Diseño de datos:
Mapeo Objetos Relacional

por Fernando Dodino
Franco Bulgarelli

Versión 3.3
Marzo 2018

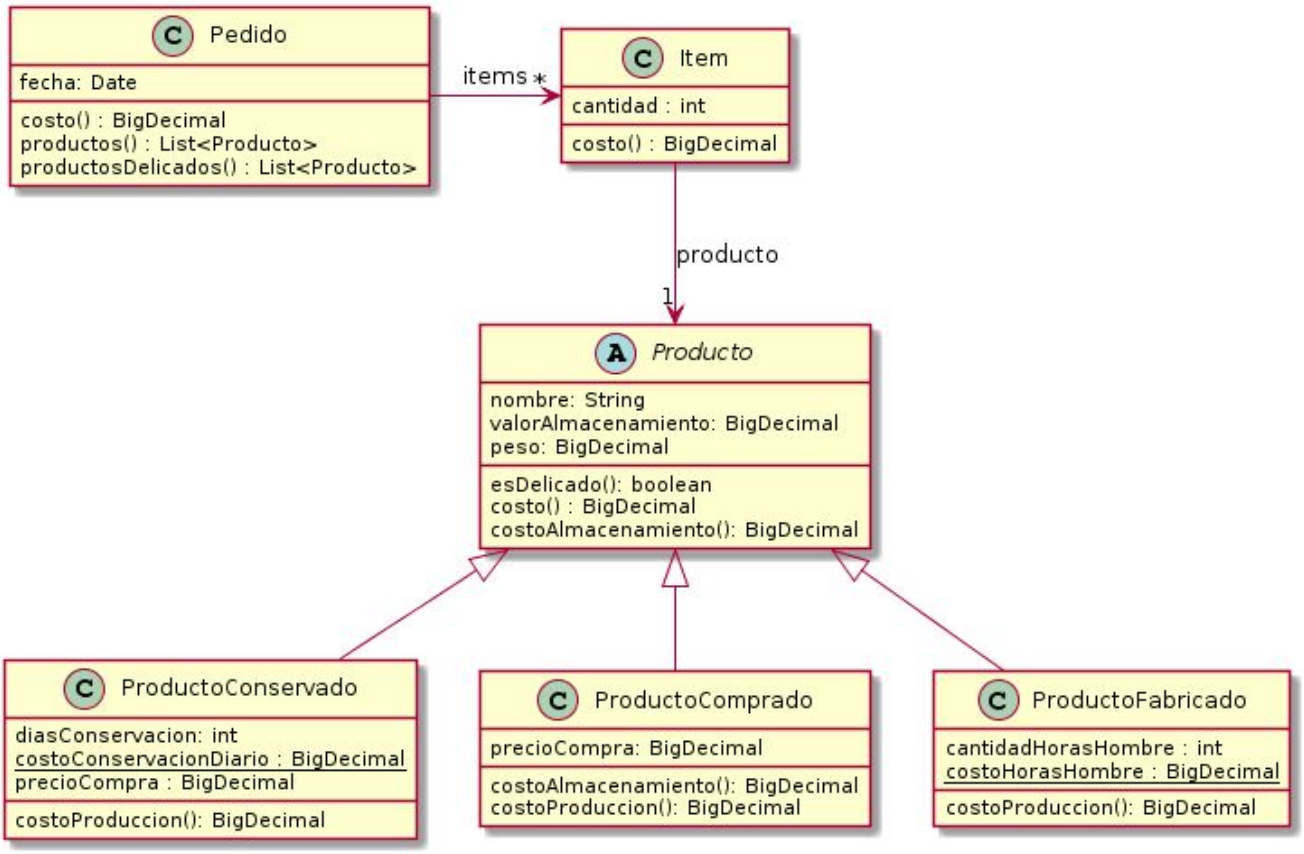
Indice

1. Introducción

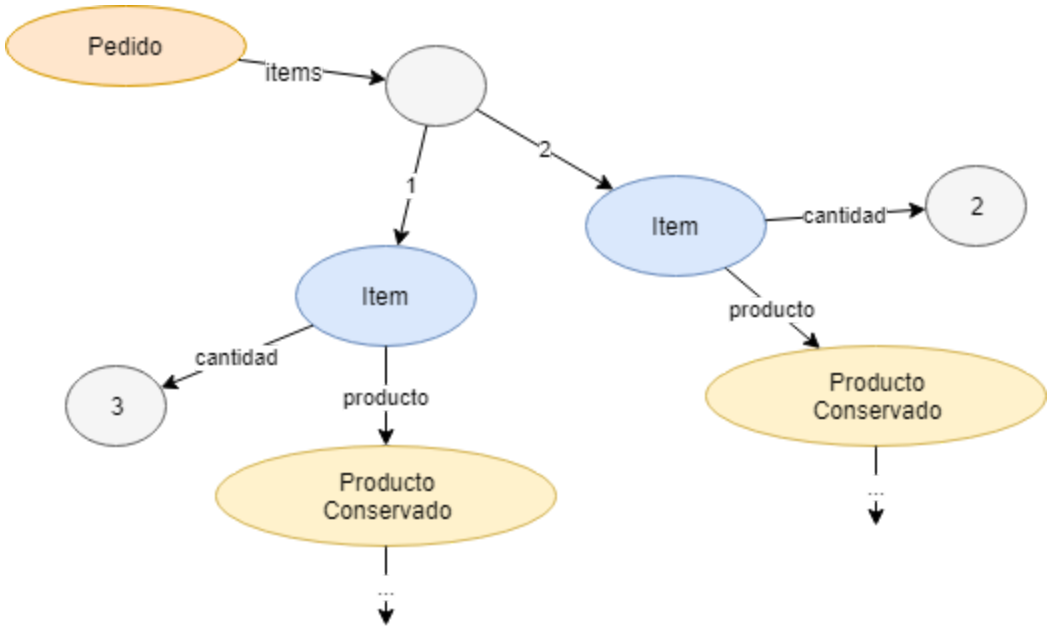
En una aplicación OO “viva” los objetos viven en un ambiente en memoria. A la hora de persistir nuestros objetos, una de las opciones es hacerlo en un modelo relacional. Lo que estudiaremos en este apunte son las diferencias entre ambos mundos (*impedance mismatch*), y cómo podemos conciliarlas, haciendo el mapeo a través de un *framework* ORM, como por ejemplo JPA / Hibernate.

1.1. El problema

Si tenemos un modelo de objetos como el siguiente



¿Qué tipo de estructura siguen los objetos en memoria? Un grafo dirigido



Para persistir este grafo utilizaremos una **base de datos relacional** como repositorio de información.

1.2. Impedance mismatch

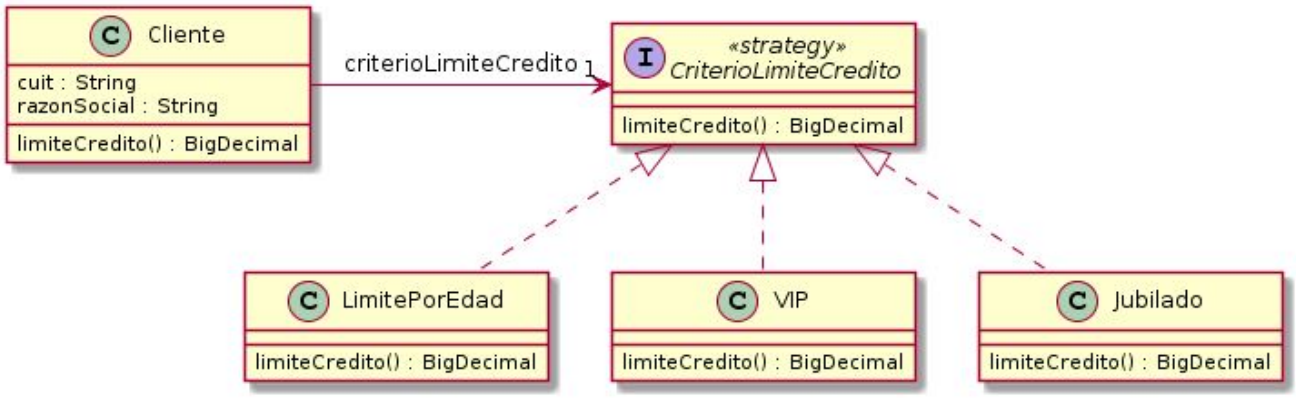
La pregunta es: este grafo que acabamos de dibujar ¿encaja justo en una estructura basada en el álgebra relacional? Mmm... claramente son distintas formas de relacionar entidades.

Describamos brevemente similitudes y diferencias entre ambos modelos

- Originalmente podemos establecer una equivalencia entre el concepto de relación o tabla y el de clase, y una tupla o registro y la instancia de una clase. Más adelante empezaremos a tener ciertas dificultades para que este mapeo sea tan lineal.
- Los objetos tienen comportamiento, mientras que las tablas sólo permiten habilitar ciertos controles de integridad (constraints) o validaciones antes o después de una actualización (triggers). Los stored procedures no están asociados a una tabla, entonces cualquier modificación en la estructura de una tabla impacta en los datos almacenados por un lado y en los programas que actualizan los datos por el otro.
- Los objetos encapsulan información, no para protegerse (siempre puedo acceder a sus atributos), sino para favorecer la abstracción del observador. Una tabla no tiene esa habilidad: si tengo una entidad con un atributo ACTIVO que es VARCHAR2(1), tengo que entrar al constraint check o bien tirar un SELECT para

saber si el ACTIVO es “1” ó “0”, “Y” o “N”, “S” o “N”, e inferir en el mejor de los casos qué significan los valores de ese campo. Puedo documentarlo en un diccionario de datos, algo ajeno a la base que estoy tocando, y que necesita una sincronización propia de gente pulcra y obsesiva.

