Bases de datos II

Bases de datos objeto-relacional

27/12/2007

Francisco Charte Ojeda

Tabla de contenido

1. Conceptos teóricos

Actualmente la práctica totalidad de los lenguajes contemplan el desarrollo de software sobre el paradigma de la orientación a objetos, el cual permite crear modelos que guardan un gran paralelismo con el dominio del problema para el que va a diseñar una solución. El estado de esos objetos, que representa la información sobre la que operan, ha de ser conservada de forma persistente y el soporte seleccionado para ello suele ser una base de datos gestionada por el correspondiente DBMS.

Las bases de datos relacionales clásicas (RDBMS), en las cuales es necesario reducir a tablas formadas por filas y columnas las entidades del mundo real y establecer las relaciones existentes entre éstas mediante el conocido mecanismo de claves primarias/claves externas, tienen limitaciones que no les hace adecuadas para las aplicaciones más complejas. En el extremo opuesto se encuentran las bases de datos orientadas a objetos (OODBMS), con características puras de orientación a objetos. Estos productos cuentan con un lenguaje de definición de objetos similar al IDL (*Interface Definition Language*) usado en estándares como CORBA, así como un lenguaje de consulta específico para operar con objetos.

A medio camino entre los RDBMS y los OODBMS se encuentran los ORDBMS, sistemas de gestión de bases de datos que combinan el modelo relacional con ciertas características de la orientación a objetos. Los productos ORDBMS, entre los que se encuentran las últimas versiones de Oracle, almacenan la información como cualquier RDBMS, es decir, en forma de tablas, filas y columnas y no como objetos propiamente dichos, como sí hacen los OODBMS. No obstante incorporan extensiones a SQL que facilitan el tratamiento de los datos como si fuesen objetos, ofreciendo un cierto nivel de abstracción que permite diseñar soluciones de mayor complejidad con menos esfuerzo.

1.1. Características ORDBMS de Oracle

Oracle cuenta con una serie de construcciones en el lenguaje PL/SQL que permiten definir tipos de objetos, estos tipos serían equivalentes a las clases de C++ o Java, formados por atributos y métodos. Dichos tipos pueden formar jerarquías, actuando unos como base para definir otros más complejos y específicos, lo cual sería equivalente al mecanismo de herencia típico de la OOP.

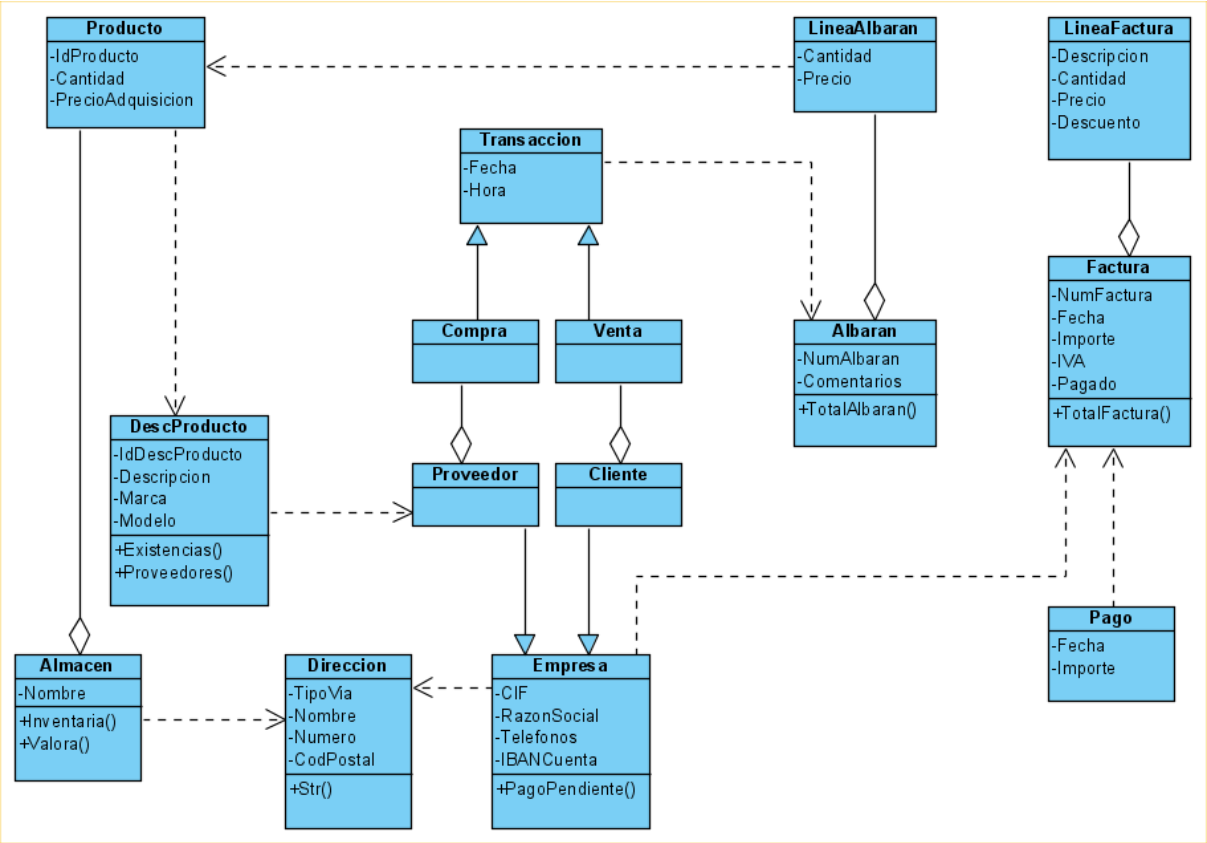
Mediante las referencias a tipos y las tablas anidadas de Oracle se modelan tanto las agregaciones como las composiciones, dos patrones muy usuales al modelar objetos. Las referencias actúan también como punteros, de forma que un atributo de un cierto tipo de objeto puede hacer referencia a otro objeto, sin importar dónde se encuentre éste almacenado (si no se limita el ámbito).

La asociación de métodos a los tipos, con código que actúa sobre los objetos, aporta una funcionalidad que los datos de un RDBMS clásico difícilmente pueden alcanzar, a pesar de que puedan crearse funciones, procedimientos almacenados y disparadores.

En conjunto, esta serie de características hacen posible que el modelo orientado a objetos de una solución software se convierta en un modelo de base de datos con bastante agilidad, sin necesidad de descomponer cada uno de los objetos en una o más tablas, preservar las relaciones lógicas mediante claves, etc.

1.2. Modelo lógico de la solución

El primer paso en la fase de diseño de la base de datos objeto-relacional para la aplicación de facturación ha sido construir el modelo lógico, para lo cual he usado un diagrama de clases en el que he identificado los objetos necesarios, con sus atributos y métodos, y las relaciones existentes entre ellos. Hay especializaciones, dependencias y composiciones, pero ningún detalle específico relativo al modelo de datos. El diagrama de clases usado como referencia es el siguiente:



Las clases `Proveedor` y `Cliente` son una especialización de `Empresa`, diferenciándose una de otra en que la primera contiene una colección de objetos `Compra` y el segundo de objetos `Venta`, especializaciones ambos del objeto `Transaccion`. La clase `Proveedor` mantiene, además, una referencia a los productos que ofrece.

Cada `Transaccion` lleva asociado un `Albaran` de entrega, compuesto de una serie de líneas representadas por `LineaAlbaran`. De forma análoga, una `Factura` se compone de un conjunto de `LineaFactura`, existiendo una dependencia entre facturas y las empresas a las que pertenecen.

Las líneas de albarán hacen referencia a objetos `Producto`, con los que se representan los lotes de cada tipo de producto que existen en un determinado almacén. Los almacenes se representan con objetos `Almacen`, conteniendo cada uno de ellos una colección de lotes de productos.

