# *Java Data Objects – JDO*

Wladimiro Díaz

Diciembre de 2005

### Java y bases de datos

- Java es un lenguaje que define un entorno de ejecución sobre el que se ejecutan las clases definidas por el usuario.
- Las instancias de estas clases pueden representar datos del mundo real que se encuentran almacenados en una base de datos.
- Pero esto es un gran problema:
  - Las técnicas de acceso a los datos difieren para cada tipo de fuente de datos.
  - Esto es un reto para los desarrolladores, que deben utilizar *API's* diferentes para cada tipo de fuente de datos.
  - El desarrollador debe conocer, al menos, dos lenguajes: el lenguaje de programación Java y el lenguaje especializado de acceso a los datos requerido por la fuente de datos.
  - Además es muy probable que el lenguaje de acceso a los datos cambie con cada fuente de datos utilizada.
  - Aumentan los costes de desarrollo.



### Antecedentes

Tres estándares de almacenamiento de los datos en Java:

- **Serialización**.
  - Se utiliza para escribir el estado de un objeto y el grafo de objetos que éste referencia en un flujo de salida (*output stream*).
  - Preserva las relaciones entre los objetos Java, de modo que es posible reconstruir el grafo completo con posterioridad.
  - No soporta transacciones, consultas o acceso compartido a los datos entre usuarios múltiples.
  - La serialización se utiliza sólo para proporcionar persistencia en aplicaciones simples o en entornos empotrados que no pueden gestionar una base de datos de forma eficiente.
- ***Java Database Connectivity*** (JDBC).
  - Es necesario gestionar explícitamente los valores de los campos y su proyección en tablas de una base de datos relacional.
  - Por tanto, hay que tratar con dos modelos de datos, lenguajes y paradigmas de acceso a los datos, muy diferentes (Java y SQL).
  - El esfuerzo necesario para implementar el *mapping* entre el modelo relacional y el modelo de objetos es demasiado grande:
    - Muchos desarrolladores nunca llegan a definir un modelo de objetos para sus datos.
    - Se limitan a escribir código Java procedural para manipular las tablas de la base de datos relacional subyacente.
    - Se pierden los beneficios y ventajas del desarrollo orientado a objetos.
- ***Enterprise Java Beans*** (EJB) ***Container Managed Persistence*** (CMP).
  - La aquitectura EJB está diseñada para dar soporte a la computación distribuida de objetos.
  - También proporciona persistencia a través de *Container Managed Persistence* (CMP).
  - Fundamentalmente debido a su capacidad distribuida, las aplicaciones EJB son más complejas y más pesadas de ejecutar que JDO.
- Todos estos mecanismos requieren que el programador conozca los detalles estructurales de la base de datos subyacente.
- Muchos de ellos incluso requieren que sea el propio programador quien se responsabilice de la gestión de los detalles de persistencia.
- Para *liberar* al programador de esta tarea, el estándar *Java Data Objects* (JDO en adelante) proporciona un grado de abstracción mayor: *persistencia transparente*.

## Persistencia transparente

*Persistencia transparente* significa que:

- La persistencia de los objetos es automática.
- Toda la lógica necesaria para procesar los objetos persistentes se expresa en lenguaje Java puro.
- El programador de la aplicación no necesita conocer ningún lenguaje de base de datos, ni siquiera SQL.
- La *proyección* entre los objetos Java y los almacenados en la base de datos:
  - ☐ Es realizada fuera de escena por la implementación JDO.
  - ☐ Es totalmente transparente para el programador de la aplicación.
- **Resultado:** Desde el punto de vista del programador, los objetos persistentes se manipulan exactamente igual que los transitorios.



## Objetivos de JDO

- Dos objetivos fundamentales:
  - ☐ Proporcionar una interface estándar entre los objetos de la aplicación y el almacen de los datos (sea cual sea éste: base de datos relacional, orientada a objetos o incluso el propio sistema de archivos).
  - ☐ Simplificar el desarrollo de aplicaciones seguras y escalables proprocionando a los desarrolladores con un modelo único para trabajar con los datos persistentes.
- Adicionalmente:
  - ☐ JDO reduce la necesidad de escribir código SQL explícito y de gestionar las transacciones del sistema.
  - ☐ Actúa *blindando* al desarrollador de Java de los detalles del mecanismo subyacente que proporciona la pesistencia.

## Historia

- El JDO cuenta con una larga historia, aunque sólo una parte es visible si nos fijamos el desarrollo del propio estándar.
- Las raíces se encuentran en el ODMG (*Object Data Management Group*).
  - Primer intento de estandarizar el acceso transparente a bases de datos desde lenguajes de programación orientados a objetos.
  - El estándar ODMG es muy anterior al lenguaje Java.
  - Desarrollado en una época en la que el mayor debate entre la comunidad era qué lenguaje orientado a objetos dominaría en el futuro: C++ o Smalltalk.
  - El debate se quedó en el terreno académico mientras que Java se convitió en el estándar *de facto*.
  - El ODMG respondió adaptando sus interfaces C++ y Smalltalk al lenguaje Java
- JDO como ahora lo conocemos se origina por la *Java Specification Request* JSR-012 propuesta en 1999.
- Tras 3 años de un largo proceso de discusión por la *Comunidad Java*, finalmente en 2002 se aprobó como una especificación oficial.
- En el interín, muchas otras especificaciones se han convertido en estándares. Por tanto, JDO debe integrarse en el entorno proporcionado por esas especificaciones relacionadas.
  - *servlets* y *session EJBs* pueden manipular directamente instancias persistentes de JDO en lugar de tratar con el almacén de datos.
  - *Entity EJBs* con persistencia *bean-managed* pueden delegar la lógica de negocio y la gestión de la persistencia a las clases JDO, liberando al desarrollador de escribir todos los comandos SQL en la implementación de las clases.
- En la actualidad está sometido a un continuo e intenso desarrollo.



## Comparación

Comparación de los principales mecanismos de persistencia en Java:

| | Serialización | JDBC | EJB | JDO |
|---|---|---|---|---|
| **Persitencia transparente** | Transparente | No transparente | No/parcialmente transparente | Transparente |
| **Modelo de objetos** | Orientado a objetos | No orientado a objetos | Modelo de objetos simple. Sin herencia | Modelo orientado a objetos totalmente soportado |
| **Consultas** | No soportado | Si, escribiendo código SQL | Si, escribiendo código SQL | Soporte extendido vía JDO QL |
| **Transacciones** | No soportado | Soportado | Soportado | Soportado |
| **Portabilidad** | No aplicable | Débil soporte para BD relacionales. Requiere reescribir SQL. No soporta BD OO ni XML | El mismo nivel de soporte que JDBC | Excelente |

## Interfaces JDO



### Preliminares

- La interface JDO está definida en dos paquetes:
  - javax.jdo. Define la interface utilizada para desarrollar las aplicaciones y es la que desarrollaremos a continuación.
  - javax.jdo.spi. Contiene interfaces específicas de la implementación (*service provider interface*). No es una práctica recomendable utilizar directamente esta interface.