- Consideremos el siguiente ejemplo



Para resolver el monto en pesos que tienen como límite de crédito un cliente, en la solución con objetos delegamos a un objeto que modela cada criterio. Tanto la interfaz como las estrategias no tienen referencias, no guardan estado, y al ser *stateless* no tiene sentido convertirlas a entidades del modelo relacional.

- La herencia es una relación estática que se da entre clases, que favorece agrupar comportamiento y atributos en común. Cuando instanciamos un objeto recibimos la definición propia de la clase y de todas las superclases de las cuales hereda. En el modelo lógico de un Diagrama Entidad/Relación tenemos supertipos y subtipos, pero en la implementación física del modelo relacional las tablas no soportan el concepto de herencia. Entonces tenemos que hacer las adaptaciones correspondientes (como veremos más adelante).

- Al no existir el comportamiento en las tablas y no estar presente el concepto de interfaz no hay polimorfismo en el álgebra relacional: podemos recibir un objeto sin saber exactamente de qué clase es, y no nos interesa averiguarlo, sólo nos concentramos en lo que le puedo pedir y en su contrato. Al diseñar una consulta, necesitamos saber qué tablas y qué campos están involucrados.

- las variables de clase (*static*) no se pueden asociar a un registro, porque son globales para todos los objetos. Una solución es crear una tabla de Parámetros con un solo registro que especifica valores generales de la aplicación: 20% de descuento para clientes VIP, 30 días de demora para pedidos al interior, etc.

Todas estas cuestiones que hemos nombrado dificultan la traslación de conceptos entre ambos mundos, lo que se conoce como "**Impedance mismatch**".

> Impedance mismatch se traduce al español como "desadaptación de impedancia", término que es muy aclarador... ¡para muy pocos!
>
> En realidad el término proviene del campo de la electrónica, y sirve para explicar el fenómeno por el cual tenemos, por ejemplo, un parlante conectado a un un amplificador, pero entre la entrada y salida de estos hay diferencias en una cierta propiedad (la impedancia), provocando que el audio no se escuche como se debe.
>
> Por extensión, se usa el término en otros campos, siempre hablando de conectar dos elementos que no se adaptan bien.

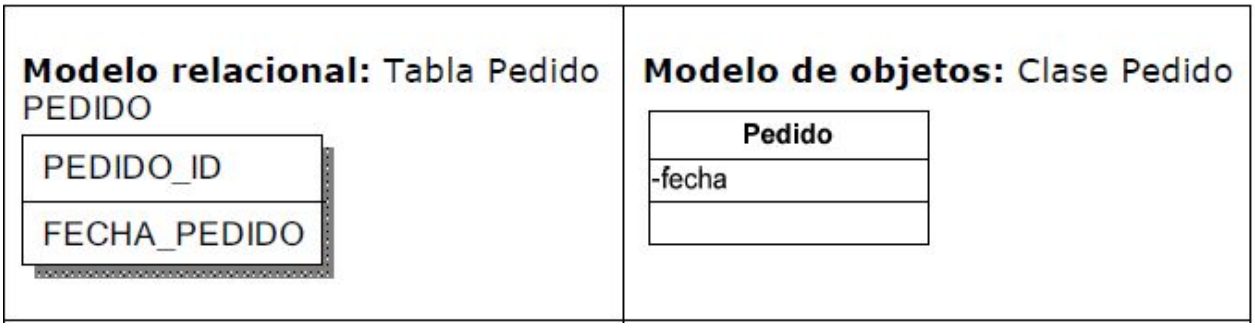
Veamos entonces cómo adaptar el modelo relacional al de objetos, es decir, cómo establecer un mapeo entre nuestro mundo de objetos y nuestro mundo de filas....

Para explicar estas cuestiones generaremos el modelo físico relacional asociado al diagrama de clases de la página 3. Tendremos que encargarnos de algunas cuestiones:

- la identidad
- los tipos de datos
- las relaciones (cardinalidad)
- los tipos de colección (manejo de duplicados y orden)
- la dirección (forma de navegar)
- la herencia

2. Manejo de la identidad

Creamos la entidad Pedidos:



Fíjense qué diferencia tenemos en la tabla Pedido vs. la clase Pedido. Además del comportamiento (el tercer compartimento de la definición de la clase), necesitamos una clave que *identifique unívocamente* a cada registro de la tabla Pedido.

Al diseñar la clave para una entidad, siempre, las opciones son:

- **clave natural:** elijo un subconjunto de los campos de la tabla en base al negocio: CUIT del cliente, número de teléfono del abonado, etc.
- **clave subrogada:** ninguna de las claves naturales me parece suficiente, entonces creo una clave ficticia (al azar, generada automáticamente[1], etc.): en nuestro caso, el que se asigna en base a una secuencia.

Cada una tiene sus ventajas y desventajas. Por ejemplo, las claves naturales muchas veces simplifican algunas consultas y nos llevan a tablas que son más fáciles de entender[2]. Por otro lado, hay que tener más cuidado con claves naturales, tanto si la ingresa el usuario como si es un atributo susceptible de cambiar porque la misma seguramente participa de una o más relaciones[3].

Finalmente, las claves naturales a veces nos llevan a tener claves compuestas (es decir, el subconjunto que identifica a la fila no es unitario). Esto no parece ser algo inherentemente malo, pero los frameworks ORM que veremos más adelante **no se llevan bien con esta idea**.

El PEDIDO_ID permite identificar unívocamente una fila de otra. ¿Qué pasa en objetos? Manejamos el concepto de **identidad**: cada objeto sabe que es él y ningún otro objeto, entonces no es necesario trabajar con un identificador. Ahora bien, cuando persistimos

[1] Típicamente autoincremental

[2] O al menos, más fáciles de entender para quien provenga del mundo de bases de datos relacionales

[3] Por ejemplo, en 1998 se implementó un cambio de número para los abonados telefónicos incorporando el prefijo “4”. También el código postal ahora tiene tres caracteres alfanuméricos extra que identifican la manzana.

un objeto a la base, y lo recuperamos en otro momento, puede pasar que en diferentes sesiones tengamos dos referencias que en realidad estén apuntando al mismo objeto. Si el ambiente vive creando y recreando permanentemente los objetos persistidos en la base, entonces ya no puedo confiar en la identidad.

2.1. Consecuencias para el modelo de objetos

Como queremos integrar ambos modelos (el relacional y el de objetos) debemos generar en el modelo de objetos un atributo id en la clase Pedido. Este nuevo atributo *id* cumple varias funciones:

1. identificar la fila de la tabla donde se guarde
2. determinar si es un objeto nuevo o un objeto que alguna vez me traje de la base (insert cuando id es nulo o inválido / update en caso contrario)

3. Conversiones de tipo

Un dato no menor son los problemas de conversión de tipos que se originan entre el dominio de objetos y el de la base de datos, donde podemos tener:

- un String sin limitaciones de tamaño vs. el VARCHAR/CHAR que requiere definirle longitud
- ídem para campos numéricos respecto a la precisión de decimales, o el rango de valores máximos y mínimos; incluso cuando el dominio es “un número de 0 a 3000”
- las fechas tienen tipos no siempre compatibles entre sí: LocalDate de Java vs. Date-Datetime-Time-tinyblob del motor
- tipos booleanos
- tipos enumerados (como los *enums* de Java)

4. Relación de cardinalidad

4.1. Uno a muchos: Pedido a ítems

Un pedido tiene muchos ítems, pero cada ítem pertenece a un solo pedido. No quiero reutilizar los ítems en distintos pedidos, porque están representando instancias diferentes: si pedí 2 hormas de queso camembert un día, y 2 hormas del mismo queso otro día estamos hablando de diferentes instancias, por lo tanto de diferentes filas en la tabla de pedidos.

¿Cómo modelamos la relación entre pedido e ítems en el modelo relacional? Por la definición del modelo relacional, sabemos que no existen atributos multivaluados, no

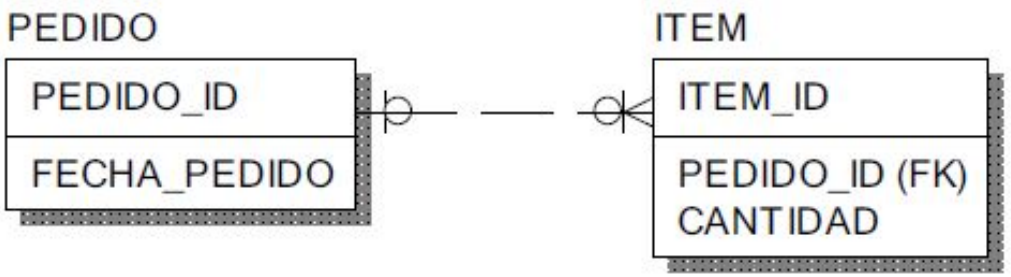
podemos ubicar en pedido una colección de ítems. Entonces, cuando tenemos una relación uno-a-muchos (one-to-many), se ubica el identificador del padre (uno) en la tabla hija (many). En este caso, el identificador del Pedido se incorpora a la tabla de Items. Esto se puede hacer de dos maneras:

- **Opción a:** el identificador del pedido es un atributo que no forma parte de la clave de la tabla hija (non-identifying). El identificador del ítem es una clave subrogada autoincremental.
- **Opción b:** el identificador del pedido es un atributo que forma parte de la clave de la tabla hija (identifying).

