UNIVERSITAT OBERTA DE CATALUNYA

Enginyeria Tècnica en informàtica de gestió

LA CAPA DE PERSISTÈNCIA: JAVA DATA OBJECTS
The persistence layer: Java Data Objects

TFC: Bases de dades.
Alumne/a: Francesc Xavier de Ros Villar
Dirigit per: Jordi Cabot Sagrera

CURS 2003-04 (Setembre/Gener)

LA CAPA DE PERSISTÈNCIA: JAVA DATA OBJECTS.

La Java Data Objects, també coneguda com JDO, és una de les moltes capes de persistència que podem trobar en el món de la programació. Aquesta capa de persistència té com a principal objectiu proporcionar un lligam entre dues tecnologies de caire diferent, les bases de dades i el llenguatge Java, i fer que els objectes propis del llenguatge siguin persistents en la base de dades. L'objectiu que es proposa aquest document es conèixer la problemàtica de la persistència dels objectes, trobar i estudiar les diferents solucions existents i estudiar-ne una de concreta, la capa de persistència JDO. S'estudiarà quines són seves les avantatges i desavantatges respecte els altres sistemes de persistència i el que suposen tots ells dins del món de les bases de dades i la informació. El document es divideix en diferents parts que aniran desenvolupant el que suposa JDO i el seu funcionament. La primera ens introdueixen en el marc de la problemàtica de la persistència dels objectes i els diferents sistemes de persistència que la resolen. Més endavant en la resta del document s'estudiarà la capa de persistència JDO, com la solució més avançada de la persistència dels objectes. S'introduirà JDO i s'estudiarà quina és la seva estructura i funcionament, d'aquesta manera es veurà com es pot aplicar la capa de persistència amb les diferents bases de dades. Més tard es veurà JDO de forma pràctica, s'enumeraran les més importants implementacions, i entre elles s'escollirà una per fer-ne un cas pràctic d'aplicació. Al finalitzar el document es podrà entendre de forma precisa, com la tecnologia que implementa JDO pot arribar a influir molt en el treball i la implementació de programes amb les bases de dades.

LA CAPA DE PERSISTÈNCIA: JAVA DATA OBJECTS. (Castellà)

La Java Data Objects, más conocida como JDO, es una de las muchas capas de persistencia que podemos encontrar en el mundo de la programación. Esta capa de persistencia tiene como principal objetivo proporcionar un enlace entre dos tecnologías de aspecto diferente, las bases de datos y el lenguaje Java, y además conseguir que los objetos propios del lenguaje sean persistentes en la base de datos. El objetivo que se propone este documento es conocer la problemática de la persistencia de los objetos, encontrar y estudiar sus diferentes soluciones existentes y estudiar una de ellas de forma concreta, la capa de persistencia JDO. Se estudiaran cuales son tanto sus ventajas y desventajas respecto otros sistemas de persistencia y que suponen dentro del mundo de les bases de datos y la información. El documento se divide en diferentes partes que desarrollan lo que supone JDO y su funcionamiento. La primera nos introduce en el marco del problema de la persistencia de los objetos y los diferentes sistemas de persistencia que lo resuelven. Más tarde, en el resto del documento se estudiara la capa de persistencia JDO, como la solución más avanzada de la persistencia de los objetos. Se introducirá JDO y se estudiara como es su estructura y funcionamiento, así se vera como se puede aplicar la capa de persistencia con las diferentes bases de datos. Seguidamente se explicara JDO de forma practica, se enumeraran las implementaciones más importantes, y escogeremos de entre todas una, para realizar un caso practico de aplicación. Al finalizar el documento se podrá entender de forma precisa, lo mucho que la tecnología que implementa JDO puede llegar a influir en el trabajo y la implementació de programas con bases de datos.

THE PERSISTENCE LAYER: JAVA DATA OBJECTS.

The Java Data Objects, more well-known like JDO, is one of the many layers of persistence that we can find in the world of the programming. This layer of persistence has as main objective to provide a connection between two technologies of different aspect, the data bases and the Java language, and in addition to obtain that the own objects of the language are persistent in the data base. The objective that sets out east document is to know the problematic one the persistence of the objects, to find and to study its different existing solutions and to study one of them of concrete form, the layer of persistence JDO. They studied as they are as much its advantages and disadvantages respect other systems to persistence and that they suppose within the world of them data bases and the information. The document is divided in different parts that they develop what supposes JDO and their operation. First it within the framework introduces of the problem of the persistence of the objects and the different systems to us from persistence that solves it. Later, in the rest of the document the layer of persistence JDO studied, like the solution more outpost of the persistence of the objects. JDO will be introduced and side studied as it is his structure and operation, therefore as the layer of persistence with the different data bases can be applied. Next JDO of form was explained practices, the most important implementations were enumerated, and we will choose of between all one, to make a case I practice of application. When finalizing the document will be able to be understood much of precise form, that the technology that implements JDO can get to influence in the work and implementació of programs with data bases.

ÍNDEX

LA CAPA DE PERSISTÈNCIA: JAVA DATA OBJECTS. 2

LA CAPA DE PERSISTÈNCIA: JAVA DATA OBJECTS. (CASTELLÀ) 3

THE PERSISTENCE LAYER: JAVA DATA OBJECTS. 4

ÍNDEX 5

GRÀFICS 8

TAULES 8

CAPÍTOL 1:LA PERSISTÈNCIA DELS OBJECTES. 12

1.1 Introducció: 12

1.2 Què és una capa de persistència? El Model de dades: 12

1.3 Principals objectius de les capes de persistència: 14

CAPÍTOL 2: INTRODUCCIÓ A LES DIFERENTS CAPES DE PERSISTÈNCIA. 17

2.1 Introducció: els diferents sistemes de persistència: 17

2.2 Sistemes de persistència. Parts d'una capa de persistència: 18
2.2.1 Comunicadors: 18
-ODBC, JDBC: 18
2.2.2 Sistemes: 19
- La Serialització i la correspondència o cartografia Objecte/Relacional (O/R): 19
2.2.3 Sistemes de bases de dades 20
- ODBMS: 20
- XDBMS: 20
2.2.4 Capes de persistència: 20
- ODMG: 20
- ADO: 21
- CMP (arquitectura EJB): 21
- JDO: 21

CAPÍTOL 3:ESTUDI D'UNA CAPA DE PERSISTÈNCIA: JDO 24

3.1 Característiques generals i conceptes introductoris: 24
3.1.1 Característiques generals: 24
3.1.2 Conceptes introductoris a JDO: 24

3.2 Arquitectura de JDO: 25
3.2.1 Classificació de les instàncies: 25

3.2.2 La identitat JDO: 26

3.3 El cicle de vida de les instàncies: 27
3.3.1 Els estats obligatoris del cicle de vida: 28
3.3.2 Transicions dels cicles de vida obligatoris: 29
3.3.2 Els cicles de vida opcionals: 32
3.3.4 Transicions dels cicles de vida opcionals: 32

3.4 El model de persistència dels objectes. 33
3.4.1 Transparència: 33
3.4.2 Les instàncies JDO: 34
3.4.3 Classes d'objectes de primera i segona: 34
3.4.4 Herència: 35

3.5 Classes i interfícies del paquet JDO: 35
3.5.1 Introducció al paquet JDO: 35
3.5.2 PersistenceCapable: 35
3.5.3 PersistenceManagerFactory: 36
3.5.4 PersistenceManager: 36
3.5.5 JDOHelper: 38
3.5.4 JDOImplHelper: 39
3.5.7 InstanceCallbacks: 40

3.6 Transaccions: 40

3.7 Consultes: 41
3.7.1 JDOQL: 41
3.7.2 Construcció de consultes: 42
3.7.3 La interfície Query: 42
3.7.4. Els filtres de les consultes: 42

3.8 Excepcions JDO: 43
3.8.1 La jerarquia d'excepcions de JDO: 43
3.8.2 Classes bàsiques de les excepcions: 44

3.9 El descriptor de persistència. Breu introducció a XML. 44
3.9.1 Breu introducció a XML orientada a JDO: 44
3.9.2 Descriptor de persistència: 45

3.10 Arquitectures de dos capes: 46

CAPÍTOL 4: INTRODUCCIÓ A DIFERENTS IMPLEMENTACIONS DE JDO 47

4.1 Enumeració de fabricants comercials : 47
4.1.1 FastObjects de Poet software: 47
4.1.2 FrontierSuite per JDO d'ObjectFrontier: 48
4.1.3 Kodo JDO de SolarMetric: 48
4.1.4 LiDO de LIBeLis: 48
4.1.5 OpenFusion JDO de PrimsTechnologies: 49
4.1.6 JCREDO de Riflexo : 49

4.2 Enumeració de fabricants gratuïts : 49
4.2.1 TJDO de SourceForge: 49
4.2.2 OJB d'Apache Software Foundation: 49

CAPÍTOL 5: ESTUDI D'UNA IMPLEMENTACIÓ JDO: KODO 3.0 DE SOLARMETRIC. 51

5.1 Perquè KODO 3.0 ? : 51

5.2 Característiques principals: 51
5.2.1 La especificació 2.0: 51
5.2.2 Especificacions anteriors: 52

CAPÍTOL 6: EXEMPLE D'APLICACIÓ PRÀCTICA DEL JDO 54

6.1 Plantejament general: 54

6.2 Especificació de l'exemple: 54

6.3 Instal·lar i configurar Kodo 3.0: 55

6.4 Les llibreries JDO i Kodo: 56

6.5 Configurar JDO amb la base de dades: 56

6.6 Definició dels objectes persistents: 57

6.7 Definició del DTD: 58

6.8 Transaccions entre el programa i la base de dades: 59

6.9 Consultes JDOQL: 60

6.10 Compilació i execució de l'exemple : 61

CONCLUSIÓ: 62

ANNEXA A: KODO 3.0: 63

A.1 Instal·lació de Kodo 3.0: 63

A.2 Descripció de les llibreries de Kodo 3.0: 64

A.3 Descripció de les eines de Kodo 3.0: 65

A.4 Descripció de l'arxiu *.properties: 69

A.5 Descripció del DTD de Kodo 3.0: 71
A.5.1 Relacions: 71
A.5.2 Esquema: 72
A.5.3 Correspondència: 72
A.5.4 Altres: 73

A.6 Incorporant l'opció de registre (log): 73

ANNEXA B: ALGUNS ASPECTES DE L'EXEMPLE: 74

B.1 Arxius de l'exemple: 74

B.2 Petit manual de l'exemple: 74

BIBLIOGRAFIA: 78

PAGINES WEB DE REFERÈNCIA: 80

Gràfics

Gràfic 1 14
Gràfic 2 16
Gràfic 3 29
Gràfic 4 29
Gràfic 5 29
Gràfic 6 30
Gràfic 7 30
Gràfic 8 30
Gràfic 9 31
Gràfic 10 31
Gràfic 11 31
Gràfic 12 32
Gràfic 13 33
Gràfic 14 33
Gràfic 15 43
Gràfic 16 54
Gràfic 17 55

Taules

Taula 1 18
Taula 2 42
Taula 3 50

Glossari de termes:

L'objectiu d'aquest glossari és recollir les paraules referents que ens seran útils per la lectura d'aquest document.

- **Base de dades:** són la representació integra dels conjunts d'entitats o instàncies d'informació i les seves interrelacions.

- **Java:** llenguatge de programació orientat a objectes, creat per Sun Microsystems per generar petites aplicacions exportables a Internet.

- **J2EE:** paquet del llenguatge Java que recull sobretot llibreries orientades a servidors d'aplicacions.

- **J2SE:** paquet del llenguatge Java que recull les llibreries bàsiques d'aquest llenguatge.

- **Capa :** també coneguda com API (Interface de Programació d'Aplicacions). És un conjunt de convencions de programació que defineixen com s'ha d'invocar un servei des d'un programa.

- **Persistència:** propietat que fa referència al fet que les dades que tracta programa, no es perdin en acabar la seva execució.

- **Transparència:** terme que fa referència al fet que qualsevol interacció entre dos aplicacions, si depèn d'una tercera, aquestes treballaran com si no hi fos. Això vol dir que no resta rendiment ni propietats, a cap de les dues aplicacions.

- **Implementació:** fa referència a programar una aplicació a partir d'un model d'aplicació teòric o esquemàtic.

- **Especificació:** és un document que especifica com ha de ser un sistema, un llenguatge, una capa, etc.

- **Correspondència:** terme que en angles traduiríem com "mapping", tot i que la seva traducció literària seria cartografia. Fa referència al fet que es dona quan dos elements són el mateix en si però de diferent natura.

- **Classe:** tipus de dades definit pel usuari que especifica un conjunt d'objectes que comparteixen les mateixes propietats.

- **Objecte:** cas de classe. És la representació d'un objecte real. És producte d'una Classe i compleix les propietats requerides del mateix objecte real que creiem que el defineixen i que són necessaris per la nostre aplicació. Es composa d'atributs, mètodes i relacions amb altres objectes.

- **RDBMS:** nomenclatura que fa referència a les bases de dades de tipus relacional.

- **ODBMS:** nomenclatura que fa referència a les bases de dades que tenen com unitat l'objecte.

- **XML:** també conegut com Extensible Markup Language, llenguatge extensible de marques. És un llenguatge dirigit a l'etiquetat de les parts que formen un document i és a més un metallenguatge que permet definir altres llenguatges, i que facilita la comunicació entre llenguatges de diferent natura.

- **Model de dades:** terme que fa referència al sistema formal i abstracte que ens permet descriure les dades a partir d'una sèrie de regles i convenis predeterminats.

- **SQL:** llenguatge de consulta creat per IBM que facilita el tractament de dades de qualsevol base de dades relacional.

- **Transacció:** és una acció. Mou una dada d'un lloc a un altre. És dona entre bases de dades, d'una aplicació a una base de dades, a l' inrevés, o entre aplicacions. No s'ha de confondre amb el terme propi de les bases de dades.

Encapsulació: una de les 3 propietats principals dels objectes. L'encapsulat es defineix com la propietat dels objectes que permet l'accés al seu estat únicament a través del seu interfície o de les relacions preestablertes amb altres objectes.

- **Servidors d'aplicacions:** qualsevol servidor que conté la major part de la lògica d'una aplicació i la manipulació de dades. No conté cap informació en si, sinó el tractament i distribució d'aquestes.

- **Controlador**: també coneguts com controladors, són els encarregats del treball Conjunt entre l'hardware i el software.

- **Incrustat:** en angles embedded. Bé associat al terme llenguatge incrustat. Permet diferenciar un llenguatge dintre d'un altre.

- **Rollback:** literalment vol dir marxa endarrera. Fa referència al terme propi de base de dades que ens permet recuperar l'estat anterior a una operació sobre la informació en una base de dades. No s'ha de confondre amb el mateix terme propi de les bases de dades, tot i guardar molta semblança.

- **Commit:** terme que defineix com l'acció de confirmar la transacció actual. Tots els canvis realitzats en la transacció es traslladen de forma permanent en la base de dades. El terme té un cert paral·lelisme amb el propi de les bases de dades, però no s'han de confondre.

- **Cache:** memòria que permet millorar el rendiment del sistema en emmagatzemar les dades més comunes i repetides. Donada la seva mida és molt ràpida, fet que justifica el seu ús.

- **PDA:** petits ordinadors de mà.

- **Instància:** El terme instància fa referència sempre a un objecte creat per la classe en temps d'execució conservant sempre les propietats i mètodes.

- **TAD:** tipus abstracte de dades. Estructura de dades que permet representar tipus de dades avançats.

- **Herència d'objectes:** una de les 3 propietats principals dels objectes. Permet generar classes i objectes a partir d'altres ja existents, recollint algunes de les seves propietats.

- **Clau d'identificació:** Clau que garanteix la unicitat de les instàncies o diferents elements.

- **Unicitat:** una de les 3 propietats principals dels objectes. Permet diferenciar com entitat un objecte en si d'un altre d'idèntiques propietats i valors en els seus atributs.

- **Cicle de vida:** conjunt d'etapes d'un objecte o instància.

- **PersistenceManager:** Agent administrador de la persistència dels objectes.

- **PersistenceManagerFactory:** Administrador de tots els agents de persistència de qualsevol aplicació.

- **Metadada:** terme que defineix un document que descriu el contingut, la qualitat, la condició i altres característiques una dada.

- **Consulta:** qüestió realitzada a una base de dades, en la que demana informació o informacions concretes en funció d'uns criteris definits.

- **Excepció:** terme propi dels llenguatges de programació, i fa referència als errors que . es poden produir durant l'execució d'una aplicació.

- **Encriptació:** tècnica per la que la informació es fa il·legible a terceres persones. Per poder accedir a la informació es sempre necessària una clau que només coneixen l'emissor i el receptor.

Capítol 1: La persistència dels objectes.

1.1 Introducció:

En el món de la informàtica han estat moltes les vegades que l'evolució d'una tecnologia ha influenciat en l'evolució d'altres tecnologies. Aquest és el cas de la programació orientada a objectes i les bases de dades.

Els llenguatges orientats a objectes o P.O.O, van provocar un d'aquests canvis. Modificaren l'estructura i visió que es tenia de la programació, introduint el concepte de TAD i d'objecte, a partir d'un símil amb la realitat. Aquest fet facilità la programació, a més d'aconseguir un important estalvi de feina que fins aleshores era molt feixuga i complexa. La P.O.O va aconseguir objectius fins al moment desitjats però no realitzats, com són la portabilitat, la reutilització, l'herència, etc. Amb el temps, la P.O.O s'ha anat consolidant passant d'una alternativa de la programació a una realitat. Un dels fruits d'aquesta evolució són llenguatges com JAVA de Sun Microsystems, entre altres. El llenguatge JAVA ha arrelat gràcies sobretot a internet, passant a ser el referent en aquest àmbit. I internet com ja es conegut, també s'ha consolidat com un mitjà de comunicació.

Tal com dèiem, un dels camps afectats directament pels P.O.O varen ser els sistemes d'emmagatzemat. Les bases de dades fins llavors existents, partien d'una realitat diferent; els sistemes eren aleshores les anomenades bases de dades relacionals i els sistemes distribuïts, i tenien la seva pròpia dimensió de problemes. Però va arribar la necessitat de què les dues tecnologies treballessin conjuntament, gràcies sobretot a l'èxit d'internet. La difusió d'OO a internet, va afectar a les bases de dades relacionals, que no estaven preparades per donar suport als objectes, però que si estaven i encara ho estan, implantades a la major part de la indústria. Calia doncs que el programadors i la indústria anessin adaptant-se a la nova realitat de la programació.

Qualsevol aplicació per si mateixa, no pot emmagatzemar les dades que genera l'usuari. Una aplicació OO, no es cap excepció. No pot guardar per si mateixa els objectes ni la seva informació, requereix de sistemes d'emmagatzematge independents, com poden ser els fitxers externs, entre altres, sistema només profitós per a petites quantitats d'informació o petites aplicacions. I els potents sistemes de base de dades relacionals no s'adapten, com ja hem dit, als objectes. És l'anomenada problemàtica de la persistència de dades sobre els objectes.

Aquesta necessitat impulsà en el seu moment, que s'intentes trobar un vincle comú de treball entre les aplicacions de llenguatges OO i les bases de dades relacionals, l'anomenada persistència dels objectes.

1.2 Què és una capa de persistència? El Model de dades:

Entenem per persistència, el fet d'emmagatzemar la informació generada per una aplicació en un sistema d'emmagatzemat, amb l'objectiu de què es pugui fer servir més endavant.

La persistència dels objectes és, com el seu nom indica, l'emmagatzemen dels objectes. Aquest fet permet que sempre es mantinguin disponibles per l'aplicació, per què aquesta els pugui consultar, modificar, combinar, etc.; en resum gestionar-los. La persistència dels objectes ha de conservar l'objecte en l'estat que l'usuari cregui interessant, pels seus objectius.

A continuació, aprofundirem en alguns dels aspectes sobre els objectes, que ens permetrà entendre millor el perquè de la problemàtica de la persistència dels objectes.

Un objecte en si es compon d'atributs, mètodes i relacions amb altres objectes. Els atributs podríem dir que són l'equivalent als registres o camps d'una base de dades. Cadascun defineix els aspectes que composen un objecte, com podria ser el color en un cotxe, el tipus de motor del cotxe, en definitiva tots aquells aspectes que creiem que defineixen l'objecte dins la realitat i ens són imprescindibles definir en el nostre programa. Els atributs poden ser de tipus primitius, referenciats o altres objectes. Els mètodes els podem definir com funcions o operacions que treballen sobre els mateixos atributs del objecte, o sobre relacions amb altres objectes. Les relacions entre objectes poden ser l'herència, d'ús o associació, parametritzada i la composició, entre altres.

Com ja hem dit abans, les bases de dades treballen amb registres. Els registres són les unitats bàsiques de treball de la informació de les bases de dades relacionals. Aquestes unitats es classifiquen en diferents tipus, segons la informació: numèrics, caràcters, etc.. Són en definitiva els tipus primitius, i formen el que anomenem model de dades.

Tots els llenguatges "no orientats a objectes", tenen un model de dades molt semblant a les bases de dades relacionals. Les variables són de tipus primitiu al igual que els registres de les bases de dades relacionals.

Una aplicació que treballa amb variables tipus primitiu pot treballar directament i requereix de poques adaptacions amb una base de dades relacional, però com podem entendre una base de dades relacional qualsevol, no pot emmagatzemar un objecte a partir de registres. Pot arribar a tenir un equivalent, a partir dels registres, semblant a l'objecte, però mai podrà emmagatzemar totes les seves propietats, mètodes i relacions amb altres objectes que el defineixen, ja que una base de dades relacional no hi està preparada. La base de dades relacional no té forma d'emmagatzemar de forma directa els mètodes i les relacions. Cal doncs adaptar d'alguna manera els models de dades. Abans hem definit de forma general com és el model de dades dels objectes en si, i ja coneixem de forma general el de les bases de dades relacionals. Ara la definirem amb una aspecte més formal. Segons Antonio Moreno Ortiz, el model de dades d'objectes el podem descriure com:

"Una col·lecció de conceptes ben definits matemàticament que ajuden a expressar les propietats estàtiques i dinàmiques d'una aplicació amb un ús de dades intensiu. Conceptualment, una aplicació potser caracteritzada per:

- Propietats estàtiques: entitats (o objectes), propietats, (o atributs) d'aquestes entitat, i relacions entre aquestes entitats.
- Propietats dinàmiques: operacions sobre entitats, sobre propietats o relacions entre operacions.
- Regles d'integritat sobre entitats i les operacions (per exemple, transaccions) ".

La diferencia dels models de dades va fer que els programadors amb la necessitat per trobar la forma de que els objectes siguin persistents necessitessin adaptar els models de dades d'alguna manera. Es tractava que trobessin un model corresponent o equivalent entre els dos models de dades, el dels llenguatges OO i els de les bases de dades.

Un exemple d'adaptació que es pot fer és a partir una taula de la base de dades definir una sèrie de registres com si fossin els atributs d'un objecte. Aquesta adaptació ens serviria per poder fer moltes instàncies del mateix objecte. Aquest cas ens serviria per classes de definició d'objectes molt simples, ja que no podrien ni contemplar d'emmagatzemar els seus mètodes ni relacions amb altres objectes, i menys encara fer un objecte compost per altres.

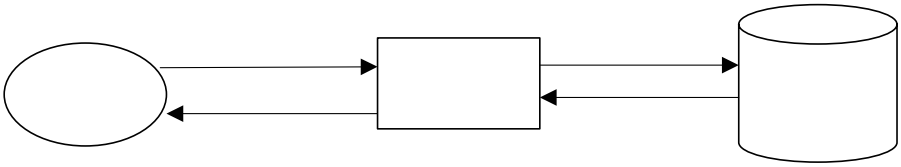
Com aquest exemple, varen aparèixer varis sistemes, alguns igual de rudimentaris o més complexos que intentaven fer una correspondència entre els models de dades de les bases de dades relacionals i els objectes.

Però també van aparèixer idees que solucionaven el problema de la persistència de forma més directe i automatitzada. Algunes d'aquestes idees les veurem en el següent capítol. I com veurem també, varen evolucionar gràcies sobretot a les necessitats cap un sistema automatitzat que aconseguís la correspondència entre els models de dades, les capes de persistència.

1.3 Principals objectius de les capes de persistència:

El plantejament a priori de la capa de persistència ens deixa un interrogant. Quines característiques ha de tenir per poder arribar a ser un sistema consolidat?

La idea principal de la capa de persistència la podem entendre millor en aquest gràfic:



Gràfic 1

Segons Scott W.Ambler una capa de persistència robusta dirigida cap bases de dades relacionals ha de complir els següents requeriments:

- **Diversificació dels mecanismes de persistència.**
  Ha de poder treballa amb diversos sistemes emmagatzemat de dades, no amb un de sol. Alguns podrien ser bases de dades, sistemes de fitxers, etc..

- **Encapsulació complerta del mecanisme de persistència.**
  El sistema només ha de poder emmagatzemar, eliminar i rebre els objectes. De la resta d'operacions s'ha d'encarregar la capa de persistència.

- **Accions sobre múltiples objectes.**
  La capa ha de consultar o eliminar múltiples objectes a la vegada.

- **Transaccions.**

Ha de que treballar amb accions combinades, com consultar - esborrar. O permetre parametritzar les transaccions. O sigui realitzar totes les operacions pròpies del llenguatge SQL, en una base de dades.

- **Extensibilitat.**
  Ha de permetre que l'aplicació generi nous tipus d'objectes, i capacitat de modificar els mecanismes de persistència fàcilment. Flexible a les necessitats.