La clase `Pago` sirve para representar los distintos pagos, siempre asociados a una `Factura` y que pueden corresponder al total de ésta o bien ser un pago parcial. El total de la factura se obtendría a partir de las líneas de factura, que contienen cantidad y precio de los productos entregados, obtenidos a partir del albarán, y del descuento asociado a cada línea.

A medida que se registren transacciones (ventas o compras) se irán actualizando los lotes existentes en los almacenes y, al tiempo, generando los albaranes de entrega. Éstos representarían las mercancías entregadas pero pendientes de facturar. En el momento en que se facturasen generarían facturas pendientes de cobro o pago, que pasarían a estar pagadas cuando se registrase el pago asociado.

La clase `Direccion` es la más simple y es de tipo auxiliar, empleándose para establecer la dirección de las empresas y almacenes. Las empresas cuentan, además, con un atributo multivalor: `Telefonos`, que debe permitir almacenar varios teléfonos.

1.3. Obtención del modelo de datos

Si fuésemos a utilizar un esquema relacional clásico, partiendo del anterior diagrama de clases habría que obtener un diagrama entidad-relación, obtener el modelo lógico de datos, normalizar, etc. A los atributos establecidos en un principio habría que agregar otros que permitiesen mantener las relaciones existentes, en forma de claves primarias y claves externas.

Al aprovechar las características ORDBMS de Oracle, sin embargo, esos pasos no son necesarios y la traducción es casi inmediata. Los criterios seguidos para obtener el modelo de datos han sido los siguientes:

- Cada una de las clases que aparecen en el diagrama de clases previo se ha convertido en un tipo de objeto PL/SQL, para lo cual se ha utilizado la construcción `CREATE TYPE AS OBJECT`.
- Aquellas clases que aparecen como especializaciones de otras tienen un tratamiento especial, definiéndose como derivadas (`UNDER` en Oracle) y marcándose las que actúan como base con el modificador `NOT FINAL`.
- Los métodos de las clases se convierten en funciones miembro de los tipos de objeto, implementados en PL/SQL con `CREATE TYPE BODY`.
- Ciertas dependencias entre tipos se convierten sencillamente en atributos de tipo complejo. Es lo que ocurre, por ejemplo, con el atributo `Direccion` de `Almacen` y `Empresa`, cuyo tipo no será básico sino un objeto con varios atributos y un método propio.

- Para mantener las relaciones entre objetos no se agregarán claves primarias y claves externas, sino que se utilizarán los OID que identifican de manera única (en toda la base de datos) a cada objeto de cada tipo, recurriendo para ello a atributos de tipo referencia (`REF tipo`).
- Los atributos multivalorados con un límite concreto en el número de elementos, como es el caso de `Telefonos`, se implementarán como un tipo `VARRAY` de Oracle.
- Las composiciones/agregaciones se traducirán en tablas anidadas de objetos. El tipo correspondiente a la clase `Factura`, por ejemplo, contendrá en una tabla anidada todas las líneas de factura. Sería una composición. En otros casos esa tabla anidada contendrá referencias a objetos almacenados externamente, como es el caso de los proveedores respecto de las descripciones de los productos. Sería una agregación.

Una vez definidos los distintos tipos de objetos, se procederá a crear las tablas a partir de ellos. En determinados casos esa creación agregará algún elemento más, como puede ser el establecimiento de un atributo como clave primaria para facilitar la búsqueda. Es lo que ocurre con las empresas, referenciadas mediante su CIF, o con las descripciones de productos, que suponemos tendrán asociado un código de barras o similar. Las únicas claves primarias definidas son aquellas que por lógica existirían en el mundo real, como el citado CIF de una empresa o el número de una factura, pero no generan claves artificiales que permitan relacionar los objetos, usando en su lugar los OID y el tipo referencia de Oracle.

1.4. Explotación de la base de datos

Oracle cuenta con una serie de extensiones que permiten operar desde sentencias SQL con bases de datos que siguen el diseño objeto-relacional, de forma que las aplicaciones que vayan a explotar esta base de datos únicamente tienen que conocer algunos detalles específicos de este sistema de base de datos.

Cada una de las tuplas de las tablas que se crean es tratada como un objeto, identificada de manera única con su OID, accediéndose a sus atributos y métodos con la habitual notación `objeto.miembro`. Es una notación habitual en SQL: `tabla.columna`. Esta notación se extiende también a los atributos de una tupla que no sean simples, sino objetos como es el caso de `Direccion`.

Al ejecutar consultas, en caso de que un atributo de los seleccionados sea una referencia a un objeto será preciso utilizar la función `VALUE` a fin de indicar a Oracle que utilice ese OID para localizar el objeto y poder acceder al mismo. De manera análoga, cuando se necesite obtener una referencia a un objeto (su OID), habrá que utilizar la función `REF`. Suele ser habitual al insertar o actualizar datos en una tabla que cuenta con atributos que son referencias. La referencia puede obtenerse mediante una consulta, si el objeto ya existe en una tabla, o bien en el mismo momento en que se haya insertado, con el modificador `RETURNING REF INTO` de la sentencia `INSERT`.

La inserción de atributos que sean objetos deberá pasar por el uso de los constructores con que cuentan todos los tipos, facilitando como argumentos los atributos de ese tipo de objeto. Esto es aplicable también para atributos multivalor (`VARARRAY`) y los que sean tablas anidadas, casos en los cuales la lista de parámetros puede quedar vacía para no introducir inicialmente ningún valor.

Tanto la selección como la inserción de valores en tablas anidadas de tuplas ya existentes, por ejemplo existe un albarán al que se quiere añadir una línea o del que es preciso obtener sus líneas, hará uso del operador `THE (consulta)`. Éste ejecuta la consulta indicada entre paréntesis para seleccionar la tabla anidada destino del la sentencia SQL, por ejemplo: `INSERT INTO THE (SELECT Productos FROM Proveedor ..) VALUES(..)`. Esta sentencia seleccionaría el atributo `Productos` de la tabla `Proveedor`, atributo que es una tabla anidada, e introduciría en ella la lista de valores indicada.

A fin de ocultar esta relativa complejidad de uso de la base de datos, y que el diseñador de la aplicación no tuviese que conocer estas especificidades de Oracle, lo adecuado sería definir una serie de vistas y procedimientos almacenados que ofreciesen un esquema lógico de la información y se encargasen de todas las operaciones usuales: obtención de una factura y sus filas, registro de un nuevo proveedor, realización de una compra o venta, etc. En el desarrollo de este trabajo, no obstante, me he centrado en el uso de las cualidades objeto-relacional de Oracle, más que en conseguir una implementación completa de la lógica que se ejecutaría en el servidor y que simplificaría el desarrollo de las aplicaciones.

2. Implementación

Para llevar a cabo la implementación de esta base de datos objeto-relacional se ha partido del diagrama de clases de la página 4, empleando como herramientas un editor de textos simple (el Bloc de notas de Windows), la consola de Oracle para ejecutar los guiones y el capítulo *9. Object Types* del manual *PL/SQL User's Guide and Reference* como principal material de consulta.

En los siguientes apartados se describe el contenido de cada uno de los guiones, mostrando el código PL/SQL y facilitando los detalles que he creído apropiados sobre cada uno de los tipos, tablas, métodos, etc.

2.1. Definición de tipos

El archivo `DefineTipos.sql` contiene todas las definiciones de tipos de objetos, tanto los principales, que pueden deducirse del diagrama de clases de la página 4, como los secundarios que permiten modelar las agregaciones y composiciones.