***DBD – Java Data Objects*** 10 / 50

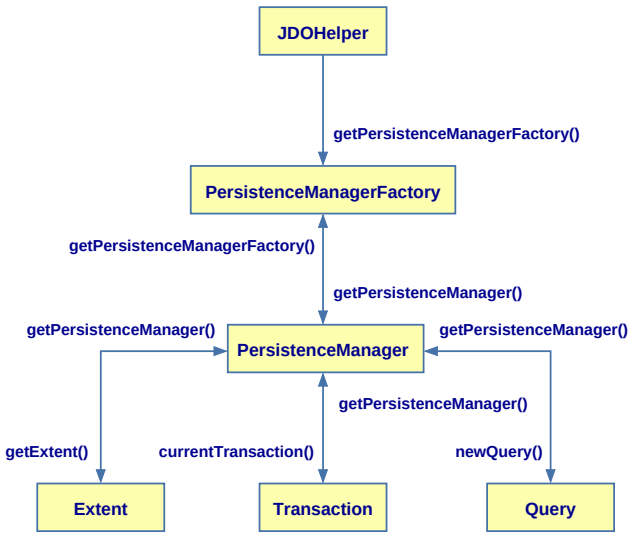
### El paquete javax.jdo

- JDO es una API realmente pequeña, ya que sólo necesita seis interfaces:
  - PersistenceManager.
  - PersistenceManagerFactory.
  - Transaction.
  - Extent.
  - Query.
  - InstanceCallback.
- También contiene la clase JDOHelper y un conjunto de clases de excepción.

***DBD – Java Data Objects*** 11 / 50

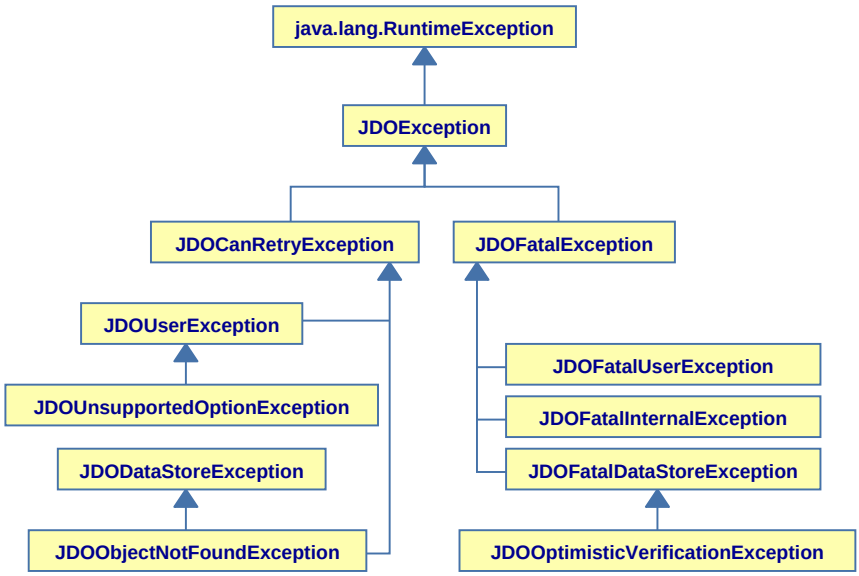
## Descripción de clase e interfaces

- PersistenceManager
  - Es la interface primaria de JDO.
  - Proporciona los métodos para crear las consultas, las transacciones y gestionar el ciclo de vida de las instancias persisentes.
- PersistenceManagerFactory
  - Es el responsable de configurar y crear las instancias PersistenceManager.
  - Representa la implementación particular de JDO utilizada: dispone de métodos para determinar las propiedades y características opcionales específicas de la implementación.
  - También proporciona los métodos para definir la conexión con el almacén y determinar la configuración del entorno sobre el que se ejecutan las instancias de la clase PersistenceManager.
- JDOHelper
  - Proporciona varios métodos de utilidad estáticos.
  - Se utiliza para construir una instancia de PersistenceManagerFactory a partir de un objeto Properties.
- Transaction
  - La interface Transaction proporciona métodos para definir el ámbito (begin y commit/rollback) de las transacciones.
  - Cada instancia PersistenceManager cuenta con una instancia Transaction asociada, que es accesible a través del método currentTransaction().
  - También proporciona métodos para controlar las características opcionales de la transacción.
- Extent
  - La interface Extent se utiliza para acceder a todas las instancias de una clase (y potencialmente a todas las instancias de sus subclases).
  - Se obtiene una instanca invocando al método getExtent() de la clase PersistenceManager.
  - Sobre el Extent es posible:
    - Iterar con cualquiera de los métodos convencionales de Java.
    - Realizar una consulta.
- Query
  - Una consulta es un filtro expresado en el *JDO Query Language* **(JDOQL)**.
  - Una instancia de Query consta de varios componentes cuyos valores se modifican mediante la interface proporcionada por esta clase.
- InstanceCallback
  - Proporciona un mecanismo para especificar el comportamiento de un objeto persistente cuando suceden determinados eventos en su ciclo de vida.
  - Mediante esta interface se definen los métodos que son invocados cuando la instancia sufre dicho cambio.
- Relación entre las instancias de las interfaces JDO:



## Excepciones en JDO

- Son muchos los puntos en los que puede fallar un componente y que no se encuentran bajo el control directo de la aplicación.
- La fuente del error puede estar en:
  - En la propia aplicación.
  - En la implementación del JDO.
  - En el almacenamiento (*datastore*) subyacente.
  - En cualquiera de los niveles de la arquitectura *multi-tier* del sistema.
- Para preservar la transparencia de JDO, la filosofía de diseño es:
  - Tratar todas las excepciones como excepciones en tiempo de ejecución (*runtime*).
  - Permitir que el usuario decida qué excepciones específicas capturar dependiendo de los requisitos de la aplicación.
- Las excepciones en JDO se clasifican en varias categorías:
  - Errores de programa que se pueden corregir y por tanto, reintentar la ejecución.
  - Errores de programa que no se pueden corregir porque el estado de los componentes se ha modificado.
  - Errores lógicos internos a la implementación de JDO y que son responsabilidad del desarrollador de la interface.
  - Errores asociados al *datastore* subyacente y que pueden ser corregidos.
  - Errores asociados al *datastore* subyacente pero que no pueden ser corregidos debido a un error en el propio *datastore* o en la comunicación con éste.
- JDO utiliza interfaces externas al propio API (por ejemplo, Collection). Cualquier excepción en estos componentes se utiliza directamente sin modificación.