- **Identificadors dels objectes.**
  Atribut numèric per cada objecte que permeti identificar-lo respecte altres totalment iguals.

- **Cursors.**
  Com sistema per poder rebre els resultats de les operacions de la base de dades.

- **Proxies.**
  És una aproximació completaria als cursors. És un objecte representatiu d'un altre però no fa incursió en el mateix, sinó per sobre, del que l'objecte representa. És com un apuntador, sobre les dades d'un objecte.

- **Registres.**
  En el sentit de generar informes de resultats.

- **Múltiples arquitectures.**
  S'ha de poder adaptar tant a arquitectures client/servidor de dues capes com de n- capes.

- **Treballar amb diferents bases de dades de diferent fabricant.**
  La capa de persistència a de poder adaptar-se fàcilment sense tindré que modificar l'aplicació.

- **Múltiples connexions.**
  Ha de poder treballar amb sistemes que no tenen la base de dades de forma centralitzada en un sol lloc sinó en variïs.

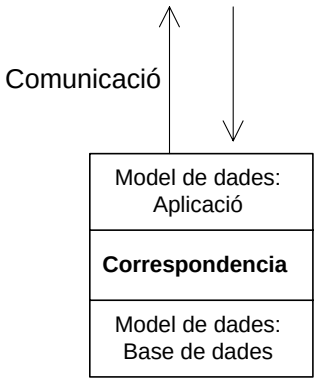
- **Controladors natius o no.**
  Ha de treballar amb els sistemes d'accés a les bases de dades més comuns o amb els proporcionats pels fabricants.

- **Consultes SQL.**
  Ha de permetre fer consultes SQL, no de forma obligada, sinó com una excepció, i ens casos molt forçats. "

Aquestes són les principals característiques que de forma general ha d'implementar qualsevol capa de persistència. Aquestes característiques ben fixades i implementades, han de poder conformar una capa de persistència consistent i fiable, per tal que un programador experimentat pugui confiar-ne i escollir-la davant altres opcions.

Totes aquestes característiques conformen l'anomenada transparència. Aquesta característica té com objectiu entre altres no restar-li cap característica ni a les aplicacions generades pel llenguatge O.O, ni a les bases de dades, aprofitant al màxim totes les propietats i característiques, en benefici del conjunt. La transparència de la capa, permet que tant l'aplicació com la base de dades, treballin com si es comuniquessin entre elles de forma directe, configurant un model de dades comú. Inclòs si canviéssim la base de dades, l'aplicació no té perquè veure's afectada en cap sentit, i permet continuar treballant de la mateixa manera. La transparència s'encarrega de tornar a lligar els diferents models de dades de forma automatitzada, sense restar rendiment i propietats. Com podem veure la transparència respecta la independència a les dues parts a la vegada que permet que treballin de forma conjunta com un sistema únic. Aquest és el motiu pel que la persistència per capes també s'anomena persistència transparent.

Tots els aspectes anomenats per Ambler, conformen la persistència transparent, sense menysprear cap de les característiques esmentades, ni donant més importància a una que a una l'altre. Això permet, com ja he dit anteriorment, conformar un sistema sòlid, útil al programador, i que li permet eliminar un gran nombre d'errors en la programació, enlloc de produir-ne de més. Anomenades les característiques de Ambler, podem dividir-les en dos grups de forma general, i segons el seu objectiu, la comunicació i la correspondència de models.



Gràfic 2

En resum, la transparència apropa les avantatges que ens donen de les dues parts i els seus models de dades, sense restar funcionalitat i característiques a cap de les parts, i de la manera més automatitzada possible.

Capítol 2: Introducció a les diferents capes de persistència.

2.1 Introducció: els diferents sistemes de persistència:

La necessitat de la persistència dels objectes, ha fet que al llarg del temps, s'hagin desenvolupat diferents sistemes per tal d'aconseguir aquest l'objectiu. Els sistemes més primitius han estat la primera pedra que ha permès donar pas a sistemes més evolucionats com són les capes de persistència, l'últim esglaó evolutiu dins de l'escala evolutiva de la persistència d'objectes, on fins ara hem arribat. Quasi tots dels diferents sistemes de persistència encara s'utilitzen en major o menor grau. En aquest capítol introduirem alguns d'aquests sistemes, veurem l'evolució i també parlarem d'alternatives com les bases de dades d'objectes.

Els principals sistemes d'emmagatzemat de dades amb els que es treballa habitualment, són les bases de dades. Entre les bases de dades és distingeixen dues de fonamentals, les relacionals i les orientades a objectes, a més de la forma objecte-relacionals barreja de les dues bases de dades. Altres sistemes amb els que ens trobem que permeten realitzar emmagatzemat de dades, són els sistemes de fitxers, com poden ser fitxers ASCII o els documents XML. Altres sistemes que podem anomenar per la seva creixent importància són els que treballen amb arquitectures d'objectes distribuïts com EJB, i les bases de dades XML.

Dins de l'evolució dels sistemes de persistència d'objectes, com podrem veure, van treballar sobre els diferents sistemes d'emmagatzemat . Els primers principalment sobre fitxers, i base de dades relacionals. Els més evolucionats, que són les capes de persistència, a més permeten treballar sobre sistemes distribuïts, i es comença a parlar de treballar sobre bases de dades XML.

Cal destacar però, que tot i la diversitat de sistemes de persistència, com podrem veure en la següent taula, només les capes de persistència estan orientades, tal i com va indicar Ambler, a múltiples sistemes d'emmagatzemat, fitxers, diferents bases de dades, etc.. La següent taula enumera alguns dels diferents sistemes i capes de persistència. Tots ells són comparats segons les característiques de Ambler. Com podem veure, els únics sistemes de persistència d'objectes que compleixen la majoria de les característiques són les capes de persistència.

Taula comparativa de les característiques de Ambler.

Requeriment	Serialització	JDBC-ODBC	ODBMS	EJB	JDO	ODMG	ADO
Diferents tipus mecanismes	Sistemes d'arxius	RDBMS	ODBMS	RDBMS,EAI, Sistemes d'arxius, etc.	Sistema d'arxius, RDBMS, ODBMS, EAI, monitors TP, bases de dades Cobol,i altres	Sistema d'arxius, RDBMS, ODBMS, EAI, monitors TP, etc..	RDBMS

Encapsulació complerta del mecanisme de persistència.	Si	Si	Si	Si	Si	Si	Si
Accions multi-objectes.	No	Si	Si	Si	Si	Si	Si
Transaccions.	No	Si	Si	Si	Si	Si	Si
Extensibilitat.	No	No	No	No	Si	Si	Només tipus d'objectes
Identificadors dels objectes.	Si, Manual	Si, manual	Si	Si	Si	Si	Si
Cursors.	No	Si	No	Si	Si	Si	Si
Proxies.	No	No	Si	Si	Si	Si	Si
Registres	Si, manual	Si, manual	Si	Si	Si	Si	Si
Múltiples arquitectures.	No	No	No	No	Si	No	No
Treballar amb diferents bases de dades de diferent fabricant.	No	Si	No	Si	Si	Si	Si
Múltiples connexions.	No	No	Si	Si	Si	No	No
Controladors natius o no.	No	No natius	No	No natius	No natius	No natius	No natius
Consultes SQL.	No	Si	Si	Si	Si	No	Si

Taula 1

2.2 Sistemes de persistència. Parts d'una capa de persistència:

Hem classificat de manera lliure els diferents sistemes i capes de persistència. Com ja hem definit, una capa de persistència té dues parts o capes, la part de comunicació entre l'aplicació i el sistema d'emmagatzemat, i la de correspondència dels models de dades. Tot i això hem inclòs com un sistema de persistència, els sistemes de comunicació, que anomenem comunicadors, ja que són el sistema més primitiu de fer persistència de dades i objectes. Entre els sistemes fem referència a diferents maneres més elaborades de fer persistència, i que no són ni una capa de persistència ni una base de dades en si, i tenen una correspondència més complex i sistematitzada que els comunicadors. Apart de les capes de persistència també anomenem les diferents bases de dades, ja que en certa manera tenen el seu paper dintre de la persistència dels objectes.

2.2.1 Comunicadors:

-ODBC, JDBC:

Potser són la forma més primitiva de persistència dels objectes. No són en si una capa de persistència, com ja hem dit, però els anomenem perquè encara es fan servir (JDBC té l'especificació 3.0), i perquè són una peça fonamental per entendre l'evolució dels sistemes de persistència d'objectes cap a les capes de persistència dels objectes, objectiu d'aquest document. Són el que hem anomenat com part de comunicació de la capa de persistència de Ambler. Anomenem junts els dos sistemes perquè el que permeten fer en si és el mateix. JDBC però està només orientat a llenguatge Java, i ODBC és de caràcter més general, cap a plataformes de Microsoft sobretot, el seu fabricant. Sun a més ha generat un sistema mixta, JDBC-ODBC bridge, inclòs al Java2 SE, per poder treballar des de Java, amb els dos comunicadors.

La seva missió principal és comunicar per mitja d'un controlador les aplicacions i les bases de dades. Permeten implementar l'anomenat SQL incrustat, perquè des de l'aplicació, puguem realitzar consultes SQL a les bases de dades. Com podem veure no tracten amb els models de dades, és el mateix programador qui fa les correspondències entre els diferents models. Aquest fet provoca que el programador ha de tenir un nivell molt alt d'especialització, en les dues matèries, si no vol que les seves aplicacions tinguin un gran nombre d'errades i siguin poc dinàmiques. A més li resta funcionalitat als objectes, al no conservar les seves relacions, mètodes i altres aspectes propis. Són sistemes senzills i òptims per petites aplicacions, però no aptes per grans aplicacions.
JDBC es troba molt relacionada amb la capa de persistència JDO. És l'element de comunicació entre JDO i la base de dades, com més tard veurem. Per tant és molt important conèixer el seu funcionament amb diferent bases de dades.

2.2.2 Sistemes:

- La Serialització i la correspondència o cartografia Objecte/Relacional (O/R):

Són en si el principi del que anomenem part de correspondència de les característiques d'una capa de Ambler.

També es coneix a la serialització com navegació. No és cap capa, sinó que en si és més una estratègia de programació per fer la correspondència de models entre les aplicacions i l'emmagatzemat dels objectes. Aquest sistema no treballa amb una base de dades sinó amb fitxers principalment. La seva principal avantatge és que treballa directament amb el model de dades de Java o qualsevol llenguatge amb el que es vulgui treballar. Escriu l'estat en què es troba un objecte i les seves relacions, en una cua de sortida o un sistema de fitxers. Això permet reconstruir completament els objectes i les seves relacions en quan vulguem i el nombre de vegades que vulguem. Aquest sistema es desenvolupa directament en el codi de l'aplicació. El problema és que no suporta fer transaccions, consultes, la compartició de dades entre molts usuaris a la vegada o funcions automàtiques d'integració de dades, donat que no treballa amb un sistema de base de dades. Totes aquestes funcions pròpies d'una base de dades les hem de suplir amb codi dintre de la pròpia aplicació. La serialització va ser implementada dins de les capes de persistència com sistema per tractar els objectes, i els models de dades.

La correspondència és una evolució de la serialització dels objectes, orientada cap a les bases de dades relacionals, enlloc de sistemes de fitxers. És en realitat un conjunt d'eines de traducció dels models de dades, fet que ens eviten que ho tinguem que fer directament al codi de l'aplicació. Aquest sistema és molt més elaborat i automatitzat que els anteriors. Ens estalvia una mica la complicació de la creació d'objectes a treballar des de l'aplicació, i solucionar el problema de la serialització al no permetre fer operacions pròpies de bases de

dades com les consultes. És un sistema molt laboriós d'implementar i suposa un esforç extra si volem fer actualitzacions del codi de l'aplicació, ja que hem de tornar ha crear l'objecte, amb l'eina corresponent. És un sistema poc dinàmic. A més sol ser un sistema propietari que no treballa amb totes les possibilitats, només ho fa amb les pròpies del fabricant. De correspondències O/R n'hi ha de molts tipus, entre elles cal anomenar: la taula per classe concreta, la taula per subclasse, i la taula per classe jeràrquica. Algunes d'aquestes correspondències O/R són estratègies que més tard apliquen diferents capes de persistència que inclòs les combinen. Al igual que algunes capes de persistència és comuniquen amb la base de dades a traves de JDBC. Només es fa servir en aplicacions senzilles o sistemes informàtics incrustats, com PDAs i mòbils, amb poc suport de bases de dades efectives.

2.2.3 Sistemes de bases de dades

- ODBMS:

També coneguts com bases de dades d'objectes. Varen començar la seva vida cap a principi del 90. Potser no tenen cabuda com capa de persistència, però són un altre sistema de persistència dels objectes, tot i que són una base de dades en si. La seva importància encara és relativa donat que li queda molt per poder arribar a ser un sistema ben desenvolupat. Encara no treballen amb tots models de dades dels diferents llenguatges O.O, cosa que fa que de vegades requereixin capes de persistència simplement per posar d'acord els models de dades. Podrien arribar a eliminar la necessitat de les capes. Si eliminessin les capes de persistència segurament milloraria el rendiment del conjunt aplicació OO- base de dades, a l'eliminar un intermediari. Però el problema és que són sistemes que no han estat gaire implantats pel moment, poc estandarditzats i que tenen alguns aspectes per solucionar encara, com poden ser els sistemes de consulta. Aprendre el seu funcionament tot i que són semblants a altres, és un esforç afegit per la persona que vol deixar de treballar amb RDBMS cap a ODBMS.

- XDBMS:

És una especificació de base de dades de la que es comença a parlar. El seu objecte de treball és l'XML. Poden arribar a eliminar la necessitat de les ODBMS, ja que permeten treballar amb descripcions d'objectes, i amb registres, a la vegada. Aquest potser el futur de la persistència dels objectes.

2.2.4 Capes de persistència:

Les capes de persistència podríem dir que són l'últim esglaó dintre de l'evolució dels sistemes de persistència. En aquest punt és una recopilació representativa de les diferents capes de persistència que es poden trobar. Entre les que es descriuen podem trobar tant especificacions de capes de persistència com productes basats en les característiques de Ambler, com capes que no compleixen cap d'aquests aspectes.

- ODMG:

Les seves sigles són Object Data Management Group. Cal aclarir abans d'explicar ODMG, que no és en si una capa de persistència, sinó una especificació, que alguns fabricants de capes de persistència han adquirit com referent. En si és un estàndard que compleix moltes de les característiques de Ambler. La seva especificació (actualment la versió 3.0) va ser creada per un consorci d'empreses i gent interessada en el desenvolupament d'ODBMS i correspondència entre objectes RDBMS a finals de 1991. Va ser un primer intent de realitzar una API estandarditzada per accedir a les bases de dades relacionals de forma transparent des d'aplicacions OO. Més tard ha anat evolucionat cap altres necessitats i permet treballar amb

les ODBMS. Va ser l'estàndard precursor de JDO. Treballa amb un model de dades propi, el model d'objectes OML. Aquest model es basa en els models d'algunes bases de dades i llenguatges OO, i permet afegir objectes per adaptar-se a les necessitats en qualsevol circumstancia o requeriment. ODMG és a més un sistema de múltiples plataformes, per la seva pròpia natura, i treballa amb els llenguatges C++, Smalltalk i Java. El seu principal problema és que va precedir en el temps a Java, i estava més orientat cap a llenguatges com C++ o Smalltalk. Això ha fet que no s'adaptes gairebé a Java, principalment en no treballar amb JNDI, ni entorns d'arquitectura de dues capes o tres. Comparativament ODMG respecte JDO, tenint en compte que JDO agafa com referència ODMG, aquest és més avançat. JDO té per inconvenient que només treballa amb un únic llenguatge : Java, però ha evolucionat més enllà. Entre les capes de persistència que tenen com referència ODMG cal destacar, OBJ de Apache Foundation, FastObjects de Poet,

- ADO:

ADO (Active Data Objects) és una API d'alt nivell creada per Microsoft a l'hivern de 1996. La seva funció principal és donar un accés a les dades emmagatzemades en un ampli ventall de bases de dades. La primera versió només treballava amb bases de dades relacionals, amb un model de dades molt prefixat. A partir de la versió 2.5 va començar treballar amb altres sistemes d'emmagatzemat. A partir de la versió 2.6 va poder treballar amb l'estàndard XML. Amb la seva versió ADO.Net intentarà solucionar alguns dels seus problemes i anar més enllà en la persistència dels objectes. ADO és sobretot un model d'accés a bases de dades orientat a objectes, i permet també treballar amb models de bases de dades anteriors, com relacionals. Això gràcies sobretot a què treballa sobre OLE DB i ODBC. El seu gran problema és que es tracta d'un sistema propietari de Microsoft, i a més que es basa en dos sistemes d'accés com ODBC i OLE DB, en comptes de treballar per ell mateix. Els seus inconvenients a nivell de programació són, la seva falta de seguretat i la complexitat en la seva aplicació en programació.

- CMP (arquitectura EJB):

L'arquitectura de components EJB (Enterprise Java Beans), ha estat dissenyada per treballar en servidor d'aplicacions Java, i que per tant, permet la distribució d'aplicacions. EJB proporciona dos tipus de components per desenvolupar aplicacions distribuïdes. Entre la seva arquitectura d'EJB trobem un suport de persistència dels objectes, CMP (Container Managed Persistence), cap a bases de dades. Aquest suport permet administrar la persistència de cada component definit a EJB. Aquest fet és degut gràcies a què fa servir l'API JCA, una interfície adaptadora de recursos, que també forma part d'EJB. Tota la gestió dels objectes és molt avançada, permet múltiples connexions a les dades i sistemes de caches, gràcies a la pròpia natura de distribució de components d'EJB. També permet gestionar rollbacks en operacions de la base de dades, degut al seu sistema de cache que treballa amb les operacions i les dades. El sistema pot treballar sobre múltiples plataformes al treballar sobre Java. JDO pot ser implementat com eina dintre de l'arquitectura EJB. Això potser interessant depenent de les necessitats de l'aplicació. La relació amb EJB/CMP no és del tot contraposada, sinó que a vegades pot ser complementaria, en benefici de l'aplicació. El problema que trobem és la complexitat del treball i els requeriments i recursos que necessita per treballar.

- JDO:

És l'API hereva natural d'ODMG, la serialització i la correspondència O/R. Es troba totalment orientada al llenguatge Java. Treballa la seva correspondència amb un arxiu XML. A més al basar-se en el model Java és un sistema de plataforma múltiple, i es pot aplicar a tot un

tipus de sistemes, com sistemes incrustats, o sistemes de caràcter empresarial, al igual que ODMG.

La JCP (Java Community Process), un organisme d'estandardització depenent de Sun Microsystems, la va començar a desenvolupar com especificació model , la número JSR-000012, amb la intenció de resoldre els problemes que no va poder solucionar l'API ODMG amb Java, a més d'afegir-li algunes avantatges més respecte els altres sistemes de persistència dels objectes. Craig Russell va ser l'encarregat de JCP per realitzar l'especificació que dones pas la capa de persistència JDO. JPC es va formar a l'estiu de 1999. I la publicació del primer esborrany del document va ser publicat al Maig del 2000. L'especificació va aparèixer al Maig del 2001 i va ser aprovada com definitiva al Maig del 2002. Aquesta va ser l'anomenada versió 1.0 de l'especificació. La segona i última especificació és una revisió de l'anterior i s'anomena 1.0.1. Dintre de poc temps, i està prevista la publicació de l'especificació 2.0, i ja es comença a parlar del que suposarà en articles d'experts publicats a internet. Però que és JDO? Robin Roos al seu llibre "Java Data Objects" fa la següent descripció de la capa de persistència JDO :

" Java Data Objects és una interfície basada en la definició dels objectes persistents per al llenguatge Java, a partir del qual descrivim l'emmagatzemat, les consultes i la recuperació dels objectes amb origen als magatzems de dades.

JDO esta obligat a respectar, en tots els sentits el concepte de persistència transparent, ja que:

- JDO manega de forma transparent, la correspondència de les instàncies JDO cap al magatzem de dades; la mal anomenada unió objecte- relacional.
- JDO és transparent als objectes de Java persistits. No es tenen perquè afegir ni mètodes ni atributs a les classes o alterar els modificadors de visibilitat. Els camps encapsulats amb visibilitat privada, i sense mètodes get i set, no són un problema per JDO, aquest adapta els objectes amb tots els elements necessaris per la persistència.
- JDO es pot fer servir contra un gran nombre de sistemes d'emmagatzemat incloent bases de dades relacionals, bases de dades d'objectes, sistemes d'arxius, i documents XML. Amb el temps les implementacions de JDO poden emergir per accedir a persistència sostinguda en aplicacions heretades, per fer servir en projectes d'EAI (Enterprise Application Integration).
- JDO és transparent al seu propi magatzem de dades, així les aplicacions poden ser portades a qualsevol sistema d'emmagatzemat per al que tinguem una implementació JDO apropiada. La compatibilitat binària de les instàncies JDO, garantida per l'especificació de JDO, significa que poden ser arxivades sense recopilació de successos, deixant qualsevol modificació a nivell de codi.
- Si una aplicació referència un objecte persistent i modifica el seu estat en la memòria , la implementació de JDO implícitament actualitzarà les dades en el magatzem quan la transacció activa sigui confirmada. Això permet al programador de manera repetida programar operacions d'emmagatzemar."

El principal problema amb el que es troba ara mateix JDO, és que es una tecnologia jove, i encara Sun no l'ha implantat dins Java, tot i que ha anunciat que ho farà en futures versions. Això li pot donar una empenta per consolidar-se com una tecnologia preferent per molts programadors. Les avantatges que comporta JDO, són sobretot degudes a la definició de

transparència ,com deia Roos, fet que facilita la seva implementació, en diferents situacions i entorns. Permet que un programador treballi amb un sol model de dades, el de Java. No requereix de grans recursos com EJB i no té cap necessitat extra per la seva implementació, estalviant a qualsevol programador entre un 20 i un 30% de codi escrit. Permet a més múltiples connexions, i una gran adaptació a qualsevol base de dades. Com podem veure compleix molts dels requeriments de Ambler, i per tant fa que sigui una capa de persistència com cal, en tots els aspectes.

Donada les seves avantatges d'implantació i treball, és una tecnologia que podria consolidar-se ben aviat com una norma. Estudiarem JDO més profundament en els següents capítols.

Capítol 3:Estudi d'una capa de persistència: JDO

Hem escollit JDO com el sistema de persistència ha estudiar, ja que creiem que és la millor opció de totes les estudiades i la que millor s'adapta a les característiques requerides per la persistència d'objectes. JDO és un sistema de persistència estandarditzat gràcies a la seva especificació en forma de capa o API. Compleix molts dels requeriments de Ambler (Capítols 1 i 2), i recull moltes de les característiques beneficioses d'altres sistemes, i va més enllà, com es pot comprovar en el fet de que permet qualsevol sistema d'emmagatzemat i d'entorn de treball. A més s'adapta perfectament a qualsevol model de les bases de dades i respecta per complert el de Java al permetre totes les seves característiques. Escollir JDO, ens ha permetre definir i conèixer millor, que és la persistència dels objectes, i implementar-la fàcilment de forma pràctica.

Un altre factor a favor de JDO, com referent d'estudi, és la gran quantitat de recursos que hi podem trobar, així com d'implementacions. Tot això segurament degut a què és una de les poques capes de persistència que parteix d'una especificació que fa que sigui un estàndard dins de la indústria, i que a més podria arribar a formar part del propi llenguatge de programació Java de Sun Microsystems. Totes aquestes raons fan que JDO amb el temps es pugui consolidar com un referent dins de les capes de persistència, i fets com la creació d'una segona especificació, fan que se li pugui endevinar un futur molt prometedor.

En aquest capítol aprofundirem en el funcionament intern i tècnic de JDO com capa de persistència, podrem conèixer les seves avantatges i els seus inconvenients. En els següents capítols desenvoluparem un exemple que implementarà alguns dels aspectes definits i permetrà aprofundir en l'arquitectura de JDO i la seva aplicació.

3.1 Característiques generals i conceptes introductoris:

3.1.1 Característiques generals:

L'especificació de JDO es marca com principals objectius la transparència i la integració de les bases de dades amb servidors d'aplicacions, com EJB. Això es degut a què com ja hem dit abans, intenta resoldre alguns dels problemes amb els que es va trobar ODMG sobretot amb les diferents arquitectures client/servidor, i recollir les avantatges que ja aportava, com la transparència.

JDO hi esta doncs orientat cap a dos entorns o escenaris, els més comuns que són els que anomenarem no- administrats i que fan referència, a un entorn client/servidor de primera generació on tractem les dades de manera local. I a un entorn, que anomenarem administrat, que treballa amb arquitectura client/servidor de dues capes, referint-nos sobretot a EJB.

3.1.2 Conceptes introductoris a JDO:

De forma introductòria enumerarem alguns dels conceptes que ens permetran entendre d'una forma més fàcil el funcionament de JDO. Primerament enumerarem alguns dels interfícies més comunes dins de JDO i que formen part del paquet desenvolupat per Sun Microsystems, javax.jdo.

1. **Instància JDO:**
   Conjunt d'objectes propis de l'aplicació que poden ser persistents, dins el magatzem de dades. Les instàncies JDO implementen la interfície PersistenceCapable, fet que permet que donar-li a JDO el caràcter de persistent a la instància.

2. **Implementació JDO:**
   Una implementació JDO és una col·lecció de classes, recopilades i desenvolupades per un grup de programadors o una empresa, que tenen com referent l'especificació de JDO. Els anomenarem venedors de JDO.