TDireccion

La dirección es un objeto que va asociado a las empresas (clientes y proveedores) y también a los almacenes con que pueda contar la empresa que utiliza la aplicación. Cuenta con cuatro atributos básicos y un método cuya finalidad es facilitar una cadena con la dirección completa:

```
create or replace type TDireccion as object (
  TipoVia  varchar(5),
  NombreVia     varchar(40),
  Numero   integer,
  CodPostal     varchar(5),
  member function Str return varchar
);
```

TTelefonos

Una empresa puede tener varios números de teléfono (fijos y móviles), por lo que defino un tipo que permita contener hasta 5 teléfonos como máximo. Éste no es un tipo de objeto propiamente dicho, sino que sería equivalente a una vector o *array* de C. Lo interesante es que puede ser utilizado como tipo de un atributo.

```
create type TTelefonos as varray(5) of varchar(10);
```

TFacturas

Aunque el tipo de objeto `TFactura` aún no se ha definido, una declaración avanzada del tipo `create type TFactura;` nos permite utilizarlo para crear otros que dependen del mismo, como es el caso de `TFacturas`. Este tipo es una tabla de referencias a facturas, como puede verse a continuación:

```
create or replace type TFacturas as table of ref TFactura;
```

TEmpresa

Existe un tipo genérico para todas las empresas, ya sean clientes o proveedores, ya que todas ellas cuentan con una dirección, una lista de teléfonos, una cuenta bancaria a través de la que se efectuarían los pagos o se girarían los cobros y una colección de facturas recibidas o emitidas. Esa colección es de referencias a objetos `Factura`, es decir, se trata de una agregación. El método `PagoPendiente()` devolverá el importe que queda por pagar o cobrar de esta empresa. El tipo, como puede verse al final, está marcado como `not final`, lo que indica que no se ha diseñado para crear objetos del mismo sino para crear otros tipos tomándolo como base:

```
create or replace type TEmpresa as object (
  CIF      varchar(10),
  RazonSocial   varchar(60),
  Direccion     TDireccion,
  Telefonos     TTelefonos,
  IBANCuenta    varchar(20),
  Facturas      TFacturas,
  member function PagoPendiente return real
) not final;
```

TListaProveedores

Este tipo auxiliar será el tipo del valor de retorno del método `Proveedores()` del tipo `TDescProducto`: una tabla de cadenas de caracteres con la denominación de las empresas que proveen un cierto producto. En este caso se utiliza un tipo tabla, en lugar de un `VARARRAY`, porque no se sabe cuántos proveedores pueden devolverse y no tiene sentido limitar dicho número, como se hace con los teléfonos donde con cinco posibles es más que suficiente:

```
create or replace type TListaProveedores
     as table of varchar(60);
```

TDescProducto

Cada uno de los productos con que trabaje la empresa tendrá una serie de atributos genéricos, que son los que se recogen en este tipo de objeto, existiendo uno o más lotes disponibles en los distintos almacenes y adquiridos a un cierto precio, datos estos últimos que se agruparían en otro tipo. Los atributos de `TDescProducto`, cuatro en total, son simples cadenas de caracteres. El método `Existencias()` obtiene las existencias que hay de un tipo de producto, localizando todos los lotes disponibles. El método `Proveedores` facilitará una lista de los proveedores a los que puede recurrirse para adquirir este producto:

```
create or replace type TDescProducto as object (
  IdDesProducto varchar(10),
  Marca         varchar(20),
  Modelo   varchar(20),
  Descripcion   varchar(60),
  member function Existencias(Pr ref TDescProducto)
      return integer,
  member function Proveedores(Pr ref TDescProducto)
      return TListaProveedores
);
```

TProducto

De cada tipo de producto pueden existir varios lotes, adquiridos a un cierto precio y con una cantidad concreta disponible. Este tipo sería el utilizado para representar esos lotes, siendo el único aspecto destacable el hecho de que cada producto cuenta con un atributo que hace referencia a la descripción genérica que le corresponde

```
create or replace type TProducto as object (
  IdProducto    varchar(10),
  DesProducto   ref TDescProducto,
  Cantidad      integer,
  PrecioAdq     real
);
```

Creo que merece la pena destacar la diferencia existente entre la columna `Direccion` del tipo `TEmpresa` y la columna `DesProducto` del tipo `TProducto`, ya que pueden parecer iguales pero existe una diferencia realmente importante: la palabra `ref` delante del tipo de esta última. Con ella se indica que la descripción asociada a cada producto no está contenida en los objetos `TProducto`, almacenándose únicamente una referencia a un objeto `TDescProducto`.

A la hora de generar las tablas de la base de datos a partir de estos tipos, la columna `Direccion` de una empresa contendría realmente los datos de la dirección: tipo de vía, nombre, número, etc., información que no sería compartida con ningún otro objeto ya que es altamente improbable que haya varias empresas o almacenes exactamente en la misma dirección. En la columna `DesProducto` de un lote de productos, por el contrario, se guardaría una referencia a un objeto `TDescProducto` que estaría almacenado independientemente, en otra tabla, lo cual favorece que esa información pueda ser compartida entre varios lotes de productos del mismo tipo.

Si en la tabla `TProducto` el atributo `DesProducto` fuese de tipo `TDescProducto`, en lugar de `ref TDescProducto`, estaría duplicándose información de manera innecesaria desde el momento en que se tuviesen dos o más lotes del mismo producto.

Este mismo razonamiento es extensible a los demás tipos de objeto, como pueden ser `TFactura` y `TAlbaran` y las líneas asociadas, unas líneas que forman parte exclusivamente de una cierta factura o albarán, por lo que no tiene sentido que se almacenen de manera independiente a fin de facilitar que puedan ser compartidas.

TProductos

Los distintos lotes de productos se guardarán en forma de tablas anidadas en los almacenes donde se encuentren, siendo este tipo el que representaría la lista de lotes de productos de un almacén:

```
create or replace type TProductos as table of TProducto;
```

TAlmacen

Este tipo representa cada uno de los almacenes con que puede contar la empresa y que estarán identificados por un nombre, contando con dos atributos adicionales de tipo complejo: `Direccion` y `Productos`. Este último contendrá los lotes de productos que existan en el almacén, siendo el primer ejemplo de composición de este diseño. El atributo `Productos` no contiene un único objeto, ni una referencia a un objeto, sino una colección de objetos completa que, en el momento de la creación de la tabla asociada, se almacenarán en una tabla anidada, no en una estructura independiente. Toda la gestión de esa colección: creación, adición de elementos, extracción y borrado, recae en el propio tipo `TAlmacen`, como ocurre en una composición según el diseño orientado a objetos.

El método `Inventaria()` contabilizará el número total de productos que hay en el almacén, mientras que `Valora()` calculará el valor de coste de todos esos productos según el precio de adquisición de cada lote.

```
create or replace type TAlmacen as object (
  Nombre        varchar(20),
  Direccion     TDireccion,
  Productos     TProductos,
```

```
  member function Inventaria return integer,
  member function Valora return real
);
```

TLineaAlbaran

Los albaranes se generan con cada transacción que exista en la empresa, ya sea una compra o una venta, registrando los productos que se recogen o entregan y que quedan pendientes de facturar en ese momento. Un mismo albarán constará de múltiples líneas y este tipo de objeto sería el que representaría cada una de ellas, almacenando la referencia al producto, la cantidad y el precio.

```
create or replace type TLineaAlbaran as object (
  Producto ref TDescProducto,
  Cantidad integer,
  Precio   real
);
```

En este caso `Producto` es una referencia a un `TDescProducto`, es decir, a la descripción del producto. Ese producto se tomará de un lote que está en un cierto almacén y que debe reducirse o incrementarse en una cierta cantidad de unidades.