Graficamos en un DER ambas soluciones:

Opción a) PEDIDO_ID no forma parte de la clave

Cada fila de la tabla ITEMS se identifica con una clave subrogada autoincremental.



ITEM_ID (PK)	**PEDIDO_ID (FK, pero no PK)**	**CANTIDAD**
1	1	...
2	1	...
3	2	...
4	2	...

Opción b) PEDIDO_ID forma parte de la clave

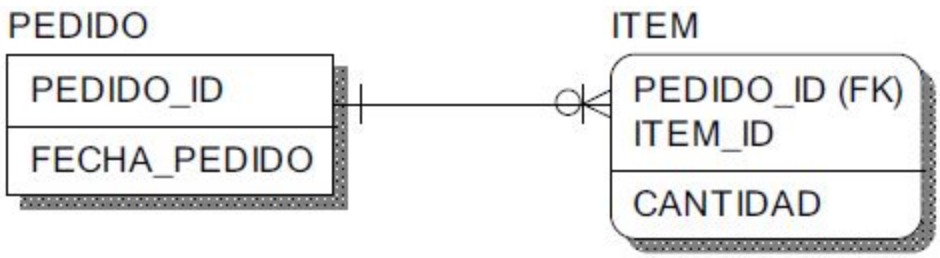
- Entonces las opciones son:
  - que la clave del ítem se componga de PEDIDO_ID + PRODUCTO_ID. Esta opción es válida si el pedido tiene un conjunto de productos, sin repetidos.

PEDIDO_ID (PK/FK)	**PRODUCTO_ID (PK/FK)**	**CANTIDAD**
1	**5**	...
1	**7**	...

2	**6**	...
2	**5**	...

- o bien tener un identificador del ítem que sea autoincremental teniendo en cuenta el identificador del pedido. ¿Qué pedido soy? El 2, ah, entonces tengo que averiguar mi último Item_Id + 1 para el Pedido 2. Esta operación debe ser atómica y no debe permitir accesos concurrentes. Esta estrategia es más compleja, pero es una alternativa cuando un pedido puede tener varios ítems con los mismos productos.

PEDIDO_ID (PK/FK)	**ITEM_ID (PK)**	**CANTIDAD**
1	**1**	...
1	**2**	...
2	**1**	...
2	**2**	...



- Quienes modelan con objetos prefieren trabajar con claves autoincrementales, delegando la responsabilidad de manejar los identificadores a la base de datos. Esto es cómodo porque como hemos visto en objetos
  - no necesitamos resolver esa responsabilidad (los objetos definen su propia estrategia para manejar la identidad)
  - si tenemos varias VMs sincronizadas contra la base necesitamos asegurarnos que dos objetos de distintos ambientes no tengan el mismo ID (también habría que pensar si la identidad no conviene redefinirla basándola en el id).
- Quienes piensen el diseño a partir del modelo de datos probablemente prefieran pensar en que el ítem se identifica mediante una clave compuesta, ya que no tiene sentido un ítem sin su pedido.

4.1.1. Tipos de colecciones

En objetos, para definir una colección necesitamos preguntarnos

- si nos importa respetar el orden de los elementos

- si la colección admite duplicados
- cómo es la estrategia para acceder a los elementos

y en base a estas preguntas, sabemos qué tipo de colección utilizaremos.

> Repasemos rápidamente los tipos de colecciones más importantes:
>
> *Set*: es un **conjunto** de elementos, que no preserva orden ni soporta duplicados.
> *Bag*: es una **bolsa** de elementos, que no preserva orden pero soporta duplicados
> *List/OrderedCollection*: es una **lista** de elementos, que preserva el orden y soporta duplicados
> Y por último, **conjuntos ordenados**, que preservan orden y no admiten repetidos.

En el caso del pedido con sus ítems, ¿qué tipo de colección estuvimos modelando? Un set, que no maneja orden, por el concepto mismo de relación.

¿Cómo diseñar una fila de supermercado, donde el orden es importante, y en la que no queremos tener repetidos (conjunto ordenado)? Necesitamos incorporar un campo adicional:

Ej: si tenemos

- **fila 1:** [cliente 155]
- **fila 2:** [cliente 201, cliente 172, cliente 127]
- **fila 3:** [cliente 144]

Filas

FILA_ID (PK)	**...**	**...**
1		...
2		...
3		...

Filas_Clientes

FILA_ID (PK, FK)	**CLIENTE_ID (PK, FK)**	**ORDEN**
1	**155**	0
2	**172**	1
2	**201**	0
2	**127**	2

3	**144**	0

Como vemos, el orden no participa de la clave, porque no se pueden repetir los mismos clientes en la fila.

Si por el contrario, debemos admitir repetidos además de preservar el orden (una lista), sí tendremos que incorporar la columna orden como parte de la PK.

Y finalmente, si lo que queremos persistir es un bag, una colección que admite duplicados pero no tiene orden, aunque la posición relativa de cada elemento no nos importe, aún así necesitaremos de una columna extra (orden o similar) que evite la producción de claves duplicadas.

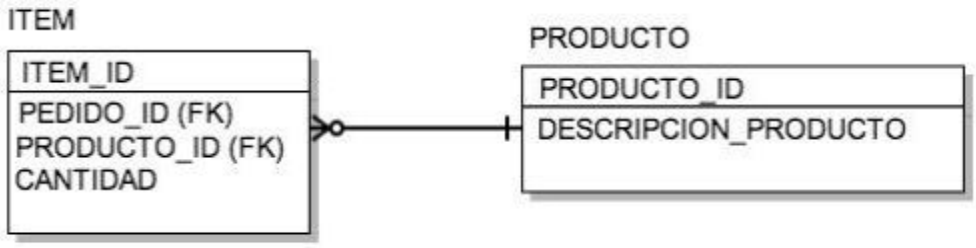
En resumen:

Tipo de colección	¿Requiere columna de orden?	¿Forma parte de la PK?
Set	No	No
Conjunto ordenado	Sí	No
List	Sí	Sí
Bag	Sí, o cualquier columna que evite repeticiones	Sí

4.2. Muchos a uno: Ítem a Producto

Analizamos la relación:

- cada ítem tiene un producto
- pero el producto puede estar en varios ítems. Como señalamos en el ejemplo de las hormas de queso por cada venta de un producto vamos a tener la referencia al mismo producto múltiples veces en la tabla Items.



Entonces aquí la relación es muchos-a-uno (many-to-one) y se resuelve de la misma manera que en la relación one-to-many: en la tabla hija se agrega el identificador del padre. Es decir, en la tabla Items (muchos) se incorpora el PRODUCTO_ID que referencia a una fila en la tabla Productos (uno)

Items

ITEM_ID (PK)	*PEDIDO_ID (FK)*	**PRODUCTO_ID (FK)**	*CANTIDAD*
1	*1*	***5***	...
2	*1*	***7***	...
3	*2*	***6***	...
4	*2*	***5***	...

Productos

PRODUCTO_ID (PK)	**DESCRIPCION**	**...**
5	Queso camembert	...
6	Queso cuartirolo	...
7	Panceta ahumada	...
8	Longaniza nepalesa	...

one-to-many vs. many-to-one es importante distinguirlo en objetos porque son dos relaciones diferentes:

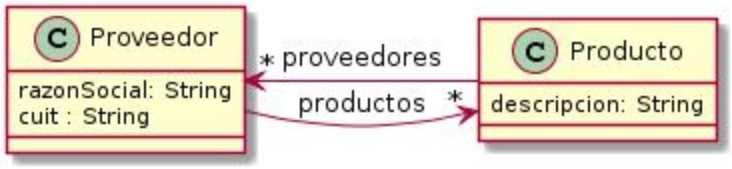
- many-to-one: un objeto A tiene un objeto B del otro lado (el many se "pierde" si no agrego la relación en el objeto B)
- one-to-many: un objeto A tiene una colección de objetos B

4.3. Muchos a muchos: proveedores y productos

Ejemplo 1: “Un proveedor vende muchos productos y un producto es vendido por muchos proveedores”. ¿Cómo lo modelo en objetos? Son colecciones desde los objetos

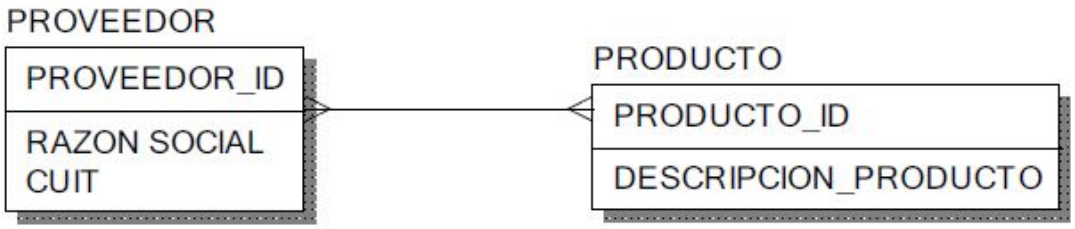
raíces (proveedor y producto respectivamente).