## El proceso de desarrollo JDO 14 / 50

### Diseño

- El diseñador de una aplicación JDO **no necesita** ocuparse del tipo de base de datos ni del esquema de la base de datos.
- Sin embargo, es conveniente dividir las clases de Java en dos categorías:
  - **Clases persistentes**, que son aquellas que contienen los datos gestionados en la base de datos.
  - **Clases de negocio**, que son las que contienen la lógica de la aplicación y las consultas sobre los datos persistentes.
- Las clases persistentes son simples clases de Java con atributos, métodos de acceso básicos (*getters*) y si es necesario, métodos de manipulación (*setter* y otros).
- La especificación JDO no impone esta separación lógica, sin embargo:
  - Se considera una buena práctica separar la lógica de negocio de los datos persistentes.
  - Hace que las clases persistentes sean más simples y más fáciles de generar automáticamente.

***DBD – Java Data Objects*** 15 / 50

### Las clases de negocio

- Las clases de negocio (*bussiness classes*) contienen toda la lógica de negocio.
- Estas clases acceden a los datos persistentes a través de instancias de las clases persistentes.
  - Puesto que JDO proporciona persistencia transparente, no es necesario invocar a la base de datos (¡no SQL!) en las clases de negocio.
  - No es necesario que sean clases persistentes, lo que reduce la sobrecarga y mejora el rendimiento de las aplicaciones.

***DBD – Java Data Objects*** 16 / 50

## Las clases persistentes

- Los datos representados por los objetos persistentes son en última instancia almacenados en una base de datos subyacente.
- Por tanto, estos objetos deben ser proyectados en entidades de la base de datos.
  - La proyección la lleva a cabo el *enhancer* de JDO utilizando la información proporcionada por un fichero XML de metadatos.
  - En muchos casos existe también una herramienta para crear el esquema de datos sobre la base de datos (*Schematool*).
- En el fichero XML de metadatos, no sólo se definen todas las clases persistentes, sino que también se definen sus relaciones.
  - Este fichero se crea manualmente, pero muchas implementaciones de JDO proporcionan herramientas de ayuda a esta tarea.

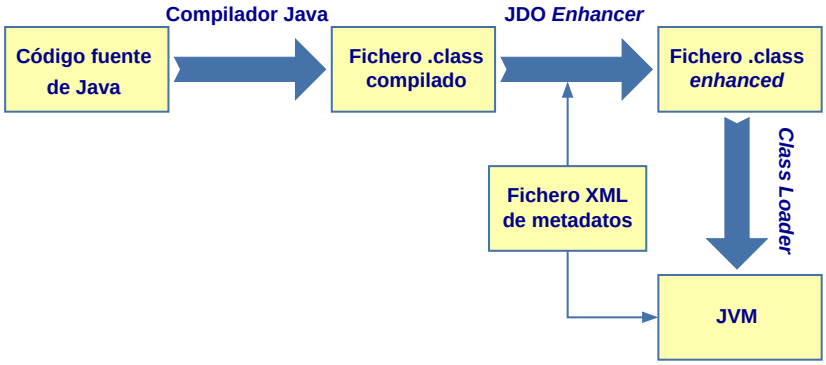


## Proyección en el *datastore*

- JDO permite el almacenamiento del modelo de objetos en una gran variedad de arquitecturas de bases de datos.
- **Base de datos relacional**.
  - Organizada como un conjunto de tablas, cada una de las cuales cuenta con conjunto de filas y columnas.
  - Una columna puede almacenar valores de un tipo atómico determinado.
  - Cada celdilla de la tabla en una fila particular contiene un único valor del tipo de la columna. El valor almacenado también puede ser un valor *nulo*.
  - Las instancias se identifican unívocamente por el valor de la columna que contiene la *clave primaria*.
  - Las relaciones se definen y se fuerzan mediante columan específicas que contienen *claves foráneas* que referencian columnas de una tabla.
- **Base de datos orientada a objetos pura**.
  - Una extensión del modelo de objetos. Los dominios de objetos se almacenan con sus campos primitivos, del mismo modo que existen en la memoria de la aplicación.
  - Las instancias se identifican mediante un identificador único generado por el sistema.
  - Las referencias se almacenan como objetos y los objetos ya no referenciados son eliminados mediante un *garbage collector*.
  - El *extent* no es una construcción intrínseca en una base de objetos pura, sino que se implementa como una clase que contiene un conjunto de objetos.
  - En este modelo cualquier tipo referenciado puede ser compartido entre múltiples objetos y los cambios realizados sobre la instancia son visibles a todos los objetos que lo referencian.
- ***Application Programming Interface* (API)**.
  - API's que definen métodos para crear, leer, actualizar y borrar instancias de dominios astractos.
  - La implementación del almacenamiento subyacente queda completamente oculta por la API.
- La implementación de JDO proporciona soporte para bases de datos relacionales así como para bases de datos orientadas a objetos (puras o híbridas).
- A medida que la implementación de JDO esté disponible para otras arquitecturas de base de datos, estarán disponibles nuevas proyecciones.
- Por ejemplo, existen bases de datos basadas en el modelo XML, por lo que es previsible que dispongamos en breve de esa proyección

## *Class Enhancement*

- Una vez que las clases han sido compiladas (en *bytecode*), son alimentadas al *enhancer* JDO.
- El *enhancer*, a partir de la información contenida en el fichero XML de metadatos, modifica el *bytecode* del siguiente modo:
  - Hace que las clases implementen la interface javax.jdo.PersistenceCapable.
  - Implementa los métodos requeridos por la interface javax.jdo.PersistenceCapable basándose en las relaciones entre objetos y la estructura de la base de datos.
  - Crea (si es necesario) un esquema de base de datos para los datos pesistentes.
  - Añade cualquier funcionalidad adicional necesaria para la gestión óptima y transparente de la persistencia.
- Estas clases JDO-*enhanced* son cargadas en la *Java Virtual Machine* (JVM) en tiempo de ejecución.
- La JVM también lee en tiempo de ejecución el fichero XML de metadatos para gestionar los datos persistentes.
- En la siguiente figura se esquematiza el proceso de desarrollo y ejecución de una aplicación JDO.






## Consideraciones finales

- Como se observa en el equema, en la JVM sólo se cargan las clases *enhanced*.
- Es necesario tener en cuenta, por tanto, el coste en rendimiento:
  - Las clases *enhanced* son en general más grandes que las clases normales. Esto justifica el uso de las clases de negocio.
  - El tráfico con la base de datos (que típicamente reside en otro nodo de la red) deteriora el rendimiento. Para soslayar este potencial punto débil muchas implementaciones cuentan con complejos sistemas de *caching*.

### Fases de implementación

- Como hemos visto, en la implementación de un sistema JDO se pueden distinguir dos fases:
  - La primera fase de definición de las clases que deben ser persistentes. Esto se hace mediante la definición del fichero XML de metadatos.
  - La segunda fase consiste en definir el código necesario para proporcionar persistencia a estas clases, permitir su recuperación y reutilización y su eliminación cuando alcanzan el final de su ciclo de vida.

