultación de la información y al polimorfismo. Una clase podría no ser persistente pero otras descendientes directos o indirectos sí. ¿Qué sucede con las características privadas heredadas? También, puede ocurrir que una clase sea persistente tenga descendientes que hacen privadas características heredadas, características heredadas que cambian de tipo...

El polimorfismo implica perdida de información que es necesaria para materializar la ligadura tardía. Para posibilitar el polimorfismo de instancias persistentes en transparencia de datos, el conocimiento sobre el tipo debe ser también persistente, para poder a partir de este, decidir cual es el adecuado a usar al revivir una instancia, cuando el contexto de ejecución no permite inferir el tipo o clase correcto.

Si el lenguaje de programación permite herencia, polimorfismo y ligadura dinámica, entonces el servicio de persistencia debería facilitar la operación sobre objetos persistentes con polimorfismo.

El patrón de diseño Bridge [13] es un ejemplo de la necesidad descrita en el párrafo anterior, aplicado en [26], en el que la implementación de un diccionario es cambiada según el volumen de información, esto significaría utilizar estados persistentes que serán tipos distintos en diferentes momentos. Otro, son las colecciones persistentes que contienen

objetos que pertenecen a distintas clases dentro una jerarquía de herencia. Algunos de los productos revisados, no soportan polimorfismo, como por ejemplo Castor.

Integridad de acceso

La integridad requiere protección además de exactitud, asegurar que el acceso a los datos, sea realizado acorde a los privilegios establecidos. Los mecanismos de persistencia ofrecen distintas capacidades de control de acceso a los datos, para establecer políticas de privilegios sobre la lectura y modificación de estos, en base a sistemas de credenciales tales como los certificados digitales, nombres de usuario con contraseñas, kerberos u otros. Los ejemplos vistos del Banco y el supermercado describen dos entornos que requieren asegurar las políticas de privilegios establecidas.

Un servicio de persistencia debería asegurar la seguridad de acceso a las instancias que maneja. El servicio de persistencia en su papel de mediador debe trasladar hasta los clientes, usuarios o procesos, las políticas establecidas con el concurso de los mecanismos de persistencia usados.

Un punto de vista importante, es conseguir también la ortogonalidad entre persistencia y seguridad, consiguiendo que los programas sean independientes de la seguridad, que el código fuente no sea alterado con comprobaciones de privilegios, resulta así, un código más seguro, porque el programador no tiene capacidad para alterar o conocer las políticas de privilegios, y además, la independencia facilita un mejor mantenimiento, porque los cambios en los permisos de acceso no afectan al código. Las referencias inteligentes o proxys son un posible instrumento para plasmar esta necesidad de protección del acceso, sin producir una revelación de privilegios [13] [27].

Los sistemas de bases de datos relacionales articulan los privilegios de acceso sobre las vistas, las sentencias grant y las figuras de usuario, grupo o rol con contraseñas, creando con todo esto un esquema de autorización y privilegios. En las bases de datos basadas en objetos, sin la posibilidad de vistas de objetos, no hay un estándar que establezca cual debe ser el mecanismo de autorización, cada fabricante aporta su punto vista, ofreciendo políticas de acceso en uno o más niveles, desde la página de almacenamiento, colección o contenedor, hasta la clase, el objeto y sus atributos. Por ejemplo, Versant permite definir para cada clase de una base de datos, cual es el acceso a las instancias contenidas en función de la identidad del usuario y el grupo; Gemstone permite definir políticas de acceso para los objetos almacenados juntos en contenedores, denominados cluster, sin considerar la clase de los objetos; Objectivity brinda una serie de librerías para construir soluciones ad hoc que manejan el concepto de credencial y privilegio sobre instancias, clases, contenedores y bases de datos. En Java, la API Security o el estándar CORBA Common Secure Interoperability Version 2 son dos marcos de referencia sobre los que construir las políticas de privilegios de un servicio de persistencia, siendo en esta última posible definir políticas de acceso a nivel instancias concretas.

Transparencia de datos

Hasta el momento han sido expuestos distintos aspectos que sientan las bases para concretar el objetivo inicial de transparencia de datos justificado por sus ventajas, también han sido presentados buena parte de los requisitos necesarios para conseguir esta. Requisitos expuestos, que son relacionados aquí para centrar la atención sobre la transparencia de datos. Es necesario para tener transparencia de datos:

1. *La recuperación del estado almacenado de los objetos, incluidas las relaciones con otros objetos.*

2. *La construcción o materialización de los objetos persistentes referenciados en los atributos de los objetos persistentes accedidos.*
3. *El cumplimiento de las restricciones de integridad de identidad y de las referencias.*
4. *La sincronización dentro de una transacción del estado persistente de los objetos persistentes con el estado almacenado.*
5. *Seguimiento automático de los cambios producidos en el estado de los objetos y su consolidación en los medios de persistencia.*
6. *Mantener la información suficiente de las modificaciones producidas en el transcurso de una transacción, para poder descartar estos cambios, si la transacción es rechazada.*
7. *Convertir de modo automático objetos transitorios a persistentes si son referenciados por algún atributo persistente de un objeto cuando la transacción es consolidada, esto es, plasmar persistencia por alcance*

Veamos el siguiente código dentro del caso de contexto del banco, como funciona la transparencia de datos.

```
private void comofuncionaTransparencia()
{
    Cliente cli = ServPersistencia.buscarCliente("....");

    cli.direccion.calle = "Avda. Mediterraneo";
    cli.direccion.portal = "s/n";

    Cuenta[] ncu = new Cuenta[27];

    ncu[27].movimientoApertura(5000);
    cli.setCartera(ncu);

  }

private void invocarOperacionComoFT()
{
  try {
       Transaction t = new Transaction();
       t.begin();
       comofuncionaTransaparencia();
       t.commit();

  }
  catch (Exception e) {
         rollback();
         e.printStackTrace();

  }
```

Listado 10 Ejemplo de transparencia de datos

El primer método `comofuncionaTransparencia` comienza con la localización de un objeto cliente, que estará disponible mediante la variable de referencia del tipo `Cliente`, `cli`, objeto persistente, que si es modificado usando la magia de la transparencia de datos, entonces implica los hechos siguientes:

- La modificación de la calle y el portal del objeto `dirección` contenido en el objeto referenciado por la variable `cli`, es extraído desde la base de datos sin que el programador indique expresamente su recuperación. La navegación implica, localizar los datos correspondientes, a los estados almacenados de las referencias accedidas, construir con los datos recuperados el objeto de tipo adecuado, en esta ocasión del tipo `Dirección`.

- Las modificaciones llevadas a cabo son automáticamente reflejadas en los estados almacenados en la base de datos. Las asignaciones de la calle y portal son trasladas en su momento, hasta el estado almacenado del objeto `dirección` apuntado desde la referencia `cli`, otra vez más, sin que el programador intervenga explícitamente.
- Los objetos transitorios referenciados por un objeto persistente se convierten en persistentes, el vector `ncu` pasa a ser persistente tras la invocación del método `setCartera`. El programador no codifica la transición de temporal a persistente.
- Las relaciones entre los objetos son mantenidas de forma consistente, en el listado anterior el objeto `dirección` y el nuevo vector de las cuentas asociadas al cliente, son reflejadas sobre la base de datos vinculados al objeto `cliente` que los contiene.
- Todas las modificaciones y los nuevos objetos hechos persistentes son grabados en la base de datos. El programador no incluye código expreso de grabación en base de datos.
- Como durante el proceso es posible que surja un problema entre el programa y la base de datos, los datos necesarios para efectuar la recuperación ante una excepción son conservados, sin la intervención del programador.
- El método de este ejemplo, `comofuncionaTransparencia`, opera tanto con instancias que son persistentes, como con transitorias, no hay diferencias, salvo que se solicita la localización de una instancia a un objeto que representa el servicio de persistencia y debe ser invocado dentro de una transacción.
- El contexto donde es invocado el método, debe tener en marcha una transacción que garantice la integridad.

Referencias inteligentes

El listado siguiente maneja un objeto persistente `micuenta`, que contiene a su vez a otros del tipo `Movimiento`, cuando la referencia `m`, apunta a los apuntes de `micuenta`, pueden suceder dos cosas, o todos los objetos movimiento han sido previamente recuperados, desde la base de datos por el programador, o bien son recuperados solo cuando es estrictamente necesario por el sistema, cuando pasan a ser copiados como texto en el objeto `listMovsCuenta`, que forma parte de la interfaz visual. Navegar entre los objetos provocará la recuperación de los objetos, si todavía no estaban en el espacio de memoria de la aplicación. La operación de recorrido con recuperación es costosa, por lo que mejorar la navegación será una meta a lograr. En [7] [20], se presentan sucintamente algunas de las ideas empleadas para mejorar la navegación, como son la conversión dinámica de identificadores de objetos a direcciones de memoria y viceversa, swizzling y la indización por identificador de objetos y direcciones memoria. En [12], es descrito el concepto de referencias inteligentes que quieren mejorar el rendimiento de acceso a los objetos persistentes. Las referencias inteligentes amortiguan la sobrecarga producida por la recuperación de objetos durante la navegación. También pueden ser utilizadas para asegurar la integridad de las referencias y el control del acceso concurrente, de modo transparente para el programador al mediar entre acceso y la recuperación desde los sistemas de datos, posibilitan la incorporación del código necesario para concretar estas funcionalidades. Las referencias inteligentes o punteros inteligentes, son una de las cuatro aplicaciones del patrón de diseño llamado Proxy que podemos consultar en [13]. Las referencias inteligentes son utilizadas por los principales sistemas gestores de bases de

datos objetuales y en diseños reconocidos como Pjama 0 para Java sobre gestor de objetos y 0, 0 y [i26] para objetos soportados por bases relacionales.

```
  private void mostrarCuenta(){
        Movimiento [] m;
        DecimalFormat d = new DecimalFormat() ;
        String sld ;
        lblCodCuenta.setText(micuenta.getCodigocuenta());
        labelCredito.setText(d.format(micuenta.getLimiteCredito()));
        sld = d.format(micuenta.getSaldo());
        lblSaldo.setText(sld);
        m = micuenta.getApuntes();
        for( int i= listMovsCuenta.getItemCount() ; i< m.length ; i++){
           listMovsCuenta.add(((Movimiento) m[i]).toString() + " Saldo: " +
sld, i);
        }
}//mostrarCuenta
```

Listado 11 Método con transparencia de datos

Un servicio de persistencia debería emplear referencias inteligentes para mejorar el rendimiento, reduciendo el coste del esfuerzo de recuperación de los objetos, posponiendolo hasta el instante antes de ser necesario su uso y para facilitar el control de la integridad referencial y acceso concurrente, sin intervención del programador. La operación de recuperación de objetos, además del acceso a los mecanismos de persistencia, requiere de la asignación de memoria y su inicialización, la creación de una instancia, su inicialización, la asignación de los valores recuperados, etc., esto llega a ser un problema cuando trabajamos con objetos verdaderamente grandes como colecciones persistentes enormes, y por otra parte, la necesidad de asegurar la integridad puede ser acometida validando las referencias antes de ser usadas, y al gestionar el bloqueo concurrente se evitan situaciones de bloqueo y actualización inconsistente.

1.2.1.3. Consultas ex profeso

La transparencia de datos permite navegar desde una instancia persistente por la telaraña de referencias de su cierre de persistencia, pero al menos una referencia, la del objeto raíz de ese cierre debe ser obtenida expresamente. Esta primera referencia puede ser obtenida a partir de un identificador de objetos, un nombre asociado previamente, o con una búsqueda condicional sobre las instancias persistentes, mediante consultas ad hoc. Consultas y navegación entre objetos se complementan, las consultas permiten localizar los objetos desde los que navegar hasta a aquellos que desea manipular.

Los sistemas gestores relacionales proporcionan SQL para realizar consultas ad hoc, en el lado de los gestores objetuales también tienen su estándar en ODMG 3.0, OQL [7]. Los productos consultados ofrecen interfaces de programación que permiten la definición y ejecución de consultas ad hoc Objectivity, Versant, Fast Object y Apache OJB. Este requisito esta presente en el manifiesto sobre bases de datos de objetos, que establece como último requisito, la facilidad de consultas para expresar que objetos queremos alcanzar, con un alto nivel de abstracción, eficiencia e independencia. Una facilidad de consultas ad hoc ayuda a simplificar la programación y a mejorar la extensibilidad [2].

Si un servicio de persistencia pretende ser el punto de acceso a los servicios de datos, deberá ofrecer la posibilidad de efectuar consultas ad hoc a través suya. Una vez más, la aplicación de la orientación a objetos, posibilita ofrecer una solución que evite las diferencias entre los mecanismos soportados bajo una misma interfaz, en esta ocasión, de consultas con alto nivel de abstracción, eficiencia e independencia de los mecanismos. El tema es muy amplio, en [7] y [4] podemos hallar una breve introducción a la problemática de las consultas con objetos.

Funcionalidades relativas al rendimiento

Son necesarias otras funcionalidades para lograr la persistencia con la eficacia y eficiencia adecuadas, las operaciones de persistencia deben ser ejecutadas de forma efectiva y económica en recursos. La ilustración de un escenario sobre el ejemplo del banco es el pretexto para presentar nuevos requisitos relativos al rendimiento, el escenario de la Figura 4.

Las aplicaciones acceden a las cuentas del banco para llevar a cabo cargos y abonos, esto puede implicar extraer cada objeto `cuenta` al que se aplica la operación de cargo o abono; cada una de estas operaciones, añade una nueva instancia `movimiento` a la colección de numerosos movimientos que una cuenta tiene, en suma, una situación donde se podrían plantear algunas preguntas relativas al rendimiento:

- ¿Procesar un objeto cada vez o varios simultáneamente? Acceder objeto por objeto en secuencia, es más lento que agrupar las operaciones, de forma que se solapen aquellas actividades que pueden ser simultáneas, o que agrupadas reducen la latencia en el acceso a los datos.
- ¿Extraer todos los movimientos de cada objeto cuenta o solo una referencia al que es necesario? Con ciertas operaciones no es preciso conseguir el objeto al completo, simplemente disponer de una alguna representación suficiente, para el tratamiento a efectuar, extraer y manejar exactamente, los datos precisos para llevar a cabo el proceso.
- ¿Por qué no modificar el estado de los objetos directamente en la base de datos sin reconstruir los objetos, de forma transparente para el programador? Hay ciertas ocasiones, donde cada elemento de una colección es actualizado, sin provocar efectos laterales en otros objetos; con la indicación expresa del programador, la magia de la persistencia podría resolver, aplicar directamente las transformaciones en los estados almacenados en lugar sobre cada objeto afectado.

Gestión automática de caché

La utilización de técnicas de lectura anticipada y escrituras diferidas, sistemas de caché, permite mejorar el rendimiento del flujo de almacenamiento y recuperación de objetos desde los gestores de datos. El empleo inteligente de caches reduce los costes de acceso los servicios de datos [41]. Las aplicaciones esperan acceder a objetos proporcionados por el servicio de persistencia, que obtiene los datos desde los mecanismos de persistencia para ser convertidos en objetos y viceversa, el servicio recibe objetos que deben ser convertidos, en datos apropiados a los mecanismos de persistencia. Todo ello, requiere un espacio de memoria y un tiempo de cómputo, que afecta al tiempo de respuesta de las operaciones de las aplicaciones, mientras que los usuarios esperan respuestas inmediatas. El rendimiento del acceso y actualización de objetos puede aumentar, si las operaciones para materializar y desmaterializar instancias son sustentadas con una gestión de caches adecuada, que pueden facilitar la agrupación de múltiples operaciones y objetos por acceso, para reducir así número de peticiones a los mecanismos de persistencia y la latencia total en el acceso a los objetos [41] [32]. Con caches y el agrupamiento de operaciones por acceso, son dos de los mecanismos para acometer, aumentos importantes, escaladas, en la carga [6].

Un servicio de persistencia debería implantar una gestión de caché de objetos para mejorar el rendimiento de acceso a los objetos persistentes. Debería ser posible extraer y almacenar grupos de varios objetos en cada acceso a los servicios de datos.

El tema no es trivial. La mayoría de los mecanismos de persistencia emplean algún tipo de caché o memoria intermedia para gestionar el acceso y actualización de los datos. A la dificultad propia de la gestión de caché, se añade la problemática de la integración, coherencia y validación de caches diferentes, los de cada mecanismo de persistencia y los de un servicio de persistencia. En los últimos años, varios fabricantes como Oracle, Gemstone y Poet, viene haciendo especial énfasis sobre este particular, con el anuncio de nuevos avances patentados de gran incidencia sobre el rendimiento escalable. El tema es complejo y está fuera del ámbito de estas líneas. Unas referencias que pueden servir como primera aproximación, son [6] [19] [41] [32].

Iteradores

A lo largo del texto han sido mencionados distintos ejemplos, en los que era necesario procesar los objetos de una colección, una secuencia de objetos. La iteración sobre los elementos de una colección debe ser realizada con eficacia y eficiencia, esto es una necesidad asumida en parte por los mecanismos de persistencia, que suelen ofrecer los medios para iterar de un modo controlado sobre secuencias de elementos. Las bases de datos relacionales ofrecen la facilidad de iterar sobre las filas resultantes de una consulta, mediante los cursores, los OODBMS también ofrecen objetos iteradores sobre colecciones de objetos persistentes.

En la Figura 1 las relaciones son [0..n], estas relaciones serán plasmadas mediante colecciones de objetos arbitrariamente grandes; en la situación anterior de añadir un movimiento a la `Cuenta`, el empleo de un iterador podría evitar extraer todos los movimientos de la cuenta afectada, por ejemplo calculando la referencia adecuada sin necesidad de recuperar objeto alguno, para incluir un nuevo movimiento en la colección almacenada correspondiente. Los trabajos de W. Keller [42] M. Atkison [33] y S. W. Ambler [32], por citar solo tres, incluyen expresamente la conveniencia del soporte de cursores para el acceso a los datos. Asumiendo que los cursores son, en definitiva, una forma de iterador, la necesidad de su soporte es clara, pues ayudan al recorrido de colecciones manejando los recursos necesarios, para crear la ilusión de navegación por los objetos contenidos. *Un servicio de persistencia debería ofrecer iteradores sobre colecciones de objetos persistentes*.

Proyecciones u Objetos Representantes (proxies)

Las cuestiones al principio de este apartado y lo escrito en párrafos anteriores, permiten cuestionar si realmente es necesario extraer un objeto al completo, para operar con él, ¿no bastaría con extraer aquello que va a ser afectado? Imaginemos un estudio de ingeniería de obras públicas faraónicas, donde su sistema de información maneja objetos tales como un objeto *proyecto*, agregado de muchos más objetos y muy grandes, como p. Ej. planos, imaginemos que un objeto *proyecto* es accedido para modificar la asignación de un *capítulo* de su *presupuesto*. ¿Alguien puede creer que sería eficiente extraer todo el objeto para modificar los pocos caracteres que ocupan un importe? La solución es utilizar objetos especiales que actúan en representación de los objetos a tratar, distintos a los originales, ideados para contener justo lo necesario para posibilitar la ilusión de interacción con los objetos persistentes, estos objetos son proyecciones de los objetos que representan, son llamados objetos representantes (proxies). Las referencias inteligentes, los iteradores y los caches complementan esta funcionalidad, necesaria para acometer la recuperación selectiva del estado de los objetos. Varios de los autores ya mencionados, S.W.Ambler, W. Keller han incluido la funcionalidad de objetos representantes en sus propuestas de servicios de persistencia. *Un servicio de persistencia debería soportar objetos representantes para mejorar el rendimiento del acceso transparente*.

En [13] se describe el patrón Proxy (representante) que es fundamento básico para comprender y acometer la realización de esta funcionalidad. En [19] también ilustra la conveniencia de objetos y clases de utilidad para la aplicación que actúan como representantes de forma expresa.

Aprovechar las características avanzadas de rendimiento de los servicios de datos

La última cuestión planteada al comienzo del apartado, considera que ciertas operaciones de actualización podrían ser más rápidas de efectuar directamente sobre los datos sin necesidad de recuperar objetos, siempre que sea posible asegurar que otros objetos en ejecución no se ven afectados por estos cambios. Esta forma de actuar requiere de aprovechar las capacidades particulares de los servicios de persistencia, lo que puede parecer un contrasentido respecto del objetivo de ortogonalidad enunciado más atrás, plantear la utilización del lenguaje del mecanismo de persistencia particular; pero, precisamente las técnicas orientadas a objetos, posibilitan sacar partido de las características avanzadas de los mecanismos de persistencia, sin revelar los detalles a los clientes del servicio de persistencia y sin trastocar demasiado la ortogonalidad. Son considerados tres aspectos principales a las bases de datos relacionales cuya aportación al aumento de rendimiento no deberían ser ignorados por una mal entendida pureza de la orientación a objetos de la soluciones [41][32]:

- SQL nativo
- Modos de operación avanzados
- Procedimientos almacenados

SQL nativo

El lenguaje de consulta de bases de datos estructurado, SQL, que ofrecen gestores relacionales, presenta variaciones que afectan también a como son efectuadas las sentencias. A título de ejemplo, Oracle añade la cláusula `connect by`, a la sentencia `select` para presentar resultados según un orden jerárquico; en sus versiones 7, el orden de inclusión en la cláusula `from` de las relaciones, afectaba expresamente al rendimiento, con la elección de la relación directriz del acceso a los datos. SQLBase 5.x permitía la utilización de expresiones calculadas en la definición de los índices, decidir entre usar la semántica de Oracle, o la estándar para la consultas con proyecciones, sobre correspondencias de claves ajenas compuesta con nulos (outer joins). Así, un largo etcétera con SqlServer, Potgress, RDB, DB2, Sysbase... SQL, aunque es un lenguaje declarativo, requiere de un conocimiento profundo del esquema manipulado y de las operaciones a realizar, ya que una misma operación expresada con sentencias distintas, e idéntico resultado, podrían tener un coste de cómputo y tiempo muy dispares incluso en varios órdenes de magnitud.

Las directivas, instrucciones dirigidas al intérprete o compilador de SQL, son otra de las capacidades para mejorar el rendimiento propias del SQL de cada producto. Estas permiten alterar el plan de ejecución que elabora el compilador, por ejemplo, alterar el número de hebras al paralelar una consulta, el índice que guíe la selección,...De una forma u otra, estas extensiones permiten modificar el plan de ejecución de las sentencias SQL con la pretensión de mejorar el rendimiento y la productividad. Hay productos entre los que se ha tenido acceso que expresamente permiten SQL nativo como Hibernate.

Modos de operación

Algunos gestores relacionales, ofrecen modos para la extracción y actualización de datos especialmente concebidos para mejorar el rendimiento. Consultas paraleladas y de

solo lectura, actualizaciones sin posibilidad de recuperación, y procesamiento asíncrono y diferido de las operaciones. Estas capacidades son presentadas, a veces, como nuevas sentencias SQL nativas, o como interfaces de programación especiales, ejemplos de ambas opciones, Oracle con su sentencia `alter session` y la interfaz de SQLBase de Centura Corp.

Procedimientos almacenados

Los principales sistemas gestores de bases de datos relacionales actuales permiten la ejecución de código en el lado del propio servidor, este es almacenado en la base de datos misma, que lo ofrece en forma de paquetes o librerías de procedimientos y funciones, y como código asociado a ciertas operaciones de acceso a datos (disparadores). El código almacenado, permite definir reglas de integridad complejas y empotrar operaciones del negocio, en la propia base de datos, que difícilmente podrían ser expresadas como simples sentencias SQL, por su falta de completitud computacional. En muchas ocasiones, el código almacenado constituye la única solución efectiva. También los sistemas de las bases de objetos brindan esta posibilidad de partir el código entre clientes y servidores, como servicios de aplicaciones integrados en la base de objetos, por ejemplo Gemstone/S, que presenta sendos entornos de ejecución Smalltalk y Java junto al servidor de objetos.

En definitiva, *sería conveniente poder disponer de las capacidades para mejorar el rendimiento, ofrecidas por los mecanismos de persistencia utilizados. Idealmente deberían ser presentadas por el servicio de persistencia a modo de conjuntos de servicios o propiedades relativas a los mecanismos.*

Un servicio de persistencia con soporte relacional debería permitir la utilización de código SQL nativo, modos de operación nativos y procedimientos almacenados de los servicios de datos relacionales soportados.

Funcionalidades avanzadas

Una vez cubiertos los aspectos básicos de la problemática de la persistencia, veamos otros aspectos que podrían ser tenidos en cuenta por los servicios de persistencia, porque la puesta en práctica de un servicio de persistencia desvela nuevas áreas de interés que se espera sean resueltas con la ayuda del servicio de persistencia:

- La gestión del cambio del esquema.
- Servicio de persistencia como fuente de datos.
- Múltiple transacciones coordinadas.
- Procesamiento de sucesos y eventos relativos a errores, transparencia.
- Medidas de productividad y rendimiento.
- La eficiencia de almacenamiento.
- Trazabilidad.
- Neutralidad respecto al lenguaje de programación.

Gestión del cambio del esquema de persistencia

¿Qué ocurre cuando una clase con instancias persistentes guardadas, es modificada en sus características? ¿Cómo asimilar los cambios en el esquema de los servicios de datos?

Estas preguntas destacan la problemática del cambio en el esquema de persistencia. Acometer la gestión de los cambios en el esquema necesita disponer de la capacidad de manejar meta información, datos sobre los esquemas de las clases y de los datos; meta

información que es utilizada en automatizar la gestión de las modificaciones, en ambos lados de la correspondencia clase-datos. El tema es muy amplio para ser descrito en este trabajo. Una descripción de la taxonomía del cambio se encuentra en [4]. Veamos solo tres cuestiones relativas a la evolución e integración de esquemas.

Evolución del esquema

En numerosas ocasiones los esquemas de persistencia son dinámicos, cambian con el tiempo, pero la evolución del esquema de clases y de persistencia, no siempre pueden ser simultaneas, pensemos en una base de datos con una colección de instancias que ocupa varios terabytes de datos, plantear un cambio instantáneo en la estructura de almacenamiento de tantos datos no es factible, tendría más sentido evolucionar, según un modelo de actualización bajo demanda. Esto implica la necesidad de poder manejar el esquema de persistencia en ejecución. Lo que conduce a que u*na facilidad opcional para un servicio de persistencia, sería la administración del esquema de persistencia en tiempo de ejecución.*

Esto permitiría cierta flexibilidad y capacidad de adaptación, ante cambios en los modelos de datos y del esquema de persistencia, e incluso, ante cambios en el modelo de objetos en tiempo de ejecución, posibilidad que esta disponible en ciertos lenguajes de programación como Smalltalk y Java.

Versiones

La necesidad de evolución comprende situaciones donde los objetos, instancias individuales, pueden cambiar no solo en el contenido de sus atributos, sino en sus características, en su funcionamiento, manteniendo constante su identidad; estos cambios crean distintas variantes de un mismo objeto, versiones, que representan la evolución de un objeto. Pensemos por ejemplo en un seguimiento de cambios o trazas de documentos o programas, donde es necesario mantener cada versión, que representa un mismo objeto en instantes diferentes [4].

Automatización de la correspondencia y generación de esquemas.

La falta de correspondencia supone un esfuerzo de programación que puede ser evitada mediante la aceptación de algún modelo general de correspondencia como [6] o [24]. Esto permitiría la automatización del proceso de correspondencia manejando meta información sobre objetos y los tipos de los mecanismos de persistencia. La automatización debería ser efectiva en los dos sentidos, desde los objetos a los datos y al contrario, desde los datos a los objetos. En particular, las bases de datos relacionales posibilitan esta forma de actuar, porque facilitan la operación sobre sus catálogos con SQL. *Un servicio de persistencia podría facilitar con la meta información adecuada:*

- *La generación de esquemas de datos a partir de las clases.*
- *La generación automática de clases a partir de los esquemas de datos.*
- *La gestión del cambio a cada lado de la correspondencia clase-datos.*

Servicio de persistencia como fuente de datos

Un servicio de persistencia podría ofrecer datos en forma de registros, adecuados para muchas de las herramientas de consulta de usuario final que existen o para la generación de archivos en diferentes formatos (html, csv o XML). S.W. Ambler propone expresamente esta opción [32]. *Los servicios de persistencia podrían presentar los datos que obtienen en forma de registros. Con ello se potencia el papel de mediador de datos de los servicios de persistencia ampliando sus posibilidades como pasarela transparente. Los servicios de persistencia a*

su vez se convierten así en fuentes de datos para otros servicios, como las herramientas de generación y ejecución consultas e informes.

Múltiple transacciones coordinadas.

El planteamiento establecido en secciones anteriores, considera que los servicios de persistencia deben integrar distintos mecanismos de persistencia; trabajar con varios mecanismos de persistencia de forma simultánea, esto necesita de coordinar las transacciones entre los diferentes mecanismos, cuando los objetos persistentes implicados en una misma operación residen sobre mecanismos dispares.

El modelo de transacciones de un servicio de persistencia podría asumir la integración y coordinación de los modelos de transacciones aportados por los mecanismos de persistencia que soporta, evitando delegar esta actividad en el programador. Así también, las transacciones desplegadas por los servicios de persistencia deberían poder ser coordinadas a un nivel superior, integrando las transacciones en servidores de aplicaciones y monitores transaccionales, permitiendo de esta forma la integración de los servicios de persistencia con otros tipos de servicios, presentados en estos servidores de aplicaciones y transacciones.

Procesamiento de eventos en ejecución

Durante la ejecución de los objetos persistentes surgen señales desde el entorno y desde las distintas piezas que sustentan el servicio de persistencia, señales que pueden cambiar el flujo habitual de la ejecución. Estas señales corresponden a situaciones de error debidas a acontecimientos inesperados o fallos, y a sucesos o eventos consecuencia de la actividad de persistencia. En ambos casos, los eventos pueden requerir la intervención del programador con la incorporación de código adecuado.

Los errores que surgen en la operación con los mecanismos de persistencia, deberían ser transmitidos hasta los programas para poder ser tratados adecuadamente. Errores como la duplicidad en claves únicas, la asignación de valores nulos o la violación de reglas de integridad complejas ejecutadas por los servicios de datos, deberían ser manejados por las aplicaciones, en la misma forma que cualquier otro error en tiempo de ejecución. Desgraciadamente las codificaciones hechas, para unas mismas situaciones de error, por productos distintos son también distintas, con el consiguiente impacto en la programación de su procesamiento. Las diferencias entre los distintos mecanismos pueden ser allanadas con un diseño que envuelva, unifique y homogenice el tratamiento de los errores desplegados desde los mecanismos de persistencia. *Todo servicio de persistencia debería definir una gestión de errores que unifique, homogenice los errores presentados por los mecanismos soportados*. Típicamente la gestión de errores estará sustentada por la utilización de variables de estado y excepciones. Las variables u objetos de estado recogen el resultado de éxito o fracaso de las operaciones. Las excepciones recogerán las situaciones de error, anormales e inesperadas que no pueden ser tratadas en el flujo normal del programa, por ejemplo, la interrupción de la comunicación.

Cuando la transparencia no es completamente ortogonal, afecta al modo en el que objetos dependientes se relacionan en tiempo de ejecución, aunque las aplicaciones manejan los objetos sin necesidad de saber si la instancia esta o no en memoria realmente, simplemente usan los objetos de igual forma que otros no persistentes, en determinadas casos es necesario conocer, si una instancia ya esta disponible en memoria, si será enviada a persistir en el próximo ciclo de acceso. Tres aspectos son considerados aquí: la consistencia de referencias desde objetos transitorios a objetos persistentes, los atributos derivados y

reglas de integridad complejas. En todos los casos puede ser necesario atrapar las señales o eventos producidos por la actividad de persistencia.

El cierre de persistencia de un objeto no incluye los objetos que apuntan a ese objeto raíz del cierre. Objetos transitorios que apuntan a objetos persistentes que podrían no estar en memoria, así se plantea la validez de esas referencias que de alguna forma deben ser controladas. Una solución podría ser emplear referencias inteligentes o proxies por doquier.

Los atributos derivados son resultado de un cálculo basado en los valores de los atributos de una o más instancias, que pueden necesitar que los atributos no persistentes deban ser inicializados justo después de la recuperación de los datos. También podría ser necesario que el valor de un atributo deba ser obtenido justo en instante de consolidar en el depósito de datos, como ejemplo, el caso de las claves primarias automáticas generadas por el mecanismo de persistencia.

Utilizar reglas de integridad escritas una sola vez y evitar una programación defensiva requiere habilitar mecanismos que se activen ante determinadas operaciones o cambios en el ciclo de vida de las instancias persistentes. Por ejemplo cuando se pretende eliminar una instancia podría ser necesario verificar si es posible realizar la operación. Estas y otras situaciones similares muestran que sería adecuado disponer de la funcionalidad de poder asociar acciones a determinadas transiciones del ciclo de vida de una instancia persistente. Monitorizar reiteradamente el estado asociado a una instancia, no es la solución idónea. Una solución basada en los patrones de diseño, los métodos plantilla, observador o comando [13], resultan una aproximación eficiente donde los objetos interesados en el control de determinados eventos implantan cierto protocolo que indica a los servicios de persistencia, quienes deberán ser notificados de la ocurrencia de los sucesos oportunos. *Los servicios de persistencia podrían ofrecer la capacidad de notificar los eventos relativos al ciclo de vida de las instancias persistentes manejadas por el servicio.*

Medidas de productividad y rendimiento

Un aspecto opcional pero interesante desde el punto de vista de la ingeniería, es la capacidad de tomar medidas sobre el rendimiento y la productividad. Estas medidas tienen interés en las fases de optimización de las aplicaciones e incluso de diseño y ajuste del propio servicio de persistencia. Tiempos de transferencia y respuesta, sobrecarga en la comunicación, número de peticiones por minuto a los servicios los de datos, nivel de concurrencia, números de operaciones por petición, tasas de acierto y fallos de caché,... Es posible definir toda una pléyade de mediciones, que sirven al propósito de cuantificar el coste y rendimiento reales, obtenidos con los diseños puestos en práctica. Debemos considerar que el papel desempeñado por los servicios en transparencia es activo, con numerosas operaciones internas, no visibles desde el exterior de los servicios, que también tiene sentido medir. *Los servicios de persistencia podrían aportar estadísticas sobre su funcionamiento para posibilitar el análisis del funcionamiento de las soluciones implantadas.*

La eficiencia de almacenamiento

Una cuestión relacionada con la correspondencia entre objetos y soportes de datos es el almacenamiento. Tener separados los datos y los objetos permite separar la responsabilidad dejando a cada parte que cumpla su función de la mejor manera posible. Es más, es un objetivo esencial conseguir la independencia de datos protegiendo a las aplicaciones contra cambios en la estrategia de acceso y almacenamiento [9]. A primera vista, no parece necesario que los programas se ocupen de dónde o cómo son almacenados

físicamente sus objetos, los mecanismos de persistencia ofrecen la gestión del almacenamiento, al menos, los sistemas de bases datos relacionales y las de objetos.

El almacenamiento determina la velocidad de acceso, recuperación y actualización de los datos, y los límites de concurrencia, volumen y espacio. Parámetros de almacenamiento que son críticos para que las aplicaciones obtengan adecuados tiempos de respuesta. Los principales sistemas de bases de datos utilizan una gestión del almacenamiento compleja y sofisticada, que cubre normalmente la necesidad de satisfacer la demanda de prestaciones, pero también tiene sus límites. Habitualmente existe una unidad lógica básica de almacenamiento que será posible gestionar desde las aplicaciones, en los sistemas relacionales las tablas e índices, en los de objetos colecciones, páginas (Gemstone, Versant), particiones (Versant) o contenedores (objectivity), y en otros archivos, documentos, etc. Estas unidades permiten a las aplicaciones escoger dónde guardar sus datos, para satisfacer sus requerimientos sin comprometer la independencia de datos.

No es una cuestión baladí la elección e implantación de una estrategia de almacenamiento, ni tampoco un requisito sólo de grandes y críticos sistemas. En la práctica son numerosos los ejemplos que utilizan una gestión dinámica del almacenamiento en tiempo de ejecución. Por ejemplo, en los ancestros de la aplicaciones de gestión, la contabilidad y facturación era, y es, habitual que en función de la fecha de trabajo o ejercicio, los registros irán a distintos archivos o tablas; las aplicaciones con archivos de movimientos locales que posteriormente son consolidados contra servidores centrales; aplicaciones con necesidad de un elevado número de transacciones que balacean la carga sobre diverso almacenamiento, como el ejemplo visto del supermercado. Una ilustración del tratamiento al vuelo del almacenamiento son los contenedores dinámicos de Objectivity, las particiones lógicas de Versant y las tablas divididas o particionadas de Oracle.

Sería deseable que todo servicio de persistencia permitiera administrar el esquema de persistencia en tiempo de ejecución y disponer de la facilidad de interactuar con una interfaz de gestión lógica del almacenamiento de los mecanismos soportados. Resolver la cuestión con las bases de datos relacionales será fácil, en principio bastará con lanzar sentencias SQL apropiadas, para lograr el almacenamiento deseado; la dificultad mayor estriba en las diferentes interfaces empleadas por otros mecanismos de persistencia que ofrecen interfaces de programación propias, como Versant y Objectivity, compatibles con ODMG 3.0.

La eficacia de acceso a los datos depende, también, en gran medida de la forma en la que la correspondencia objetos-datos sea establecida y del patrón de acceso a los datos, factores determinantes para el rendimiento.

En definitiva, *los servicios de persistencia podrían ofrecer la facilidad de ajustar dinámicamente los parámetros de funcionamiento interno que afectan al rendimiento.*

Trazabilidad

La trazabilidad es una cualidad deseable para todo diseño de programación, de manera que, permita analizar la corrección del funcionamiento, y asistir en la resolución de los defectos. El empleo de bitácoras de actividad, los famosos log, es habitual a todo tipo de librerías y servicios de datos. *Los servicios de persistencia podrían facilitar la creación de registros de actividad que permitan asistir en las tareas de prueba y corrección de las soluciones implementadas.*

Neutralidad respecto al lenguaje

Trabajando con bases de datos relacionales, es posible emplear distintos lenguajes de programación para tratar las mismas tablas, es habitual encontrar soluciones que combinan más de un lenguaje. En cambio, la posibilidad de que objetos creados con un lenguaje de programación sean recuperados para ser usados con otro lenguaje, es una característica avanzada, poco común entre los productos revisados. Crear objetos hacerlos persistir con un lenguaje, para posteriormente recuperar estos y modificar su estado desde otro lenguaje, no es practica habitual, salvo en el mundo OMG CORBA. Solo algunos sistemas de bases de datos de objetos, a los que se ha tenido acceso, ofrecen cierta neutralidad entre C++, Java y Smalltalk pero con limitaciones.

El interés de esta neutralidad, es aprovechar la ventaja, que puede suponer elegir el lenguaje más adecuado para cada tarea, la arquitectura dirigida por el modelo (Model Driven Architecture, MDA) y las soluciones CORBA son un claro ejemplo. No obstante, el tema es muy complejo quedando muy lejos del propósito de este documento.

Sin duda hay numerosas cuestiones más que todavía no han sido tratadas. Las que aquí han sido expuestas, parecen ser suficientes para establecer un marco de análisis que permita estudiar las cualidades del estándar que va a ser objeto de este texto. En el tintero quedan por ejemplo:

- Ámbito de identificadores. Los servicios de persistencia como mediadores deben manipular identificadores con ámbitos distintos, a un lado los lenguajes de programación, al otro los servicios de datos, cada uno con sus reglas y restricciones ámbito y uso.
- Movilidad entre plataformas. La capacidad para manejar objetos persistentes desde distintas plataformas. El formato de almacenamiento, entre otras características necesarias y deseables debería permitir cierta neutralidad, respecto a la plataforma donde los datos son almacenados, posibilitando el acceso y manipulación sobre otras plataformas. El mundo OMG y sus soluciones pueden enseñarnos bastante al respecto.
- Neutralidad respecto del fabricante. Sería ideal que los objetos hechos persistentes sobre un producto de un fabricante pudieran ser rescatados por productos de otros fabricantes, facilidad que puede ser obtenida en las bases de datos relacionales con SQL. Esto evitaría una dependencia excesiva de un solo proveedor de servicios de persistencia.

Todos los requisitos expuestos contribuyen favorablemente al rendimiento, mantenimiento y coste de los servicios de persistencia que cubran los requisitos expuestos.

1.2.2. Dimensión organizacional

Estamos interesados, en conocer como la organización de la empresa sería afectada por la incorporación de un servicio de persistencia ortogonal. Cuestiones tales, como qué piezas de la cadena de valor serían alteradas, cambios en las relaciones de los actores implicados, qué nuevas relaciones surgen, etcétera. Obviamente cada organización introducirá los cambios que estime oportuno, si bien, desde aquí intentaremos anticipar con una visión general, donde incide la incorporación de un sistema de persistencia con soporte de transparencia de datos. La adopción de un servicio de persistencia impacta en la producción de programas. Afecta tanto a las entidades que producen programas, bien para el mercado o su propio consumo, y las empresas que los consumen. Las organizaciones productoras deberán adaptar su proceso de desarrollo, como veremos. Las organizaciones

no productoras, tendrán que revisar sus procesos de contratación y gestión de sistemas de información.

La producción de programas, se sustenta en el trabajo de personal muy cualificado y entregado a su labor. La producción industrial de programas, no puede ser entendida, como la aplicación mecánica de trabajo no cualificado y rutinario, tareas repetitivas que no requieren de esfuerzo intelectual. [3]. Un servicio de persistencia, como ocurre con cualquier otra librería o utilidad para la programación, no va a permitir producir programas correctos con personal, sin la formación y preparación adecuadas.

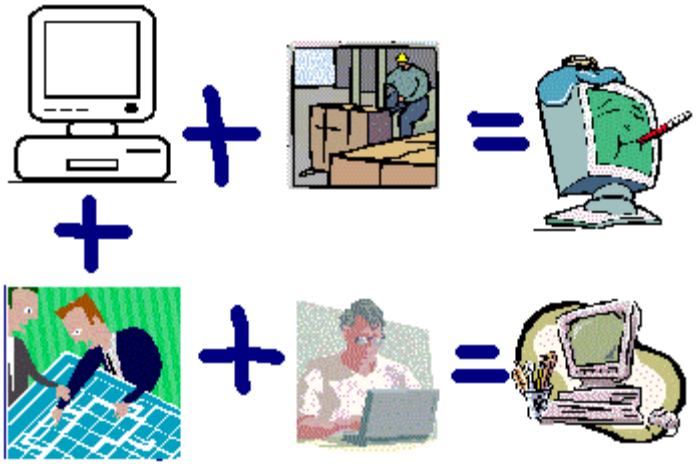


Figura 6 La programación requiere cualificación

Algunos autores estiman que un 30% del código de las aplicaciones esta dedicado a tratar la persistencia, siendo esta una de las causas, que impulsa la incorporación de servicios de persistencia, evitar el reiterado código necesario, para guardar y recuperar objetos transformando datos. La mera incorporación, como utilidad de programación, de un servicio de persistencia podría reducir el número de líneas de código efectivo, pero no es suficiente, el proceso completo de desarrollo de sistemas de información debería ser revisado. Con las medidas adecuadas, los costes de desarrollo y mantenimiento deberían mejorar. Tener menos código facilita conseguir código de calidad. Dependiendo de los sistemas de información de que se trate y de la cultura de la organización, el reuso y sus beneficios podrían aumentar.

Emplear un servicio de persistencia con transparencia de datos, permite desacoplar los objetos de los esquemas de bases de datos. Aumenta la flexibilidad ante cambios en los requisitos de las aplicaciones y de los sistemas de almacenamiento. Al separar, es posible posponer decisiones de diseño que no afectan a las partes. Posponer la decisiones, facilita profundizar en el conocimiento del problema y de la solución, adoptando finalmente las decisiones más acertadas. Nunca posponer decisiones, debe significar eludir los problemas que deben ser resueltos.

La adopción de un servicio de persistencia de objetos conduce a que, el esquema de datos disminuya de importancia en el diseño de las soluciones frente al modelado de objetos.

A nivel de papeles o figuras dentro del equipo de desarrollo de sistemas de información, podría ser efectiva una nueva división de las responsabilidades. Al habitual reparto entre el programador de objetos y el administrador o diseñador de bases de datos, se incorpora un tercer personaje, que asume la responsabilidad de la gestión de los esquemas de persistencia, el experto en persistencia. Con una denominación u otra, la responsabilidad es dividida entre quienes tratan con objetos exclusivamente, los que están

ocupados con los almacenes de datos de la empresa y, ahora, aquellos cuya preocupación es, establecer la mejor correspondencia entre objetos y los datos de la empresa.

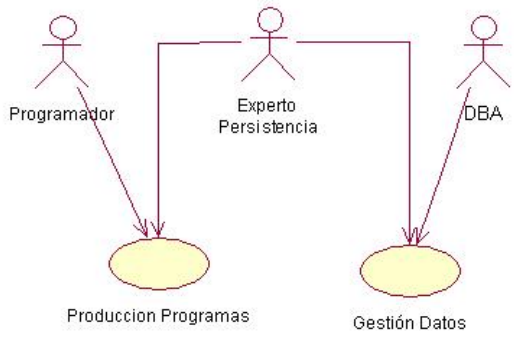


Figura 7 Papeles en desarrollo con persistencia

Desde un punto de vista de la división del trabajo y de la cadena de valor, esto supone una especialización en actividades separadas; la especialización facilita crecer acumulando conocimiento y experiencia sobre la actividad asignada. También, puede suponer una mejora del reuso del conocimiento y del personal, las personas adquieren mayor destreza, que redunda en mayor calidad y menores tiempos de desarrollo, tanto por la habilidad adquirida, como al posibilitar la concurrencia de tareas, si los papeles son desempeñados por personas diferentes.

Pero la especialización tiene sus inconvenientes, el principal inconveniente es, el conflicto de intereses entre el desarrollador y el administrador de bases de datos. La solución pasa, por un mayor conocimiento de ambas partes de las tareas del otro, y una mejor comunicación, para ello una posibilidad es, la rotación en los papeles asignados, transcurrido un tiempo adecuado desempeñando una actividad, se emplaza a asumir el papel del otro. Esto conduce a un mejor reuso del conocimiento, una mejor comunicación, a reducir los conflictos de intereses, pues los objetivos convergen, mayor adaptabilidad y tolerancia a cambios en las necesidades de la cadena de valor.

Los esquemas rígidos de asignación de funciones a personas, no terminan de funcionar bien con las tecnologías orientadas a objetos, las metodologías y métodos modernos como los presentados en [25] [18] y [3] [15], apuestan por fomentar la figura del ingeniero que ocupa distintos papeles en distintos momentos, según su perfil personal y destrezas.

La adopción de un servicio de persistencia no es asunto exclusivo de los programadores. Alcanzar la correcta implantación de un servicio de persistencia, en el proceso de producción de programas, requiere de que todos los actores implicados en la cadena de valor, conozcan las capacidades y limitaciones del servicio de persistencia adoptado, con un conocimiento adecuado a su posición en la cadena de valor.

Para las empresas productoras de programas un servicio de persistencia debería permitir, desde el punto de vista organizativo:

- Posibilitar un aprendizaje rápido.
- Reducir el número de líneas de código efectivo para llevar a cabo la persistencia.
- Introducir mayor flexibilidad en el proceso de producción: Centrar el desarrollo en el modelo de objetos. Desacoplar las actividades relacionadas con

los cambios en el esquema físico, el esquema de persistencia y el modelo de clases.

- Mejorar la comunicación dentro del proceso de producción y ayudar a reducir los conflictos entre programadores, analistas y administradores.

Estos objetivos no tienen nada que ver con cuestiones técnicas, en un principio, pero la documentación, las herramientas y la notación, aportadas con un servicio de persistencia, ayudan, o por el contrario dificultan, conseguir la implantación de un nuevo servicio. En este sentido, la documentación de un servicio de persistencia, debería ejemplificar distintas posibilidades de integración en el proceso de producción de programas, documentar las estrategias de integración que sirvan de referencia o guía a las organizaciones.