3. **JDO Enhacer:**
   És un programa que modifica el codi byte de les classes que defineixen els components de les aplicacions Java, amb l'objectiu de permetre la carrega i restauració de forma transparent de les dades de les instàncies persistents.

3.2 Arquitectura de JDO:

L'arquitectura de JDO, es basa en una sèrie de conceptes, quasi tots dirigits principalment a la gestió de les instàncies generades a partir dels objectes definits per les classes, i que són l'objecte sobre el que treballa JDO.A partir de les classes de l'aplicació i les instàncies que defineix aquesta, JDO genera el que anomenem instàncies JDO. Les instàncies JDO doncs representen tots els objectes que l'aplicació generà, i que són reflectits en la pròpia base de dades.

El primer concepte que introduïm en el nostre estudi és la distinció entre instàncies persistents o transitòries, i quines ha de tractar, per poder emmagatzemar-les en la base de dades. El segon serà com distingim les instàncies JDO. El tercer quins són els cicles de vida de les instàncies i el quart el model de l'objecte persistent, propi de JDO.

3.2.1 Classificació de les instàncies:

JDO fa dues classificacions a priori de les instàncies. La primera classificació distingeix si la instància és persistent o transitòria. La instància persistent és aquella que emmagatzemem en la base de dades i que podem recuperar en qualsevol moment. La instància transitòria és la que simplement no s'emmagatzema, però que és necessària dins dels processos de les aplicacions, i que alguns cops pot arribar a ser persistent. Els dos tipus d'instància que hem definit, tenen un cert caràcter dinàmic, ja que poden canviar la seva classificació o estat durant l'execució de l'aplicació. Una manera d'aconseguir aquest propòsit seria el que anomenem persistència per proximitat: una instància transitòria pot convertir-se en persistent per referència d'una instància que ja és persistent. Per un altre banda, una instància persistent pot canviar la seva classificació i ser transitòria, conservant la seva informació dins de la base de dades, sense afectar-li las modificacions que fem a partir del canvi d'estat.

La segona distinció que fa JDO d'una instància, de forma diferenciada de l'anterior classificació, és entre transaccional o no. Una instància és transaccional quan es troba emmagatzemada a la cache de JDO esperant la confirmació de la transacció amb la base de dades Queda molt clar quan una instància no és transaccional, quan no es troba en la cache esperant la seva modificació en la base de dades.

Les dues classificacions es poden combinar, donant una classificació més general, en la que es distingeixen 4 tipus d'instàncies. L'especificació del JCP marca que l'únic dels 4 tipus que té caràcter obligatori per a una compilació de JDO per part d'un fabricant d'eines de programació, és la persistent transaccional. Com podem entendre, una instància persistent transaccional és una instància que es troba tant a la cache de JDO com en la base de dades. L'especificació, dona caràcter opcional per qualsevol implementació de JDO, a las instàncies persistents no transaccionals i les transitòries transaccionals, ja que s'entén que una instància transitòria no transaccional és qualsevol instància Java de l'aplicació, i que mai serà objecte de persistència.

Les instàncies transitòries transaccionals, són instàncies que no representen cap objecte emmagatzemat a la base de dades de forma directe, sinó que són una ajuda a les instàncies que són modificades, o una referència a algunes de les seves dades. Poden arribar ha esdevindré persistents. Com que són transaccionals, es veuen afectades per les transaccions amb la base de dades, però no poden ser mai emmagatzemades a la base de dades. En el cas que la implementació de JDO que escollida no les contemplés, podríem fer servir una sincronització de dades per simular-les. Les instàncies persistents no transaccionals són aquelles instàncies que no tenim a la cache per fer la transacció en aquell moment, però que si trobem a la base de dades.

3.2.2 La identitat JDO:

Quan treballem amb Java, moltes vegades ho fem amb diferents instàncies d'una mateixa classe. Quan trobem dues instàncies d'aquest tipus, cal primerament distingir-les per poder treballar amb elles i així evitar confusions. Java distingeix entre dos tipus d'igualtat entre dues instàncies, l'equivalència i la igualtat en si. La primera ens la trobem quan les instàncies apunten al mateix objecte de la memòria, i la segona quan són objectes diferents en si però iguals en tots els aspectes. Però a JDO no li es suficient per poder distingir dues instàncies. Ja que podem trobar el mateix objecte definit en varies instàncies, de varies base de dades i en diferents transaccions. I això pot portar a una gran confusió en les dades. Per això JDO inclou un mecanisme propi de distinció d'instàncies, la identitat JDO, que parteix de la classe Object ID.

La identitat JDO, conserva el caràcter de persistència d'un objecte. Hi tenim diferents formes de conservar la identitat, depenent del que definim a la nostre aplicació. La primera forma que trobem es mitjançant una clau primària definida per l'aplicació i que la base de dades ha de respectar. Aquesta forma d'identificació és la que més sovint s'utilitza quan es treballa amb les bases de dades relacionals i és anomenada com identitat d'aplicació. Un altre forma d'identitat és la identitat de la base de dades, on és la pròpia base de dades qui dona el valor a la identitat. Aquest sistema és més propi per a base de dades per objectes, i s'anomena identitat de magatzem. L'ultima forma d'identitat és una identitat no administrada, que es compleix quan una taula d'una base de dades relacional no té una clau primària. Es anomenada identitat no durable.

En la identitat d'aplicació, com ja hem dit, és la mateixa aplicació la responsable de la identitat de la instància. La identitat es generada a partir dels valors de la classe a la que originen la instància, i genera una classe que direm clau primària. Aquesta classe és la que ens permet identificar al ser una referència, les diferents instàncies. La identitat dona un o més camps que ens serveixen de clau primària per a la classe, junt amb un nom per a la classe. La classe la generem sempre a partir de la classe Object ID. El programador de l'aplicació

usualment és qui escriu la classe de la clau primària, tot i que algunes eines tenen la capacitat de generar les classes que utilitzen aquest tipus d'identitat.

Trobem però algunes restriccions a l'hora de generar la classe de clau primària. La classe ha de tenir caràcter públic i ha d'implementar java.io.Serializtable . A més, la classe origen de la instància ha d'implementar un constructor no argumentat i un constructor de cadenes. El mètode toString() es sobrecarrega per a què la cadena que resultant es pugui fer servir com un argument, i poder crear una instància equivalent de la clau primària. Tots els camps no estàtics tenen que ser primitius, o referències de classes no serialitzables. Cada camp que forma la identitat d'una instància, ha de tenir correspondència amb un camp del mateix nom a la classe de clau primària. Tots aquests camps han de poder ser utilitzats pels mètodes de la classe de clau primària equals() i hashcode() per la determinació d'equivalència. Podrien haver camps addicionals a la classe de clau primària, però això es inusual i no tindrien implicació en el procés d'unicitat.

La identitat de la base de dades és el mecanisme d'identitat per defecte de JDO. La identitat es atribuïda a l'objecte que es fet persistent. La forma d'arxivar-la, i la natura de la classe Object ID, són interns per la implementació JDO i el magatzem de dades . Com sempre es determina una sola identitat i pot utilitzar-se en futures consultes per rebre un objecte en particular. La identitat de magatzem s'utilitza típicament amb objectes dependents.

La identitat no durable de JDO es fa servir amb objectes persistents on no té cap sentit diferenciar-los. La determinació i creació d'una clau al magatzem de dades por ser una operació molt repetitiva. La identitat no durable es fa servir sovint per ajudar a la ràpida persistència de noves instàncies.

3.3 El cicle de vida de les instàncies:

Aquest apartat podríem dir que forma part de l'arquitectura de JDO, però donada la seva capital importància per entendre el funcionament de JDO l'hem volgut destacar amb un apartat propi.

Fins ara només hem conegut alguns dels conceptes bàsics sobre els es fonamenta JDO. Ara veurem com treballen entre si les instàncies, i quins són els seus cicles de vida dintre de JDO. El cicle de vida d'una instància de JDO es defineix a partir una sèrie d'estats i transicions associades als mateixos estats. A JDO, la referència de cada instància JDO és l'únic mecanisme que coneix quin és l'estat de la instància. Si nosaltres volguéssim conèixer el seu estat, ho tindríem que fer mitjançant una representació de l'estat de la instància, amb els mètodes de la classe JDOHelper, però mai ho podrem fer de manera directe, al tractar-se del procés intern de JDO. Més tard aclarirem que és la classe JDOHelper. Els objectius que es marca l'especificació de JDO sobre el que vol aconseguir amb el cicle de vida de les instàncies són:

- La persistència ha de ser transparent per tot programador.
- Les instàncies de JDO han de poder fer-se servir de manera efectiva en diferents entorns.
- Múltiples JDO PersistenceManagers (en parlarem més tard) residents i amb la capacitat de compartir les diferents classes persistents.

L'especificació de JDO, divideix en dues categories els estats del cicle de vida, obligatoris i opcionals. En primer lloc ens centrarem en els obligatoris i les possibles transicions, per més tard parlar dels opcionals. Hem de tenir en compte que cada estat té com a entorn de treball la transacció en la que es troba. Com transacció entenem el mateix terme que s'utilitza en qualsevol base de dades, doncs treballem directament amb la base de dades.

3.3.1 Els estats obligatoris del cicle de vida:

1. **Transitori:**

Aquest és l'estat inicial de qualsevol instància. Quan es genera la instància, aquesta encara no té definició dins de l'entorn de persistència, i se la considera transitòria no transaccional. Com no forma part de l'entorn de persistència, no pot generar cap error.

2. **Persistència - nova:**

Aquest és l'estat de les instàncies que s'han fet persistents durant la transacció actual. Durant la transacció de transitori a persistent , el administrador de persistència associat compleix:

- Esdevé el responsable d'implementar l'interrogatori d'estats i els estats de transició.
- Conserva els valors dels camps persistents i transaccionals no-persistents, que es faran servir durant la restauració actualitzada.
- Assignar una identitat JDO a la instància.

3. **Persistència –nova - eliminada:**

Aquest estat se li assigna a les instàncies que primerament han estat transitòries i més tard eliminades. Si volem llegir qualsevol dels seus valors, obtindrem una excepció JDO, al no tenir permís per llegir-la.

4. **Buida:**

Les instàncies en aquest estat tenen alguns dels seus valors buits. Aquestes instàncies tenen carregada però la seva identitat JDO. Poden modificar el seu estat si omplim els camps, però no les podrem llegir mentrestant.

5. **Persistència neta:**

En aquest esta s'hi troben les instàncies que contenen dades persistents transaccionals en el magatzem de dades, i que els seus valors ha estat llegits en la transacció actual, sense ser modificats. En el cas que no fos llegida en la transacció actual voldria dir que es troba en estat Buida.

6. **Persistència bruta:**

Es dona quan els valors de la persistència neta han estat modificats.

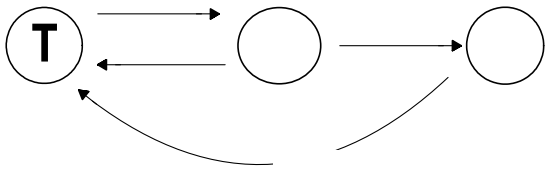
7. **Persistència – eliminada:**

En aquest estat tenim a les instàncies que es generen a partir del

magatzem de dades, enlloc de ser de nova creació, i que han estat eliminades, durant la present transició. Tenim premis per llegir la seva clau primària, però no els seus valors, ja que ens donaria una excepció dins de l'entorn de JDO.

3.3.2 Transicions dels cicles de vida obligatoris:

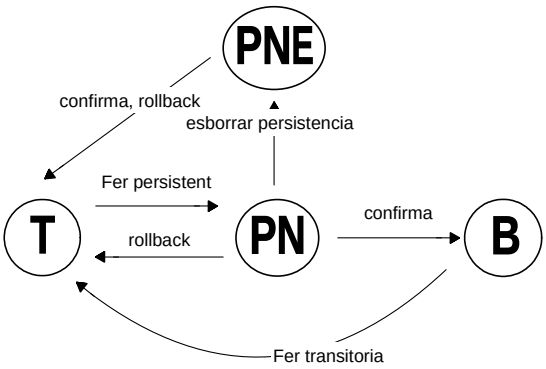
1. **Persistint una instància transitòria:**



Gràfic 3

Aquesta transició és fonamental dins de la persistència a JDO. Permet crear una nova entitat dins de la base de dades. Si l'entitat ja existeix o hi ha una d'igual, o ja tenim una clau primària igual, JDO ens donarà una excepció d'error, i no crearà l'entitat nova.

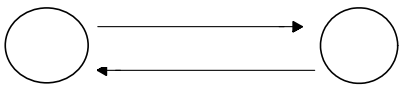
2. **Crear/esborrar una nova transacció:**



Gràfic 4

Aquest cas es dona quan tenim una instància en estat Persistència nova – esborrada. Quan una instància fa transició cap a transitòria pot tractar-se d'una representació d'una entitat de la base de dades.

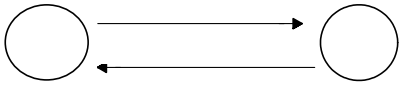
3. **Llegint els valors dels camps:**



Gràfic 5

Quan llegim per primer cop les valors d'una instància buida, aquesta fa transició cap l'estat de persistència – neta. Algunes vegades una transició persistent – neta pot fer la transició a l'inrevés.

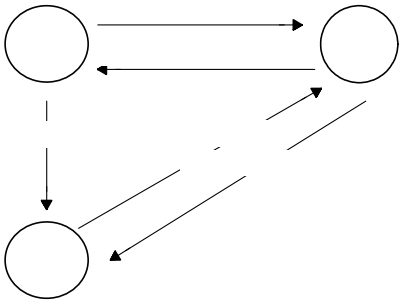
4. **Expulsió:**



Gràfic 6

Aquest estat es produeix quan l'aplicació, notifica que una instància transaccional deixarà de ser transaccional. Aquest estat ens dona optimització a la cache de JDO. No es aconsellable fer aquesta transició si tenim la intenció de tornar a utilitzar aviat la mateixa instància, ja que això va en detriment de la velocitat de l'aplicació.

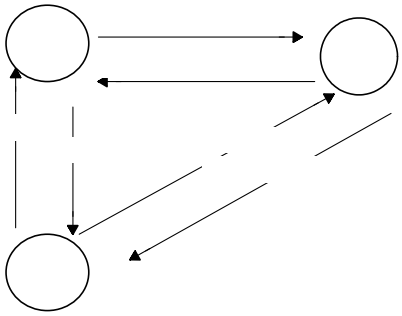
5. **Actualitzant el valor de les dades:**



Gràfic 7

Quan modifiquem un sol valor d'un camp d'una instància, aquesta fa la transició cap a Persistent – bruta. Com podem veure només ho pot fer si prèviament estava en estat de buida o persistent - neta.

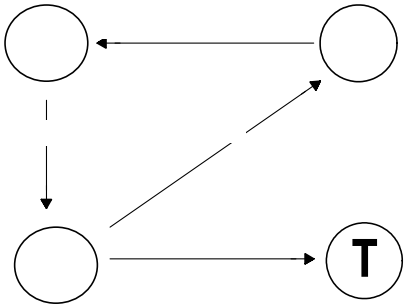
6. **Refrescant els valor de les dades:**



Gràfic 8

És una ampliació de la transició anterior. I es dona quan volem retornar els valors previs a la modificació dels camps, abans de fer la transacció.

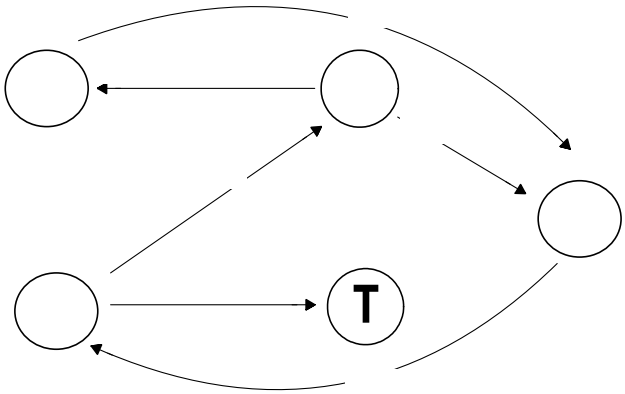
7. **Eliminant una instància persistent - neta:**



Gràfic 9

Quan una instància d'una transacció prèvia es eliminada, passa a l'estat Persistent – eliminada. Si l'eliminació es confirmada a la base de dades, la instància passa a ser transitòria, o sinó a buida.

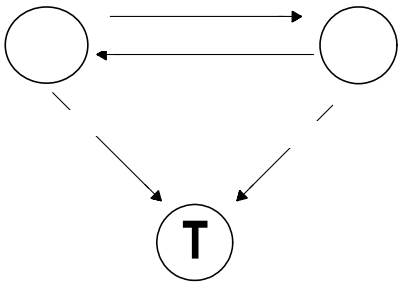
8. Eliminant una instància persistent – bruta:



Gràfic 10

Quan una instància persistent abans de la transacció actual, es modificada i esborrada en la mateixa transacció, aquesta passa de persistent – neta a bruta i més tard a eliminada. Si l'eliminació és confirmada a la base de dades, la instància passa a ser transitòria.

9. Fent una instància transitòria:



Gràfic 11

A vegades per absurd que sembli necessitarem que una instància torni a ser transitòria. Això pot passar durant el procés de correspondència o en un entorn administrat. Hem d'afegir, que les dades del magatzem, al

modificar l'estat de la instància, no elimina les seves dades. Aquestes poden tornar a generar una instància, impedint que la instància prèvia que ara és transitòria pugui tornar a ser persistent, al generar duplicitat. A més al passar a ser transitòria, perd la seva identitat JDO.

3.3.2 Els cicles de vida opcionals:

1. Persistent – no transaccional: Pnt

Aquest estat es troba associat als estats persistents ja descrits. Normalment una instància passa aquest estat, quan l'aplicació la col·loca fora de la cache de treball de JDO. Aquesta instància té caràcter persistent, i per tant la seva informació es troba a la base de dades.

2. Transitori – net:

Aquest estat i el següent es fan servir en la implementació de instàncies transaccionals transitòries. És una instància transitòria que no ha estat modificada en la transacció present.

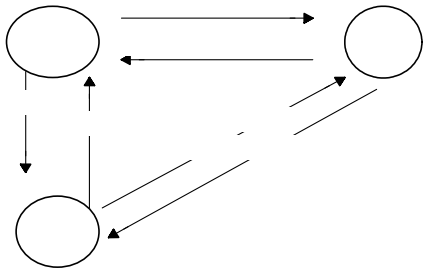
3. Transitori – brut:

Direm que la instància es troba en aquest estat si ha estat modificada durant la transacció actual.

3.3.4 Transicions dels cicles de vida opcionals:

En aquestes transicions treballem tant amb els estats opcionals com els obligatoris. Amb això completarem el cicle complert de vida de les instàncies.

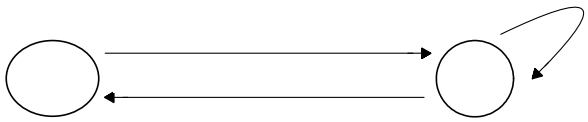
1. Transaccions optimistes:



Gràfic 12

La transició optimista es basa en l'estat persistent – no transitori. Aquest estat funciona d'enllaç entre els estats buida i persistent –brut, millorant la feina sobre una instància. Intenta avançar-se a la següent operació a realitzar sobre la instància. Per exemple, al voler llegir una instància buida, l'estat de la instància passa a ser persistent – no transitori, fet que no modifica cap dada de la base de dades. Si volem modificar les dades, l'estat canvia a persistent – brut, i espera la transacció.

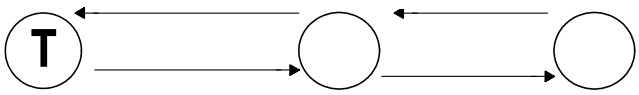
2. Transaccions exteriors a l'accés persistent:



Gràfic 13

Aquesta transició permet la lectura i escriptura de dades, fora del circuit de dades transaccional.

3. Accés transaccional a les instàncies transitòries:



Gràfic 14

Aquesta transició es dona quan una instància transitòria no transaccional passa a ser transaccional, i més tard fa transició cap a transitòria – neta. Si es modifica algun dels seus camps passa a transitòria – bruta. Un cop la transacció es realitza, la instància passa un altre cop a transitòria – neta, ja s[illegible] perquè recupera els valors anteriors, com que no.

Pnt Lect

3.4 El model de persistència dels objectes.

Amb aquest terme volem descriure un conjunt de regles que deriven dels tipus d'objectes permesos, les configuracions d'objectes administrats per JDO i la seva interrelació. El model de persistència dels objectes, s'assembla molt al model d'objectes de Java, ja que és d'aquest del que parteix com referent. Els principals objectius i intencions que es marca l'especificació de JDO amb el model de persistència dels objectes són:

- Proporcionar un suport el més semblant possible al que dona Java, incorporant els tipus de dades primitius i per referència, en les instàncies JDO.
- Incorporar els mateixos modificadors que Java, per a les diferents instàncies JDO. Alguns dels modificadors són private, public, final, etc..
- Permetre que les classes definides pel usuari siguin persistents.
- Permetre que algunes classes del sistema, sobretot les que permeten el modelatge de les aplicacions, puguin ser persistents.

3.4.1 Transparència:

Com ja hem indicat és l'objectiu principal de tota implementació de JDO. Per poder dur a terme aquest objectiu, la implementació ha d'aconseguir tant que no es perdi rendiment al fer la correspondència, com que aquesta sigui el més efectiva possible. Per tant s'ha d'encarregar tant de la gestió de la cache d'instàncies, com de les instàncies persistents, ja que les instàncies transitòries de l'aplicació les ha de gestionar la JVM (Maquina virtual de Java) per no perdre rendiment. També s'ha d'encarregar del mecanisme de correspondència entre els models de Java i la base de dades. La implementació de la correspondència però és de forma lliure ja que no està argumentada dins de l'especificació del JCP.

Fer no transa

Fer transac

La transparència doncs ha d'aconseguir que tota instància persistent que es troba en la memòria estigui disponible de forma immediata, gràcies a la gestió de caches, l'actualització implícita d'instàncies persistents brutes en el magatzem sobre la transacció confirmada, i la correspondència automàtica de tipus Java al tipus natiu del magatzem de dades.

3.4.2 Les instàncies JDO:

A les instàncies JDO ens podem trobar que els seus camps d'informació poden ser persistents o no, ja que JDO fa persistents els objectes camp a camp. Mitjançant el descriptor de persistència especifiquem quins camps, són persistents. Per defecte, els camps que poden ser persistents, són aquells que ja hem anomenant anteriorment com suportats per JDO. Cal tenir en compte de forma general, que no han d'incorporar els modificadors static, final o transient per poder ser persistents. Tot i que algunes d'aquestes restriccions depenen de la implementació JDO amb la que treballem. Aquesta restricció no afecta als camps no persistents, o que mai ho seran.

L'especificació de JDO dicta un conjunt de classes obligatòries que suporten la persistència i que les diferents implementacions han de suportar. També però proporciona d'opcionals. Algunes de tipus collection concretament no són suportades, tot i suportar la interfície, ja que JDO requereix concretament de hashSet, membre d'aquest conjunt de classes.

3.4.3 Classes d'objectes de primera i segona:

Un altre divisió que fa JDO de les instàncies i que formen el model de persistència d'objectes són, les instàncies en classe de primera i de segona, FCO i SCO respectivament.

Les FCO són instàncies de classes que implementen la interfície PersistenceCapable, i que per tant tenen identitat pròpia amb tot el que suposa, persistència directe, podem consultar-los i suporten unicitat. Com és una instància persistent, si un camp d'una FCO és modificat, l'objecte canvia d'estat, persistent – net a persistent- brut. Un altre característica d'una FCO és que pot ser referenciat per altres FCO a la vegada.

Una SCO és una instància d'una classe com FCO, però que no té identitat JDO, i per tant tampoc té suport d'unicitat. Només se emmagatzema com una part d'una FCO. Quan són modificades, no assumeixen l'estat de bruta per si mateixes, només quan la FCO de la que depenen esdevé bruta. En algunes implementacions JDO, s'encarreguen de representar les relacions entre instàncies.

La diferencia més significativa que trobem entre les FCO i les SCO és la visibilitat dels canvis fets pels objectes. Si ens trobem que en un objecte una FCO forma part d'un altre FCO, i aquesta l'última és modificada, la primera també. Amb les SCO succeeix el contrari, si l'objecte que té com part una FCO és una SCO, quan es modifica, no modifica la FCO. Això es causat perquè les SCO i les FCO tenen l'associació amb com relació. Com ja hem dit alguns SCO són al igual que els FCO, PersistenceCapable. La majoria però SCO són simplement classes Java amb suport de persistència específic, com podrien ser el cas dels persistents per referència. Mitjançant el descriptor de les FCO decidim si són FCO o SCO.

Un cas apart i que alguns autors, com Robin Roos les anomenen "tercera classe", són els arrays d'objectes. El suport per aquest tipus és opcional a nivell d'especificació. Si una implementació dona suport per arrays, aquest hauran de formar part d'un FCO dins del

magatzem de dades, mai per si sols. Els arrays d'objectes a més no tenen unicitat, al no tenir identitat JDO, al igual que els SCO.