TLineasAlbaran

Cada albarán se compondrá de una serie de líneas, una colección de objetos `TLineaAlbaran` que es la representada por este tipo:

```
create or replace type TLineasAlbaran
          as table of TLineaAlbaran;
```

TAlbaran

Este tipo representa lo que sería un albarán de entrega (compra o venta) en el que se recogerán una o más líneas, indicando en cada una el producto, cantidad y precio, y opcionalmente unos comentarios. El método `TotalAlbaran()` se encargará de totalizar el albarán sin tener en cuenta, puesto que no es aún una factura, ni el IVA ni los posibles descuentos:

```
create or replace type TAlbaran as object (
  NumAlbaran    varchar(10),
  Lineas        TLineasAlbaran,
  Comentarios   varchar(200),
  member function TotalAlbaran return real
);
```

TLineaFactura

Una línea de factura se genera a partir de una línea de albarán, de la que toma la descripción del producto, la cantidad y precio. En cuanto al descuento, se recoge por línea de factura el importe descontado, contemplando así la posibilidad de que se efectúen descuentos diferentes para cada producto dentro de una misma factura:

```
create or replace type TLineaFactura as object (
  Descripcion   varchar(60),
  Cantidad      integer,
  Precio        real,
  Descuento     real
);
```

TLineasFactura

Este tipo reúne las líneas de factura de cada factura en una colección, de manera análoga a como se hace con las líneas de albarán:

```
create or replace type TLineasFactura
     as table of TLineaFactura;
```

TFactura

Cada factura lleva asignado un número (lo establece quien la emite, como en el caso de los albaranes) y en ella figuran una fecha, el importe y el IVA, así como el conjunto de líneas con los detalles: producto, cantidad, precio y descuento. Las líneas de factura, como ocurre con las de albarán, pertenecen al objeto `TFactura` al ser una composición, ya que no tienen sentido fuera de ese contexto.

El tipo de IVA aplicable a cada empresa puede diferir, por lo que el atributo `IVA` lo que almacenaría sería el importe. El atributo `Pagado`, que no se reflejaría en la factura física, contendría el importe pagado hasta el momento de esa factura, que no tiene necesariamente que ser el total. El método `TotalFactura()` devolvería el total de la factura:

```
create or replace type TFactura as object (
  NumFactura    varchar(10),
  Fecha         date,
  Importe  real,
  IVA      real,
```

```
  Pagado   real,
  Lineas   TLineasFactura,
  member function TotalFactura return real
);
```

TTransacion

Este tipo representa cualquier tipo de transacción, ya sea una compra o una venta, asociándola con un cierto albarán de entrega o recepción y dándole una fecha:

```
create or replace type TTransacion as object (
  Fecha    date,
  Albaran  ref TAlbaran
);
```

TCompras y TVentas

Las compras que realizamos a un proveedor o las ventas efectuadas a un cliente son conjuntos de transacciones, de ahí que se definan como colecciones de objetos `TTransacion`:

```
create or replace type TCompras as table of TTransacion;
create or replace type TVentas as table of TTransacion;
```

TListaProductos

Cada proveedor tendrá asociada una lista de productos que nos ofrece, pero las descripciones de los productos, como se explicó anteriormente, son compartidas porque un mismo producto podría obtenerse de más de un proveedor, por eso este tipo es una tabla de referencias a `TDescProducto` y no de objetos `TDescProducto`:

```
create or replace type TListaProductos
     as table of ref TDescProducto;
```

TProveedor

Este tipo está derivado de `TEmpresa`, por lo que cuenta con una serie de atributos que hereda de esa base y a los que añade la lista de productos que ofrece el proveedor y la colección de transacciones de compra que le hemos hecho:

```
create or replace type TProveedor under TEmpresa (
  Productos     TListaProductos,
  Compras       TCompras
);
```

TCliente

Al igual que el anterior, este tipo de objeto deriva de `TEmpresa` y lo único que agrega es la colección de transacciones de venta que se hayan hecho con cada cliente:

```
create or replace type TCliente under TEmpresa (
  Ventas   TVentas
);
```

TPago

Este tipo identifica los pagos a proveedores o los pagos procedentes de clientes, de manera indistinta. Cada pago se identifica por la fecha en que se realiza y la factura a la que va asociado, yendo acompañado de un importe ya que el pago no tiene necesariamente que ser por el total de esa factura:

```
create or replace type TPago as object (
  Fecha    date,
  Factura  ref TFactura,
  Importe  real
);
```

2.2. Implementación de métodos

Por facilidad de mantenimiento, y también con la intención de separar lo que podría considerarse la interfaz de la solución de los detalles sobre cómo está implementada, el código de los métodos, mencionados antes en la definición de los tipos, se ha llevado a un módulo separado llamado `ImplementaCuerpos.sql`.

TDireccion.Str()

La llamada a `Str()` devolverá una cadena con la dirección completa, compuesta mediante concatenación de varios de sus atributos. Es un ejemplo sencillo de cómo asociar un método a un tipo de objeto en Oracle:

```
  member function Str return varchar is
  begin
    return TipoVia || ' ' || NombreVia || ', ' || Numero;
  end;
```

TEmpresa.PagoPendiente()

Este método calcula el pago pendiente total a un proveedor o de un cliente, para lo cual obtiene el total de todas las facturas asociadas a dicha empresa, a continuación el total de los pagos asociados a dichas facturas y devuelve la diferencia. En las consultas puede verse cómo se utiliza la palabra clave `TABLE` para acceder a una tabla anidada que aparece como atributo de otra, así como el operador `VALUE` para convertir la referencia a la factura en un valor y poder así invocar a su método `TotalFactura()`.

La referencia `SELF` permite acceder a los atributos del objeto actual, para el que se está invocando al método `PagoPendiente()`, seleccionando así las facturas que corresponden a esta empresa y no a cualquier otra.

Por último, es interesante observar cómo se comprueba que los pagos recuperados en la segunda consulta correspondan a cada una de las facturas de esta empresa, mediante la condición destacada en negrita. `pa.Factura` es la referencia a la factura en el objeto `TPago`, mientras que `f` es la referencia a cada una de las facturas del proveedor, un alias para `table(p.Facturas)`:

```
member function PagoPendiente return real is
  Total real;
  Pagado real;
begin
  select sum(value(f).TotalFactura()) into Total
  from Proveedor p, table(p.Facturas) f
  where p.CIF=self.CIF;

  select sum(pa.Importe) into Pagado
  from Proveedor p, table(p.Facturas) f, Pago pa
  where p.CIF=self.CIF and pa.Factura = value(f);

  return Total-Pagado;
end;
```

TDescProducto.Existencias()

Este método obtiene las existencias de un cierto tipo de producto, para lo cual suma todas las cantidades disponibles en los distintos almacenes con que cuenta la empresa. Dado que lo que contienen los almacenes es una tabla de objetos `TProducto`, y en cada fila de esta tabla hay una referencia a `TDescProducto`, el filtro de selección ha de comparar esa referencia con la del `TDescProducto` para el que se está invocando el método `Existencias()`.

El parámetro implícito `SELF` representa una instancia del propio tipo, no una referencia, por lo que no puede compararse directamente `p.DesProducto=self`. Podría pensarse en obtener la referencia con `REF`, usando la condición `p.DesProducto=ref(self)`. De hecho fue lo que hice, pero no funcionaba cuando aparentemente debía hacerlo. Probé las distintas posibilidades que se me ocurrieron pero no había forma de obtener la referencia del objeto implícito, hasta que buscando información al respecto supe que Oracle no permite usar `REF` sobre el parámetro `SELF`, siendo más una limitación de PL/SQL que una característica propiamente dicha.

Siguiendo las indicaciones dadas en el capítulo 18 de Oracle PL/SQL Programming, en el apartado 18.5.3 (http://www.unix.org.ua/orelly/oracle/prog2/ch18_05.htm), solucioné el problema agregando a este método una parámetro que fuese la referencia al propio objeto para el que se efectúa la llamada:

```
member function Existencias(Pr ref TDescProducto)
      return integer is

    n integer;

begin

    select sum(p.Cantidad) into n

    from Almacen al, table(al.Productos) p

    where p.DesProducto = Pr;


    return n;

  end;
```

Al invocar a este método, por tanto, será preciso facilitar como argumento la referencia al objeto para el que se le llama, dato que se obtendrá fácilmente con el operador `REF` como puede verse en algunas de las consultas de ejemplo propuestas más adelante. Lo mismo es aplicable para el método siguiente del tipo `TDescProducto`.