Ahora veamos, que beneficio pueden obtener las organizaciones que son clientes de las anteriores, consumidores de programas. Para las organizaciones clientes de los desarrolladores de sistemas de información, la incorporación de un servicio de persistencia debería mejorar la flexibilidad ante cambios de plataforma tecnológica y proveedor, conseguir así, menores costes de desarrollo, implantación y mantenimiento.

La adopción de un servicio de persistencia adecuado, permitiría dialogar con los desarrolladores en términos propios del negocio, la comunicación entre usuarios y desarrolladores debería mejorar. Se logra una mejor comunicación al reducir el salto semántico entre el lenguaje del negocio y el de los programadores, esto ayuda a mejorar la especificación y mitigar los errores de interpretación. Un buen servicio de persistencia, basado en estándares reales, podría favorecer disponer de un mayor número de proveedores. El choque con la cultura presente en la organización, sobre bases datos relacionales y la reticencia a considerar otras alternativas no relacionales, es una dificultad añadida para la integración de un servicio de persistencia, que puede llegar a ser un escollo insalvable.

La cuestión no es tan simple como comprar un CD y ya está. Los cambios de organización y de tecnología en las empresas, no se pueden efectuar de un momento para otro, sin más. Introducir JDO afecta al mismo corazón de la fabrica de programas. Es un tema estratégico. Y como tal, requiere un cuidadoso planeamiento, de forma y manera que minimicé los riesgos, al mismo tiempo, el cambio de ser rápido, para evitar perdida de competitividad e incurrir en costes de oportunidad importantes. La transición tiene que ser guiada y cuidadosamente manejada. El cambio debe ser evolutivo, pero resulta finalmente radical.

No hay muchos misterios al respecto. El sentido común, la cultura de la organización, el número y preparación del personal afectado, son los elementos que ayudan a llevar a cabo un planteamiento acertado. El consejo típico es comenzar con un pequeño proyecto piloto de costes reducidos, con un equipo reducido pero escogido de miembros de la organización, aprovechar la experiencia para difundirla y aplicarla en sucesivas fases al resto de la organización, acometiendo cada vez proyectos más importantes con el personal más experimentado, que a su vez contribuye con su conocimiento a mejorar el proceso y formar nuevos programadores en JDO. Creo personalmente, que el cambio debe efectuarse en tres fases de seis meses cada una, como máximo.

1.2.3. Dimensión Económica

Los aspectos técnicos son esenciales, pero son los factores económicos los que determinan éxito de un nuevo estándar. Una tecnología imperfecta en un mercado en funcionamiento es sostenible; una tecnología perfecta sin mercado, se desvanecerá [31].

Echando la vista atrás solo unos pocos años, podemos recordar el fracaso en la adopción de tecnologías calificadas como la mejor del momento, basadas en el consenso entre un importante número de los principales productores. Recordemos la pugna Opendoc frente COM, claramente ganada por el segundo. La estandarización supone un elemento esencial en el abaratamiento de los costes de producción, siempre y cuando la implantación del estándar sea efectiva. La estrategia de divulgación y de introducción en el mercado son claves para alcanzar la aceptación efectiva.

El Gartner Group estimaba a principios del 2001 que, los mercados para la tecnología Java muestran las reglas siguientes [48]: El mercado se mueve hacia usuarios que adoptan una tecnología cuando esta ha demostrado su utilidad, cuya elección viene estratégicamente planificada, y a la vista de la experiencia de otros. Estos usuarios reclaman nuevos métodos, nuevas herramientas y estrategias de integración.

- Las decisiones de compra están dirigidas más, a soluciones con respaldo que al mejor producto. Soluciones completas de un solo proveedor y de los grandes canales maestros de distribución.
- J2EE ha llegado a ser la plataforma para el lado servidor de la soluciones Java.
- Las nuevas herramientas centrarán su interés en los modelos de desarrollo de aplicaciones para los nuevos emergentes e-servicios. Estas herramientas deben permitir crear soluciones reales con desarrolladores poco cualificados. La falta de desarrolladores cualificados es y será la principal dificultad para adopción de Java entre las organizaciones que ofrecen servicios en la Red.

Los párrafos anteriores describen un escenario del mercado donde JDO deberá mostrar su utilidad ante la cautela de las organizaciones para elegir entre JDO y la competencia de Microsoft .NET.

La realidad económica de la empresa se impone sobre los criterios meramente técnicos, la inversión en servicios de persistencia debe ser rentable. Si las cuentas no cuadran, difícilmente las empresas invertirán en persistencia Java.

El retorno a la inversión en servicios de persistencia debería proceder de la reducción efectiva de los costes de producción, mantenimiento y amortización de los sistemas en los que son aplicados. A menores costes, manteniendo economía y productividad, mejora el retorno de las inversiones en desarrollos de sistemas de información. También, el riesgo en las inversiones sobre TIC, así construidos debería ser menor, habida cuenta de la mayor oferta y de la flexibilidad que debería aportar un servicio de persistencia estándar.

1.3. Resumen

Este primer capítulo ha presentado qué se entiende por persistencia de objetos, algunos de los conceptos más importantes en torno al problema de hacer perdurar los objetos más allá de la ejecución de las aplicaciones que los crean.

La idea fundamental es que la capacidad necesaria para persistir los objetos debe ser lo más independiente del lenguaje de programación y del medio utilizado para soportar el estado persistente de los objetos que perduran.

La dificultad de incluir la capacidad de persistir como una extensión del lenguaje de programación, y la tendencia del mercado, es solventar la persistencia mediante la utilización de servicios de persistencia de objetos bajo la forma de framework de persistencia.

Han sido concretado una serie de requerimientos debe satisfacer un servicio de persistencia de objetos, que ofrezca transparencia de datos, persistencia por alcance, sea ortogonal a tipos y mecanismos de persistencia, que asegure la integridad transaccional y referencial de los objetos persistentes y sus relaciones, adaptable a diferentes contextos y arquitecturas, y que logre un rendimiento adecuado.

En el plano de la empresa, la adopción de un servicio de persistencia implica cambios en la producción de programas con nuevas figuras, responsabilidades y actividades, pero siempre considerado que la inversión en servicios de persistencia debe ser viable y rentable.

Capítulo 2
EQUIVALENCIA CLASE - TABLAS RELACIONALES

Este trabajo señala en su primer capítulo, que todo servicio de persistencia de objetos, debe utilizar las bases de datos relacionales, para recoger los estados de los objetos persistentes La cuestión de cómo hacer corresponder objetos en relaciones y viceversa, siempre ha sido de objeto interés de estudio, análisis y propuesta de solución en numerosos trabajos de merecido reconocimiento, algunos de esos trabajos son la base de este capítulo, las propuestas de A. Keller, S. Ambler, C. Priese, J.Rumbaugh W. Keller y R.G.G.Cattell, sustentan las ideas recogidas a continuación.

Éste capítulo describe las ideas fundamentales de la solución al problema de la correspondencia clase-tabla, sin entrar en profundidad, pues necesitaría de todo un proyecto fin de carrera. Las técnicas de correspondencia presentadas, serán ilustradas con diagramas y listados de ejemplos prácticos.

2.1. Falta de Correspondencia Clase-Tabla

Clase y tabla son conceptos diferentes, utilizados para modelar las realidades, que después manejan las aplicaciones. Una clase especifica los atributos y el comportamiento de la realidad que modela, define objetos. Una tabla especifica los datos de la entidad que representa, concreta una relación entre valores de los dominios de los datos escogidos. Ambos capturan características de interés para la aplicación, pero la tabla ignora aquellas características ligadas al comportamiento en ejecución, cosa que si modela la clase. Para la clase, la persistencia es una característica más, posiblemente independiente, en cambio, la tabla toma su sentido esencialmente, de servir al propósito de persistir y compartir los datos que modela. No hay por tanto, una equivalencia exacta y única entre ambos conceptos. Tampoco a nivel de la implementación es posible una equivalencia directa entre los tipos de datos sobre los que se construyen clases y tablas, una clase aprovecha el sistema de tipos primitivos y estructurados, provisto por el lenguaje de programación usado; una tabla no puede utilizar un vector o árbol como atributo, necesita de crear otras relaciones (tablas o vistas), con las que modelar estos atributos complejos, habitualmente su sistema de tipos es diferente de aquellos presentes en los principales lenguajes de programación, y no permite utilizar estructuras de datos como dominios de los atributos (hoy SQL99 sí soporta tipos estructurados, como atributos). En definitiva, existe una dificultad o falta de correspondencia entre ambos modos de representar las entidades que manejan nuestros programas, que deberemos ser capaces de resolver. La tarea de crear una correspondencia entre un modelo de objetos y un esquema de relaciones requiere de establecer una equivalencia entre:

- Identidades, la identidad de un objeto y su representación en las tablas.
- Clases y relaciones.
- Relaciones entre las clases y su equivalencia en relaciones entre tablas.
- La herencia sobre tablas.

Todo ello a simple vista parece ser sencillo, directo y sin complicaciones, pero resulta una tarea difícil cuando se requiere conseguir eficiencia, alto rendimiento,

flexibilidad al cambio, facilidad de mantenimiento y se requiere dar respuesta a la herencia y el polimorfismo, conceptos no soportados por el modelo relacional. La cuestión puede ser planteada desde un enfoque dominado por uno de los extremos de la correspondencia, los objetos o por los datos, resultando en modelos que no se ajustan a los principios clásicos de diseño e implementación establecidos en lado dominado. La idea clave en la correspondencia entre clases y tablas, es conseguir un esquema simple, sencillo y claro, los esquemas de traducción complejos sin un cierto patrón sencillo, introducen rigidez, falta de flexibilidad al cambio, a la evolución, y es freno a nuevas mejoras.

La creación del esquema de correspondencia, puede ser realizado desde dos perspectivas, una centrada en el modelo de datos, donde las tablas dirigen el proceso de creación de la correspondencia, buscando generar las clases que se amoldan a cada tabla; y otra perspectiva, enfocada desde el modelo de objetos, que impulsa encontrar un modelo de tablas que sea el más apropiado para representar de un forma eficaz los objetos.

En mi modesta opinión, experiencia, como partidario de la orientación a objetos y desde la perspectiva de este trabajo, que es utilizar las bases de datos para almacenar los estados de los objetos, el modelo de objetos de dominar el esquema de persistencia basado en un modelo de entidad-relación, se debe procurar no forzar cambios en el modelo de objetos, por el hecho de emplear una base de datos relacional para persistir los objetos, pero al mismo tiempo, es necesario sacar partido de las capacidades ofrecidas por el modelo relacional, para conseguir un rendimiento e integridad oportunos. En la práctica, un modelo puro de objetos, trasladado a un esquema de relaciones, suele casar con dificultad y resulta en un pobre rendimiento; el enfoque contrario, centrado sobre el modelo entidad-relación, favorece el rendimiento concurrente y es más fácil de convertir en clases 0, pero suele carecer de ventajas de un buen diseño orientado a objetos; es fácil caer en clases vacías de funcionalidad, que más bien representan registros con nula reusabilidad. Los modelos de tablas creados a partir de un modelo de objetos, pueden conducir a modelos relacionales poco útiles, desde el punto de vista de los usuarios finales, pero gracias a la potencia expresiva de SQL y la vistas, normalmente, se logra construir una visión de los datos más adecuada para el usuario final, sin grave penalización para el rendimiento. Son muchos los que consideran mejor, partir de una representación relacional por su facilidad para ser manipulada y entendida, además, de su mejor rendimiento en concurrencia, para después, en base a las tablas crear un modelo de objetos acorde. Entre quienes proceden de la cultura de la programación orientada a objetos, existe una postura de considerar una aproximación oportunista, donde se genera el mejor esquema de relaciones adecuado, a un modelo de objetos, salvo que se parta de un sistema legado, con un esquema relacional previo a continuar, en cuyo caso es mejor seguir un enfoque dirigido desde el modelo esquema relacional. En una primera aproximación, para establecer una correspondencia entre clases y tablas podemos partir de equivalencias de la Tabla 1.

Tabla 1 Correspondencia Modelo Objetos – Esquema Relacional

Modelo de objetos	➔	*Modelo Relacional*
Identidad	➔	Clave primaria
Clase	➔	Relación (Tabla, Vista)
Instancia	➔	Fila
Atributo de instancia persistente con dominio un tipo básico o primitivo	➔	Columna
Atributo persistente Referencia a objetos con tipo conocido	➔	Clave Ajena
Atributo persistente tipo colección	➔	No hay equivalencia directa, debe ser una relación, si la cardinalidad es (m:n), entonces una entidad de asociación; Si (1:n), entonces atributos en las relaciones correspondientes.
Atributo persistente de Clase	➔	Una tabla con las características comunes a todas las instancias de una clase.
Herencia	➔	Una relación agregada con todos los atributos de instancia de la jerarquía herencia completa o múltiples tablas

A la vista de la tabla anterior, no parece que la actividad de definir la correspondencia sea algo complejo, pero como se verá, en apartados siguientes y con la ilustración de los propios ejemplos, quedará patente su dificultad, por ejemplo, son habituales los casos, donde una sola clase necesite más de una tabla para persistir sus instancias, que sea necesario añadir columnas para dar soporte al bloqueo optimista, o que no existan un conjunto de atributos que pueda ser clave primaria.

2.2. Identidad

La equivalencia entre identidades de objetos y relaciones, es la correspondencia de identidad persistente a claves primarias. El identificador persistente de los objetos puede ser, o una característica ajena a los objetos construida y gestionada por el servicio de persistencia, o bien, un conjunto de atributos de los objetos persistentes que satisfacen la singularidad de clave primaria, esto es, identificar a cada objeto.

Debemos mencionar dos aspectos, uno la elección de claves primarias, el otro es la necesidad de conocer el tipo de los objetos polimórficos. Los objetos concretan su identidad mediante identificadores internos al sistema, carentes de significado para el usuario; en lado relacional, hoy por hoy, también los principales SGBR emplean identificadores de filas automáticos e internos, pero habitualmente, las claves elegidas tienen significado, son dependientes del valor de unos o más atributos con sentido para el usuario final, lo que ante cambios de la estructura de estos atributos, el esquema y código de los programas tienen que ser modificados. Desde aquí, se recomienda la utilización de claves primarias sin significado en los esquemas relacionales, utilizando secuencias, contadores o tickets. Conservar el tipo de las instancias, su clase, junto a los datos de estas instancias es un requisito, visto en el primer capítulo, que puede ser satisfecho con distintas alternativas, por ejemplo, una tabla utilizada como diccionario, una columna utilizada como selector, o en la propia identidad embutida en la clave primaria.

Un enfoque bastante práctico y que facilita un nivel rendimiento adecuado, para lograr la identificación en un gran número de casos, útil para ambos lados de la equivalencia es, el uso de una clave primaria automática, que incluya un indicador de la clase del objeto correspondiente. Esto permite aprovechar la potencia de la integridad referencial de las bases de datos relacionales, utilizando una sola columna, evita accesos adicionales a los datos para determinar el tipo, no fuerza la introducción de campos adicionales, permite un mayor reuso, mejor mantenimiento e introduce menos complejidad en el código de los programas, aunque conlleva una ligera sobrecarga por extraer e insertar la señal de tipo en el campo de clave primaria. La clave automática podría ser un contador obtenido, a partir de una tabla de contadores común, o una secuencia generada por el SGBDR, como en Oracle, SqlServer, SqlBase, o PostgreSQL, o mediante alguna formulación hashing. En JDO, el modelo de identidad por aplicación da soporte a la identidad por claves primarias, exigiendo la inclusión de nuevas clases que representan estas claves primarias verificando ciertas propiedades.

En el ejemplo del banco Figura 1 con las clases Cliente, Cuenta y Movimiento, desde un punto de vista relacional, sería posible establecer las siguientes equivalencia para claves primarias:

Tabla 2 Claves candidatas en el Banco

Clase	**Atributo**	**Tabla**	**Columna**
Cliente	Nif	Cliente	Nif

Cuenta	CodigoCuenta	Cuenta	CodigoCuenta
Movimiento	-	Movimiento	-

La gestión de identidad con claves primarias haciendo equivaler atributos a columnas, necesita del código adecuado en el lado de los objetos, que podría ser transparente para el programador. En general, introducir claves primarias en un modelo de objetos, requiere de cierto esfuerzo de programación, que es mayor a utilizar en un modelo de identidad gestionada por el mecanismo de persistencia. La práctica totalidad de los productos revisados, que consiguen persistir objetos en tablas implantan, los aspectos mencionados de una identidad, basada en una secuencia o contador, y de conservar el tipo de los objetos. Veamos cómo se plasma la identidad persistente de la clase Cliente.

```
public class Cliente
    implements Serializable
{
  /**
   * Identificación del cliente
   */
  protected String nif; ….
```

```
CREATE TABLE `cliente` (
 `CLASS` varchar(255)default NULL,
 `OID`   bigint(20) NOT NULL default '0',
 …
 `NIF`    varchar(255) default NULL,
…
  PRIMARY KEY  (`OID`),
…
# Generador de identidades
CREATE TABLE `sequence` (
  `ID` tinyint(4) NOT NULL default '0',
  `SEQUENCE_VALUE` bigint(20) default NULL,
  PRIMARY KEY  (`ID`)
) TYPE=InnoDB;
```

Listado 12 Identidad de la clase Cliente en tablas con MySQL.

La correspondencia escogida en este caso, se vale de un tabla contador para generar el valor de la columna OID, que es la clave primaria y el campo CLASS para conservar el tipo. Obviamente, CLASS hace falta para cubrir el polimorfismo. Si es seguro que la clase Cliente no formará parte de una relación de herencia, Cliente nunca usará de polimorfismo, entonces la columna CLASS podría ser eliminada. Como vemos el `nif`, es simplemente un atributo por el que consultar, aunque con índices en el RDBMS, se podría forzar que existan valores únicos sin afectar al código Java.

Otra cuestión que se revela interesante la Tabla 2, es la ausencia de atributos que puedan ser clave primaria para la clase Movimiento. Esta situación es normal en los modelos de objetos, lo cual es otro argumento más, para utilizar claves primarias sin significado, gestionadas por el RDBMS como el identificador OID del ejemplo, que obtiene su valor de un contador basado en la tabla sequence.

2.3. Conversión de clases en tablas

La correspondencia más directa y simple, es hacer corresponder cada clase con una tabla, y cada objeto, con una fila de la tabla, éste es el punto de partida para la construcción de la correspondencia entre un modelo de objetos y un esquema relacional. Los trabajos de A. Keller [41], A.Brown [6], S.Ambler [32], enseñan que la correspondencia entre clases y tablas debe ser directa, de una clase a una tabla, y viceversa, esta fórmula es correcta para la

mayoría de los casos, porque planteamientos más complejos, pueden implicar falta de rendimiento y problemas en las actualizaciones.

La tabla anterior Tabla 1, es punto de partida para efectuar el proceso de traducción entre los modelos relacionales y de objetos, donde en principio, a cada clase se le hace corresponder con una tabla, a cada atributo persistente una columna en la tabla de su clase. De esta forma, se construye un modelo de tablas, que inicialmente debería estar normalizado.

Los modelos normalizados, con frecuencia adolecen de bajo rendimiento al recuperar o desgranar los objetos sobre relaciones. Es entonces cuando se debe proceder a desnormalizar los relaciones para mejorar el rendimiento y valorar la oportunidad de introducir cambios como son, modificar los esquemas legados existentes con nuevas columnas, serializar ciertos atributos, objetos completos que forman parte del cierre de persistencia, son almacenados en columnas blob, e incluso dado el caso, hacer persistir atributos fuera de la base de datos, como archivos. Situaciones que complican, el proceso de traducción, tomando decisiones, que deben considerar cómo van a ser accedidos y actualizados los objetos, si el acceso se produce al iterar sobre un conjunto o directamente con la identidad, concurrentemente por varios procesos simultáneos o navegando desde objeto contenedor...

Es importante que los servicios de persistencia permitan planteamientos flexibles al proceso de correspondencia clase-tabla, conjugando el propósito de persistencia, con el acceso correcto a los datos y un rendimiento suficiente. Esta flexibilidad es una de las características, que más distinguen a buen producto para crear esquemas de persistencia clase-tabla.

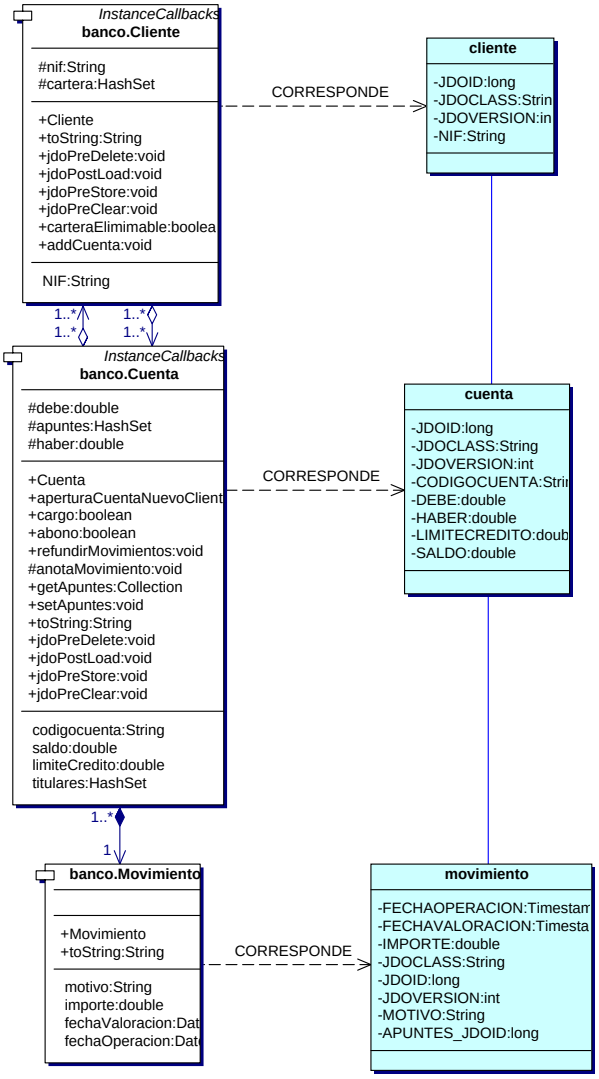


Figura 8 Equivalencia clases del Banco en tablas sin relaciones

En la Figura 8, cada clase corresponde con una tabla, cada atributo corresponde con la columna del mismo nombre, y a la vista esta, que los tipos de datos de las columnas son distintos, pero equivalentes a aquellos de los atributos que representan. Como se verá la regla de uno a uno entre clases a tablas, se debe romper cuando se persiste atributos multivaluados (arrays, vectores,…) y jerarquías de herencia. Otros elementos que atraen la atención son las columnas que no corresponden con los atributos de clase: JDOI, que es el identificador clave primaria y JDOVERSION, agregada para incorporar la funcionalidades de versionado y bloqueo optimista. En la Figura 8 es una correspondencia incompleta, hay atributos donde no aparece su correspondiente columna, son los atributos que definen las relaciones de agregación y composición entre las clases.

2.4. Relaciones de Agregación y Composición a claves ajenas

Las relaciones de agregación y composición entre clases, son traducidas a relaciones entre tablas mediante claves ajenas, así las relaciones entre objetos son trasladas a relaciones entre filas, a través de los valores de las columnas que son clave ajena. Las relaciones entre clases se materializan como atributos de instancia, cuyo tipo es la clase asociada, o un tipo colección con objetos de la clase relacionada. Dependiendo de la

cardinalidad de la relación, del tipo de asociación, agregación o composición, y por último, de la necesidad de consulta y actualización, son aplicadas distintas técnicas, que pueden ser combinadas en una misma clase en su relaciones con otras:

A. Embeber las claves ajenas en las tablas correspondientes a las clases con relaciones 1:1 o 1:n. En una relación de uno a uno, cada tabla añade una columna clave ajena de la otra. Para la relación de uno a varios (Departamento-Empleado), las tablas de los objetos que son agregados (las partes), añaden la clave ajena (clave ajena inversa) a la tabla de la clase que hace de contenedor (el todo), incluso si la clase contenida no tenía un atributo equivalente. El atributo colección de la clase contenedor (lado uno de la relación) no necesita mapeo a una columna. Este planteamiento muestra un rendimiento intermedio frente a otras alternativas, es flexible al cambio de la estructuras y permite toda suerte de actualizaciones y consultas.

B. Utilizar tablas de asociación si las relaciones entre las clases son m:n y 1:n, o cuando es necesario consultar y manipular individualmente cada elemento de la relación, en concurrencia de procesos diferentes. Cada clase en la relación tiene asociada una tabla, y la asociación entre estas clases, se representa sobre otra tabla, que mantiene las claves ajenas a cada una de tablas respectivas asociadas a las clases. Las tablas de asociación son también, la correspondencia adecuada para las clases de asociación [5] y los atributos tipo diccionario (Map, HashMap, etc), donde además de la claves ajenas, son representados los atributos de la asociación o los atributos índice o selector del índice del diccionario. Si relación es una composición, la supresión de una fila del todo, o contenedor, implica la supresión de las filas con clave ajena, la clave eliminada. Se trata de la fórmula más flexible, menos acoplada y sencilla para relacionar varias clases, pero también la de peor rendimiento al navegar, través de la asociación, de unos a otros objetos con reiteración.

C. Los objetos contenidos como parte de otros, son representados en la tabla de la clase que los contiene, o insertando en la tabla sus atributos, o incluyendo un campo especial. Cuando cada clase tiene un atributo del tipo de la otra clase asociada, ambas clases son asignadas a la misma tabla. Un ejemplo sería clase dirección embebida en la clase Persona, que añade a la tabla de PERSONA, los mismos campos que la tabla DIRECCION. Si la relación es 1:n de composición (p.ej. árbol compuesto por ramas), el atributo colección que aglutina los objetos contenidos, podría ser hecho corresponder con una columna blob o similar, que contendrá los identificadores (las claves primarias) de los objetos, o los propios estados esos objetos contenidos. La elección dependerá de que tratamiento de base datos se aplique a los objetos contenidos. Esta alternativa es bastante eficaz respecto al rendimiento, dado que requiere menos accesos, pero es menos flexible al cambio, al reuso y no permite la actualización concurrente sobre los elementos individuales de las colecciones desde otros procesos, presenta la dificultad de mantenimiento al representar una misma clase en diferentes contextos, por ejemplo, la clase Direccion integrada dentro de distintas clases no relacionadas (Proveedor y Empleado).

Insistir que estas alternativas no son excluyentes entre sí, pueden ser combinadas en la definición de la correspondencia para una misma clase. Cada estrategia difiere en su rendimiento, sus posibilidades de recuperación selectiva y actualización concurrente, su nivel de acoplamiento y en la facilidad para ser manejados con SQL con sencillez (el acceso a los campos blob desde SQL no es ni obvio, ni fácil en la mayoría de los principales

sistemas de bases de datos relacionales). Entre los productos revisados los hay que desgranan las colecciones sobre tablas, con una fila por objeto exclusivamente y otros, que permiten además almacenar las colecciones en columnas blob, con gran rendimiento en recuperación o grabación, pero lentos en la localización de un objeto contenido concreto que satisface cierta condición; los productos vistos mayoritariamente aprovechan la capacidad de propagación de las actualizaciones del RDBMS anfitrión en las relaciones de composición, y todos permiten ajustar la correspondencia en base las técnicas expuestas. Veamos aplicadas las técnicas de este apartado sobre el ejemplo del Banco.

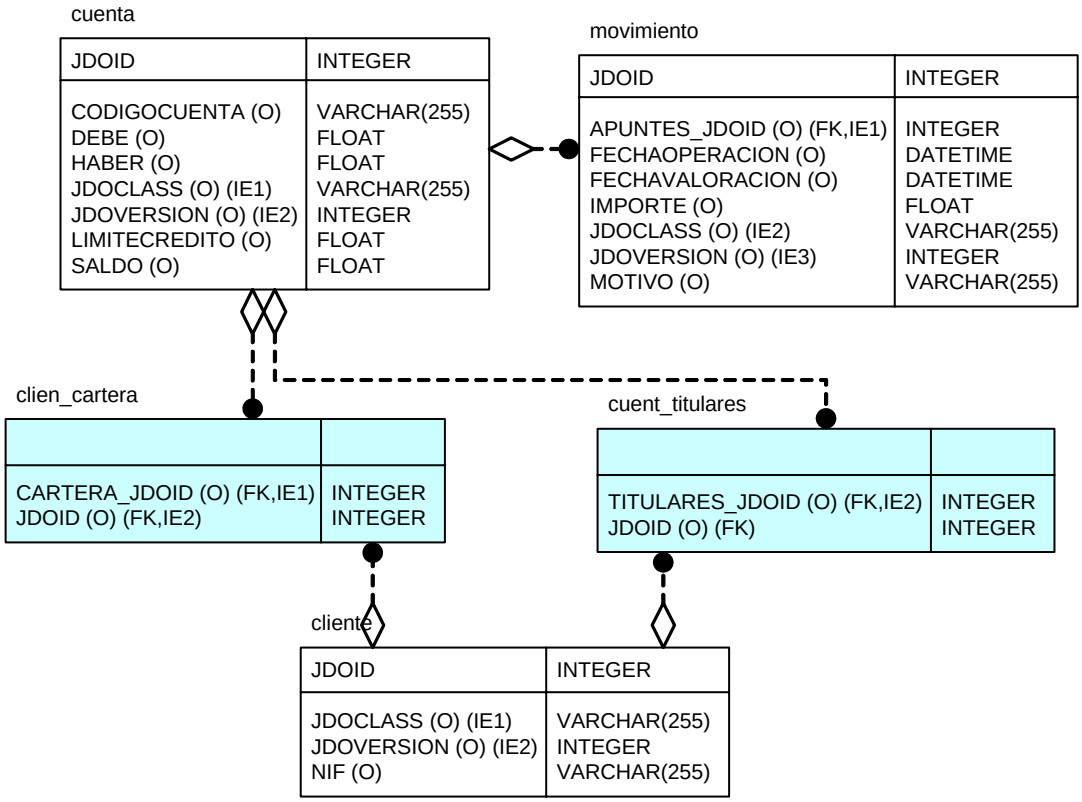


Figura 9 Mapeo de las relaciones de las clases del Banco en RDBMS MySQL

La figura anterior muestra la representación en tablas de las relaciones entre las clases del banco aplicando las técnicas B y A. Las relaciones entre Cliente y Cuenta, expresadas por los respectivos atributos *cartera* y *titulares*, son trasladas a sendas tablas aplicando la técnica del apartado B (entidades azules), porque una misma cuenta puede figurar en la cartera de diferentes clientes, un cliente puede aparecer como titular de más de una cuenta y es posible, que una cuenta esté en la cartera de un cliente no siendo éste titular de esa cuenta. Las tablas de relación han sido nombradas siguiendo la denominación del nombre de tabla de la clase que contiene el atributo más nombre del atributo. Las columnas OID, en CUEN_TITULARES es clave ajena a CUENTA y en CLIENT_CARTERA clave foránea a CLIENTE, no son clave primaria en ambos casos; la columna TITULARES_JDOI es clave ajena de CLIENTE y APUNTES_JDOI, de CUENTA, ambas columnas son nominadas con el nombre del atributo colección en sus respectivas clases de origen. La relación composición de Movimientos en Cuenta, donde cada Movimiento pertenece a una única Cuenta que lo contiene, se traduce con la técnica del apartado A, añadiendo a la tabla MOVIMIENTO, una columna clave ajena a CUENTA, APUNTES_OID, que contendrá el valor de la clave primaría de la instancia Cuenta a la que esta asociado el objeto movimiento.

Falta ilustrar la técnica del apartado C donde una misma tabla aglutina objetos de más de una clase, veamos la traducción a tablas de la clase Persona con su atributo dirección.

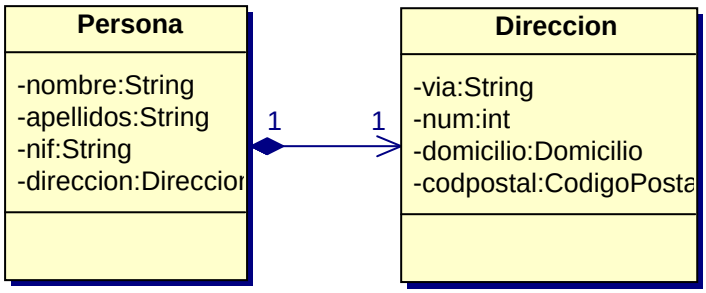


Figura 10 Clase Pesona y Direccion

```
public class Persona
{
  private String nombre;
  private String apellidos;
  private Direccion direccion;
  private String nif;
}


public class Direccion
{
  private String via;
  private int num;
  private CodigoPostal
codpostal;
}
```

```
CREATE TABLE `persona` (
  `CLASS` varchar(255) default NULL,
  `OID` bigint(20) NOT NULL default '0',
  `VERSION` int(11) default NULL,
  `NOMBRE` varchar(255) default NULL,
  `APELLIDOS` varchar(255) default NULL,
  `NIF` varchar(255) default NULL,

  `DIRECCION_NULL` tinyint(4) default NULL,
  `VIA` varchar(255) default NULL,
  `NUM` int(11) default NULL,
  `CODPOSTAL_OID` bigint(20) default NULL,
  PRIMARY KEY  (`OID`),
    FOREIGN KEY (`CODPOSTAL_OID`)
      REFERENCES `codigopostal` (`OID`)
)
```

Listado 13 Corespondencia Clase Persona que embebe instancias del tipo Direccion

El listado anterior enseña como la correspondencia escogida, proyecta la clase *Persona* sobre la tabla del mismo nombre, incluye además los atributos necesarios para almacenar el estado de sus instancias Dirección, junto al resto del estado de Persona, sin necesidad de usar otras tablas. Esta correspondencia de composición donde las instancias Dirección, son embebidas en cada objeto Persona, es independiente de que clase Dirección sea objeto de otras correspondencias.

Otro ejemplo para mostrar la importancia de la serialización y la utilización de campos blobs, cuando se representan objetos complejos y recursivos, se muestra en el diagrama de Figura 11 que definen objetos conectados en una estructura de árbol:

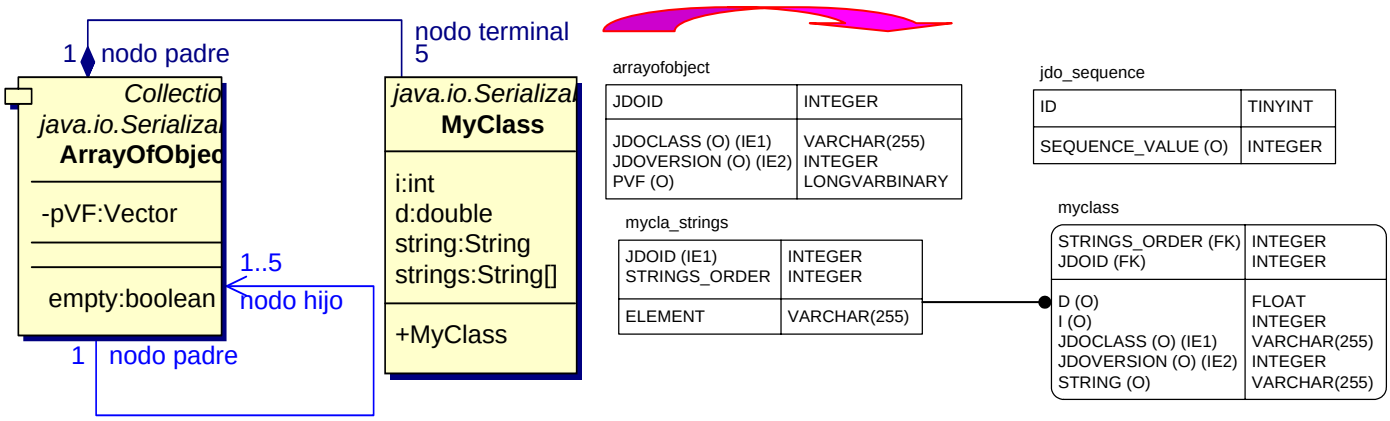


Figura 11 Diagrama de clases de un árbol de cinco niveles y cinco hijos su equivalente E-R.

En la anterior figura se aprecian que el atributo pVF de tipo Vector, es traslado a una columna de tipo Blob, donde se almacena el árbol serializado, que pende del nodo raíz insertado como fila en la tabla arrayobject.

2.5. La Herencia en filas y reunión (join)

Al igual que sucede con las relaciones entre clases, cómo sea definida la correspondencia de la herencia entre objetos, tiene un marcado impacto en el rendimiento, en particular, las sentencias SQL de reunión para obtener los valores de los atributos heredados. Las referencias [16], [32], [41] coinciden en el planteamiento, que tiene su fundamento, en las propuestas de J. Rumbaugh y M. Blaha, en su método OMT [30]. Son propuestas atendiendo a cómo son distribuidos los atributos sobre relaciones, tres modos de correspondencia para representar una jerarquía de herencia sobre tablas,:

A. División Vertical. Cada clase en la jerarquía persiste en una tabla distinta, en correspondencia uno a uno. Los atributos propios (no heredados) son representados sólo en las columnas de tabla correspondiente a la clase que los declara. De esta forma, un mismo objeto podría necesitar persistir sobre más de una tabla, las tablas de sus ancestros y la tabla de su tipo. La clave primaria de la tabla de la clave raíz de la herencia, es la clave ajena en las tablas de los descendientes. Es el modo que se adapta mejor al modelo de objetos, incluida herencia múltiple, y el que mejor mantenimiento presenta. El problema de esta fórmula es, la penalización al rendimiento y a la concurrencia cuando, la jerarquía de herencia es profunda, tres o más niveles, pues son necesarias sentencias SQL con una reunión (join) de todas las tablas implicadas. Este es el error clásico en la correspondencia clase-tabla, cuando los atributos de los objetos están dispersos verticalmente en la jerarquía, o los objetos pueden pertenecer a varias clases simultáneamente. Es apropiado utilizar la división vertical de la herencia, cuando la profundidad de la jerarquía es baja, y los objetos no representan diferentes papeles simultáneamente.

B. División Horizontal. Cada clase concreta terminal (sin descendientes) en la jerarquía, le corresponde una tabla que contiene todos los atributos, heredados y propios, para los atributos redefinidos son añadidas nuevas columnas adecuadas al tipo redefinido. Ofrece un buen rendimiento pues solo hay que acceder a una fila, pero una pobre adaptabilidad al cambio en la estructura. El soporte del polimorfismo requiere el trasvase de datos y conseguir la integridad con herencia múltiple es complicado.

C. División por tipos. Todas las clases de una jerarquía, son proyectadas sobre una única tabla, todos los distintos atributos presentes en la jerarquía tienen su columna, cuando existente atributos cuyo tipo es redefinido en las subclase son introducidas nuevas columnas adecuadas al tipo redefinido. La clase a la que pertenece un objeto, es determinada por una columna que hace de selector o discriminador de tipo. El modelo de identificación propuesto más atrás, sirve para determinar el tipo o tipos actuales de un objeto. El rendimiento es elevado, una fila o un objeto. Permite el soporte de herencia múltiple con la codificación adecuada.

Veamos un ejemplo aplicado de las técnicas descritas de correspondencia de la herencia. El siguiente diagrama de clases, muestra la relación de herencia será traducida con cada una de las técnicas expuestas, ayudando a ilustrar el criterio para elegir la elección más adecuada en la práctica.

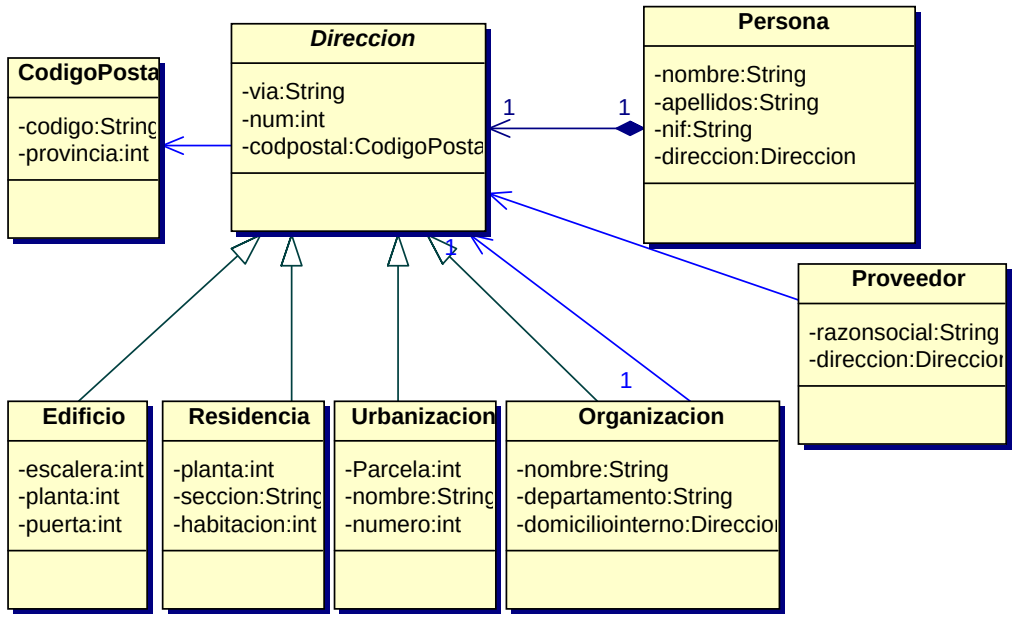


Figura 12 Diagrama de clases de Dirección

La clase *Dirección* es una clase abstracta, no puede construir instancias, sirve para compartir el código de las subclases y permite que las clases cliente Persona y Proveedor utilicen una referencia polimórfica a objetos Edificio, Residencia, Urbanización u Organización. La clase Dirección se debe indicar que es persistente, pese a ser abstracta, pues Direccion debe persistir su atributo domicilio. Los productos necesitan esta información para construir el esquema de traducción. Veamos una primera traducción de la jerarquía anterior en relaciones, empleando la división por tipos, elegida porque nunca una misma instancia mutará de un tipo a otro, o mostrará dos papeles, los tipos son exclusivos.

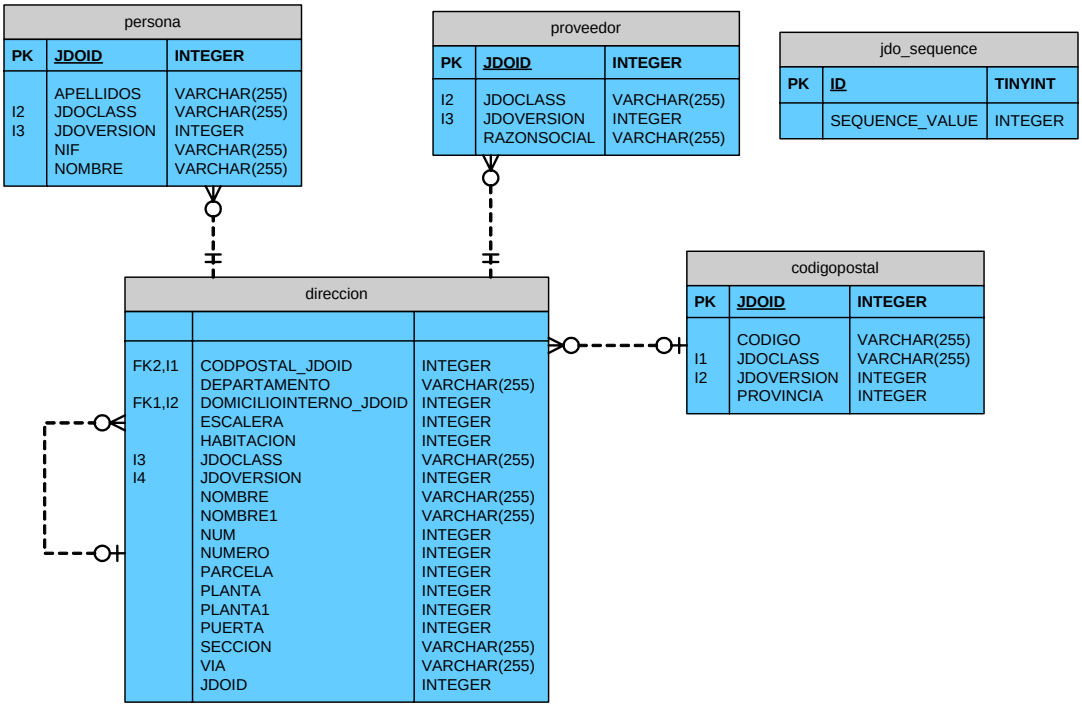


Figura 13 Correspondencia de Dirección según método de división por tipo.

En la figura anterior vemos como se aprovecha la capacidad del RDBMS, para asegurar la integridad referencial con claves ajenas. El otro aspecto que destaca es la presencia de las columnas nombre, nombre1, correspondientes con los atributos denominados nombre en Organización y Urbanización respectivamente, y las columnas planta y planta1 que corresponden con el atributo planta de Edificio y de Residencia.

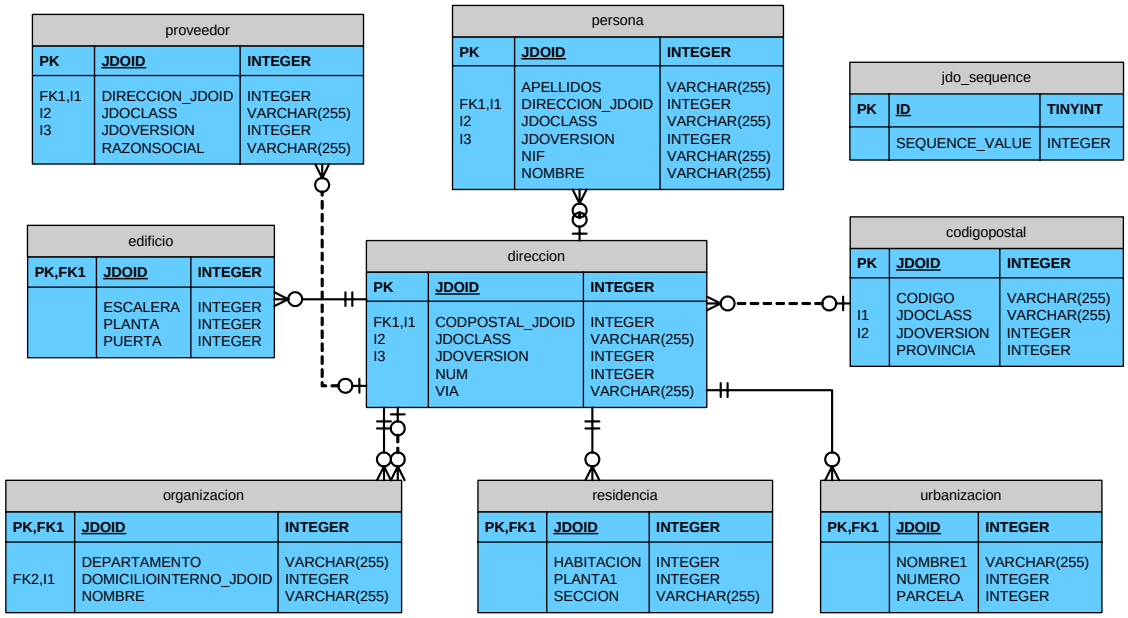


Figura 14 Correspondencia vertical jerarquía de herencia Domicilio

La correspondencia producida por división vertical, resulta en relaciones normalizadas en tercera forma normal (3FN). No hay redundancia, y es posible aprovechar la integridad referencial ofrecida por las bases de datos relacionales. El esquema físico de relaciones es paralelo a la jerarquia de herencia, la regla de uno a uno vista se cumple. Esta técnica ofrece la mayor flexibilidad a la modificación e incorporación de atributos. El mayor

escollo se produce, si la jerarquía de herencia tiene varios niveles, entonces se penaliza el rendimiento, debido al número de tablas enlazadas en la sentencia SQL de recuperación y actualización de un objeto,

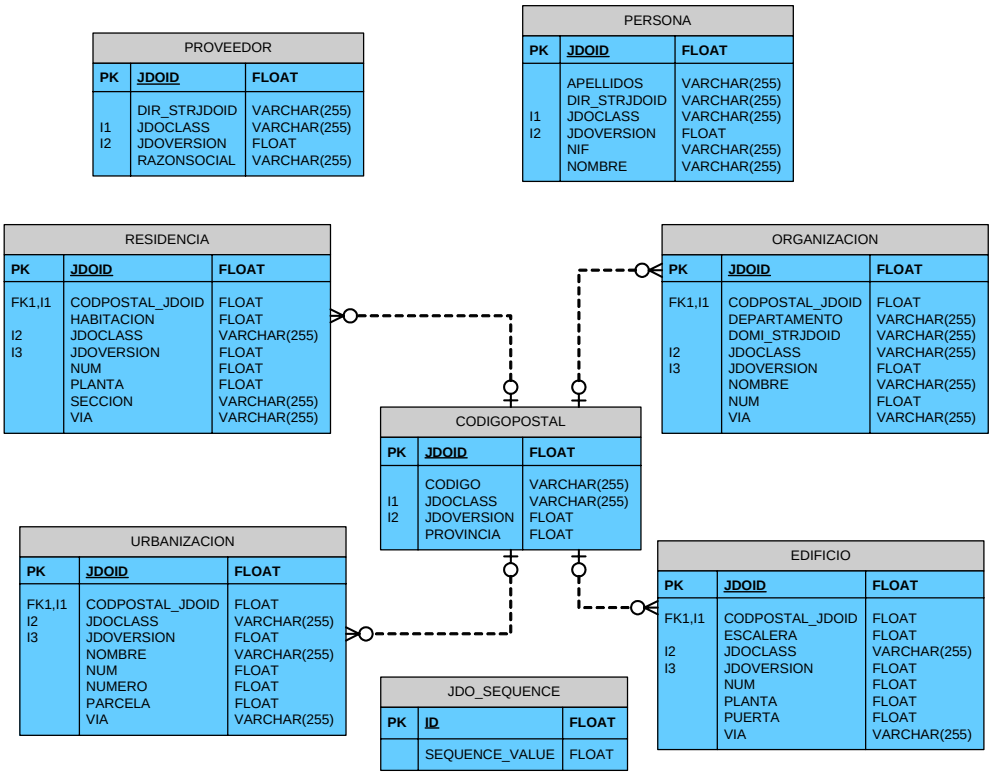


Figura 15Tablas para las clases de Direccion usando división horizontal.