Diagrama de clases

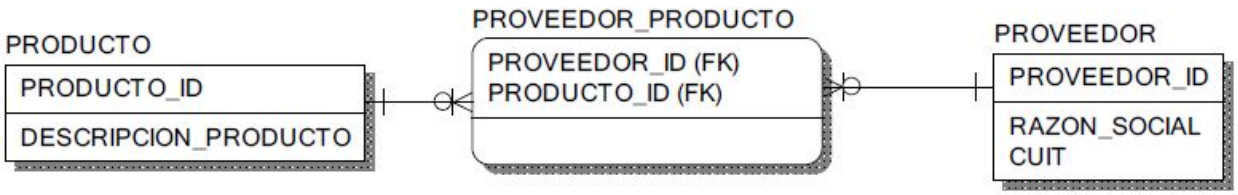


El modelo lógico del Diagrama Entidad-Relación se parece bastante al modelo de objetos:

Modelo Lógico



Pero al implementar el modelo físico sí o sí necesito manejarlo con una Tabla de Relación:



Ejemplo 2: Modelar el stock de una empresa que tiene varios depósitos. Un producto tiene distintas cantidades en stock en cada depósito. O sea: un producto está en varios depósitos y cada depósito tiene muchos productos. La diferencia es que aquí además de tener una relación muchos a muchos, hay atributos que salen de esa relación: la cantidad en stock, por ejemplo. Entonces, en objetos ya no me alcanza con tener dos colecciones desde cada objeto original, necesito sí o sí un objeto que represente la relación entre ambas entidades:

Esta solución permite conocer el stock de un producto a partir de un depósito. Pero también podría definir la navegación de manera que un producto pueda conocer su stock:

Ahora el Stock tiene una referencia al depósito: en Objetos puedo aumentar la navegabilidad (lo que conoce un objeto), pero tengo que

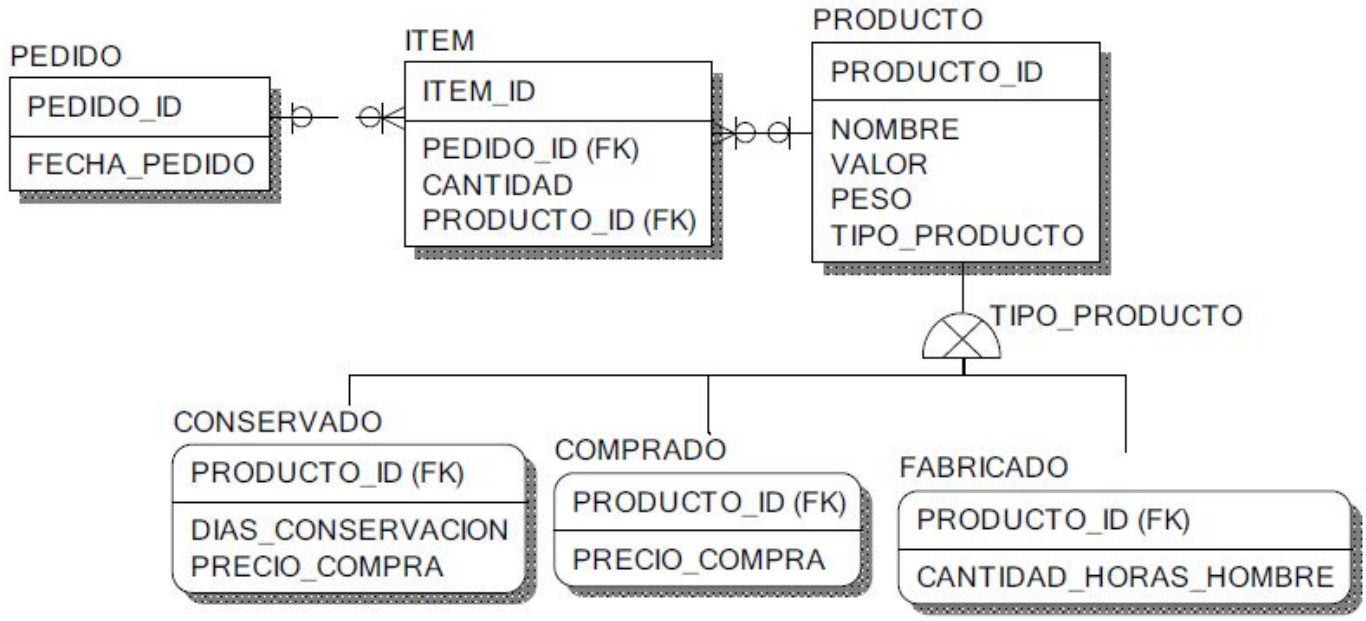
1. agregar más referencias y
2. asegurar de dejar consistente mi modelo (cuando genero un Stock tengo que avisarle al Depósito y al Producto para que tengan actualizada la colección de stocks de cada uno).

En relacional, tengo una vez más una tabla de Relación:

En el modelo relacional puedo determinar el stock de los productos para un depósito o bien puedo saber cuánta cantidad hay de un producto en cada uno de los depósitos sin necesidad de modificar ninguna relación. Este modelo es muy flexible, permite cualquier tipo de navegación, mientras que en el modelo de objetos tengo que entender los casos de uso para definir si me conviene que una asociación sea unidireccional o bidireccional.

5. Subtipos en el modelo relacional

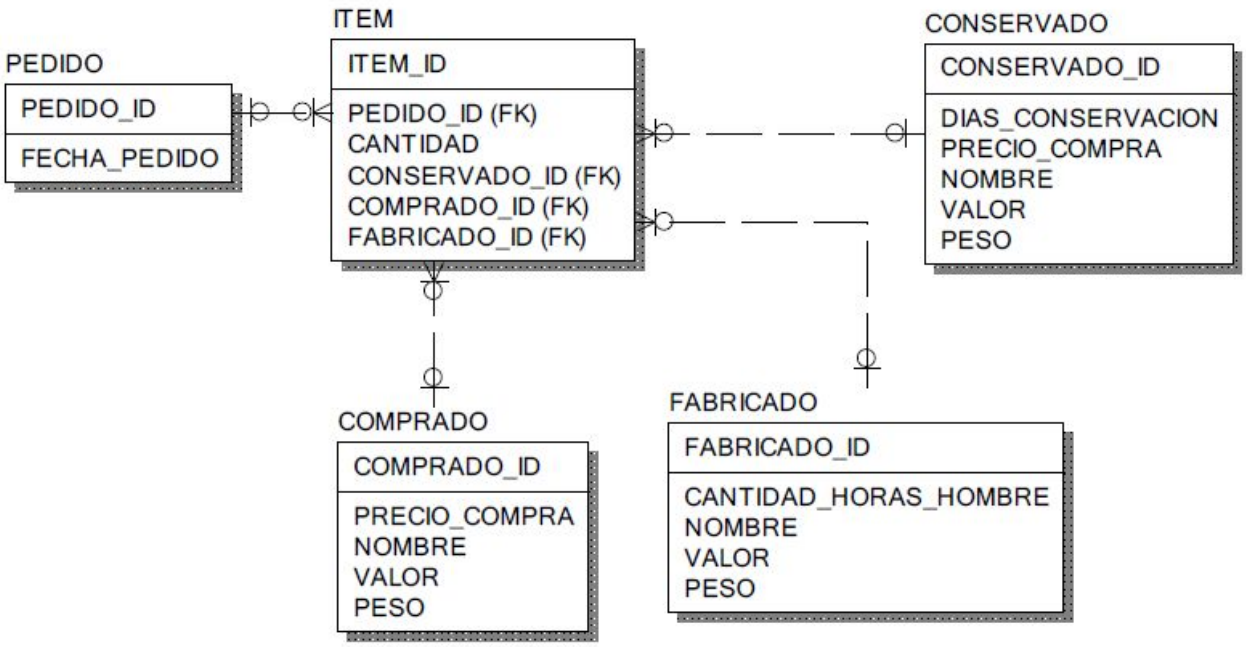
Ahora definiremos en detalle la entidad Producto. Ar memos primero el modelo lógico equivalente a la clase Producto con sus subclases:



Entre el modelo lógico relacional y el modelo de objetos parece no haber mucha diferencia. Ahora bien, al implementar físicamente este modelo tenemos 3 opciones:

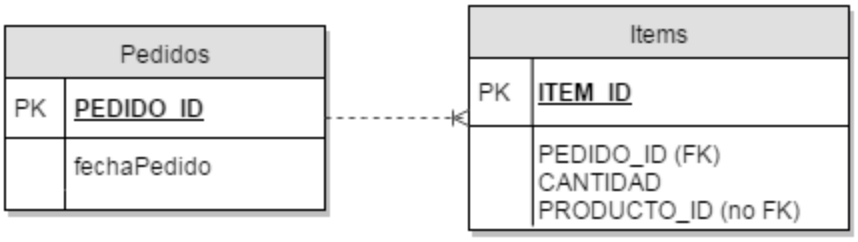
5.1. Una tabla por cada clase concreta: TABLE_PER_CLASS

El primer modelo que veremos consiste en tener una tabla por cada clase concreta, modelo que se conoce como Table Per Class (JPA), Table Per Concrete Class (Hibernate) o Polymorphic Association (ActiveRecord).