Tant els FCO i els SCO tenen restriccions en la seva formació, si no treballen amb els tipus suportats per JDO. Però el conjunt en si, sobre tot als que hi fa referència, pot variar segons la implementació JDO amb la que treballem. Els tipus en general amb els que podem treballar i que per tant formen el model de dades, alguns de forma opcional, són :

- Primitius : boolean, byte, short, int, long, char, float i double.
- Objectes immutables: Boolean, Byte, Short, Integer, Long, Double , String, Character, Float, Locale, BigDecimal i BigInteger.
- Objectes mutables obligatoris per la implementació JDO: Date i HashSet.
- Objectes mutables opcionals per la implementació JDO: ArrayList, HashMap, HashTable, LinkedList, TreeMap, TreeSet i Vector.
- Objecte per referència : Object
- Interfícies de Col·leccions obligatòries: Collection i Set,
- Interfícies de Col·leccions opcionals: Map i List
- La resta d'interfícies Java que no siguin de col·leccions i els arrays .

3.4.4 Herència:

Al igual que qualsevol objecte Java, aquests es poden relacionar amb altres objectes a partir de l'herència. El suport que se li dona a l'herència és bastant flexible, ja que en
els estats del cicle de vida no poden produir herència. Ens podem trobar que una classe que no és persistent, té subclasses persistents. El mateix succeeix amb les transaccions. L'herència de classes és independent de la persistència. Aquest fet és donat perquè és en el mateix descriptor de persistència on definim el caràcter de la classe, i aquesta definició sempre es fa de manera concreta per a cada classe, i no per herència.

3.5 Classes i interfícies del paquet JDO:

3.5.1 Introducció al paquet JDO:

JDO es troba format bàsicament per interfícies, i algunes classes. És sobretot a partir de les interfícies que les aplicacions poden accedir a la funcionalitat de persistència dels seus objectes. El paquet de JDO es troba dividit en dues parts. Una part és l'API de JDO, javax.jdo, i l'altre és la part interna de JDO o SPI, javax.jdo.spi.

Tant per generar implementar JDO en una aplicació, com per crear una nova implementació de JDO, els fabricants o programadors han de conèixer a fons el paquet JDO i els seus components. Inclòs si volgués fer una nova implementació del paquet, ha de tenir l'especificació del JCP per complir un mínim de normes per que es consideri una API JDO.

A mesura que descrivim les interfícies i les classes, explicarem el seu caràcter i la seva funció dins de l'estructura de JDO. Per tal de no fer llarga l'explicació, hem resumit alguns dels aspectes dels components de JDO.

3.5.2 PersistenceCapable:

Totes les instàncies que pot administrat JDO i que per tant han de tenir la capacitat de ser persistents en qualsevol moment, han originar-se a partir classes en les que s'ha implementat la interfície pública PersistenceCapable. Aquesta interfície forma part del paquet SPI. La interfície implementa en les classes, els mètodes necessaris per l'administració de les seves instàncies per part de la infrastructura de persistència, en temps d'execució. Tots aquests mètodes es caracteritzen perquè els seus noms comencen sempre pel prefix "jdo", per tal d'evitar possibles conflictes amb mètodes ja existents. Ens permet conèixer aspectes de les instàncies en temps d'execució, com per exemple en quin estat del cicle de vida es troben, conèixer a quin PersistenceManager estan associats, etc..

En principi el programador no té perquè fer servir aquesta interfície de manera directe en les seves classes. Aquesta treballa a traves del PersistenceManager, i de JDOHelper, tal i com ara veurem, degut al seu caràcter intern en JDO.

3.5.3 PersistenceManagerFactory:

Aquesta interfície és capital en el funcionament correcte de qualsevol implementació de JDO. Forma part de l'API de JDO, i la trobem a javax.jdo. És la responsable de generar les instàncies PersistenceManager per a l'aplicació amb la que s'està treballant. L'ha de proporcionar el programador de la implementació JDO i no és de caràcter obligatori en entorns no administrats. El problema que trobaríem però és que l'aplicació perdria rendiment, agilitat de funcionament i opcions de treball.

Proporciona els mètodes per accessos, modificacions i administració de les PersistenceManager (get, set). Tots els mètodes poden ser cridats des de JDOHelper i com ja hem explicat, permeten des de generar diferents instàncies de PersistenceManagerFactory, configurar les propietats, etc.. (veure JDOHelper).

3.5.4 PersistenceManager:

És la interfície primària de JDO, on es fonamenta la persistència dels objectes. S'encarrega de moltes de les funcions de JDO, des l'administració dels estats de les instàncies, les diferents caches, les identitats JDO, i les consultes a la base de dades. Ens permet fer persistents les classes que vulguem o ens facin falta, tant pròpies com no. Cada instància de PersistenceManager té la capacitat d'administrar múltiples instàncies a la vegada. Es troba fortament lligada a PersistenceManagerFactory, ja que aquesta s'encarrega d'administrar cadascuna de les PersistenceManager que l'aplicació generi durant l'execució.

L'arquitectura de la PersistenceManager permet que puguem canviar la PersistenceManager per una altre, d'un fabricant diferent, si compleix les regles de portabilitat. A més ens permet que treballem sense modificar el codi de l'aplicació, amb els diferents entorns, ja que s'encarrega de l'adaptació de JDO a l'entorn. Per gestionar una sola instància tenim sempre Object pc com paràmetre dels mètodes, i per a col·leccions d'instàncies o arrays, Collection c o Object [] o. Els mètodes per grups varien en afegir *All* al final de la denominació del mètode, per exemple *makePersistentAll* per *makePersistent.* En cas que no sigui així els indicarem amb negreta. Passem a descriure els mètodes que incorpora i la defineixen millor:

1. Administració de la cache:

1.1. Expulsar:

En principi, l'expulsió de la cache es realitza de forma automàtica durant les transaccions, però per alguns casos interessa que sigui manual. Depèn de les banderes RestoreValues i RetainValues, configurades a la PersistenceManagerFactory que l'administra. A més crida a jdoPreClear si tenim implementades les InstanceCallBacks. El mètode és, *evict* per tal d'expulsa les instàncies persistents de la cache.

1.2. Refrescar:

Permet recompondre els valors de les instàncies de la cache com al començar la transacció. Cadascuna cridarà a jdoPostLoad si tenim les InstanceCallback implementades. El mètode és *refresh* per refrescar una instància de la cache.

1.3. Recuperar:

Aquest grup de mètodes permeten modificar el permís de lectura des de la base de dades, dels camps d'una instància. Cadascuna cridarà a jdoPostLoad si tenim les InstanceCallback implementades. Cridem *retrieve* per tal de modificar el permís. En tots els casos del mètode, és optatiu incorporar com paràmetre DFGOnly, propi de l'especificació 1.0.1, i que indica que només els camps donats amb el grup per defecte, són necessaris de recuperar, sinó es rebrem tots.

2. Cicle de vida d'una instància:

Aquests mètodes permeten gestionar el cicle de vida de les instàncies. Enumerem:

2.1 Fer persistent una instància transitòria : *makePersistent.*

2.2 Eliminar les instàncies persistents: *deletePersistent.*

Aquests grup de mètodes, permet eliminar instàncies persistents de la base de dades. Només es poden trucar quan tinguem una transacció activa. Quan finalitza la transacció, les instàncies passaran a ser transitòries.

2.3 Fer transitòria una instància persistent: *makeTransient.*

2.4 Fer transaccional una instància no transaccional: *makeTransactional.*

2.5 Fer no transaccional una instància transaccional: *makeNonTransactional.*

3. Administrar les identitats JDO:

PersistenceManager incorpora una sèrie de mètodes per poder administrar les identitats JDO de les instàncies. getObjectId, getTransaccionalObjectId, *getObjectIdClass* ***(Class pcClass)*** i *newObjectIdInstance* ***(Class cls, String representation)*** .

4. Funcions administratives:

Aquest mètodes permeten administrar la persistència dins l'aplicació. Trobem: *close()*,*isClosed()*,*setUserObject*, *getUserObject()*, *getPersistenceManagerFactory()*, *currentTransaction ()*, *setMultithreaded* ***(boolean Multithreaded)*** i *getMultithreaded()*.

5.Obtenir instàncies:

L'extensió d'una classe representa el conjunt complert d'instàncies

persistents de la classe. Per gestionar-la tenim *getExtent* ***(Class cl, boolean subclasses)*** que retorna l'extensió d'una classe i *getObjectbyId que* recull una instància persistent de la base de dades, a partir del identificador Object ID del objecte donat.

6. **Fabricació de consultes:**
   NewQuery() que genera una consulta nova. (aquest tema l'aplicarem més tard al apartat de consultes.)

3.5.5 JDOHelper:

Ja hem parlat abans de JDOHelper. És una classe de l'API de JDO, o sigui que no té caràcter intern, que conté mètodes estàtics i que es fa servir per a les classes informatives de persistència. Cadascun del seus mètodes té com objecte interrogar a les instàncies persistents, o sigui que implementen PersistenceCapable. Podem dividir els seus mètodes amb interrogadors d'estats, administratius i els anomenats per Robin Roos, d'arrancada flexible.

1. **Interrogadors:**
   Serveixen per conèixer en quin estat es troben les instàncies dins del seu cicle de vida. Trobem, *isDirty* per l'estat bruta, *isTransaccional* per l'estat transaccional, *isPersistent* per l'estat persistent, *isNew* per l'estat nova i *isDeleted* per l'estat eliminada. Tots aquests mètodes tenen com paràmetre *(Object pc).*

2. **Administratius:**
   Ens permeten administrar les instàncies generades per l'aplicació. Els mètodes són: *getPersistenceManager(Object pc): PersistenceManager* , que ens dona el PersistenceManager a la instància, *getObjectId (Object pc)* , que ens permet conèixer l'identificador de la instància, *getTransaccionalObjectId (Object pc)* : que ens referencia la identitat actual tenint en compte les modificacions fetes a la clau primari durant la transacció, i per últim *makeDirty (Object pc, String fieldname)* , que marca el camp donat com brut, d'aquesta manera el seu valor serà modificat a la base de dades quan es confirmi la transacció. Opcionalment podem incloure amb el nom del camp, el paquet i la classe a la que pertany el camp. Ens torna false si la instància no té capacitat per ser persistent, o si és transitòria.

3. **Arrancada flexible:**
   Aquest mètodes en permeten obtenir PersistenceManagerFactory per la nostra Aplicació. Són *getPersistenceManagerFactory (Properties p)* que ens retornarà un PersistenceManagerFactory conforme les propietats donades com paràmetre i *getPersistenceManagerFactory (Properties p, ClassLoader cl)* que varia de l'anterior en que hem d'especificar el carregador de la classe que volem.

4. **Opcions sobre els mètodes d'arrancada flexible:**
   Les propietats de configuració de les PersistenceManagerFactorys defineixen com volem que es comporti JDO en referència a la base de dades i l'aplicació. Tenim:

   - javax.jdo.PersistenceManagerFactoryClass: que referència el paquet que conté les classes de la implementació JDO.

   - javax.jdo.option.Optimistic: que permet per part de la PersistenceManagerFactory fer l'administració de forma optimista. És una bandera.

- javax.jdo.option.RetainValues: permet l'expulsió automàtica de les instàncies persistents quan la transacció ha finalitzat l'eliminació. És una bandera.

- javax.jdo.option.RestoreValues: permet que les instàncies recuperin el seu valor anterior a una transacció si l'usuari ho demanda. És una bandera.

- javax.jdo.option.IgnoreCache: fa que les consultes JDOQL puguin ser resoltes sense referir-se als canvis de les instàncies presents en la cache i encara no sincronitzades amb el magatzem de dades. És una bandera.

- javax.jdo.option.NonTransaccionalRead: és una bandera. Provoca que les instàncies persistents puguin ser llegides des de fora de les transaccions.
- javax.jdo.option.NonTransacionalWrite: és una bandera. En aquesta opció les instàncies no transaccionals poden modificar-se des de fora de la transacció.

- javax.jdo.option.Multithreaded: permet gestionar la sincronització de tots els enllaços amb l'administrador de persistència. És una bandera.

- javax.jdo.option.ConnectionDriverName: recull el nom del controlador de la base de dades amb la que treballem. Requereix de les opcions següents, per poder accedir a la base de dades.

- javax.jdo.option.ConnectionUserName: recull el nom del usuari que pot treballar amb la base de dades.

- javax.jdo.option.ConnectionPassword: recull la contrasenya del usuari.

- javax.jdo.option.ConnectionURL: recull la direcció de la base de dades.

3.5.4 JDOImplHelper:

És una de les classes que ens ha d'ajudar a la implementació de JDO al igual que l'anterior. Conté un registre de metadades per cada classe. Pertany a la part interna de JDO. S'encarrega d'enregistrar les classes PersistenceCapable en un arxiu de metadades durant la carrega d'aquestes. Els seus mètodes ens permet evitar l'ús de la reflexió en l'execució de l'aplicació.

JDO disposa de varis mètodes per treballar amb aquesta classe, els dividim en:

1. **D'accés de la classe:** *getInstance()* .
2. **Accés a metadades:** *getFieldNames (Class pcClass), getFieldTypes (Class pcClass), getFieldFlags (Class pcClass)* i *getPersistenceCapableSuperclass (Class pcClass).*

3. **Fabricació d'instàncies amb capacitat de ser persistents:** *newInstance (Class pcClass, StateManager sm*) i *newInstance (Class pcClass, StateManager sm, Object oid).*

4. **Enregistrament de classes amb capacitat de ser persistents:** *registerClass*

(Class pcClass, String [] fieldNames, Class [] fieldTypes, byte[] fieldFlags, Class persistenceCapableSuperclass, PersistenceCapable pcInstance).

5. **Notificació dels registres de classes amb capacitat de ser persistents:** *addRegisterClassListener (RegisterClassListener rcl)* i *removeRegisterClassListener (RegisterClassListener rcl).*

6. **Administració de la seguretat:** *registerAuthorizedStateManagerClass (Class smClass)* i *checkAuthorizedStateManager (StateManager sm).*

7. **Administració de les identitats de l'aplicació:** *newObjectIdInstance (Class pcClass), newObjectIdInstance (Class pcClass, String str), copyKeysFieldsToObjectId (Class pcClass, PersistenceCapable .ObjectIdFieldSupplier fs, Object oid) i copyKeysFieldsFromObjectId (Class pcClass, PersistenceCapable .ObjectIdFieldConsumer fc, Object oid).*

3.5.7 InstanceCallbacks:

És una interfície que conté només 4 mètodes. Es troba a javax.jdo, i per tant no forma part del nucli intern de JDO. Ens informa dels esdeveniments que afecten a les instàncies durant el seu cicle de vida i així ens permet prendre possibles accions al respecte. Els mètodes són:

- *jdoPostload()* : aquest mètode es cridat en la instància desprès que el grup de camps per defecte siguin carregats.

- *jdoPreStore()* : aquest mètode es cridat immediatament abans que es sincronitzin els valors de la instància amb la base de dades.

- *jdoPreClear()* : aquest mètode es cridat abans que les valors d'una instància siguin esborrats, i la instància canviï el seu estat a buida.

- *jdoPreDeleted()* : aquest mètode es cridat abans que la instància canviï el seu estat cap a persistent – deleted o persistent – new – deleted.

3.6 Transaccions:

Les podem definir com les operacions sobre les instàncies persistents, que normalment modifiquen les seves dades. El seu referent en si són les mateixes transaccions de la base de dades. Tot i que de fet podem dir que són el mateix, ja que quan fem una transacció sobre una instància, estem fent la mateixa transacció sobre la base de dades. JDBC treballa amb 4 nivells aïllats de transaccions amb qualsevol base de dades: Read Uncommited (llegir sense confirmar) , Read commited (llegir confirmant), Repeatable Read (lectura repetitiva) i Read serialitzable (lectura serialitzable). JDO no especifica de manera explicita, sobre quin nivell i treballa, ja que això no li permetria treballar amb diferents magatzems de dades a la vegada i fer operacions al mateix temps. Però podríem dir que JDO treballa amb Read Commited, de manera general.

Mitjançant el mètode currentTransaction () de la PersistenceManager, definim el procés que permet fer les transaccions del que s'encarregarà l'administrador de persistència. Mentre l'administrador de persistència existeixi, es podran realitzar varies transaccions en si, una

darrera de l'altre. Si volguessin treballar amb varies transaccions a la vegada, necessitarem tants administradors de persistència com transaccions. Cal dir que JDO no admet de moment les transaccions incrustades com les bases de dades.

JDO, permet dues estratègies d'administració de transaccions, l'anomenada pessimista o emmagatzemant de dades, i l'optimista. La primera és obligada per l'especificació de JDO, i la segona és opcional per l'especificació.

L'administració pessimista, té un comportament òptim, per transaccions de vida curta i automatitzades, ja que requereix l'exclusivitat en l'accés a les dades, i si tinguéssim moltes transaccions de llarga vida, repercutiria en el rendiment general.

Quan tenim transaccions de llarga durada, normalment no es demana l'exclusivitat. Aquestes intenten treballar amb les menys exclusivitats possibles, per poder millorar els rendiments. Aquesta política es basa en una idea optimista, normalment quan unes dades són modificades per una transacció, cap altre pot modificar les dades mentre la primera no hagi estat confirmada.

Les transaccions presenten una sèrie d'opcions, moltes de les quals ja hem introduït de forma indirecta, en forma de banderes. *NonTransactionalRead* , que dona pas a què es pugui llegir les instàncies persistents fora de la transacció activa. *NonTransactionalWrite* que permet que es pugui escriure en les instàncies persistents fora de la transacció activa, *RestoreValues* que modifica el tractament de les transaccions de les dades, com les operacions de restauració (rollback), *RetainValues*: que modifica el comportament de les transaccions. Les instàncies desprès d'una transacció o canvien a buides, o es queden en estat persistent no transaccional, mentre no siguin transaccionals o expulsades.

3.7 Consultes:

JDO té 3 formes de fer les consultes a JDO. El primer és a partir del mètode getObjectbyId() de l'administrador de persistència, i l'Object ID de la instància. El segon amb iteracions sobre l'extensió d'una classe PersistenceCapable. I l'últim a partir del llenguatge de consultes natiu a JDO, JDOQL i la interfície que la implementa, Query.

3.7.1 JDOQL:

JDOQL és un llenguatge independent de la base de dades, que es basa en la sintaxi de Java, per realitzar les consultes sobre les instàncies treballant directament amb la base de dades com si fos el llenguatge SQL. És el que anomenen alguns autors llenguatge neutral. JDOQL ens permet fer consultes sobre sistemes de fitxers i documents XML. Hem de recordar que aquest era un dels punts febles d'altres sistemes de persistència com ODMG i la serialització.

Les consultes treballen principalment sobre col·leccions. Recopilen les instàncies segons el criteri donat en una col·lecció, que és el resultat de la consulta, és el que anomenem col·lecció candidata. Les consultes permeten valors amb paràmetres, al treballar directament en Java. Els elements que formen principalment una consulta són: la col·lecció de classes candidates, la col·lecció candidata i el filtre de consulta. De forma addicional poden formar una consulta les declaracions de paràmetres, els valors de paràmetres, declaracions de variables, importacions

de declaracions i criteris d'ordenació de la consulta. JDOQL no permet repeticions de noms de cap dels seus elements dins de la mateixa consulta.

3.7.2 Construcció de consultes:

La construcció de les consultes es basa en el mètode sobrecarregat *newQuery*. Els paràmetres que permet podem donar-li són varis com, una altre consulta, una consulta d'un altre llenguatge, amb una classe candidata i una col·lecció candidata, amb una classe candidata, una col·lecció candidata i un filtre de les instàncies, amb l'extensió de la classe candidata , amb l'extensió de la classe candidata i el filtre d'instàncies i una classe candidata i un filtre.

3.7.3 La interfície Query:

Com ja hem dit abans, la interfície Query és l'altre component de les consultes. Aquest interfície es troba implementada a la PersistenceManager, que com ja hem explicat s'encarrega de fer les consultes. Com es pot deduir pel que hem estudiat, una PersistenceManager pot tenir múltiples instàncies de consultes. Alguns dels mètodes que implementa la interfície Query, i que permeten administrar les consultes són: *getPersistenceManager()* que ens indica el PersistenceManager al que la consulta esta associat, *setClass()* que configura la classe candidata per la consulta, *setCandidates (Collection candidatecollection)* que configura la col·lecció candidata, *setFilter (string filter)* que configura el filtre, *declareVariables(string variables)* que configura les variables, *declareParametres (string parametres)* que configura els paràmetres, *setOrdening (string ordening)* que configura l'ordenació, *compile()* que compila les consultes, corregeix els errors sintàctics de les expressions, i millora el rendiment, *execute()* que executa la consulta amb fins a tres paràmetres i *close (Object queryResult)* que tanca els recursos associats a un resultat de la consulta.

3.7.4. Els filtres de les consultes:

Són opcionals segons l'especificació de JDO. Ja que si no tenim un filtre, aquest no pot avaluar les resultats i la consulta donarà sempre com resultat la col·lecció candidata. Els filtres treballen sobre expressions booleanes, i fan servir els operadors propis. La correspondència de símbols i operadors és la següent:

No	i	i condicional	o	o condicional
!	**&**	**&&**	**|**	**||**

Taula 2

Aquests operadors es fonamenten en la lògica matemàtica, afegim els operadors comparatius, típics: == , !=, >, <, >=, <= +, * i /. (Els últims corresponen a la suma, multiplicació i la divisió); operadors de referència, "(Class) bastidor de l'objecte" i "· referència un camp tipus" i operadors de singularitat," ~ complement integral", i "- inversió numèrica". Incorporem als filtres la paraula "this", pròpia de Java, per referir-nos a la instància de la col·lecció candidata que el filtre avalua.

Tot i que JDOQL esta dissenyat per poder fer consultes des de Java, substituint SQL, aquest llenguatge i presenta una sèrie de diferencies amb Java, en els mètodes d'invocació, igualtat,

ordenació, assignació i navegació ja que són diferents dels que utilitza Java, com ara ho comprovarem :

1. **Mètodes d'invocació:** JDOQL no es pot fer invocacions a mètodes. Només pot invocar *isEmpty(), contains(Object o), startsWith (String s)* i *endsWith (String s)*, sent opcionals *indexOf()*, i *Lower()*. Els dos primers comproven si no es troba o no una instància persistent, i els dos següents si troba una i comença o acaba amb el paràmetre donat.

2. **igualtat:** A diferencia de Java es pot fer servir == entre tipus primitius i ajustats. Ex. int a == integer b.

3. **Ordenació:** Com la igualtat, la resta d'operadors d'ordenació es poden fer servir entre tipus primitius i ajustats.

4. **Assignar:** Un filtre no pot modificar el valor d'un camp persistent. Treballa sobre el valor com referència.

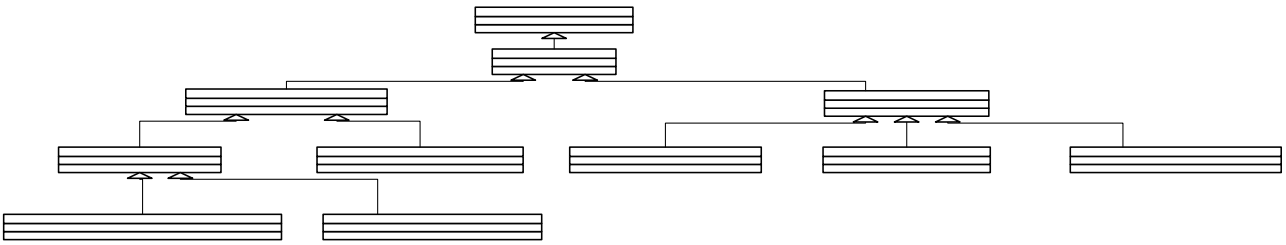
5. **Navegació:** Per poder navegar, no es fan servir els operadors de referència i singularitat, apart del mètode contains().

3.8 Excepcions JDO:

JDO presenta una sèrie d'excepcions que responen als possibles errors que ens podem trobar al implementar-lo i que no obeeixen als models de dades ni de Java ni de la base de dades, només de JDO.

3.8.1 La jerarquia d'excepcions de JDO:

Les categories en què es divideixen les excepcions són, les provocades per l'usuari al invocar els serveis de JDO, per la base de dades implementada, o pel propi JDO.



Gràfic 15

3.8.2 Classes bàsiques de les excepcions:

JDOException: És la classe base de totes les excepcions de JDO, i és a la vegada una extensió d'Exception del propi Java. És excepció la pare de:

1. ***JDOFatalException:*** És la base de les excepcions que no poden ser revisades de codi. Impliquen normalment que en una transacció, ha estat restaurada i deuria de ser abandonada per l'aplicació.

2. ***JDOCanRetryException:*** És la base de les excepcions que poden ser revisades de codi.

3. JDOUserOptionException: Són les que es poden revisar causades per l'usuari. Com una clau primària igual a una que ja existeix, referenciar una instància que no existeix o al recollir una extensió d'una classe que JDO no administra.

3. ***JDOUnsupportedOptionException:*** Es produeix quan volem fer servir una característica opcional de JDO que la implementació no suporta.

4. ***JDOFatalUserException:*** Són les excepcions d'usuari que no tenen solució.

5. ***JDOFatalInternalException:*** Són pròpies de la implementació JDO del venedor o fabricant.

6. ***JDODataStoreException:*** Es poden arreglar i són pròpies de l'emmagatzemat de dades.

7. ***JDODataFatalStoreException:*** No tenen solució i fan referència a l'emmagatzemat de dades.