En la división horizontal los atributos heredados y propios son colocados en una misma tabla, resulta muy eficaz el acceso a los datos. La principal desventaja de esta técnica de división horizontal, es su peor flexibilidad al cambio, frente la modificación o adición de atributos a las clases, porque obliga a revisar todas las relaciones. Otro elemento patente es la imposibilidad para aprovechar la integridad referencial, como en la figura anterior, las clases Proveedor y Persona no concretan una relación de claves ajenas, el servicio de persistencia responde de la integridad de los vínculos. En este ejemplo, basado en el producto de Solarmetric, KODO, las entidades cliente, Persona y Proveedor, incluyen una columna DIR_STROID, que contiene el identificador persistente y el tipo del objeto asociado, así resuelve el polimorfismo con referencias persistentes y la integridad referncial.

2.6. De tablas a Clases

Aunque estemos más interesados en la práctica de convertir objetos en túplas, también es necesario, al menos, introducir brevemente cómo transformar tablas en clases. La idea esencial es la misma vista cuando se proponía la equivalencia en el sentido clase a tabla, para cada tabla se crea una clase que representa la tabla dentro del modelo de objetos. El proceso es aplicar las reglas vistas en los apartados anteriores a la inversa, comenzando por asociar a cada tabla una clase, cada columna de la tabla con un atributo en la clase, aprovechar las claves ajenas para definir las clases de asociación y los atributos de referencia, y hacer corresponder las claves primarias con atributos que preservan la misma propiedad de servir de identificador de los objetos. Una posibilidad para mantener la

correspondencia entre claves primarias y referencias a objetos es emplear objetos tipo Map, utilizados estos para los atributos de referencia entre clases para la agregación y la composición.

Es habitual que de esta forma, se concrete un modelo de objetos vacío de funcionalidad, sin apenas comportamiento. Siempre que sea posible, se deben modificar los esquemas de tablas existentes, para lograr un modelo de objetos con funcionalidad y sentido en la lógica del negocio, aprovechando las capacidades del RDBMS para ofrecer distintas visiones de datos, y con la aplicación de ciertos patrones de diseño, como el Bridge y el Proxy [13][18]. El mayor problema son los sistemas legados, en ocasiones, impiden poder modificar el esquema de tablas, para ser adaptado para persistir objetos.

2.7. Resumen

En este capítulo se ha presentado las técnicas básicas para transformar las clases en tablas, ilustrando cada caso con un ejemplo práctico. La esencia del proceso consiste en corresponder cada cada clase con una tabla principal, cada atributo con una columna, cada referencia o colección de referencias de las relaciones de agregación o composición, con claves ajenas, tablas de asociación, o columnas con capacidad para contener la imagen serializada de objetos, tipos blob (binary large object). La herencia se transforma según tres formulas: a una tabla base que contiene todos los atributos y una columna discriminador de tipo final de los objetos representados; una jerarquía de tablas paralela a la jerarquía de herencia; o un conjunto de tablas, una por subclase terminal con los atributos propios y heredados.

Cada alternativa ofrece diferentes capacidades para la actualización, rapidez de acceso, concurrencia y facilidad de mantenimiento y continuidad. Dependiendo del caso, será escogida la fórmula que más se ajuste las necesidades concretas del caso real.

En mi humilde opinión, el proceso de transformar objetos a filas, no hay que ser dogmático, ni forzar la semántica en ninguno de los extremos de la traducción de objetos a filas, y aprovechar las ventajas y capacidades de cada mundo el de las clases y el de las gestores de bases de datos relacionales.

Capítulo 3
JAVA DATA OBJECTS

Desde mayo de 2002 ha sido propuesto como nuevo estándar de persistencia para objetos Java. La especificación Java Data Objects (JDO), que será el foco de atención del resto del trabajo, el análisis de Java Data Objects.

En este capítulo, se describe la especificación del estándar Java Data Objects, versión 1.0.1 [51], con un nivel de detalle acorde al propósito de analizar este estándar de persistencia, bajo la perspectiva del marco que definen los requisitos considerados en el primer capítulo, para ello, el capítulo esta organizado en dos secciones. La primera presenta de forma breve una visión general de JDO, que permite conocer la especificación, sus objetivos, principales elementos y su situación a la fecha de ser escrito este capítulo. La segunda sección desgrana los detalles de la especificación con la profundidad adecuada para discutir su nivel de conformidad con el marco de requisitos, utiliza la ilustración de una pequeña aplicación cuyos objetos deben persistir con JDO, y seguidamente se exponen los entresijos de JDO, de forma clara y concisa con el objeto de permitir discutir, más adelante, sobre sus cualidades y sus debilidades, sobre su congruencia con el planteamiento elaborado más atrás de cómo resolver la persistencia.

El capítulo se apoya en los textos de la propia especificación [51] y en dos libros de idéntico título, "Java Data Objects" uno de R. Roos [29] y el otro de los reconocidos C. Russell, lider de la especificación, y D. Jordan [16], miembro del comité experto, en este último libro se emplea la misma idea de tomar una aplicación como vehículo que introduce en el estudio, compresión y análisis de JDO[3].

3.1. JDO a vista de pájaro

La especificación Java Data Objects (JDO) es el estándar de la comunidad Java que define las interfaces y clases para manejar las clases del dominio (objetos de negocio y de aplicación), cuyas instancias tienen que persistir, y especifica los contratos entre proveedores de clases con capacidad de perdurar sus instancias, y el entorno de ejecución que es parte de la propia implantación de JDO.

JDO no es una propuesta para definir un OODBMS. JDO quiere ser el estándar para interactuar con los sistemas de gestión de datos posibilitando una visión de los datos consistente y uniforme, constituida por objetos Java, con independencia del sistema de gestión de datos donde los datos residan.

JDO no es un servicio de persistencia. Es la especificación de cómo los programadores ven un modelo de servicios de persistencia con transparencia de datos. JDO es la parte visible del un sistema complejo que es un servicio de persistencia.

JDO resume en unas interfaces de programación los servicios necesarios para alcanzar la persistencia de objetos Java sobre distintos sistemas de gestión de datos, de forma uniforme, sin necesidad de conocer los mecanismos de persistencia realmente

[3] Hacer notar que el planteamiento del director de proyecto tres años antes de escribir este capítulo, era presentar una aplicación sobre la que describir los distintos elementos que constituyen la funcionalidad. Azares de la vida han querido que la edición del libro de David Jordan y Craig Russell, y la redacción de estas líneas sean contemporáneas.

utilizados. Y así pues, que no sea necesario escribir código de persistencia para la infraestructura de acceso a los mecanismos de persistencia.

La transparencia de datos de JDO, no necesita alterar los fuentes Java, pero requiere de la definición de la correspondencia datos-objetos y la inclusión del código binario necesario en las clases cuyos objetos serán persistentes.

JDO propone el uso de la meta información necesaria en ficheros XML, separados del código, y la posibilidad del uso un procesador (postprocesador) de clases compiladas, para alcanzar un buen nivel de persistencia ortogonal asegurando la transparencia de datos.

3.1.1. Objetivos Fundamentales

Los objetivos esenciales de la especificación JDO son dos: Proporcionar soporte a la persistencia de objetos Java, que representan objetos de negocio y datos de aplicación, ofreciendo a los programadores una perspectiva centrada en Java de la información persistente, de forma transparente e independiente del sistema gestor de datos. Y facilitar la realización de implementaciones de acceso a los sistemas de gestión de datos que puedan ser integradas en los servidores de aplicaciones de empresa Java [51] [16] [29]. Dicho de otra forma, ***Persistencia Java Transparente, Neutral, Escalable e Integrable***, *p*ersistencia transparente neutral respecto al tipo de sistema de almacenamiento de datos, que sea capaz de procesar cargas de trabajo importantes y que se integre con los servidores de aplicaciones Java aprovechando sus servicios.

JDO quiere ofrecer una solución al problema de la persistencia Java, que presenta una única visión de los datos persistentes, consistente con el modelo de objetos Java, donde no es necesario conocer cada uno de los lenguajes de acceso a datos, impuestos por los diferentes sistemas de gestión de datos, que son empleados en el sostenimiento de los almacenes de datos. Los componentes de aplicación con JDO perciben una visión Java de los datos organizados en clases con relaciones y colecciones de construcciones nativas Java, un modelo de información que es un modelo de objetos Java. Adicionalmente JDO considera, la integración con los entornos de servidores de aplicaciones, definiendo los contratos entre el servidor de aplicaciones y la implantación de JDO, para facilitar un uso transparente de los mecanismos ofrecidos por los servidores de aplicaciones como son, la distribución y coordinación de transacciones, la seguridad, y la gestión de la conexión. De esta forma los programadores centran el esfuerzo de desarrollo en la lógica de negocio y de presentación, despreocupados por las cuestiones de conectividad y acceso a los sistemas de datos concretos.

JDO define dos tipos de interfaces agrupados en sendos paquetes: una interfaz de programación aplicaciones (API) dirigido a los programadores de aplicaciones, la JDO API, y otro, para los fabricantes de contenedores de datos e implementaciones, la JDO SPI. Estos serán comentados en las secciones posteriores.

3.1.2. Entornos de ejecución objetivo

Dos son los entornos de ejecución que deben ser soportados por las implementaciones de JDO:

- Entornos no administrados, entornos locales o de arquitectura en dos capas. Este modo de ejecución trata la persistencia transparente, ocultando a los componentes de aplicación las cuestiones específicas para localizar, obtener y guardar objetos con un servicio de información de empresa

concreto (EIS). Las aplicaciones interaccionan con una visión Java nativa de clases que representan los datos de los servicios de *EIS* (bases de datos, sistemas CRM, ERP,...).

- Entornos administrados, usados en servidores de aplicaciones o arquitecturas de tres o más capas. Las aplicaciones dirigidas a estos entornos además de la persistencia transparente, disponen de la transparencia de los servicios ofrecidos a las aplicaciones, por los sistemas de servidor de aplicaciones Java, servicios relativos a transacciones, seguridad, conexiones, etc. Este modo utiliza la arquitectura *J2EE Conector.*

El resultado en ambos casos pretende ser que las aplicaciones consigan ignorar la problemática de la conectividad con un EIS específico y un marco de desarrollo sencillo de usar.

El entorno de dos capas es típico de las aplicaciones ejecutadas en el ordenador de escritorio o el teléfono móvil Java, aplicaciones de consola y applet, que presentan datos de archivos y bases de datos, el ámbito de la aplicación y de los datos concierne al sistema donde es ejecutada la aplicación, con independencia de los datos sean locales al sistema de ejecución o remotos en los almacenes de datos de empresa. Esta forma de trabajar requiere que la aplicación conozca, para lograr la persistencia, de objetos los interfaces *javax.jdo.PersistenceManager* y *javax.jdo.spi.PersistenceCapable.*

El entorno de servidores de aplicaciones al que JDO esta adaptada, es Enterprise Java Beans (EJB) utilizando para ello la arquitectura *J2EE Conector.* En este entorno se brindan distintos modelos de aplicación, que según que alternativa sea escogida, la aplicación dispondrá de persistencia automática gestionada por el servidor de aplicaciones (Container Manager Persistente, CMP) o mediante la interfaz *javax.jdo.PersistenceManager*, además de disponer de los servicios de transacción, seguridad y conexión ofrecidos por todo servidor de aplicaciones J2EE. Esta forma de construir aplicaciones es más compleja de entender y utilizar, pero prepara la aplicación ante el aumento o escalada de la carga a acometer. Es un modo de operar cada vez más usado por los servicios Web que se despliegan con el concurso de servidores de aplicaciones J2EE.

3.1.3. Capacidad de las clases para persistir. Procesador de código Java

Las clases cuyas instancias pueden ser persistentes, deben incorporar el código necesario para hacer efectiva la magia de la persistencia transparente, esto supone, o bien que el programador añade el código preciso para materializar la capacidad de persistir, o es utilizada una herramienta que amplíe, extienda el código original fuente o compilado con las modificaciones precisas, evitando al programador este importante esfuerzo. La especificación JDO propone la figura del procesador de clases en código ejecutable java, Enhancer, que es un programa que modifica los archivos compilados de las clases, añadiendo el código ejecutable necesario para efectuar grabación y recuperación transparente de los atributos de las instancias persistentes. De esta forma desde el punto de vista del programador solo añade una etapa más a su ciclo de desarrollo, la extensión de las clases persistentes.

JDO pretende que la mayoría de las clases escritas por los programadores puedan ser persistentes, que los objetos pertenecientes a clases concretas definidas por el programador puedan perdurar su estado, con la única limitación de que el estado persistente de las instancias a perdurar, este compuesto por los atributos persistentes que sean independientes del contexto de ejecución: tipos primitivos, tipos de referencia e interfaz y algunas clases del sistema que modelan estado (Array, Date,…).

La fórmula consignada para indicar qué clases y atributos de éstas serán persistentes, es utilizar un fichero de configuración XML, **el descriptor de persistencia**, cuya meta información es usada para, el pre o post procesamiento con el procesador de código y en tiempo de ejecución para lograr la persistencia de los estados de los objetos.

Además de introducir la funcionalidad, para que las instancias de las clases con capacidad de persistir puedan perdurar en los sistemas gestores de datos, es necesario, establecer la correspondencia entre los objetos y su estado persistente, proyectar los objetos como datos, cuando el depósito de datos es una base de datos relacional u otro EIS no basado en objetos. La especificación JDO no establece cómo el mapeo objeto-datos debe ser realizado. Es una cuestión abierta al planteamiento que adopte, quien proporciona la implementación para acceder a un sistema gestor de datos concreto. La nueva versión de JDO en desarrollo especificará el mapeo entre objetos y datos para los sistemas de bases de datos relacionales.

JDO ha sido diseñada para que fabricantes de sistemas gestores de datos, sistemas de acceso a datos y desarrolladores Java sean los beneficiarios de su aceptación. Los fabricantes de EIS (bases de datos, sistemas CRM, ERP,...) que implementen las interfaces de JDO, ofrecerían un acceso de datos estandarizado que podría ser usado por cualquier componente de aplicaciones que use JDO. Los componentes de aplicación diseñados para usar JDO, no tendrían que adaptarse para acceder a cada sistema de datos, podrán acceder a todos los sistemas que implementen la especificación JDO.

3.1.4. Estado actual

JDO es un estándar desde el 25 de Marzo de 2002 habiendo seguido la formalización del proceso de estandarización abierta del programa de Java Community Process 2.1 .

La especificación alcanzó el status de estándar por 10 votos a favor sobre 12 y ninguno en contra. Las empresas que votaron a favor fueron Sun Microsystems, IBM, HP, Compaq, Apple, Nokia Networks, CISCO, Fujitsu BEA, Macromedia y Oracle. Apache y Caldera no votaron.

Todas las especificaciones antes de ser consideradas completas, deben demostrar que funcionan en la práctica, mediante una implementación de referencia y unas baterías de pruebas de conformidad, que son disponibles libremente en [51].

Habiendo sido adoptado como estándar, el proceso de revisión para alcanzar y resolver nuevos requisitos ya se ha iniciado en el 2003. Algunos de las empresas de mayor relevancia en el panorama de la sistemas gestores de datos y servicios de aplicación, han anunciado su apoyo y respaldo al estándar, caso de Sysbase, Informix, Versant, Poet (hoy parte de Versant), SAP, IONA, Software AG, Rational, Oracle, Ericsson.

El pasado año 2003 hemos asistido al crecimiento del número de participes en esta nueva tecnología, cuyo punto de encuentro es el portal ww.jdocentral.com, con más de 15.000 miembros activos. Este portal podemos encontrar documentación, noticias, foros de discusión y al menos disponible una docena de productos comerciales (Genei, Solarmetric, Hemisphere,…) y de código abierto, (Yakarta OJB, XORM, JPOX, TJDO, Speedo).

El 21 de agosto del 2003, cinco de los fabricantes de implementaciones comerciales de JDO de más éxito, algunos pocos de los expertos más reconocidos de JDO, entre otros sus creadores, gente del código abierto, y representantes de las compañías Oracle y SAP, mantuvieron un encuentro para discutir sobre la evolución de JDO a la versión 2.0. Esta reunión no oficial constituye el verdadero punto de inicio desde el que se ha promovido la

petición formal de revisión de la especificación java JDO, que deberá ser aprobada en su día. En esta reunión han sido discutidas las nuevas funcionalidades que deberá incorporar en el futuro JDO, sin perder vista los objetivos iniciales salvo el de compatibilidad binaria, los cambios impulsados en este encuentro añaden funcionalidad para estandarizar el mapeo a las bases de datos relacionales, su representación y comportamiento, la incorporación a las consultas de funciones sumarias, la posibilidad de consultas de eliminación sin recuperación previa de objetos, el uso de las conexiones para ejecución de sentencias en bases de datos, funcionalidades pensadas para el uso en aplicaciones Web, como la codificación automática de los identificadores de objetos para facilitar su uso como parámetros en URL, y otros muchas más que redundan en conseguir la persistencia transparente de objetos Java, con la facilidad de uso y simplicidad que caracteriza a JDO.

En transcurso del 2004, se inició el proceso de revisión JSR-243 Java Data Objects 2.0, con los votos en contra de IBM, Oracle y BEA. El 7 de agosto 2004 finalizó el plazo de revisión pública del borrador inicial de la revisión de JDO.

3.2. JDO BAJO LOS FOCOS

En este punto se inicia una inmersión en JDO con código JAVA de una aplicación de ejemplo, la idea es llamar a la atención sobre las estructuras, la forma de usar y los cambios que introduce JDO, en la manera de hacer programas Java con persistencia, para seguidamente mostrar JDO, revisar la especificación de JDO y así, comprobar qué aspectos de los requisitos del marco del capítulo primero son cubiertos y cuales no.

3.2.1. JDO a través de un ejemplo

La aplicación ejemplo se basa en la ilustración del Banco utilizado en el primer capítulo cuyo modelo de objetos aparece en la siguiente figura.

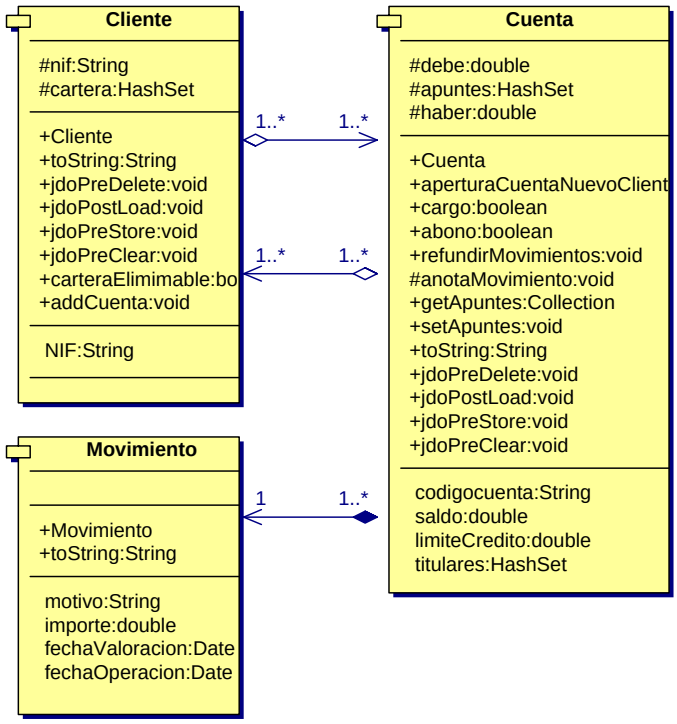


Figura 16 Diagrama de Clases del Banco

La aplicación va a efectuar una secuencia de operaciones para crear un cliente, una cuenta y algunos movimientos que actualicen las instancias de Cuenta, como fue visto en el primer capítulo, pero aquí las instancias de estas clases deben perdurar, ser persistentes.

Las clases java correspondientes al modelo de objetos de la figura anterior, codifican las relaciones y las operaciones sobre instancias de las clases Cliente, Cuenta y Movimiento. Veamos el texto fuente de estas clases y que cambios implica JDO. Como plataforma para probar los ejemplos ha sido utilizada la implementación de referencia de JDO (JDORI) [16].

Listado 14 Clases Cliente y Cuenta original

```
/**
 * Cliente
 * Version 0.2
 * Marzo 2002
 * Copyright Juan Marmol Trabajo fin de carrera
 */
package banco;
import java.io.Serializable;
import java.util.HashSet;
import java.util.Iterator;
/**
 * Cliente del banco
 * @author juan marmol
 * @version 0.2
 * @since Marzo 2002
 */
public class Cliente
    implements Serializable
{
  /**
   * Identificación del cliente
   */
  protected String nif;
  /**
   * @link aggregation
   */
  protected HashSet cartera;

  public Cliente()
  {
  }
  /**
   * Access method for the NIF property.
   * @return   the current value of the NIF property
   */
  public String getNIF()
  {
    return nif;
  }
  /**
   * Sets the value of the NIF property.
   * @param unNIF the new value of the NIF property
   */
  public void setNIF(String unNIF)
  {
    nif = unNIF;
  }
  /**
   * Access method for the cartera property.
   *
   * @return   the current value of the cartera property
   */
  public HashSet getCartera()
```

No hay que importar módulos de jdo, en las clases persistentes. Solo en las que las manejan las operaciones de la API JDO.

El estado persistente JDO de una instancia esta definido por el valor de sus campos. El valor de los campos no puede depender de de objetos inaccesibles o remotos ligados a la ejecución única del programa.

No hay métodos de acceso a datos en el lenguaje del sistema donde persisten las instancias

```
  {
    return cartera;
  }
  /**
   * Sets the value of the cartera property.
   * @param unaCartera the new value of the cartera property
   */
  public void setCartera(HashSet unaCartera)
  {
    cartera = unaCartera;
  }
  /**
   * Retorna la instancia Cliente como cadena de caracteres
   * @return String Cliente como cadena
   */
  public String toString()
  {
    String str = new String("Cliente:" + getNIF() + "\n");
    Iterator it = getCartera().iterator();
    while (it.hasNext()) {
      Cuenta cu = (Cuenta) it.next();
      str.concat(cu.toString() + "\n");
    }
    return str;
  }
/**
 * Cuenta
 * Version 0.2
 * Marzo 2002
 * Copyright Juan Marmol Trabajo fin de carrera
 */
package banco;
import java.util.Calendar;
import java.util.Collection;
import java.util.Date;
import java.util.HashSet;
import java.util.Iterator;
/**
 * Cuenta
 * @version 0.2
 * @author juan marmol
 * @since Marzo 2002*/
public class Cuenta
{
  /**
   * Identificación de la cuenta
   */
  protected String codigocuenta;
  /**
   * Saldo de la cuenta. Refleja la suma de los movimientos de la cuenta
   */
  protected double saldo;
  /**
   * Acumulado cargos. Refleja la suma de los movimientos de la cuenta
   */
  protected double debe;
  /**
   * Acumulado abonos la cuenta. Refleja la suma de los movimientos de la cuenta
   */
  protected double haber;
  /**
   * Limite de credito concedido para esta cuenta
   */
  protected double limiteCredito;
  /**
   * @link aggregationByValue
   * @associates Movimiento
   * @supplierCardinality 0..*
   */
  protected Collection apuntes;
  /**
   * @link aggregation
   * @associates banco.Cliente
```

El estándar establece como obligatorio el soporte de algunos de los tipo colección, el resto opcionales a elección del fabricante JDO.

Las relaciones entre objetos son relaciones nativas Java.

```
   * @supplierCardinality 1..*
   * @clientCardinality 1..*
   */
  protected HashSet titulares;

  public Cuenta()
  {
    codigocuenta = new String();
    saldo = 0;
    limiteCredito = 0;
    titulares = new HashSet();
    apuntes = new HashSet();
  }

  /**
   * Asigna los datos para una nueva cuenta
   * @param nif Numero de Identificacion Fiscal del cliente
   * @param ccc Codigo Cuenta Cliente Asignado segun formato legal
   * @param aportacion Importe en EU de la aportación del apunte de apertura
   * @param limcredito Limite de Credito segun la clasificación de riesgo del
cliente
   */
  public void aperturaCuentaNuevoCliente(String nif, String ccc,
                                         double aportacion,
                                         double limcredito)
  {
    synchronized (this) {
      Cliente nuevocliente = new Cliente();
      nuevocliente.setNIF(nif);
      setCodigocuenta(ccc);
      setLimiteCredito(limcredito);
      Calendar almanaque = Calendar.getInstance();
      Date fecha = almanaque.getTime();
      abono(aportacion, "Apunte Apertura", fecha);
      titulares.add(nuevocliente);
      nuevocliente.getCartera().add(this);
    }
  }
  /**
       * Decrementa el saldo de cuenta anotando los detalles de la operación
   * @param cantidad importe del cargo > 0
   * @param motivo de cargo
   * @param fechaValor del cargo
   * @return true si fue realizado el cargo, false en otro caso
   */
  public boolean cargo(double cantidad, String motivo, Date fechaValor)
  {
    synchronized (this) {
      double sld = saldo;
      sld -= cantidad;
      if ( (sld >= saldo) || ( -sld > limiteCredito)) {
        return false;
      }
      anotaMovimiento( -cantidad, motivo, fechaValor);
      saldo = sld;
      debe += cantidad;
      return true;
    }
  }
  /**
   * Abono en cuenta.
   * Se añade el movimiento de abono
   * @param cantidad importe del abono
   * @param motivo del abono
   * @param valor fecha valor
   * @return true si fue realizado el cargo, false en otro caso
   */
  public boolean abono(double cantidad, String motivo, Date valor)
  {
    synchronized (this) {
      double sld = saldo;
      sld += cantidad;
      if (sld <= saldo) {
        return false;
      }
```

Jdo exige que exista un constructor sin argumentos.

Las operaciones son nativas Java. No cambia la semántica por trabajar con instancias persistentes o transitorias. La lógica del negocio es puro Java.

Los cambios sobre las instancias persistentes son traslados automáticamente al consolidar los cambios dentro de una transacción.

```
    anotaMovimiento(cantidad, motivo, valor);
    saldo = sld;
    haber += cantidad;
    return true;
  }
}
/**
 * Anotar movimiento en la cuenta
 */
public void refundirMovimientos()
{
  synchronized (this) {
    if (getApuntes().size() < 2) {
      return;
    }
    Calendar cal = Calendar.getInstance();
    Date fecv = cal.getTime();
    HashSet apuntesRefundidos = new HashSet();
    setApuntes(apuntesRefundidos); //los movimientos seran recolectados
    if (haber > 0) {
      anotaMovimiento(haber, "Fusion operaciones abonos", fecv);
    }
    if (debe < 0) {
      anotaMovimiento(debe, "Fusion operaciones cargo", fecv);
    }
  }
}
/**
 * Anotacion de nuevos movimientos a la cuenta
 * @param importe Importe en EU del asiento
 * @param motivo  Descripción motivo del asiento
 * @param fecValor Fecha de valoración del apunte
 */
protected void anotaMovimiento(double importe, String motivo,
                               Date fecValor)
{
  synchronized (this) {
    Movimiento mov = new Movimiento();
    mov.setImporte(importe);
    mov.setMotivo(motivo);
    mov.setFechaValoracion(fecValor);
    Calendar cal = Calendar.getInstance();
    Date fop = cal.getTime();
    mov.setFechaOperacion(fop);
    apuntes.add(mov);
  }
}
/**
 * Access method for the codigocuenta property.
 *  @return   the current value of the codigocuenta property
 */
public String getCodigocuenta()
{
  return codigocuenta;
}
/**
 * Sets the value of the codigocuenta property.
 * @param unCodigocuenta de la cuenta
 */
public void setCodigocuenta(String unCodigocuenta)
{
  codigocuenta = unCodigocuenta;
}

/**
 * Access method for the saldo property.
 * @return   the current value of the saldo property
 */
public double getSaldo()
{
  return saldo;
}

/**
 * Sets the value of the saldo property.
```

Una limitación práctica es no eliminar los estados **de forma transaparente de la BD, ni satisfacer la integridad las referencias afectadas.** JDO facilita mecanismos que deben ser incorporados al modelo, las funciones retorno de llamada (callbacks) para algunos eventos de la persistencia.

```
   * @param aSaldo the new value of the saldo property
   */
  public void setSaldo(double aSaldo)
  {
    saldo = aSaldo;
  }
  /**
   * Access method for the limiteCredito property.
   * @return   the current value of the limiteCredito property
   */
  public double getLimiteCredito()
  {
    return limiteCredito;
  }
  /**
   * Sets the value of the limiteCredito property.
   * @param unlimiteCredito para descubierto
   */
  public void setLimiteCredito(double unlimiteCredito)
  {
    limiteCredito = unlimiteCredito;
  }
  /**
   * Access method for the apuntes property.
   * @return   the current value of the apuntes property
   */
  public Collection getApuntes()
  {
    return apuntes;
  }
  /**
   * Sets the value of the apuntes property.
   * @param unosApuntes de la cuenta
   */
  public void setApuntes(HashSet unosApuntes)
  {
    // Cuidado más de una aplicación podría intentar esto,
    //  asi pues, ...
    apuntes = unosApuntes;
  }

  /**
   * Retorna la cuenta como un cadena de caracteres
   * @return String con los datos básicos
   */
  public String toString()
  {
    String str = new String("Cuenta:" + getCodigocuenta());
    str = str + " saldo: " + Double.toString(getSaldo()) + "\n";
    Iterator it = getTitulares().iterator();
    while (it.hasNext()) {
      Cliente cli = (Cliente) it.next(); //navegacion transparente
      str.concat(" " + cli.getNIF());
    }
    str += "\n";
    return str;
  }
  /**
   * Access method for the titulares property.
   * @return   the current value of the titulares property
   */
  public HashSet getTitulares()
  {
    return titulares;
  }
  /**
   * Asigna los titulares de la cuenta
   * @param unostitulares Clientes titulares de la cuenta
   */
  public void setTitulares(HashSet unostitulares)
  {
    titulares = unostitulares;
  }
}
```

A la vista del código Java anterior, se comprueba como JDO no fuerza cambios significativos en las clases del dominio, cuyas instancias serán persistentes, impone ciertas limitaciones y obligaciones. Una obligación importante es asegurar la integridad referencial, antes eliminar una instancia persistente, no hay recolección de basuras sobre los estados almacenados, salvo que el sistema gestor anfitrión lo ofrezca, esto implica añadir un código extra en las clases del modelo. No son definidos nuevos métodos o atributos porque las clases sean persistentes, el código para acceder y manipular los atributos y las relaciones de las clases no sufre alteraciones.

¿Cómo hacer que una clase sea persistente y que sus objetos perduren?

1. Indicar que será persistente en un archivo XML.
2. Establecer los parámetros de configuración que definen las condiciones de ejecución de la persistencia.
3. Procesar el código compilado de las clases persistentes y de las que manejan la persistencia de las primeras.
4. Finalmente, es posible operar con la persistencia aplicando algunos métodos que desencadenan la magia de la persistencia.

   Veamos la secuencia más cerca sobre el ejemplo que nos ocupa.

3.2.1.1. Declarar qué clases son persistentes

En un fichero XML se indica qué clases y qué atributos de estas son persistentes, en qué forma clientela y herencia se mezclan con persistencia, cuales son los modelos de identidad empleados, y cual es la correspondencia entre objetos y datos. En el ejemplo del banco se utiliza un mecanismo de persistencia de objetos, que no requiere expresar la conexión entre datos y objetos, ni el tipo de identidad a utilizar, solo que es persistente y la relación entre clases persistentes, se utiliza para gestionar la persistencia la plataforma de referencia de JDO (RI), cuyo mecanismo de persistencia es un sistema de ficheros indexados.

La definición del estado persistente de una clase, está representado por los valores de los atributos escogidos para persistir. Estos valores no pueden ser dependientes de objetos inaccesibles, remotos o que representan entidades de ejecución Java, porque si no al recuperar el objeto, las dependencias no podrían ser reconstruidas.

En el ejemplo, el siguiente listado muestra el descriptor de persistencia, que describe que será persistente:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE jdo SYSTEM "http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
         <package name="banco">
         <class name="Cuenta" >
                  <field name="titulares">
                           <collection element-type="Cliente"/>
                       </field>
                       <field name="apuntes">
                           <collection element-type="Movimiento"/>
                       </field>
                           <field name="codigocuenta">
                       </field>
                       <field name="limiteCredito">
                      </field>
                       <field name="saldo">
                       </field>
                       <field name="debe">
                </field>
                       <field name="haber">
```

```
                </field>
              </class>
              <class name="Cliente">
                      <field name = "cartera">
                           <collection element-type="Cuenta"/>
                      </field>
                      <field name = "nif">
                      </field>
              </class>
              <class name="Movimiento">
              </class>
       </package>
</jdo>
```

Listado 15 Descriptor de persistencia del banco

El descriptor de persistencia establece que clases tendrán instancias persistentes, qué atributos de estas pueden persistir, para los atributos que expresan relaciones se indica, si se trata de una colección o correspondencia clave-valor, cuyos elementos tendrán que ser compatibles, con la clase indicada en la declaración *element-type* y si estos elementos serán parte del objeto o referencias a otros objetos persistentes. En el ejemplo todos los atributos son persistentes y las colecciones contienen referencias a objetos persistentes.

Los archivos descriptores de persistencia deben ser nombrados como (literalmente) `package.jdo`, cuando contiene la descripción del estado persistente de las clases de la aplicación o paquete, o cuando solo contiene la definición de persistencia de una clase sola, entonces, `<nombre la clase>.jdo`.

Veamos otros ejemplos más complicados, basados en ejemplos utilizados en capítulos siguientes, señalan las alternativas escogidas por los fabricantes de JDO, para expresar el mapeo de clases en tablas, bien se utiliza el mecanismo previsto en JDO, de los tag extension y vendor, y la otra alternativa, consiste en incluir los metadatos de la transformación de clases a tablas, en ficheros aparte, frecuentemente utilizando XML. Así pues, según el fabricante encontraremos una alternativa, otra o ambas.

```
<?xml version="1.0"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
  <package name="jdo.test.performance">
…     <class identity-type="datastore" name="MyClass">
       <field name="i">
         <extension vendor-name="jrelay" key="column" value="I">
           <extension vendor-name="jrelay" key="jdbc-type" value="INTEGER"/>
         </extension>
         <extension vendor-name="jrelay" key="java-type" value="int"/>
       </field>
       <field name="d">
         <extension vendor-name="jrelay" key="column" value="D">
           <extension vendor-name="jrelay" key="jdbc-type" value="FLOAT"/>
         </extension>
         <extension vendor-name="jrelay" key="java-type" value="double"/>
       </field>
       <field name="string">
         <extension vendor-name="jrelay" key="column" value="STRING">
           <extension vendor-name="jrelay" key="jdbc-type" value="VARCHAR"/>
           <extension vendor-name="jrelay" key="size" value="60"/>
         </extension>
         <extension vendor-name="jrelay" key="java-type"
value="java.lang.String"/>
       </field>
       <field name="strings">
         <array/>
         <extension vendor-name="jrelay" key="column" value="STRINGS">
           <extension vendor-name="jrelay" key="jdbc-type"
value="VARBINARY"/>
```

```
        </extension>
        <extension vendor-name="jrelay" key="java-type"
value="java.lang.String[]"/>
      </field>
      <extension key="class-id" value="MyC" vendor-name="jrelay"/>
      <extension key="table" value="MYCLASS" vendor-name="jrelay">
        <extension vendor-name="jrelay" key="id-column" value="ID">
          <extension vendor-name="jrelay" key="jdbc-type" value="CHAR"/>
          <extension vendor-name="jrelay" key="size" value="20"/>
        </extension>
        <extension vendor-name="jrelay" key="class-id-column"
value="CLASS_ID">
          <extension vendor-name="jrelay" key="jdbc-type" value="CHAR"/>
          <extension vendor-name="jrelay" key="size" value="3"/>
        </extension>
        <extension vendor-name="jrelay" key="version-column"
value="VERSION">
          <extension vendor-name="jrelay" key="jdbc-type" value="INTEGER"/>
        </extension>
      </extension>
    </class>
  </package>
  <extension key="version" value="Sun Aug 31 21:53:38 CEST 2003" vendor-
name="jrelay"/>
</jdo>
```

Listado 16 Mapeo de un árbol de objetos con JRELAY

La otra alternativa ofrecida por otro fabricante sobre la misma figura, es como sigue.

```
<?xml version="1.0" encoding="UTF-8"?>
<mapping>
    <package name="jdo.test.performance">
        <class name="MyClass">
            <jdbc-class-map type="base" pk-column="JDOID" table="MYCLASS"/>
            <jdbc-version-ind type="version-number" column="JDOVERSION"/>
            <jdbc-class-ind type="in-class-name" column="JDOCLASS"/>
            <field name="d">
                <jdbc-field-map type="value" column="D"/>
            </field>
            <field name="i">
                <jdbc-field-map type="value" column="I"/>
            </field>
            <field name="string">
                <jdbc-field-map type="value" column="STRING"/>
            </field>
            <field name="strings">
                <jdbc-field-map type="collection" element-column="ELEMENT"
order-column="STRINGS_ORDER" ref-column.JDOID="JDOID"
table="MYCLA_STRINGS"/>
            </field>
        </class>
    </package>
```

Listado 17 Mapeo de un árbol de objetos con KODO

A la vista está, que la cuestión del mapeo tiene su dificultad, por lo cual se aconseja utilizar productos que faciliten el manejo de los descriptores de persistencia.

3.2.1.2. Preparar el entorno de trabajo

El entorno de ejecución con persistencia JDO requiere, como cabría de esperar, que las clases que implantan JDO y las de soporte del mecanismo de persistencia, estén accesibles como otras librerías necesarias en ejecución, pero además también deben estar los archivos de:

- Las propiedades que determinan la conectividad, normalmente en un fichero de texto figura la configuración de la conexión: tipo de mecanismo empleado, usuario, contraseña, y otras características de la conexión y funcionamiento.
- Los descriptores de persistencia deben estar accesibles durante la ejecución, para aportar a la implementación de JDO la metainformación necesaria, que puede figurar empotrada en el código Java. Habitualmente en los directorio META-INF, WEB-INF, el correspondiente al paquete donde están las clases.
- La versión modificada de las clases persistentes y su versión original, ya que algunas operaciones, como la duplicación de instancias para su transmisión por la Red, solo son posibles aplicar a las clases originales.

En nuestro ejemplo, la variable de entorno CLASSPATH, contiene los caminos hasta los archivos necesarios.

3.2.1.3. Procesar las clases para la persistencia

Una vez preparadas las clases del dominio que van a ser persistentes, una vez compiladas y antes de ejecutar la aplicación es necesario aplicar el Enhancer, que extiende o amplía las clases que persisten y aquellas que manejan la persistencia, generando la imagen ejecutable de las clases dotadas de la capacidad de persistir y de manejar persistencia. Utilizando Apache Ant [45] la tarea es como sigue.

```
<target name= "enhanceJDORI101">
       <java fork="yes" failonerror="yes"
            classname="com.sun.jdori.enhancer.Main"
            classpathref="project.class.path">
            <arg line="-v -d $${enh.home}
                       ${enh.home}\banco\package.jdo
                       ${dest}\banco\Movimiento.class
                       ${dest}\banco\Cuenta.class
                       ${dest}\banco\Cliente.class"
            />
       </java>
```

Listado 18 Tarea Amplicar clases con JDO RI.

Cada vez que las fuentes sean compiladas, los archivos de clases compilados deben ser procesados antes de ejecutar la aplicación. Algunos entornos JDO ofrecen la modificación de las clases en tiempo de ejecución, cuando las clases son cargadas en la máquina virtual Java.

3.2.1.4. Conectar con el sistema de gestión de datos

Para poder operar con persistencia sobre los objetos de la aplicación, hay que crear un canal entre el mecanismo de persistencia utilizado y la aplicación en ejecución. Este canal proporciona el contexto, sobre el que son efectuadas las operaciones persistentes, que logran que los datos y sus modificaciones perduren. Cada una de las operaciones cuyos resultados deben perdurar, tiene que ser realizada dentro de una transacción, que garantiza la consistencia e integridad. En JDO, el contexto de la conexión, la transacción y las operaciones internas con persistencia, son manejados con un objeto de la clase PersistenceManager. Las operaciones del ejemplo, están diseñadas aplicando el patrón de diseño comando, que encontramos en [13] y ilustrado en [19], también son similares, a los casos de uso, o prueba, de JUnit. Todas las operaciones que requieren de persistencia, cuando se crea el caso de uso, obtienen un objeto gestor de persistencia de la clase,

`PersistenceManager`, y partir de éste la referencia a un objeto `transacción`. El `PersistenceManager` establece la conexión con el mecanismo de persistencia, por el programador.

En el ejemplo cada operación con objetos persistentes, es una subclase de Operaciones de la Aplicación, AppOper, que concreta la funcionalidad de persistencia común a todas las operaciones, y provoca la ejecución del método diferido `ejecutarOperacion`, que cada subclase concreta con las instrucciones Java a ejecutar sobre estos objetos.

```
  /**
   * Crear e iniciar el contexto de persistencia.
   * Inicia la conexión con el sistema gestor de bases de datos.
   * Asigna la transaccion con cuyo concurso son efectuadas las
operaciones persistentes
   */
  public AppOper()
  {
    try {

      pmf = JDOHelper.getPersistenceManagerFactory(
          obtenerPropiedadesJDO());
      pm = pmf.getPersistenceManager();
      tx = pm.currentTransaction();
    }
    catch (Exception ex) {
      ex.printStackTrace(System.err);
      System.exit( -1);
    }
  }

  public void ejecutar()
  {
    try {
      tx.begin();
      ejecutarOperacion();
      tx.commit();
    }
    catch (Exception ex) {
      ex.printStackTrace(System.err);
      if (tx.isActive()) {
        tx.rollback();
      }
    }
  }

  protected abstract void ejecutarOperacion();
```

Una factoría proporciona gestores de persistencia

`pm` gestiona la persistencia. Ofrece las operaciones para hacer persistente un objeto, conocer su status persistente y eliminar permanentemente un objeto.

Normalmente toda operación sobre objetos del modelo que persisten, debe ser realizada dentro de una transacción.

`tx` es un objeto transacción que asume el papel de garantizar la integridad los cambios en base de datos

Listado 19 Operaciones de Aplicación, AppOper

Crear base de datos

Antes de operar con objetos persistentes deberemos proporcionar el espacio donde son almacenados los datos de sus estados. En JDORI esto consiste en abrir una conexión con una propiedad que provoca la creación de los archivos de la base de datos al consolidar una transacción.

```
  public CrearBD(String fichbd)
  {
    try {
      jdoproperties = obtenerPropiedadesJDO();
      System.out.print("propiedades conexion");
      jdoproperties.list(System.out);
      jdoproperties.put("com.sun.jdori.option.ConnectionCreate", "true");
      jdoproperties.put("javax.jdo.option.ConnectionURL",
```

```
                    "fostore:../db/" + fichbd);
        PersistenceManagerFactory pmfc = JDOHelper.
                                    getPersistenceManagerFactory(
            jdoproperties);
        PersistenceManager pmc = pmfc.getPersistenceManager();
        Transaction tx = pmc.currentTransaction();
        tx.begin();
        tx.commit();
        pmc.close();
        jdoproperties.put("com.sun.jdori.option.ConnectionCreate",
                    "false");
        //verificacion conexion, excepcion en caso de error
        pmf = JDOHelper.getPersistenceManagerFactory(jdoproperties);
        pm = pmf.getPersistenceManager();
        tx = pm.currentTransaction();

        OutputStream fop = new java.io.FileOutputStream("jdo.properties");
        jdoproperties.store(fop, "parametros persistencia");
        System.out.print("Base de datos:" + fichbd + " creada");
             }
             catch (Exception ex) {
               ex.printStackTrace(System.err);
             }
```

Listado 20 Creación de la base de datos con JDO RI

Una vez creada la base de datos fijamos los valores que determinan las características de las sucesivas conexiones sobre un archivo, este ejemplo denominado `jdo.properties`. Algunas de las características de conexión, pueden ser cambiadas durante la ejecución para aprovechar funcionalidades como las transacciones de solo lectura, bloqueo optimista, retener los estados al finalizar las transacciones o permitir instancias transitorias transaccionales, cuyo estado es preservado durante una transacción para poder deshacer las modificaciones, si la transacción es rechazada.

Con la creación de la base de datos y la generación del archivo de propiedades para la conexión, la fase de preparación está completa.

3.2.1.5. Operar sobre las instancias persistentes

JOD permite una verdadera transparencia, consigue que operar con instancias persistentes no sea distinto de manejar instancias transitorias. En el ejemplo la clase abstracta AppOper, aísla a las operaciones definidas en subclases, del patrón de inicio ejecución y consolidación, de toda operación persistente.

Hacer persistentes instancias

Fácil. Crear la instancia y pedir al gestor de persistencia que la convierta en persistente.