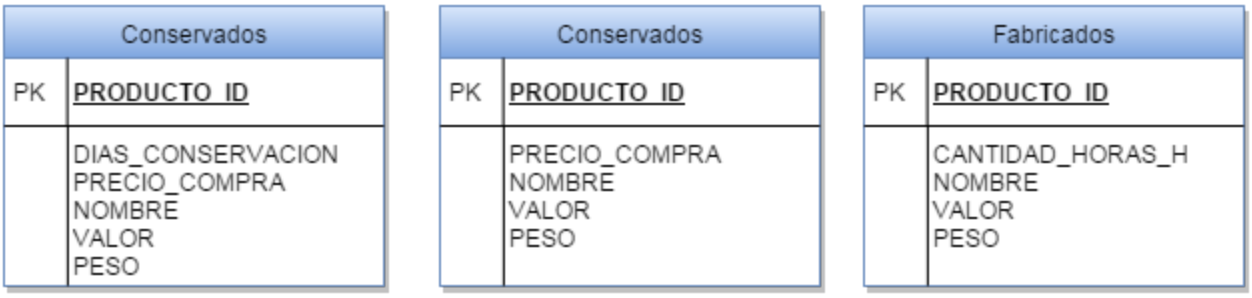


- No existe una tabla Producto, sino que
  - cada subtipo de producto se implementa como una tabla separada
- El ítem tiene 3 claves foráneas que admiten null (porque si comprado_id tiene un valor esto significa que fabricado_id y conservado_id quedan con valores nulos)
- En Comprado, Fabricado y Conservado se repiten todos los atributos definidos para producto: nombre, valor y peso

Cabe destacar que si bien conceptualmente se tienen 3 FKs que apuntan a cada una de las tablas, a nivel implementación (al menos en JPA y ActiveRecord) se suele utilizar una sola columna (que podría llamarse PRODUCTO_ID) que apunta alternativamente a cualquiera de las tres tablas, con lo cual el espacio de las FKs se reduce a una. Sin embargo, el costo a pagar por eso es que para permitir apuntar a cualquier tabla, se elimina el chequeo de la integridad referencial.



No hay Foreign Keys entre Items y los productos



Imaginemos que tenemos estos pedidos:

- Pedido 1, con fecha 12/07/2013 y dos ítems
  - 2 unidades de Queso Camembert (producto fabricado)
  - 1 unidad de Queso Gruyère (producto conservado)
- Pedido 2, con fecha 18/08/2013 y dos ítems
  - 7 unidades de Jamón Cocido (producto comprado)
  - 3 unidades de Queso Camembert (producto fabricado)

Veamos cómo se implementa el modelo en base a la definición de las tablas:

Pedidos

PEDIDO_ID (PK)	**FECHA_PEDIDO**	**CLIENTE_ID (FK)**
1	12/07/2013	...
2	18/08/2013	...

Item

ITEM_ID (PK)	**PEDIDO_ID (FK)**	**CANTIDAD**	**FABRIC_ID (FK)**	**CONSERV_ID (FK)**	**COMPRADO_ID (FK)**
1	1	2	52	null	null

2	1	1	null	22	null
3	2	7	null	null	31
4	2	3	52	null	null

o bien

Item (segunda opción, la que implementan los frameworks basados en JPA)

ITEM_ID (PK)	PEDIDO_ID (FK)	CANTIDAD	PRODUCTO_ID (no es FK, no puede tener constraint contra 3 tablas distintas)
1	1	2	52 (fabricado)
2	1	1	22 (conservado)
3	2	7	31 (comprado)
4	2	3	52 (fabricado)

Conservados

ID (PK)	DIAS	PRECIO	NOMBRE	VALOR	PESO
22	21	57,00,	Queso Gruyère	16,25	1,87

Fabricados

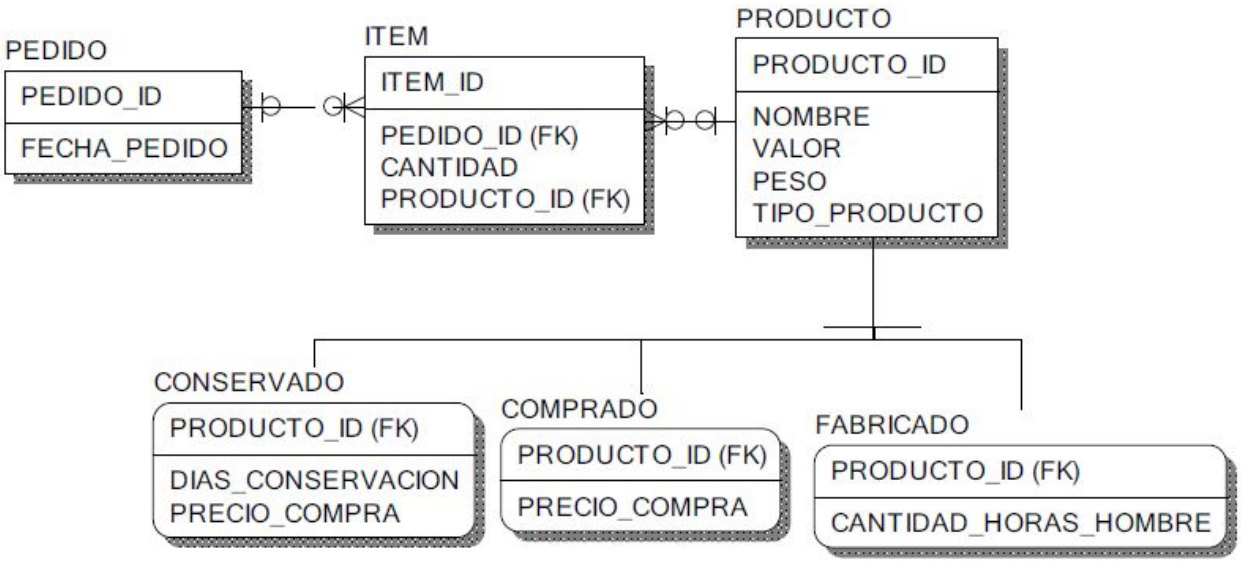
ID (PK)	CANT_HORAS_HOMBRE	NOMBRE	VALOR	PESO
52	15	Queso Camembert	25,67	0,45

Comprados

ID (PK)	PRECIO	NOMBRE	VALOR	PESO
31	75,48	Jamón Cocido	27, 98	0,912

5.2. Una tabla por cada clase + joins: JOINED

Este modelo consiste en tener una tabla por cada clase, ya sea concreta o abstracta, modelo que se conoce como Joined (JPA) o Table Per Subclass (Hibernate)



- además de la tabla Producto, creamos una tabla por cada subclase
    - la tabla madre tiene una relación one-to-one con cada subclase
- pareciera ser la mejor adaptación del modelo de objetos al relacional.
- no se duplican atributos en las tablas.
- se unifica la foreign key de ítem hacia la tabla producto.
- el ID_PRODUCTO en las subclases es al mismo tiempo PK y FK.
- como desventaja, para conocer la información de un producto comprado, fabricado o conservado tengo un grado más de indirección (necesito un JOIN adicional contra esa tabla). La excepción a esto es que se consulte información solamente en la tabla hija o en la tabla padre sin que necesite obtener el objeto Comprado, Conservado o Fabricado, pero para construir una instancia concreta necesitamos acceder a la tabla donde están los datos propios de la subclase.
- insertar un producto requiere dos inserts.
- en este caso aparece el tipo de producto como discriminante, aunque su implementación es opcional, ya que para determinar si un producto es comprado, conservado o fabricado eso lo determina la entidad subclase contra la cual se hace el JOIN exitoso

Pedidos

PEDIDO_ID (PK)	FECHA_PEDIDO	CLIENTE_ID (FK)
1	12/07/2013	...
2	18/08/2013	...

Item

ITEM_ID (PK)	PEDIDO_ID (FK)	CANTIDAD	PRODUCTO_ID (FK)
1	1	2	*52*
2	1	1	*22*
3	2	7	*31*
4	2	3	*52*
...			

Productos

ID (PK)	TIPO_PROD (discr)	NOMBRE	VALOR	PESO
22	CONS	Queso Gruyère	16,25	1,87
31	COMP	Queso Camembert	25,67	0,45
52	FABR	Jamón Cocido	27, 98	0,912
...				

Conservados

ID (PK)	DIAS	PRECIO
22	21	57,00
...		

Fabricados

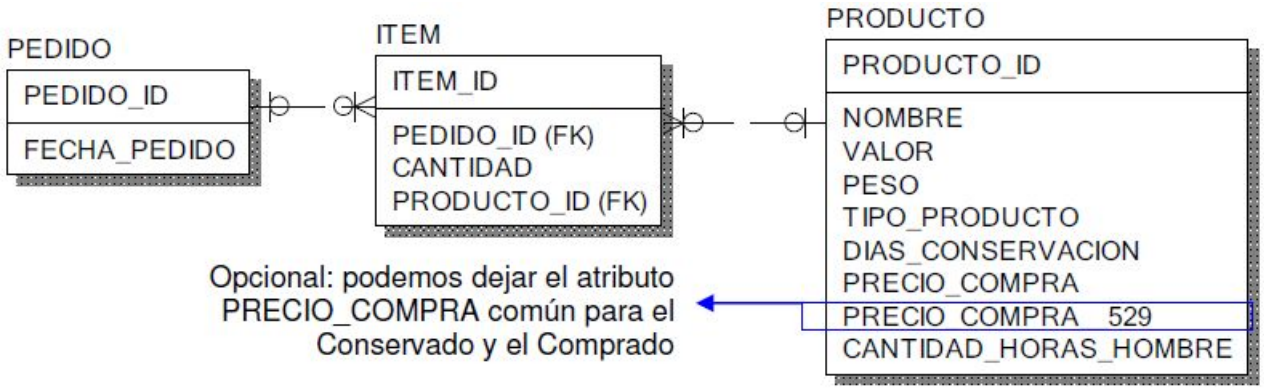
ID (PK)	CANTIDAD_HORAS_HOMBRE
52	15
...	