TDescProducto.Proveedores()

Facilita los proveedores a los que se puede pedir el producto, en forma de tabla de cadenas de caracteres, para lo cual se efectúa una consulta con algunas peculiaridades. Dado que lo que se quiere recuperar es una lista de valores, no un único valor, se emplea `BULK COLLECT INTO` en lugar de `INTO`, siendo el destino no una variable simple sino un objeto `TListaProveedores`.

Para obtener en esa lista la razón social de los proveedores que ofrecen el producto, es necesario buscar en la tabla de productos que ofrece cada uno de ellos la referencia que corresponde al producto. Es lo que se hace en el apartado `WHERE`, donde se verifica que la referencia `TDescProducto` recibida como parámetro esté en la lista de productos de cada proveedor.

Finalmente, el modificador `UNIQUE` impide que se incluyan en la lista de resultados varias referencias al mismo proveedor.

```
member function Proveedores(Pr ref TDescProducto)
    return TListaProveedores is

  Prov TListaProveedores;

begin

  select unique p.RazonSocial bulk collect into Prov

  from Proveedor p, table(p.Productos) t

  where Pr in (select value(t) from table(p.Productos));


  return Prov;

end;
```

TAlmacen.Inventaria()

Este método efectúa el inventario de un almacén, devolviendo el número de artículos que debería haber en el mismo. Para ello consulta la tabla anidada `Productos` sumando las cantidades de todos los productos disponibles:

```
member function Inventaria return integer is

  n integer;

begin
```

```
  select sum(p.Cantidad) into n
  from Almacen al, table(al.Productos) p
  where al.Nombre=self.Nombre;
  return n;
end;
```

TAlmacen.Valora()

El funcionamiento de este método es muy similar al anterior. La diferencia estriba en que lo se suma no es la cantidad, sino el producto de cada cantidad de producto y su precio de adquisición. De esta forma se sabrá el valor de coste de toda la mercancía que hay en el almacén:

```
member function Valora return real is
  n real;
begin
  select sum(p.cantidad*p.precioadq) into n
  from Almacen al, table(al.Productos) p
  where al.Nombre=self.Nombre;
  return n;
end;
```

TAlbaran.TotalAlbaran()

Este método calcula el total de un albarán, sin incluir IVA ni descuentos, para lo cual toma todas las líneas que lo componen, suma cada cantidad de producto por su precio y suma todos esos valores parciales obteniendo el importe total:

```
member function TotalAlbaran return real is
  n real;
begin
  select sum(li.Cantidad*li.Precio) into n
  from Albaran al, table(al.Lineas) li
  where al.NumAlbaran = self.NumAlbaran;

  return n;
end;
```

TFactura.TotalFactura()

Calcula el total de la factura, sencillamente devolviendo la suma del importe y el IVA correspondiente. Es un método muy simple en el que se muestra cómo operar directamente sobre los valores del objeto actual, sin necesidad de efectuar ninguna consulta:

```
member function TotalFactura return real is
begin
  return self.Importe + self.IVA;
end;
```

2.3. Creación de tablas

El guión `CreaTablas.sql` contiene las sentencias DDL para crear las tablas de la base de datos, utilizando para ellos los tipos definidos previamente. No obstante, no hay una tabla (físicamente sí, pero no desde una perspectiva lógica) para cada tipo ya que muchos de ellos se almacenan en tablas anidadas. Éstas no son accesibles de forma directa, sino únicamente a través de las tuplas de la tabla padre en que está embebida.

DescProducto

Las descripciones de los productos tienen una tabla propia, ya que un mismo producto podríamos obtenerlo de distintos proveedores. El identificador que le asociemos actuará como clave primaria, siendo su finalidad facilitar la selección de un producto a partir de un código de barras o similar:

```
create table DescProducto of TDescProducto (
  IdDesProducto primary key
);
```

Proveedor

La tabla de proveedores permitirá localizar la empresa por su CIF, un dato único para cada empresa, y lleva anidadas tres tablas: una de referencias a los productos que nos ofrece, otra con las compras que le hemos efectuado y la tercera con las referencias a las facturas:

```
create table Proveedor of TProveedor (
  CIF      primary key
)
  nested table Productos store as TblProductos,
  nested table Compras store as TblCompras,
  nested table Facturas store as TblFacturasCompra;
```

Cliente

La tabla de clientes permitirá localizar la empresa por su CIF y lleva anidada la tabla que contiene todas las ventas que hemos efectuado con él y la tabla de facturas:

```
create table Cliente of TCliente (
  CIF      primary key
)
  nested table Ventas store as TblVentas,
  nested table Facturas store as TblFacturasVenta;
```

Pago

La tabla para registrar los pagos es la más simple, ya que no tiene claves ni tablas anidadas:

```
create table Pago of TPago;
```

Factura

La tabla de facturas tiene anidada la de las líneas que le corresponden:

```
create table Factura of TFactura
  nested table Lineas store as TblLinFacturas;
```

Albaran

La tabla de albaranes tiene anidada la de las líneas que le corresponden:

```
create table Albaran of TAlbaran
  nested table Lineas store as TblLinAlbaran;
```

Almacen

La tabla con los almacenes tiene anidada la tabla de lotes de productos que existen en cada almacén:

```
create table Almacen of TAlmacen
  nested table Productos store as TblLotesProductos;
```

Además de las claves primarias y las tablas anidadas, en estas sentencias podrían incluirse también otras restricciones asociadas a los atributos de los distintos tipos, como se haría en una descripción cualquiera de una tabla. En el caso de las referencias, podría utilizarse `SCOPE IS` para asegurar que el objeto referenciado existe en una tabla determinada.

3. Manual de instalación

Para instalar este proyecto lo único que se precisa es una consola SQL de Oracle habiendo iniciado sesión con una cuenta cuyos privilegios le permita ejecutar sentencias DDL a fin de definir los tipos y crear las tablas. Partiendo de esta premisa, a continuación se indican cuáles serían los procedimientos de instalación y desinstalación.

3.1. Instalación

Los pasos a seguir para instalar la base de datos serán los indicados a continuación:

1. Ejecución del guión `DefineTipos.sql` para definir los distintos tipos de objetos que se han descrito anteriormente.
2. Ejecución del guión `CreaTablas.sql` que se encargará de crear las tablas a partir de los datos previos.
3. Ejecución del guión `ImplementaCuerpos.sql` completando así la definición de los tipos con la implementación de los métodos asociados.

Concluida la ejecución de los guiones la base de datos habrá quedado preparada para trabajar, sin contener en principio dato alguno. En el apartado siguiente se describen los procedimientos habituales que podrían llevarse a cabo sobre esta base de datos.

También se facilita un guión `InsertaDatos.sql` que puede ejecutarse completo para introducir en la base de datos la información de los procedimientos de ejemplo explicados en la sección *Manual de usuario*, así como un guión `Consultas.sql` con el código de las consultas.

3.2. Desinstalación

Para facilitar la desinstalación, en caso de que no se haya creado una base de datos vacía en la que crear el proyecto con los guiones anteriores, no hay más que ejecutar el guión `EliminaTipos.sql`. Éste se encarga de eliminar primero las tablas y después los tipos, en el orden adecuado para evitar fallos por las dependencias que existen entre tipos.