```
/**
 * AperturaCuenta.java
 * Version 0.1
 * Enero 2003
 * Copyright Juan Marmol Trabajo fin de carrera
 */
package casosdeuso;

import java.util.Iterator;
import banco.Cuenta;
import banco.Movimiento;

/**
 * Hacer persistente un objeto explicitamente
 * @author Juan Marmol
 * @version 0.1
```

```
 * @since Febrero 2004
 */
public class AperturaCuenta
    extends AppOper

{
  /**
   * Numero Identificacion Fiscal
   */
  private String nif;
  /**
   * Codigo Cuenta Cliente
   */
  private String ccc;
  /**
   * Limite de Credito Concedido
   */
  private double limcredito;
  /**
   * Aportacion en euros
   */
  private double aportacion;

  /**
   * Apertura de cuenta para nuevo cliente
   * @param args Nif, CCC, credito, cantidad euros
   */
  public static void main(String[] args)
  {
    AperturaCuenta nc;
    if (args.length < NUMARGC) {
      System.err.print("Por favor,NIF, CCC,credito, cantidad");
      return;
    }
    nc = new AperturaCuenta(args[NIFARGC], args[CCCARGC],
                            Double.parseDouble(args[CREARGC]),
                            Double.parseDouble(args[CANARGC]));
    nc.ejecutar();

  /**
   * Apertura Cuenta Nuevo Cliente
   * @param pnif NIF del cliente
   * @param pccc Codigo Cuenta Cliente Asignado
   * @param plm  Limite de Credito concedido segun su clasificacion de
riesgo
   * @param pimp Importe en euros de la aportación inicial
   */
  public AperturaCuenta(String pnif, String pccc, double plm,
                        double pimp)
  {
    super();
    nif = pnif;
    ccc = pccc;
    limcredito = plm;
    aportacion = pimp;
  }

  protected void ejecutarOperacion()
  {
    Cuenta cu = new Cuenta();
    cu.aperturaCuentaNuevoCliente(nif, ccc, limcredito, aportacion);
    pm.makePersistent(cu); // Hecho persistir explicitamente
    // los objetos dependientes son hechos persistentes por alcance
    System.out.println("Apertura Cuenta:");
    System.out.print(cu.toString());
    Iterator it = cu.getApuntes().iterator();
    while (it.hasNext()) {
```

La conexión y la transacción hechas en la superclase.

El objeto es hecho persistir expresament, pero los objetos de su cierre de persistencia también, sin intervención expresa del programador.

```
      Movimiento mov = (Movimiento) it.next();
      System.out.print(mov.toString());
    }
  }
}
```

Listado 21 Apertura de nueva cuenta

Acceder a las instancias

Hechos persistir los objetos, estos pueden ser recuperados de la base de datos de tres formas:

- Recorrer la extensión de una clase.
- Navegar por el modelo de objetos persistente.
- Ejecutar una consulta sobre las instancias de una clase, su extensión, una colección con instancias de ese tipo, o sobre otra consulta.

Recorrer la extensión de una clase, consiste en iterar sobre el conjunto de todas las instancias de determinado tipo. Una extensión, objeto del tipo `Extent`, es una facilidad de JDO para acceder al conjunto de todas las instancias de una clase dotada de persistencia, o de la clase y sus descendientes igualmente dotados de persistencia. Una vez que una instancia es recuperada, el acceso a sus objetos relacionados, provoca la recuperación estos a memoria automáticamente, esto es, navegar por el modelo de objetos con transparencia de datos. La ejecución de una consulta, resulta en una colección de aquellas instancias de cierta clase, referenciadas en una colección o por su extensión, que satisfacen, un predicado sobre los atributos y referencias de las instancias consultadas. Las tres formas de acceso pueden ser combinadas entre sí, un patrón muy habitual en las aplicaciones, será localizar con una consulta aquella instancia que cumple cierta condición de búsqueda, a partir de ésta navegar por su cierre de persistencia, cuyos objetos a su vez pueden ser objeto de nuevas consultas. Veamos un ejemplo de lo dicho.

Colección de las instancias persistentes de una clase

```
protected void ejecutarOperacion()
{ // Recorrer la extension de todas las instancias persistentes Cliente
  Extent extensionClientes = pm.getExtent(Cliente.class, true);
  Iterator iter = extensionClientes.iterator();
  while (iter.hasNext()) {
    Cliente cli = (Cliente) iter.next();
    System.out.print(cli.toString());
  }
  extensionClientes.close(iter);

}
```

Listado 22 Operación Listado de Clientes

El código anterior recorre la colección de todas las instancias de la clase `Cliente` y todas las subclases persistentes de `Cliente` (porque el segundo argumento de `getExtent`, es "`true`"). Al finalizar la iteración, los recursos asignados son liberados al invocar el método **`close`**. Si el sistema gestor de datos asociado proporciona una visión de datos consistente, entonces el listado anterior mostrará el estado de los clientes al inicio de la transacción.

Navegar entre objetos del modelo

```
/**
 * Retorna la cuenta como un cadena de caracteres
 * @return String con los datos básicos
 */
public String toString()
{
```

```
    String str = new String("Cuenta:" + getCodigocuenta());
    str = str + " saldo: " + Double.toString(getSaldo()) + "\n";
    Iterator it = getTitulares().iterator();
    while (it.hasNext()) {
      Cliente cli = (Cliente) it.next(); //navegacion transparente
      str.concat(" " + cli.getNIF());
    }
    str += "\n";
    return str;
  }
```

Listado 23 Cuenta método toString

El método `toString` de la clase `Cuenta`, actúa sobre la colección de los titulares mostrando el NIF de cada de uno de estos. No ha sido necesario codificar otro método distinto para trabajar sobre instancias persistentes. Si éste método se aplica a una instancia `Cuenta` persistente, al acceder a cada referencia del tipo `Cliente`, el objeto correspondiente es instanciado de nuevo con los datos obtenidos desde la base de datos por JDO, sin indicación expresa en el código.

Consultar

Consultar permite localizar y recuperar las instancias persistentes contenidas en su extensión correspondiente, o en una colección arbitraría de objetos persistentes, que cumplen cierto predicado de selección o filtro, en JDOQL, que tiene sintaxis Java. Veamos como localizar y mostrar los detalles de un cliente del que conocemos su NIF.

```
protected void ejecutarOperacion()
{
  // Consultar cuentas  de un cliente
  Extent extensionClientes = pm.getExtent(Cliente.class, true);
  Query consulta = pm.newQuery(extensionClientes, "nif == pnif");
  consulta.declareParameters("String pnif");
 Collection resultado = (Collection)
           consulta.execute(nifconsulta);
  Iterator iter = resultado.iterator();
  while (iter.hasNext()) {
    cliente = (Cliente) iter.next();
    System.out.print(cliente.toString());
  }
  consulta.close(resultado);
```

Conjunto instancias Cliente y subclases de Cliente persistentes.

Condición de Selección JDQOL
nif es un atributo de Cliente

Listado 24 Consultar un Cliente y sus cuentas

Cada consulta requiere indicar la clase de las instancias a procesar, clase candidata, para especificar el tipo de las instancias del resultado, y el ámbito de los identificadores que aparecen en las condiciones del filtro, que selecciona las instancias incluidas en la colección del resultado. Cuando la colección consultada, colección candidata, es una extensión, la clase candidata es la correspondiente a la extensión que se consulta; cuando se trata de una colección arbitraria de objetos persistentes, como puede ser el resultado de otra consulta previa, se debe asignar la clase candidata de la consulta invocando el método `setCandidates`.

Actualización instancias

La actualización de una instancia persistente es transparente, en principio. Esto es, utilizamos la asignación y la invocación de métodos modificadores, tal y cómo se realiza con objetos no persistentes; la implementación de JDO, se encarga de que los cambios en el estado de los objetos persistentes perduren, queden grabados. El siguiente listado es un sencillo ejemplo ilustrativo.

```
protected void ejecutarOperacion()
{
```

```
Extent cuentas = pm.getExtent(Cuenta.class, true);
Query consulta = pm.newQuery(cuentas, "codigocuenta == ccc");
consulta.declareParameters("String ccc");
Collection resultado = (Collection) consulta.execute(ccc);
Iterator iter = resultado.iterator();

if (iter.hasNext()) {
  cuenta = (Cuenta) iter.next();
  System.out.print(cuenta.toString());
}
consulta.close(resultado);
Calendar cal = Calendar.getInstance();
Date fec = cal.getTime();
if (cuenta == null) {
  System.err.print("Codigo de cuenta incorrecto");
  return;
}
// el abono o el cargo de un movimiento es hecho persistente por alcance
if (cantidad >= 0) {
  cuenta.abono(cantidad, motivo, fec);
}
else {
  cuenta.cargo( -cantidad, motivo, fec);
}
System.out.print(cuenta.toString());
```

La cuenta es modificada sin hacer nada de especial por ser persistente

Listado 25 Operación de cargo o abono

En el listado anterior recordar que el método **`ejecutaroperación`** es invocado dentro una transacción, cuando la transacción es consolidada (commit), los nuevos datos estarán disponibles en la base de datos del Banco.

Las actualizaciones deberían ser aplicadas siempre a los atributos de los objetos persistentes, no a referencias a las instancias manejadas por estos atributos, ya que el mecanismo habitual empleado para conservar el estado al inicio de una transacción, es crear un copia superficial de los objetos persistentes a modificar, así pues, los atributos que no forman parte del objeto copiado, no puede ser recuperadas si la transacción es rechazada.

Existe una limitación sobre el control automático de las modificaciones de los objetos persistentes agregados formando parte de otros, y además, son compartidos entre varios objetos persistentes, las actualizaciones de estas instancias deben ser advertidas expresamente a los distintos objetos que los contienen, con la ejecución del método **`makeDirty`**. Esta situación podrá habitualmente ser evitada con ajustes en el diseño de las clases.

Borrar instancias

En JDO los estados almacenados de las instancias persistentes deben ser eliminados explícitamente de los sistemas de datos donde permanecen, excepción hecha de algunos sistemas gestores de datos basados en objetos. En Java, cuando las instancias dejan de ser referenciadas por otros objetos, son eliminadas automáticamente, pero esto no puede significar la eliminación de los sus estados persistentes. Para eliminar los estados almacenados se debe codificar expresamente su eliminación para la mayoría de los sistemas. Una estrategia de eliminación automática centrada en la capa del servicio de persistencia, requiere una asignación de recursos importante y provocaría la contención en el acceso a datos concurrente por múltiples procesos. Veamos un par de ejemplos de código.

```
protected void ejecutarOperacion()
{
```

```
    Collection movs;
    ConsultaCuenta consultacuenta = new ConsultaCuenta(codigoCuenta);
    consultacuenta.ejecutar();
    cuenta = consultacuenta.obtenerCuenta();
    if (cuenta == null) {
      System.err.print("Codigo de cuenta erroneo");
      System.err.println(codigoCuenta);
      return;
    }
    movs = cuenta.getApuntes();
    cuenta.refundirMovimientos();
    pm.deletePersistentAll(movs);
    System.out.print(cuenta.toString());
}
```

La operación aprovecha la capacidad de Java de recolección de memoria e invoca el borrado de los estados con **deletePersitentAll**

Listado 26 Eliminar Instancias Persistentes. Refundir movimientos.

La eliminación de los estados persistentes suscita la cuestión de la integridad de las referencias vista en el primer capítulo. JDO permite asegurar la integridad de las referencias con la implementación de una interfaz `InstanceCallbacks` en las clases del dominio. En este ejemplo de banco son expuestos dos casos, la eliminación de un cliente y la eliminación de una cuenta: cuando se elimina un cliente este deja de ser titular de sus cuentas, de cuyos fondos el Banco sabe bien que hacer, cuando una cuenta es eliminada, los movimientos correspondientes también son eliminados (¡recordar que es un ejemplo! en la realidad los datos pasarían a un sistema de históricos).

```
protected void ejecutarOperacion()
{ Cliente cli;
  ConsultaCliente concli = new ConsultaCliente(nif);

  concli.ejecutarOperacion();
  cli = concli.obtenerCliente();
  if (cli!= null && cli.carteraElimimable()) {
    synchronized(cli) {
        pm.deletePersistent(cli);
    }
  }
}
```

En la clase `Cliente`

```
public void jdoPreDelete()
{
  Cuenta cu;
  PersistenceManager pm = JDOHelper.getPersistenceManager(this);
  Iterator it = getCartera().iterator();
  while (it.hasNext()) {
    cu = (Cuenta) it.next();
    cu.getTitulares().remove(this);
        /*Los clientes eliminados dejan de ser titulares de sus cuentas*/
    if (cu.getTitulares().size() == 0 && !JDOHelper.isDeleted(cu)) {
      pm.deletePersistent(cu);
    }
  }
}
```

Listado 27 Propagar la eliminación de un cliente a las cuentas.

Los métodos de la interfaz `InstanceCallbacks` son invocados por la implementación JDO de forma automática para determinados eventos del ciclo de vida de las instancias persistentes, el programador no debería invocar los métodos esta interfaz

directamente. El objetivo es mantener la consistencia, no probar la validez de la operación, que es asunto para la lógica del negocio.

```
public void jdoPreDelete()  // En la clase Cuenta
{
/** Propagar la eliminación a los movimientos
   *   O la base de datos ofrece la eliminación en cascada
   *   O bien con JDO a nivel del servicio o en la aplicación
   */
   if (this.getTitulares().size() == 0) {
     return;
   }
  javax.jdo.PersistenceManager pm = JDOHelper.getPersistenceManager(this);
        pm.deletePersistentAll(this.apuntes);
   Iterator it = this.getTitulares().iterator();
   while (it.hasNext()) {
     Cliente cli = (Cliente) it.next();
     if (!JDOHelper.isDeleted(cli)) {
       cli.getCartera().remove(this);
       if (cli.getCartera().size() == 0) {
         pm.deletePersistent(cli);
       }
     }
   }
 }
```

Listado 28 Eliminar las instancias persistentes vinculadas a la Cuenta

El código anterior muestra una problemática interesante, el soporte de las relaciones entre los objetos persistentes. En relación con esta cuestión, JDO no maneja las relaciones entre los objetos persistentes, directamente sobre la base de datos, exige la recuperación del objeto a eliminar para ser eliminado, pero los objetos persistentes relacionados por agregación, instancias de vidas independientes, no actualizan automáticamente la consistencia de sus referencias, por eso los `InstanceCallbacks`. Otros planteamientos son posibles, la revisión 2.0 de JDO, resuelve esto, con la gestión automática incluida la eliminación de los estados persistentes, sin necesidad de recuperación y de forma automática.

3.2.1.6. Relaciones entre objetos persistentes: Agregación y Herencia en JDO

Las relaciones entre objetos persistentes Java de agregación, referencia y herencia con JDO, no son distintas de las relaciones entre objetos transitorios Java, si bien existen algunas restricciones y pautas a tener en cuenta.

Las referencias Java, utilizadas para definir atributos de instancia, de clase, parámetros o variables de métodos, pueden estar conectadas a objetos transitorios o persistentes indistintamente. JDO no exige la utilización de ciertos tipos concretos dedicados a ser referencias de objetos persistentes, más atrás los listados de este capítulo son ejemplos de ello. Pero no todos los tipos pueden llegar a ser objetos persistentes, los objetos ligados al sistema de ejecución (clases del sistema, objetos remotos…), no pueden ser persistentes. JDO permite la utilización de referencias polimórficas, referencias declaradas de un clase o interfaz en una jerarquía de herencia, que en ejecución son asignadas a un objeto subtipo descendiente de esa clase o interfaz. También es posible usar referencias, cuyo tipo sea una interfaz o la clase Object, como atributos de una instancia, procurando que en ejecución los tipos concretos asignados a estas referencias, serán tipos permitidos por JDO.

El otro elemento indispensable para expresar relaciones en Java, son las colecciones, JDO especifica los tipos Collection, Set, HashSet como tipos ofrecidos necesariamente por toda implementación JDO, y además, contempla otros tipos como

opcionales (Hastable, TreeSet, ArrayList, LinkedList, Vector, HashMap, TreeMap), cuyo soporte queda a elección de los fabricantes de implementaciones JDO. Los elementos componentes de las colecciones son especificados en el descriptor de persistencia, por defecto es tipo de los elementos en las colecciones es Object.

Los cambios producidos en los objetos persistentes son automáticamente sincronizados en sus estados almacenados, para los objetos dependientes agregados por referencia, pero no así para los objetos contenidos como parte integrante de otro (éste extremo se explica en los detalles de la especificación, más adelante).

¿Cómo afecta la persistencia a la herencia entre clases? La persistencia JDO de una clase es independiente de su posición en la jerarquía de herencia a la que pertenezca, que las instancias de una clase puedan perdurar, no depende de que su clase sea descendiente, o no, de otra clase; depende de que si la clase haya sido declarada en el descriptor de persistencia. Los atributos declarados como persistentes en una clase, son persistentes también en las subclases persistentes de esta.

Veámoslo con el siguiente diagrama de clases que presenta como ejemplo una jerarquía de clases que quiere representar las condiciones y productos financieros de crédito según el tipo de cliente, por supuesto, seguramente otra clasificación sería más apropiada, pero como ilustración es útil.

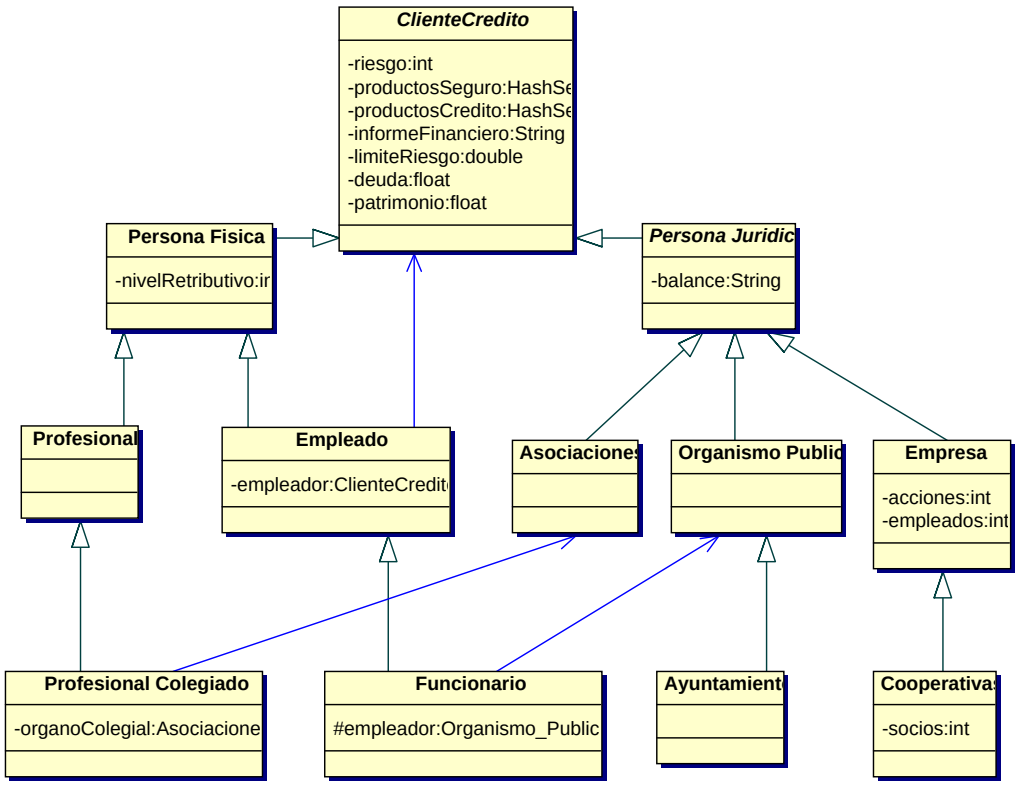


Figura 17 Diagrama de clases de clientes de crédito

Partiendo de la jerarquía de clases anterior se ilustra la interacción entre herencia y persistencia JDO mediante el siguiente código, que verifica la capacidad de cada clase para persistir sus instancias conforme a la declaración hecha en el fichero descriptor de persistencia empleado.

```
package casosdeuso;
```

```
/**
 * <p>Title: Persistencia y JDO. Ilustracion 2</p>
 * <p>Description: Desarrollo de ejemplos ilustrastivos para~nel proyecto
fin de carrera de Juan Mármol Castillo</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: Proyecto Fin de Carrera Persistencia y JDO</p>
 * @author Juan Mármol
 * @version 1.0
 */
import java.util.Iterator;
import java.util.Vector;
import javax.jdo.spi.PersistenceCapable;
import clientescredito.*;

public class PruebaHerencia
    extends AppOper
{
  private Vector vIns;

  public PruebaHerencia()
  {
    vIns = new Vector();
    vIns.add(Profesional.class);
…
    vIns.add(Empleado.class);
    vIns.add(Persona_Fisica.class);
…
    vIns.add(Cooperativas.class);
    vIns.add(ClienteCredito.class);
    vIns.add(Object.class);
  }

  public static void main(String[] args)
  {
    PruebaHerencia pruebaHerencia1 = new PruebaHerencia();
    pruebaHerencia1.ejecutar();
  }

  protected void ejecutarOperacion()
        {
    Iterator it = vIns.iterator();
    Object cls = null;
    String sidClase = null;
    while (it.hasNext()) {
      cls = (it.next());
      sidClase = ( (Class) cls).getName();
      System.out.println("Verificando persistencia para:" + sidClase);
      System.out.println(" Implementa Interfaz PersistenceCapable: " +
                         (
                         PersistenceCapable.class).isAssignableFrom( ( (Class)
                                                                    cls)));
      testPersit( (Class) cls, sidClase);
    }
  }

  protected void testPersit(Class clase, String strClase)
  {
    try {
      Object objTest = clase.newInstance();
      this.pm.makePersistent(objTest);
      System.out.println(" Las instancias de " + strClase +
                         " pueden persistir");
    }
    catch (Exception ex) {
      System.out.println(" Las instancias clase " + strClase +
          " no pueden persistir");
    }
```

```
  }
}
```

Listado 29 Ilustrando la relación entre Herencia y Persistencia

El código recorre la jerarquía de clases mostrando para cada clase si implementa la interfaz de JDO `PersistenceCapable`, necesaria para que pueda persistir la instancia con JDO, y verifica si es posible guardar una instancia. Utilizamos dos casos de ejemplo que muestran el nivel de independencia entre la herencia y la persistencia JDO, en una primera situación donde una clase abstracta, que no puede tener instancias concretas, está declarada como persistente y un descendiente indirecto necesita persistir; y en segundo caso, una clase con ancestros y descendientes, es declarada persistente de forma aislada.

Situación Clase Persistente y descendiente no directo

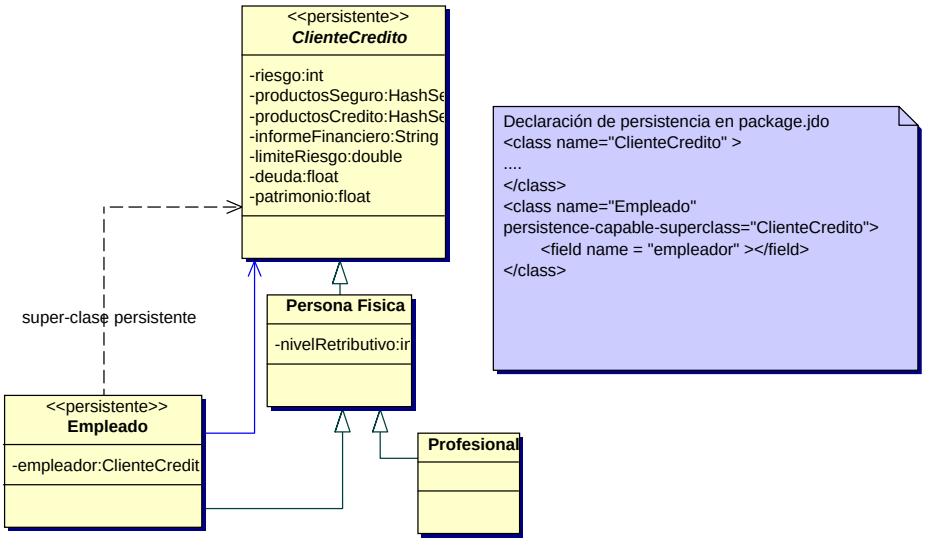


Figura 18 Diagrama de clases ilustrando la independencia entre persistencia y herencia.

¿Qué sentido tiene esta situación? La necesidad de persistir y manipular atributos heredados. Actualmente, JDO no gestiona automáticamente los atributos heredados que pertenecen a clases no persistentes. En el ejemplo anterior el atributo `nivelRetributivo`, no es controlado por las implementaciones de JDO. Esto puede suceder cuando los atributos en las clases intermedias no tienen que persistir.

La salida de la ejecución del programa para las clases de la figura anterior aparece como sigue:

```
Verificando persistencia para:clientescredito.Empleado
 Implementa Interfaz PersistenceCapable: true
 Las instancias de clientescredito.Empleado pueden persistir

Verificando persistencia para:clientescredito.ClienteCredito
 Implementa Interfaz PersistenceCapable: true
 Las instancias clase clientescredito.ClienteCredito no pueden persistir

Verificando persistencia para:java.lang.Object
 Implementa Interfaz PersistenceCapable: false
 Las instancias clase java.lang.Object no pueden persistir
```

Vemos como la herencia de implementación de la interfaz PersistenceCapable, no es suficiente para conseguir la persistencia de las instancias de una clase, ya que, es necesario además indicar a JDO, que clases pueden persistir en el fichero descriptor de persistencia.

Situación Superclase no persistente, descendiente directo persistente sin descendientes capaces de perdurar.

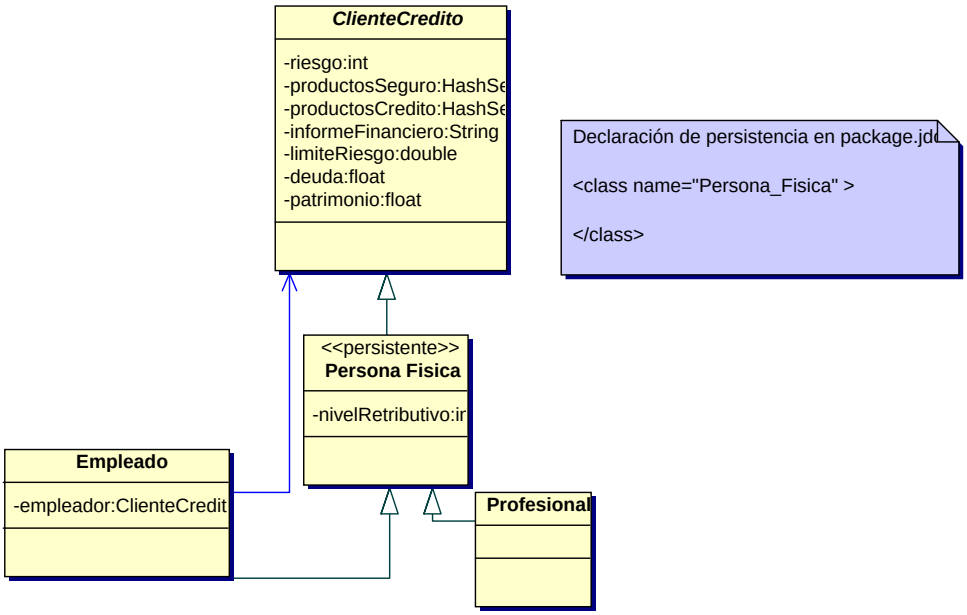


Figura 19 Diagrama de clases ilustrando la independencia entre herencia y persistencia

La situación descrita en la anterior figura tiene sentido en aquellas situaciones donde las superclases de una clase persistente, no contiene atributos que necesiten persistir, como aquellos calificados como transitorios (transient, volatile), o constantes (final), caso de atributos de clase que representan variables de semáforo, señales o constantes compartidas, suelen ser clases abstractas.

La ejecución del programa anterior Listado 29, para el descriptor de la Figura 19 es la siguiente:

```
Verificando persistencia para:clientescredito.Profesional
 Implementa Interfaz PersistenceCapable: true
 Las instancias clase clientescredito.Profesional no pueden persistir
Verificando persistencia para:clientescredito.Empleado
 Implementa Interfaz PersistenceCapable: true
 Las instancias clase clientescredito.Empleado no pueden persistir
Verificando persistencia para:clientescredito.Persona_Fisica
 Implementa Interfaz PersistenceCapable: true
 Las instancias de clientescredito.Persona_Fisica pueden persistir
Verificando persistencia para:clientescredito.ClienteCredito
 Implementa Interfaz PersistenceCapable: false
 Las instancias clase clientescredito.ClienteCredito no pueden persistir
Verificando persistencia para:java.lang.Object
 Implementa Interfaz PersistenceCapable: false
 Las instancias clase java.lang.Object no pueden persistir
```

La ilustración de las dos situaciones vistas, muestran el tratamiento de la herencia en JDO. Todavía hay otras situaciones que considerar, cuando es utilizado el modelo de identidad gestionado por la aplicación (identidad de aplicación), que exige seguir ciertas pautas [51], pero no son de interés para el enfoque de este apartado, más adelante, algunos puntos sobre la cuestión son tratados al revisar el contenido de la especificación.

3.2.2. Modelo de persistencia

JDO ha sido diseñada en base un modelo de persistencia, que como cualquier otro, queda caracterizado por:

- Unos fundamentos que dan soporte al modelo.
- Una serie de definiciones de los conceptos y elementos descritos en el modelo.
- Una serie de reglas, que determinan la funcionalidad del modelo, constituyen las restricciones, los requisitos y la forma operar el modelo.

Un modelo de persistencia es una representación que describe los conceptos, elementos (tipos y objetos) y el conjunto de reglas que rigen la interacción entre los elementos del modelo.

Algunos de los notables fundamentos del modelo de persistencia de JDO, proceden del modelo de persistencia impulsado por el estándar para Java del ODMG, frente al que JDO muestra sustanciales diferencias. La principal aportación asumida del ODMG es *la transparencia de datos*. Los conceptos esenciales en los que la especificación se sustenta, son:

- La persistencia transparente.
- La definición de instancias JDO, sus categorías y restricciones de tipo y herencia.
- La proposición del ciclo de vida de las instancias dentro del modelo.
- La idea generación automática de código.
- La persistencia dirigida por una clase control, gestor de persistencia, *PersistenceManager*.

3.2.2.1. Área de interés

Los objetos que interesan a JDO pesistir son los objetos de las clases del dominio, las clases con la lógica del negocio, definidas por los programadores.

3.2.2.2. Transparencia

La transparencia, se refiere a:

- La ilusión de que las instancias persistentes son manejadas como si estuvieran siempre disponibles en memoria. La navegación entre objetos provoca que los objetos visitados sean recuperados, a partir del estado almacenado, como si nunca hubieran sido destruidos de la memoria.
- La actualización implícita en los EIS (base de datos, sistema de archivos,…), del estado correspondiente a cada instancia modificada en una transacción que se consolida.
- La conversión automática entre los tipos de datos nativos de los sistemas de datos, donde el estado de lo objetos es guardado, y los tipos Java usados para los atributos de los objetos persistentes.

3.2.2.3. Instancias JDO

El modelo empleado por JDO limita las clases cuyas instancias pueden ser persistentes. Las clases que representan aspectos ligados al entorno de ejecución, a los elementos de procesamiento y procesos, no pueden ser persistentes, ni tampoco aquellas cuyo estado dependa de objetos con estado inaccesible o remoto. Instancias de las clases definidas en los paquetes como java.lang, java.net, java.io no podrán ser persistentes, no tiene sentido recuperar objetos ligados a un contexto de ejecución que dejo de existir.

Las instancias correspondientes a objetos que serán persistentes y manejados con JDO, son denominadas instancias JDO. El estado persistente de una instancia JDO, constituido por los atributos cuyos tipos puedan y deban ser persistentes, es gestionado por el servicio de persistencia que implementa la especificación JDO. Mediante un descriptor de persistencia escrito en XML escrito por el programador o quien administre el esquema de persistencia, se indica al Enhancer, que atributos serán tenidos en cuenta, en la modificación del código compilado Java de las clases cuyas instancias persistirán. Por defecto, todos los atributos declarados de los tipos soportados para persistir, salvo los calificados como **transient, static** y **final**, podrán formar estados persistentes. Todos los modificadores Java son soportados por JDO.

3.2.2.4. Objetos de primera clase y de segunda

JDO clasifica los objetos persistentes en dos categorías en función como son almacenados, si como objetos independientes directamente accesibles, o integrados dentro del estado almacenado de otro. Esto determina la necesidad de poseer identidad propia para ser referenciado por otros objetos, o prescindir de identidad por ser parte incluida de un objeto. Esto es similar a lo que sucede en lenguajes orientados a objetos como Eiffel o C++, con la semántica por referencia o valor.

Los objetos con identidad propia, son los *objetos de primera categoría*. Los objetos contenidos (embebidos) en otros, que no precisan tener identidad propia, son los *objetos de segunda categoría*. Para que un objeto pueda ser de primera categoría, su clase deberá implantar la interfaz `PersistenceCapable` y se deberá incluir las declaraciones oportunas en el descriptor de persistencia.

La pertenencia a una de estas categorías afecta a la funcionalidad de compartir, integridad y transparencia. Para los objetos de primera clase, la implementación de JDO debe asegurar la singularidad, que recuperado el objeto a caché sólo habrá una instancia que lo represente, el seguimiento de las actualizaciones y su sincronización con el estado almacenado. Los objetos de segunda categoría no son compartidos entre distintos objetos, no se asegura su singularidad y deben gestionar sus actualizaciones notificando a su objeto contenedor de primera clase. En el caso particular de los Arrays, la especificación contempla como opcional la gestión automática de las actualizaciones.

3.2.2.5. Identidad JDO

En JDO la identidad de un objeto persistente, es distinta de la identificación interna del objeto en la máquina virtual Java. Relaciona unívocamente identificador java interno, con el identificador empleado por el mecanismo de persistencia para localizar los estados almacenados correspondientes.

Cada instancia persistente de primera categoría tendrá un identificador de objeto JDO que permite su localización, asegurar su singularidad y lograr la sincronización entre estado almacenado y las modificaciones consolidadas de la instancia, esto es, cada objeto persistente tiene una identidad JDO.

La especificación incluye tres distintos tipos de identidad JDO: *identidad por base de datos, identidad por aplicación e identidad no perdurable*. La *identidad por base de datos* es manejada por la implementación de JDO o directamente por el mecanismo de persistencia subyacente, y es independiente del valor de los atributos de instancia. La *identidad por aplicación*, su valor depende de uno o más atributos, atributos clave, y su singularidad es asegurada por la implementación o el mecanismo de persistencia, los campos clave constituyen una clave primaria que identifica unívocamente a cada instancia; cada clave primaria debe ser representada por una clase con idénticos atributos en nombre y tipo. La identidad *no perdurable o provisional*, es utilizada para identificar a cada instancia JDO mientras permanece en la máquina virtual java, pero que no es usada, ni conservada por los mecanismos de persistencia, el acceso a los registros de actividad, es un ejemplo de este uso. Las implementaciones de JDO pueden optar por facilitar la identidad por aplicación, la identidad por gestor de datos o ambas. La identidad no perdurable es opcional. Toda implementación JDO deber proporcionar una clase de soporte de la identidad JDO para las clases que no utilicen de identidad por aplicación.

Las clases para las claves primarias usadas para la identidad por aplicación, deben ser conforme con los siguientes requisitos:

- La clase debe ser declarada pública.
- Implementar la interfaz *Serializable*.
- Debe disponer de constructor por defecto o sin argumentos.
- Los atributos no estáticos (*static*) deben ser del tipo *Serializable* y acceso público.
- Los nombres de los atributos utilizados como clave y los así declarados en el archivo descriptor de persistencia para las clases del dominio que usan esa clase clave, deben ser los mismos.
- La métodos ***equals()*** y ***hashCode()*** deben usar los valores de los campos.

3.2.2.6. Tipos de los Atributos persistentes

JDO restringe los tipos posibles de los atributos que persisten a:

- Los tipos primitivos, considerados de segunda categoría.
- Clases de objetos inmutables, como **Boolean, String, Locales, BigDecimal**,... pudiendo ser de primera o segunda categoría, a criterio de la implementación JDO.
- Clases de objetos mutables: **Date, HashSet**; la implementación de JDO, determina la categoría de su soporte, como primera categoría o segunda.
- Clases que implantan la interfaz *PersistenceCapable.*
- Tipo **Object** como tipo de primera categoría.
- Tipos de las interfaces **Collection**, obligatorias **Set** y **Collection**, otras categorías son elegidas por la implementación de JDO.
- Otras interfaces son consideradas como tipos de primera categoría.

3.2.2.7. Herencia

JDO persigue la ortogonalidad entre herencia y persistencia, que ambas puedan ser usadas sin interferencia mutua. En apartados anteriores queda ilustrado como dentro de

una jerarquía herencia de clases, las clases podrán disponer de persistencia, independientemente de su posición en la jerarquía. No obstante los diseños de clases deberán tener presente ciertas reglas:

- Los atributos declarados persistentes en el descriptor XML de una clase, no pueden ser atributos heredados esto es el comportamiento para manejar los valores de los atributos, no depende de los ascendientes de la clase declarada. Los atributos declarados persistentes de una clase serán persistentes en las subclases.
- Las clases sin la cualidad de persistir con algún descendiente inmediato persistente, deben incluir accesible un constructor sin argumentos.
- Cuando la identidad es establecida mediante la aplicación, la clase usada como clave para la identidad debe ser la misma para toda la jerarquía. Los atributos que designan la clave deben estar declarados únicamente en la clase menos derivada, más alta, de la jerarquía que implemente la interfaz *PersistenceCapable*. (una ilustración adecuada en [29])

3.2.2.8. Ciclo de vida

Podemos describir las distintas situaciones requeridas en el tratamiento del ciclo de vida de una instancia que persiste mediante un diagrama de estados.

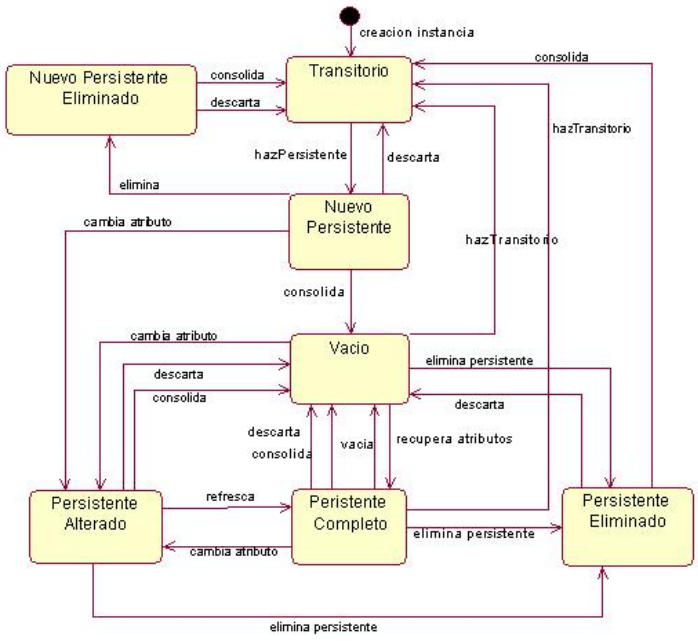


Figura 20 Diagrama de estados del ciclo de vida de instancia JDO

JDO establece diez estados de los cuales, siete son obligatorios, correspondientes a la funcionalidad especificada, y tres pertenecientes a funcionalidades opcionales. Veamos los estados obligatorios.

Estados

Transitorio

Todas las instancias son creadas inicialmente como transitorias, tanto aquellas cuyas clases sean persistentes como las que no. Su comportamiento es el propio de una instancia no persistente. El servicio de soporte de persistencia no interviene. Las nuevas instancias parten de este estado. También, alcanzan el estado Transitorio, todas las

instancias que dejan de ser persistentes, por eliminación consistente de los datos almacenados o por invocación del método `makeTransient(obj).` Una instancia cambia al estado *Nuevo Persistente* al suceder el evento *hazPersistente*, esto es, cuando es invocado el método `makePersistent(obj)` explícitamente o por aplicación de la persistencia por alcance, cuando una instancia persistente se almacena, las instancias referenciadas pasan a ser también persistentes.

Nuevo Persistente

Alcanzan este estado todas las instancias transitorias convertidas en persistentes durante la transacción en curso. Las instancias en este estado pasan a estar bajo el control de servicio de persistencia JDO. La transición a este estado provoca que las instancias pasen a ser manejadas por objetos de la clase `PersitenceManager` que controlarán las restantes transiciones del ciclo de vida y acciones relativas a la persistencia: asignación de identidad JDO, satisfacción de la singularidad, respaldo de los valores de los atributos para su revocación, coordinación de la participación por otras instancias, sincronización con los mecanismos de persistencia, etc. La modificación del valor de un atributo provoca la transición al estado *Persistente Alterado.* La consolidación de la transición en curso provoca el almacenamiento consistente del objeto en el sistema de datos asociado, y alcanzar el estado *Vacío*. La eliminación lleva al estado de *Nuevo Persistente Eliminado*.

Vacío

Este estado es para las instancias JDO que representan objetos almacenados con identidad ya determinada pero, cuyos atributos no están asignados o sincronizados con estas instancias que los representan, son instancias JDO vacías. El estado vacío satisface la singularidad de las instancias entre transacciones, mantiene la identidad JDO y la asociación con la instancia `PersistenceManager` correspondiente. Este estado puede ser no visible para las aplicaciones porque es necesario a nivel de la implementación JDO. La modificación de un atributo provoca la transición al estado *Persistente Alterado*. El acceso al valor de los atributos con su consiguiente recuperación activa la transición a *Persistente Completo*.

Persistente Completo

Este es el estado de las instancias JDO cuyos atributos han sido recuperados por la implementación JDO y que no han sido modificados en la transacción en curso. Cuando el valor de un atributo es accedido, si es necesario su valor es obtenido del sistema de datos asociado por el servicio de persistencia JDO. La modificación de un atributo provoca la transición al estado de *Persistente Alterado*. La eliminación de la instancia fuerza la transición al estado de *Persistente Eliminado*. La consolidación, revocación de la transacción en curso provoca la transición al estado *Vacío*, con la actualización adecuada en el sistema de datos. El descarte intencionado del caché de una instancia JDO, método ***evict(obj)*** (vaciar), conduce al estado *Vacío*.

Persistente Alterado

Cuando una instancia JDO es modificada y la modificación no ha sido consolidada sobre el sistema de datos, esta alcanza el estado de *Persistente Alterado*. La consolidación o revocación de las modificaciones efectuadas, cambia la situación al estado de *Vacío*. La eliminación lleva al estado de *Persistente Eliminado*. El refresco de los atributos con los datos actuales en la transacción en curso, conlleva conseguir el estado de *Persistente Completo*.

Nuevo persistente eliminado

Cuando una instancia JDO, el estado *NuevoePersistente*, es eliminada expresamente pasa a éste estado provisional, en tanto se decide consolidar o rechazar la eliminación, que provoca la eliminación del estado almacenado, dado el caso. Terminada la transacción la instancia para a ser transitoria, al estados *Transitorio*.

Persistente Alterado

Una instancia en estado *Persistente Completo*, recuperada del mecanismo de persistencia, es marcada con el método `deletePersistent` o `delePersistenAll`, pasa a este estado temporal, mientras la transacción no es completada. Si la transacción es rechazada, la instancia pasa al estado de *Vacío*, lo que significa que quizás tenga que volver a ser refrescada desde el sistema de datos correspondiente. Si se consolida la transacción, entonces, la instancia pasa a ser transitoria, alcanza el estado *Transitorio.*

Como podemos ver el ciclo de vida descrito recoge todas las situaciones posibles relativas a la persistencia de una instancia y sus efectos.

3.2.3. Lenguaje de Consultas

JDOQL, Java Data Objects Query Language, es el lenguaje con el que interrogar a los mecanismos de persistencia, sobre las instancias persistentes que verifican las condiciones de la consulta, todo expresado con una sintaxis Java, que no obliga a conocer otro lenguaje sintácticamente distinto. En los apartados anteriores se ilustrado con ejemplos, como trabajar con JDOQL. Aclararemos en este apartado, algunos aspectos no vistos de JDOQL. Sabemos que el resultado de las consultas siempre es una colección, la colección es inmutable de las instancias seleccionadas. La clase `PersistenceManager` actúa como factoría de instancias de *Query* cuya clase debe ser proporcionada por la implementación JDO. Las consultas JDOQL, son manejadas mediante una interfaz de JDO, la interfaz `Query` El diseño de JDOQL y de la interfaz que manipula las consultas, tienen presente los siguientes objetivos:

- La neutralidad del lenguaje, independencia del lenguaje frente a los mecanismos donde persisten las instancias.
- Disponer de la capacidad para la optimización de las consultas.
- Contemplar dos modos de procesamiento de consultas, procesamiento en memoria o delegado a los sistemas de consulta de los gestores datos accedidos.
- Soportar conjuntos resultado enormes, esto es, iterar y procesar un elevado número de instancias con un limitado uso de los recursos del sistema.
- El uso de consultas compiladas, evitando repetir el proceso de preparación de las consultas.
- El anidamiento de las consultas. El resultado de una consulta puede ser el objeto sobre el que volver a refinar una consulta.

Tres son los elementos básicos requeridos para poder llevar a cabo una consulta a través de los objetos de la interfaz Query son: La clase de las instancias que son consultadas (clase candidata), el tipo de la colección resultado, bien *java.util.Collection* o ***Extent***; y el filtro de consulta, una expresión booleana Java que concreta el criterio de selección. Los filtros no pueden incluir la invocación de los métodos de los objetos consultados. Los

atributos de la clase candidata del tipo colección pueden ser incluidos en los filtros para verificar si están vacíos o incluyen un cierto elemento ligado a una variable. Las cadenas de caracteres pueden ser procesadas con los métodos de s*tartsWith* y *endsWith*. La interfaz permite la inclusión en a la condición de búsqueda de parámetros y variables que forman parte del filtro de consulta. El empleo de variables es tema de debate respecto de la movilidad, que los autores de JDO afirman clarificar en la próxima especificación de JDO.

3.2.4. Operaciones

Las operaciones necesarias para manejar el modelo de persistencia de la especificación JDO están recogidas en la serie de interfaces y clases que son descritas en la especificación. A lo largo del capítulo, sea ilustrado diverso código Java de los principales elementos necesarios para operar con instancias persistentes y JDO, ahora serán especificados de forma breve los principales componentes de la interfaz de programación que propone JDO. Para conocer detalles concretos, la propia especificación es el texto de referencia.

3.2.5. Interfaces y clases

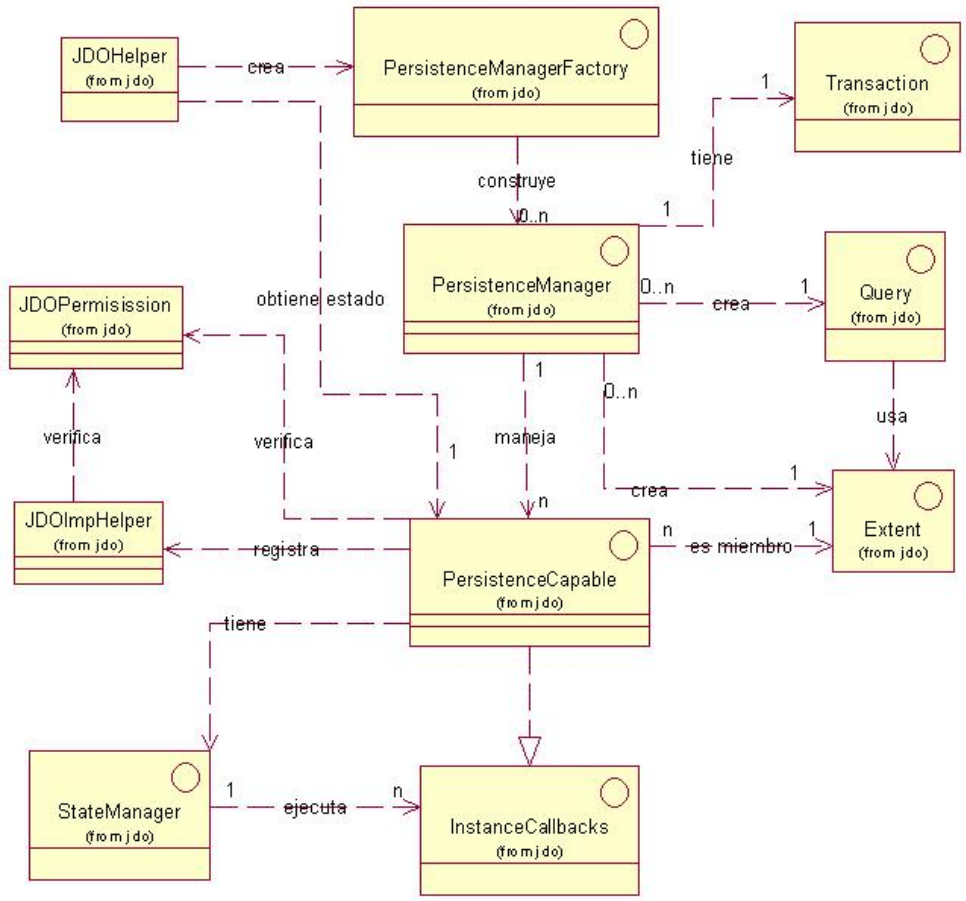


Figura 21Diagrama de clases de JDO

3.2.5.1. Interfaz PersistenceCapable

Esta es la interfaz que tiene que implantar toda clase que necesite de la capacidad de persistir sus instancias. Hay dos opciones: teclear el código o utilizar el procesador de código Java, ***JDO Reference* Enhancer** (procesador de referencias JDO) que inserta el código

ejecutable necesario en las clases Java compiladas, los atributos y métodos que permiten a la implementación manejar las instancias JDO. Y todo ello sin necesidad de modificar el código fuente java, basta con incluir las oportunas declaraciones en el descriptor de persistencia XML. Los métodos definidos en esta interfaz permiten obtener las referencias al gestor de persistencia, a la identidad JDO, al gestor de estado, obtener el estado en curso y modificar el estado a Persistente Alterado. Las aplicaciones no deberían usar esta interfaz, es una interfaz facilitada para los fabricantes de implementaciones de JDO. Los programadores de aplicaciones deberían utilizar los métodos de la clase `JDOHelper` en lugar de los métodos definidos en `PersistenceCapable`.

3.2.5.2. Interfaz PersistenceManager

Los componentes de aplicación con responsabilidad en la manipulación del ciclo de vida de las instancias y en la consulta de objetos almacenados, utilizarán esta interfaz. Los objetos de aplicación accederán a los servicios de esta interfaz mediante instancias de la implementación JDO que materializan esta interfaz. Habitualmente las referencias iniciales a los objetos que implementan *PersistenceManager*, son obtenidas mediante la interfaz `PersistenceManagerFactory`. Un objeto PersistenceManager debería ser capaz de manejar un número indeterminado de instancias. Las implementaciones JDO no pueden usar los objetos de usuario como barreras de sincronización, salvo para la sustitución del gestor de estado asociado a cada instancia JDO.

3.2.5.3. Clase JDOHelper

Las aplicaciones deberían usar los métodos de esta clase en lugar de los establecidos en `PersistenceCapable`. Esta clase recoge el enfoque recomendado para acceder al estado de persistencia de la instancia JDO de forma indirecta, y para obtener una referencia válida al generador de instancias gestoras de persistencia, *PersistenceManagerFactory*, cuya clase debe ser facilitada por el fabricante de la implementación JDO. `JDOHelper` ofrece los métodos estáticos de consulta del estado de persistencia, cambio al estado de *Persistente Alterado* y el método estático `getPersistenceManagerFactory`, que es la fórmula recomendada para preservar la movilidad en la obtención de la primera referencia a instancias de *PersistenceManager*, entre implementaciones de distintos fabricantes. Los métodos para acceder a la situación de persistencia de las instancias JDO de `JDOHelper`, delegan en las instancias, parámetro actual de la invocación, retornando valores adecuados aun cuando los objetos sean transitorios. Los componentes de aplicación encargados de manejar las instancias JDO utilizan esta interfaz para acceder al status de persistencia de las instancias JDO, el estado dentro del ciclo de vida de las instancias JDO.

3.2.5.4. Interfaz PersistenceManagerFactory

La responsabilidad de la creación de las instancias de `PersistenceManager` se concreta en la interfaz `PersistenceManagerFactory`. El enfoque recomendado es utilizar esta interfaz para obtener referencias válidas a objetos del tipo *PersistenceManager*. La referencias a los objetos de esta interfaz, deberían ser obtenidos a través del método estático de la clase `JDOHelper`, `getPersistenceManagerFactory`, según queda recomendado en la propia especificación de JDO. Los fabricantes de implementaciones JDO deben proporcionar una clase que implemente esta interfaz. La interfaz configura las características de las instancias gestores de persistencia (`PersistenceManager`). Estas características siguen el patrón Java Beans para facilitar el empleo de los métodos de configuración. Las características son los parámetros de conexión (usuario, fuente y contraseña), del modelo transaccional, concurrencia, etc. Es recomendada la utilización de factorías de conexiones (`ConectionFactory`) cuando pueden ser necesarias otras

características de conexión como el dispositivo de conexión, controlador, puertos, etc. La interfaz incluye un método para consultar vía objetos *Properties* las opciones no configurables, y otro para la obtener la colección de las características opcionales que dispone la implementación de JDO utilizada.

3.2.5.5. Interfaz Transaction

La interfaz `Transaction` establece la separación de responsabilidades dentro de las atribuciones de *PersistenceManager* ya que existe una relación uno a uno entre `PersistenceManager` y `Transaction`. Con esta interfaz se proporciona una gestión de las opciones disponibles para el manejo de transacciones, que puede ser aprovechada por los desarrolladores de clases y componentes de aplicación, en un entorno sin servidor de aplicaciones. Las estrategias de gestión transaccional ofrecidas por esta interfaz de JDO son dos: gestión transaccional mediante el mecanismo de persistencia subyacente y control optimista de transacciones. En el primero, exigido por la especificación JDO, desde el primer acceso a los datos hasta la consolidación o revocación de las modificaciones efectuadas, se mantiene activa una transacción en el mecanismo de persistencia correspondiente. En el segundo, opcional, cuando los datos son volcados al mecanismo es activada una transacción, verificando la consistencia de los datos modificados frente a actualizaciones paralelas. La arquitectura posibilita así, que un objeto `PersistenceManager` presente una vista consistente de los datos de los objetos que controla, sustentada en el sistema transaccional donde los datos persisten. La operación con transacciones implica el establecimiento de conexiones con los sistemas de datos, donde el estado de los objetos perdura. JDO no concreta cuales son las interfaces entre la implementación JDO y los componentes de conexión, ni su vinculación con las transacciones y el perfeccionamiento de las mismas (consolidación o revocación), esto queda interno a la implementación elaborada por cada fabricante. JDO si establece que la configuración de las conexiones es responsabilidad de la implementación realizada de `PersistenceManager`. En entornos de servidores de aplicaciones J2EE, la implementación debe establecer el vinculo entre `Transaction` y las conexiones mediante la aplicación de la *Arquitectura de Conexión J2EE (J2EE Connection)* proporcionando las implementaciones de los protocolos necesarios.

3.2.5.6. Interfaz InstanceCallbacks

El papel de la interfaz InstanceCallbacks es proporcionar un mecanismo para efectuar acciones que son asociadas a determinados eventos del ciclo de vida de la instancia JDO. Los métodos de esta interfaz son ejecutados por *StateManager*. De esta forma es posible la incorporación de reglas de integridad complejas en el ciclo de vida de la instancia JDO. El código de las acciones y la declaración de implementación no son automáticas, deben ser codificadas por el programador. Los eventos considerados del diagrama de estados más atrás son:

- *Recupera atributos*: Tras la carga de los atributos a la instancia, del grupo por defecto de extracción, se efectúa el método `jdoPostLoad`.
- *Consolidación*: Previa a la sincronización de las modificaciones con el estado almacenado, es invocado el método `jdoPreStore`.
- *Elimina* y *Elimina Persistente*: la transición a los estados *persistente eliminado* y *nuevo persistente eliminado* conlleva la ejecución del método `jdoPreDelete`.
- Los eventos que conducen al estado *Vacio*, provocan la ejecución del método `jdoPreClear`.

3.2.5.7. Interfaz Query. El interfaz con JDOQL, el lenguaje de consultas

La interfaz `Query` permite recuperar instancias JDO mediante consultas con criterios de búsqueda expresados con esta interfaz y el lenguaje de consulta de JDOL, descrito en apartados anteriores.

3.2.5.8. Interfaz Extent

Cada instancia de la clase `Extent` representa la colección completa de todas las instancias de una clase almacenadas en el sistema de datos accedido. También es posible que decidir incluir o excluir las instancias de las subclases de la clase asociada, base de la colección del *Extent.* Los usos de `Extent` son dos: iterar sobre todas las instancias de una determinada clase y ejecutar una `Query` en el gestor de datos que contiene todas instancias de una cierta clase. Esto permite el manejo de grandes conjuntos de datos y la gestión de los recursos afectados. Las consultas efectuadas sobre una instancia *Extent,* son procesadas en el sistema gestor de datos donde persisten los estados de las instancias consultadas.

3.2.5.9. Interfaz StateManager

Los programadores no deben acceder a esta interfaz, que permite controlar el comportamiento de las clases que pueden persistir, las que implantan `PersistenceCapable` que es la única interfaz que accede a `StateManager`. La implementación de JDO debe proporcionar una clase que implemente esta interfaz. Los métodos de esta interfaz recogen la consulta sobre el estado persistente, el acceso y modificación de los atributos, y la transición entre algunos estados del ciclo de vida de instancias JDO. El código generado automáticamente para las clases *PersistenceCapable* incluye las invocaciones a los métodos de esta interfaz. La especificación JDO no impone un modelo de servicio concreto de control del estado de las instancias JDO, por tanto, pueden ser una única instancia *StateManager* quien sirve o múltiples. Esta interfaz genera la secuencia de invocaciones de los métodos relativos a los cambios al ciclo de vida de instancias JDO, recogidos en la interfaz `InstanceCallbacks`.

3.2.5.10. Clases de JDO Exception y derivadas

JDOException es la clase base para todas las excepciones JDO.

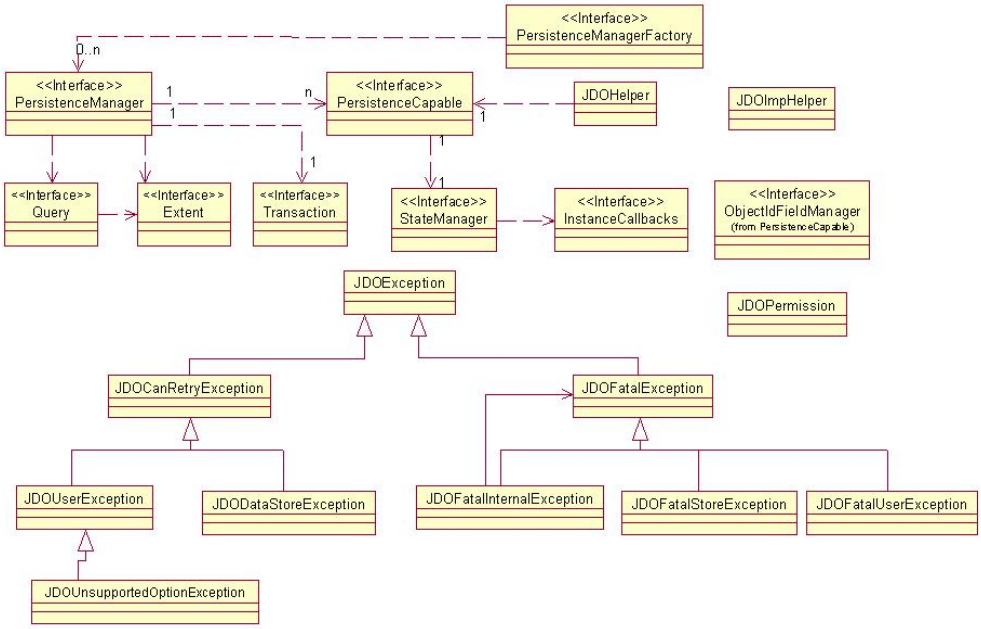


Figura 22 Diagrama de clases de Excepcion JDO

Esta clase y derivadas permiten la gestión disciplinada de las excepciones producidas en tiempo de ejecución debidas a situaciones error provocadas por un contexto de ejecución inadecuado, bien por un problema de la infraestructura o por el uso incorrecto de los servicios. Las clasificación de las excepciones responde a la fuente u origen del error: el usuario, el mecanismo de persistencia o la implantación de JDO; y a la posibilidad de corregir y reintentar. Errores comunes pueden ser corregidos en ejecución sin necesidad de plantear un estilo de programación defensivo.

3.2.5.11. Clase JDOImplHelper

Esta es una clase de asistencia para el lado de la implementación JDO, facilita un registro de meta datos de las clases, evitando la necesidad de reflexión en tiempo de ejecución. Las clases con capacidad para persistir (`PersistenceCapable`) registran la meta información durante la iniciación.

3.2.5.12. Clase JDOPermission

Esta clase permite reservar determinadas operaciones a la implementación JDO, operaciones que no deberían ser ejecutadas por otros objetos, de otro modo sería posible el acceso a información privilegiada sobre los meta datos de las clases persistentes, o provocar contenidos inconsistentes entre aplicación y sistemas de datos. Esta interfaz no será usada por las aplicaciones.

3.2.5.13. Interfaz ObjectIdFieldSupplier

Esta interfaz interna a `PersistenceCapable`, es usada para copiar campos desde el identificador de objetos de la instancia, al gestor de campos empleado por el manejador del estado de la instancia asociado, objeto de la clase *StateManager*. Esta interfaz no será usada por las aplicaciones. Cada método de esta interfaz facilita la copia según el tipo de datos desde el gestor de los valores de los atributos al atributo usado en la identificación de una instancia persistente.

3.2.5.14. Interfaz ObjectIdFieldConsumer

Esta interfaz interna a `PersistenceCapable`, es usada para copiar los campos de identificador de objetos de cada instancia JDO desde el gestor de campos correspondiente que controla el estado de la instancia. Esta interfaz no será usada por las aplicaciones. Cada método de esta interfaz facilita la copia según el tipo de datos desde los atributos clave de la instancia al gestor de los valores de los atributos.

3.2.5.15. Interfaz ObjectIdFieldManager

Esta interfaz interna a `PersistenceCapable`, extiende los dos anteriores por conveniencia de implementación.

Hasta aquí la descripción de los principales aspectos de las interfaces y clases que forman el paquete de servicios que debe ofrecer cualquier implementación JDO. Ahora pasamos a desgranar otros elementos que forma parte de la especificación.

3.2.6. Descriptor de Persistencia XML

Más atrás, en repetidas ocasiones ha quedado dicho que JDO requiere un archivo en XML, donde se describe cuales son las clases cuyas instancias podrán persistir y cómo objetos y datos están vinculados. El descriptor de persistencia es la base de conocimiento con la meta información necesaria, para generar el código de persistencia apropiado y para traducir objetos a datos y viceversa, definir la correspondencia objeto-dato.

El contenido de los archivos utilizados cómo descriptor de persistencia, viene determinado por la gramática establecida en la especificación JDO dentro de su capítulo 18. El fichero es utilizado por el procesador de código, procesador de referencias JDO (*Enhancer)*, para añadir el código apropiado en las clases persistentes y en las referencias a las instancias de estas clases. También la implementación JDO podría usar la información del archivo en tiempo de ejecución.

Destacar que JDO no especifica como establecer la correspondencia entre objetos y datos. Son los fabricantes de implantaciones JDO, quienes proporcionan mediante el mecanismo de extensión previsto, como especificar la relación entre datos y objetos.

3.2.6.1. Elementos de descripción. Etiquetas XML

JDO establece que el archivo descriptor debe ser denominado con la extensión *jdo* y el nombre la clase que tendrá instancias persistentes, con el nombre del paquete que contiene las clases a considerar o simplemente package.jdo ubicado donde el mismo paquete cuyas clases podrían persistir.

El capítulo 18 de la especificación JDO define el documento de definición de tipos, *DTD*, que determina la gramática que rige las definiciones contenidas en cada descriptor de persistencia. Las etiquetas principales que encontramos en un descriptor persistencia JDO son:

- **<jdo>** Elemento raíz del documento que debe aparecer dentro del ámbito de esta etiqueta estarán contenidas las definiciones de uno o más paquetes y especificaciones de extensiones de fabricante.
- **<package>** Establece el ámbito de la especificación de las clases a modificar para la persistencia de sus instancias.
- **<class>** Define las características de persistencia para la clase que referencia: modo de identidad, atributos clave, características de los atributos y si dispondrá de *Extent*.
- **<field>** Esta etiqueta es usada para especificar las características de los atributos de instancia: ser persistente o solo transaccional, formar parte de la clave primaria para identidad por aplicación, gestión de los nulos, formar parte del grupo por defecto de extracción, ser recuperado en la primera lectura del sistema de datos, y estar agregado por valor o por referencia para los tipos no primitivos.
- **<collection>** Los atributos del tipo colección son especificados mediante esta etiqueta y sus términos son el tipo de elementos contenidos y modo de agregación.
- **<array>** Identifica a los atributos de instancia del tipo *array*, permite especificar el modo de agregación de los elementos contenidos.
- **<extensión>** Esta etiqueta y los elementos que incorpora, constituyen el mecanismo de extensión que facilita JDO para que los fabricantes incorporen nuevas funcionalidades, nuevas características y el modelado de la relación objeto datos.

Veamos un ejemplo extraído de la propia especificación de JDO, modificado con tabulaciones y color para facilitar su lectura

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "jdo.dtd">
<jdo>
```

```
          <package name="com.xyz.hr">
             <class name="Employee" identity-type="application"
objectidclass="EmployeeKey">
                <field name="name" primary-key="true">
                   <extension vendor-name="sunw" key="index" value="btree"/>
                </field>
                <field name="salary" default-fetch-group="true"/>
                <field name="dept">
                   <extension vendor-name="sunw" key="inverse" value="emps"/>
                </field>
                <field name="boss"/>
             </class>
                <class name="Department" identity-type="application"
objectidclass="DepartmentKey">
                    <field name="name" primary-key="true"/>
                    <field name="emps">
                        <collection element-type="Employee">
                            <extension vendor-name="sunw"
                                key="element-inverse" value="dept"/>
                        </collection>
                    </field>
             </class>
          </package>
       </jdo>
```