3.9 El descriptor de persistència. Breu introducció a XML.

El descriptor de persistència és l'última part en si pròpia de JDO que ens queda descriure, i que podríem dir que és la més important. Com hem pogut veure, es fa servir en el temps d'implementació per identificar les classes que poden ser persistents. Aquesta informació es fa servir per la implementació en plena execució, i alguns fabricants el fan servir per la informació de correspondència. Com ara veurem el descriptor és un simple document XML:

3.9.1 Breu introducció a XML orientada a JDO:

El llenguatge extensible de marques, XML, és un format estàndard d'estructuració de dades. XML és un llenguatge molt versàtil que es pot programar des de qualsevol editor de text. És independent de la plataforma treball, té compatibilitat binària i gran portabilitat, i per tant és un complement perfecte de Java. Com complement de JDO, el llenguatge XML permet emmagatzemar les dades de configuració de l'aplicació, les instàncies persistents, etc.. en l'anomenat descriptor de persistència DTD, que és el que facilitarà la comunicació entre la base de dades relacional i l'aplicació orientada a Objectes.

Un document XML, permet dades estructurades per etiquetes o tags. En el cas del DTD les tags que inclou són *<jdo>*, *<package>* i *<class>*. Totes elles permeten la definició de

sentencies JDO. XML compleix unes regles de formació, al igual que tots els llenguatges de programació, i per tant quan obrim una tag, l'hem de tancar, per fer això hem d'etiquetar *<\jdo>*. Tota la informació que vulguem incorporar dins la tag s'ha de que situar entre l'obertura i el tancament. XML, permet tags incrustades. Al igual que qualsevol llenguatge es permeten atributs, per exemple: *< class name = "client">* Contingut del tag *</class>*. On "client" és l'atribut que defineix el nom de la classe. Podem escriure tags buit, fent: *<class name ="client"/>*. Per que el DTD tingui un format coherent, definirem una gramàtica pròpia.

En el DTD ens podem trobar problemes de nombre respecte atributs o tags. Indicarem aquest fet de la següent manera, un "*" representa zero, un o molts, i un "+" per un o molts. Si diem que DTD conté de zero a molts camps i extensions ho farem de la següent manera:

< !ELEMENT class (field| extension)>*
<!ATLIST class name CDATA #REQUIRED>

El DTD és un arxiu anomenat jdo.dtd. Cada descriptor de persistència es referència al DTD amb una directiva DOCTYPE, descrivint on es troba.Un descriptor de persistència sol contindré per si mateix la descripció d'una o varies classes, se li dona el mateix nom de la classe o paquet que conté. L'extensió que es dona de forma opcional, és *.jdo, ja que indica la seva procedència.

Aquests són en principi els trets bàsics que ens permetran entendre millor com es genera el DTD i com funciona. En els següents apartats anirem endinsant-nos cada cop més en el DTD.

3.9.2 Descriptor de persistència:

Les principals tags que formen un descriptor de persistència són:

1. ***<extension>*** : descriu les extensions pròpies dels fabricants, dintre del descriptor, perquè es puguin fer servir en el procés de persistència. Permet 3 atributs: vendor – name, key i value.

2. ***<jdo>*** : Aquest tag és obligatori present, ja que descriu el lloc on es troba el document. No té atributs, però dintre del seu contingut és on es defineixen els paquets i les extensions.

3. ***<package>*** : defineix un paquet. Té només per atribut el del seu nom. Conté com elements les classes que formen els paquet, i extensions necessàries per les classes.

4. ***<class>*** : defineix les classes. Tenen 4 atributs. *Name*: que dona el nom a la classe. *Identity-type*: que descriu la identitat JDO, sent sempre datastore per defecte. *Objectid-class*: només s'especifica en les classe amb identitat d'aplicació. I *Requires-extent*: especifica si l'extensió de la classe la pot administrar JDO.

5. ***<field>*** : Descriu cadascun dels camps i les seves característiques pròpies dintre de l'entorn de la persistència. Té els següents atributs: *name, persistence-modifier, primary- key, null-value, default- fetch-group,* i *embedded.*

6. ***<collection>*** : Aquest element es fa servir per identificar els camps col·leccions. Té dos atributs, *element-type* i e*mbedded-element*.

7. ***<map>*** : Aquest element permet treballar amb mapes com a camp. Té quatre atributs, *key-type, value-type, embedded-key* i *embedded-value*

.

8. ***<array>*** : Permet definir com camp el tipus array. Té un sol atribut. *Embedded-element*.

3.10 Arquitectures de dos capes:

Com ja hem explicat al llarg del capítol, JDO permet ser implementat en el que hem anomenat entorn administrat, o arquitectura de dues capes. Farem una petita introducció sense profunditzar gaire, en com treballa JDO amb aquesta arquitectura.Com es conegut, J2EE, és una especificació d'un servidor d'aplicacions. Aquest es compon principalment de dues capes, la web i l'EJB. Els elements de la primera són de tipus HTML, i els de la segona tenen com components objectes java. El tractament que fa de JDO, J2EE és la serialització, ja que es comunica el servidor d'aplicacions els objectes al seus clients, serialitzant-los. JDO, implementa ell mateix la serialització de les instàncies persistents i transitòries. Això facilita al servidor d'aplicacions el treball, i permet que JDO només calgui ser present en el servidor. JDO es sincronitza les seves transaccions amb les de J2EE per poder treballar amb els diferents objectes, com si es tractes d'una base de dades. Com podem veure JDO treballa de la mateixa manera en una arquitectura com en l'altra, gràcies sobretot a la seva transparència.

Capítol 4: Introducció a diferents implementacions de JDO

Fins ara hem estudiat com tindria que ser el funcionament de JDO segons l'especificació 1.0.1 de JDO. Podríem dir que és la part teòrica del funcionament de JDO. Però, realment JDO funciona de forma pràctica tal com indica la part teòrica?. Per comprovar el funcionament real de JDO i considerar-la com una alternativa real a altres sistemes de persistència, estudiarem les algunes de les implementacions més importants de JDO que han estat generades per diversos fabricants i programadors. Entre les diferents implementacions fem una divisió general, entre implementacions gratuïtes i les de pagaments, que anomenarem aquí com comercials o no.

Entre les versions comercials estudiarem com més destacades: Kodo JDO de Solarmetric, Lido de Liberis, OpenFusion de PrismTechnologies, FastObjects de Poet software, FrontierSuite d'ObjectFrontier, i JCredo de Riflexo. Tot i això per no allargar l'estudi cal destacar altres que no inclourem com : IntelliBo de SignSoft, JDO Genie de Hemisphere Technologies, JRelay d'Object Industries, Orient d'Orient Technologies, Pe:J de HYWY Software, Exadel JDO d'Exadel, i RexIP de TradeCity CiberSoft, entre altres.

De les versions no comercials estudiarem Triactive JDO (TJDO) de SourceForge i OBJ d'Apache Software Foundation. Cal però destacar també, JORM d'ObjectWeb, XORM de SourceForge, relacionada amb TJDO.

Hem trobat moltes més implementacions de JDO, però que la majoria segueixen especificacions pròpies dels fabricants, i entraríem en una dinàmica d'estudi diferent. Entre aquestes però cal destacar Castor d'Exolab.

Tot i això cal destacar un fet comú de totes les implementacions, i és que sempre són un grup de llibreries o APIs al voltant de les llibreries de l'especificació definida per JCP, jdo.jar i jdori.jar, i moltes d'elles no les inclouen en el seu paquet, sinó que les hem d'aconseguir nosaltres mateixos de la web del JCP.

4.1 Enumeració de fabricants comercials :

4.1.1 FastObjects de Poet software:

Poet, és un fabricant que desenvolupa sobretot bases de dades incrustades, i eines de treball sobre aquest tipus de bases de dades. La seva implementació, FastObjects no és purament JDO, ja que incorpora també ODMG per poder treballar amb llenguatge C++. FastObjects, té com última versió la 9.5., i implementa l'especificació 1.01, en la part de JDO. Pot treballar amb moltes de les bases de dades del mercat, a més de les pròpies del fabricant. Però aquesta implementació, no és un producte homogeni, ja que es divideix en diferents implementacions. Depenent de l'escenari amb el que treballem s'ha d'implementar una de diferent: j2 per un sol usuari, base de dades local i programació Java, e7 per igual que l'anterior però orientat a C++, i e7 per múltiples usuaris, múltiples bases i tots dos llenguatges. Les seves característiques generals són :

- Implementa ODMG i JDO en una mateix API.

- Implementa l'especificació 1.01.
- Afegeix a JDO, un llenguatge propi de consulta a més de JDOQL, filtre d'objectes.
- Encriptació de les dades a la base de dades.
- Pot treballar amb entorns administrats.
- Permet escalabilitat.
- Optimitzacions pròpies pel rendiment del conjunt.
- Permet integració UML, pel desenvolupament.
- Pot incorporar-se com opció a diferents entorns de desenvolupament, com JBuiler 6 de Borland.

4.1.2 FrontierSuite per JDO d'ObjectFrontier:

ObjectFrontier és una empresa que dedicada a desenvolupar aplicacions d'entorn administrat. Aquesta implementació, que treballa sobre la versió 3.1, es composa d'una sèrie d'eines que permeten entre altres aspectes implementar JDO en aplicacions ja existents, o treballar a partir d'una base de dades existent. L'especificació JDO que implementa és la 1.0 amb totes les seves característiques. Incorpora sistemes de caches pròpies i personalitzades en els processos. Permet treballar amb diferents entorns de desenvolupament com JBuilder, Eclipse, i Forte. Permet integració UML, pel desenvolupament. El principal inconvenient que presenta és que només pot treballar amb les bases de dades següents, sense què el programador la pugui adaptar de forma personalitzada: Oracle, MS SQL, DB2, PointBase, Cloudscape, MS Acces, SAPDB, Hypersonic SQL i Sybase. I els entorns administrats: Weblogic, WebSphere, Orbix E2A J2EE server, Jboss, Oracle 9iAS, Hp AS, Orion AS i JOnAS.

4.1.3 Kodo JDO de SolarMetric:

SolarMetric és una empresa que té com principal objectiu desenvolupar productes Java orientats a web i xarxa local. Col·labora amb la companyia Versant en el desenvolupament del seu producte Kodo JDO 3.0. Es potser l'eina més avançada en la implementació de JDO, ja que a més de poder treballar amb RDBMS i ODBMS, incorporar algunes de les funcions de la futura especificació 2.0, a més de totes les declarades opcionals en anteriors especificacions. És l'eina més complerta en aquest aspecte. Permet a més treballar amb bases de dades ja existents, i automatitza el treball amb la base de dades d'una aplicació ja existent. Permet treballar amb les bases de dades que implementin l'API JDBC. Té la majoria de les bases de dades del mercat certificades, però permet que el programador adapti JDO a qualsevol base de dades. Implementa la majoria de sistemes de correspondència coneguts, i pot treballar amb sistemes d'implementació d'entorn administrat. Com noves característiques que implementa de l'especificació 2.0 són poder implementar expressions SQL a més de JDOQL. Permet treballar amb més tipus de dades, millora la transferència d'objectes, i altres més encara no especificades de forma definitiva que tindrà l'especificació 2.0. Pot treballar amb varis sistemes de bases de dades diferents a la vegada, en una mateixa aplicació. Permet ésser incorporat a la majoria d'entorns de desenvolupament com Eclipse, Sun ONE, Forte, JBuilder, etc..

4.1.4 LiDO de LIBeLis:

La empresa Libelis, fabricant de Lido, es una de les empreses membre del JCP.
És una empresa dedicada a sistemes de transaccions de dades. La seva implementació és un conjunt d'eines i productes com Navilis, Lido standard edition, etc.. que permeten la

implementació de JDO. La seva implementació es basa en l'especificació 1.0 del JCP. Esta dirigida principalment cap a bases de dades relacionals: Oracle, DB2, Informix, Sybase, MS SQL, PointBase, Cloudscape, InstantDB, MySQL, PostGres i Interbase; bases de dades d'objectes: Versant; i sistemes d'arxius i XML. Permet a més treballar amb entorns administrats, com Weblogic, Websphere, JBoss, Borland AS, JOnAs, en resum qualsevol entorn que treballi amb el connector JCA de J2EE. Entre els productes, a més dels anomenats cal diferenciar: enterprise edition, dirigit a empreses i és el producte més complert, profesional edition, també dirigit cap a empreses i al treball sobre entorns administrats, i la comunity edition, dirigida a la investigació i la docència. Cal afegir que Lido proporciona de forma personalitzada sota demanda del programador els controladors de la bases de dades amb la que treballarà. Aquest és un gran inconvenient, ja que dificulta el desenvolupament en múltiples bases a la vegada. Actualment hi treballen en una nova versió, la 2.0, encara que sobre l'especificació 1.0, amb noves eines com la correspondència reversible.

4.1.5 OpenFusion JDO de PrimsTechnologies:

L'empresa PrismTechnologies és una empresa sistemes d'integració d'entorns administrats i serveis web. Actualment és membre del JCP. La seva implementació, la versió 1.1, esta basada en l'especificació 1.0, i treballa orientada a RDBMS. Permet la seva integració en diferents plataformes com Solaris, Windows, etc.. OpenFusion, afegeix algunes característiques pròpies a JDO en aspectes com l'automatització de la correspondència, semàntica pròpia en l'eliminació d'objectes, etc.. Treballa principalment amb les bases de dades: HSQL, DB2, Informix, Oracle, MS SQL, MySQL i Sybase. I no permet l'adaptació, de moment. Permet també treballar amb arxius XML. Treballa amb tots els entorns administrats que incorporin la compilació 1.3 de J2EE.

4.1.6 JCREDO de Riflexo :

Riflexo és una companya europea que treballa en el desenvolupament d'eines Java. La seva implementació segueix la futura especificació 2.0 del JCP. I és doncs de les implementacions més avançades, tot i ser molt jove. És una implementació molt complerta, que es pot incorporar a diferents entorns de desenvolupament com JBuilder i Eclipse. A més d'incorporar sistemes propis i eines que segons Riflexo millora el rendiment intern de JDO. Té certificades les bases de dades Oracle, MySQL, MS SQL, DB2, PostgreSQL, PointBase, InterBase, Hypersonic SQL, i Sybase, tot i que pot adaptar-se a diferents bases de dades. Té certificats els entorns administrats, Weblogic, Borland, Websphere, JBoss, JRun, Oracle 9iAS, SunOne iPlanet, EaServer, Tomcat.

4.2 Enumeració de fabricants gratuïts :

4.2.1 TJDO de SourceForge:

També conegut com TriActive JDO, és un projecte de codi obert. Treballa a partir de l'especificació JDO 1.0.1, i la compleix en quasi tots els seus aspectes. A més permet generar sentencies SQL directament i vistes SQL, fet propi de l'especificació 2.0. Treballa amb diferents bases de dades com Oracle 8i, Cloudscape, DB2, FireBird, MySQL, PostgreSQL, SAP DB, i MS SQL. Encara no ha incorporat suport per entorns administrats, en la seva última versió 2.0.

4.2.2 OJB d'Apache Software Foundation:

Projecte membre entre altres del projecte Apache Foundation. Aquesta capa de persistència esta formada per les capes ODMG 3.0 i JDO 1.0. Permet treballar amb JDBC, i per tant en principi admet qualsevol base de dades relacional amb API JDBC propi. La seva implementació però no segueix l'especificació 1.0 de forma complerta, tot i utilitzar les llibreries natives de JDO 1.0. Moltes de les característiques de JDO, com la correspondència, són comunes amb ODMG en aquesta API. Permet la integració en entorns administrats EJB. La seva versió 1.0 encara no ha estat implementada, i encara es troba en procés de construcció beta. Apache té previst incorporar de forma total l'especificació de JDO en la versió 2.0.

4.3 Comparatiu d'implementacions JDO:

Característiques	FastObjects	FrontierSuite	Kodo	Lido	OpenFusion	JCredo	TJDO	OJB
Especificació	1.0.1	1.0	2.0	1.0	1.0	2.0	1.0.1	1.0
RDBMS	Si	Si	Si	Si	Si	Si	Si	Si
ODBMS	No	No	Si	Si	No	No	No	No
Opcions especificació	Si	Si	Si	Algunes	Algunes	Algunes	Algunes	No
Entorn Administrat	Depenent de la implementació	Si	Si	Si	Si	Si	No	Si
Adaptable a entorns de programació	No	Si	Si	Si	No	Si	No	No
Correspondència reversible	No	Si	Si	No	No	Si	No	Si
Versió d'avaluació	No manté tot	Complerta	Complerta	No manté tot	No	Complerta	Si	Si
Opcions pròpies	Si	Si	Si	Si	Si	Si	No	Si
MultiBase	Si	Si	Si	Depenent del fabricant	Si	Si	Si	Si

Taula 3

Capítol 5: Estudi d'una implementació JDO: KODO 3.0 de SolarMetric.

5.1 Perquè KODO 3.0 ? :

És possible que observant tota la mostra d'implementacions diferents que hem estudiat per sobre, calgui destacar la creada per Solarmetric i Versant, Kodo 3.0. És la implementació més avançada en tots els aspectes, ja que hi treballa amb algunes de les futures característiques de la futura especificació 2.0 encara en procés de discussió. Pot semblar que és una mica arriscat parlar d'una especificació que encara no ha vist la llum, però sembla ser que algunes d'aquestes noves característiques han estat introduïdes pel JCP com alguns dels aspectes a millorar a partir de l'especificació 1.0.1 que aquí hem estudiat. Aprofitarem l'estudi més a fons de Kodo 3.0 per poder introduir aquestes noves característiques de la futura especificació. Però aquest no es l'aspecte principal pel que he escollit Kodo, sinó que a més és la implementació més d'acord a l'especificació 1.0.1 i això ens permetrà estudiar bé a nivell pràctic l'objecte d'aquest document, l'especificació 1.0.1. Un altre aspecte és la seva versió d'avaluació que es complerta en tots els aspectes i ens permetrà fer l'estudi de forma complerta.

5.2 Característiques principals:

Ja hem fet una introducció general dels principals trets de Kodo 3.0. En aquest apartat estudiarem de forma més profunda les seves característiques:

5.2.1 La especificació 2.0:

Com ja hem destacat abans un del principals trets de Kodo 3.0 es l'adaptació de noves característiques de l'especificació 2.0. Però quines són aquests nous trets? Ara passem a anomenar-los de forma general:

- Crear JDO/R, el sistema de correspondència i quedava en mans del fabricant de la implementació, i es vol fer una correspondència estandarditzada.
- Millorar el llenguatge de consulta ampliant opcions com anomenar les consultes, i permeten incloure sentencies SQL.
- Ampliar la gamma d'implementacions, permetent la persistència també d'interfícies.
- Permetre desconnectar i tornar a connectar amb la base de dades, treballant des de la cache, i actualitzant les dades que es vulguin. Aquest tret és sobretot orientat a entorns web.
- Millorar la facilitat d'ús.
- Millorar l'adaptació a entorns administrats.

Molts d'aquests aspectes que en principi esta discutint el JCP, Kodo ja les ha adaptat d'acord amb el que ja ha estat acordat com definitiu, i per exemple ja implementa funcions com l'anomenat de les consultes, la desconnexió i la torna amb la base de dades aplicant un sistema de caches, i una millora general en la facilitat d'ús, i la correspondència i l'adaptació a

entorns administrats. Encara li queden alguns dels trets no aclarits com JDO/R i la persistència d'interfícies.

5.2.2 Especificacions anteriors:

En el seu conjunt les especificacions 1.0 i 1.0.1 són implementades fidelment, fet que ens permet estudiar de forma complerta l'especificació hem estudiat, la 1.0.1. Kodo implementa tant les opcions generals com les que l'especificació anomena com opcionals, i per tant ens permet veure el funcionament de molts dels aspectes i fets anomenats en capítols anteriors. Un dels fets destacats en l'especificació, és la filosofia que JDO pugui treballar amb totes les bases de dades existents de forma fàcil. En aquest aspecte, a nivell pràctic això no es compleix del tot, ja que JDO depèn totalment dels API JDBC i JCA, depenent de l'entorn. Kodo afegeix eines per poder fer aquest aspecte el més aproximat possible al que expressa l'especificació. En l'estudi del desenvolupament d'una aplicació amb Kodo 3.0 anirem anomenant alguns dels aspectes de l'especificació que Kodo implementa i veurem com ho fa.

5.2.3 Eines i característiques pròpies:

Un dels aspectes que destaca Kodo, és en la idea de facilitar la feina de JDO. Si un dels aspectes que hem destacat de JDO es que ens permet fer persistència d'objectes de forma fàcil, no es compleix, JDO té els dies comptats vers altres sistemes existents o futurs. Solarmetric doncs s'ha pres aquesta política molt seriosament i per això ha intentat adaptar JDO per facilitar l'ús mitjançant les seves característiques pròpies i eines molt útils. Un primer aspecte a destacar és l'adaptació a entorns de programació Java ben coneguts com poden ser Ant, Xdoclet, JBuilder, Sun One o Eclipse. Tots aquests entorns permeten a l'usuari afegir noves eines externes, i Kodo implementa la seva eina per cadascun d'aquests entorns.

Un segon aspecte és l'adaptació a les diferents bases de dades del mercat. El problema que trobem principalment no és pas el controlador JDBC de connexió amb la base de dades, ja que Kodo permet donar totes les opcions possibles a JDBC, sinó més aviat l'adaptació dels models de dades de les dues parts. En principi Kodo treballa amb les diferents bases de dades fent servir una adaptació dels models de dades, a partir del que anomena diccionaris. L'únic problema que ens podem trobar es que abans de treballar amb la nostra base de dades, si aquesta no és de les anomenades suportades, tinguem que indicar quines són les característiques de la nostra base de dades per què Kodo generi el nostre diccionari. Tot i això no es garanteix un funcionament 100% amb la nostre base de dades, ja que aquesta pot tenir opcions que Kodo no contempli en el llistat de propietats del diccionari. Les anomenades suportades són: DB2, Hypersonic, Informix, MS Access, MS SQL, MS Foxpro, MySQL, Oracle, PointBase, PostgreSQL, Sybase, i JDataStore. Un diccionari que esta en procés de creació i certificació és el de l'ODBMS Versant.

Com ja hem dit Kodo incorpora moltes eines de treball, cal destacar entre altres *Schematool*, *Mappingtool*, *Metadatatool*, *Schemagen*, *remotemanagementtool*, *reversemappingtool*,*dbsequencefactory*, *dbschemafactory*, *dbmappingfactory*, i *appidtool*. Principalment aquestes eines s'adapten a l'entorn de programació amb el que treballem, però també és possible treballar amb elles a partir de la consola, al igual que Java. Les funcions d'algunes de les eines són les següents:

- *Mappingtool*: permet fer la correspondència de les classes a partir d'un arxiu de metadades .jdo (el DTD) on llistem les classes que ha correspondre i algunes de les seves característiques a tenir en compte, o un arxiu .class amb la classe o un arxiu .java. Afegeix també altres opcions secundaries, totes relacionades amb la correspondència.
- *Schematool*: forma part de *mappingtool*. Manté actualitzada la base de dades a partir del l'esquema. L'esquema és l'arxiu XML que genera mappingtool a partir del qual genera les sentencies SQL del model de dades de la base de dades.
- *Schemagen*: forma part de mappingtool. Genera l'esquema.
- *appidtool*: permet que Kodo treballi amb identitat d'aplicació. Ja que per defecte treballa amb la de base de dades.
- *Reversemappingtool*: permet fer la correspondència de forma inversa, de la base de dades a la classe. Forma part de *mappingtool* ja que és important mantenir una correspondència perfecte en tots dos sentits.

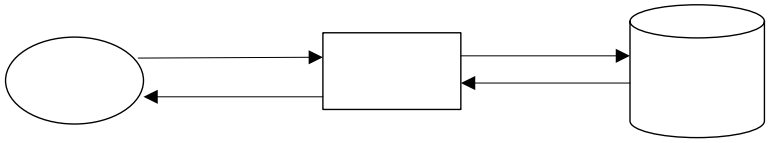
Altres aspectes que té Kodo són: permetre fer registres del procés de correspondència, fet que ens permet millorar la nostra aplicació vers la persistència. Un servei d'administració i escolta dels rendiment de l'aplicació, inclòs de forma remota. Sistemes de millora de rendiment de l'aplicació: com caches, agrupacions de dades en els procés d'intercanvi de dades aplicació- base de dades. Extensió del model de consulta de JDOQL. Consultes directes SQL.I filtres d'objectes per a les consultes. El funcionament general de Kodo 3.0 el veurem en el capítol següent de desenvolupament de l'exemple i l'annexa A.

Capítol 6: Exemple d'aplicació pràctica del JDO

6.1 Plantejament general:

Fins ara hem vist com funcionen teòricament JDO, algunes de les implementacions més importants de JDO, però encara no hem vist la forma pràctica d'implementar JDO al programar una nova aplicació. Això és el que veurem en els dos últims capítols.

El plantejament inicial que fem doncs és realitzar una aplicació en Java que treballi amb una base de dades fen servir JDO. De la única cosa que ens hem de preocupar és de fer les classes Java que implementin l'aplicació, i indicar a JDO quina és la base de dades amb la que treballarem. En el nostre cas serà la coneguda base de dades Oracle en la seva versió 9i. Tot i que és una base de dades que pot implementar classes Java, la manera com ho fa i es relaciona amb una aplicació encara es molt complexa i justifica del tot que treballem amb JDO. El cas que treballem però amb Oracle no és del tot important, ja que podríem haver fer servir qualsevol base de dades, però veure el funcionament de JDO que és el nostre principal objectiu. Per poder realitzar aquest exemple doncs hem de tenir instal·lada la base de dades en un ordinador local o en xarxa. Un altre fet a tindré en compte és la implementació JDO que farem servir, que en el nostre cas és Kodo 3.0 en la seva versió d'avaluació. En aquest esquema podem veure el plantejament general de com ha de funcionar l'exemple:



Gràfic 16

Hem d'aconseguir que totes les interaccions entre la base de dades i l'aplicació passin sempre per JDO. D'aquesta manera aconseguirem el nostre principal objectiu, automatitzar tot el procés de treball amb la base de dades, de manera que només ens hem de preocupar del funcionament de l'aplicació. JDO s'encarrega de la resta.