Comprados

ID (PK)	PRECIO
31	75,48
...	

5.3. Implementar una única tabla (SINGLE_TABLE)

La idea de esta estrategia es persistir la jerarquía completa en una sola tabla. Este esquema se conoce como Single Table (JPA) o Single Table Inheritance (Active Record). En este caso el campo discriminante es obligatorio para determinar la clase concreta a la que pertenece el dato almacenado.



Pedidos

PEDIDO_ID (PK)	FECHA_PEDIDO	CLIENTE_ID (FK)
1	12/07/2013	...
2	18/08/2013	...

Item

ITEM_ID (PK)	PEDIDO_ID (FK)	CANTIDAD	PRODUCTO_ID (FK)
1	1	2	*52*
2	1	1	*22*
3	2	7	*31*
4	2	3	*52*

Productos

ID (PK)	TIPO_ PROD	NOMBRE	VALOR	PESO	DIAS	*PRECIO*	CANT_HORAS _HOMBRE
22	CONS	Q.Gruyère	16,25	1,87	21	57,00	*null*
31	COMP	Q.Camembert	25,67	0,45	*null*	*75,48*	*null*
52	FABR	Jamón C.	27,98	0,912	*null*	*null*	15

5.4. Análisis Comparativo

5.4.1. Relaciones many

Queremos conocer los productos de un proveedor dado que pesen más de 1 kg.[4]

SINGLE_TABLE

Es muy fácil lograr este requerimiento

```
SELECT *
  FROM PRODUCTOS prd
 WHERE prd.PESO > 1
   AND prd.PROVEEDOR_ID = :ID_PROVEEDOR_A_BUSCAR
```

TABLE PER CLASS

Es necesario hacer la consulta de las tres entidades:

```
SELECT con.ID
  FROM CONSERVADOS con
 WHERE con.PESO > 1
   AND con.PROVEEDOR_ID = :ID_PROVEEDOR_A_BUSCAR
(UNION)
SELECT com.ID
  FROM COMPRADOS com
 WHERE com.PESO > 1
   AND com.PROVEEDOR_ID = :ID_PROVEEDOR_A_BUSCAR
(UNION)
SELECT fab.ID
  FROM FABRICADOS fab
 WHERE fab.PESO > 1
   AND fab.PROVEEDOR_ID = :ID_PROVEEDOR_A_BUSCAR
```

Aunque requiere de una consulta específica por cada subclase (lo cual implica un costo adicional de performance que hay que estar dispuesto a pagar), de esta manera

[4] Algunos frameworks como Hibernate llaman **queries polimórficos** a las consultas de objetos que comparten una misma superclase en común, como podría ser el caso de los productos. Para más referencia el lector puede consultar los puntos 3.6 Mapping class inheritance y 6.4 Mapping polymorphic associations del libro *Hibernate In Action*, de Christian Bauer y Gavin King, Manning Publications, 2005 (disponible en pdf)

podemos reconstruir una colección de productos polimórficos en el proveedor.

JOINED

```
SELECT *
  FROM PRODUCTOS prd
       LEFT JOIN CONSERVADOS CON
       ON CON.PRODUCTO_ID = P.ID
       LEFT JOIN COMPRADOS COM
       ON COM.PRODUCTO_ID = P.ID
       LEFT JOIN FABRICADOS FAB
       ON FAB.PRODUCTO_ID = P.ID
 WHERE prd.PESO > 1
   AND prd.PROVEEDOR_ID = :ID_PROVEEDOR_A_BUSCAR
```

Si bien podemos resolverlo mediante un solo query, se requiere trabajar con LEFT JOIN contra todas las tablas que representan las subclases, para poder reconstruir los objetos Producto. Esta indirección supone también un costo adicional de performance.

5.4.2. Relaciones -one

Supongamos ahora que queremos incorporar al modelo cuál es el producto estrella de cada proveedor. Esto implica tener una relación adicional many-to-one entre proveedor y producto. Veamos cómo se resuelve en cada caso:

- **SINGLE_TABLE**: es sencillo, el proveedor agrega una FK a la entidad PRODUCTOS, utilizamos un campo PRODUCTO_ESTRELLA_ID
- **JOINED**: lo mismo, se incorpora un PRODUCTO_ESTRELLA_ID que referencia a la entidad PRODUCTOS (que mapea la clase abstracta)
- **TABLE_PER_CLASS**: aquí tenemos un problema, porque no podemos utilizar una clave foránea, a lo sumo las opciones son:

PROVEEDORES ID (PK) ... PRODUCTO_ESTRELLA_ID : Numeric

hacer la búsqueda en cada una de las tablas que representa una subclase de producto (de esta manera no tenemos FK)

o bien tener 3 claves foráneas que acepten nulos:

PROVEEDORES
ID (PK)
...
CONSERVADO_ESTRELLA_ID (FK nullable) : Numeric
COMPRADO_ESTRELLA_ID (FK nullable): Numeric
FABRICADO_ESTRELLA_ID (FK nullable): Numeric

Resumen

Concepto	A favor	En contra
1 tabla	- Simple - Buena performance general - Soporta todo tipo de relaciones polimórficas - Evita generar muchas tablas	- Campos no utilizados que deben aceptar valores nulos - Tentación de “reutilizar” atributos para cosas distintas - Necesita tener un campo discriminador para reconstruir el objeto (rehidratarlo)
1 tabla por clase concreta (n)	- Permite establecer campos no nulos para cada subclase - No requiere tener un campo discriminador	- Para trabajar con colecciones de objetos polimórficos tengo que hacer consultas *union* y adaptarlas, - no soporta foreign-keys cuando la relación es *-to-one - Cada subclase repite atributos “heredados” de la superclase, tiene mayor costo agregar campos en la superclase
1 tabla por cada clase (n + 1)	- Es el modelo “ideal” según las reglas de normalización (evita redundancia de definiciones en la estructura de las tablas) - Permite establecer campos no nulos para cada subclase - Soporta todo tipo de relaciones polimórficas - No requiere campo discriminador	- Es la opción que más entidades requiere crear - y la que más cantidad de accesos a la base requiere (mayor costo de performance)

Armamos una tabla comparativa:

Concepto	Cuando conviene
1 tabla	Cuando las subclases comparten muchos atributos en común Cuando quiero evitar muchos JOINs al hacer las consultas
tabla por clase concreta (n)	Es la técnica utilizada para las entidades independientes (las clases que heredan de Object en la jerarquía: en el ejemplo de la página 1 serían Pedido, Item y Producto) Cuando las subclases comparten muy pocos atributos entre sí
1 tabla por cada clase (n + 1)	Cuando hay muchos atributos comunes entre las subclases pero también varios atributos propios en cada subclase. Si no hay una cantidad significativa de datos y sabemos que no afectará la performance.

6. Selección del grafo de objetos (hidratación)

En el grafo de objetos navegamos a partir de un objeto raíz considerando que todos los objetos están en memoria. Tengo un cliente, le pido el total y el cliente tiene acceso a sus pedidos. Para conocer el monto total, cada pedido tiene acceso a sus ítems, y cada ítem conoce su cantidad y tiene una referencia al producto.

Si el Pedido tiene ítems y cada Item tiene Producto, por un lado es cómodo traer toda la estructura del objeto Pedido para poder navegarlo libremente. Por otro lado, quizás arme toda una estructura sólo para averiguar la fecha de un pedido.

Gráficamente:

1. Obtengo sólo el pedido.

Tabla: PEDIDOS

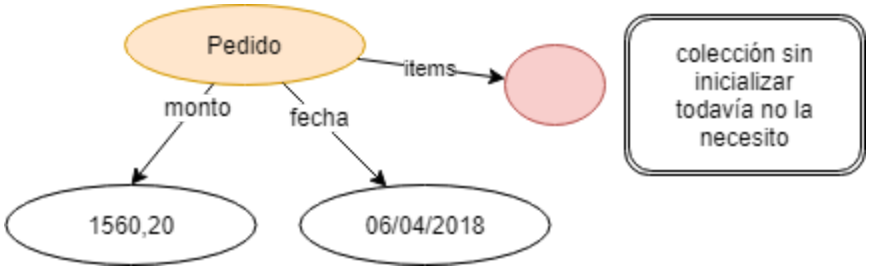
ID_PEDIDO (PK)	**FECHA_PEDIDO**	**CLIENTE_ID (FK)**	**MONTO**
1	10/10/2007	122	1562,23
2	11/10/2007	336	769,83

Queremos conocer todos los pedidos de un cliente:

```
SELECT *
  FROM PEDIDOS
 WHERE CLIENTE_ID = 122
```

Esto se transforma en un cliente con muchos objetos pedido, sin entrar a conocer los

ítems.