Listado 30 Ejemplo deFichero descriptor de persistencia

El listado anterior específica qué podrá perdurar y algo más. Podrán persistir los objetos de la clase Empleado y Departamento en sus atributos señalados; cada clase usa el modelo de identidad por aplicación que requiere de sendas clases para claves primarias según la etiqueta *objectidclass*, las clases `EmploeeKey` y `DepartmentKey`; el atributo `name` además de persistir su valor, es la clave primaria en su clase. El atributo `emps` es una colección cuyos elementos son del tipo `Employee`. Las marcas *extensión vendor*, en este caso de Sun, concretan aspectos ligados a la implementación JDO, como el uso de índices btree y de relaciones inversas entre los atributos que vinculan ambas clases Empleado y Departamento.

3.2.7. Guía para la portabilidad

Uno de los objetivos de JDO es conseguir que las aplicaciones sean portátiles entre implementaciones realizadas por fabricantes distintos, esto decir, una aplicación debería funcionar sin cambios utilizando implementaciones de JDO de fabricantes diferentes. En el capítulo 19 de la especificación se recogen las recomendaciones de movilidad que aparecen dispersas por la especificación. Si una aplicación sigue las reglas de movilidad, entonces funcionará con cualquier implementación JDO que sea conforme con la especificación.

Las primeras recomendaciones para conseguir un código ubicuo (que funcione en todas partes) son sobre el tratamiento de las características opcionales que los fabricantes podrían soportar en sus implementaciones, capturar la excepción de característica no soportada y programar ignorando las capacidades opcionales, evitando la dependencia con la infraestructura de JDO utilizada. Otro aspecto con impacto para lograr aplicaciones ubicuas es el modelado de objetos, para ello las referencias entre objetos declarados `PersistenceCapable` deben ser de primera categoría, las instancias de segunda categoría, vectores incluidos, no deben ser compartidas entre objetos distintos, no se debe incluir en la signatura métodos o atributos cuyo nombre comience por jdo, en el soporte de la identidad JDO cada fabricante puede decidir entre usar identidad fijada por la aplicación, por la base de datos o ambas, las aplicaciones no deberían requerir de los estados opcionales o el estado vacío del ciclo de vida de las instancias JDO. En cuanto a la conectividad a diferentes bases de datos, las aplicaciones deberían evitar que su manejo del bloqueo durante las transacciones, sea otro distinto de la lectura consistente proporcionada

por los mecanismos de persistencia. La especificación incluye otras consideraciones para que el código funcione en todas partes y con cualquier fabricante, que ya fueron mencionadas anteriormente, y no serán repetidas de nuevo en este apartado. Obviamente, empleando una tecnología y un lenguaje de programación orientados objetos, es posible conciliar la independencia de la implementación y la ganancia de características especiales, sin renuncia, precisamente la flexibilidad de seleccionar el código ejecutable apropiado en el trascurso de la ejecución, es una de las ventajas de la programación orientada objetos dotada de polimorfismo.

3.2.8. Procesador de Referencias JDO

El capítulo 20 de la especificación JDO, detalla todo cuanto es necesario en tiempo de ejecución, para satisfacer el contrato entre las clases dotadas de persistencia y los gestores de persistencia de cualquier implementación JDO. Las clases ampliadas con las modificaciones necesarias para implementar la interfaz *PersistenceCapable*, pueden ser sincronizadas sus instancias persistentes con los correspondientes estados almacenados en los mecanismos de persistencia.

La propuesta hecha desde la especificación, es modificar el código ejecutable de la clases para añadir el código objeto preciso, que hará efectiva la persistencia transparente. El código objeto de acceso a la lectura y escritura de los atributos de las instancias, es sustituido por un código equivalente que invoca nuevos métodos añadidos a la clase, que garantizan el correcto tratamiento de la persistencia. Son añadidos nuevos atributos para facilitar el tratamiento de la introspección, la transparencia en tiempo de ejecución, la gestión del estado del ciclo de vida, el tratamiento de la herencia, la serialización y la gestión de la identidad y claves primarías. El programa encargado de ampliar el significado de las referencias JDO es denominado como ampliador o mejorador, JDO Enhancer. La especificación indica que las implementaciones pueden escoger y utilizar otras aproximaciones como la de generar código fuente procesando el código java de las clases. Mayoritariamente los fabricantes han optado por la generación de código objeto.

3.2.8.1. Objetivos

Los objetivos principales establecidos para llevar a cabo la modificación de las clases son:

- La compatibilidad y movilidad de las clases de aplicación entre implementaciones JDO de diferentes fabricantes.
- Compatibilidad binaria entre las clases así modificadas por distintas implementaciones de JDO.
- Mínima intrusión en los métodos y en instancias de las clases.
- Proporcionar meta datos en tiempo de ejecución sin necesidad de conceder permisos de reflexión sobre los atributos privados.
- No obligar a los programadores a escribir métodos para acceso y modificación de los atributos.
- Mantener la notación Java de navegación.
- Gestión automática de los modificaciones de valor de los atributos, sin intervención de las componentes de aplicación.
- Reducir la sobrecarga de código para las instancias transitorias.

- Soporte a todos los calificadores de clase y atributo.
- Preservar la las seguridad de las instancias persistentes evitando el acceso no autorizado.
- Facilitar la depuración mediante la inserción de números de línea para depuración.
- Uso transparente de la persistencia.

En la especificación podemos encontrar una explicación detallada de los cambios, que aquí veremos con un ejemplo incluido en la propia especificación, donde puede ver los cambios que serían introducidos en el fuente de clase modificada. El código original aparece en azul, el código añadido esta en negro, el fichero fuente original es como sigue

```
package com.xyz.hr;
class EmployeeKey {
                int empid;
        }
class Employee
        {
        Employee boss; // campo de posición relativa 0
        Department dept; // campo de posición relativa 1
        int empid; // campo de posición relativa 2, campo clave
        String name; // campo de posición relativa 3
} // end class definition
```

Listado 31 Clase original Empleado y su clave primaria

El código fuente correspondiente a la clase ampliada con la capacidad de persistir, es el siguiente con algunos comentarios para que sea más fácil de entender:

```
package com.xyz.hr;
import javax.jdo.spi.*; // generado por el enhancer
class EmployeeKey {
                int empid;
        }
class Employee
        implements PersistenceCapable // generado por  enhancer...
        {
        Employee boss; // campo de posición relativa 0
        Department dept; // campo de posición relativa 1
        int empid; // campo de posición relativa 2, campo clave
        String name; // campo de posición relativa 3
        // Campos añadidos
        protected transient javax.jdo.spi.StateManager jdoStateManager = null;
        //Manejador que controla el estado de la instancia
        protected transient byte jdoFlags =
                        javax.jdo.PersistenceCapable.READ_WRITE_OK;
        // Si no hay superclase, añadir lo siguiente:
        private final static int jdoInheritedFieldCount = 0;
        /* en otro caso,
        private final static int jdoInheritedFieldCount =
        <persistence-capable-superclass>.jdoGetManagedFieldCount();

        */
        private final static String[] jdoFieldNames = {“boss”, “dept”, “empid”,
        “name”};
private final static Class[] jdoFieldTypes = {Employee.class, Department.class,
 int.class, String.class};
// Indicadores para sincronizacion del estado memoria-base de datos
        private final static byte[] jdoFieldFlags = {
                MEDIATE_READ+MEDIATE_WRITE,
                MEDIATE_READ+MEDIATE_WRITE,
                MEDIATE_WRITE,
                CHECK_READ+CHECK_WRITE
        };
        // Si no hay superclase, añadir lo siguiente:
        private final static Class jdoPersistenceCapableSuperclass = null;
        /* en otro caso,
        private final static Class jdoPersistenceCapableSuperclass = <pcsuper>;
```

```
        private final static long serialVersionUID = 1234567890L; //valor para
serialización
        */
        //Método de inicialización generado
    static {
    javax.jdo.spi.JDOImplHelper.registerClass (
                Employee.class,
                jdoFieldNames,
                jdoFieldTypes,
                jdoFieldFlags,
                jdoPersistenceCapableSuperclass,
                new Employee());
    }
    // Consulta de estado añadidas
    public final boolean jdoIsPersistent() {
            return jdoStateManager==null?false:
                        jdoStateManager.isPersistent(this);
    }
    public final boolean jdoIsTransactional(){
                return jdoStateManager==null?false:
                        jdoStateManager.isTransactional(this);
    }
    public final boolean jdoIsNew(){
                return jdoStateManager==null?false:
                    jdoStateManager.isNew(this);
    }
    public final boolean jdoIsDirty(){
                return jdoStateManager==null?false:
                        jdoStateManager.isDirty(this);
    }
    public final boolean jdoIsDeleted(){
                return jdoStateManager==null?false:
                        jdoStateManager.isDeleted(this);
    }
    public final void jdoMakeDirty (String fieldName){
                if (jdoStateManager==null) return;
                        jdoStateManager.makeDirty(this, fieldName);
    }
    public final PersistenceManager jdoGetPersistenceManager(){
                return jdoStateManager==null?null:
                        jdoStateManager.getPersistenceManager(this);
    }
    public final Object jdoGetObjectId(){
                return jdoStateManager==null?null:
                        jdoStateManager.getObjectId(this);
    }
    public final Object jdoGetTransactionalObjectId(){
                return jdoStateManager==null?null:
                        jdoStateManager.getTransactionalObjectId(this);
    }
    /*jdoReplaceStateManager
        El método solicita al gestor de estado actual la aprobación para la
sustitución o la
        autorización del invocador para asignar el estado
        */
    public final synchronized void jdoReplaceStateManager
                (javax.jdo.spi.StateManager   sm)    {
        // lanza excepción si el sm vigente no permite el cambio
                if (jdoStateManager   != null) {
                        jdoStateManager       =
jdoStateManager.replacingStateManager(this,sm);
                } else {
                        SecurityManager sec   = System.getSecurityManager();
                        if (sec != null) {
                                sec.checkPermission( //      throws  exception  if  not
authorized
                                        javax.jdo.JDOPermission.SET_STATE_MANAGER);
                }
                jdoStateManager       = sm;
                }
        }
    //jdoReplaceFlags
    public final void jdoReplaceFlags () {
                if (jdoStateManager != null) {
                        jdoFlags = jdoStateManager.replacingFlags (this);
```

```
                }
        }
        // jdoNewInstance

/*El primero de los métodos auxiliares asigna el valor del parámetro actual gestor
de de estado para la nueva instancia creada
           */
        public PersistenceCapable jdoNewInstance(StateManager sm) {
        // Si la clase es abstracta, throw new JDOFatalInternalException()
                Employee pc = new Employee ();
                pc.jdoStateManager = sm;
                pc.jdoFlags = LOAD_REQUIRED;
                return pc;
        }
/*  El Segundo de los métodos auxiliares además del gestor de estado asigna los
valores de la clave primaria usada como identificador JDO
        */
        public PersistenceCapable jdoNewInstance(StateManager sm, Object
        oid) {
        // Si es abstracta, throw new JDOFatalInternalException()
                Employee pc = new Employee ();
                pc.jdoStateManager = sm;
                pc.jdoFlags = LOAD_REQUIRED;
        // Copia los campos clave en identificador JDO de la instancia
                jdoCopyKeyFieldsFromObjectId (oid);
                return pc;
        }
        // jdoGetManagedFieldCount
/*Este método generado devuelve el número de atributos persistentes propios más los
heredados que son manejados. Este método será usado durante la carga de la clase en
el entorno de ejecución.
           */
        protected static int jdoGetManagedFieldCount () {
                return jdoInheritedFieldCount + jdoFieldNames.length;
        }
//Métodos generados para lectura del valor de los atributos jdoGetXXX. Uno por campo
persistente
/*Los calificativos de acceso son los mismos que los declarados para el campo
original. Así pues, la política de privilegios es la misma que la del campo
original.
           */
        final static String
        jdoGetname(Employee x) {
        // este campo pertenece al grupo de extracción por defecto(CHECK_READ)
                if (x.jdoFlags <= READ_WRITE_OK) {
                        // ok to read
                        return x.name;
                }
                // el campo necesita ser extraido por el gestor de estado
                // esta llamada podría resultar en lugar del valor almacenad
instancia
                StateManager sm = x.jdoStateManager;
                if (sm != null) {
                        if (sm.isLoaded (x, jdoInheritedFieldCount + 3))
                                return x.name;
                        return sm.getStringField(x, jdoInheritedFieldCount + 3,
                                                                        x.name);
                } else {
                        return x.name;
                }
        }
        final static com.xyz.hr.Department
        jdoGetdept(Employee x) {
                // este no pertenece al grupo de extracción por defecto(MEDIATE_READ)
                StateManager sm = x.jdoStateManager;
                if (sm != null) {
                        if (sm.isLoaded (x, jdoInheritedFieldCount + 1))
                                return x.dept;
                        return (com.xyz.hr.Department)
                                sm.getObjectField(x,
                                jdoInheritedFieldCount + 1,
                                x.dept);

                } else {
                    return x.dept;
```

Métodos de creación de instancias

Métodos de lectura de atributo. Si el atributo no esta en memoria se recupera del gestor de estado que a su vez lo podría extraer de la base de datos.

```
                }
    }
    //Método de asignación de valor al atributo jdoSetXXX. Uno por campo persistente
/*Los calificativos de acceso son los mismos que los del campo original. Por tanto,
la misma política de privilegios de acceso origina es aplicada.
       */
    final static void
    jdoSetname(Employee x, String newValue) {
                // este campo pertenece al grupo por defecto
                if (x.jdoFlags == READ_WRITE_OK) {
                        // ok para leer o escribir
                        x.name = newValue;
                        return;
 }
    StateManager sm = x.jdoStateManager;
    if (sm != null) {
                sm.setStringField(x,
                        jdoInheritedFieldCount + 3,
                        x.name,
                        newValue);
 } else {
                x.name = newValue;
         }
    }
    final static void
    jdoSetdept(Employee x, com.xyz.hr.Department newValue) {
                // este campo no pertenece al grupo por defecto de extracción
                StateManager sm = x.jdoStateManager;
                if (sm != null) {
                        sm.setObjectField(x,
                                jdoInheritedFieldCount + 1,
                                x.dept, newValue);
                } else {
                        x.dept = newValue;
                }

    }
    // jdoReplaceField y jdoReplaceFields
/*El método generado jdoReplaceField obtiene un Nuevo valor desde el gestor de
estado asociado para un campo especifico en base a su posición. Este método es
invocado por el gestor de estado, cada vez que actualiza el valor del campo en la
instancia
       */
    public void jdoReplaceField (int fieldNumber) {
    int relativeField = fieldNumber - jdoInheritedFieldCount;
                switch (relativeField) {
                        case (0): boss = (Employee)
                                jdoStateManager.replacingObjectField (this,

       fieldNumber);
                                break;
                        case (1): dept = (Department)
                                jdoStateManager.replacingObjectField (this,

       fieldNumber);
                                break;
                        case (2): empid =
                                jdoStateManager.replacingIntField (this,

       fieldNumber);
                                break;
                        case (3): name =
                                jdoStateManager.replacingStringField (this,

       fieldNumber);
                                break;
                        default:
                        /* Si hay superclase, delegar a esa
                        if (relativeField < 0) {
                        super.jdoReplaceField (fieldNumber);
                        } else {
                        throw new IllegalArgumentException("fieldNumber");
                        }
                        */
                        // Si no hay superclase, throw an exception
```

Métodos de asignación de atributo. Traslada la modificación al gestor de estado, si es necesario.

```
                        throw new IllegalArgumentException("fieldNumber");
                } // switch

        }
        public final void jdoReplaceFields (int[] fieldNumbers) {
                for (int i = 0; i < fieldNumbers.length; ++i) {
                        int fieldNumber = fieldNumbers[i];
                        jdoReplaceField (fieldNumber);
                }
        }
        // jdoProvideField y jdoProvideFields
/*El método generado jdoProvideField entrega los valores al gestor de estado en base
a su posición, por ejemplo cuando el campo es consolidado persistente.
        */
        public void jdoProvideField (int fieldNumber) {
                int relativeField = fieldNumber - jdoInheritedFieldCount;
                switch (relativeField) {
                        case (0): jdoStateManager.providedObjectField(this,

           fieldNumber, boss);
                                  break;
                        case (1): jdoStateManager.providedObjectField(this,

           fieldNumber, dept);
                                  break;
                        case (2): jdoStateManager.providedIntField(this,

           fieldNumber, empid);
                                  break;
                        case (3): jdoStateManager.providedStringField(this,

           fieldNumber, name);
                                  break;
                        default:
                                  /* Si hay superclase, delegar a esa
                                  if (relativeField < 0) {
                                          super.jdoProvideField (fieldNumber);
                                  } else {
                                  throw new IllegalArgumentException("fieldNumber");
                                  }
                                  */
                                  //  Si  no  hay  superclasepc  superclass,  throw  an
exception
                                  throw new IllegalArgumentException("fieldNumber");
                } // switch
        }

        public final void jdoProvideFields (int[] fieldNumbers) {
                for (int i = 0; i < fieldNumbers.length; ++i) {
                        int fieldNumber = fieldNumbers[i];
                        jdoProvideField (fieldNumber);
                }
        }
        // jdoCopyField and jdoCopyFields
        /*Estos métodos copian los atributos desde otra instancia a esta. Estos podrían
ser utilizados por StateManager asociado para crear imagenes previas para su uso en
el descarte de transacciones. Para evitar problemas de privilegios, el método
jdoCopyField puede ser invocado unicamente cuando las instancias afectadas
pertenecen al mismo StateManager.
*/
        public void jdoCopyFields (Object pc, int[] fieldNumbers){
                // la otra instancia debe pertenecer al mismo StateManager
                // y su referencia no debe ser nula
               if (((PersistenceCapable)other).jdoStateManager
                        != this.jdoStateManager)
                        throw new IllegalArgumentException("this.jdoStateManager !=

           other.jdoStateManager");
                if (this.jdoStateManager == null)
                        throw new IllegalStateException("this.jdoStateManager ==

           null");
                // throw ClassCastException, si el otro clase es inadecuada
                Employee other = (Employee) pc;
                for (int i = 0; i < fieldNumbers.length; ++i) {
```

Métodos de transferencia de datos al gestor de estado.

```
                    jdoCopyField (other, fieldNumbers[i]);
                } // for loop
    } // jdoCopyFields

    protected void jdoCopyField (Employee other, int fieldNumber) {
            int relativeField = fieldNumber - jdoInheritedFieldCount;
                switch (relativeField) {
                case (0): this.boss = other.boss;
                        break;
                case (1): this.dept = other.dept;
                        break;
                case (2): this.empid = other.empid;
                        break;
                case (3): this.name = other.name;
                        break;
                default: // other fields handled in superclass
                // esta clase no tiene superclase, lanza una excepción
                throw new IllegalArgumentException("fieldNumber");
                /* Si tiene superclase, el campo se trata como sigue:
                super.jdoCopyField (other, fieldNumber);
                */
                        break;
                } // switch
    } // jdoCopyField
    // writeObject
/*Si no existe un método programado explicitamente writeObject, entonces uno
es generado. writeObject asegura que todos los atributos peristentes y
transacionales que son serializable, serán cargados en la instancia y que entonces
el la salida generada por defecto es volcada sobre el argumento.
        */
    private void writeObject(java.io.ObjectOutputStream out)
                        throws java.io.IOException{
                jdoPreSerialize();
                out.defaultWriteObject ();
    }
    // jdoPreSerialize
    /*El método jdoPreSerialize  asegura que todos los atributos persistentes y
transaccionales
    que son serializable son cargados en la instancia por delegación en el
StateManager.
        */
    private final void jdoPreSerialize() {
    if (jdoStateManager != null)
                jdoStateManager.preSerialize(this);
    }
    // jdoNewObjectIdInstance
//Este método añadido crea y devuelve una nueva instancia del identificador JDO,
usado //para la identificación por Aplicación.
    public Object jdoNewObjectIdInstance() {
                return new EmployeeKey();
    }

    public Object jdoNewObjectIdInstance(String str) {
                return new EmployeeKey(str);
    }
    // jdoCopyKeyFieldsToObjectId
/*Este método copia los atributos clave primaria desde otra instancia persistente o
desde otro ObjectIdFieldSupplier.
        */
    public void jdoCopyKeyFieldsToObjectId (ObjectIdFieldSupplier fs,
    Object oid) {
                ((EmployeeKey)oid).empid = fs.fetchIntField (2);
    }
    public void jdoCopyKeyFieldsToObjectId (Object oid) {
                ((EmployeeKey)oid).empid = empid;
    }
    /* jdoCopyKeyFieldsFromObjectId
Este método copia los atributos clave primaria a otra instancia persistente o
ObjectIdFieldConsumer.
        */
    public void jdoCopyKeyFieldsFromObjectId (ObjectIdFieldConsumer
    fc, Object oid) {
                fc.storeIntField (2, ((EmployeeKey)oid).empid);
    }
    protected void jdoCopyKeyFieldsFromObjectId (Object oid) {
```

Métodos para gestionar la serialización.

Métodos de generación de identidad para nuevas instancias.

```
                empid = ((EmployeeKey)oid).empid;
    }
} // end class definition
```

Listado 32Clase Empleado ampliada

Básicamente el código añadido consiste en interceptar los accesos y asignaciones a los atributos de las instancias persistentes para asegurar la recuperación desde y en correspondencia con el almacen de datos, delegando a la implementación estas responsabilidades. Otros métodos facilitan la interacción con la implementación, como la transferencia entre los campos de las instancias y el gestor de estado asociado. La gestión de la identidad JDO, su asignación y la copia de los campos clave primaria o la serialización, son otros de los métodos necesarios incorporados automáticamente. La serialización queda de forma que es posible la restauración en una instancia de la clase ampliada o en la clase original.

3.2.8.2. Limitaciones

La especificación establece algunas limitaciones:

- Duplicado de instancias (cloning): Las instancias copia de otra persistente, deben ser marcadas como transitorias. El Enhancer, modifica ciertos métodos invocados por el método `clone` pero hay dos situaciones no cubiertas: la invocación de un método sobre la copia y cuando la clase raíz de herencia no declara el método *clone*, el clon no queda desconectado, hasta que este produce un primer acceso al StateManager.
- Introinspección: No son introducidos cambios en la funcionalidad de introinspección, que es un objetivo para futuras versiones de JDO. Todos los campos son expuestos a las interfaces de programación. Sería de esperar la posibilidad de navegación entre instancias persistentes durante la introinspección.
- Referencias públicas: Los atributos públicos se recomienda que no sean declarados persistentes, la razón es que tal caso las clases que acceden estos atributos deben ser ampliadas con procesador de código.
- Herencia: Es necesario utilizar una misma clase para facilitar la identidad JDO por aplicación en una misma jerarquía de herencia. Actualmente no son gestionados los atributos de las clases ancestro que no son persistentes. Es una funcionalidad que será recogida en próximas versiones de JDO.

Estas restricciones pueden ser evitadas en ciertos rodeos y en la práctica, afectan a un número de situaciones reducido.

3.2.9. Integración con servidores de aplicaciones Java

La integración en servidores de aplicaciones Java de JDO es un tema amplio y complejo, que la especificación trata de forma breve y densa, cuando no se conoce J2EE. JDO ha sido diseñado considerando dos escenarios principales dentro de los servicios de aplicación Java: Session Beans con clases persistentes JDO para implementar objetos dependientes, y Entity Bean con clases persistentes JDO utilizando la persistencia manejada por los beans (BMP) o por el contenedor(CMP). La cuestión es oscura, pero hay unas cuantas ideas clave a tener en cuenta. La utilización de JNDI para localizar la factoría de creación de gestores de persistencia, `PersistenceManagerFactory`. La integración con las transacciones de servidor de aplicaciones, es preferible usar los objetos `Transaction` proporcionados por servidor de aplicaciones. Y no usar instancias JDO persistentes como retorno de invocaciones en referencias remotas hechas por los clientes conectados a los

servicios de aplicaciones. La especificación apunta ciertos patrones de código que ayudan a la implantar e integrar el uso de JDO en un servidor de aplicaciones.

3.2.10. Asuntos pendientes

La especificación JDO recoge en el último capítulo, una lista de cuestiones abiertas pendientes de resolver. Un total de quince cuestiones son recogidas. Algunas de las cuales son de especial interés para el enfoque aquí considerado.

- Transacciones anidadas.
- Puntos de salvaguarda y retroceso.
- Referencias entre instancias gestionadas por distintos gestores de persistencia.
- Interfaz de programación del procesador de código.
- Interfaz de programación para la recuperación anticipada de objetos.
- Manejo de objetos no estructurados enormes.
- Soporte para las relaciones.
- Desconexión y cierre de la factoría de conexiones.
- Cuestiones de manejo de cadenas en consultas.
- Proyecciones en las consultas.

Todas estos temas se pretende estén resueltos, en una próxima revisión de la especificación, que ya esta ha iniciado su curso.

3.2.11. La especificación: su organización y redacción.

JDO esta descrita en 24 capítulos y 3 apéndices, en algo más de un centenar y medio de páginas; a primera vista y acostumbrados a especificaciones mucho más largas, puede parecer que la especificación no sea completa, pero no es así. Podríamos distinguir seis bloques en la especificación:

Un primer bloque, capítulos 1 a 4, donde se delimita el contexto de la especificación: ámbito, objetivos, arquitecturas, actores implicados y sus papeles, y sendos escenarios ejemplo de los dos modelos de arquitectura considerados.

El segundo, dedicado a la definición del modelo de persistencia que propugna la especificación, capítulos del 5 al 6. Presenta el ciclo de vida de las instancias a persistir y el modelo de objetos para la persistencia, que se pretende sea el mismo modelo de objetos de aplicación y negocio utilizado en una correspondencia sin separaciones.

El tercer bloque formado por los capítulos 7 al 15, detalla los interfaces de programación a utilizar para manejar la persistencia.

Bloque cuatro, capítulo 16, presenta la integración entre JDO y los granos de Java para empresa.

El quinto, capítulos 17 a 22, se detalla la especificación de distintos aspectos técnicos a cubrir por los servicios ofrecidos con una implementación JDO desde la perspectiva de integración e interoperatividad.

Y un último bloque compuesto por el capítulo 23, que presenta la gramática BNF del lenguaje de consultas de JDO, y el capítulo final 24, que recoge una lista de cuestiones pendientes pospuestas hasta la siguiente versión de la especificación.

3.2.12. Apéndices

Al final de la documentación de la especificación de JDO, están incluidos unos apéndices que recogen:

- Las referencias a documentación Java de consideración para la especificación
- Las decisiones de diseño tomadas para determinar las interfaces y las alternativas de implementación.
- El historial de revisión del documento

3.2.13. Implementación de Referencia y Equipo de compatibilidad de la tecnología

JDO ha sido desarrollada siguiendo el proceso de estandarización impulsado por Sun, el Java Community Process (JCP). Este proceso de estandarización obliga a la realización de una prueba de concepto de la especificación y de una batería de pruebas de compatibilidad antes de que la especificación alcance la calificación de versión definitiva, y los productos puedan ser marcados con la conformidad a JDO. La implementación de referencia concreta y hace ejecutable la especificación, constituye la prueba de concepto de la tecnología. El equipo de compatibilidad de la tecnología, es el conjunto de pruebas, herramientas y documentación que permite a quien implementa la especificación verificar si su realización es conforme a la especificación.

JDO incluye ambos conjuntos de programas y documentación, que son clave para permitir efectuar las pruebas oportunas sobre la utilidad de JDO y la verificación de los productos que dicen ser compatibles con JDO. Todo ello permite manejar más adecuadamente los riesgos en la apuesta por esta tecnología

3.2.14. Qué no aborda JDO: Correspondencia objeto-dato

Quizás tras la lectura de los párrafos anteriores nos encontramos que aún considerando la lista de asuntos pendientes, parece faltar algo más: Cómo vincular los objetos y sus atributos con el modelo de datos de los sistemas de gestión de datos utilizados como almacén de los estados de los objetos. Desde el enfoque de este trabajo, la correspondencia objeto datos es un requisito que debe ser cubierto por los servicios de persistencia. En efecto, JDO no especifica como concretar la relación entre objetos y datos en los mecanismos de persistencia. El medio para expresar éste vínculo es incluir, en el descriptor de persistencia, la declaración la correspondencia mediante las etiquetas *extensión*. El problema de la correspondencia es relegado a nivel de la implementación JDO y de los mecanismos propuestos por su fabricante. Este aspecto de la persistencia es un eje central de la nueva versión de JDO 2.0, que esta en fase de estandarización.

3.3. Resumen

En éste capítulo la especificación JDO ha sido comentada, para poner de manifiesto los elementos principales de la especificación. La especificación esta enfocada a la persistencia del modelo de objetos del dominio, como un servicio de programación Java. JDO establece un modelo de persistencia para las clases del dominio de las aplicaciones consistente con el modelo de objetos Java, un modelo que no fuerza la utilización de elementos ligados a la implementación concreta del servicio de persistencia.

El ciclo de vida de las instancias JDO tiene siete estados esenciales que comprenden todas las etapas, que una instancia persistente recorre desde su creación, modificación hasta su destrucción persistente. Están especificados en JDO, otros estados de implementación opcional.

Las instancias persistentes necesitan de una identidad ligada a la persistencia. JDO ofrece tres modos de identidad: identidad por aplicación, identidad por gestor de datos e identidad no perdurable.

JDO concreta en once interfaces los contratos de programación básicos para conseguir una persistencia transparente y una facilidad para la recuperación selectiva de instancias mediante el lenguaje de consulta JDOQL.

Parte de las interfaces están dirigidas a los fabricantes de implementaciones JDO, para garantizar su interoperatividad. Un programador debería dominar para conseguir la persistencia de objetos, básicamente conocer y utilizar `JDOHelper, Transaction, InstanceCallbacks, Query` y `Extent.`

La especificación concreta cómo declarar que va a persistir mediante archivos xml. Los archivos descriptores de persistencia son utilizados por la implantación y por el procesador de referencias JDO, para generar el código necesario a incluir, en las clases con instancias persistentes. También, ha sido incluida en la especificación, las recomendaciones básicas para alcanzar un buen nivel de movilidad, entre diferentes productos conformes con la norma que JDO establece.