***DBD – Java Data Objects*** 22 / 50

### Diseño de la capa de persistencia

- El diseño de la capa de persistencia puede realizarse de tres formas:
  - *Forward Mapping* – Se dispone de un modelo de clases y se diseña un esquema de base de datos para representar estas clases.
  - *Reverse Mapping* – Ya existe un esquema de base de datos y se diseñan las clases para que se adecúen a este esquema.
  - *Meet in the Middle Mapping* – Se dispone tanto de un esquema de base de datos como de clases nuevas y se desea generar un sistema coherente.
- El proceso implica tomar decisiones acerca de qué clases deben ser persistentes. En JDO, el resultado del proceso es un conjunto de ficheros de metadatos.
- Son muchos los factores que intervienen en la toma de decisiones asociada a este proceso:
  - **Tipos** – Persistencia de los tipo de Java.
  - **Identidad** – En JDO todos los objetos tienen identidad única.
  - **Esquema** – Esquema nuevo o reutilización de un esquema previo.
  - **Proyección Objeto-Relacional** – Las clases se relacionan mediante:
    - Asociaciones de cardinalidades *1–1*, *1–N* y *M–N*.
    - Relaciones de herencia propias de lenguajes orientados a objetos.

***DBD – Java Data Objects*** 23 / 50

## El fichero de metadatos

- El fichero XML de metadatos se utiliza para definir cómo deben persistir las clases y qué atributos de la clases deben persistir.
- Constituye una parte esencial del sistema JDO y se utiliza tanto en el proceso de *enhancement* como en tiempo de ejecución.
- La meta-información es necesaria para las siguientes clases:
  - Las clases persistentes (*PersistenceCapable*).
  - Cualquier clase que modifique directamente los campos de una clase persistente (*PersistenceAware*).
- Las clases *PersistenceAware* requiere una entrada de la forma:

```
<class name="unaClase" persistence-modifier="persistence-aware"/>
```



## Localización de los metadatos

- JDO espera que el fichero (o ficheros) con los metadatos se localice en posiciones particulares dentro de la estructura de la aplicación.
- Un ejemplo: si disponemos de una clase dbd.sgp.bussines.Cliente, JDO buscará los siguientes ficheros en el orden especificado:
  - META-INF/package.jdo
  - WEB-INF/package.jdo
  - package.jdo
  - dbd/package.jdo
  - dbd/sgp/package.jdo
  - dbd/sgp/bussines/package.jdo
  - dbd/sgp/bussines/Cliente.jdo
- Adicionalmente, también es posible desboblar los metadatos en varios ficheros. Supongamos que tenemos las siguientes clases:

```
dbd/sgp/bussines/Cliente.class
dbd/sgp/bussines/Stock.class
dbd/sgp/bussines/Pedido.class
dbd/sgp/bussines/utiles/Direccion.class
dbd/sgp/bussines/utiles/Telefono.class
```

- Los metadatos para estas cinco clases se pueden definir de varias formas:
  - Poner todas las definiciones en dbd/sgp/bussines/package.jdo.
  - Poner las definiciones de Direccion y Telefono en dbd/sgp/bussines/utiles/package.jdo y las definiciones de Cliente, Stock y Pedido en dbd/sgp/bussines/package.jdo.
  - Utilizar un fichero .jdo para cada clase utilizando el nombre de la clase (por ejemplo dbd/sgp/bussines/utiles/Direccion.jdo).
  - Una mezcla de las estrategias anteriores.

## Un ejemplo

- Supongamos que queremos proporcionar persistencia a la siguiente clase Java de ejemplo:

```
package dbd.sgp.bussines;
import java. util .*;
/**
*  Clase Cliente
*/
public class Cliente implements Comparable {
  private int numCliente;
  private String nombre;
  protected Direccion direccion;
  protected Collection<Telefono> telefonos =  new HashSet<Telefono>();
  protected Collection<Pedido> pedidos = new HashSet<Pedido>();
   ...
}
```

- Necesitamos el siguiente fichero de metadatos:

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
   "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
   "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
   <package name="dbd.sgp.bussines">
      <!-- Clase cliente -->
      <class name="Cliente" identity-type="application">
         <inheritance strategy="new-table"/>
         <field name="numCliente"
            persistence-modifier="persistent"
            primary-key="true"
            value-strategy="autoassign"/>
         <field name="nombre" persistence-modifier="persistent"/>
         <field name="direccion" persistence-modifier="persistent"/>
         <field name="telefonos" persistence-modifier="persistent">
            <collection element-type="dbd.sgp.bussines.Telefono"/>
         </field>
         <field name="pedidos"
            persistence-modifier="persistent"
            mapped-by="cliente">
            <collection element-type="dbd.sgp.bussines.Pedido"/>
         </field>
      </class>
   </package>
</jdo>
```

## Java y el modelo relacional

- El modelo de objetos de Java y el modelo relacional constituyen dos modelos de datos distintos:
  - Cuentan con tipos de datos muy diferentes.
  - Difieren enormemente en la forma en que representan los datos y expresan la lógica de programa.
- La diferencias más destacadas son:

| Modelo Java | Modelo relacional |
|---|---|
| Clase | Tabla |
| Campo | Columna |
| Instancia | Fila |
| Identidad | Clave primaria |
| Referencia | Clave foránea |
| Interface | Sin equivalente |
| Colección | Sin equivalente |
| Herencia de clases | Una o varias tablas |

## Colecciones Java

- Las colecciones en JDO se pueden representar sólo como instancias en memoria y que carecen de representación directa como tales en el *datastore*.
- Se instancian bajo demanda y se descartan cuando ya no son necesarias.
  - Hay sin embargo excepciones a esas reglas generales y algunas implementaciones soportan proyecciones más avanzadas.



## Proyección automática

- Si comenzamos con un conjunto de clases Java y dejamos que la implementación del JDO genere el esquema relacional:
  - Éste escogerá una representación relacional apropiada para las clases de Java.
  - Realizará la proyección entre las clases y las tablas relacionales.
  - La implementación tomará una serie de decisiones, incluyendo el nombre de las tablas y columnas, los tipos de columna para los campos de Java y cómo se representan las colecciones y relaciones del modelo.
- Es útil comprender las diferentes decisiones de proyección que se pueden realizar.
  - Esto permite valorar la flexibilidad que ofrecen las varias implementaciones de JDO y determinar cuáles se integran más fácilmente en el entorno de desarrollo utilizado.

## Implementación del modelo de objetos

- **Proyección de los dominios** definidos durante el diseño detallado.
  - En el caso de dominios simples, basta con sustituir por los tipos de datos y tamaños adecuados.
  - El caso de los dominios complejos (enumeración, campos multievaluados, etc. . . ) requieren un mayor esfuerzo.
- **Definir la identidad**. La identidad puede definirse a través de la aplicación o haciendo uso del *datastore*.
- **Definir las tablas**. Proyectar el modelo de objetos (clases, asociaciones, generalizaciones, etc. . . ) en tablas.



## Proyección de dominios

- Los tipos de datos que se pueden utilizar para un tipo de Java específico varían entre los sistemas relacionales y la implementación de JDO.
- Los tipos de columnas soportados para cada tipo Java en cada almecenamiento subyacente se especifica en la documentación específica de la implementación de JDO.
- En la tabla se muestra una lista de los tipos de dato relacionales comunmente soportados para los tipos de dato Java soportados por JDO.

| Tipo de dato Java | Tipo de columna relacional |
|---|---|
| Boolean, boolean | BIT, TINYIT, SMALLINT, BYTE, INT2 |
| Byte, byte | TINYINT, SMALLINT, BYTE, INT2 |
| Character, char | INTEGER, CHAR, VARCHAR |
| Short, short | SMALLINT, INTEGER, NUMBER, INT2 |
| Integer, int | INTEGER, NUMBER, INT4 |
| Long, long | BIGINT, DECIMAL, INT8 |
| Float, float | FLOAT, DECIMAL, REAL |
| Double, double | DOUBLE, NUMBER, DECIMAL |
| BigInteger | DECIMAL, NUMBER, NUMERIC, BIGINT |
| BigDecimal | DECIMAL, NUMBER, DOUBLE |
| String | CHAR, VARCHAR, VARCHAR2, LONGVARCHAR, CLOB |
| Date | TIMESTAMP, DATE, DATETIME |
| Locale | VARCHAR |

## Definición de la identidad – *datastore*

- Identidad *datastore*.
  - La asignación de identificadores de objeto es responsabilidad del JDO.
  - La clase no cuenta con un campo específico para implementar la identidad.
  - El propio entorno JDO creará una clave que contará con su propia columna en el almacenamiento persistente (*datastore*).
  - Para definir que una clase debe contar con identidad *datastore*, es necesario especificar la siguiente metainformación para la clase:

```
<class name="Telefono" identity-type="datastore">
    <inheritance strategy="new-table"/>
    <field name="tipo" persistence-modifier="persistent"/>
    <field name="telefono" persistence-modifier="persistent"/>
</class>
```

  - El proceso de generación de la identidad es delegado a la implementación del JDO.
  - Esto no significa que no tengamos control sobre cómo se lleva a cabo este proceso. JDO defines varias formas de definir esta identidad:
    - **native** – Permite a JDO seleccionar la identidad que mejor se ajusta al almacenamiento subyacente.
    - **increment** – Utiliza el SequenceTablePoidGenerator de JPOX que cuenta con una tabla en el *datastore* con una fila por tabla en la que se almacena el último valor del identificador.
    - **autoassign** – Utiliza autoincremento suponiendo que el *datastore* lo soporte.
    - **identity** – Identidad soportada por el *datastore*.
    - **sequence** – Utiliza una secuencia, que debe ser soportada por el *datastore*.
    - **etc. . .**
  - Definir qué tipo de generador de identidad se utilizará es una simple cuestión de añadir la información en los metadatos:

```
<class name="MyClass" identity-type="datastore">
    <datastore-identity strategy="sequence" sequence="MY_SEQUENCE"/>
    ...
</class>

<class name="MyClass" identity-type="datastore">
    <datastore-identity strategy="autoassign"/>
    ...
</class>

<class name="MyClass" identity-type="datastore">
    <datastore-identity strategy="identity"/>
    ...
</class>

<class name="MyClass" identity-type="datastore">
    <datastore-identity strategy="uuid-string"/>
    ...
</class>
```

  - Algunos de estos métodos requieren atributos adicionales, pero son simples de especificar.
  - En este tipo de identidad, no es posible obtener la identidad del objeto accediendo a un campo asociado (porque no existe).
  - Existen sin embargo otras formas de acceder a la identidad:

```
/*
* pm es una instancia de PersistenceManager
*/
Object id = pm.getObjectId(obj);
```

    y también:

```
Object id = JDOHelper.getObjectId(obj);
```

## Definición de la identidad – *application*

- Identidad *application*.
  - El desarrollador toma el control de la especificación de los *id's* de JDO.
  - Esto requiere que las clases dispongan de un atributo de tipo **clave primaria**.
  - El campo o campos de la clave primaria deben estar presentes en la propia definición de la clase.
  - Para especificar que la clase utilizará identidad generada por la aplicación, es necesario añadir los siguientes metadados para la clase:

```
<class name="Stock" identity-type="application">
    <inheritance strategy="new-table"/>
    <field name="numStock"
        persistence-modifier="persistent"
        primary-key="true"
        value-strategy="autoassign"/>
    <field name="descripcion" persistence-modifier="persistent"/>
    <field name="precio" persistence-modifier="persistent"/>
    <field name="tasa" persistence-modifier="persistent"/>
</class>
```

## Generación de la identidad

- Mediante la especificación de identidad *datastore* o *application* se define quién tiene control sobre el proceso de asignación de la identidad.
- Sin embargo, existen mecanismos para definir cómo se asignan automáticamente estos valores (incluso aunque no sean valores que definan la identidad).
- JDO exige que la implementación soporte la generación automática de los valores de las claves tanto para identidad *datastore* como *application*.
- Los métodos para generar los identificadores únicos son en muchos casos dependientes del almacenamiento subyacente, pero se dispone también de métodos “neutros”.
  - Los métodos dependientes del *datastore* son en general más eficientes.
- Para la identidad *datastore* el elemento de metainformación es:

```
<class name="MyClass" identity−type="datastore">
    <datastore−identity strategy="...">
    <datastore−identity/>
     ...
</class>
```

- Para identidad *application*, los metadatos toman la forma:

```
<class name="MyClass" identity−type="application">
    <field  value−strategy="...">
    <field />
     ...
</class>
```

- En donde hay disponibles varias *strategy*:
  - **native** –La opción por omisión y selecciona la estrategia más adecuada para el *datastore*.
  - **sequence** – Utiliza una *sequence-SQL* del *datastore*.
  - **autoassign/identity** – Utiliza la característica *autoassign/identity* del *datastore*.
  - **increment** – Método “neutro” que incrementa una secuencia.
  - **max** – Utiliza un método “$\text{máx}(\text{columna}) + 1$”.
  - **etc. . .**

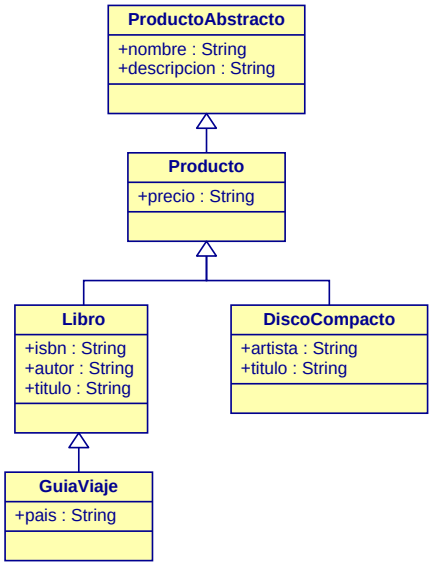


## Proyección de las clases

- Cuando se añade persistencia a una clase es necesario decidir cómo se proyecta en el *datastore*.
- La forma más simple es proyectar cada clase en su propia tabla. Este es el comportamiento normal en JDO.
- No es necesario especificar la tabla ni el nombre de las columnas: JDO proporcionará los nombres adecuados en cada caso.
- Es posible, sin embargo, especificar todos los detalles de la proyección, de modo que:
  - Se puede ligar la clase con un esquema que ya existe en la base de datos.
  - Se pueden crear las tablas indicando el máximo nivel de detalle.

## Implementación de la herencia

- En Java es muy frecuente utilizar la herencia entre clases.
- En JDO existe la posibilidad de decidir cómo se implementa la persistencia de las clases en la jerarquía de herencia. Las elecciones posibles son:
  - Cada clase en su propia tabla del *datastore* (**new-table**).
  - Hacer que la superclase mantenga sus campos en la tabla de sus subclases (**subclass-table**).
  - Subir los campos de las subclases a la tabla que contiene la superclase (**superclass-table**).
- Utilizaremos el siguiente digrama UML para ilustrar las posibilidades:



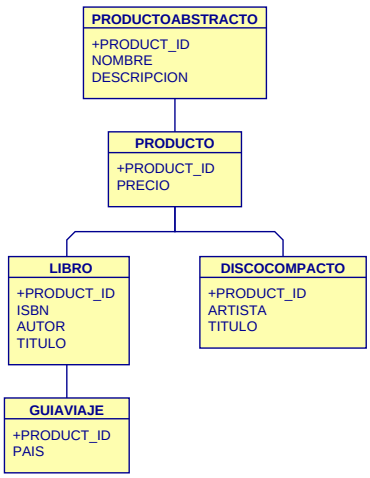
## Herencia – *new-table*

- Utilizamos una tabla separada para cada clase.
  - **Ventajas:** Es la definición de los datos más “normal”.
  - **Inconvenientes:** Peor rendimiento, ya que es necesario acceder a múltiples tablas para recuperar los objetos de un subtipo.

```
<package name="ejemplo.almacen">
  <class name="ProductoAbstracto">
    <inheritance—strategy="new—table"/>
    <field name="nombre">
      <column length="100" jdbc—type="VARCHAR"/>
    </field>
    <field name="descripcion">
      <column length="255" jdbc—type="VARCHAR"/>
    </field>
  </class>
  <class name="Producto"
    persistence—capable—superclass="ejemplo.almacen.ProductoAbstracto">
    <inheritance strategy="new—table"/>
    <field name="price"/>
  </class>
  ...
```

```
<class name="Libro" persistence—capable—superclass="ejemplo.almacen.Producto">
  <inheritance strategy="new—table"/>
  <field name="isbn">
    <column length="20" jdbc—type="VARCHAR"/>
  </field>
  <field name="autor">
    <column length="40" jdbc—type="VARCHAR"/>
  </field>
  <field name="titulo">
    <column length="40" jdbc—type="VARCHAR"/>
  </field>
</class>
<class name="GuiaViaje" persistence—capable—superclass="ejemplo.almacen.Libro">
  <inheritance strategy="new—table"/>
  <field name="pais">
    <column length="40" jdbc—type="VARCHAR"/>
  </field>
</class>
<class name="DiscoCompacto" persistence—capable—superclass="...">
  <inheritance strategy="new—table"/>
  <field name="artista">
    <column length="40" jdbc—type="VARCHAR"/>
  </field>
  <field name="titulo">
    <column length="40" jdbc—type="VARCHAR"/>
  </field>
</class>
```

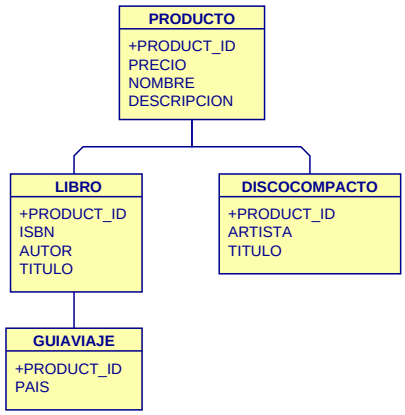
- En la base de datos, cada clase en la jerarquía de herencia se representa en su propia tabla.
- Las tablas de las subclases cuentan con una clave foránea entre su clave primaria y la clave primaria de la tabla de la superclase.



## Herencia – *subclass-table*

- En algunos casos es conveniente definir la persistencia de una clase en la tabla de sus subclases.
- Esto es especialmente cierto cuando se dispone de una clase base abstracta y carece de sentido disponer de una tabla separada para esa clase (siempre vacía).

```
<package name="ejemplo.almacen">
  <class name="ProductoAbstracto">
    <inheritance strategy="subclass—table"/>
    <field name="name">
      <column length="100" jdbc—type="VARCHAR"/>
    </field>
    <field name="description">
      <column length="255" jdbc—type="VARCHAR"/>
    </field>
  </class>
  ...
```




- Cuando se inserta en la base de datos un objeto `GuiaViaje`, se produce la inserción de una tupla en las tablas `PRODUCTO`, `LIBRO` y `GUIAVIAJE`
- ¡Cuidado con las relaciones embebidas!

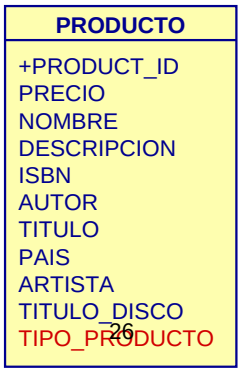
## Herencia – *superclass-table*

- Las tablas de las subclases se almacenan en la tabla de la superclase.
- Los atributos de las subclases tienen que embeberse en la tabla de la superclase.
- **Ventajas:**
  - Es posible recuperar un completo objeto accediendo sólo a una tabla.
- **Desventajas:**
  - La tabla resultante puede contener un gran número de columnas, lo que dificulta la legibilidad de la base de datos y en algunos casos degradar el rendimiento.
  - Es necesario añadir una columna para discriminar los diferentes tipos de objetos.
- Existen varias formas de realizar la compactación de la herencia. Veremos la más general:
  - Utilizaremos un discriminador (columna) `TIPO_PRODUCTO`.
  - Definiremos para cada clase el “valor” que puede tomar esa columna.