4. Manual de usuario

Una vez se haya completado la instalación, creando los tipos de objeto y tablas de la base de datos, llegará el momento de que la aplicación, bien sea directamente o a través de vistas y procedimientos almacenados definidos con tal propósito, ejecuten las tareas propias de la explotación: registro de productos, recepción de artículos procesando los albaranes, preparación de facturas, etc.

En los apartados siguientes describo cómo se llevarían a cabo las operaciones que creo serían más habituales, mostrando el código SQL de ejemplos concretos. Aunque me centro en la adquisición de productos, albaranes y facturas de proveedores, el tratamiento de los equivalentes para clientes serían análogos, puesto que las ventas son idénticas a las compras (una transacción), y los albaranes y facturas también se tratan igual, la diferencia es que apuntarían a una empresa que sería la de un cliente, en lugar de la de un proveedor.

4.1. Inserción de datos

Puesto que la base de datos está inicialmente vacía, las primeras operaciones serán necesariamente de inserción para disponer de información con la que poder trabajar.

Creación de almacenes

Asumiendo que la empresa acaba de instalar la aplicación en su sistema, el primer paso sería establecer los almacenes con los que cuenta. En este ejemplo supondremos que hay dos almacenes:

```
insert into Almacen values (
  'Principal',
  TDireccion('Cl.','Las Cruces',25,'21242'), TProductos());

insert into Almacen values (
   'Pgno. Esperanza',
   TDireccion('Avda.','Granada',4,'18342'), TProductos());
```

De estas sentencias lo destacable es el uso del constructor del tipo `TDireccion`, para definir la dirección de cada almacén, y el constructor de `TProductos` a fin de asociar una lista de productos inicialmente vacía. De esta forma se tienen dos almacenes sin artículos, pero con una lista preparada para recibirlos.

Si al definir un almacén no se crease la lista vacía de productos, con `TProductos()`, sino que se entregase el valor `NULL` o se omitiese el valor para esa columna (que quedaría como `NULL`), antes de poder añadir productos con sentencias `INSERT` como las de puntos posteriores habría que ejecutar una sentencia `UPDATE` sobre cada almacén para crear la lista.

Descripción de algunos productos

El siguiente paso será la inserción en la tabla `DescProducto` de algunas descripciones de productos, mediante sentencias `INSERT` corrientes sin nada en especial ya que esa tabla no cuenta más que columnas de tipos básicos:

```
insert into DescProducto values (
  1, 'Loretto', 'Xire', 'Vestido crep con sobrefalda');

insert into DescProducto values (
  2, 'Kira', 'Dropes', 'Pantalón micropana pinzas');

insert into DescProducto values (
  3, 'Kira', 'Tablas', 'Falda tableada antelina');
```

Registro de proveedores y asociación de productos

Los proveedores y los clientes son entidades prácticamente idénticas, salvo por que los primeros tienen asociada una lista de productos que permite a la empresa saber a qué proveedores puede recurrir cuando precisa un cierto artículo. Esa asociación puede efectuarse en dos pasos independientes, creando primero el proveedor:

```
insert into Proveedor values (
  'B-38299173', 'Mirto S.L.',
  TDireccion('Cl.','Rosas rojas', 7, '43234'),
  TTelefonos('916376274','672817382'),
  '3120-4032-17-4215333', TFacturas(),
  TListaProductos(), TCompras());
```

Una de las columnas es de tipo `TDireccion`, como en el caso de los almacenes. La lista de teléfonos se crea de forma similar, si bien el número de elementos puede variar de 0 a 5. Los tres últimos valores generan listas vacías de facturas, productos y compras asociadas al proveedor.

A continuación tomaríamos la lista de productos asociada a este proveedor y le agregaríamos las referencias a las descripciones de producto que nos ofrezcan, lo que podemos hacer con una sentencia como la siguiente:

```
insert into the (select prv.Productos
                 from Proveedor prv where CIF='B-38299173')
  select ref(d) from DescProducto d
  where d.IdDesProducto <= 2;
```

Con ella seleccionamos de la tabla `DesProducto` las filas cuyo identificador de producto sea igual o inferior a 2, es decir, los dos primeros productos registrados antes. De esas filas se obtiene el identificador con el operador `REF`, una referencia que es lo que se inserta en la tabla `Productos` del proveedor. Al ser ésta una tabla anidada se usa `the` para seleccionarla con una sentencia `SELECT`, ya que no es posible hacer referencia directa a la misma.

Otra posibilidad, a la hora de registrar un proveedor y asociar los productos que ofrece, sería la mostrada en el siguiente bloque PL/SQL:

```
declare

  RefP ref TDescProducto;

begin
```

```
  insert into DescProducto d values (
     4, 'Aixa', 'Plete', 'Camisa doble cuello')
     returning ref(d) into RefP;


  insert into Proveedor values (
    'B-73599774', 'Creaciones Miranda',
    TDireccion('Cl.','Rojas rosas', 7, '23432'),
    TTelefonos('53376274','95817382'),
    '0120-1037-12-0005312', TFacturas(),
    TListaProductos(RefP), TCompras());

end;
```

Al insertar la descripción del producto en la tabla `DescProducto` se recupera la referencia a esa nueva fila mediante la cláusula `RETURNING REF`, referencia que se facilita como argumento al constructor `TListaProductos` en el momento en que se crea el proveedor.

Compra de un lote de productos

Teniendo en la base de datos productos y proveedores que los facilitan, la compra de un lote de productos implicará varios pasos que podrían implementarse como un procedimiento almacenado que recibiese los argumentos. El código estaría basado en el siguiente, en el que se adquieren varias unidades de dos productos a un mismo proveedor y se llevan en parte a un almacén y en parte a otro.

```
declare

  RefAlb  ref TAlbaran;

  RefDes1  ref TDescProducto;

  RefDes2  ref TDescProducto;

begin

  -- Obtengo las referencias a los productos recibidos

  select ref(d) into RefDes1
     from DescProducto d where IdDesProducto = '1';

  select ref(d) into RefDes2
     from DescProducto d where IdDesProducto = '2';
```

```
    -- Se registran los productos en el almacén donde
    -- se han recibido

    insert into the (select al.Productos
        from Almacen al where Nombre='Principal')
      values (TProducto('1', RefDes1, 5, 20.5));

    insert into the (select al.Productos
        from Almacen al where Nombre='Principal')
      values (TProducto('2', RefDes2, 7, 16.35));

    insert into the (select al.Productos
        from Almacen al where Nombre='Pgno. Esperanza')
      values (TProducto('1', RefDes1, 10, 20.5));



    -- Genero el albarán de entrada de los productos

    insert into Albaran alb values (
     '121/07',
     TLineasAlbaran(
       TLineaAlbaran(RefDes1, 15, 20.5),
       TLineaAlbaran(RefDes2, 7, 16.35)
     ), 'Primera compra a este proveedor')
     returning ref(alb) into RefAlb;



    -- Para agregar la nueva transacción

    insert into the(select Compras
        from Proveedor where CIF='B-38299173')
      values (TTransacion(sysdate, RefAlb));

end;
```

El proceso comienza recuperando las referencias a los productos que se reciben, tras lo cual se crean en los almacenes los lotes en que se ha dividido la mercancía. En este caso del producto 1 se reciben 15 unidades, 5 de las cuales quedan en el almacén principal y los otros 10 se llevan al segundo almacén. Acto seguido se genera el albarán, introduciendo las líneas y recuperando con `RETURNING REF` la referencia a ese nuevo albarán. Esta referencia servirá, finalmente, para registrar la transacción asociándola al proveedor que nos ha enviado la mercancía.

Para registrar una venta se actuaría de manera parecida, sustituyendo las inserciones en los almacenes por actualizaciones en los que se reduciría la cantidad disponible, creando el albarán de igual forma y registran la transacción en la tabla anidada `Ventas` de la tabla `Cliente` en lugar de hacerlo en la tabla `Compras` de `Proveedor`.