La especificación estándar Java Data Objects, desarrollada en el marco de Java Community Process, define una interfaz de programación de aplicaciones que posibilita la persistencia transparente, escalable desde sistemas empotrados a empresariales e integrable utilizando el estándar EJB y J2EE. Dos son los modelos de ejecución: dos capas sin servidor de aplicaciones y tres capas integrado en servidor de aplicaciones Java.

Capítulo 4
COMPARATIVA JDO VS OTROS

4.1. Ámbito de la comparativa

En el primer capítulo eran presentadas distintas tecnologías para guardar y recuperar datos Java persistentes: la serialización, JDBC, SQLJ y la adaptación para Java del ODMG (ODMG Binding for Java), y fue establecido un marco de requisitos a satisfacer por un servicio de persistencia orientado objetos. El anterior se ha presentado JDO.

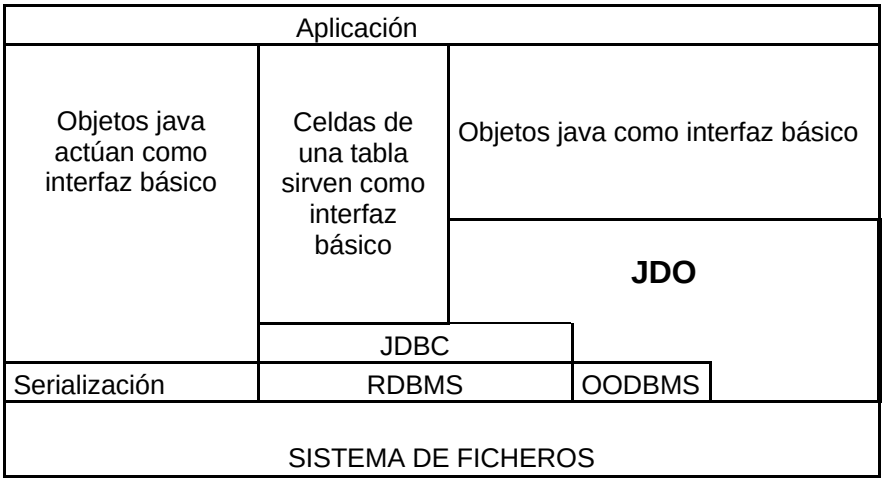
Este capítulo pretende comparar JDO frente a JDBC, SQLJ y ODMG adaptado para Java, desde las perspectivas del rendimiento, la productividad y el marco de requisitos, que fue establecido unos capítulos antes. La comparativa que se pretende elaborar, tiene presentes las opiniones, sobre la cuestión vertidas en los trabajos sobre la cuestión de K.Dittrich [10], Tornsten y D.Barry[35], E. Cecchet [37] y D. Jordan, trabajos entre los cuales se compara la serialización, JDBC y JDO [38].

A lo largo del trabajo realizado hasta el momento, han sido presentados algunos fundamentos, opiniones y comentarios que sustentan una base sobre la quien lee puede ya enjuiciar JDO frente a otras tecnologías de persistencia.

Para llevar a cabo la tarea de comparar, atendiendo a lo visto hasta el momento, se podría conseguir el objetivo de comparar con unas pruebas de rendimiento sencillas, ver que es más productivo y finalmente, elaborar un resumen donde se compara cuales de los requisitos propuestos son cubiertos por las tecnologías en comparación.

La siguiente tabla, obtenida del artículo de D.Jordan [38], explica la posición de JDO desde la perspectiva de una arquitectura en capas, que sirve para describir el contexto de la comparativa.

Tabla 3 Alternativas para persistencia en la comparativa



4.2. Comparativa

4.2.1. Rendimiento

El rendimiento es para muchos un factor primordial, en la elección de una tecnología. Una tecnología con aparentes grandes cualidades pero sin la capacidad para acometer determinadas cargas de trabajo, en tiempos de respuesta, considerados comúnmente aceptables, es desechada. Hay circunstancias bajo las que esta es, la premisa que más pesa en la decisión de elegir una tecnología y un producto.

Las pruebas de rendimiento entrañan una dificultad importante para extrapolar conclusiones generales, ya que, los resultados dependen de los productos y entorno concretos, que influyen en la medida, de forma que, los resultados cambian, incluso mucho, en función los productos escogidos y los ajustes hechos en la plataforma. De ahí, lo sofisticado y laborioso de unas auténticas pruebas de rendimiento, que serían objeto de un trabajo fin de carrera completo.

Por todo ello, a la hora de plantear este apartado de rendimiento, se buscaron pruebas estándar y sus resultados, para ser tenidos en cuenta. Fueron encontradas, tres pruebas relacionadas con este trabajo, pero para ninguna de la tres, están disponibles sus resultados, cuando estaban siendo redactados estos párrafos:

- RUBIS [53] es un banco de pruebas para patrones de diseño de aplicación y escalabilidad de rendimiento de servidores de aplicaciones. David Jordan ha realizado una la adaptación de esta para usar JDO, cuyos resultados viene presentado en diversas conferencias.
- OO7 [44] permite medir el rendimiento de diferentes productos JDO sobre la base un modelo usado en 1994, para medir el rendimiento de las bases de datos de objetos.
- TORPEDO-2004 [55] que mide la sofisticación de las herramientas de mapeo Objeto-Relacional (OR), en lugar del rendimiento.

En este trabajo, las pruebas están encaminadas a confirmar o descartar, los pronósticos hechos desde los siguientes apartados, si los resultados muestran, diferencias significativas entre tecnologías.

Diseñar una prueba de rendimiento exige decidir qué se mide, cómo y bajo que circunstancias o carga. ¿Qué considerar? Tiempo de respuesta por operación, número de operaciones por segundo, número de transacciones,... El rendimiento debe ser considerado con una ponderación de factores distintos, como rendimiento holístico y no integral.

El propósito de las prueba es medir, si el rendimiento comparado de JDO, es tan adecuado como para que JDO pueda sustituir a las otras tecnologías de la comparativa, o por el contrario, el rendimiento obtenido resulte tan pésimo, que no sea conveniente usar JDO. El objetivo no es medir el rendimiento de las bases de datos, por lo que no es necesario generar una prueba en concurrencia.

Las aplicaciones de negocio son el objetivo de JDO, donde el aspecto del rendimiento más relevante, es la percepción que tienen los usuarios de lo rápido que funcionan las aplicaciones, esto es, el tiempo de respuesta de las operaciones interactivas y de las operaciones que procesan volúmenes importantes de datos complejos. En razón de lo cual, las pruebas medirán el tiempo necesario para llevar a cabo las transacciones respectivas de crear un objeto compuesto de varios de forma interactiva y, por otro lado,

simular una actualización masiva con la creación y recuperación de una estructura de objetos compleja y voluminosa.

La primera prueba se simula con la ejecución de una transacción interactiva sobre el ejemplo del Banco, con la creación de una cuenta para un cliente nuevo. Podrían efectuarse otras, la consulta, actualización y eliminación, tras comentar los resultados de esta veremos, si son necesarias las otras operaciones.

La segunda de las transacciones, se simulará con la creación y el recorrido de los nodos de un árbol de cinco ramas con cinco niveles de profundidad, árboles de 19.531 nodos. Los árboles son un ejemplo de complejidad, el número de nodos escogido es importante para el entorno de ejecución, y además, son la forma de representar entidades formadas por numerosos componentes en muchos dominios de los negocios, ejemplos tenemos desde un documento como un contrato, la estructura de una organización, productos coches o sistemas CAD/CAM.

La comparativa considera las siguientes situaciones

- JDO frente a JDBC accediendo a sistemas relacionales más reconocidos. No son realizadas pruebas sobre SQLJ, porque la prueba con SQLJ no aportaría variaciones significativas, dado que SQLJ, utiliza JDBC para las sentencias SQL dinámicas, y mayor su ventaja es su sintaxis y la codificación de bucles de acceso a datos, por lo demás SQLJ y JDBC están al mismo nivel.
- JDO frente a ODMG accediendo a bases de datos de objetos más extendidas. Si bien ODMG 3.0, considera el acceso a los RDBMS, con la plataforma de pruebas y los productos disponibles, las diferencias mínimas para poder inclinar la balanza a favor de uno de los dos.
- La medición del rendimiento de la implementación de referencia (RI), que usa archivos indexados.

No tiene sentido medir el rendimiento, SQLJ y ODMG 3.0 para RDBMS, por su poco peso especifico en el mercado. Los fabricantes de RDBMS centran su interés en ofrecer un acceso JDBC. Incluso en OODBMS se ofrece acceso JDBC, caso de Versant, Matisse u Objectivity.

El entorno de ejecución de las pruebas esta configurado por un simple portátil PIV a 2.4GHZ con RAM 256MB, como mecanismos de persistencia para llevar a cabo las prueba se van utilizar la implementación de referencia de JDO, que maneja archivos indexados, las bases de datos de objetos compatibles con ODMG 3.0. ObjectStore, Versant y FastObject, y las bases de datos relacional MysSql, SqlServer y Oracle. Los sistemas gestores de bases de datos son los más representativos del momento actual..

El entorno ejecución no es ideal, otras configuraciones con un servidor de bases de datos en red, con unos cuantos ajustes en los parámetros de funcionamiento, permitirían obtener unos resultados más próximos a una situación real, pero aún así, es posible extraer conclusiones de interés al propósito que nos ocupa, porque las diferencias en comparación son muy relevantes, como se verá, persistiendo a buen seguro, con otros entornos de prueba.

4.2.1.1. Prueba de Transacción Interactiva en el Banco

La prueba consiste en el caso de uso de la creación de una cuenta para un nuevo cliente del Banco. Es una situación típica de las transacciones en los modelos de negocios, dominio que los sistemas relacionales imperan, que permite comprobar, si JDO es

apropiado para las aplicaciones dedicadas a los negocios en este estilo de aplicaciones interactivas.

De esta forma, son preparados dos programas Java, uno para JDO y otro para JDBC, que codifican la creación de los objetos implicados, clientes, los movimientos de apertura, las cuentas y la relaciones correspondientes entre estos, de cliente a cuenta, cartera, de cuenta a clientes, titulares, y del movimiento a la cuenta que lo contiene. Los programas ejecutados insertan los datos de los objetos en una instancia del SGBR MySql empleando el mismo controlador de acceso a datos favoreciendo la comparación. Se ejecutan dentro del mismo programa dos transacciones consecutivas, para amortiguar el efecto de inicio, con las carga inicial e creación de caches, etcétera.

Los programas adaptan el ejemplo de las clases AperturaCuenta y AppOper del Listado 19, para medir el tiempo de ejecución en milisegundos de la creación de dos cuentas para sendos nuevos clientes. La primera transacción queda afectada por el efecto inicial de carga de las clases y metadatos necesarios para operar, la segunda debería aprovechar la estructuras creadas en la primera transacción mejorando la respuesta. Esto facilita también la comparación entre JDO y JDBC.

La esencia del programa para JDO esta en el siguiente listado Listado 33:

```
  private Cuenta cu;
  /**
   * Apertura de cuenta para nuevo cliente
   * @param args Nif, CCC, credito, cantidad euros
   */
  public static void main(String[] args)
  {
    AperturaCuenta nc;
    if (args.length < NUMARGC) {
      System.err.print("Por favor,NIF, CCC,credito, cantidad");
      return;
    }
    nc = new AperturaCuenta(args[NIFARGC], args[CCCARGC],
                            Double.parseDouble(args[CREARGC]),
                            Double.parseDouble(args[CANARGC]));
    nc.ejecutar();
    System.out.println("Tiempo ejecución (ms) :" +
                       nc.obtenerTiempoRespueta());
    nc.ccc = args[CCCARGC + 4];
    nc.nif = args[NIFARGC + 4];
    nc.ejecutar();
    System.out.println("Tiempo ejecución (ms)" +
                       nc.obtenerTiempoRespueta());
…
```

En la clase ancestro AppOper:

```
  public void ejecutar()
  {
    try {
      long start, stop;
      start = System.currentTimeMillis();
      tx.begin();
      ejecutarOperacion();
      tx.commit();
      stop = System.currentTimeMillis();
      tiempoRespuesta = stop - start;
    }
    catch (Exception ex) {
      ex.printStackTrace(System.err);
      if (tx.isActive()) {
        tx.rollback();
      }
```

Listado 33 Prueba de rendimiento Interactivo JDO

El programa anterior será ejecutado sobre la implementación JDO RI y sobre dos productos comerciales JDO para RDBMS, Kodo Solarmetric y JDO Genie, ambos sobre JDBC accediendo a MySQL usando el mismo controlador conector. El programa adaptado para JDBC, se diferencia en la codificación de la operación que consigue la persistencia, el método concreto ejecutarOperacion,

```
protected void ejecutarOperacion()
{
  String strSQL;
  cu = new Cuenta();
  cu.aperturaCuentaNuevoCliente(nif, ccc, limcredito, aportacion);

  // AperturaCuentaNuevoCliente
  try {
    //Alta cliente
    strSQL = "Insert into Cliente (nif) values ( \"" + nif + "\")";
    pstmt = con.prepareStatement(strSQL);

    pstmt.execute();
    //Alta cuenta
    strSQL =
        "Insert into Cuenta ( codigocuenta,debe,haber,limiteCredito,saldo )" +
        " values ( \"" + ccc + " \", ?, ?, ? ,?)";
    pstmt = con.prepareStatement(strSQL);
    pstmt.setDouble(1, cu.getdebe());
    pstmt.setDouble(2, cu.gethaber());
    pstmt.setDouble(3, cu.getLimiteCredito());
    pstmt.setDouble(4, cu.getSaldo());
    pstmt.execute();
    //Alta Movimiento
    strSQL = "Insert into Movimiento ( codigocuenta," +
             " fechaOperacion, fechaValoracion," +
             " motivo, importe ) values" + "( \"" + ccc +
             "\", ?, ?, ? , ?)";
    pstmt = con.prepareStatement(strSQL);

    long fo, fv;
    Iterator it = cu.getApuntes().iterator();
    while (it.hasNext()) {
      Movimiento mov = (Movimiento) it.next();
      fo = mov.getFechaOperacion().getTime();
      fv = mov.getFechaValoracion().getTime();
      pstmt.setTime(1, new java.sql.Time(fo));
      pstmt.setTime(2, new java.sql.Time(fv));
      pstmt.setString(3, mov.getMotivo());
      pstmt.setDouble(4, mov.getImporte());
      pstmt.execute();
    }
    // Alta en relaciones Cartera y Titulares
    String valnc = " values( \"" + nif + "\", \"" + ccc + "\" )";
    strSQL = "Insert into cartera ( nif, codigocuenta )" + valnc;
    pstmt.execute(strSQL);
    strSQL = "Insert into titulares ( nif, codigocuenta )" + valnc;
    pstmt.execute(strSQL);
  }
  catch (Exception e) {
    e.printStackTrace(System.err);
```

```
try {
  con.rollback();
}
catch (Exception a) {
  a.printStackTrace();
}
```

Listado 34 Persistir una cuenta con JDBC

En un primer vistazo, la diferencia esencial entre la implementación JDO y la JDBC es, el aumento de las líneas de código necesarias para lograr la persistencia de los objetos con JDBC, hecho de relevancia para la productividad. Antes de obtener los resultados numéricos de la ejecución de los correspondientes programas y analizar los resultados, mencionar que se espera respecto al rendimiento de JDO frente JDBC y SQLJ.

JDO en acceso a bases de datos relacionales vs JDBC/SQLJ

Al contrastar JDO frente JDBC y SQLJ, estamos comparando tecnologías, que tienen un objetivo común de persistir datos, pero por lo demás, satisfacen objetivos diferentes, resulta entonces una comparación desigual. Las implementaciones JDO revisadas durante este trabajo, utilizan JDBC internamente; JDO esta situado encima, en una capa de programación que utiliza de los servicios de JDBC. Igualmente podría suceder con SQLJ, una implementación JDO podría modificar el código ejecutable de las clases con código embebido SQLJ estático y usaría JDBC para el acceso dinámico SQL. Bajo esta perspectiva, una implementación JDO sustentada sobre JDBC o SQLJ, es más lenta en el acceso a los datos del RDBMS **siempre**. ¿Pero que sucede con los objetos?

Insistir que el interés aquí, es el efecto sobre los objetos, no el acceso a los datos. SQLJ y JDBC no manejan objetos sin intervención del programador, los datos extraídos deben ser transformados en objetos por el programador ¡JDBC 3.0 y SQLJ manejan tipos de datos de usuario de SQL99, con la asistencia objetos para correspondencia que implementan el interfaz SQLData, pero sin transparencia! En resumidas cuentas, en el mejor de los casos cuando se trata con objetos, la latencia resultante debería tender a ser muy similares.

Es de esperar que con un modelo de objetos muy simple, JDO debería ser más lento que la solución JDBC sin modelo de objetos, debido a que JDBC no tiene la sobrecarga de del servicio JDO, con sus caches, control de modificaciones,…¿Pero con modelos de objetos complejos, qué sucede? En los modelos complejos es donde los mecanismos para mejorar el rendimiento desempeñan su papel, aventajando a una aplicación JDBC no preparada.

La opiniones en el foro de referencia para todo lo concerniente a JDO, www.jdocentral.com, como la de David Jordan, expresan que cuanto más complejo es el modelo de objetos (relaciones de herencia, clientela, colecciones grandes, navegación entre objetos...) mayor es el rendimiento de una buena implementación frente al equivalente JDBC (y SQLJ), que probablemente requiera un mayor esfuerzo de programación y obtendría peores resultados. En una situación ideal los programadores alcanzarían una solución cuyo código tendría mucho de común con una implementación de JDO con elementos como caches, bloqueos inteligentes, etc. Las pruebas de rendimiento con árboles más adelante, nos ayudarán a confirmar o refutar este planteamiento.

Echando la vista atrás buscando situaciones parecidas, los primeros sistemas de bases de datos relacionales se enfrentaron a una situación análoga frente a quienes usaban solo ficheros indexados como los sistemas VSAM, comercializado todavía hoy. Es de esperar importantes mejoras en el rendimiento de las implementaciones, según maduren los productos JDO, cuyos fundamentos técnicos recogen los avances de más de dos décadas de experiencia y desarrollo en la persistencia de objetos.

Aceptemos que efectivamente JDO es más lento que JDBC y SQLJ, manejando datos de forma interactiva. ¿Sería el retraso introducido suficiente como para desdeñar JDO? Pasemos a ver los resultados obtenidos para analizarlos con la perspectiva de los comentarios hechos.

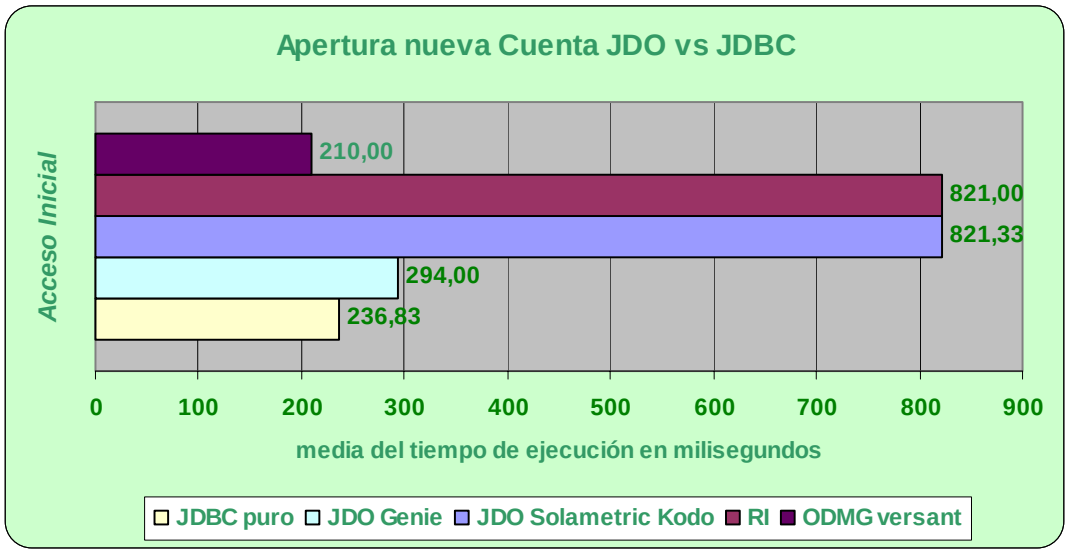


Figura 23 Tiempo inicial necesario para persistir los objetos vinculados a la creación de una Cuenta.

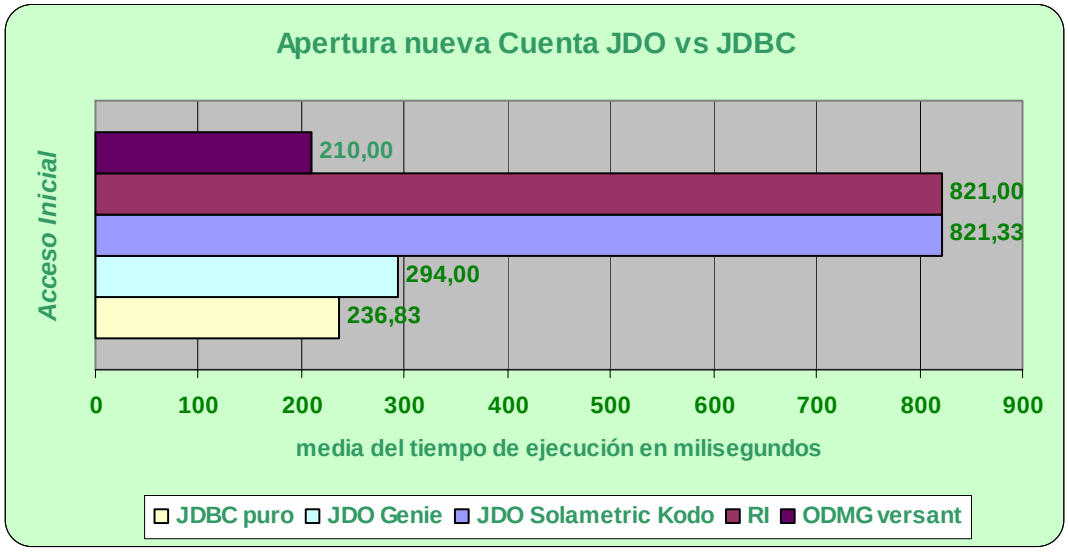


Figura 24 Tiempo para persistir los objetos implicados en la creación de una Cuenta, ignorando el retraso de primera operación.

Los resultados presentados han sido obtenidos de una batería de pruebas consistente, en ejecutar cada programa de forma aislada, libres de concurrencia desde la línea de comandos, en 30 ocasiones, agrupando series de 10 ejecuciones consecutivas del mismo programa, donde los cuatro peores resultados son descartados, y eligiendo como serie representativa, para cada programa, aquella de resultados más regulares donde los valores convergen entre sucesivas ejecuciones. Para las series finales se calculan las medias de tiempo presentadas en los gráficos. Este complicado procedimiento, pretende evitar la distorsión introducida por el entorno de ejecución.

Las figuras Figura 23 y Figura 24, muestran que con los mejores datos obtenidos, para el primer acceso, JDO es un rendimiento 24,14% menos eficaz, un 19,44% más lento que JDBC, (JDOGenie 294 ms, frente 236,83 ms de JDBC puro), debido en esencia, a la construcción inicial de las estructuras internas del servicio persistencia, en especial, el procesamiento del conocimiento del mapping objetos a tablas, conocimiento que en el caso de JDBC esta embutido en el código de forma invariable. En el segundo acceso, con las estructuras y el mapeo establecidos en memoria, para los mejores resultados se obtiene que, JDO es solo 4% menos eficaz que JDBC, un 3.85% más lento para este caso (Kodo 43,33 ms frente 41,67 ms de JDBC puro).

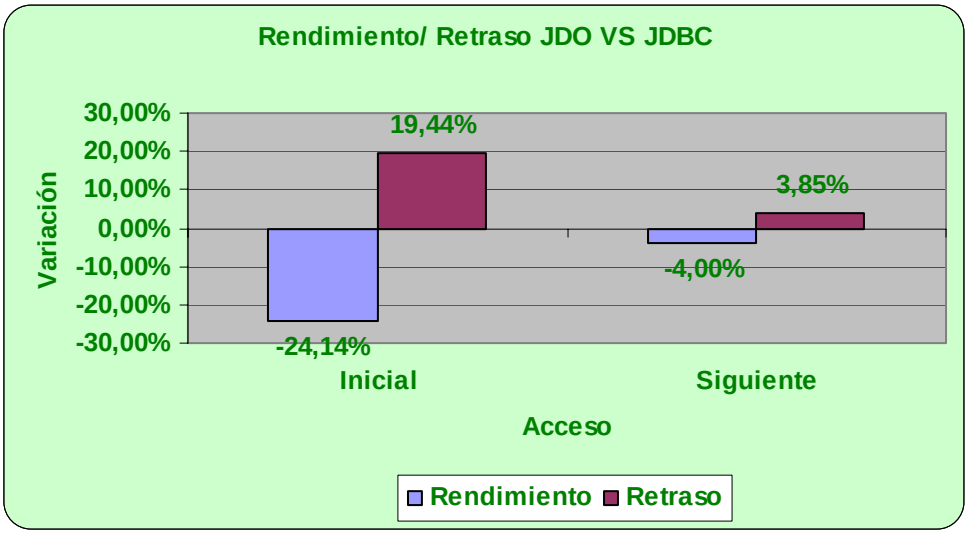


Figura 25 JDO vs JDBC variación del rendimiento y retraso

A la vista de los resultados, *JDO JDBC es más lento que JDBC en transacciones interactiva con modelos sencillos de objetos*, confirmando los argumentos expuestos más atrás. Pero ¿es la diferencia suficiente para rechazar JDO frente a JDBC? parece que no, porque los tiempos de respuesta finalmente, son de un orden de magnitud similar, inferiores al segundo, tiempo aceptable para el común de los usuarios al interactuar con una aplicación.

Nuevas pruebas, para llevar a cabo la consulta, modificación eliminación de un objeto, de forma interactiva, confirmarían con mayores diferencias, recalcarían los pronósticos realizados, por lo cual no merece la pena el esfuerzo de su realización. El peor caso sería el de la eliminación que requiere la recuperación de la instancia antes de su borrado definitivo, cosa que evidentemente favorece a JDBC frente a JDO.

Para contrastar la verosimilitud de los resultados obtenidos, se han conseguido los datos de rendimiento de una implementación JDO no comercial, JDO MAX [50] sobre MySql 4.0, que muestra tiempos de creación similares (48 ms), del mismo orden de magnitud.

JDO vs ODMG sobre bases de datos de objetos

Al comparar JDO con ODMG, teóricamente no debería haber diferencias de rendimiento, pues ocupan el mismo hueco entre objetos y bases de datos, sea esta relacional o de objetos. Algunos fabricantes de sistemas de bases de datos de objetos han lanzado interfaces JDO para acceder a sus sistemas de bases de datos, por ejemplo, Progress, Versant y Poet. Las principales diferencias de rendimiento procederán de las mejoras introducidas en la interfaces de programación de los sistemas empleados, que no tienen equivalencia en JDO, y, del uso de tipos de datos predefinidos optimizados, tipos

predefinidos ODMG y los propios del fabricante. En un acceso interactivo no hay las diferencias significativas entre JDO y ODMG.

Las figuras Figura 23 y Figura 24, muestran los resultados de la ejecución de la misma prueba de apertura de una cuenta, sobre un OODBMS ODMG Versant. En la segunda transacción, el producto Versant ofrece peores resultados que las soluciones JDO JDBC sobre RDBMS probada. Pero en todos los casos los resultados son del mismo orden de magnitud, un segundo, tiempo de espera razonable en una transacción interactiva.

En resumidas cuentas, los resultados están en un mismo orden de magnitud y desde el punto de vista de usuario los tiempos de respuesta son aceptables, por lo cual, parece razonable no descartar JDO para aplicaciones con transacciones interactivas, si JDO aporta otros beneficios, que compensan su falta de rendimiento en el terreno de las aplicaciones interactivas.

4.2.1.2. JDO frente a las herramientas Objeto relacionales (OR)

En el siguiente apartado, se presenta una prueba que maneja un modelo de objetos complejo que persiste sobre unos RDBMS. Este es un campo donde existen implantadas soluciones, con una gran tradición, las herramientas OR, verdadera competencia de JDO accediendo a RDBMS: Cocobase, Castor, Toplink, Jakart OJB o Hibernate, son algunos de los actuales y más extendidos de los productos OR.

En teoría, las herramientas OR y JDO JDBC, deberían ofrecer rendimientos muy similares, realizan un trabajo muy similar, convertir datos en filas en objetos, y viceversa, con eficacia y rendimiento, disponen de distintos mecanismos para mejorar el rendimiento cuando se acceden volúmenes importantes y complicados de datos: generación inteligente de SQL, caches, cursores,…, dicho de otra forma, en una visión general de la arquitectura en capas de las soluciones y de JDO, ambas están a un mismo nivel y ocupan la misma posición. La ventaja de JDO sobre los productos OR, no es el rendimiento, procede de otros aspectos, más concretamente, ser una solución general, no solo para RDBMS, ser más productiva, ser un estándar y ofrecer la mejor persistencia ortogonal, con transparencia de datos, persistencia por alcance, sin necesidad de modificar el fuente. En las conclusiones volveremos sobre el tema, para aclarar porqué elegir JDO frente a las OR citadas.

La cuestión es suficientemente interesante, como para dedicar un proyecto fin de carrera. En este trabajo, se asume el planteamiento de la igualdad de rendimiento entre las OR y JDO accediendo a los RDBMS.

4.2.1.3. Prueba de Transacción manejo de un gran volumen de datos complejos.

La prueba ideada para simular una transacción que maneja un gran volumen de objetos complicados, es crear y recuperar árboles 5-arios de profundidad 5. Esta prueba permite contrastar la idoneidad de JDO para manejar situaciones con modelos complejos y grandes volúmenes de datos, y al mismo tiempo sirve también para ver, la influencia del modelo de relaciones empleado para persistir los objetos, cuando se utiliza un RDBMS como almacén de objetos. También además, se ilustra como JDO cubre el RDMBS ofreciendo una realidad basada solo en objetos. Otro aspecto de la prueba es, que es más adecuada para comparar el rendimiento de JDO frente ODMG en OODBMS, porque ODMG tiene más aceptación en los OODBMS, que sobre los RDBMS, y los OODBMS son aplicados tradicionalmente a estas situaciones de datos complejos y gran volumen.

Para la prueba han sido elaborados unos programas, basados en un test de rendimiento distribuido con FastObjects. La creación de 5 árboles 5 arios de profundidad 5, con un total de 19,531 nodos por árbol, resultando un total de 97.655 objetos, que son

hechos persistir. Y otro, para la posterior recuperación transparente mediante el recorrido de cada uno de los árboles creados, en un millar de ocasiones (97.655 objetos x 1000 veces). Los programas son configurados y adaptados, para ser ejecutados sobre varios productos JDO para RDBMS MySQL, Oracle y SQLServer, frente a otros ODMG OODBMS: Versant, Fast Objects y ObjectStore.

La prueba no incluye un programa con JDBC solo, porque para lograr un rendimiento adecuado en la situación modelada, se deberían incluir funcionalidades como caches de objetos, de cursores,…, que complican bastante la programación para alcanzar el rendimiento de una solución JDO, cuyas implementaciones sobre JDBC, se suponen están optimizadas para acometer estas situaciones complejas y con grandes volúmenes de información. Así, se asume que para este tipo de trabajos, mejor que programar en JDBC directamente, es preferible utilizar JDO JDBC, que debería.

La prueba toma como referencia el tiempo de ejecución necesario para realizar las tareas de creación y recorrido de los árboles de prueba empleando solo objetos transitorios, lo cual sirve para ver la sobrecarga introducida por la persistencia y lo ajustado de las implementaciones JDO.

El código java del programa de creación de los árboles es el siguiente

```
class Write {
  static private Database db;
  public final static String strfoid = "tempoids";

  static ArrayOfObject createTree(int depth, int nodesPerLeaf) {
    ArrayOfObject arr = new ArrayOfObject();
    for (int i = 0; i < nodesPerLeaf; ++i) {
      if (depth == 0) {
        arr.add(new MyClass());
      }
      else {
        arr.add(createTree(depth - 1, nodesPerLeaf));
      }
    }
    return arr;
  }

  static Vector write(int depth, int nodesPerLeaf, int nTrees) throws
      JDOException {
    ArrayOfObject ao = null;
    Object oid = null;
    Vector trees = new Vector();

    try {
      for (int i = 0; i < nTrees; ++i) {
        Transaction txn = db.currentTransaction();
        txn.begin();
        ao = createTree(depth, nodesPerLeaf);
        db.makePersistent(ao);
        trees.add(db.getPM().getObjectId(ao));
        txn.commit();
      }
    }
    catch (Exception exc) {
      exc.printStackTrace();
    }
    return trees;
  }

  public static void main(String[] args) throws JDOException {
    if (args.length < 3) {
      System.out.println(
          "please specify depth, nodes/leaf and number of trees.");
      System.exit(1);
    }

    long start, stop;
```

Métodos de creación recursivo, sin distinción entre persistente o transitorio.

Código para hacer persistentes los árboles. La versión no persistente comenta la demarcación de la transacción y makePersistent

```
    Vector trees;

    int depth = Integer.parseInt(args[0]);
    int nodesPerLeaf = Integer.parseInt(args[27]);
    int nTrees = Integer.parseInt(args[9]);

    System.out.println("Creating " + nTrees + " trees with depth " + depth +
                       " and " +
                       nodesPerLeaf + " nodes per leaf");

    try {
      db = new KodoJDO();
      db.open();
    }
    catch (JDOException exc) {
      System.out.print("Error de conexión");
      exc.printStackTrace();
      throw exc;
    }

    try {
      start = System.currentTimeMillis();
      trees = write(depth, nodesPerLeaf, nTrees);
      stop = System.currentTimeMillis();

    }
    finally {
      db.close();
    }
      System.out.println("Time to write: " + (stop - start) / 1000 + " sec");

    try {
      ObjectOutputStream oids = new ObjectOutputStream(
          new FileOutputStream(strfoid));
      oids.writeObject(trees);
      oids.close();
    } catch (Exception e) {
      e.printStackTrace();
    }

  }
```

Código entre las versiones difiere en la clase utilizada como base de datos. La sintaxis utilizada es como la ODMG, para facilitar la migración.

Listado 35 Creación de los árboles del test.

El código anterior mide el tiempo total para persistir todos los árboles creados en su ejecución. Cada árbol es creado en memoria en una rutina recursiva, y hecho persistir expresamente con el método `makePersistent`. Al medir el tiempo total del conjunto de las transacciones, se consigue incluir en la medida la influencia de los resortes para el rendimiento del mecanismo de persistencia y se reduce la distorsión por los tiempos de inicialización.

Para la persistencia en RDMBS, son considerados dos modelos de datos, uno indicado en los gráficos por **árbol en filas**, donde los objetos quedan desgranados en filas hasta sus datos primitivos y cada elemento de una colección corresponde a una fila, y otro, concebido para la rapidez, donde los árboles son serializados y almacenados en único campo tipo blob, indicado en las gráficas como **árbol serializados**. Para la comparativa con ODMG, se utilizan dos métodos para alcanzar la persistencia, la persistencia por alcance dada una referencia inicial, y el método `bind`, asignando un nombre a la raíz de cada uno de los árboles.

El código de la prueba de recorrido de los árboles, que recupera reconstruye los árboles, y recorre sus nodos siguiendo un patrón de recorrido en orden de forma recursiva, está en el listado siguiente:

```
  static int countLeafs(Object obj) {
    int count = 1;
    if (obj instanceof ArrayOfObject) {
      ArrayOfObject arr = (ArrayOfObject) obj;
```

```
      Iterator i = arr.iterator();
      while (i.hasNext()) {
        count += countLeafs(i.next());
      }
    }
    return count;
}
  static void read(Vector trees, int nTrees, int nTraversals) throws
JDOException {
    try {

      Transaction txn = db.currentTransaction();

      ArrayOfObject tree ;
      txn.begin();

      for (int iTraversals = 0; iTraversals < nTraversals; ++iTraversals) {
        Iterator it = trees.iterator();
        for (int iTrees = 0; iTrees < nTrees && it.hasNext(); ++iTrees) {
          tree = (ArrayOfObject ) db.getPM().getObjectById(it.next(),true);
          System.out.println("Tree has " +
                             countLeafs(tree));
        }
      }
      txn.commit();
    }
    catch (Exception exc) {
      exc.printStackTrace();
    }
```

Listado 36 Recuperación transparente de los árboles

En listado anterior Listado 36 solo una instrucción, marcada en amarillo, indica que el objeto `tree` debe ser recuperado de la base de datos. Veamos ahora que se puede esperar de los resultados, sabiendo que la especificación JDO asume como objetivo la conveniencia de la optimización en el acceso a datos y el manejo de grandes conjuntos resultados (JDO Pág. 100 14.2 Objetivos).

Antes de ejecutar las pruebas, comprobamos como JDO efectivamente oculta el RDMBS, ofreciendo una abstracción basada exclusivamente en objetos, manejados con independencia de si son persistentes o no.

Con JDO el manejo de grandes volúmenes de datos, que resultan en actualizaciones masivas, pueden ser fuente de problemas de contención, al recorrer y actualizar colecciones de objetos simultáneamente accedidas por distintos procesos, con el acceso transparente y el bloqueo automático, que JDO implanta. Un control transaccional optimista, puede mitigar la contención en estas situaciones, pero esta una funcionalidad es opcional en el estándar, que en la práctica todos los productos revisados ofrecen.

Continuando con el problema de manejar y actualizar gran cantidad de datos, considerando como destino de la persistencia los RDBMS, que ofrecen código empotrado en el servidor de base de datos. Es cierto que JDO no supera la rapidez de los procedimientos almacenados en bases de datos en el tratamiento masivo de datos. Tampoco JDBC, ni SQLJ superan a estos procesos embutidos dentro del SGBDR. De los productos JDO probados, algunos permiten ejecutar código almacenado en las bases de datos Oracle, SQLserver y SAPdb. Esto permite disponer en JDO, de la efectividad del código en base de datos, mejorando el rendimiento.

Hay una situación de manejo de cantidades importantes de información, en la que JDO 1.0, es ostensiblemente más lento que JDBC y SQLJ, es la eliminación de los estados persistentes. JDO obliga a recuperar las instancias para proceder a su eliminación, siendo

posible suprimir las imágenes almacenadas de los objetos, sin la sobrecarga de la operación de recuperación, con una sencilla sentencia DELETE ejecutada en el RDBMS. Es fácil evitar la recuperación de instancias para su eliminación persistente, aprovechando los disparadores y código empotrado de los RDBMS. Esa limitación en la eliminación de objetos, se corrige en la siguiente versión JDO, con métodos al efecto, sin necesidad de recuperar los objetos correspondientes.

Aprovechando las posibilidades de los RDBMS modernos, JDO podría alcanzar cotas similares a JDBC y SQLJ, que también necesitan aprovechar la funcionalidad empotrada en RDBMS para ser más eficaces manipulando gran cantidad de datos. La cuestión importante ¿Es la diferencia tal como para despreciar JDO frente a JDBC/SQLJ?. Mi opinión es que no, si sabe aprovechar las capacidades de los RDBMS, combinadas con JDO. Solo un programador novel, sin experiencia, trata de procesar millares de datos en única transacción de forma remota, trasegando ingentes volúmenes de datos por las redes, esperando que la potencia de computo y el ancho de banda, resuelva la contingencia.

Frente a los sistemas OODBMS ODMG, la situación de JDO es de igual a igual. El rendimiento en los sistemas de datos de objetos en operaciones de actualización masiva, puede ser incluso superior a los relacionales pues una sola modificación de un objeto queda consolidada para todas sus referencias, evitando el trasiego de grandes volúmenes de datos. De hecho el elevado rendimiento de los sistemas de bases de datos de objetos es uno de los argumentos para su elección frente a los sistemas de bases de datos relacionales. El acelerador de partículas nucleares de Stanford y del CERN, son ejemplo de referencia. Existen ya implementaciones comerciales de OODBMS que ofrecen interfaz JDO, como FastObjects, por lo cual JDO puede aprovechar el mayor de rendimiento de los OODBMS en este terreno. Parece verosímil que los rendimientos en tareas de actualizaciones masivas sean similares entre JDO y ODMG.

En todos los casos se utilice un RDMBS u OODBMS, *el factor decisivo contribuyente al rendimiento es plasmar un buen diseño de las clases, de los modelos de persistencia y del esquema físico en bases de datos.*

Una primera conclusión, previa inclusión a los resultados, es que en todos los casos, RDMBS, OODBMS o ficheros, JDO permite emplear el mecanismo más conveniente a la problemática concreta sin ser necesario programar para cada mecanismo de persistencia expresamente, esta es la principal ventaja que transluce a la prueba. La adaptación requerida en los programas JDO para usar una u otra plataforma ha sido la de emplear unos archivos de configuración y entornos de librerías de ejecución diferentes.

Comprobemos si se confirman los argumentos presentados con las programas de prueba. Los resultados presentados en las siguientes figuras, han sido obtenidos de la ejecución de los programas de carga repetidas veces para reducir la distorsión debida al entorno de pruebas. No han sido incluidos datos obtenidos con otros productos, bien por alguna limitación de los mismos a la hora de prueba, o bien porque los resultados no tenían cabida, por ser excesivamente largos.

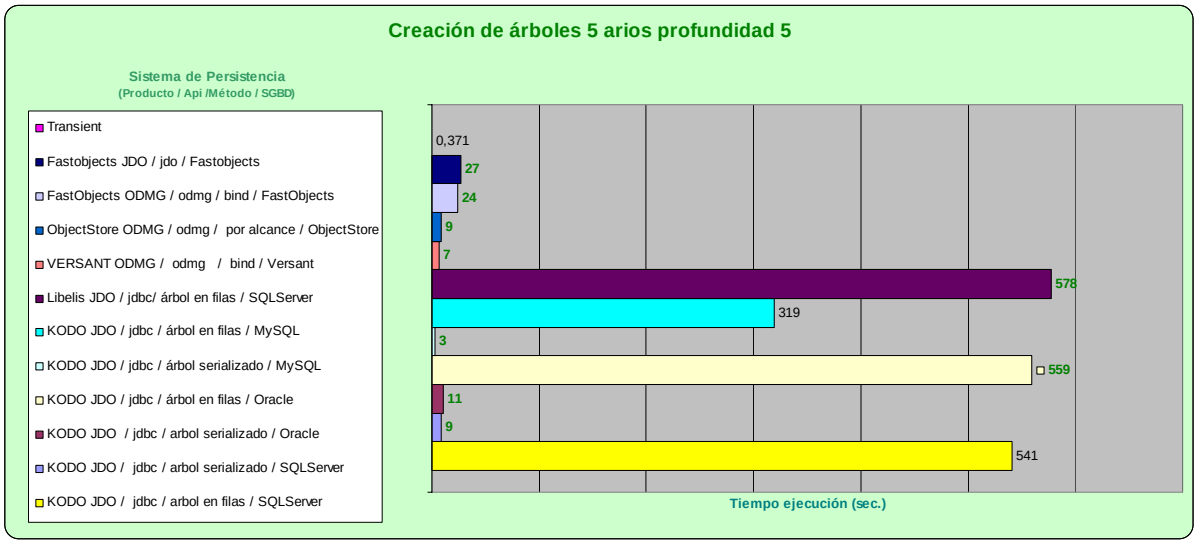


Figura 26 Resultados prueba de rendimiento para objetos complejos

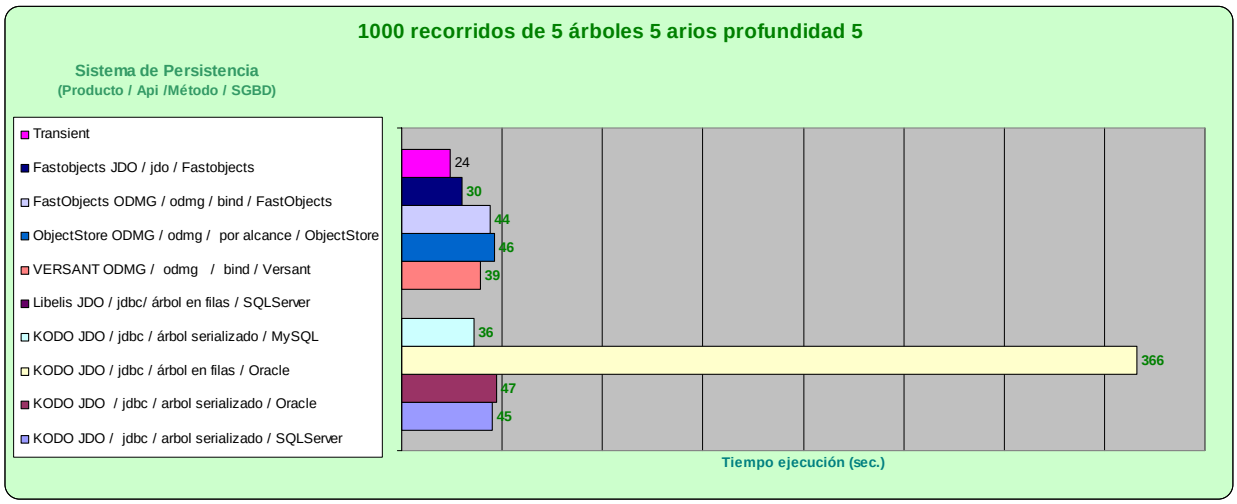


Figura 27 Tiempos de ejecución del recorrido transparente de árboles 5 arios profundidad 5, 1000 veces

Las dos figuras precedentes muestran unos resultados de los que a primera vista destaca una primera gran interrogante ¿Por qué hay tanta diferencia entre las series que utilizan un modelo de datos normalizado y los que usan columnas blob para almacenar los objetos serializados?

La diferencia surge por el esfuerzo de computo necesario para que los 97.655 objetos, se conviertan en 585.930 filas sobre un modelo normalizado, donde los objetos y relaciones son desagregados. Mientras que utilizando columnas tipo blob el número total de objetos, queda grabado en 5 filas, una por cada árbol. Igualmente ocurre con la operación de lectura, al recuperar los árboles, la reconstrucción a partir de más de medio millar de filas, supone un tarea enorme, que requiere aquí, en el mejor caso, algo más de 6 minutos, 366 s. con Oracle, los demás RDBMS, tras más de 15 m., no habían completado la prueba de lectura. También, destaca el hecho de que bajo estas circunstancias, los OODBMS aventajan a los RDMBS.

Otro aspecto destacado de los resultados que enfrenta a OODBMS y RDMBS, es que los tiempos de las pruebas para el modelo de datos que contienen los árboles

serializados en un campo blob, están en los mismos ordenes de magnitud que los programas de carga equivalentes para ODBMS ODMG, desde unos pocos segundos a menos de un minuto, tanto para la creación, como para la recuperación.

Queda patente la dificultad e implicaciones de un buen modelo de datos adecuado para el rendimiento. Los esquemas normalizados, son fáciles de entender y de usar, pero para ciertos tipos de tareas no apropiados. Además de rendimientos dispares, los esquemas ofrecen distintas posibilidades de actualización y consulta. En este caso, el modelo desagregado permite el bloqueo y consulta individual de una instancia cualesquiera para su actualización, mientras que el modelo con blob, deja todo el árbol bloqueado en el RDBMS cuando es actualizado un nodo, salvo que sea emplee un bloqueo optimista. La mayor desventaja del modelo con serializado, es la imposibilidad para consultar ad hoc con SQL directamente el nodo cualesquiera de un árbol que cumple ciertas condiciones.

La tesis sostenida sobre si JDO es adecuado para situaciones de manejo gran número de objetos complejos, queda respaldada por la pequeña batería de pruebas efectuada, en especial, en consideración de los tiempos de respuesta mejores obtenidos de la solución JDO RDBMS frente a ODMG OODMS, en teoría más eficaz; y a tenor del esfuerzo que supondría programar una solución equivalente en JDBC.