```
<class name="Producto">
  <inheritance strategy="new-table">
    <discriminator strategy="value-map" value="PRODUCTO">
      <column name="TIPO_PRODUCTO"/>
    </discriminator>
  </inheritance>
  <field name="precio"/>
</class>
<class name="Libro" persistence-capable-superclass="ejemplo.almacen.Producto">
  <inheritance strategy="superclass-table">
    <discriminator value="LIBRO"/>
  </inheritance>
  <field name="isbn">
    <column length="20" jdbc-type="VARCHAR"/>
  </field>
  <field name="author">
    <column length="40" jdbc-type="VARCHAR"/>
  </field>
  <field name="title">
    <column length="40" jdbc-type="VARCHAR"/>
  </field>
</class>
...
```

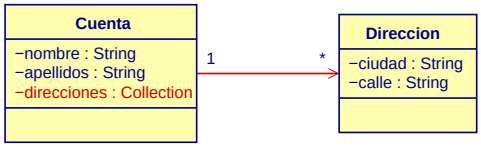
```
<class name="GuiaViaje" persistence-capable-superclass="ejemplo.almacen.Libro">
  <inheritance strategy="superclass-table">
    <discriminator value="TRAVELGUIDE"/>
  </inheritance>
  <field name="country">
    <column length="40" jdbc-type="VARCHAR"/>
  </field>
</class>
<class name="DiscoCompacto" persistence-capable-superclass="...">
  <inheritance strategy="superclass-table">
    <discriminator value="DISCOCOMPACTO"/>
  </inheritance>
  <field name="artista">
    <column length="40" jdbc-type="VARCHAR"/>
  </field>
  <field name="titulo">
    <column name="TITULO_DISCO" length="40" jdbc-type="VARCHAR"/>
  </field>
</class>
```

- Es importante notar que ha sido necesario renombrar el atributo `DiscoCompacto.titulo` como `TITULO_DISCO` sobre la tabla de la base de datos.
- De otra forma, el nombre coincidiría con el atributo `Libro.titulo`.
- El resultado final se muestra en la figura:

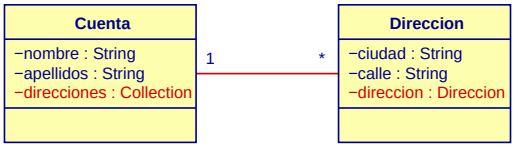


## Relaciones

- Las relaciones se caracterizan por su cardinalidad.
  - Relaciones 1-a-1, 1-a-$N$ y $M$-a-$N$.
- Las relaciones también pueden ser unidireccionales o bidireccionales:
  - En una relación unidireccional, una instancia `A` se relaciona con una instancia `B` pero no al contrario. La relación sólo puede recorrerse desde `A` hasta el extremo `B`.

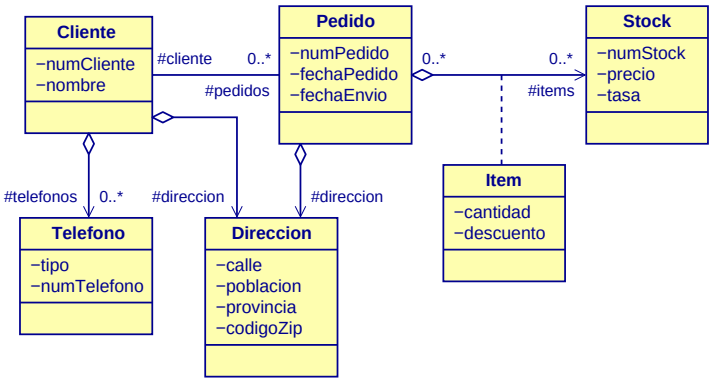



  - Las relaciones bidireccionales son recíprocas; cada una hace referencia a la otra. Si existe una relación bidireccional entre las clases `A` y `B`, entonces la clase `A` hace referencia a `B` y `B` hace referencia a `A`.

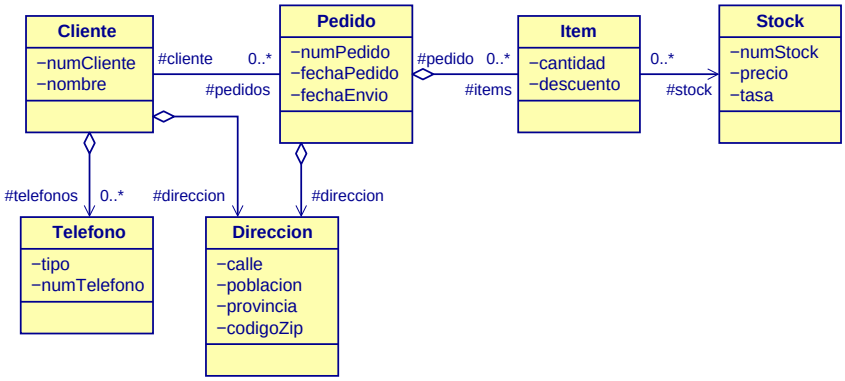


## Gestión de pedidos

- Diagrama de clases UML original:




- Diagrama de clases UML modificado:



## La clase *Teléfono*

```
package dbd.sgp.bussines;
public class Telefono {
  private String tipo;
  private String telefono;
  public Telefono(String ti, String tl) {
    tipo = ti;
    telefono = tl;
  }
  public String getTipo() { return tipo; }
  public String getTelefono() { return telefono; }
  public void setTipo(String ti) { tipo = ti; }
  public void setTelefono(String tl) { telefono = tl; }
  public String toString() { return ""+tipo+": "+telefono; }
}
```



## La clase *Dirección*

```
package dbd.sgp.bussines;
public class Direccion {
  private String calle, poblacion, provincia, codigoZip;
  public Direccion(String cl, String po, String pr, String cz) {
    calle = cl;
    poblacion = po;
    provincia = pr;
    codigoZip = cz;
  }
  public String getCalle() { return calle; }
  public String getPoblacion() { return poblacion; }
  public String getProvincia() { return provincia; }
  public String getCodigoZip() { return codigoZip; }
  public void setCalle(String cl) { calle = cl; }
  public void setPoblacion(String po) { poblacion = po; }
  public void setProvincia(String pr) { provincia = pr; }
  public void setCodigoZip(String cz) { codigoZip = cz; }
  public String toString() { return ""+calle+". "+codigoZip+"—"
    +poblacion+" ("+provincia+")"; }
}
```

## La clase *Stock*

```
package dbd.sgp.bussines;
public class Stock implements Comparable {
  private int numStock;
  private String descripcion;
  private double precio;
  private double tasa;
  // Constructor, getters y setters
  ...
  public String toString() {
    return numStock + "; " + descripcion + "; Precio: " + precio +
    "; I.V.A: " + 100.0*tasa + "%";
  }
  public int compareTo(Object o) {
    Stock s;
    if (o instanceof Stock) {
      s = (Stock)o;
      return numStock - s.numStock;
    }
    else
      return 0;
  }
}
```



## La clase *Item*