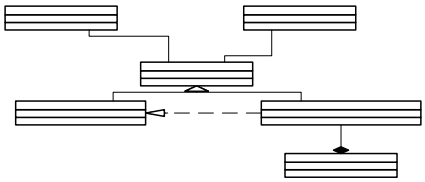
Pot semblar però que agafem la llibreries JDO i aquestes ja fan tot de manera intel·ligent. Abans hem de dir-li a JDO amb quina base de dades volem treballar i de quina manera. Començarem l'exemple doncs veient com configurar JDO perquè treballi amb la nostra base de dades.

L'objectiu principal del nostre exemple és poder veure tots els aspectes possibles i més comuns de la persistència, que defineix l'especificació 1.0.1 i algunes de les característiques que incorpora Kodo, així com el seu funcionament pràctic. Els aspectes que hem de poder veure en l'exemple són la identitat, la classificació de classes, el cicle de vida, el model de persistència, l'herència, implementació de les classes pròpies de JDO en l'aplicació, les transicions dels objectes, les consultes JDOQL i com és el DTD. Partirem de l'especificació de l'exemple, definirem una sèrie de classes amb les que treballarem. Definirem la seva persistència en la base de dades, i més tard veurem com treballa l'exemple com a resultat de tot el procés d'implementació.

6.2 Especificació de l'exemple:

La proposta d'exemple que fem és el desenvolupament d'un programa que permeti controlar diferents magatzems d'una empresa a partir d'un catàleg de productes que l'usuari definirà prèviament. També hem de poder modificar i predeterminar alguns aspectes de la configuració dels productes com la marca.

Primerament hem de veure quines són les característiques dels productes del catàleg i del propi magatzem i que l'empresa vol que el programa contempli. En el primer cas són el codi intern del producte, el model del producte, la marca del producte i la línia del producte. Aquest últim aspecte simplement defineix de forma general el producte, com a grup. O sigui el model XX és del grup televisor. Tant la marca com la línia de producte ha de ser un aspecte modificable en tot moment. L'eina ens ha de permetre veure en tot moment quins són els productes que comptabilitza cadascun dels magatzems, fer entrades de nous productes, veure la seva quantitat total tant per magatzem com de forma global. A més hem de poder trobar un producte concret dins dels magatzems, per això definirem el numero de producte, que ens permetrà diferenciar els productes en si sent el mateix model, ja que en un magatzem podem tenir diferents productes del mateix model . O sigui diferenciar el producte 23 del model XX, del producte 24 del model XX també. Aquest és doncs el segon cas. L'aplicació també ha de poder realitzar diferents consultes sobre els magatzems, catàleg i productes, tractar els diferents productes, eliminant-los, modificant-los o canviant-los de magatzem. Per resumir la seva especificació i fer-la més comprensible donem el seu diagrama UML de les classes bàsiques:



Gràfic 17

Tot aquesta especificació ens permetrà veure molts dels aspectes de JDO, cal dir però que no tots, ja que és molt difícil de trobar un exemple que implementi totes les característiques i capacitats de fer JDO. Però l'exemple ens permetrà veure molts dels aspectes generals i ens servirà per introduir la implementació de JDO en una aplicació. Veurem com fa persistents les dades, com realitzar diferents consultes, i com tracta les dades que emmagatzema la base de dades.

6.3 Instal·lar i configurar Kodo 3.0:

La instal·lació de la versió d'avaluació de Kodo 3.0, necessària per poder executar el nostre exemple, no és gaire complexa, ja que només ens genera un directori on guarda tant la documentació del producte, com exemples d'implementació, o les eines de persistència. La part que cal destacar i que és necessària per qualsevol aplicació Java que vulguem fer persistent, són les llibreries. En elles es basa tota la implementació de Kodo. I per què aquestes funcioni amb qualsevol aplicació han de ser definides al classpath de l'ordinador, d'igual forma que ho fem amb les llibreries de Java. En el nostre cas podem definir un KodoDir , o incloure les llibreries dintre del nostre directori de Java. O com hem fet en el nostre exemple, incloure-les en el directori de la nostra aplicació.

6.4 Les llibreries JDO i Kodo:

Les llibreries de treball de JDO com ja hem indicat abans són dues, jdo.jar i jdori.jar. Kodo 3.0 implementa les llibreries bàsiques de JDO d'un altre manera i per tant si les volem veure al directori lib, de Kodo, no les trobarem. Jdo.jar dins per Kodo, passa a ser jdo-1.0.1.jar i les funcions de jdori.jar són absorbides per altres de les llibreries de Kodo.

Un aspecte molt important per poder treballar amb JDO i el nostre exemple, és afegir la llibreria del nostre controlador JDBC, que en el nostre cas és ojdbc14.jar, controlador natiu d'Oracle, i també afegir la llibreria serp.jar que Kodo anomena com inclosa en el seu paquet però que no ho fa de forma efectiva. Les dues llibreries es tenen que afegir al nostre directori lib, de Kodo 3.0.

Entre totes les llibreries es pot observar log4j-1.2.6, que és un producte de la fundació Apache, que ens genera arxius log que ens indiquen totes les incidències de Kodo. La inclusió d'aquest aspecte a la nostre implementació és opcional.

A més de la llibreria log4j-1.2.6, també trobem més llibreries de la fundació Apache, llibreries per fer aplicacions amb JDO i entorns administrats, d'aspecte més complex i que aquí no analitzarem, però que són necessàries i fonamentals per Kodo 3.0.

6.5 Configurar JDO amb la base de dades:

En aquest pas veurem com configurar Kodo per treballar amb la nostre base de dades. Per poder fer això es dona per suposat que ja tenim una base de dades instal·lada i en marxa en un ordinador. L'exemple l'hem realitzat sobre un ordinador de forma que la base de dades i esta instal·lada de forma local, tot i això com veurem no hi ha problema si tenim la base de dades de forma remota. És però requisit previ, carregar una base de dades qualsevol a Oracle 9i, que ens permetrà incorporar més tard el nostre esquema. També és important definir un usuari que no té perquè ser l'administrador de la base de dades amb suficients rols per poder treballar amb una base de dades.

Per definir amb quina base de dades treballarem ho farem definint tots els aspectes en l'arxiu Magatzems.properties. Aquest arxiu configura la PersistenceManagerFactory, i la PersistenceManagerImpl, que ens permetran fer la persistència com ja veurem més endavant. I per un altre banda, totes les opcions que necessita la PersistenceManagerFactory per fer la connexió i treballar amb Oracle 9i. Són les que hem anomenat dins de la classe JDOHelper opcions sobre els mètodes d'arrancada flexible. Inicialment les opcions bàsiques que en fan falta per només connectar amb la base de dades Oracle 9i són:

- ConnectionDriverName: oracle.jdbc.driver.OracleDriver
- ConnectionURL: jdbc:oracle:thin:@localhost:1521:”nom de la base de dades”.
- ConnectionUserName: “nom d’un usuari de la nostra base de dades.
- ConnectionPassword: “contrasenya del usuari que hem anomenat abans”.

Totes aquestes opcions varien segons la configuració del nostre ordinador, i és molt important configurar-les segons la configuració que tinguem. Sinó no podríem fer servir l’exemple que aquí expliquem en qualsevol ordinador. La primera opció, defineix el controlador JDBC de la nostra base de dades Oracle. La segona opció, és la configuració que li dona JDO al controlador JDBC de la nostra base de dades. Oracle té variacions en aquest aspecte, en el nostre cas hem escollit el que es fa servir de manera més general segons Oracle. Aquí definim el nom o la direcció ip de l’ordinador que té instal·lada la base de dades Oracle. En el nostre cas al ser de forma local hem escrit “localhost”, tot i que podríem escriure “127.0.0.1”. “1521” és la configuració del port de connexió per defecte de qualsevol base de dades Oracle. Si aquesta varia ho hem d’indicar segons el port que tinguem. Per últim, donarem el nom de la base de dades amb la que treballem, i que ha estat ja definida prèviament a Oracle.

Les altres dues opcions, “UserName” i “Password”, on les que ens permetran indicar l’usuari de la base de dades amb el que treballem. Aquest ha de tenir suficients drets perquè JDO pugui treballar amb Oracle.

La resta d’opcions que veiem a l’arxiu, defineixen el comportament que ha de tenir la base de dades i que JDO ha de tenir en compte a l’hora de treballar. Totes elles ja les hem descrit anteriorment a l’apart, 3.5.5 on parlàvem de JDOHelper. En aquest aspecte Kodo afegeix algunes opcions que més tard explicarem a l’annexa A, junt amb una ampliació de les pròpies de JDO més bàsiques.

6.6 Definició dels objectes persistents:

Fins ara hem fins alguns dels aspectes a tindré en compte per poder fer el nostre exemple. Ara passarem a explicar la seva implementació en si. Com ja hem indicat, l’aplicació, ha de poder tractar una sèrie d’objectes, els productes. Aquest tenen una sèrie diversa de característiques pròpies que ja hem definit anteriorment. Per tant la primera acció que hem de fer és generar una classe que hem anomenat “AparellCataleg”. Aquesta classe s’encarregarà de definir els productes o aparells propis del catàleg. I més tard fer generar una classe “AparellMagatzems”. Les dues són filles de “Aparells”. Quin és el sentit que té fer aquesta distinció, ens podem preguntar, JDO al fer la persistència de les classes, genera una taula per cada classe a la base de dades. Si volem que els aparells que es defineixen al catàleg siguin únics, al fer una mateixa taula per a tots, tot i que es diferenciïn en només en el “Nombre de producte”, el nostre catàleg inclourà també els diferents aparells que anem afegint dins dels magatzems. Per tant per poder tenir un catàleg prèviament definit, l’hem d’aïllar dels aparells del magatzems. A més d’aquesta manera fem una taula complerta pels magatzems i un altre pel catàleg.

Com que podem tenir molts magatzems, i no coneixem quins són realment o si esta previst si variarà el seu nombre, generem una classe amb l’objecte Magatzem. Aquest fet ens permetrà veure un dels aspectes més interessants de JDO, com treballa una objecte compost

per un altre, ja que dins del Magatzem definirem una llista d'Aparells de magatzem. El que hem anomenat tercer classe dins del model de persistència, tot i ser una classe de segona.

Per un altre costat, definirem dues classes més, Marca i LineaP, que definiran alguns dels aspectes dels productes i que com ja hem dit són alguns dels aspectes que han de ser modificables, i predeterminats.

Aquestes són doncs les classes principals que defineixen en si l'exemple. Però com fem persistents aquests objectes?

Un primer aspecte que hem de definir és quin és el tipus d'identificació que volem fer servir, si d'aplicació o de base de dades. Ja que segons això definirem d'una manera o un altre les nostres classes. La primera opció és més elaborada d'aplicar i treballar, que no pas el segona, però ens pot permetre definir aspectes concrets sobre el tractament de les nostres dades. A l'annexa A, explicarem com portar a terme la identificació de classes. En el nostre cas apliquem la identificació de base de dades que és molt més simple i automatitzada.

Un segon aspecte és definir la PersistenceManagerFactory i la PersistenceManagerImpl, que són la base de la persistència de les diferents instàncies que generi la nostra aplicació. Com ja hem vist abans, les configurem a l'arxiu Magatzems.properties on les hem definit.

L'últim aspecte que hem de tenir en compte per poder fer la persistència d'objectes a partir de les classes que hem definit, és generar el DTD.

6.7 Definició del DTD:

El DTD és, com ja hem vist a la tercer capítol, un arxiu amb terminació .jdo, de format XML, on definim el paquet, les classes, els tipus especials d'atributs i altres aspectes a tenir en compte, a partir del que es realitza la persistència dels objectes.

Com ja hem vist al final del tercer capítol, la definició del DTD no es gaire complexa. Només hem de conèixer les "tag" que ens fan falta per fer la nostre persistència. En el nostre cas, Kodo no varia en res la definició que ja hem fet, i per tant respecte la definició del DTD de l'especificació 1.0.1.

En el nostre exemple el DTD és l'arxiu package.jdo, que acompanya a la resta d'arxius de l'exemple. Es pot editar amb qualsevol programa editor XML, o programa editor de text. Abans però explicar la seva estructura hem de recordar que tot el que definim en aquest arxiu, és el que serà tindrà capacitat de ser persistent, i per tant tot allò que vulguem fer persistent en algun moment, ha de ser definit com classe i més tard introduït en aquest arxiu. En el nostre exemple l'arxiu DTD és de la següent forma:

```
<?xml version="1.0" encoding="UTF-8"?>

<jdo>
  <package name="">
    <class name="Aparells"/>
    <class name="AparellCataleg" persistence-capable-superclass="Aparells" />
    <class name="AparellMagatzems" persistence-capable-superclass="Aparells" />
    <class name="Marca"/>
    <class name="LineaP"/>
    <class name="Magatzem">
      <field name="aparells">
```

```
            <collection element-type="AparellMagatzems"/>
          </field>
        </class>
    </package>
</jdo>
```

Les primeres 3 línies són de caràcter obligatori en qualsevol DTD de JDO, com ja hem vist anteriorment. Les 5 següents línies defineixen les classes que volem fer persistents. Cal denotar el que hem definit a les línies 5, 6, 9, 10 i 11. Les dues primeres les indiquem com filles de la classe "Aparells". En les 9, 10, i 11 destaquem el caràcter d'un dels atributs de la classe Magatzem, on indiquem primer quin és l'atribut, i a la línia 10 indiquem que es tracta d'una col·lecció d'aparells del tipus de la classe "AparellMagatzems". Aquest aspecte és molt important, ja que a la classe l'hem definit a una simple variable de tipus List (llista), i en el nostre cas li estem indicant a la base de dades quin tipus de dada ha emmagatzemar aquesta llista. És important aquest fet, ja que aquí podem veure una certa interacció entre objectes persistents i és la manera de com definim els objectes amb atributs tipus array, collection, etc.. definits per altes classes i no variables simples, on no caldria indicar res.

6.8 Transaccions entre el programa i la base de dades:

L'exemple, incorpora una sèrie de classes on estan definides pantalles gràfiques, més agradables i intuïtives pel usuari. Algunes d'elles podem veure diferents operacions que podem fer respectes als objectes persistents. Aquestes operacions ja les hem anomenat anteriorment, són les transaccions entre l'aplicació i la base de dades. Veurem algunes de les transaccions de l'exemple:

Com ja hem explicat anteriorment, mestre una instància d'un objecte no es fa persistent, JDO la considerarà com transitòria. I aquesta no es farà persistent fins què no ho indiqui la nostra aplicació. Agafem per exemple l'arxiu "PantallaMagatzems", en aquest arxiu definim tot el tractament de dades que volem fer entorn a l'objecte Magatzem. Primerament poder fer les transaccions i persistència d'objectes, farem la importació de la llibreria jdo, que és la que ens permetrà fer les operacions de persistència: `import javax.jdo.*;`

Definim dos atributs de caire general i imprescindibles per la persistència, la PersistenceManagerFactory que anomenem pmf, i la PersistenceManager que anomenem Listpm. La primera operació que fem en l'arxiu, és definir la pmf a partir de l'arxiu "Magatzems.properties". I més tard la pm, a partir de la pmf i un atribut tipus llista "MagatzemsList", que contindrà tots els magatzems que definim o que ja han estat definits prèviament. Aquest atribut llista, també l'hem definit prèviament. La següent operació que realitzem és omplir la llista de magatzems a partir de la base de dades. Per fer això, definim un atribut "Extent", on indiquem quin és l'objecte, en aquest cas magatzem, que volem que la llista contingui: `Extent e = listPM.getExtent (Magatzem.class, true);` Com podem veure l'encarregat de recollir les instàncies de l'objecte magatzem, és la PersistenceManager.

Un cop tenim el grup d'instàncies sobre el que volem treballar, farem les diferents transaccions. Per cadascuna de les transaccions definim un altre PersistenceManager, tot i que podríem fer servir una de general per tota l'aplicació, però d'aquesta manera podem comprovar com en la mateixa aplicació pot haver més d'un PersistenceManager i inclous de diferent objecte. La primera transacció que fem es afegir un nou Magatzem a la base de dades, primerament definim tots els atributs que volem que tingui, en aquest cas el nom i una petita descripció. Realitzem una consulta que ja estudiarem més tard, i fem una nova instància de

magatzem que en aquest moment és transitòria. Al cridar més tard a la PersistenceManager per fer persistent la instància del magatzem, `pm.makePersistent (a);` , modifiquem l'estat d'aquesta i engeguem tot un procés entre l'aplicació i la base de dades, acabem de fer persistent un objecte. Si anem a la definició de taules d'Oracle, podrem veure les dades que hem disposat en la instància que hem fet persistir. Més tard confirmem la transacció `pm.currentTransaction ().commit ();` i tanquem la PersistenceManager. En el cas de què no volguéssim confirmar l'operació amb la base de dades podríem haver indicat una sentencia rollback per exemple, enlloc d'una commit, al igual que qualsevol base de dades.

Una altre transacció que fem sobre els magatzems és deletePersistent, on podem eliminar una instància persistent de la base de dades. Aquestes són en definitiva les transaccions que fem en el nostre exemple, i també en el tractament d'aparells.

El cas de les instàncies dels aparells del magatzem és una mica més complex, ja que són instàncies secundaries, que formen part d'aquest. D'aquesta manera fem persistents dades que depenen d'una altre instància. L'únic fet que varia, respecte de la persistència del magatzem com instància primària, és que després de la definició dels aparells, els fem persistents a l'afegir-los a l'atribut llista del magatzem, i fet al magatzem persistent. Tot això ho podem veure a l'arxiu "PantallaCMagatzem".

Un dels aspectes que hem implementat a cadascun dels objectes són les InstanceCallBacks. Aquestes ens permeten veure en tot moment que succeeix entre l'aplicació i la base de dades. A més ens poden permetre fer una sèrie d'accions respecte la persistència dels objectes com per exemple fer marxa enrera en la confirmació de l'operació realitzada.

Hem vist doncs que són de forma pràctica algunes transaccions i que són les instàncies primàries i secundaries.

6.9 Consultes JDOQL:

Hem explicat anteriorment que en a l'arxiu "PantallaMagatzems" fèiem una consulta. Les consultes com ja hem explicat anteriorment es realitzem en un llenguatge propi de JDO, el JDOQL. La implementació de Kodo, afegeix més opcions a les consultes de JDO, algunes pròpies de la futura especificació 2.0. En el nombre primer cas de consulta que trobem, volem trobar que no hi hagi a la llista prèvia de magatzems cap altre magatzem amb el mateix nom que hem donat. Per fer això, primerament hem recollit en una variable el valor del nom que volem donar d'alta com instància persistent. Més tard definim una variable Extent, que fa referència a partir del PersistenceManager a l'objecte sobre la que fem la consulta, Magatzem. I definim la consulta de la següent manera:

```
Query L = pm.newQuery(q, filtreCod);
Collection result = (Collection) L.execute ();
```

Com podem veure hem anomenat a la consulta L. El PersistenceManager , pm, obre una nova consulta, a partir de la variable Extent q, i el filtre de la consulta, la variable filterCod. El valor que hem donat a filterCod és: `String filtreCod =("Nom.contains (\""+pNom+"\") ");` On "Nom" es l'atribut del nom del magatzem, i "contains" indica que trobi tot el que contingui la paraula donada, la variable "pNom". Aquesta no és l'única consulta de l'exemple, en diferents pantalles en trobem més, com per exemple a "ConsultaAMagatzems", o "PantallaCMagatzems",..

JDOQL és un llenguatge amb una gran potencia, tot i això aquest és un dels punts on JDO encara ha d'evolucionar més, ja que la cosa es pot complicar quan fem operacions de consulta complexes com fem amb SQL. De moment però Kodo ens permet fer consultes SQL, aquest fet ens permet resoldre el problema, però trenca amb una de les polítiques de JDO, tenir un llenguatge propi de consulta que no depengui de SQL.

6.10 Compilació i execució de l'exemple :

Fins ara hem vist una mica per sobre els aspectes més importants que calien destacar de l'exemple d'implementació JDO que acompanya a aquest document. Primerament hem definit la connexió amb la base de dades, quins eren els aspectes de codi importants i que ens feien persistents les instàncies, i com fer consultes i transaccions amb la base de dades. Però com fa Kodo la correspondència de les classes donades amb la base de dades? Fins ara en cap moment hem dit com indiquem que volem fer persistència amb la base de dades. Per poder fer això prèviament teníem que definir cada peça del trencaclosques. Ara es l'hora d'ajuntar-les.

El primer pas que cal fer és preparar les classes persistents per que les administri JDO. Primer les compilem, a partir del seu estat .java, amb l'ordre "javac" com fem ham qualsevol de les classes de Java, sobre la que treballem. I més tard mitjançant la comanda "jdoc", li donem a JDO l'administració de les classes. L'eina JDO és el que anomenàvem en l'apartat 3.1, JDO Enhacer de Kodo. Desprès d'aquest sentencia, JDO s'encarrega de la persistència de les classes, afegint i modificant el codi que li fa falta a cada classe per poder ser classes amb capacitat de persistència. En el nostre cas cridarem juntament amb la sentencia cridarem al DTD que hem definit "jdoc package.jdo". El DTD, li indica a l'Enhacer quines són les classes a modificar i que té fer.

El següent pas és generar la correspondència entre l'aplicació i la base de dades, a més de l'esquema de treball a la base de dades. Per fer això cridem a l'eina "mappingtool". I en el nostre cas la cridem amb les opcions "- action refresh". Aquestes opcions indiquen que volem fer una actualització de l'arxiu de correspondència i de l'esquema de dades a la base de dades. Com que no existeix res d'això, l'eina "mappingtool" les genera. Si anem a la consola de la base de dades podrem veure tant l'esquema generat com les taules que ha generat la correspondència de les classes, i taules amb el mateix nom que les nostres classes. En cas que no indiquem l'esquema, JDO agafa un preexistent.

Amb l'arxiu de correspondència creat, JDO genera un lligam indestructible entre les dues parts. Hem d'explicar però que gràcies a la natura de l'eina "mappingtool", el lligam funciona en tots dos sentits, i si es modifica la classe hem de tornar a cridar a "mappingtool" que perquè modifiqui la taula corresponent de la nostre base de dades.

Per poder comprovar els funcionament de l'exemple, criden des de la consola: "java ControlMagatzem", o l'arxiu "Executa.bat". Anteriorment hem d'haver compilat tots els arxius .java.

Kodo permet la implementació d'un arxiu log mitjançant la llibreria de Log4J. Aquest ens ha de permetre veure les incidències de l'exemple durant la seva execució, així com les excepcions pròpies de JDO.

Conclusió:

Com hem pogut veure JDO és una eina de treball molt potent que ens permet resoldre de forma senzilla el problema que hem plantejat al principi del document, la persistència dels objectes. A més, i de forma més concreta hem treballat sobre la persistència dels objectes del llenguatge Java, cap al que esta orientat JDO.

Com hem pogut comprovar la incorporació de JDO com una eina de treball tant per als programadors Java com per als especialistes en bases de dades és molt important, al trencar en cerca manera la desvinculació que hi havia entre les dues parts, ja que les tècniques i sistemes anteriors necessitaven de grans coneixements de les dues disciplines per poder fer una aplicació potent i que aprofites a versatilitat dels objectes i la potencia de les bases de dades.

Un exemple molt clar l'hem trobat en Kodo 3.0. Una implementació molt complerta de JDO, que ens permet veure com funciona de manera pràctica i ens mostra tota la potencia i facilitat d'ús que es demanava qualsevol capa de persistència.

També però hem pogut observar que JDO és una eina que ha d'evolucionar encara molt, però que ja és tot un referent om a eina dins de la programació d'objectes. El seu futur passa per la següent especificació, la 2.0, encara en procés de discussió, i de la que hem avançat alguns aspectes que l'especificació 1.0.1 ja hi donava. Aquest especificació suposarà d'acord amb els experts la consolidació de JDO pel futur.

Annexa A: Kodo 3.0:

Aquest annexa es permetrà conèixer més a fons tot el funcionament de Kodo 3.0 i intenta ser un petit manual d'ús de la implementació Kodo 3.0.

A.1 Instal·lació de Kodo 3.0:

La implementació de Kodo 3.0 la podem trobar de diferents formes, a la web de Solarmetric: `kodo-jdo-3.0.0.tar.gz` (versió per als sistemes operatius Mac Os X i Unix.), `kodo-jdo-3.0.0.exe` (versió per als sistemes operatius Microsoft Windows NT, 2000 i XP. Inclou un instal·lador) i `kodo-jdo-3.0.0.zip` (versió .zip per a Microsoft Windows NT, 2000 i XP.)

En la versió d'avaluació, a més dels fitxers d'instal·lació hem descarregar de la web una clau d'avaluació que el propi fabricant ens proporcionarà. Aquesta clau ha de ser incorporada a l'arxiu *.properties que ha d'acompanyar a qualsevol aplicació que fem ham Kodo. L'hem d'escriure a la capçalera de l'arxiu de la següent manera: Sense aquesta clau tant en la versió oficial com la d'avaluació no podrem fer servir Kodo. `kodo.LicenseKey: XXX.`

Les versions instal·lades es composen dels següents directoris:

- Bin: les eines de Kodo.
- Docs: la documentació en Angles tant en format web com .pdf. A més de l'especificació JDO 1.0.1 del JCP.
- Eclipse: plugins que permeten fer servir Kodo en l'entorn de programació Eclipse.
- Jca: controladors i eines per treballar amb entorns administrats.
- Lib: llibreries essencials pel funcionament de Kodo. (pot falta la llibreria serp.jar necessària pel us de Kodo i que cal afegir a aquest directori).
- Reversetutorial: exemple que explica la utilització de l'eina "reversemappingtool".
- Samples: diversitat d'exemples, molt útils ja que tracten diferents casos.
- Src: conté codi obert de JDO que podem modificar pel nostre ús i adaptar-lo a les nostres necessitats.
- Tutorial: conté un exemple pas per pas de com fer una aplicació Java que faci servir Kodo 3.0.