Generación de facturas

Las facturas se crean a partir de los albaranes, pudiendo seleccionar un albarán por su número, para emitir una factura puntual, o bien mediante una consulta de las compras o ventas acotada por fechas, para generar una factura de varios albaranes. Como en el caso del registro de una compra o una venta, el código del ejemplo siguiente podría generalizarse y convertirse en un procedimiento almacenado al que se invocase desde la aplicación:

```
declare

 LineasFact TLineasFactura;
 Total real := 0;
 RefFact ref TFactura;

begin

  select TLineaFactura(li.Producto.Descripcion,li.Cantidad,
     li.Precio,li.Precio*li.Cantidad*0.05)
       bulk collect into LineasFact
       from the(select p.Compras from Proveedor p
          where CIF='B-38299173') co,
       table(co.Albaran.Lineas) li
       where round(co.Fecha,'day') = round(sysdate,'day');



  -- Con las filas generadas por la consulta anterior se
  -- calcula el total

  select sum(li.Precio*li.Cantidad-li.Descuento) into Total
  from table(LineasFact) li;



  -- Creación de la factura

  insert into Factura f values(
    '19/07', sysdate, Total, Total*0.16, 0, LineasFact)
  returning ref(f) into RefFact;



  -- Se añade la referencia a la factura a la tabla
  -- Facturas del proveedor

  insert into the(select p.Facturas
      from Proveedor p where CIF='B-38299173')
     values(RefFact);

end;
```

Con la primera consulta se preparan las líneas que compondrán la factura con los datos del albarán y calculando la columna del descuento, que será de un 5%. Los albaranes elegidos

serán los correspondientes a las transacciones del día actual correspondientes a un cierto proveedor, por lo que generará un resultado siempre que se ejecute el código en la misma fecha que el ejemplo previo en el que se preparaba el albarán. En esta consulta se utiliza una combinación de varias características objeto-relacional de Oracle:

- Usando el constructor `TLineaFactura` como argumento del `SELECT` se generan las líneas de factura a partir de datos extraídos de otras tablas y datos calculados, como es el caso del descuento.
- Mediante `BULK COLLECT` se recuperan varias filas en una variable, preparando así la lista de líneas que compondrá la factura.
- En la cláusula `FROM` se utilizan `THE` y `TABLE` para conseguir extraer las líneas de los albaranes que corresponden a las transacciones de un cierto proveedor. `Compras` es una tabla anidada de `Proveedor`, `Albaran` es un atributo de esa tabla anidada que apunta al albarán asociado y `Lineas` es una tabla anidada de `Albaran`.

En la consulta siguiente se obtiene el total de la factura, teniendo en cuenta los descuentos antes calculados. Es una sentencia interesante porque muestra cómo una variable, en este caso `LineasFact`, puede ser utilizada en una sentencia `SELECT` gracias a `TABLE`.

El paso siguiente es crear la factura, estableciendo su número, fecha de emisión, total, IVA e importa pagado, así como la colección de líneas que la forman. Con `RETURNING REF` se recupera la referencia a esa nueva factura para agregarla a la tabla anidada `Facturas` del proveedor correspondiente.

Para generar una factura de un cliente el procedimiento sería análogo, prácticamente la única diferencia sería cambiar `Proveedor` por `Cliente` y `Compras` por `Ventas`.

Registro de pagos

Cada vez que se haga o reciba un pago se registraría en la tabla `Pago`, asociándolos con la factura a que corresponda. Un pago no tiene necesariamente que ser por el total, puede haber pagos parciales en fechas distintas asociados a una misma factura, como ocurre en el siguiente ejemplo:

```
insert into Pago
 select sysdate, ref(f), 45
 from Factura f where NumFactura = '19/07';

 insert into Pago
 select sysdate+1, ref(f), 30
 from Factura f where NumFactura = '19/07';
```

Tomando como base estos ejemplos podría insertarse cualquier otro tipo de información en la base de datos, así como actualizar la cantidad de mercancía disponible o eliminar datos, básicamente no hay más que cambiar los `INSERT` por `UPDATE` o `DELETE` manteniendo el resto de la estructura de las sentencias.

4.2. Consultas

Asumiendo que se han ejecutado todos los procedimientos del punto previo, contenidos en el guion `InsertaDatos.sql`, la base de datos contiene información suficiente para efectuar las consultas que se describen a continuación.

Almacenes y sus direcciones

Es una consulta en la que se observa cómo llamar a un método de una columna que es un objeto:

```
select a.Nombre, a.Direccion.Str() "Dirección"
from Almacen a;
```

Results:

	NOMBRE	Dirección
1	Principal	Cl. Las Cruces, 25
2	Pgno. Esperanza	Avda. Granada, 4

Productos que podemos solicitar a un proveedor

Los proveedores cuentan con una tabla anidada que mantiene referencias a los productos que nos ofrecen, cuya descripción obtendríamos con la siguiente consulta:

```
select value(prod).Descripcion "Producto"
from the (select prv.Productos from Proveedor prv
     where CIF='B-38299173') prod;
```

Results:

	Producto
1	Vestido crep con sobrefalda
2	Pantalón micropana pinzas

La sentencia `SELECT` interna, la que hay entre los paréntesis de la cláusula `THE`, devolvería una fila por producto existente en la tabla anidada de productos de ese proveedor. Cada fila es una referencia `TDescProducto`, por lo que se utiliza `VALUE` para acceder al valor de esa referencia y recuperar la descripción del producto.

Compras hechas a un proveedor

Con esta consulta recuperamos la fecha, líneas de albarán y comentarios de cada transacción de compra hecha a un proveedor:

```
select co.Fecha, co.Albaran.Lineas, co.Albaran.Comentarios
from the(select p.Compras from Proveedor p
     where CIF='B-38299173') co;
```

Results:

	FECHA	ALBARAN.LINEAS	ALBARAN.COMENTARIOS
1	26/12/07	G102.TLINEA.ALBARAN(	Primera compra a este proveedor

Como se puede observar en la imagen, las líneas de albarán aparecen como un objeto y no como valores simples que puedan mostrarse directamente. Para ver los datos de cada línea de albarán se podría rescribir la consulta así:

```
select co.Fecha, deref(li.Producto).Descripcion "Producto",
     li.Cantidad, li.Precio, co.Albaran.Comentarios
from the(select p.Compras from Proveedor p
     where CIF='B-38299173') co,
table(co.Albaran.Lineas) li;
```

Results:

	FECHA	Producto	CANTIDAD	PRECIO	ALBARAN.COMENTARIOS
1	26/12/07	Vestido crep con sobrefalda	15	20,5	Primera compra a este proveedor
2	26/12/07	Pantalón micropana pinzas	7	16,35	Primera compra a este proveedor

En este caso la tabla anidada `Lineas` correspondiente al albarán pasa a ser parte de la cláusula `FROM` gracias a `TABLE`, lo cual permite acceder a sus atributos: `Producto`, `Cantidad` y `Precio`. Dado que `Producto` es, a su vez, una referencia a `TDescProducto`, se utiliza la función `DEREF` para resolverla y recuperar la descripción del producto.

Artículos disponibles en almacenes

Cada almacén cuenta con una tabla anidada en la que conserva los lotes de productos disponibles, con una referencia a la descripción, la cantidad y el precio de adquisición. La siguiente consulta facilitará una lista de esos productos:

```
select al.Nombre, deref(p.DesProducto).Descripcion,
     p.Cantidad, p.PrecioAdq
from Almacen al, table(al.Productos) p;
```

Results:

	NOMBRE	DEREF(P.DESPRODUCTO).DESCRIPCION	CANTIDAD	PRECIOADQ
1	Principal	Vestido crep con sobrefalda	5	20,5
2	Principal	Pantalón micropana pinzas	7	16,35
3	Pgno. Esperanza	Vestido crep con sobrefalda	10	20,5

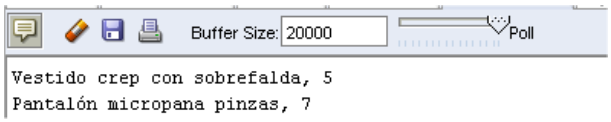
Nuevamente se recurre a `TABLE` para tratar el atributo que representa a la tabla anidada como una tabla más en la cláusula `FROM`, así como la función `DEREF` para resolver la referencia a la descripción del producto.