```
package dbd.sgp.bussines;
public class Item {
    private int cantidad;
    private double descuento;
    protected Pedido pedido;
    protected Stock stock;
    // Constructores, getters y setters
    ...
    public String toString() {
      return stock.getNumStock() + " - " + stock.getDescripcion() +
        "; Precio: " + stock.getPrecio() + "; Tasa: " +
        100.0*stock.getTasa() + "%" + "; Cantidad: " + cantidad +
        "; Descuento: " + 100.0*descuento + "%";
    }
  }
```

## La clase *Pedido*

```
package dbd.sgp.bussines;
import java. util .*;
import java.text .*;
public class Pedido implements Comparable {
  private int numPedido;
  private Calendar fechaPedido;
  private Calendar fechaEnvio;
  protected Cliente cliente ;
  protected Direccion direccion;
  protected Collection<Item> items = new ArrayList<Item>();
  // Constructores, getters y setters
  ...
  public Collection<Item> getItems() { return items; }
  ...
  public void addItem(Item i) { items.add(i); }
  public void removeItem(Item i) { items.remove(i); }
  public int compareTo(Object o) {
    ...
  }
  public void printPedido() {
    ...
  }
}
```



## La clase *Cliente*

```
public class Cliente implements Comparable {
  private int numCliente;
  private String nombre;
  protected Direccion direccion;
  protected Collection<Telefono> telefonos = new HashSet<Telefono>();
  protected Collection<Pedido> pedidos = new HashSet<Pedido>();
  public Cliente(String n) { nombre = n; }
  // getters
  ...
  public Collection<Telefono> getTelefonos() { return telefonos; }
  public Collection<Pedido> getPedidos() { return pedidos; }
  // setters
  ...
  public void addTelefono(Telefono tlfno) { telefonos.add(tlfno); }
  public void addPedido(Pedido p) { pedidos.add(p); }
  public int compareTo(Object o) {
    ...
  }
  ...
}
```

## El fichero *package.jdo*

```
<?xml version="1.0"?>
<!DOCTYPE jdo PUBLIC
  "-//Sun Microsystems, Inc.//DTD Java Data Objects Metadata 2.0//EN"
  "http://java.sun.com/dtd/jdo_2_0.dtd">
<jdo>
  <package name="dbd.sgp.bussines">
  <!--
    # Telefono: identidad controlada por JDO.
    #
    # Atributos:
    #     tipo          (varchar).
    #     telefono      (varchar).
    # Relaciones: ninguna
    #
  -->
  <class name="Telefono" identity-type="datastore">
    <inheritance strategy="new-table"/>
    <field name="tipo" persistence-modifier="persistent"/>
    <field name="telefono" persistence-modifier="persistent"/>
  </class>
  ...
```

```
  <!--
    # Stock: identidad controlada por la aplicacion y generada por
    #          la base de datos.
    #
    # Atributos:
    #     numStock      (integer PK, AutoInc).
    #     descripcion   (varchar).
    #     precio,tasa   (double).
    # Relaciones: ninguna
    #
  -->
  <class name="Stock" identity-type="application">
    <inheritance strategy="new-table"/>
    <field name="numStock" persistence-modifier="persistent" primary-key="true"
      value-strategy="autoassign"/>
    <field name="descripcion" persistence-modifier="persistent"/>
    <field name="precio" persistence-modifier="persistent"/>
    <field name="tasa" persistence-modifier="persistent"/>
  </class>
  ...
```

```
  <!--
    # Direccion: identidad controlada por JDO.
    #
    # Atributos:
    #     calle         (varchar).
    #     poblacion     (varchar).
    #     provincia     (varchar).
    #     codigoZip     (varchar).
    # Relaciones: ninguna
    #
  -->
  <class name="Direccion" identity-type="datastore">
    <inheritance strategy="new-table"/>
    <field name="calle" persistence-modifier="persistent"/>
    <field name="poblacion" persistence-modifier="persistent"/>
    <field name="provincia" persistence-modifier="persistent"/>
    <field name="codigoZip" persistence-modifier="persistent"/>
  </class>
  ...
```

```
  <!--
    # Cliente: identidad controlada por la aplicacion y generada por
    #            la base de datos.
    #
    # Atributos:
    #     numCliente    (integer PK, AutoInc).
    #     nombre        (varchar).
    # Relaciones:
    #     direccion     (1:1 unidireccional ->)
    #     telefonos     (1:N unidireccional ->)
    #     pedidos       (1:N bidireccional )
    #
  -->
  <class name="Cliente" identity-type="application">
    <inheritance strategy="new-table"/>
    <field name="numCliente" persistence-modifier="persistent"
      primary-key="true" value-strategy="autoassign"/>
    <field name="nombre" persistence-modifier="persistent"/>
    <field name="direccion" persistence-modifier="persistent"/>
    <field name="telefonos" persistence-modifier="persistent">
      <collection element-type="dbd.sgp.bussines.Telefono"/>
    </field>
    <field name="pedidos" persistence-modifier="persistent"
      mapped-by="cliente">
      <collection element-type="dbd.sgp.bussines.Pedido"/>
    </field>
  </class>
```

```
  <!--
    # Pedido: identidad controlada por la aplicacion y generada por
    #           la base de datos.
    #
    # Atributos:
    #     numPedido     (integer PK, AutoInc).
    #     fechaPedido   (date).
    #     fechaEnvio    (date).
    # Relaciones:
    #     cliente       (1:1 bidireccional )
    #     direccion     (1:1 unidireccional ->)
    #     items         (1:N bidireccional )
    #
  -->
  <class name="Pedido" identity-type="application">
    <inheritance strategy="new-table"/>
    <field name="numPedido" persistence-modifier="persistent"
      primary-key="true" value-strategy="autoassign"/>
    <field name="fechaPedido" persistence-modifier="persistent"/>
    <field name="fechaEnvio" persistence-modifier="persistent"/>
    <field name="cliente" persistence-modifier="persistent"/>
    <field name="direccion" persistence-modifier="persistent"/>
    <field name="items" persistence-modifier="persistent" mapped-by="pedido">
      <collection element-type="dbd.sgp.bussines.Item"/>
    </field>
  </class>
```

```
  <!--
    # Item: identidad controlada por JDO.
    #
    # Atributos:
    #     cantidad      (integer).
    #     descuento     (double).
    # Relaciones:
    #     pedido        (N:1 bidireccional )
    #     stock         (N:1 unidireccional ->)
    #
  -->
  <class name="Item" identity-type="datastore">
    <inheritance strategy="new-table"/>
    <field name="cantidad" persistence-modifier="persistent"/>
    <field name="descuento" persistence-modifier="persistent"/>
    <field name="pedido" persistence-modifier="persistent"/>
    <field name="stock" persistence-modifier="persistent"/>
  </class>
</package>
</jdo>
```