A més el directori principal conté una sèrie d'arxius dels que cal destacar: jdo.dtd (conté tota la sintaxi d'un arxiu DTD de Kodo), kodo.properties (exemple de com ha de ser l'arxiu *.properties de qualsevol aplicació que incorpori Kodo) i Log4j.properties (que ens explica la sintaxi de configuració de Log4J per què treballi amb Kodo).

Els requeriments de funcionament de Kodo són: una base de dades, i el paquet Java JDK 1.2 o versions posteriors, instal·lats correctament.

Per instal·lar Kodo ho podem fer de dues maneres mitjançant la versió executable que ens proporciona una instal·lació guiada pas per pas. O bé amb la versió comprimida .zip que un

cop ja no hi està comprimida carregarem cridant a l'eina "jdocmd.bat", si tenim un Windows NT, 2000 o XP, o "jdocommand", si la nostra versió del sistema és 95,98 o ME; amb la consola del sistema. Aquestes eines es troben a la carpeta "Bin".

A.2 Descripció de les llibreries de Kodo 3.0:

Principalment les llibreries de Kodo 3.0 les trobem dins del directori Lib, tal com hem explicat. En aquest apartat explicarem la utilitat de cadascuna i el seu origen, ja que no totes són pròpies de Kodo, sinó que són implementades per altres fabricants:

- **kodo-jdo.jar** : permet la implementació de Kodo 3.0 en el desenvolupament d'aplicacions.
- **kodo-jdo-runtime.jar** : permet l'ús de Kodo en l'execució.
- **jakarta-commons-collections-2.1.jar** : Col·lecció de classes, creada per l'Apache Foundation. Necessària per tant en el desenvolupament d'aplicacions com en l'execució.
- **jakarta-commons-lang-1.0.1.jar** : Col·lecció d'eines, també d'origen Apache, i necessàries pel desenvolupament i l'execució.
- **jakarta-commons-logging-1.0.3.jar** : Grup de classes d'incrustació de mesures (logging wrapper). D'origen Apache, imprescindibles pel funcionament general de Kodo 3.0.
- **jakarta-commons-pool-1.0.1.jar** : Classes d'agrupament (pooling). D'origen Apache, i molt necessàries en tot moment per Kodo 3.0.
- **jakarta-regexp-1.1.jar**: llibreria que conté expressions regulars. De procedència Apache, i molt necessària per Kodo tant en la programació com l'execució d'aplicacions.
- **jca1.0.jar** : Connector d'arquitectura Java, necessari per aplicacions d'entorn administrat. Forma part del paquet J2EE de Sun.
- **jdbc-hsql-1_7_0.jar** : controlador jdbc de la base de dades HSQL. Es fa servir per explicar com incorporar un controlador jdbc en els exemples de Kodo 3.0.
- **jdbc2_0-sdtext.jar** : Extensió del paquet JDBC2. Necessari per què pugui treballar Kodo amb versions inferiors al paquet JDK 1.4 del Java de Sun.
- **jdo-1.0.1.jar** : Llibreria imprescindible que conté la norma de JDO, de l'especificació 1.0.1 del JCP. L'origen el té al JCP.
- **jndi.jar** : Interfícies JNDI. Necessari per aplicacions en entorns administrats, i versions JDK inferiors a la versió 1.4. Pròpia de Sun.
- **jta-spec1_0_1.jar** : Interfícies JTA. Necessari per aplicacions en entorns administrats. Es fa servir com extensió pels paquets JDK inferiors a 1.4. L'origen és de Sun.
- **log4j-1.2.6.jar** : Permet la implementació d'una utilitat de log. D'origen Apache, és una llibreria opcional.
- **serp.jar** : grup d'eines imprescindibles pel funcionament de Kodo 3.0. La trobem a la web de serp.
- **xalan.jar** : llibreria que ens permet fer transformacions XML de dades en la nostra aplicació. Necessària pel DTD i la correspondència, quan el paquet Java JDK és d'una versió inferior a 1.4 . És programada per Apache.

- **xercesImpl.jar** : Llibreria companya de l'anterior, que permet la implementació de XML.
- **xml-apis.jar** : Companya de les dues llibreries anteriors, incorpora interfícies XML.

A més de totes aquestes llibreries hem d'incorporar les llibreries jdbc de la nostra base de dades, necessàries per la implementació de les nostres aplicacions amb la base de dades escollida. Com exemple podem veure l'ús de jdbc_hsql-1_7_0.jar. En el cas del nostre exemple hem incorporat la llibreria jdbc: ojdbc14.jar creada per Oracle per la base de dades Oracle 9i.

A.3 Descripció de les eines de Kodo 3.0:

Com hem pogut veure en el desenvolupament del document, Kodo incorpora una sèrie d'eines que ens permeten treballar de forma àgil amb JDO. Tot i que ja les hem anomenades, recordem quines són: *Schematool*, *Mappingtool*, *Metadatatool*, *Schemagen*, *remotemanagementtool*, *reversemappingtool*, *dbsequencefactory*, *dbschemafactory*, *dbmappingfactory*, i *appidtool*. Tot i que algunes ja les hem explicat en el nostre exemple, intentarem aprofundir una mica més en totes elles i afegirem informació per les que no coneixem. A més dels arguments propis de cada eina, totes les eines de Kodo tenen en comú els següents arguments:

- *-properties/-p <properties file or resource>* : donem l'arxiu .properties que fem servir, per connectar amb la base de dades i donar la PersistenceManagerFactory, i algun altre dels aspectes que explicarem més tard, en aquest annexa.
- *-<property name> <property value>* : indiquem una propietat com les que indiquem a l'arxiu .properties de forma directe.
- *-codeFormat./-cf.tabSpaces <spaces>* : donem el nombre d'espais que escrivim entre paraula i paraula.
- *-codeFormat./-cf.spaceBeforeParen <true/t | false/f>*: indiquem si volem un espai davant de parèntesis.
- *-codeFormat./-cf.spaceInParen <true/t | false/f>>*: indiquem si volem un espai darrera de cada parèntesis, tant a l'obrir com al tancar.
- *-codeFormat./-cf.braceOnSameLine <true/t | false/f>*: indiquem si volem les claus d'obertura de sentencia en la mateixa línia que la definició del mètode.
- *-codeFormat./-cf.braceAtSameTabLevel <true/t | false/f>*: expliquem on volem la clau d'obertura si l'opció anterior es desactivada.
- *-codeFormat./-cf.scoreBeforeFieldName <true/t | false/f>*: activem l'opció de col·locar davant del nom les variables privades, un guió.
- *-codeFormat./-cf.linesBetweenSections <lines>* : donem el nombre de línies que volem entre diferents parts de codi.

Passem ara a descriure les eines i els seus arguments propis:

- **Schematool**: La funció principal d'aquesta eina és a partir de la definició d'un esquema XML, veure quines són les diferencies amb la base de dades i l'esquema XML, i aplica els canvis necessaris a la base de dades per a què hi tinguin correspondència. Per poder fer això, l'eina té en compte quin és el dialecte SQL que fa servir la nostra base de dades. La podem fer servir per poder manipular esquemes SQL. *Schematool* es cridada

de manera interna per l'eina *mappingtool*. Quan cridem l'eina podem afegir una series d'arguments:

- *–ignoreErrors/-i <true/t | false/f>*: permet tractar les excepcions de la base de dades.
- *–file/-f <stdout | output file>* : Permet escriure les comandes SQL en un arxiu.
- *dropTables/-dt <true/t | false/f>*: configura quines són les taules a expulsar a l'efectuar accions de refresh i retain.
- *kodoTables/*-kt <true/t | false/f> : permet generar informació a partir de les taules especials generades per Kodo, així com successions de taules, com part de l'esquema de la base de dades.
- *–indexes/-ix <true/t | false/f>*: permet manipular els índexs de les taules existents.
- *–primaryKeys/-pk <true/t | false/f>*: permet manipular les claus primàries de les taules existents.
- *– foreignKes/-fk <true/t | false/f>*: permet manipular claus externes de taules existents.
- *–record/-r <true/t |false/f>*: permet que no escriure els canvis fets per l'eina en la fàbrica d'esquemes actual.

L'eina a més incorpora la bandera *–action* que ens permet fer les següents accions:

- *add*: permet afegir a l'esquema taules, columnes , índexs, etc.
- *retain* : Guarda tots els components descrits a l'esquema XML, i expulsa a la resta.
- *drop*: Expulsa tots els components descrits a l'esquema. Les taules de l'esquema són expulsades si no li queden columnes desprès d'expulsar totes les columnes descrites a l'esquema.
- *Refresh*: Després de fer la mateixa acció de retain, fa add.
- *createDB*: tornar a crear la base de dades actual Es sol cridar amb la bandera *–file* per escriure el script en un arxiu.
- *dropDB*: Expulsa la base de dades actual. Es sol combinar amb *–file*.
- *Import*: importa un esquema donat a l'esquema de fabrica actual.
- *Export*: exporta l'esquema actual.

- **Schemagen** : Aquesta eina fa servir la llibreria DatabaseMetaData pròpia dels controladors JDBC que fem servir, per construir una representació interna de l'esquema de la nostra base de dades. Aquesta representació es generada en un arxiu XML, i que podem manipular amb *schematool*. Aquesta eina es cridada de forma interna per *mappingtool* i les fàbriques d'esquemes. La seva sintaxi complerta és `-indexes/-ix <true/t | false/f> -primaryKeys/-pk <true/t | false/f> -foreignKeys/-fk <true/t | false/f> -kodoTables/-kt <true/t | false/f> -schemas/-s <schema and table names> -file/-f <stdout | output file>`. Totes aquestes banderes tenen el mateix ús que al *schematool*.

- **Mappingtool**: en certa manera ja coneixem com és el funcionament d'aquesta eina. S'encarrega de crear l'esquema i la correspondència a partir de classes Java ja definides als arxius .java o .class tractats per jdoc, o l'arxiu .jdoc que les llista. També la podem fer servir per validar la correspondència que hem generat de forma manual, o importar o exportar correspondència de dades de la fabrica de correspondència. Hem de dir que aquesta eina ens permet fer la correspondència orientada a objectes cap a l'esquema, esquema cap a l'objecte i terme mig, i totes aquestes orientacions són donades per les accions. Podem cridar l'eina juntament amb una sèrie d'opcions:

  - *-file/-f <stdout | output file>*: permet escriure les correspondències en un arxiu XML.
  - *-schemaAction/-sa <add | refresh | retain | none>* : Permet fer les mateixes accions que *schematool* sobre l'esquema.
  - *-schemaFile/-sf <stdout | output file>*: permet escriure un arxiu XML modificable, que més tard podem enviar mitjançant l'eina *Schematool* a la base de dades per realitzar les modificacions.
  - *-dropTables/-dt <true/t | false/f>* : realitza la mateixa acció que amb l'eina *Schematool*.
  - *-ignoreErrors/-i <true/t | false/f>*: realitza la mateixa acció que amb l'eina *Schematool*.
  - *-schemas/-s <schema and table names>*: realitza la mateixa acció que amb l'eina *Schematool*.
  - *-readSchema/-rs <true/t | false/f>*: si configurem amb valor true aquesta opció, l'eina llegirà de manera complerta l'esquema. D'aquesta manera assegurem que no hi ha cap conflicte en l'esquema.
  - *-primaryKeys/-pk <true/t | false/f>*: permet llegir i manipular les claus primàries de les taules.
  - *-foreignKeys/-fk <true/t | false/f>*: permet llegir i manipular les claus externes de les taules.
  - *-indexes/-ix <true/t | false/f>*: permet llegir i manipular els índexs de les taules.

Aquesta eina sempre requereix d'un argument d'acció per tractar les classes, apart de les opcions que hem anomenat. Aquestes opcions són:

- *refresh*: permet que actualitzem la informació de la correspondència a partir de les classes.
- *drop*: elimina la informació de la correspondència per les classes donades.
- *validate*: valida la informació de la correspondència donada a partir de les classes. No varia ni las taules ni la informació de la correspondència.
- *buildSchema*: crea l'esquema basant-se en la correspondència de les classes donades.
- *revert*: reverteix la correspondència de les classes donades al seu estat prèviament guardat. Algunes de les fabriques de correspondència no permeten la reversió.
- *import*: importa informació de correspondència de l'arxiu XML donat, i la incorpora al sistema de correspondència emmagatzemat.
- *export*: exporta la informació de correspondència de les classes donades cap a un arxiu XML.

- **reversemappingtool**: permet generar classes .java persistents, classes d'identitat d'aplicació o arxius .jdo amb la descripció del paquet i les classes, a partir de l'esquema. Si no troba l'esquema, l'eina treballa a partir de la base de dades. L'eina incorpora una sèrie d'arguments:

  - *-schemas/-s <schema and table names>*: indiquem que volem la descripció de la correspondència a un arxiu que contingui una llista determinada per comes i noms de les taules.
  - *-package/-pkg <package name>*: indiquem el nom del paquet de les classes generades.
  - *-directory/-d <output directory>*: indiquem el directori on volem que es generi tot el codi. Si no ho indiquem, es genera en el directori en el que ens trobem.
  - *-useSchemaName/-sn <true/t | false/f>*: indiquem que el nom de l'esquema i les taules quedin reflectits en les classes que es generin.
  - *-useForeignKeyName/-fkn <true/t | false/f>*: indiquem que volem que es conservin els noms de les claus externes en les relacions entre les taules. Si no agafen el nom de la classe.
  - *-nullableAsObject/-no <true/t | false/f>*: indiquem que volem que les columnes de valors null, agafin com tipus Java el seu equivalent ajustat. Si no agafen com tipus els primitius equivalents.
  - *-primaryKeyOnJoin/-pkj <true/t | false/f>*: si tenim claus primàries que relacionen totes contra totes les taules, indiquem que volem generar les classes d'aquestes taules.
  - *-oneToManyRelations/-omr <true/t | false/f>*: indiquem que volem per cada relació un cap a un que trobi, una relació en les classes un cap a moltes.
  - *-useDatastoreIdentity/-ds <true/t | false/f>*: indiquem que volem identitat de base de dades per les taules amb una sola columna amb clau primària numèrica.
  - *-metadata/-md <package | class>*: indiquem que volem que les dades resultats a nivell de paquet, s'escriguin en un arxiu DTD.
  - *-customizerClass/-cc <class name>*: indiquem sobre quina classe de reversió volem que treballi l'eina. O amb la classe "kodo.jdbc.meta.ReverseCustomizer", o la classe "PropiertiesReverseCustomizer". Aquesta última permet treballar a partir del d'un arxiu .properties donat.
  - *-customizerProperties/-cp <properties file or resource>*: indiquem l'arxiu .properties amb el que ha de treballar l'eina.
  - *-customizer./-c.<property name> <property value>*: ens permet afegir alguna propietat pròpia de Java Bean a les nostres classes.

  Hem d'aclarir que l'arxiu que .properties del que parlem no té res a veure amb el que fem servir per indicar el controlador jdbc, i altres aspectes d'una aplicació, com hem fet en l'exemple. Aquest arxiu és la definició que li volem donar a cadascuna de les taules i quina classe volem que generi i algun altre aspecte a tindré en compte. Per exemple: si en el nostre exemple tinguéssim la taula Magatzem faríem: "Magatzems.TaulaMagatzem: Magatzems.Magatzems".

- **Metadatatool:** Aquesta eina ens permet generar el DTD a partir de les classes donades. Sempre generarà les metadades a partir del que troba escrit a les classes, i no afegirà res que no hi trobi en elles. Els seus arguments propis són :

  - *-verbose/-v <true/t | false/f>*: aquesta opció ens permet indicar si volem fer persistents tots els camps de les classes.
  - *-file/-f <metadata file>*: donem el nom de l'arxiu de metadades que volem que es generi.

- **appidtool:** aquesta eina ens permet indicar que volem una identitat d'aplicació de les nostres classes. Aquesta eina però necessita d'uns requisits previs. Les classes a tractar han d'estar compilades, i el DTD generat. A més en el DTD hem d'indicar les classes que volem que treballin amb aquesta identitat, afegint `objectid-class` davant la classe i `<field>` per cada clau primària, a més de `primary-key`. Els arguments d'aquesta eina són:

  - -ignoreErrors/-i <true/t | false/f>: ens permet tractar les excepcions que es generin.
  - -directory/-d <output directory>: ens permet indicar el directori on volem que es generin el resultats.

La resta d'eines que hem anomenat en aquest document ho trobem al directori "bin" realitzen o bé treballs entremigs que ja fan algunes eines com *mappingtool*, o bé són eines d'actualització de versions anteriors de Kodo. I no només calen per algunes feines específiques.

A.4 Descripció de l'arxiu *.properties:

En aquest apartats introduirem algunes de les opcions que podem donar a les nostres aplicacions i que Kodo implementa mitjançant un arxiu .properties. Hem de dir que aquestes són una ampliació de les ja donades per l'especificació 1.0.1 i que ja hem explicat anteriorment. Recordem que les opcions que implementava l'especificació i que són pròpies dels arxius .properties: *PersistenceManagerFactoryClass, ConnectionDriverName, ConnectionFactoryName, ConnectionFactory2Name, ConnectionPassword, ConnectionURL, IgnoreCache, Multithreaded, NontransactionalRead, NontransactionalWrite, Optimistic, RestoreValues,* i *RetainValues*. Les opcions que incorpora Kodo són:

- kodo.AggregateListeners: donada una llista de cadenes separades per comes, fen referència a les consultes que farem, afegint "listeners" a la nostra aplicació a més dels ja donats que fan referència a consultes bàsiques.
- kodo.ClassResolver: és una propietat pròpia de l'especificació 1.0.1, que ens permet resoldre els noms de les classes.
- kodo.ConnectionProperties: ens permet afegir alguna propietat al controlador JDBC que creguem convenient.
- kodo.ConnectionFactoryProperties: ens permet activar alguna propietat al `javax.sql.DataSource` que estiguem fent servir.
- kodo.Connection2DriverName: és una segona versió de *ConnectionDriverName* orientada cap a connexions que no són del tipus XA dels entorns administrats.
- kodo.Connection2Password: referent al "password" de la propietat anterior.
- kodo.Connection2URL: referent a les dues propietats anteriors.

- kodo.Connection2UserName: referent a les 3 propietats anteriors.
- kodo.Connection2Properties: referent a les 4 propietats anteriors.
- kodo.ConnectionFactory2Properties: referent a les 5 propietats anteriors.
- kodo.ConnectionRetainMode: ens permet controlar com realitza Kodo la connexió amb el magatzem de dades.
- kodo.CopyObjectIds: ens permet copiar el valor de la identitat dels objectes abans de donar-lo al codi. Si volem retornar-los modificats, donem el valor true a aquesta propietat.
- kodo.DataCache: ens permet anomenar la cache de dades.
- kodo.DataCacheTimeout: configura el temps que pot estar una dada a la cache.
- kodo.EagerFetchMode: Millora el rendiment de la carrega de relacions persistents
- kodo.FetchBatchSize: donem el nombre de files que recollirem de cop del resultat donar d'una consulta.
- kodo.FetchGroups: ens permet especificar mitjançant una llista de noms que poden ser carregats quan carreguem els objectes de la base de dades, separats per comes.
- kodo.FilterListeners: ens permet especificar una llista de noms de "listeners" disponibles per les nostres consultes.
- kodo.FlushBeforeQueries: ens permet descarregar o no els canvis realitzats en una transacció abans de fer les consultes.
- kodo.LicenseKey: recull la llicència d'ús del fabricant Solarmetric.
- kodo.ManagedRuntime: Ens permet donar un nom que descrigui la implementació dels "TransaccionManager" en un entorn administrat.
- kodo.ManagementServer: ens permet donar una descripció a la implementació del "ManagerServer" per poder cridar-la de forma remota.
- kodo.ManagementUI: ens permet donar una descripció a la implementació del "ManagerUI" per poder cridar-la de forma local.
- kodo.PersistenceManagerImpl: ens permet anomenar la PersistenceManager que farem servir en la nostra aplicació.
- kodo.PersistentClasses: ens permet donar una llista de classes que poden ser instànciades quan es crea una nova PersistenceManager.
- kodo.ProxyManager: ens permet donar nom a la "ProxyManager" que ens permet tractar els objectes secundaris.
- kodo.QueryCache: ens permet anomenar la cache de consultes.
- kodo.RemoteCommitProvider: ens permet anomenar "RemoteCommintProvider" que fem servir per la notificació de successos de forma remota.
- kodo.RestoreMutableValues: ens permet restaurar els valors anteriors dels objectes de classes secundaries quan apliquem una restauració de valors (rollback).
- kodo.RetainValuesInOptimistic: permet millorar el rendiment de les restauracions.
- kodo.TransactionMode: Ens permet especificar el mode de transacció per defecte.
- kodo.jdbc.AutoIncrementConstraints: En permet indicar a SQL que trobi les dependències dels objectes, donades per les relacions, i incrementar el valor de les claus primàries.
- kodo.jdbc.ClassIndicator: ens permet anomenar el "ClassIndicator".
- kodo.jdbc.ConnectionDecorators: ens permet donar una llista d'instàncies "ConnectionDecorators" ha instal·lar en la fabrica de connexió.
- kodo.jdbc.DataSourceMode: ens permet indicar el mode d'obtenció de dades quan integrem en el servidor d'aplicacions, les transaccions administrades.
- kodo.jdbc.DBDictionary: ens permet indicar la "DBDictionary" que farem servir per treballar amb la nostra base de dades.
- kodo.jdbc.FetchDirection: indiquem l'ordre de les consultes.

- kodo.jdbc.ForeignKeyConstraints: ens permet avaluar les claus externes per SQL, amb la intenció de no trencar la integritat referèncial de la base de dades.
- kodo.jdbc.JDBCListeners: ens permet instal·lar diferents "listeners" referents a les operacions de JDBC.
- kodo.jdbc.LRSSize: Ens permet indicar l'estratègia a utilitzar per calcular la mida de les llistes de resultats.
- kodo.jdbc.MappingFactory: ens permet descriure la fabrica de correspondència que farem servir.
- kodo.jdbc.ResultSetType: ens permet indicar la forma en què volem que JDBC ens doni els resultats de les consultes.
- kodo.jdbc.SchemaFactory: permet anomenar la SchemaFactory que farem servir.
- kodo.jdbc.Schemas: ens permet donar una llista d'esquemes i taules que faran servir les dades JDO.
- kodo.jdbc.SequenceFactory: ens permet anomenar la "SequenceFactory".
- kodo.jdbc.SubclassMapping: ens permet anomenar la "ClassMapping" ha instal·lar.
- kodo.jdbc.TransactionIsolation: indica el nivell d'exposició (isolation) de JDBC.
- kodo.jdbc.UpdateManager: permet anomenar el "UpdateManager" que farem servir per actualitzar la base de dades respecte la modificacions dels objectes.
- kodo.jdbc.VersionIndicator: permet anomenar l'indicador de versió.

A.5 Descripció del DTD de Kodo 3.0:

En principi el DTD descrit a l'especificació JDO és bastant complerta. Però la implementació Kodo va una mica més enllà i amplia alguns dels aspectes del DTD. Permet escriure estructures més complexes que les que podem fer amb les que dona l'especificació. Això ens permetrà solucionar alguns casos més específics. Podem dividir les extensions de Kodo en 4 parts: Relacions, esquema, correspondència objecte - base de dades, i altres.

A.5.1 Relacions:

Les extensions que podem incorporar a la nostra DTD, ens permeten un fet que l'especificació JDO no pot, donar la relació entre les classes de manera que la persistència dels objectes sigui més fidel a la realitat del tractament dels objectes.
S'ha de tenir en compte que el tractament de les relacions pot tant ajudar-nos a realitzar una correspondència entre els models de dades més complerta, però també fa més complex el tractament de les classes, ja que hem de tenir sempre present aquest fet, al tractar qualsevol instància.

- *inverse-owner*: amb aquesta extensió podem indicar que aquell camp és part d'una relació, que té el mateix nom, entre dos classes, i que a més aquest camp la defineix.
- *dependent*: afegint el valor true amb aquesta extensió indiquem aquest objecte persistent potser eliminat si és eliminat o anul·lat el seu objecte pare.
- *element-dependent*: és com l'extensió anterior, però referit a col·leccions del objecte.
- *value-dependent*: en aquest cas enlloc de col·leccions ens referim a valors de mapes (maps).
- *key-dependent*: en aquest cas enlloc de col·leccions ens referim a claus de mapes.
- *type*: Kodo manté tres tipus o nivells de relacions . El primer nivell permet que les relacions d'objectes coneguts siguin emmagatzemades mitjançant les claus primàries. El segon nivell permet emmagatzemar relacions amb objectes persistents

desconeguts però suportats, guardant el valor identitat de la seva instància. I el tercer nivell que ens permet emmagatzemar interfícies i objectes desconeguts, serialitzant el seu valor cap la base de dades.