Recorrer las filas de una tabla anidada

En los ejemplos previos puede verse cómo se seleccionan datos de una tabla anidada para obtenerlos como resultado de una consulta. También es posible que se necesite recorrer las filas de una tabla de este tipo en un bloque PL/SQL, a fin de realizar algún tratamiento individual. El siguiente ejemplo muestra cómo se haría, recorriendo las líneas de los albaranes correspondientes a un proveedor para mostrar por la consola la descripción de cada producto y la cantidad. El mismo procedimiento sería aplicable a otros casos similares:

```
declare
 RefLi TLineasAlbaran;
 Prod TDescProducto;
begin
```

```
  -- Se recuperan en memoria las líneas del albarán

  select value(li) bulk collect into RefLi
  from the(select p.Compras from Proveedor p
     where CIF='B-38299173') co,
  table(co.Albaran.Lineas) li;

  -- Se utiliza un bucle para recorrer las líneas

  for i in RefLi.first .. RefLi.last loop

    -- En PL/SQL no puede usarse directamente una
    -- referencia para acceder a sus miembros (en SQL sí),
    -- por lo que es necesario usar esta sentencia

    select deref(RefLi(i).Producto) into Prod from dual;

    -- Ahora ya se tiene la descripción del producto y el
    -- resto de datos asociados

    dbms_output.put_line(Prod.Descripcion || ', '
      || RefLi(i).Cantidad);

  end loop;

end;
```



Datos de una factura

Al ejecutar una consulta sobre la tabla de facturas, como la siguiente, se obtendrán los datos generales y una columna que contiene las líneas:

```
select * from Factura where NumFactura='19/07';
```

Results:

	NUMFACTURA	FECHA	IMPORTE	IVA	PAGADO	LINEAS
1	19/07	26/12/07	400,8525	64,1364	0	G102.TLINEAFACTURA(G102.TLINEAFACTURA,G102.TLINEAFACTURA)

Para ver las líneas de la misma factura se utilizará la siguiente sentencia:

```
select li.* from the(select f.Lineas from Factura f
     where f.NumFactura='19/07') li;
```

Results:

	DESCRIPCION	CANTIDAD	PRECIO	DESCUENTO
1	Vestido crep con sobrefalda	15	20,5	15,375
2	Pantalón micropana pinzas	7	16,35	5,7225

Si no tenemos el número de una factura concreta, podemos recuperar el número de factura, importe y líneas de las asociadas a un proveedor con una consulta como ésta:

```
select value(f).NumFactura,value(f).Importe,l.*
from Proveedor p, table(p.Facturas) f,
     table(value(f).Lineas) l
where CIF='B-38299173';
```

Results:

	VALUE(F).NUMFACTURA	VALUE(F).IMPORTE	DESCRIPCION	CANTIDAD	PRECIO	DESCUEN...
1	19/07	400,8525	Vestido crep con sobrefalda	15	20,5	15,375
2	19/07	400,8525	Pantalón micropana pinzas	7	16,35	5,7225

Uso de los métodos asociados a los objetos

Varios de los tipos de objeto definidos en este proyecto cuentan con métodos, a los que puede invocarse como parte de una consulta de manera similar a como se accede a los atributos. Los siguiente son algunos ejemplos que demuestran este uso:

```
-- Obtener lo que queda pendiente de pago a un proveedor

select p.PagoPendiente()
from Proveedor p
where CIF='B-38299173';
```

Results:

	P.PAGOPENDIENTE()
1	389,9889

```
-- Obtener las existencias de un cierto producto

select p.Existencias(ref(p))
from DescProducto p
where p.IdDesProducto = '1';
```

Results:

	P.EXISTENCIAS(REF(P))
1	15

```
-- Obtener la lista de proveedores que nos facilita un
-- cierto producto

select value(t) "Proveedores"
from DescProducto p, table(p.Proveedores(ref(p))) t
where p.IdDesProducto = '2';
```

Results:

	Proveedores
1	Mirto S.L.

```
-- Inventario de los distintos almacenes, obteniendo su
-- nombre, el número de artículos que debe haber y su valor

select al.Nombre, al.Inventaria(), al.Valora()
from Almacen al;
```

Results:

	NOMBRE	AL.INVENTARIA()	AL.VALORA()
1	Principal	12	216,95
2	Pgno. Esperanza	10	205

```
-- Obtener albaranes y sus totales

select al.NumAlbaran, al.TotalAlbaran()
from Albaran al;
```

Results:

	NUMALBARAN	AL.TOTALALBARAN()
1	121/07	421,95

5. Referencias

Administración de bases de datos, diseño y desarrollo de aplicaciones

Michael V. Mannino

McGraw-Hill

Fundamentos de diseño de bases de datos

Silberschatz, Korth y Sudarshan

McGraw-Hill

El capítulo 18 del primero: *Sistemas de administración de bases de datos de objetos* y el capítulo 9 del segundo: *Bases de datos basadas en objetos*, me han servido para tener una visión general sobre sistemas OODBMS y ORDBMS, con una pequeña introducción a las características ORDBMS de Oracle.

Oracle PL/SQL, User's Guide and Reference

El noveno capítulo de este libro: *Object Types* (en su versión 8) y el capítulo 12: *Using PL/SQL Object Types* (en su versión 10, http://download.oracle.com/docs/cd/B12037_01/appdev.101/b10807/10_objs.htm#i7530), han sido la referencia a la que he recurrido durante todo el desarrollo del proyecto para conocer los detalles de cada sentencia, cláusula o función relacionada con el tema ORDBMS en Oracle.

Oracle PL/SQL Programming

El capítulo 18: *Object Types*, me ha servido para ver múltiples ejemplos de uso de las características ORDBMS de Oracle y cómo solucionar algunos problemas, como la imposibilidad de obtener una referencia al objeto `SELF` implícito en los métodos de los tipos de objeto.

El capítulo 19: *Nested tables and VARRAYs*, lo he utilizado para conocer detalles sobre el tratamiento de tablas anidadas y otros tipos de colecciones, como los *arrays* con un número fijo de elementos.

Aggregation and Composition in Object-Relational Database Design

Marcos, Vela, Cavero y Cáceres

http://www.science.mii.lt/ADBIS/local1/marcos.pdf

Este documento lo leí al principio, junto con los capítulos de los dos libros que cité en primer lugar, y me aportó el conocimiento necesario para saber cómo tratar en Oracle las composiciones y agregaciones usuales en el diseño orientado a objetos.

Por último he leído algunos documentos disponibles en la Web sobre las características relacionales-objeto de Oracle, entre ellos:

- *Object-Relational Features of Oracle*: http://infolab.stanford.edu/~ullman/fcdb/oracle/or-objects.html#refs.
- *Bulk Collect usando PL/SQL*: http://www.zonaoracle.com/software/bulk-collect/index.asp.
- *Experiences with object oriented development in PL/SQL*: http://www.cit.dk/cot/reports/reports/Case4/18/cot-4-18.pdf.
- *Parameter Self in member functions/procedures in object oriented PL/SQL*: http://www.adp-gmbh.ch/ora/plsql/oo/member_self.html.
- *Type inheritance - Object Oriented Oracle PL/SQL*: http://www.java2s.com/Tutorial/Oracle/0620__Object-Oriented/0080__Type-Inheritance.htm.