2. Obtengo el pedido y todas las relaciones asociadas:

Tabla: PEDIDOS

ID_PEDIDO	FECHA_PEDIDO	CLIENTE_ID	MONTO
1	10/10/2007	122	1562,23
2	11/10/2007	336	769,83

Tabla: ITEMS (que podríamos llamar PEDIDO_PRODUCTO)

ID_PEDIDO_PRODUCTO	ID_PEDIDO	PRODUCTO_ID	CANTIDAD
1	1	24	5
2	1	57	3

Tabla: PRODUCTOS

ID_ PRODUCTO	DESCRIPCION	...	PRECIO_UNITARIO
24	ARANDELA 39 ALUM		20,7433
57	RETEN TENSOR FI		300

Ahora tenemos que recuperar los pedidos del cliente:

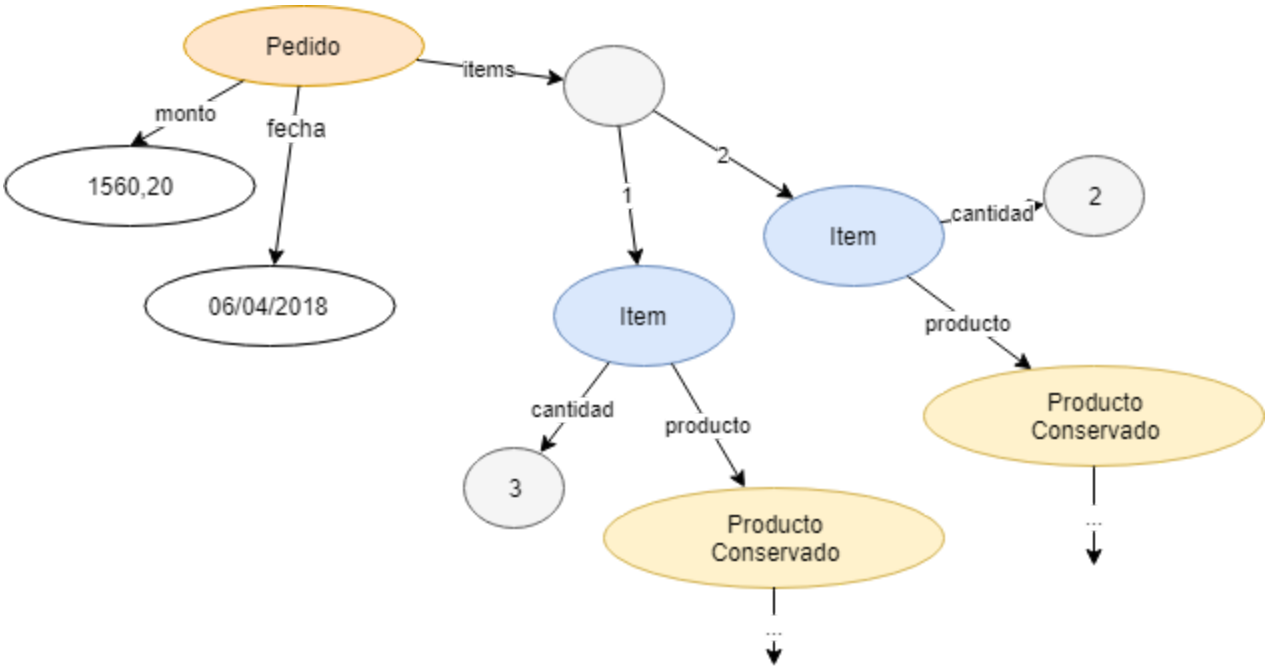
```
SELECT *
  FROM PEDIDOS
 WHERE CLIENTE_ID = 122
```

Pero también tenemos que generar los ítems y los productos. Por cada ID_PEDIDO obtenido en la consulta anterior hacemos:

```
SELECT *
  FROM ITEMS i
```

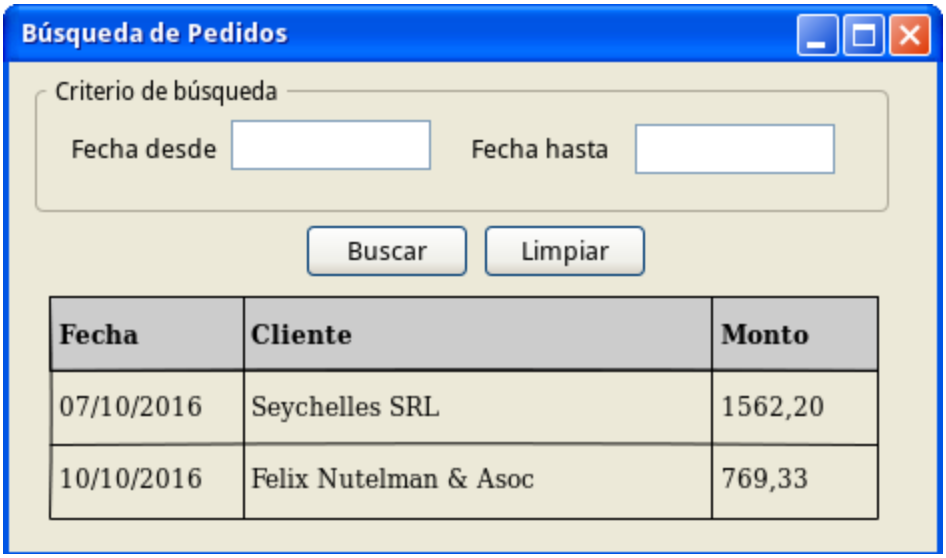
```
        INNER JOIN PRODUCTOS p
        ON p.ID_PRODUCTOS = i.ID_PRODUCTOS
WHERE i.ID_PEDIDO = ?[5]
```

Con esto el ambiente de objetos se recrea de la siguiente manera:



¿Qué me conviene? ¿ (1) ó (2) ?

Si tengo que diseñar una aplicación donde tengo una pantalla de búsqueda de Pedidos:



[5] El query puede variar dependiendo de la estrategia de mapeo de herencia elegida

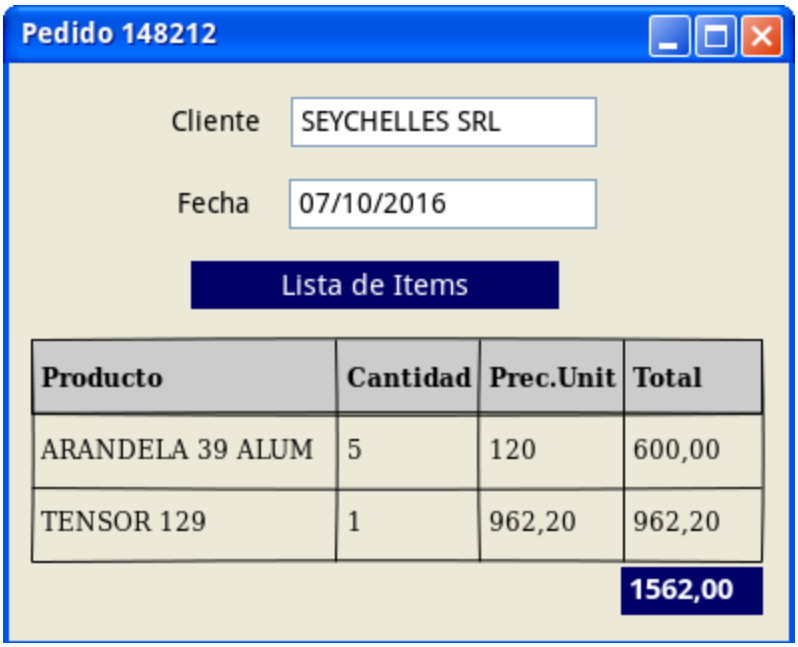
vamos a querer llenar la lista de pedidos accediendo solamente a la tabla PEDIDOS (y al cliente, relación *many-to-one*). Si el usuario seleccionó fecha desde y fecha hasta, el query sería algo similar a:

```
SELECT *
  FROM PEDIDOS p,
       JOIN CLIENTES c
       ON p.CLIENTE_ID = c.ID
 WHERE FECHA_PEDIDO BETWEEN :FechaDesde AND :FechaHasta
```

¿Necesito algo más para mostrar la lista de pedidos? No, en este caso no necesito traer los ítems ni los productos. En general en frameworks que hacen mapeo O/R yo tengo la posibilidad de definir hasta dónde voy al repositorio para generar el grafo de objetos: ese hasta dónde vale tanto en relaciones muchos a uno (del ítem al producto), como en relaciones uno a muchos (del pedido a los ítems).

En cambio cuando el usuario haga doble click en la lista de pedidos y quiera visualizar la información de ese pedido:

Fecha	Cliente	Monto
07/10/2016	Seychelles SRL	1562,20
10/10/2016	Felix Nutelman & Asoc	769,33



sí queremos traer la información del pedido y todas las entidades relacionadas.