Respecto de ODMG y JDO, se comprueba que las diferencias de tiempos, aunque en proporción son abultadas (3 s. sobre 7s. y 30 s. sobre 45 s.), los ordenes de magnitud de los resultados, tanto para JDO sobre RDMBS como ODMG sobre OODBMS, son los mismos. Es más, para FastObjects OODBMS, con frontal JDO y ODMG, es más rápido su API JDO que el API ODMG (30s por 44 s) debido al método para hacer persistente el objeto, dándole un nombre. También se confirma en parte, el pronóstico hecho de cómo JDO y ODMG deberían tener rendimientos muy similares.

En otro orden de cosas, los resultados anteriores desvelan también un hecho importante a mencionar. Que el modelo relacional no siempre es la opción más adecuada para persistir, este es un ejemplo, donde los OODBMS, ofrecen mejores resultados que las demás alternativas.

Por otro lado, respecto a la optimización de algunas de las implementaciones utilizadas, ejemplo de Kodo, se observa que la sobrecarga introducida, es incluso inferior a productos ideados para OODBMS, como Versant. Una buena implementación JDO, puede llegar a ser tan buena como un producto ODMG OODBMS en la grabación y recuperación según qué carga se maneje.

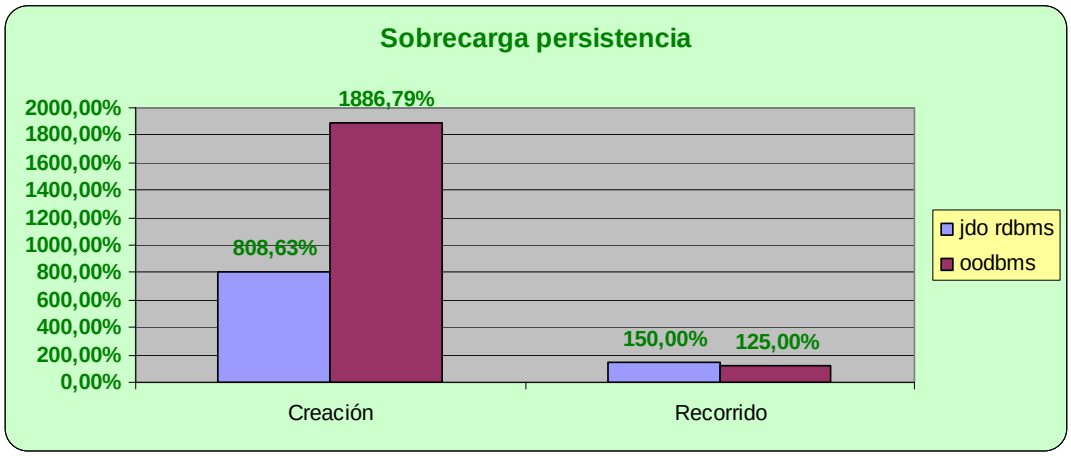


Figura 28 Sobrecarga de la persistencia en las pruebas realizadas

Las pruebas efectuadas confirman que JDO es una tecnología adecuada para su uso con modelos de objetos complejos y con elevado número de instancias. Las prestaciones de JDO pueden ser tan buenas como las de ODMG. Cuando se utilice un RDBMS, debemos comprobar que el modelo de datos cubre las expectativas de rendimiento que se espera. No siempre un sistema relacional es el mecanismo de persistencia más adecuado, no debemos desdeñar otros sistemas de persistencia, principalmente los ficheros y las bases de datos de objetos, cuando el rendimiento, la productividad y complejidad de la solución RDBMS, no son adecuados a los requisitos.

4.2.1.4. Conclusión del Rendimiento

- El rendimiento de JDO considerando el tiempo de procesamiento de una transacción en el tratamiento de objetos simples, es aceptable frente a las otras tecnologías JBDC y ODMG.
- Considerando solo el rendimiento, JDO no remplaza a JDBC y SQLJ, en aplicaciones con datos simples manejados con SQL.
- Con modelos de objetos sofisticados, que necesiten de navegar entre un número importante de objetos, JDO es más simple y consigue un rendimiento correcto.
- JDO con JDBC, y un esquemas de tablas apropiado, logran conseguir rendimientos correctos, en situaciones con gran volumen de datos complejos.
- JDO ofrece un rendimiento similar a ODMG en las pruebas efectuadas.
- Al margen de JDO, la lección aprendida es que un buen diseño de objetos y datos son un factor clave para alcanzar los niveles de rendimiento correctos.

Tabla 4 Cuadro resumen del rendimiento comparado

Rendimiento					
Tecnología	JDBC/SQLJ	JDO			ODMG
		JDBC		OODBMS	OODBMS
Esquema persistencia	normalizado	normalizado	desnormalizado y serialización	objetos	objetos
Modelo de objetos simple	Superior	Aceptable	Alto	Aceptable	Aceptable
modelo de objetos complejo	Inaceptable	Inaceptable	Alto	Alto	Alto

El rendimiento de JDO no es un factor claramente diferenciador, a favor o en contra de JDO y frente al resto de las otras tecnologías. Su aportación esencial tiene que venir de otra dimensión, la productividad.

4.2.2. Productividad

La productividad en la programación Java sobre una tecnología de persistencia, viene determinada principalmente por esfuerzo necesario para crear los programas con esa tecnología y de cuánto cuestan los productos necesarios para usar esa tecnología. Dejando a un lado el aspecto organizativo, el esfuerzo de producir programas Java, con una tecnología u otra, depende de cuanto código fuente sea necesario programar para realizar la funcionalidad requerida. La inversión en una tecnología de persistencia debe estar

justificada por el retorno de la inversión producido por la misma, debe rápidamente amortizar y producir beneficios. Si una tecnología permite desarrollar con menor esfuerzo una misma funcionalidad, y además a menor coste, entonces es más productiva, y en consecuencia más rentable.

La funcionalidad aportada por una tecnología y su integración con el modelo de objetos java, repercuten en la facilidad para expresar con menos líneas el código, por lo que reduce el esfuerzo de desarrollo. A lo largo del trabajo han sido ilustrados distintos ejemplos de la funcionalidad de las tecnologías en comparación. Respecto a su integración con el modelo de objetos java sabemos: Que JDO impone pocas limitaciones al modelo de objetos y al diseño, esencialmente no es posible persistir aquellos objetos ligados al sistema en tiempo de ejecución (hebras, *threads*,…) y remotos a otras JVM. La integración de JDBC se reduce a la equivalencia de los tipos primitivos o básicos Java con los tipos de los RDBMS en la interacción con SQL. SQLJ requiere menos líneas de código que JDBC, para las operaciones con SQL estático, pero por lo demás es idéntico a JDBC. ODMG se adapta al modelo de objetos Java, salvo en que exige de ciertos tipos, para representar algunas estructuras de datos. La funcionalidad transparencia de datos y la persistencia por alcance de JDO y ODMG, favorecen la integración con el modelo de objetos java, repercutiendo en un menor número de líneas dedicadas a la persistencia de los objetos. En JDBC, un buen número de líneas consisten en la invocación de métodos, para hacer corresponder argumentos formales de las sentencias SQL con los valores actuales o variables destino. Esto cambia en JDBC 3.0 y su soporte para tipos estructurados de usuario, que utiliza cierta clase de instancias para la correspondencia entre instancias y tipos de datos de usuario de SQL99 (Capítulos 16 y 17 de la especificación JDBC 3.0).

Dejemos de lado la integración con Java y la facilidad de desarrollo, el número de líneas de código para materializar la funcionalidad necesaria, se configura como un indicador del esfuerzo de desarrollo que permite la comparación entre cada una de la tecnologías consideradas. Menos líneas de código, significa, menor tiempo de desarrollo, mayor facilidad de mantenimiento, menos costes de desarrollo. Se consigue con menos líneas de código una mayor productividad. La ingeniería del software define otros muchos indicadores, posiblemente más adecuados, pero el número de líneas es un factor de análisis fácil de entender y usar, que no requiere mayor complejidad.

El número de líneas es un indicador que favorece la comparación, pero ¿Es posible expresar con JDO siempre la funcionalidad requerida con claridad y sencillez? Los lenguajes de consultas son otro elemento necesario que contribuye a la productividad. Es necesario disponer de la posibilidad de especificar consultas que obtienen los objetos que satisfacen cierto predicado, de integrar un lenguaje para hacer consultas, sin codificar como recuperar los objetos [2].

En los siguientes apartados, se analizará la reducción en el número de líneas de JDO frente a JDBC y ODMG, y los lenguajes de consulta JDOQL frente a SQL y OQL.

4.2.2.1. Número de líneas de JDO RDMBS frente JDBC

Torsten Stanienda y Douglas K. Barry, en su artículo "Solving the Java Storage Problem" [35], comparaban la programación con persistencia transparente, con la programación a nivel de interfaz en JDBC, comparando ODMG con JDBC. Los resultados en su artículo muestran, una reducción total de 74.21% del número de líneas de código, de escribir la solución en ODMG, a materializar la misma funcionalidad con JDBC.

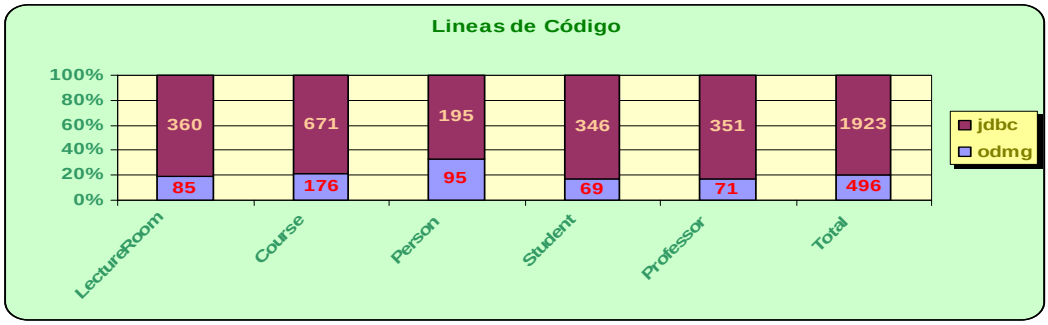


Figura 29 Comparativa del número de líneas de código para una misma aplicación escrita con JDBC y ODMG

La figura anterior enfrenta para cada clase el número de líneas de la solución JDBC y su equivalente ODMG, junto al total de líneas con una y otra solución. La siguiente presenta los porcentajes de reducción en número de líneas para cada clase.

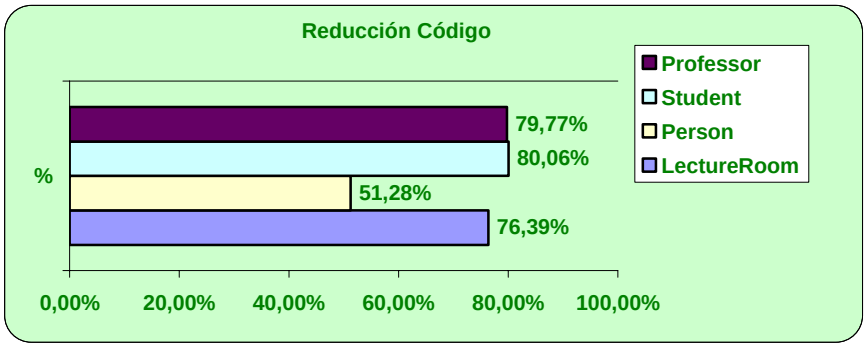


Figura 30 Porcentaje de reducción del código Java con ODMG respecto de JDBC.

La reducción vista es muy significativa, pero leyendo el artículo parece que en ese ejemplo, las clases del modelo encerraban el código de persistencia necesario. Otros planteamientos son comúnmente utilizados, en donde la persistencia no reside en las clases del modelo, aún trabajando con JDBC, por ejemplo, en [18], en tales casos la reducción es muy inferior. También el tipo de aplicación condiciona la cantidad de líneas del total, dedicadas a manejar la persistencia de lo objetos, es por eso, que vemos estimaciones sobre el esfuerzo dedicado a persistencia, que oscilan desde el 40% (Oracle) al 70% (Cocobase). En razón de todo lo cual, parece de más interés, medir el efecto sobre el código dedicado a la persistencia, el esfuerzo necesario para llevar a cabo la misma funcionalidad de persistencia de los objetos, frente a JDBC solo, sin la ayuda de herramientas OR, que también reducen significativamente el esfuerzo de desarrollo para persistir como JDO.

Para medir el ahorro trabajando con JDO, en el número de líneas dedicadas a persistir, se escoge modificar una aplicación ya realizada, más real que el ejemplo del banco, pero simple, sencilla: la aplicación de gestión de reservas de restaurantes, ClubMed, del libro de Robert Orfalli [23]. Al mismo, tiempo permite hacernos una idea, de la capacidad e impacto de JDO en la adaptación de aplicaciones heredadas, uno de los objetivos marcados en este trabajo. La versión modificada sustituye los métodos con instrucciones en JDBC, por otros equivalentes con JDO. Otros métodos de la aplicación, incluidos nuevos métodos que tienen que ver con la plataforma o productos escogidos, son ignorados. La fuente se encuentra en los anexos.

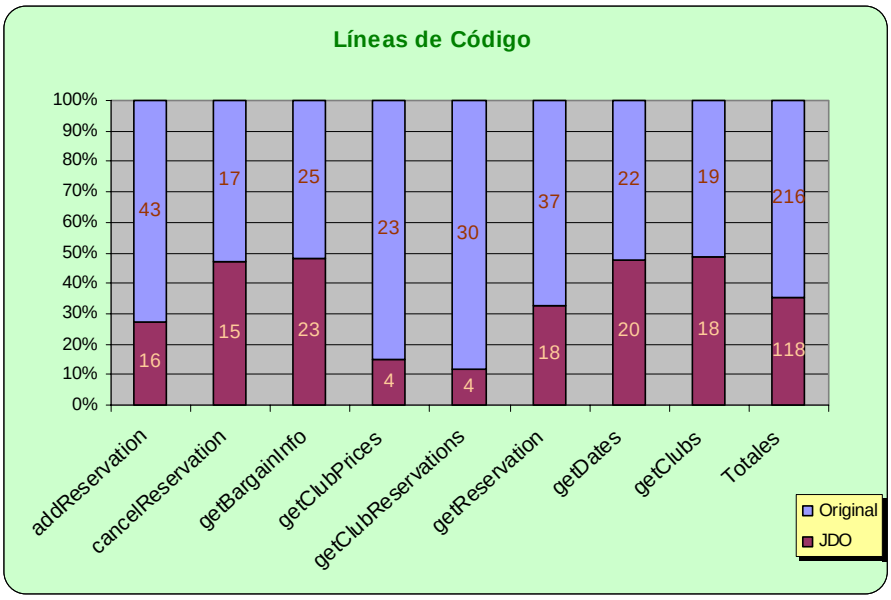


Figura 31 Líneas de código en la aplicación de ClubMed con JDO vs original JDBC

La reducción obtenida sobre este ejemplo es inferior a la obtenida en artículo de [35], de la Figura 29. La reducción total es de un 45,37%, con una media de reducción 39,69%.

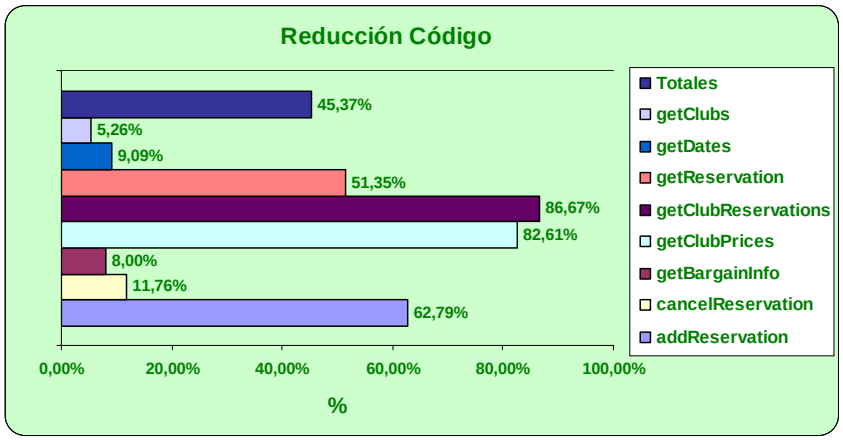


Figura 32 Reducción del número de lineas en ClubMed modificado para JDO

Evidentemente, hay fórmulas que consiguen un menor número de líneas con un código más enrevesado aprovechando la potencia sintáctica de Java. También, cambiando el diseño del modelo de objetos persistentes es posible mejorar el número de líneas. El lector puede comprobar sobre en los siguientes fragmentos de código un ejemplo de los cambios en el código.

Listado 37 Método addReservation con JDO

```
public long addReservation(Clubs aClub,
                           Reservations aResv) throws java.lang.Exception {
try {
  Transaction t = this.pm.currentTransaction();

  t.begin();

  aResv.setClub(aClub);
  this.pm.makePersistent(aResv);
  pm.refresh(aResv); //recuperar datos asignados por BD
```

```
  t.commit();
  return aResv.getResvno();
} catch (Exception e) {
  System.err.println("System Exception in getClubs");
  System.err.println(e);
  throw e;
  }
}
```

Listado 38 Método addReservation original con JDBC

```
public int addReservation(String club, Clubs.resvStruct resvData) throws
    Exception {
  try {
    pstmt = con.prepareStatement(
        "SELECT MAX(ResvNo) FROM RESERVATIONS");
    pstmt.execute();
    ResultSet rs = pstmt.getResultSet();
    rs.next();
    int resvNo = rs.getInt(1);
    resvNo = resvNo + 1;

    pstmt = con.prepareStatement(
        "INSERT INTO Reservations " +
        "(ResvNo, Club, WeekDate, LastName, " +
        "FirstName, PartyAdults, " +
        "PartyChildren, TotalFare, PaymentMethod, " +
        "PhoneNo, Address, City, State, Zip) " +
        "VALUES(?, ?, ?, ?, ?, ?, ?, ?, " +
        "?, ?, ?, ?, ?, ?)");
    pstmt.setInt(1, resvNo);
    pstmt.setString(2, club);
    pstmt.setString(3, resvData.weekDate);
    pstmt.setString(4, resvData.lname);
    pstmt.setString(5, resvData.fname);
    pstmt.setShort(6, resvData.partyAdults);
    pstmt.setShort(7, resvData.partyChildren);
    pstmt.setString(8, resvData.totalFare);
    pstmt.setString(9, resvData.payment);
    pstmt.setString(10, resvData.phone);
    pstmt.setString(11, resvData.address);
    pstmt.setString(12, resvData.city);
    pstmt.setString(13, resvData.state);
    pstmt.setString(14, resvData.zip);
    pstmt.executeUpdate();
    pstmt.close();
    con.commit();
    return resvNo;
  }
  catch (Exception e) {
    System.err.println("System Exception in addReservation");
    System.err.println(e);
    throw e;
  }
}
```

La mayor simplicidad y fácil lectura de la versión JDO frente a la plasmada con JDBC, no admite discusión. El código de la aplicación con JDBC podría ser mejorado, expresado con algunas menos líneas, implantando un objeto de mapeo del tipo SQLData. No obstante, creo que buena parte de los desarrollos Java, son del estilo del listado anterior.

En definitiva, JDO puede lograr la misma funcionalidad de persistencia con menos líneas que JDBC, JDO consigue ser más productivo que JDBC.

4.2.2.2. Número de líneas JDO frente a ODMG para Java

El número de líneas de una solución JDO y otra ODMG para Java es muy similar, ya que ambas tecnologías implantan la transparencia y la persistencia por alcance. Basta con vistazo, al código fuente de las pruebas de rendimiento, Listado 35, y al Listado 40, para confirmar que no hay diferencia notoria en el número de líneas. La diferencia esencial entre ambas en el código, es la exigencia de ODMG de utilizar ciertos tipos. En el test de rendimiento la clase ArrayOfObject, precisamente para facilitar la migración de la aplicación a ODMG, porque el estándar incluye un tipo análogo del mismo nombre ArrayOfObject, que en los programas ODMG, no hace falta declarar. En resumen, el número de líneas para programar en java la persistencia con JDO, no es claramente menor que en ODMG para Java; en este sentido, JDO no es más productivo que ODMG.

Listado 39 Clase ArrayObject

```
public class ArrayOfObject
    implements Collection, java.io.Serializable {
  // Esta clase es una First Class Object(FCO)
  // Tendra identidad propia.
  // Es complejo,  los arboles sobre
  // los que se mide el rendimiento

  private Vector pVF = new Vector();

  public int size() {
    return this.pVF.size();
}
```

Listado 40 Prueba de rendimiento de creación de árboles ODMG

```
  static Vector write(int depth, int nodesPerLeaf, int nTrees) throws
      com.versant.odmg.ODMGException {
    ArrayOfObject ao = null;
    Vector trees = new Vector();
    try {
      String s = null;
      for (int i = 0; i < nTrees; ++i) {
        Transaction txn = new Transaction();
        s = "Arbol" + i;
        txn.begin();
        ao = createTree(depth, nodesPerLeaf);
        db.bind(ao, s);
        trees.add(s);
        txn.commit();
      }
    }
    catch (Exception exc) {
      exc.printStackTrace();
    }
    return trees;
…
    try {
      db = Database.open("perfbase", Database.openExclusive);
    }
    catch (ODMGException exc) {
      System.out.print("Error de conexión");
      exc.printStackTrace();
      throw exc;
    }
    try {
      start = System.currentTimeMillis();
      trees = write(depth, nodesPerLeaf, nTrees);
      stop = System.currentTimeMillis();

       …
```

4.2.2.3. Lenguaje de consultas JDOQL frente a SQL

Los programas JDBC y SQLJ se caracterizan por la utilización del SQL, como fórmula para acceder a los datos. En ODMG y JDO, el acceso a transparente evita la necesidad de utilizar un lenguaje de consultas por todas partes, solo cuando es necesario en la localización de los objetos iniciales, desde los cuales se alcanzan otros con la navegación por la referencias.

JDOQL, consiste de una interfaz de programación, Query, y una semántica con gramática Java, que expresa las condiciones de selección, aplicadas a la colección de objetos candidatos implicados. Es un hecho reconocido la limitación expresiva y completitud computacional JDOQL frente a SQL. Pero como contrapartida, JDOQL es independiente de que el sistema sea un sistema relacional o no, esto permite desacoplar las consultas de los mecanismos de persistencia.

La menor potencia expresiva y completitud computacional JDOQL frente a SQL, es motivo de debate sobre la conveniencia de extender JDOQL, de hecho, la especificación de JDO 2.0 extenderá este con nueva funcionalidad: borrado directo, envío de SQL al SGBDR, agrupamientos, resultados calculados y otros. En el escenario de los fabricantes de bases de datos relacionales y objeto-relacionales se viene incorporando SQL99, con nuevas cláusulas relativas al manejo de tipos estructurados de datos de usuarios (objetos) aumentando su complejidad, dificultad de aprendizaje y uso 0.

El resultado de una consulta JDOQL siempre es un conjunto de objetos que cumplen las condiciones del filtro de consulta, nunca el resultado de un cálculo efectuado sobre algunos de estos objetos, cosa que si es posible en SQL sobre las filas procesadas. Cuando trabajamos con JDBC, el resultado de una SQL, es un ResulSet, un conjunto de filas cuyos elementos necesitan ser tratados con distintos métodos JDBC.

Cuando se accede con JDO a un sistema relacional, parte de la potencia expresiva, que reclaman algunos, puede ser obtenida conectando objetos a vistas o accediendo a las conexiones directas subyacentes para procesar las sentencias SQL. Los productos JDO revisados permiten el acceso a las conexiones subyacentes, para la realización de sentencias SQL que no podrían ser expresadas en JDOQL, con la economía sintáctica propia de SQL.

La simplicidad de JDOQL es una ventaja en la formación de nuevos programadores, pues no requieren conocer además de Java otro lenguaje, el SQL o el OQL. A fin de cuentas, SQL es claramente superior a JDOQL, también más complejo y necesita de una intervención mayor del programador. Si bien SQL es más potente en todos los aspectos que JDOQL, la integración con JDBC de SQL, requiere de más líneas de código Java que JDOQL.

4.2.2.4. Lenguaje de consultas JDOQL frente a OQL

OQL y JDOQL son ambos son orientados a objetos, parecidos en que los filtros de consulta acceden a los atributos y objetos miembros de los objetos consultados. ODMG 3.0 OQL admite además la composición, la proyección o un tipo primitivo como el resultado de una consulta, es un lenguaje funcional y ortogonal, que permite subconsultas, agrupamiento e invocación de métodos de instancia. OQL es más rico y complejo que JDOQL, también concordando con el planteamiento hecho, es más productivo que JDOQL. La ventaja esencial de JDO frente ODMG es JDOQL tiene una sintaxis Java, que evita tener que aprender la sintaxis de otro lenguaje.

En mi modesta opinión, si bien es verdad que JDOQL tiene una menor potencia que SQL y que OQL, el número de casos prácticos reales que necesitan de consultas de

predicados complicados es de escasa incidencia. La gran mayoría de las consultas utilizadas en los programas de negocio son simples, responden a predicados arquetipo de qué objetos verifican que sus atributos cumplen cierta igualdad, y qué objetos pertenecen o están, en tal o cual colección. Esta es una de las razones fundamentales, de porque JDOQL cubre esencialmente ese tipo de consultas. Creo, si bien OQL es mejor que JDOQL.

Por todo lo expuesto, podemos convenir que JDO es una tecnología con la que es más fácil programar la persistencia Java que con JDBC solo. Con JDO se necesitan menos líneas de código, de un menor esfuerzo de programación, para una gran mayoría de casos, a excepción de cuando hay que elaborar consultas SQL complicadas. El uso de JDO no excluye de usar JDBC, cuando sea preciso y conveniente. Frente a ODMG para Java, al igual que sucede con el rendimiento, la elaboración de programas JDO esta a la par, si no se considera el hecho de cada fabricante utiliza una adaptación conforme al estándar ODMG. Aunque poder expresar consultas, es un elemento importante para la productiva, la mayor potencia de SQL y OQL frente JDOQL, puede ser compensada por la simplicidad, y facilidad de uso de este último, en un buen número de casos. En definitiva JDO permite mejorar la productividad de la programación Java con persistencia.

Falta comprobar si se justifica la inversión en JDO, para lo cual se estimará el retorno de la inversión en JDO.

4.2.2.5. Retorno de la inversión en JDO

Para elegir una tecnología decidiendo invertir en ella, debe ser rentable, los beneficios reportados deben superar a la inversión necesaria. El retorno de la inversión (ROI) es una medida común para analizar la rentabilidad, en la apuesta por una tecnología y unos productos. Se calcula como la razón entre los beneficios reportados por la inversión necesaria, para el periodo de tiempo considerado. Cuanto mayor es el ROI, más favorable es la decisión de inversión. En este apartado se pretende estimar el retorno de invertir en JDO, asumiendo que la reducción en el coste de desarrollo, es el beneficio que debe financiar la inversión en JDO.

La adopción de entornos integrados de desarrollo plagados de asistentes para todas y cada una de las tareas de programación, conduce a una uniformidad del esfuerzo de desarrollo, aun con plataformas y productos diferentes. Herramientas para automatizar el mapping de clases a tablas como Cocobase, Toplink o Hibernate, son la verdadera competencia de JDO. Pero sin embargo, el esfuerzo de mantenimiento y depuración, tareas que requieren de la revisión y reescritura manual del código, son las actividades de producción de programas más afectadas, donde la mejora si puede ser estimada considerable. Cuyo coste según algunos autores es de hasta el 70% del coste del software [19]. En realidad, la reducción de código tiene mayores implicaciones que afectan al negocio, aumenta la calidad, posibilita el acortamiento de los ciclos de desarrollo y de puesta en el mercado de los programas. Desde este punto de vista del trabajar en el código fuente en el mantenimiento, no parece tan descabellado estimar que el 35% menos de líneas, se traduce en un 35 % menos del tiempo de desarrollo dedicado al mantenimiento de las operaciones de persistencia.

Como JDO para usar los RDBMS, normalmente necesita de JDBC, la inversión en JDO no es evita invertir en JDBC. Tampoco hay que olvidar que hay situaciones donde JDBC es más adecuado que JDO. Lo mismo que también la serialización puede ser la opción más adecuado en su caso, por ejemplo, en la prueba de rendimiento de ODMG ha sido utilizada la serialización para intercambiar los identificadores de los objetos entre los programas de creación y lectura. En todos los casos, si la incorporación de JDO, aumenta los beneficios, queda justificada su inversión.

Para estimar el cálculo del retorno de la inversión en JDO para un puesto de desarrollador en tareas de mantenimiento, desde el enfoque propuesto, se parte de las siguientes premisas:

- Periodo de cálculo, 2 anualidades, al considerar un plazo suficiente para alcanzar la formación y destrezas adecuadas usando una tecnología y proceso de desarrollo.
- Costes de producción de programas java de oscilan entre los [10,00 €, 100,00 €] a la hora.
- Los precios de mercado de las implementaciones comerciales para una licencia de desarrollo, oscilan entre los 1.000€ y los 6.000€. Examinados distintos precios fabricantes incluido mantenimiento, la inversión promedio en JDO sería de 3.349,80€. Se supone la misma inversión para JDBC, pues los precios oscilan en un rango similar.
- Una eficacia del 65%, el 65% del tiempo de producción es provechoso. Esta es una forma de considerar que no todo el tiempo trabajado es productivo, este dato corresponde a las cifras manejadas en España por alguna de las empresas del sector servicios de informática de nuestro entorno. Se considera una jornada de 40 horas semana y 52 semanas.
- El 40 % de la producción eficaz esta dedicado a codificar persistencia Java.
- JDO reduce un 35% el esfuerzo de desarrollo de la persistencia.
- El efecto de la mayor capacidad productiva por la reducción del esfuerzo de desarrollo, más cantidad de tiempo para producir, es ignorado para simplificar y presentar un escenario prudente.
- Los costes de formación, adaptación y entrenamiento para implantar la tecnología se ignoran, porque se suponen iguales y absorbidos en el porcentaje de tiempo no productivo estimado.

Resultando con este escenario de premisas, que se obtiene un rápido retorno de la inversión, en poco menos de un ejercicio la inversión en una licencia de JDO para un desarrollador, queda financiada por el ahorro en los costes de producción.

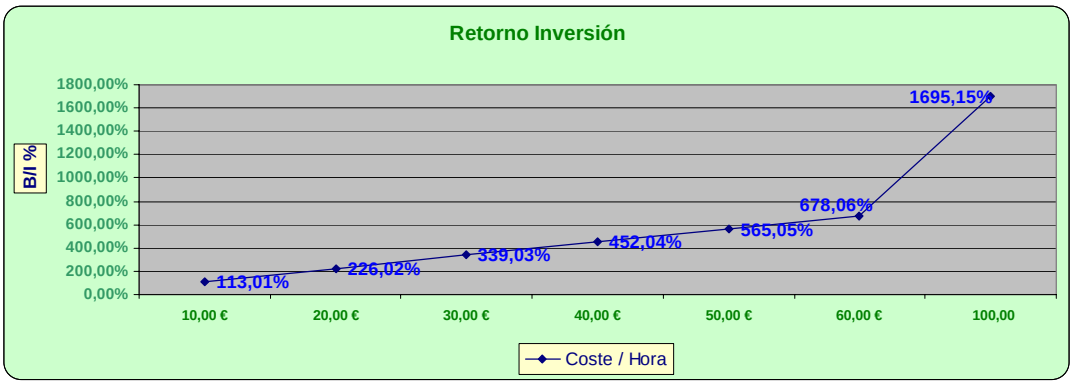


Figura 33 Retorno de Inversión en una licencia de JDO.

Un retorno de inversión (ROI) superior al 100%, señala que en el mismo periodo, inversión y beneficios se igualan, cuanto mayor sea este mejor, la inversión es más favorable. Bajo el escenario considerado, en todos los casos, la inversión en JDO sería superada por los beneficios reportados con el menor coste de producción.

La estimación hecha indica que inversión en JDO es rentable a corto plazo. Esta justificada su inversión considerando únicamente la reducción en los costes de producción debidos al menor número de líneas que mantener, en un escenario prudente que considera poco más de la cuarta parte (26%) del tiempo de producción dedicado a trabajar con persistencia en tareas de mantenimiento y depuración.

4.2.2.6. Conclusión de la productividad

JDO es más productivo que JDBC, requiere un menor esfuerzo de desarrollo, menos líneas de código, un 40% menos de código en la prueba de este trabajo.

La inversión en productos de JDO es rentable considerando exclusivamente la reducción del esfuerzo de desarrollo que aportan los productos JDO.

Respecto ODMG, JDO allana las diferencias entre los diferentes productos del mercado que adopta de forma particular el estándar. JDO no reduce el esfuerzo de desarrollo frente a ODMG, pero permite desarrollar una vez para distintas plataformas ODMG que adopten JDO. En este sentido, es más productivo y rentable JDO que ODMG.

4.3. Comparativa en los requisitos

Ha sido presentada una comparativa de JDO frente al resto de tecnologías respecto al rendimiento y su productividad. Queda conocer en que medida cada una las tecnologías satisfacen el marco de requisitos establecido en el primer capítulo. De forma breve y resumida el siguiente cuadro pretende cubrir este objetivo.

4.3.1. Estructurales

R01. Simplicidad y sencillez (Obligatorio)

SQLJ	Se apoya en JDBC. Más simple en la codificación, pocos elementos, para indicar el código embebido. Mayor complejidad al mezclar dos lenguajes y extender SQL para ser embebido. Hereda los defectos de JDBC, cuando es necesario usar SQL dinámico. Es necesario dominar SQL y Java. Más complejo dos modelos datos y objetos
JDBC	El peor, 40 clases e interfaces. Las soluciones son más complejas de plasmar que JDO Dos modelos datos y objetos
ODMG	Menos interfaces que jdo, y mayor complejidad. OQL basada en sintaxis SQL, se necesita conocer dos lenguajes. Un solo modelo ODMG adaptado a Java.
JDO	Sencillo y simple, 18 clases e interfaces. No requiere de sentencias en otro lenguaje. JDOQL es Java. Un solo modelo de objetos, el propio de Java.

R02..Fiabilidad (Obligatorio)

SQLJ	Tratamiento excepciones, verificación tipo compilación
JDBC	Tratamiento excepciones en tiempo de ejecución
ODMG	Excepciones y compilación. Es más fiable de los 4.
JDO	Excepciones y compilación

R03.Encapsulación (Obligatorio)

SQLJ	Es posible programarla, pero es fácil caer en el diseño no modular, al separar datos y objetos. Incluye capacidades para mapeo a tipos de usuario SQL99. Hay separación sin solución de continuidad entre los dos modelos, el de objetos y el de datos
JDBC	Es posible encapsular el código JDBC JDBC 3.0 incorpora clases para el mapeo entre SQL99 tipos usuario y clases java. Brecha que separa los dos modelos: Objetos y datos
ODMG	Sí, hay separación entre modelo de objetos y datos. Es el modelo ODMG Java
JDO	Si, no existe separación entre datos y objetos, un único modelo Java de objetos.

R04.Diferentes estilos de aplicación (Obligatorio)

SQLJ	Modelo transaccional. El estándar no considera la posibilidad de distintos estilos de aplicación, es una cuestión de programación y de las posibilidades SGBDR empleado. Aplicaciones OLTP. Bloqueo por SQL
JDBC	Ídem SQLj. Aplicaciones OLTP. Bloqueo SQL.
ODMG	Modelo transaccional ODMG 3.0 depende del OODBMS y el RDBMS, no incluye transacciones distribuidas o incluidas en serv. aplicaciones J2EE, manejo por programador. OLTP. Bloqueo expreso
JDO	Contempla distintos estilos de aplicación. Modelo transaccional pesimista y, optimista (opcional), bloqueo automático y según SG datos. Una Implementación estándar debe soportar las aplicaciones dirigidas a Entorno local e integrados en Serv. Aplicaciones EJB

R05. Diferentes arquitecturas (Obligatorio)

SQLJ	C/S de dos o más capas, no considera la posibilidad de objetos remotos distribuidos. El modelo base es una arquitectura monolítica, no se consideran

	otros modelos.
JDBC	Ídem SQLJ.
ODMG	Los productos comerciales pueden ser aplicados en diferentes arquitecturas, pero no es estándar.
JDO	Exigencia estándar funcionamiento en distintas arquitecturas desde empotrado a servidor aplicaciones J2SEE

R06.Extensibilidad (Opcional)

SQLJ	No se consideran otros mecanismos que no sean compatibles SQLJ. No puede ser añadidos nuevos mecanismos por el programador, extendiendo los las clases e interfaces
JDBC	Idénticamente que SQLJ
ODMG	Si, implementando las interfaces es posible persistir en otros mecanismos. Las clases base ofrecidas pueden ser extendidas con nuevas funcionalidad.
JDO	Si es extensible, se consideran distintos mecanismos relacionales y objetuales. Es posible implantar los interfaces exigidos. Además la sustitución de implementaciones esta asegurada por la exigencia de compatibilidad binaria

R07.Facilidad de uso (Obligatorio)

SQLJ	Dominar dos lenguajes Java y SQL. Presenta el código fuente menos legible.
JDBC	La facilidad de uso procede de que los patrones de código están solidamente establecidos.
ODMG	Buena, pero cada fabricante ofrece su implementación. Hay que aprender OQL
JDO	La facilidad de uso es Excelente, API común. Es necesario dominar JDOQL

R08.Escalabilidad y rendimiento

SQLJ	Si tiene muy presente el objetivo de rendimiento y del tratamiento de grandes cantidades de filas, dependiente de la implantación
JDBC	Son requisitos asumidos. Se ofrecen distintas opciones. De facto ofrecida por los fabricantes de conectores a bases de datos
ODMG	Son objetivo donde cada fabricante ofrece su propia aproximación.
JDO	Si, es una exigencia, algunas opciones son estándar. El fabricante puede decir la implantación más conveniente.

4.3.2. Funcionales

R09.Guardar y recuperar objetos (Obligatorio)

SQLJ	No. Permite SQL 99 tipos de usuario expresa. Cierre de persistencia programado a discreción
JDBC	No. Se programa para cada caso JDBC 3.0 SQLData, simplifica el desarrollo. Vía programación expresa.
ODMG	Si, hacer persistente mediante bloqueo expreso o guardar el objeto. Incluye el grafo de objetos dependientes
JDO	Si, automáticamente. Hacer persistente y ya esta. Incluye el grafo de objetos dependientes

R10.Operaciones básicas de actualización del estado (Obligatorio)

SQLJ	Programadas expresamente
JDBC	Programadas expresamente
ODMG	Si, de forma expresa en el contexto de una transacción, bloquear el objeto, provoca su consolidación al finalizar la transacción
JDO	Si, de forma explicita y transparente, JDO controla los cambios. MakeDirty(Obj)

R11.Identidad de los objetos persistentes (Obligatorio)

SQLJ	Programadas
JDBC	Programadas
ODMG	Ofrecida por el OODBMSO, o el RDBMS
JDO	Si, son considerados tres modelos: database, application, nodurable.

R12. Ortogonalidad (Obligatorio)

SQLJ	No, es ortogonal, no es fácil implantar código genérico de acceso a datos, fuerte acoplamiento entre el código, el modelo de objetos y de datos.
JDBC	No, pero es posible crear soluciones bastante ortogonales.
ODMG	Si, algunos tipos de objetos no pueden persistir. Es una exigencia el uso de ciertos tipos.
JDO	Si es el más ortogonal que el resto. Algunos objetos ligados no pueden persistir.

R13.Concurrencia (Obligatorio)

SQLJ	Si, Bloqueo expreso o implícito sobre los datos por invocación SQL
JDBC	Si, Bloqueo expreso o implícito sobre los datos por invocación SQL
ODMG	Si, manual manejada por el programador y según fabricante.
JDO	Si, el bloqueo implícito automático de los datos asociados en SGBD.

R14.Transacciones (Obligatorio)

SQLJ	Si, del SGBR. No afecta los objetos
JDBC	Si, del SDGBR. No afecta los objetos
ODMG	Si, pero dependen de la implementación de cada fabricante. Bloqueo a nivel de objetos
JDO	Si, locales o integradas en JCA (Connector interfaz), según otros API java estándar

R15.Diferentes mecanismos de persistencia (Obligatorio a RDBMS, otros opcional)

SQLJ	SI, solo mecanismos que ofrezcan un servicio JDBC.
JDBC	SI, solo mecanismos que ofrezcan un servicio JDBC.
ODMG	Si, bases de datos relacionales y de objetos.
JDO	Si, es posible, cada Gestor Persistencia asociado a uno. Un objeto queda vinculado a un solo Gestor.

R16.Distribución sobre distintos mecanismos

SQLJ	Si, a mecanismos que ofrezcan un servicio JDBC.
JDBC	Si, a mecanismos que ofrezcan un servicio JDBC.
ODMG	Si. Conexión simultáneas a BD distintas.
JDO	Si, mediante Gestor Persistencia, cada uno asociado a una BD. Conexiones múltiples

R17.Integridad objetos

SQLJ	No, Programada.

JDBC	No, Programada
ODMG	Si, no incluye eventos de BD
JDO	Si, incluye la notificación de algunos eventos de BD. Pero no considera estandarizar el mapeo de objetos a tablas, es responsabilidad de los fabricantes de soluciones JDO.

R18.Transparencia (Obligatorio)

SQLJ	No, la transparencia no se considera, pues tampoco tiene presente un modelo de objetos.
JDBC	No, la transparencia no se considera, pues tampoco tiene presente un modelo de objetos.
ODMG	Si, la transparencia es un requisito plasmado en las implementaciones ODMG
JDO	Si, es un objetivo fundamental de toda implementación

R19.Consultas (Obligatorio)

Todos cubren el requisito de construir y utilizar consultas ad hoc, utilizando un lenguaje de consultas.

SQLJ	SQL embebido para las consultas estáticas y SQL dinámico mediante JDBC
JDBC	SQL
ODMG	OQL
JDO	JDOQL

R20.Caché (Obligatorio)

SQLJ	Si.
JDBC	Si.
ODMG	Si.
JDO	Si.

R21.Iteradores (Obligatorio)

SQLJ	Si. ResultSet

JDBC	Si. ResultSet
ODMG	Si. Iteradotes para las colecciones persistentes y las consultas
JDO	Si. Iteradotes para las colecciones persistentes y las consultas

R22.Objetos representantes (Obligatorio)

SQLJ	No es una funcionalidad recogida.
JDBC	Idem
ODMG	Si considera esta capacidad.
JDO	Si. El estándar no menciona este tipo de objetos, pero la definición del grupo por defecto de carga, dividiendo la recuperación de atributos, en diferentes fases es una forma de proxy. Varios productos ofrecen esta capacidad expresamente.

R23. Aprovechar características de rendimiento de los servicios de datos

SQLJ	No es una característica estándar. Si bien es posible utilizar SQL nativo
JDBC	No es especificación estándar, aunque es posible aprovechar el SQL nativo
ODMG	La especificación no incluye referencia a este extremo.
JDO	Es una exigencia estándar

4.3.3. Funcionalidad avanzada

R24.Evolución del esquema

SQLJ	No se considera la evolución del esquema.
JDBC	No.
ODMG	No es una funcionalidad especificada por el estándar. Pero mayoritariamente los productos proporcionan esta funcionalidad..
JDO	No forma parte del estándar, pero buena parte de los productos tienen en consideración esta posibilidad.

R25.Versiones

SQLJ	No se considera esta posibilidad.
JDBC	No se considera esta posibilidad
ODMG	No es incluida esta funcionalidad en el estándar.
JDO	No es una funcionalidad estandarizada, pero los productos revisados consideran esta capacidad

R26.Automatismo actividades de mapeo

SQLJ	No
JDBC	No
ODMG	No es actividad estándar, pero se ofrece por parte de los fabricantes, aprovechando ODL y OIF.
JDO	No. El mapeo no esta considerado en el estándar 1.0, si el 2.0. Los productos más populares ofrecen esta actividad.

R27.Servicio de persistencia como fuentes de datos

SQLJ	No, es una funcionalidad presente, pues el objetivo es el acceso a datos relacionales exclusivamente
JDBC	Idem.
ODMG	No
JDO	No

R28.Coordinación Transaccional

SQLJ	No
JDBC	No
ODMG	No
JDO	Si, transacciones distribuidas en servidor J2EE

R29.Tratamiento de eventos de ejecución: Control de errores, actividad de persistencia

SQLJ	No contempla eventos del SGBD. Excepciones del SGBD
JDBC	Idem

ODMG	Excepiones levantadas por la BD.
JDO	Se atrapan las excepciones del mecanismo de persistencia y se consideran varios eventos relativos al ciclo de vida de las instancias.

R30.Medidas de rendimiento

SQLJ	Si
JDBC	Si
ODMG	-
JDO	No es un aspecto considerado en el estándar, pero si por las implementaciones JDO.

R31.Gestión del Almacenamiento

SQLJ	Si vía SQL.
JDBC	Si vía SQL.
ODMG	No. Cada fabricante implementa una clase Database, que puede incluir esta gestión.
JDO	No

R32.Trazabilidad

SQLJ	No. Según fabricante. No es requisito de la especificación.
JDBC	No.
ODMG	No.
JDO	No. Pero los productos suelen presentar algún nivel de seguimiento de la actividad de persistencia..

R33.Neutralidad respecto al lenguaje

SQLJ	No
JDBC	No
ODMG	Si es un requisito estándar, siempre que se usen los tipos estandar Java, C++ e incluso Smalltallk
JDO	Si, en cuanto que todos son objetos Java.

4.4. Resumen

Este capítulo de comparativa, ha presentado la posición de JDO frente a las JDBC/SQLJ y ODMG, en tres dimensiones rendimiento, productividad y satisfacción de los requisitos a satisfacer por toda solución de persistencia.

4.4.1. Rendimiento

El rendimiento de las implementaciones JDO, en aplicaciones interactivas es inferior, pero en el orden de magnitud que las otras tecnologías en comparación, JDBC/SQLJ y ODMG. Ante soluciones con modelos de objetos complejos y volúmenes importantes, JDO parece que aporta mejores rendimiento, que soluciones de programadas sobre SQLJ o JDBC.

En JDBC, las herramientas OR son la auténtica competencia de JDO para RDBMS.

JDO no suple a JDO con aplicaciones con modelos de objetos simples, dedicadas a presentar y actualizar datos.

Rendimiento					
Tecnología	JDBC/SQLJ	JDO			ODMG
		JDBC		OODBMS	OODBMS
Esquema persistencia	normalizado	normalizado	desnormalizado y serialización	objetos	objetos
Modelo de objetos simple	Superior	Aceptable	Alto	Aceptable	Aceptable
modelo de objetos complejo	Inaceptable	Inaceptable	Alto	Alto	Alto

4.4.2. Productividad

JDO es más productivo que las soluciones JDBC. Las pruebas aquí realizadas sostienen la tesis de la mayor productividad de JDO. Dependiendo del tipo de aplicación la reducción del código alcanza rápidamente el 40%. La valoración de la diferencia de productividad con JDBC es suficiente para rentabilizar la inversión en JDO, sin considerar otros factores.

JDOQL es menos potente que SQL. Pero con las vistas SQL y el mapping objetos-tablas, es posible aunar la potencia expresiva SQL con la facilidad, simplicidad y uniformidad de JDOQL y JDO. OQL también es mejor que JDOQL pero poco relevante en el mercado.

La siguiente versión aporta mayor funcionalidad a JDOQL, con proyecciones, sentencias y funciones con agrupamientos, y otras mejoras demandadas por la comunidad de desarrollo.

La inversión en JDO, queda justificada, por el ahorro en costes de mantenimiento. Es posible financiar la adquisición de herramientas JDO, obteniendo un retorno de inversión importante, según el escenario prudente considerado con costes de 30,00 €/hora x hombre, se obtiene un ROI de 339,03%. También están disponibles diversas implementaciones libres de pago para fines comerciales. JDO es más productivo que JDBC.