- *element-type*: és el mateix cas que l'anterior referit a col·leccions d'objectes.
- *value-type*: és el mateix cas que l'anterior referit a valors de mapes.
- *key-type*: és el mateix cas que l'anterior referit a claus de mapes.
- *lrs*: fem servir aquesta extensió de caràcter booleà, podem marcar els camps que on volem que Kodo tracti la mida dels resultats que dona.

A.5.2 Esquema:

Aquestes extensions ens permeten treballar sobre l'esquema generat per la *Mappingtool* i ha partir de les classes donades.

- *jdbc-size* : ens permet definir la mida de la columna que guarda el camp.
- *jdbc-element-size* : té la mateixa funció que l'anterior però referida a arrays i col·leccions.
- *jdbc-value-size* : té la mateixa funció que les anteriors però referida a valors dels mapes.
- *jdbc-key-size* : té la mateixa funció que les anteriors però referida a les claus dels mapes.
- *jdbc-indexed* : permet indexar la columna que guarda les dades referides.
- *jdbc-element-indexed* : té la mateixa funció que l'anterior però referida a arrays i col·leccions.
- *jdbc-value-indexed* : té la mateixa funció que les anteriors però referida a valors dels mapes.
- *jdbc-key-indexed* : té la mateixa funció que les anteriors però referia a les claus dels mapes.
- *jdbc-ref-indexed* : té la mateixa funció que les anteriors però referida valors anteriors.
- *jdbc-delete-action* : si les relacions d'un objecte donat són emmagatzemades, permet eliminar les claus externes de les relacions.
- *jdbc-element-delete-action* : té la mateixa funció que l'anterior però referida a col·leccions.
- *jdbc-value-delete-action* : té la mateixa funció que les anteriors però referida a valors dels mapes.
- *jdbc-key-delete-action* : té la mateixa funció que les anteriors però referida a les claus dels mapes.
- *jdbc-ref-delete-action* : té la mateixa funció que les anteriors però referida valors anteriors.

A.5.3 Correspondència:

Aquestes extensions ens permeten millora la correspondència dels objectes a la base de dades.

- *jdbc-class-map-name* : ens permet especificar el tipus de correspondència que volem fer per a una classe. Aquesta pot ser vertical o per pisos(flat).
- *jdbc-version-ind-name* : ens permet especificar el tipus d'indicador de versió que volem fer per a una classe.
- *jdbc-class-ind-name* : ens permet especificar el tipus d'indicador de classe que volem fer per a una classe.

- *jdbc-field-map-name* : ens permet especificar el tipus de correspondència de camp que volem fer per a un camp.
- *jdbc-ordered* : especifica en quin ordre volem que les dades d'una col·lecció s'emmagatzemin.
- *jdbc-container-meta* : ens permet emmagatzemar informació no essencial dels camps de les col·leccions i els mapes. Per exemple si està buit o si té valor null.
- *jdbc-null-ind* : es fa servir quan tenim un camp incrustats d'un altre classe, donant el seu valor null quan calgui.
- *externalizer* : ens permet fer persistent un camp que no pot ser persistent.
- *factory* : ens permet tornar a crear el valor d'un camp de forma externa.
- *jdbc-class-ind-value* : es fa servir per les classes que fan servir el valor de metadada de l'indicador de classe.

A.5.4 Altres:

- *fetch-group* : ens permet indicar un grup de carrega per a un camp.
- *data-cache* : ens permet especificar la cache de dades per a una classe.
- *data-cache-timeout* : ens permet donar una mida a la cache d'una classe.
- *jdbc-sequence-factory* : ens permet descriure la Fabrica de seqüències que farem servir per crear les identitats de la base de dades de les noves instàncies de la classe.
- *jdbc-sequence-name* : ens permet posar nom de la seqüència de la base de dades.
- *jdbc-auto-increment*: ens permet especificar que volem que la identitat de la base de dades de les instàncies de la classe sigui incremental.

A.6 Incorporant l'opció de registre (log):

Una de les eines que incorpora Kodo és Log4j, de la fundació Apache. Aquesta eina ens permet implementat arxius de registre (log) d'accions de les nostres aplicacions tant en procés de desenvolupament com posteriorment. Aquesta eina ens permet fer el registre a partir de diferents nivells: jdoc, el DTD, l'execució, la cache de dades, jdbc, l'esquema i el SQL. Tots aquests nivells els hem d'especificar en un arxiu log4j.properties. El següent document ens mostra un model de registre:

```
log4j.rootCategory=WARN, console
log4j.category.kodo.Tool=INFO
log4j.category.kodo.Runtime=INFO
log4j.category.kodo.DataCache=WARN
log4j,category.kodo.MetaData=WARN
log4j.category.kodo.Enhance=WARN
log4j.category.kodo.Query=WARN
log4j.category.kodo.jdbc.SQL=WARN
log4j.category.kodo.jdbc.JDBC=WARN
log4j.category.kodo.jdbc.Schema=WARN

log4j.appender.console=org.apache.log4j.ConsoleAppender
```

Aquí podem veure com hem de descriure cadascun dels nivells. Com podem veure els nivell poden agafar diferents opcions o formes, a nivell informatiu (Info), de forma crítica (warn) o a nivell d'errors (error). En principi sempre que implementem Log4j a la nostre aplicació, podrem fer el registre de la nostra aplicació. La forma d'implementar-lo és cridant a la llibreria log4j-1.2.6.jar a l'hora de fer la compilació i execució de la nostra aplicació.

Annexa B: Alguns aspectes de l'exemple:

Aquest annexa es prenent que sigui el manual de l'exemple que acompanya al document.

B.1 Arxius de l'exemple:

Els arxius que formen l'exemple es troben al directori codi són:

- Classes java bàsiques dels objectes:
  - **Aparells.java:** defineix la classe pare dels aparells.
  - **AparellCataleg.java:** defineix la classe filla dels aparells del catàleg.
  - **AparellMagatzems.java:** defineix la classe filla dels aparells dels magatzems.
  - **LineaP.java:** defineix les possibles línies de producte dels aparells.
  - **Marca.java:** defineix les possibles marques dels aparells.
  - **Magatzem.java:** defineix els magatzems de l'empresa.

- Classes java d'entorn gràfic de l'aplicació:
  - **ConsultaAMagatzems.java:** permet la consulta dels aparells que hi han a tots els magatzems.
  - **ConsultaAparells.java:** permet la gestió dels aparells del catàleg.
  - **ControlMagatzem.java:** Menú principal de l'aplicació.
  - **PantallaCMagatzem.java:** permet la gestió dels aparells d'un magatzem donat.
  - **PantallaLineaP.java:** permet la gestió de les línies de producte dels aparells.
  - **PantallaMagatzems.java:** permet la gestió dels magatzems.
  - **PantallaMarcas:** permet la gestió de les marques dels aparells.

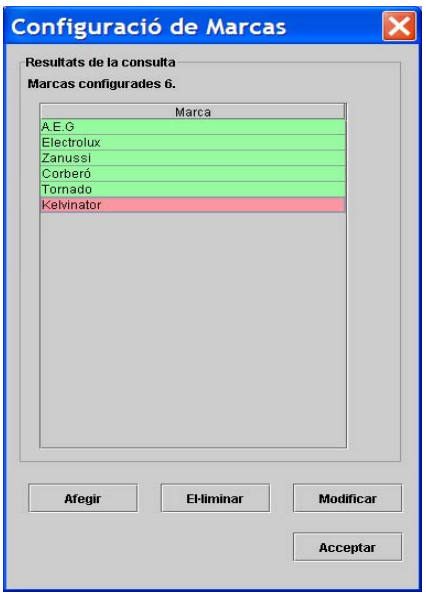
- Classes necessàries per la persistència amb Kodo 3:
  - **MagatzemsPersistenceManager.java:** defineix el PersistenceManager de l'aplicació.
  - **Magatzems.properties:** defineix alguns dels aspectes fonamentals pel funcionament de l'aplicació amb Kodo 3.
  - **package.jdo:** defineix la persistència de les classes de l'aplicació.
  - **log4j.properties:** defineix la configuració del registre log de l'aplicació.

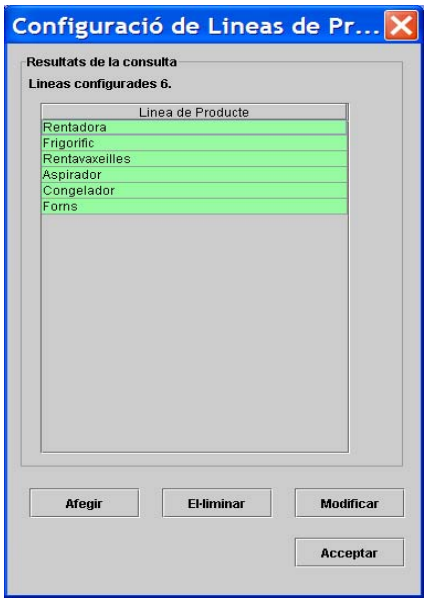
A més dels arxius de l'exemple adjuntem en un directori lib, totes les llibreries necessàries pel funcionament de Kodo3, i que hem utilitzat en la implementació del exemple, i l'arxiu per executar l'aplicació "Executa.bat"

B.2 Petit manual de l'exemple:

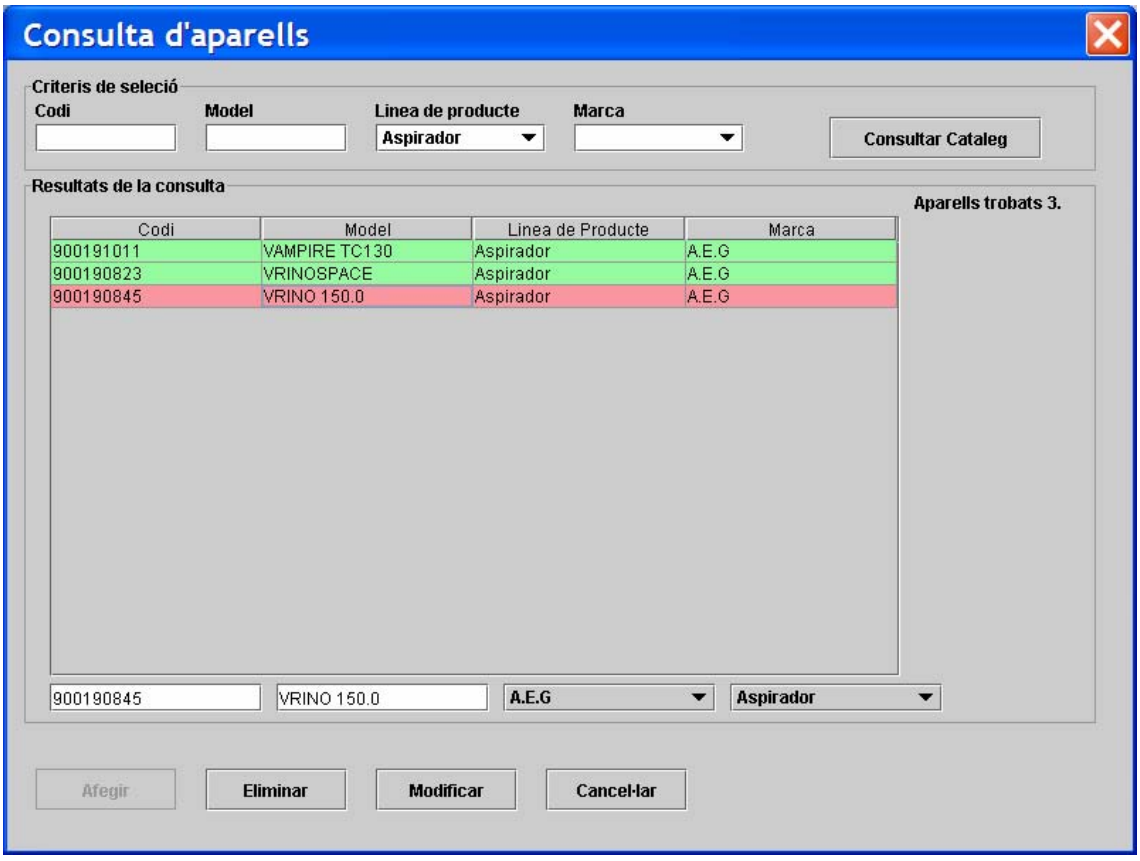
Inicialment l'exemple ens presenta una sèrie de menús amb totes les opcions que implementa. El primer menú "Arxiu" simplement ens permet sortir de l'exemple. I l'últim, ajuda, ens informa sobre l'exemple. Els dos menús principals que ens mostren el funcionament de l'exemple són el segon i tercer, Magatzem i Configuració.

Configuració com indica el seu nom, ens permet fer tota la configuració de les dades de l'exemple. El menú es divideix en dues parts, la primera indica les marques que poden tenir els diferents aparells i les línies de producte. Totes dues pantalles tenen un funcionament idèntic. Estan formades per una llista amb el contingut de les seves bases de dades, i d'una sèrie de botons que ens permeten afegir contingut nou a la llista, eliminar-lo o modificar-lo. Durant totes les operacions a la consola de treball de java, podrem veure en tot moment que succeeix entre la base de dades i les instàncies dels objectes donats.



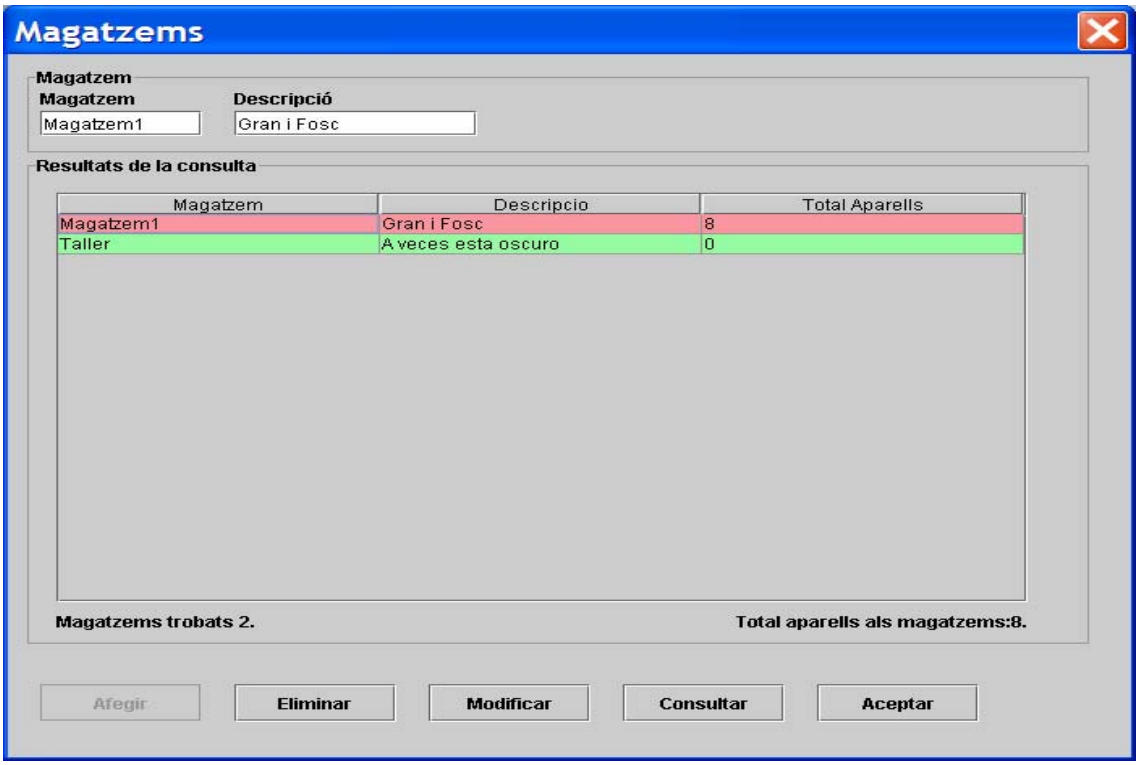


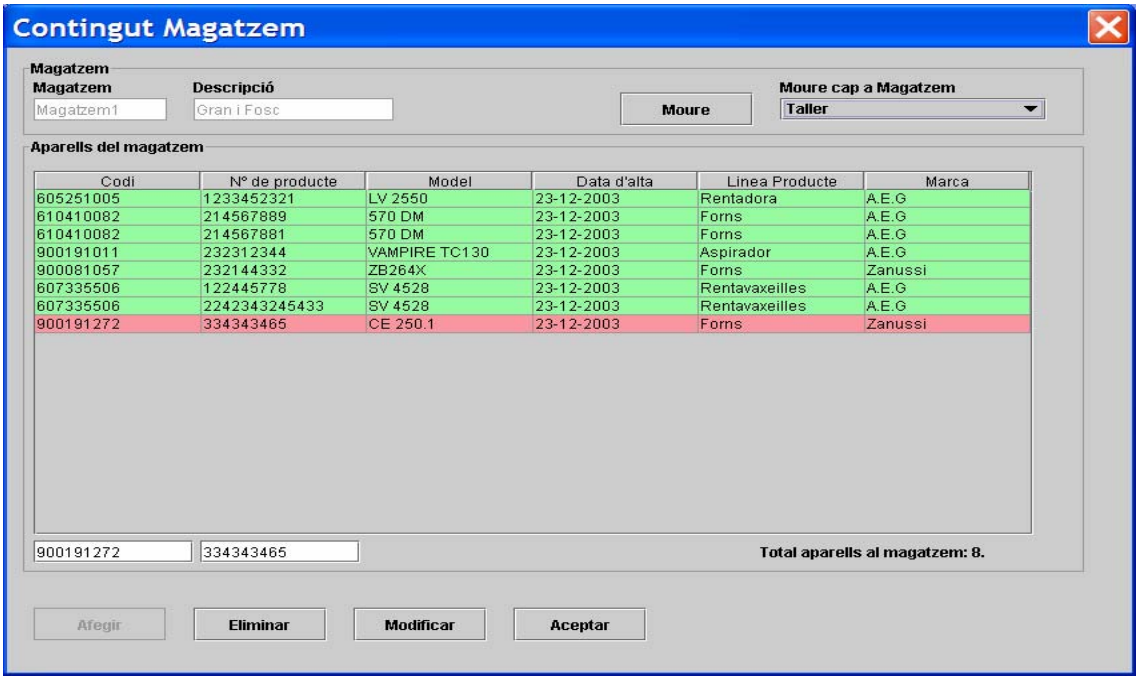
Aquest últim aspecte és comú també amb les altres pantalles on tractem objectes de la base de dades. Aquest aspecte ens ho ha permès la implementació dels "InstanceCallBacks" de JDO, que dona un missatge en cada operació entre JDO i les bases de dades.



En l'opció de catàleg podem introduir tot el catàleg d'aparells que volem introduir en els magatzems. En cadascun dels aparells introduirem les seves dades més importants de cadascun, entre les que destaquem les que ja hem introduït anteriorment de marca i línia de producte. Sense la introducció d'aquest catàleg, no pot funcionar la resta de l'exemple, el menú magatzem. A Magatzem podem introduir diferents magatzems i gestionar-los amb la

seva consulta. Quan es demana la consulta d'un magatzem, ens apareixerà una pantalla on podrem consultar quins són els aparells que té el magatzem en aquell mateix moment i fer diferents gestions. Mitjançant el botó moure de magatzem, podrem a més fer intercanvi d'aparells entre els diferents magatzems que existeixin. En tot moment sempre ens apareixerà els registrat els diferents moviments entre l'exemple i la base de dades.





Bibliografia:

- "Diseño e implementación de un lexicón computacional para lexicografia y traducción automática". Modelo de datos. Antonio Moreno Ortiz.: http://elies.rediris.es/elies9/4-2.htm.

- SOLARMETRIC KODO™ JDO 3.0.0 DEVELOPERS GUIDE (MANUAL DEL PROGRAMADOR DE KODO 3.0)
HTTP://WWW.SOLARMETRIC.COM/SOFTWARE/DOCUMENTATION/LATEST/DOCS/

- "JDO 1.0.1 SPECIFICATION." . Autor: Craig Russell. Sun MicroSystems.
HTTP://JCP.ORG/ABOUTJAVA/COMMUNITYPROCESS/FINAL/JSR012/INDEX2.HTML

- "Mapping Objects to relational Data Bases" Autor : Scott W.Ambler. Ronin International. Novembre 1998
HTTP://WWW.AMBYSOFT.COM/MAPPINGOBJECTS.HTML

- "Solving the Java Object Storage Problem". Autors : Douglas Barry, Torsten Stanienda. 1998
HTTP://CSDL.COMPUTER.ORG/COMP/MAGS/CO/1998/11/RY033ABS.HTM

- "Work starts soon on new Java Data interface." Autor: John Cox. Network World. 19 de Juliol 1999
HTTP://WWW.NWFUSION.COM/ARCHIVE/1999/70486_07-19-1999.HTML

- "An object database for embedded environments". Autor: David Jordan. Java Report. Abril 2000.
HTTP://WWW.OBJECTIDENTITY.COM/IMAGES/EMBEDDEDDB.PDF

- "An overview of JDO." Autor: David Jordan. Java Report. Juny 2000.
HTTP://WWW.OBJECTIDENTITY.COM/IMAGES/JDO_OVERVIEW.PDF

- "The design of a robust persistence layer for relational databases." Autor: Scott W.Ambler. Ronin International. 28 de novembre de 2000.
HTTP://WWW.AMBYSOFT.COM/PERSISTENCELAYER.HTML

- "Programing Persistence". Autor: Sun Microsystems. 2000.

- "XML roadmap". Autor: AQS. Febrer 2001.
HTTP://WWW.IT.UC3M.ES/~XML/DOCUMENTOS/XML-ROADMAP.PDF

- "The JDO Object model" Autor: David Jordan. Java Report. Juny 2001
HTTP://WWW.OBJECTIDENTITY.COM/IMAGES/JDO_MODEL82.PDF

- "Java Data Objects". Autor: Craig Russell. Sun Microsystems. Juliol 2001.
HTTP://WWW.THESERVERSIDE.COM/RESOURCES/ARTICLES/JDO/ARTICLE.HTML

- "A criticism of Java Data Objects (JDO)". Autor: Michael Azzi. 19 Agost 2001.
http://www.theserverside.com/home/thread.jsp?thread_id=8571

- "Why JDO is a critical component of J2EE?" Autor: Libelis. Agost 2001.

HTTP://WWW.JDOCENTRAL.COM/PDF/WHYJDO.PDF

- “JDO streamlines data access in Java “ Autor: Maggie Biggs. Novembre 2, 2001
http://archive.infoworld.com/articles/tc/xml/01/11/05/011105tcjdo.xml

- “JAVA DATA OBJECTS, a white paper“ Autor: OpenFusion. Febrer 2002.
http://www.jdocentral.com/pdf/JDO_WP_001d.pdf

-“A comparison between Java Data Objects (JDO), Serialitzation, and JDBC for Java persistence”. Autor: David Jordan. Object Identity Inc. 12 de Març del 2002.
http://www.jdocentral.com/JDO_Resources_Body.html

-“ Using data objects”. Autor : Dion Almaer. 2 Juny 2002.
http://www.onjava.com/pub/a/onjava/2002/02/06/jdo1.html

-“ The JDO query Languaje”. Autor: David Jordan. Java Pro. Juliol 2002.
http://www.fawcette.com/javapro/2002_07/magazine/features/djordan/

- “Desing considerations when using Entity Beans with CMP”. Autor: Oracle. Gener 2003.
http://otn.oracle.com/tech/java/architect/container_managed_persistence.pdf

- “JDO Genie Manual”. Autor: Genie. 2003.
http://downloads.hemtech.co.za/jdo/manual/contents.html

- “Persistence Frameworks – JDO Simplifies the “Buy versus Build” Decision”. Autor: Solarmetric. Gener 2003.
http://www.solarmetric.com/images/SolarMetric_Kodo_JDO_Buy_vs_Build.pdf

- “Castor TPV 1.2, A case study of a JDO based application”. Autor: Alberto lópez Tallon. Maig 2003.
http://www.jdocentral.com/pdf/CaseStudyCastorTPV.pdf

- “Don't Be Stubborn When Persistence Is Concerned” . Autor: Sudhakar Ramakrishnan. Oracle. 2003.
http://otn.oracle.com/oramag/webcolumns/2003/opinion/ramakrishnan_toplink.html

- “Day Four & WrapUp: Java Brings the Heat” Autor: Roger Smith. Oracle. 2003.
http://otn.oracle.com/events/javaone03/otn_daily_4.html

- “Java Data Objects” Autor: Robin M. Roos. Addison-Wesley. 2003.
http://www.ogilviepartners.com/Home.html

- “Java Data Objects” Autor: David Jordan i Craig Russell. O’Reilly & Associates. 2003.
http://www.oreilly.com/catalog/jvadtaobj/

- “Using and understanding Java Data Objects”. Autor: David Ezzio. Apress. 2003.
http://www.apress.com/book/bookDisplay.html?bID=106

- [RFC1208]. Autor: The Internet Engineering Task Force.
http://www.ietf.org/rfc/rfc1208.txt

- “Motores de persistència”. Vicent-Ramon Palasí Lallana.
http://www.programacion.com/java/articulo/joa_persistencia/

Pagines web de referència:

- Web de referència: http://access1.sun.com/jdo/

- Web de referència: http://groups.yahoo.com/group/JavaDataObjects/

- Web de referència: http://www.ambysoft.com

- Web de referència: http://www.jdocentral.com

- Web de referència: http://www.rexip.com

- Web de referència de la llibreria serp: http://serp.sourceforge.net

- Web de referència:
http://www.objectfrontier.com/Resources/DevConnect/TechConnect.html

- Web de referència: http://www.microsoft.com

- Web de referència: http://www.odmg.org

- Web de referència : http://www.hibernate.org