6.1. Lazy association

Definimos la colección de ítems del pedido como una asociación *lazy*, donde voy a traer la información "sólo cuando lo necesite". Entonces:

- si el atributo monto no estuviera en pedido, para calcular el total de un pedido y mostrarlo en la grilla original necesitaríamos acceder la información de los ítems y los productos
- si por el contrario cada línea de la grilla que muestra los pedidos (primera pantalla) solo necesita acceder a los datos de las entidades pedido y cliente, el query no debería involucrar a la tabla ítems ni a la jerarquía de productos
- esto se implementa definiendo ítems como un tipo especial de colección que solo se recrea en el primer momento en que se la necesite, por ejemplo aquí:

```
>>Pedido
def montoTotal() {
      items.fold (0, [ acum, item | acum + item.total])
```

Al enviar un mensaje fold (o filter, map, size, cualquier otro mensaje a la colección que requiera acceder a sus elementos) se debería disparar la consulta de ítems a la base.

6.2. Ciclo de vida de los objetos

Por lo que entendemos del negocio, el pedido y los ítems comparten el mismo ciclo de vida: no tiene sentido un ítem sin su correspondiente pedido, y tampoco tiene sentido un pedido sin ítems. Cuando se genera un pedido se suele crear también los ítems que lo componen. Y al actualizar información de un ítem, o al agregar o eliminar ítems de un pedido se debería disparar la actualización en **cascada del pedido** con todos sus ítems relacionados. Por último, si eliminamos un pedido deberíamos eliminar sus ítems asociados.

Por el contrario, los productos tienen un ciclo de vida diferente al de los pedidos:

- no podemos generar un pedido sin haber cargado al menos un producto
- y los productos se actualizan en forma independiente al pedido.

En caso de dudas, debemos disparar las preguntas al usuario que conoce el negocio. También nos puede ayudar partir de un análisis donde existan dos casos de uso diferentes

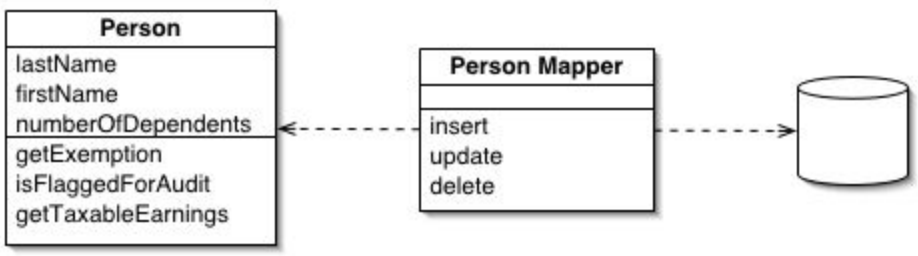
- Actualización de Productos
- Carga de Pedidos

lo que marcará dos ciclos de vida distintos para pedidos/ítems vs. productos.

7. Implementaciones

Si almacenamos nuestra información en un motor de base de datos relacional pero construimos el resto de nuestra aplicación en objetos, tenemos que tomar algunas decisiones de diseño.

7.1. Data Mapper (el Repositorio) vs. Active Record

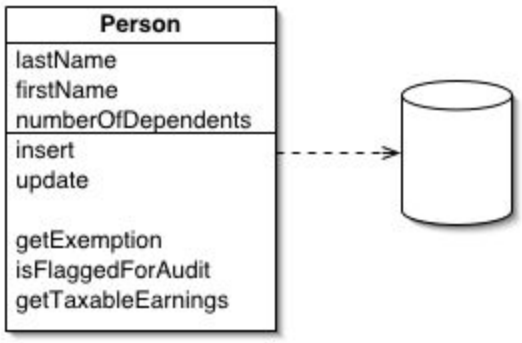


Data Mapper: en nuestros ejemplos hemos utilizado la estrategia de tener un objeto que modela el acceso a los datos: el DAO (Data Access Object), Home o Repository.

¿Qué interfaz tiene el Data Mapper para un Cliente?

- insert(Cliente)
- update(Cliente)
- delete(Cliente)
- getClientes(ClienteBusqueda) o bien getClientes(String nombre)
- getCliente(int id)

Active Record: otro approach que se popularizó a partir de frameworks como Ruby on Rails, Grails, etc. es **decorar** a los objetos de dominio agregándoles las responsabilidades de save(), update(), delete() y métodos de clase que resuelvan las búsquedas: findBy..., findAll, etc.



Para más información ver los ejemplos sobre Active Record

- Libros (con GORM, el esquema persistente que propone Grails)
- Videoclub (con GORM)

7.2. Mapeo manual o mediante un framework

El trabajo de mapear el mundo de objetos y el modelo relacional puede hacerse a mano, construyendo los queries de actualización y consulta para cada caso de uso[6].

Ventajas:

- tengo absoluto control sobre el algoritmo
    - en queries complejos esto es una ventaja, puedo trabajar al mismo nivel de SQL que en la base
    - puedo modificar la estrategia de hidratación de objetos (por ejemplo para no recrear toda la información si se que un caso de uso solo necesita el nombre de un cliente)

Desventajas:

- es un trabajo tedioso y repetitivo
- debemos definir el nivel de profundidad de lo que vamos a traer para cada uno de los casos de búsqueda
- al generar el SQL como un string que intercala datos y sentencia es difícil de mantener y dificulta la reutilización. Ej: buscar clientes activos, vs. clientes que deben más de un monto x, o bien alumnos que aprobaron 2 materias vs. los que aprobaron "Paradigmas de Programación" cuesta llevarlo a un único lugar donde se construya el query.

[6] Para ver un ejemplo concreto recomendamos descargar el siguiente ejemplo: https://github.com/uqbar-project/eg-equipos-futbol-jdbc-xtend

- requiere trabajo de bajo nivel: manejar conexiones (abrirlas y liberarlas), drivers de base de datos, etc.

Otra opción es utilizar un framework que haga el mismo trabajo, acá les dejamos una lista:
http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

Algunas características:

- los frameworks permiten subir el grado de declaratividad
  - se elimina la parte algorítmica de la conversión
  - se define un lenguaje de alto nivel para efectuar consultas
- el repository deja de tener la responsabilidad de mapeo, esto suele delegarse
  - en archivos de configuración (en formato xml o similar)
  - a annotations en los mismos objetos de dominio
  - o definiendo un DSL (domain specific language)

Ej: Hibernate que soporta las primeras dos opciones + Grails que define su propio DSL

- ¿cuál es la responsabilidad del repositorio entonces?
  - demarcar el inicio y el final de la transacción,
  - manejar la sesión o un concepto equivalente de conexión a la base, a quien delegar la actualización de los objetos de dominio (save/update/delete)
    - una ventaja que ofrecen los frameworks es que permiten manejar las referencias lazy realizando consultas mientras la sesión está activa mediante *proxies*. Ej: si estoy buscando los clientes que comiencen con Q, pero después quiero obtener el saldo de un cliente que se obtiene en base al saldo de todas las facturas pendientes, al hacer

      ```
      facturas.fold (0, [acum, factura | acum + factura.saldo ])
      ```

      en ese momento se dispara la consulta a las facturas de un cliente y así sucesivamente.
  - construir los queries de consulta en el lenguaje que cada framework ofrece
  - manejar los errores que ocurran en los llamados hacia el motor de base de datos (en la mayoría de los casos no habrá mucho para hacer más que generar una excepción de más alto nivel para avisar al usuario de que la consulta falló)

Para más información recomendamos leer http://www.hibernate.org/ y los ejemplos específicos:

- Profesores y materias

- Telefonía (versión en Scala)
- Partidos políticos (versión en Java)

8. ¿O/R o R/O?

Una vez adoptada la decisión de trabajar con bases relacionales, la pregunta es qué hacer primero:

1. Generamos el modelo relacional y luego adaptamos el modelo de objetos en base a las tablas generadas
2. Generamos el modelo de objetos y en base a éste se crean las tablas.

La opción 1) supone que es más importante la forma en que guardo los datos que las reglas de negocio que modifican esos datos. Esto puede ser cierto, si partimos de un sistema construido o enlatado, o bien si la solución OO es uno de los múltiples sistemas que accederán a este origen de datos. En la opción 2) el modelo de objetos prevalece al esquema en que se persiste. De todas maneras ya sea que vayamos por una u otra opción, es necesario comprender que debemos tomar decisiones de diseño para ajustar ambos mundos, sabiendo que es imposible mantener la pureza de cada modelo pero poniendo foco en resolver nuestro problema concreto.

9. Conclusiones

Persistir nuestro modelo de objetos en un esquema relacional requiere conocer cómo funcionan ambos mundos y entender qué decisiones de diseño están tras ese mapeo:

- manejo de la identidad
- las relaciones entre entidades
    - cardinalidad: uno a uno, uno a muchos, muchos a uno, muchos a muchos
    - dirección: cómo navego el grafo de objetos y cuánta información traigo en cada pedido
- relacionar los dominios de la base vs. los tipos que definen los objetos
- cómo manejar las colecciones (orden y duplicados)
- qué estrategia adoptar para los casos de generalización (herencia)
- cuál es la relación entre los ciclos de vida de cada entidad (cascada)
- qué índices podemos agregar para que la información sea accesible en forma más rápida
- además de las decisiones comunes a cualquier esquema de persistencia: hidratación de los objetos, persistencia del grafo en el medio, etc.

Pero además, debemos definir la arquitectura que va a dar soporte a este mapeo, adaptando la diferencia o impedance mismatch a mano o con frameworks de terceros. Y por último, debemos definir quién resuelve la conexión a la base, quién define el manejo de la transacción (cuando tenemos efecto colateral), quién dialoga con el framework o con una API de bajo nivel contra la base, y quién es el responsable de disparar las consultas y las actualizaciones.