4.4.3. Verificación de los requisitos

JDO es la tecnología, que más requisitos satisface del marco de requisitos establecido en el primer capítulo con un total 30 requisitos de 33 definidos. Cabría señalar que JDO, cuando almacena los objetos sobre un RDBMS, delega en el fabricante la especificación del mapeo de clases en tablas, que, en mi humilde opinión, debería estar incluida en el estándar.

Comparados con JDO y ODMG, JDBC y SQLJ no satisfacen requisitos exigibles como la simplicidad, facilidad de uso, la encapsulación y todos los relativos al manejo de objetos, incluida la integridad, identidad, ortogonalidad y la transparencia, además de otros.

Tabla 5 Número de requisitos cumplidos por SQLJ, JDBC, ODMG y JDO

Número de Requisitos cumplidos					
		SQLJ	JDBC	ODMG	JDO
Obligatorios	22	10	9	20	22
Opcionales	11	2	2	7	8
Total	**33**	12	11	27	30

4.4.4. ¿Es mejor ODMG que JDO?

Al comparar las tecnologías de SQLJ, JDBC, ODMG y JDO, a primera vista parece JDO no supere demasiado a ODMG. Desde una perspectiva centrada en Java, diferencias entre ambos son más claras y a favor de JDO. Ambos ofrecen persistencia por alcance y transparencia de datos, es lo que los hace tan iguales. Pero revisando más a fondo una y otra especificación, a nivel técnico hay diferencias importantes.

En primer lugar, los modelos de objetos, el de ODMG no el modelo Java, es una adaptación que exige usar ciertos tipos para las estructuras de datos.

A nivel de los interfaces para manejar la persistencia, JDO es más potente, con funciones para consultar el estado y los callbacks.

El modelo de identidad en ODMG es el aportado por el sistema de bases de datos empleado y asignar nombres a los objetos. En JDO existen más posibilidades para definir la identidad (aplicación, bases de datos y no durable).

En ODMG los objetos persistentes pueden ser manejados dentro de la transacción, los objetos persistentes duran en la JVM, lo que la transacción que los controla. En JDO las instancias JDO perduran más allá de la transacción, están ligadas a su PersistenteManager, y se pueden modificar algunas de sus propiedades.

En JDO se pueden utilizar los callbacks para algunos eventos de la actividad contra el mecanismo de persistencia. En ODMG, no forma parte del estándar una funcionalidad análoga a los callbacks.

En OMG los bloqueos se manejan explícitamente, en JDO los bloqueos son automáticos.

En JDO, las transacciones pueden estar coordinadas por un gestor de transacciones de J2EE, en ODMG no incluye la facilidad de transacciones coordinadas y distribuidas.

Quizás la diferencia más importante en cuanto a Java, es la integración con EJB, asumida JDO y ausente de ODMG.

JDO allana las diferencias entre las implantaciones de los distintos sistemas de persistencia, siendo así, JDO es una inversión atractiva frente a invertir en conocimiento y licencias de distintos sistemas de persistencia de objetos. La inversión en JDO parece que es más interesante que ODMG y que JDBC solo.

Capítulo 5
CONCLUSIONES

Éste es el capítulo final. Presenta los inconvenientes y las ventajas de trabajar con JDO, su situación actual en el mercado, su evolución en curso y finalmente, contestar a algunas de las interrogantes, que surgen tras leer los capítulos anteriores: ¿Es JDO una tecnología útil, es otra más para persistir? ¿Qué organizaciones necesitan JDO? ¿Por qué utilizar JDO para RDMBS, en lugar de productos como Cocobase, Castor, jakarta OJB, Toplink o Hibernate? ¿Tiene futuro JDO?

Mientras era redactado este capítulo, alguno de los estándares, caso JDBC 4.0 o EJB 2.0, evolucionan para alcanzar más funcionalidad, dentro del marco de requisitos establecido en el primer capítulo, iniciando nuevos procesos de revisión de sus especificaciones estándar, que generan importantes debates sobre las virtudes, críticas a JDO y elección de otras alternativas, durante la primavera y el verano del 2004. Momentos en los Oracle, IBM y BEA se descolgaban de JDO, al rechazar la revisión de JDO, JSR-243 JDO 2.0, a favor de promover la iniciativa para acometer un sistema de persistencia ligero dentro de la próxima EJB 3.0.

Sabemos que JDO es posterior a todas las otras tecnologías con las que ha sido comparada, e incorpora buena parte de la experiencia acumulada, superando algunas de las limitaciones no resueltas por aquellas. Por ejemplo, la decisión de acceder a RDBMS, OODBMS o EIS, con una misma interfaz. Pensemos que entre los miembros del comité de expertos creador de la especificación, están algunos de los autores de la especificación ODMG, de herramientas para la persistencia en bases de datos relacionales y verdaderos expertos en Java, siendo así, a buen seguro, habrán sido consideradas las carencias y dificultades que debían ser resueltas, para alcanzar como objetivo la persistencia ortogonal.

JDO 1.0.1. [51], es la especificación una interfaz de aplicaciones ambiciosa, que pretende cubrir la solución al problema de la persistencia Java en su globalidad, para todos los ámbitos donde Java, pretende ocupar una posición destacada: el mundo de los servicios Web, las pequeñas y las grandes aplicaciones. Su visión es lograr la solución de persistencia ortogonal, aprovechando la experiencia acumulada en tecnologías de persistencia Java, de la que toma los conceptos de persistencia por alcance y transparencia de datos, que permite usar los cuatro sistemas más usados para guardar datos, los RDBMS, los OODBMS, los EIS y los archivos, y trata conseguir un proceso de desarrollo uniforme adecuado para pequeñas aplicaciones y grandes desarrollos. JDO logra que el código fuente de las clases del dominio pueda ser independiente de la persistencia, imponiendo pocas cortapisas al diseño de las clases. Con ello, JDO permite el programador centrar su esfuerzo en la lógica del negocio, que es la que proporciona valor, no malgastar esfuerzos en transformar los objetos en datos. La estrategia acometida para lograr su aceptación y estandarización, es proponer una solución sencilla, simple, clara, fácil de usar y aprender, auspiciada con el proceso de estandarización de la comunidad Java, JCP, en un corto plazo de tiempo.

Veamos los inconvenientes y ventajas de trabajar con JDO. Cuando se habla de inconvenientes y ventajas sobre alguna cuestión como las que nos ocupa, existe una dificultad esencial, los puntos de vista diferentes. Lo que para unos es un inconveniente limitante insalvable, para otros, simplemente es un pequeño escollo fácil de evitar. Las ventajas se tornan inconvenientes, dependiendo del criterio e interés de quién argumente. En este sentido, los inconvenientes son comentados para discutir su importancia, si se

pueden evitar, o si, serán corregidos por la funcionalidad prevista de la nueva versión JDO 2.0.

5.1. Inconvenientes

Los principales inconvenientes de JDO proceden de limitaciones por decisiones de diseño y de la falta de claridad en algunos aspectos de su especificación. Inconvenientes que, sin duda, son causados por ser la primera versión. Aunque no es la primera ocasión, en la que se intenta conseguir un desarrollo de persistencia ortogonal Java, ODMG para Java y la rechazada propuesta del proyecto Pjama, son ejemplos de ello. Si es la primera, que sigue la estrategia común, para incorporar nuevas funcionalidades y servicios de Java, de definir un conjunto de clases e interfaces concretos dentro del proceso JCP. ODMG dejaba algunas cuestiones abiertas los fabricantes, que dificultaban su aceptación y Pjama, proponía cambiar el lenguaje Java.

La integración y creación de servicios web J2EE con JDO es, el punto más oscuro de la especificación, que requiere una revisión que explique, cómo trabajar con J2EE, la web y JDO. La bibliografía disponible compensa esta dificultad, en especial con el libro titulado "Java Data Objects", de dos de los autores de la especificación Craig Russell y David Jordan [16].

Es una solución general al problema de la persistencia, por lo que no se beneficia de la riqueza y potencia de los sistemas subyacentes concretos, no ofrece la funcionalidad aportada por los RDBMS, OODBMS, u otros sistemas a los que puede acceder.

En algunos estilos de programación sitios Web, aparece el inconveniente de tener que hacer transitorios objetos persistentes, para una vez modificados, consolidarlos haciendo que sean persistentes nuevamente. Esto sucede con aplicaciones basadas en sucesivas peticiones http y en sesiones cliente EJB modificando objetos localmente, que están desligados de sus contenedores, bases de datos o EIS. En el primer caso, las aplicaciones Web con http, unas peticiones obtienen los objetos desde los servicios y bases de datos, liberan los recursos, sucesivas peticiones http, modifican estos objetos, para finalmente, ser consolidados con sus modificaciones en su contenedor original. En las EJB clientes remotos, necesitan transferir los objetos que modifican, esto requiere hacer transitorios las instancias JDO, para serializarlas, para al final, volver a deserializar y hacer persistentes los objetos transmitidos con sus modificaciones. Así, estos tipos de aplicación no aprovechan la funcionalidad JDO de transparencia de las modificaciones, que requiere que las instancias JDO estén enganchadas a un `PersistenceManager`. En [16] vemos un ejemplo de aplicación web con JDO, que sufre de la problemática citada. Ante la demanda por cubrir situaciones como la descrita, JDO 2.0 incluye los métodos al efecto, attach y dettach, que marcan un cierre de persistencia para ser manipulado fuera del contexto de un gestor de persistencia, manteniendo el control de modificaciones, para su posterior actualización al enganchar (attach) de nuevo.

Continuando con los estilos de aplicación, JDO es menos apropiado para las aplicaciones intensivas en manejo de datos relacionales, que no requieren de un modelo de objetos o es trivial.

Hay que tener cuidado cuando se usa, por primera vez, una clase que representa una clave primaria, asegurando que la clase de objetos persistentes asociados a esa clave primaria, este previamente cargada y registrada como persistente. Este inconveniente menor, se soluciona con la inclusión en la declaración de la clase clave, de una instrucción

estática de creación de un objeto persistente de la clase asociada a la clave. En [16] podemos ver distintos ejemplos de esta solución.

Otra limitación, ya mencionada antes en otros capítulos, es tener que recuperar necesariamente los objetos persistentes para su eliminación. De un lado, esto parece útil para facilitar la implementación de la integridad referencial. Por otro, es cierto que habitualmente los programas siguen una secuencia de localizar, mostrar, editar y validar los objetos. En razón de esta secuencia y de la facilidad para la integridad, sería el eliminar los objetos una vez recuperados. No obstante, la próxima versión añade a sus interfaces, un método de borrado directo en la base de datos sin recuperación previa.

El conjunto estándar de tipos de colecciones, asumidas Collection, Set y HashSet, podría haber sido más amplio, incluyendo a las consideradas opcionales, Vector, Map, ArrayList, HasMap, Hashtable, LinkedList, TreeMap, TreeSet. Clases e interfaces opcionales, soportados a criterio a los fabricantes. Esta es una pequeña limitación, que resta potencia de modelado de clases, para usar de las estructuras de datos más adecuadas al caso.

Dejar el soporte de los atributos de tipo array, a criterio de los fabricantes, parece un pequeño inconveniente, que compromete, otra vez, la potencia de diseñar clases. Esta es una limitación menor, porque se utiliza el tipo Vector habitualmente, que ofrece más funcionalidad y seguridad que los arrays.

Un inconveniente para quienes acostumbran a construir sus diseños a partir de interfaces, es que sólo las clases pueden ser persistentes, no las definiciones de interfaz, no obstante, los atributos persistentes si pueden ser referencias a interfaces. La siguiente versión JDO 2.0, permite la persistencia de interfaces.

Las funciones programadas para ser invocadas automáticamente por el servicio de persistencia, los callbacks para algunos eventos del ciclo de vida de las instancias persistentes, parecen pocos y su utilidad para manejar la integridad, no está clara. La siguiente versión cambia e introduce nuevas signaturas, teniendo presentes más eventos del ciclo de vida persistente.

JDO no soporta objetos complejos distribuidos entre distintos mecanismos de persistencia. Un objeto y su cierre de persistencia están bajo el control de un gestor PersistenceManager, que solo esta conectado a un mecanismo de persistencia: rdms, oodbms, ficheros o EIS. Esta limitación es fácil de superar con un poco de programación extra: cuando se alcanzan ciertos atributos, que de otra forma serían referencias normales, se emplea su valor para extraer, desde otros PersistenceManager conectados a mecanismos diferentes, los correspondientes objetos vinculados.

JDO no permite que objetos remotos sean persistentes. Esta es limitación establecida en la especificación de JDO, los objetos tienen que pertenecer a la JVM, en la que persisten. Lo normal es que cada JVM, recupere versiones propias de los objetos almacenados en la base de datos, cuya integridad se logra mediante las transacciones y la gestión del bloqueo. Esto no es impedimento, para que los objetos JDO sean manejados remotamente vía RMI o CORBA. Uno de los productos JDO Genie ofrece un servidor JDO, que proporciona, a JVM remotas, la interacción con estado compartido de los objetos persistentes manejados, sin transferir objetos entre las JVM.

JDOQL es muy limitado, según fue visto en un capítulo anterior. El hecho de tener que consultar clases concretas y no poder consultas por interfaces directamente, es una dificultad reseñable, que perjudica a los programas que pretenden crear código genérico. La próxima versión trae consigo, mejoras importantes en las consultas con la posibilidad de

consultar aquellos objetos cuyas clases implantan cierto interfaz, funciones agregadas sumarias (sum, avg, min, max, count), más funciones matemáticas y de manejo de patrones de caracteres, y posibilidades distintas en el retorno del resultado de las consultas como las proyecciones, filas en más puro estilo JDBC [39].

La correspondencia tabla-objetos, es una cuestión que la especificación JDO 1.0.1., no considera, que es crucial cuando se pretende una verdadera compatibilidad entre productos, y además, es importante para facilitar su adopción accediendo a RDBMS. La versión JDO 2.0, normaliza la especificación de la correspondencia entre clases y tablas en un esquema relacional.

Otro inconveniente, más relacionado con interoperatividad, tiene que ver con la representación interna la identidad, que decide cada fabricante. Los objetos manejados con sendos PersistenceManager de productos distintos, no pueden intercambiar las identidades internas manejadas. Esto es, obtener el id interno en uno, para recuperar el mismo objeto con ese id, en otro gestor de otra marca. Esta es una situación poco normal, integrar en una misma aplicación, productos de fabricantes distintos para acceder a los mismos objetos.

JDO es lento más que JDBC en el acceso a RDBMS. No tanto, las pruebas realizadas muestran que cuanto más complejo es el modelo de objetos utilizado, mejor adoptar JDO que programar una solución especifica en JDBC. Los tiempos de respuestas son adecuados frente al esfuerzo de programación. En las pruebas de rendimiento, para el mejor caso, la sobrecarga de JDO, ronda el 4%, que no parece demasiado,

Es falso que no sea necesario conocer otro lenguaje, más concretamente SQL. Ya se comentó antes, que se puede programar en JDO, sin saber SQL en un buen número de casos, pero, un buen programador debe profundizar cuanto sea necesario, para conseguir la solución más eficaz a un problema dado aprovechando los recursos a su alcance. Si perl, awk o una librería, solucionan un problema de forma más sencilla y eficaz, ¿Por qué no usarlos?. Si SQL y el código empotrado en base de datos, sirven para solucionar un problema, con menor complejidad y esfuerzo, entonces se deben utilizar.

La sobrecarga que introduce la persistencia, en los objetos transitorios pertenecientes a las clases persistentes, parece que puede evitar, considerando que en tiempo de ejecución los ejecutables de las clases originales deben estar accesibles. Es cierto que la persistencia JDO introduce una sobrecarga. Un análisis del código extendido con persistencia, revela que la sobrecarga al acceder a un atributo de un objeto transitorio, consiste, como máximo, en la verificación de dos condiciones.

Para algunos es un gran inconveniente, que el código ejecutable sea extendido automáticamente con la funcionalidad de persistencia, principalmente, porque oculta los detalles de la actividad de persistencia, e impide optimizar el código. La especificación estándar no exige implantar una extensión del código ejecutable. Los fabricantes pueden escoger si modificar el bytecode o modificar el código fuente. Alguno de los productos JDO vistos, permite la modificación del código fuente ampliado con la persistencia. Conservar el código fuente de las clases inalterado, y las ventajas esto conlleva, es la verdadera contrapartida de este inconveniente. De una parte, los principales productos JDO, facilitan la generación de bitácoras muy detalladas de la actividad de persistencia, del texto SQL generado en el acceso RDMBS y proporcionan medidas de rendimiento de cierto número de parámetros. De otro lado, uno de los principios básicos del desarrollo de programas, es la confianza en la corrección del software utilizado en su construcción [19], es por esto, que utilizamos compiladores y no ensambladores para programar.

Utilizar para los descriptores de persistencia XML, en vez de código compilado, podría ser considerado un inconveniente respecto al rendimiento y posible fuente de errores en ejecución. Flexibilidad contra eficiencia. Ciertamente puede ser más lento, leer metainformación de un archivo XML, si el paquete maneja algunos cientos de clases persistentes, pero en contra, ayuda a evitar tener que compilar por cambios en el mapping. Este tipo de compromisos, facilita la adaptación a distintas plataformas y productos. La compatibilidad binaria, garantiza que una clase compilada con un producto, debe ser ejecutada con otro, pero no dice nada de las exigencias de metainformación, donde cada fabricante usa sus tags vendor, para representar los datos que considera oportunos, esencialmente el mapping. De esta forma, embutir en un fichero externo el conocimiento exigido por un producto, ayuda a cambiar de producto, con solo sustituir los archivos de configuración apropiados.

Con JDO el esquema físico queda oculto, y no ofrece medios para manipularlo en tiempo de ejecución. Esto es un inconveniente para sistemas de información con requisitos extremos de rendimiento del almacenamiento y recuperación de objetos. Las aplicaciones con requisitos extremadamente exigentes de rendimiento de la persistencia, se benefician de una estrategia de contenedores dinámicos, donde en tiempo de ejecución, se usa el almacenamiento más adecuado. La alternativa en JDO, es utilizar colecciones y clases contenedoras, que corresponden con las alternativas de almacenamiento precisas.

Un inconveniente de JDO, es ser un estándar en proceso de revisión, que podría sufrir cambios drásticos, dando al traste con los desarrollos llevados a cabo hasta la fecha. Es cierto que JDO, esta siendo revisado, quizás a de finales del 2004, se disponga de una nueva versión definitiva. Pero uno de los objetivos, establecidos en el borrador de la propuesta de revisión publicada, es asumir la compatibilidad hacia atrás, para evitar problemas de migración.

Ante tanto inconveniente parece que no fuera útil trabajar con JDO. Lo cierto y verdad es, que la mayoría de las desventajas expuestas, no son tan importantes que impidan resolver la necesidad de persistencia de los proyectos Java más comunes. Buena parte de las limitaciones realmente importantes, son resultas en la próxima versión, como aparece en los comentarios introducidos, el desenganche y reconexión de instancias para las aplicaciones Web, las mejoras en JOQL y la estandarización del mapping.

5.2. Ventajas

La mayor ventaja perceptible al trabajar con JDO. 1.0.1, es que consigue difuminar la frontera entre la bases de datos y los objetos en memoria ejecutados dentro de la JVM, gracias a la transparencia de datos y la persistencia por alcance. Salvo en la lógica de manejo de persistencia, con la demarcación de transacciones, manipulación de claves y estado, consultas y el aseguramiento de la integridad referencial. El resto, la autentica lógica del negocio, se expresa en Java puro, no en términos de bases de datos, ni en XML, ni en SQL, solo y exclusivamente en Java. El resultado es similar a como los compiladores y lenguajes de programación, ocultan al programador la estructura de registros del procesador, caches, etc. Son muy importantes los beneficios consecuencia de ventaja fundamental.

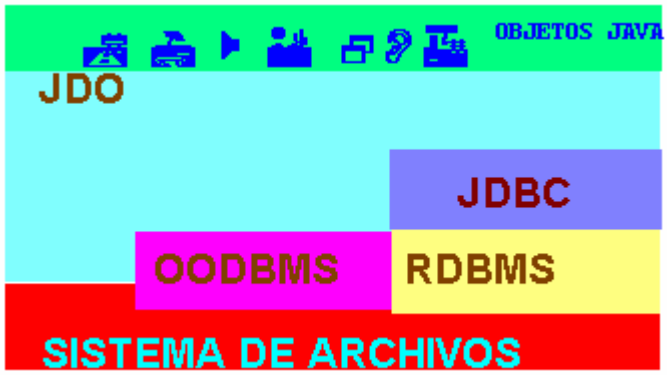


Figura 34 Posición de JDO en la persistencia java

JDO evita tener que conocer otros APIS para persistir. La figura anterior enseña gráficamente, como JDO oculta las diferencias existentes entre, los distintos medios usados para perdurar los objetos, ofreciendo una interfaz única. Tampoco es menos verdad, que hay situaciones donde es más apropiado utilizar JDBC y serializable, por rendimiento o porque no hace falta la funcionalidad de transparencia de datos y persistencia por alcance de JDO.

JDO es una solución global al problema de la persistencia de objetos Java.

Reduce el salto semántico y la complejidad. Consigue centrar el esfuerzo en el desarrollo en la lógica de negocio. Desplaza el foco de atención sobre el modelo de objetos, relegando a un segundo plano el modelo datos, punto de partida para muchos.

Una ventaja esencial de JDO es ser un estándar, en el sentido propio del término, por el procedimiento JCP, su madurez, y el número de productos disponibles. El hecho de que JDO se ha desarrollado al amparo del Java Comunity Process, es cuando menos, un gran punto a favor para llegar a ser un estándar real. Son varias las ventajas consecuencia de este hecho fundamental.

La independencia del fabricante, es una de esas consecuencias primarias de ser estándar. Los programas que usan los APIS JDO son independientes de los productos JDO, si un programa JDO sigue las recomendaciones de portabilidad, entonces la marca del producto utilizado queda oculta.

La independencia del esquema de persistencia (tablas, definición de archivos, …), es una gran ventaja. La correspondencia, esquema lógico y físico, destino de los estados de los objetos persistentes, puede ser modificada sin necesidad de cambiar los programas. Permite la evolución del esquema. Solo los productos que crean la ilusión de persistencia JDO deben conocer el mapping. Todos los productos vistos permiten comenzar la ejecución de las aplicaciones con ficheros descriptores de persistencia con mapping diferentes. En una aplicación JDBC, los cambios en el esquema provocan avalanchas de modificaciones en el código.

Independencia y neutralidad del mecanismo de persistencia. El código de los programas no está ligado a un mecanismo de persistencia concreto. Es una facilidad remplazar unas bases de datos por otras, permitiendo que la misma aplicación pueda almacenar sus objetos en distintas plataformas de almacenamiento de datos RDBMS, OODBMS, EIS y ficheros, sin necesidad de recompilar. Es posible programar sin necesidad de conocer el mecanismo de persistencia, ni tenerlo en cuenta el código. Dejar en un fichero aparte la metainformación necesaria para la persistencia, nuevamente como antes, consigue

mayor adaptabilidad y flexibilidad al cambio. Las pruebas de rendimiento en este trabajo, son un ejemplo de lo escrito.

JDO permite emplear el mecanismo más conveniente a la problemática concreta sin ser necesario de programar para cada mecanismo de persistencia expresamente.

Una posibilidad que alcanza relevancia, al utilizar JDO, es poder utilizar sistemas de bases de datos orientados a objetos, cuando el modelo relacional no cumpla con los requerimientos de rendimiento y actualización necesarios. Antes de JDO, o se desarrollaba para rdbms, o para oodbms.

Dicho ya con otras palabras, pero se debe destacar, que JDO sirve para acceder a los RDMBS, ocultando la falta de correspondencia entre tablas y objetos.

En el sentido que JDO es independiente de los mecanismos de persistencia, es una tecnología de integración ideal.

Consecuencia de las ventajas anteriores, es que JDO permite posponer decisiones de implementación, a momentos posteriores del desarrollo sin perdida de funcionalidad o el temor a tener que reconstruir el código.

El lenguaje de consultas JDOQL, tiene una sintaxis java, es simple, pero al mismo tiempo, una misma consulta sirve para acceder a sistemas radicalmente diferentes, RDBMS, OODBMS, ficheros, o EIS.

JDO es fácil de usar y aprender, los ejemplos presentados en este trabajo, así lo confirman. El número de elementos que es necesario manejar es relativamente pequeño, los patrones de código con persistencia más comunes, son como los presentados en los diferentes listados vistos en este trabajo.

En cuanto al rendimiento, JDO logra rendimientos adecuados con modelos de objetos complejos. Si el destino es un RDBMS, es posible superar el rendimiento de una solución JDBC convencional con un modelo normalizado de datos.

La programación con JDO es más productiva, requiere de menos líneas de código trabajar con persistencia JDO que con otras tecnologías. En este trabajo se ha alcanzado un 40% menos de líneas. Menor número de líneas repercute en menores costes de mantenimiento, más calidad y corrección. Algunos productos comerciales para persistencia relacional como Cocobase, argumentan que hasta el 85% del código de las aplicaciones de negocio, esta dedicado a codificar las operaciones de persistencia. Podemos imaginar que, si es cierto esto, despreocupados con JDO de codificar la traducción de objetos en filas, y viceversa, entonces se dispone de más tiempo para producir código que aporta valor, las reglas del negocio.

La generación automática de código es soportada por todos los productos de persistencia JDO vistos en la práctica. El código insertado directamente en los ejecutables, sin trastocar el fuente, facilita uniformidad, reduce las posibilidad de introducir bichos, reduce la prueba y mantenimiento del código fuente. Esta es una de esas ventajas que son inconveniente, según quien mire.

Las aplicaciones JDO son adaptables a distintas arquitecturas de aplicación, aplicaciones monolíticas locales, cliente-servidor, desplegadas en servidor Web, en servidor de aplicaciones.

El desarrollo con JDO facilita la economía de escala, en la producción de programas dirigidos a sectores concretos de los negocios, ya que, el modelo de objetos con sus reglas de negocio, permanece inalterado e independiente, de adaptar las vistas adecuadas para un

PDA o un PC, servidor Web o un servidor de aplicaciones. Empleando en cada caso los sistemas de bases de datos más apropiados y el producto JDO escogido. El caso de EJB y sus diferentes modelos de aplicación, necesita de programar cierto número de interfaces requeridos, enlazando con los servicios ofrecidos por el modelo de objetos. Un mismo producto terminado para diferentes arquitecturas y plataformas presentes en el mercado objetivo. Basta preparar distribuciones diferentes con ficheros de configuración adecuados con las propiedades de conexión, controladores de acceso, y mapping, que se adaptan al entorno de ejecución concreto, sin tener que modificar una línea en las clases del dominio.

JDO es una alternativa superior a los EJB entity beans, complicados y extensos de programar, son un error total reconocido por los propios miembros del grupo de expertos encargado del estándar EJB. El lugar que ocupan los Entity Beans, en la arquitectura J2EE, de servidor de aplicaciones, puede ser ocupado con JDO. Esta es la opinión David Jordan, expuesta en jdoCentral. No es de extrañar que la especificación del nuevo EJB.3.0, pretenda lograr un marco de persistencia ligero.

La inversión en JDO es viable y rentable. Es viable porque la propia reducción de costes es suficiente para sufragar la adquisición de licencias de productos JDO. Es rentable porque a los precios actuales, es posible lograr un alto retorno de la inversión.

Mejora la competitividad de las empresas de desarrollo en Java de aplicaciones para los negocios, en razón de las ventajas expuestas. No usar JDO, puede significar una perdida de competitividad.

El riesgo de inversión en JDO es manejable y reducido. El número de implementaciones de fuente abierta, y productos comerciales que permiten una evaluación exhaustiva de sus funcionalidades, permiten analizar la conveniencia y aplicabilidad a proyectos concretos consigue reducir el riesgo de invertir en JDO.

JDO es la tecnología, entre las elegidas en este trabajo, que satisface más requisitos del marco definido en el primer capítulo.

La adopción de un servicio de persistencia de objetos, no es una cuestión meramente de técnica introduce cambios de calado en la organización, en sus aspectos productivos.

5.3. Impacto de JDO en la organización

En el primer capítulo fueron expuestos los cambios que deben suponer en la organización de un servicio de pesistencia. Evidentemente, con JDO, se pueden lograr los planteamientos expuestos antes, posibilita un rápido aprendizaje, aumenta la productividad, flexibilidad. El programador es el principal beneficiado, porque ya no dedica esfuerzos a transformación de los estados de los objetos en datos. Con JDO es posible la división de las responsabilidades, entre programador y DBA, pues la información sobre el esquema de persistencia queda fuera de la fuente, en el descriptor de persistencia. La figura o papel del experto en persistencia, aparece para lograr adoptar los mejores compromisos entre dba y programadores, plasmados en el descriptor de persistencia.

La fórmula para adoptar JDO en la organización, debe seguir el proceso descrito en el primer capítulo. Definir un plan, su valoración, acometer unos proyectos piloto semilla del cambio, y evolucionar conforme al plan.

Insistir, que a los reiterados los beneficios de una programación con persistencia ortogonal, los fabricantes de programas ven mejorada su cadena de valor, y los consumidores, mejoran su inversión en programas añadiendo flexibilidad e independencia de sus relaciones con sus proveedores de programación y de sistemas gestores de datos, obtienen la capacidad para cambiar de sistemas de gestor de datos o combinar distintos según las necesidades, sin perdida de los inversiones previas, realizadas en el desarrollo de programas Java.

5.4. Impacto en la tecnología

Obviamente, como con cualquier otro producto JDO afecta al entorno tecnológico de la empresa. Gracias al planteamiento de ser una API más, no impone cambios radicales en la tecnología, que impidan utilizar las tecnologías previas a su incorporación. Bastará con incluir en el entorno de librerías y de ejecución, las librerías y ejecutables del producto JDO elegido.

Los scripts de desarrollo tendrán que ser, las tareas ant, maven o make con las necesarias del postprocesamiento de las clases para la ampliación de código. También puede ser necesario incluir, procedimientos para crear, actualizar, revisar y validar los esquemas de bases de datos.

La disponibilidad de la implementación de referencia, y otros productos JDO de fuente abierta, permite configurar equipos de desarrollo a coste cero. Es recomendable utilizar algún producto JDO que permitan automatizar las tareas de mapping, con un entorno gráfico manejable, e integrable con el entorno IDE Java elegido. Pero de esta forma, se pueden establecer etapas de desarrollo con responsabilidades separadas, posponiendo a las fases de producción más avanzada, los procesos de mapeo a rdbms, con generación scripts sql, creación del esquema de tablas.

5.5. Economía de JDO

Los resultados de este trabajo señalan pequeño el riesgo de invertir en JDO, frente los beneficios que puede reportar su aumento en la productividad y rentabilidad. Si la reducción en el número de líneas y tiempo de mantenimiento, son tan altos como parece por este trabajo, entonces, aquellos decididos a programar en Java, con bases de datos, deberían evaluar el coste de oportunidad y la probable perdida de competitividad por no invertir en JDO.

Antes de introducir JDO en una organización, se tiene que valorar con detenimiento y profundidad, los costes y beneficios del cambio. La idea de comenzar con una pequeña y limitada inversión, sobre algún prototipo que requiera persistencia, es una fórmula para comprobar y estimar, la variación en los costes y beneficios de invertir en JDO.

Cuanto mayor sea el coste hora-hombre, dedicado a tareas de mantenimiento mayor es el retorno de la inversión en JDO, según ha sido expuesto, en capítulo cuatro, con unos costes de 100 €, el ROI por la reducción costes, es superior al 1000%, índice alcanzado por pocas inversiones fuera del casino de la Bolsa.

5.6. Otros resultados del trabajo

El modelo relacional es adecuado en un buen número de casos, pero mirando las pruebas de rendimiento de este trabajo, vemos que no siempre. Este hecho es la razón fundamental por la que en los últimos tiempos, las empresas de motores relacionales, se apremian por incorporar en sus sistemas, otros modelos, el de objetos, el multidimensional y los basados en XML.

JDO amplia el mercado de los productores de sistemas de bases de datos objetuales, que son conscientes de la importancia de la integración con otros sistemas de bases de datos, en especial con los relacionales. Los gigantes de la bases de datos relacionales, también, van poco a poco dirigiendo sus premeditados pasos hacia sistemas objeto relacionales sin discontinuidades entre código y datos, que además también son capaces de manipular datos de otros sistemas.

En el sitio web de Doug Barry [36] y los sitios web [47] [46], encontramos referencias que desmienten que los OODBMS son lentos. Hoy por hoy, la mayor base de datos, públicamente conocida, es la base de datos Babar, con más de 890 terabytes, dispersas en más de un centenar de equipos y casi un millón de archivos [46].

En relación con los lenguajes de consulta y los OODBMS. Las principales bases de datos a las que se ha podido acceder en este trabajo, Versant, ObjectStore, Objectivity, implantan interfaces ODBC, proyectando los datos de los clases contenidas en las mismas, siguiendo un esquema directo de objetos a tablas, que permite consultar, recuperar e incluso actualizar, los datos asociados a los objetos que almacenan. Facilitando así, el acceso en consultas adhoc SQL y la utilización de herramientas de consulta y generación de informes habituales en los entornos relacionales.

5.7. Conclusiones últimas

Es momento de responder aquellas preguntas planteadas en la introducción del capítulo.

¿Es JDO una tecnología útil, es otra más para persistir?

Los argumentos expuestos en los apartados previos, los ejemplos ilustrados y las pruebas efectuadas, son argumentos de peso, que presentan JDO, como una solución global, transparente, ortogonal, y escalable al problema de la persistencia de objetos Java. Es posible acceder a suficientes productos que permiten evaluar su funcionalidad. Es una tecnología verdaderamente útil, y la mejor de las vistas en este trabajo.

¿Qué organizaciones necesitan JDO?

Todas las que necesiten un servicio de persistencia de objetos Java sobre una base de datos relacional u otro sistema, especialmente, si emplean sistemas gestores de datos de categorías distintas. Las empresas que estén evaluando usar una herramienta OR, deberían evaluar JDO.

¿Por qué utilizar JDO para RDMBS, en lugar de productos como Cocobase, Castor, jakarta OJB, Toplink o Hibernate?

Porque todos estos productos y otros, solventan exclusivamente el problema de la persistencia sobre RDMBS. Son soluciones parciales a la persistencia de objetos Java, mientras que, con JDO es posible utilizar otros sistemas: OODBMS, archivos o EIS. Porque los productos mencionados no son estándar, son productos únicos, con APIS y fórmulas

para consultar la base de datos distintas, vinculados a una organización, fundación o empresa. JDO logra una persistencia más ortogonal que el resto. En particular, Castor no soporta colecciones polimórficas Toplink de Oracle, no ofrece transperencia de datos. Cocobase, necesita cierta manipulación de las relaciones para lograr la transparencia y la sincronización expresa del estado de los objetos en la base de datos. Jakarta OJB, Hibernate o Castor, ofrecen transparencia en la recuperación, pero no de las actualizaciones, que tienen que ser sincronizadas por invocación expresa los métodos apropiados. En el verano 2004, en mi opinión, Hibernate era el lider de los productos OR, su principal inconveniente, es necesitar de programar en XML, pues además del mapping, es posible ajustar el comportamiento de hibernate mediante XML.

¿Tiene futuro JDO?

Citando un informe del Garnet Group, los fabricantes de soluciones Java, tienen la oportunidad de proporcionar una tecnología de integración ideal. JDO presenta un modelo de integración que, relega al usuario de los entresijos de los mecanismos de integración, dejando la responsabilidad a los fabricantes de sistemas gestores de datos. El usuario solo debe indicar qué, el fabricante debe facilitar cómo. La principal dificultad y riesgo de integración es la compatibilidad binaria, según el propio grupo de trabajo del JSR-12.

La evolución de JDO debería posibilitar una transición suave, desde las modelos actuales a especificaciones más evolucionadas y completas, que permitirían aumentar su cuota de mercado y la fidelidad de sus usuarios [31]. Esto puede ser conseguido con una especificación simple, fácil de usar, sujeta a un criterio de mínimos esenciales, que con el tiempo evolucione a modelos más complejos y completos. JDO tiene apenas un centenar de folios, muy pocos interfaces y detalla los modos y escenarios de uso. En mi modesta opinión, han escogido un buen camino, que garantiza su continuidad.

El mercado objetivo de la Red, esta creciendo. Se espera que ipv6 y los nuevos sistemas de telefonía móvil van a suponer la aparición de nuevas sociedades de intermediación con nuevos servicios en áreas de banca, el ocio, servicios de valor añadido,..., nuevos e-servicios. Nuevos e-servicios que deben integrar los sistemas heredados y las bases de datos. Necesidad que JDO puede ayudar cubrir como ninguna otra tecnología anterior.

El respaldo a JDO es muy representativo por parte de los más relevantes fabricantes de sistemas de gestión de datos de todos los enfoques. Basta con ver quienes formanban el grupo de expertos: Oracle, IBM, Informix, SAP, Software AG, Versant, Poet, Objectivity... Si bien algunos de los grandes, IBM, Oracle y BEA, se han descolgado en la revisión de jdo en curso, JSR-243 en curso.

JDO esta alcanzando gran eco entre la comunidad académica y en la empresa desarrollo Java. Son numerosos los foros, conferencias y artículos que encontramos donde se explica que es JDO, cuales son sus virtudes y, a veces también, sus defectos. No obstante, debería haber una estrategia de difusión especialmente dirigida a la empresa mediana y pequeña, no a las grandes empresas, que cuentan con departamentos de innovación e I+D. Microsoft si lo hace, inunda los medios con publicidad y noticias respecto a sus productos.

El sitio Web www.jdocentral.com, es el recurso de la Red referente esencial a todo lo concerniente de la actividad de difusión y debate sobre JDO. En este sitio los expertos reconocidos, los propios creadores de la especificación vierten sus opiniones sobre distintas cuestiones que surgen sobre el uso, adopción, carencias y nuevos requisitos que son demandados por la comunidad de miembros que va creciendo entorno a JDO.

El hecho de que diferentes empresas grandes y pequeñas respalden JDO, y que existan varios proyectos de fuente abierta, garantizan la continuidad de JDO, al menos, a medio plazo, en el transcurso del 2004, se produjeron diferentes movimientos del sector, con adquisiciones, alianzas, las cuentas de resultados presentadas de algunas de los productores de servicios JDO.

Los argumentos y comentarios expuestos, apuntan que JDO ocupa ya un lugar destacado entre las soluciones de persistencia para objetos Java. Parece que JDO esta en la línea de los estándares, que tienen el éxito asegurado.

Para terminar, resumir en pocas palabras el resultado del trabajo:

Los servicios de persistencia deben ser ortogonales, dotados de transparencia de datos y persistencia por alcance, superando la falta de correspondencia entre objetos y bases de datos, dotada de los recursos necesarios para forma eficaz y eficiente, difuminar la frontera entre los programas en ejecución y la base de datos, ocultando al programador los entresijos y diferencias entre los diferentes mecanismos de persistencia RDBMS, OODBMS, archivos y sistemas de información empresarial. Los servicios de persistencia ayudan a aumentar el valor en la producción y mantenimento de programas, con mayor productividad y rentabilidad.

JDO es la interfaz de programación de unos servicios de persistencia, que logran el mejor nivel de compromiso con la funcionalidad exigida por el marco definido en este trabajo, y reporta importantes beneficios con su adopción. Los servicios JDO, ofrecen ***Persistencia Java Transparente, Neutral, Escalable e Integrable***, persistencia transparente neutral respecto al tipo de sistema de almacenamiento de datos, preparada para afrontar cargas de trabajo importantes, que se adapta la arquitecturas de aplicación, actuales se integra con los servidores de aplicaciones Java aprovechando sus servicios.

JDO es la mejor tecnología de persistencia de objetos Java del momento, año 2004.

BIBLIOGRAFÍA REFERENCIADA

[1] M.ATKINSON R.MORRISON, **Orthoganally Persistent Object Systems**. The VLDB. Journal 4(3) pag 319- 401. 1995

[2] M. P. ATKINSON, F. BANCILHON, D. J. DEWITT, K. R. DITTRICH, D. MAIER, AND S. B. ZDONIK. **The object-oriented database system manifesto**. SIGMOD Conference, May 1990.

[3] KENT BECK, **Extreme Programming Explained, Embrace Change**, Addison-Wesley Octubre, 1999

[4] ELIZA BERTINO, LOREZON MARTINO. **Sistemas de bases de datos orientados a objetos**. Addison Wesley 1993

[5] GRADY BOOCH, JAMES RUMBAUGH, IVAR JACOSON, **El lenguaje Unificicado de Modelado**, Adison Wesley,1999

[6] BROWN, K. y WHITENACK, B. "**Crossing Chasms: A Pattern Language for ObjectRDBMS Integration" dentro del Pattern Languages of Program Desing** Vol 2, Addison-Wesley, 1996

[7] R.G.G. CATTELL, **Object Data Management** Addison Wesley, 1994

[8] EDGAR F. CODD, **The Relational Model for Database Management**: Version 2. Addison Wesley Publishing Co. 1990.

[9] C.J. DATE, **Introducción a los sistemas de bases de datos**, Addison-Wesley, 1986.

[10] K.R. Dittrich, conferencia seminario sobre base de datos orientadas a objetos, Murcia, 1996

[11] BRUCE ECKEL, **Thinking Java**. Prentice Hall.

[12] D.R. Edelson, **Smart Pointers: They're smart, but they're no pointers**, Usenix c++ conference, 1992.

[13] ERICH GAMMA, RICHARD HELM, RALPH JONSON, JOHN VLISSIDES, **"Design Patterns, Elements of Reusable Object-Oriented Software**", Addison Wesley, 1995

[14] CATHERINE HAMON & ARTHUR M. KELLER, "**Two-Level Caching of Composite Object Views of Relational Databases**" Int. Conf. on Data Engineering, Taipei, Taiwan, March 1995.

[15] RON JEFFIRIES, ANN ANDERSEN, CHET HENDRICKSON, **Extreme Programming Installed,** Addison-Wesley Pub Co; Octubre, 2000

[16] DAVID JORDAN, CRAIG RUSSELL **Java Data Objects.** O'Relly**.** April 2003

[17] HENRY F.KORTH, ABRAHAM SILBERSCHATZ, **Fundamentos de Bases de Datos**, MgGraw Hill.

[18] GRAIG LARMAN **Uml y Patrones**, Prentice Hall, 1999

[19] BERTRAND MEYER, **Object Oriented Software Construction 2nd Editon, Prentice Hall**, 1997

[20] SCOTT MEYERS, **More Effective C++,** Addison- Wesley, 1996

[21] NICHOLAS NEGROPONTE, **Beeing Digital**. Vintage Books. 1996

[22] OMG **Persistent State Service Specification 2.0,** formal/02-09-06

[23] R. ORFALI AND D. HARKEY, **Client/Server Programming with Java and CORBA**, John Wiley and Sons, New York, 1998. **Client/Server Programming with Java and CORBA, 2nd Edition**, Wiley, Marzo 1998

[24] CLAUS P. PRIESE, **"A flexible Type-Extensible Object-Relational Database Wrapper- Architecture"**

[25] TRYGVE REENSKAUG, **Working with Objects The OOram Software Engineering Method** Manning Publications Company,1995

[26] RIEL, ARTHUR J. **Object-Oriented Design Heuristics.** Addison-Wesley, 1996.

[27] Alejandro Roca, Juan Mármol, **CORBA SEGURO CORBA SLL Aplicaciones de Objetos distribuidas seguras** Asignatura Programación para las Comunicaciones, 2000
[28] ROGER S. PRESSMAN, **Sotware Engineering**, McGraw Hill 3ed. 1991
[29] ROBIN ROOS, **Java Data Objects,** Addison Wesley Publisher Ltd., 2002
[30] JAMES R RUMBAUGH, MICHAEL R. BLAHA, WILLIAM LORENSEN FREDERICK EDDY , WILLIAM PREMERLANI, **Object-Oriented Modeling and Desing**, Prentice Hall.1990
[31] SZYPERSKY, C. **Component Software: Beyond Object-Oriented Programming**. Addison -Wesley. Massachusetts. 1997.

Documentos en la Red

[32] SCOTT W. AMBLER Design of Robust Persistence Layer. http://www.ambysoft.com/persistenceLayer.html
[33] DR.MALCOLM ATKINSON, Pjama Project http://www.dcs.gla.ac.uk/pjava/
[34] DOUGLAS BARRY, DAVID JORDAN, ODMG: The Industry Standard for Java Object Storage http://www.objectidentity.com/images/componentstrategies.html
[35] DOUGLAS BARRY, TORSTEN STANIENDA Solving the Java Object Storage Problem http://csdl.computer.org/comp/mags/co/1998/11/ry033abs.htm
[36] Douglas Barry Web Services and Service-Oriented Architectures http://www.service-architecture.com/
[37] EMMANUEL CECCHET, Performance Comparison of Middleware Architectures for Generating Dynamic Web Content http://sardes.inrialpes.fr/papers/files/03-Cecchet-Middleware.pdf http://www.cs.rutgers.edu/~rmartin/teaching/spring04/cs553/cecchet02a.pdf 4th ACM/IFIP/USENIX International Middleware Conference, Rio de Janeiro, Brazil, junio 16-20, 2003
[38] DAVID JORDAN, Comparison Between Java Data Objects (JDO), Serialization an JDBC for Java Persistence http://www.jdocentral.com/pdf/DavidJordan_JDOversion_12Mar02.pdf
[39] David Jordan, New Features in the JDO 2.0 Query Language http://jdocentral.com/JDO_Articles_20040329.html
[40] DAVID JORDAN, The JDO Object Model, David Jordan, Java Report, 6/2001 http://www.objectidentity.com/images/jdo_model82.pdf
[41] ARTHUR M. KELLER y otros, Architecting Object Applications for High Performance with Relational Databases, 1995 www.hep.net/chep95/html/papers/p59/p59.ps
[42] WOLFGANG KELLER, Object/Relational Access Layers A Roadmap, missing Links and More Patterns http://www.objectarchitects.de/ObjectArchitects/orpatterns/
[43] Rober D. Poor. Hyphos http://www.media.mit.edu/pia/Research/Hyphos/,
[44] 007 benchmark, http:/www.objectidentity.com/images/007jdo.pdf)
[45] Apache Ant es una herramienta para gestionar tareas de compilación analoga makefile .http://ant.apache.org/
[46] Babar database http://www.slac.stanford.edu/BFROOT/www/Public/Computing/Databases/index.shtml

[47] Ejemplo alto rendimiento
http://www.fastobjects.com/us/FastObjects_IndustrySolutions_NetworkTelecom_StreamServe.asp http://www.objectivity.com/Industry/Success/telecom.html
[48] Gartner Group Java
www.gartner.com/webletter/actuate/article2/article2.html
[49] JAVA TECHNOLOGY: THE EARLY YEARS
http://java.sun.com/features/1998/05/birthday.html
[50] JDO MAX http://www.jdomax.com/c/extranet/home?e_l_id=22
[51] JSR-000012 JavaTM Data Objects Specification (Proposed Final Draft)
http://jcp.org/aboutJava/communityprocess/first/jsr012/index.html
Implementación Jdo de Referencia
http://jcp.org/aboutJava/communityprocess/final/jsr012/index2.html
[52] Productos de persistencia
http://www.persistence.com/products/edgextend/index.php,
http://www.objectmatter.com/,
http://www.chimu.com/products/form/index.html,
Cocobase(http://www.thoughtinc.com/cber_index.html),
http://www.secant.com/Products/OI/index.html,
Toplink(http://otn.oracle.com/products/ias/toplink/htdocs/sod.html)
Hibernate (http://www.hibernate.org)
ObjectRelationalBridge (OJB) http://db.apache.org/ojb/
[53] RUBIS Benchmark, http://rubis.objectweb.org
[54] Telefonia global www.iridium.com
http://www.objectivity.com/NewsEvents/Releases/archived_releases/1999/MotorolaDeliversIridium.html
[55] Torpedo http://www.middlewareresearch.com/torpedo/