UDE
UNIVERSIDAD DE LA EMPRESA

Bases de Datos 3

- Docentes: Federico Gómez, Diego Siri
- Carreras: Licenciatura & Ingeniería en Informática
- Año de la carrera: 3º

UDE
UNIVERSIDAD DE LA EMPRESA

Capítulo 5:

Acceso a BD en Arquitecturas de 3 Capas

UNIVERSIDAD DE LA EMPRESA

Introducción:

En este capítulo estudiaremos los fundamentos para el desarrollo de aplicaciones que utilizan ***bases de datos*** como mecanismo de ***persistencia*** en ***arquitecturas lógicas de 3 capas***.

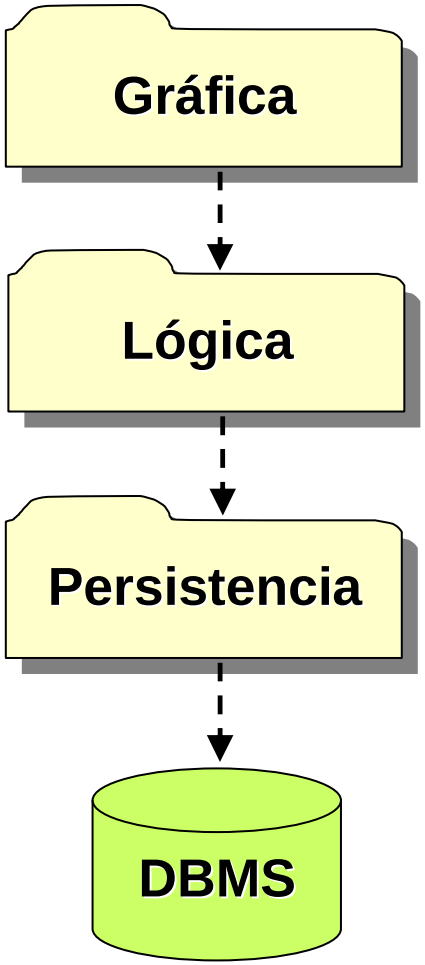
De acuerdo a lo visto en el capítulo 1, los sistemas desarrollados siguiendo arquitecturas lógicas de 3 capas cuentan con tres capas claramente diferenciadas (gráfica, lógica y persistencia).

Recordar que no existe relación específica entre la ***arquitectura lógica*** y la ***arquitectura física*** de una aplicación. Pueden existir aplicaciones de 3 capas tanto ***standalone*** como ***distribuidas***. Lo que las distingue es el hecho de que separan la presentación de la lógica y también a la lógica de la persistencia, sin importar la cantidad de equipos.

UDE
UNIVERSIDAD DE LA EMPRESA

Características de las aplicaciones en 3 capas:

- La capa gráfica se encarga de la interacción con los usuarios exclusivamente.
- La capa lógica se encarga de resolver la lógica de los requerimientos funcionales (casos de uso) de la aplicación exclusivamente.
- La capa de persistencia se encarga del respaldo y mantenimiento de la información del sistema (en BD u otro mecanismo) exclusivamente.
- Las capas gráfica y de persistencia no interactúan directamente entre sí, siempre lo hacen a través de la capa lógica.
- A nivel de código, ahora existe un límite claro entre cada una de las capas y las otras dos.



UDE
UNIVERSIDAD DE LA EMPRESA

Ventajas de las aplicaciones en 3 capas:

- Permiten migrar a una nueva capa de presentación ***sin*** tener que modificar código fuente en la lógica y persistencia (ventaja heredada de 2 capas).
- Permiten migrar a una nueva capa de persistencia ***sin*** tener que modificar código fuente en la lógica y gráfica (ventaja introducida por 3 capas).
- Permiten persistir la información en ***más de un*** mecanismo de persistencia posible, sin que esto implique modificar código fuente en las otras dos capas (ventaja introducida por 3 capas).

Las ***aplicaciones en 3 capas*** constituyen actualmente el estándar de facto para el desarrollo de nuevas aplicaciones. Alcanzan el máximo nivel posible de ***mantenibilidad*** y ***portabilidad***.

UNIVERSIDAD DE LA EMPRESA

Desventajas de las aplicaciones en 3 capas:

- Aumentan la complejidad del código fuente de la capa lógica, dado que el comportamiento (lógica de requerimientos) se resuelve mayormente en la aplicación.
- En caso de persistir en un DBMS, pueden desaprovechar las bondades del mismo para optimización de consultas y eficiencia.

Las ***aplicaciones en 3 capas*** presentan las mayores ventajas en términos de las ***cualidades del software***, pero como contraparte muchas veces se ven perjudicados aspectos de optimización de recursos y de eficiencia.

UDE
UNIVERSIDAD DE LA EMPRESA

Estrategias para separación en 3 capas:

Al igual que para la separación en ***2 capas***, un mecanismo muy difundido para la separación en ***3 capas*** es el uso de ***patrones de diseño*** en la aplicación.

Los patrones anteriormente estudiados para separación en 2 capas (***Facade***, ***Value Object***, ***MVC***) se siguen utilizando para separación en 3 capas. Separan la capa gráfica de la capa lógica, al igual que lo hacían en las aplicaciones en 2 capas.

En este capítulo presentaremos dos nuevos patrones que se usan para separar la capa lógica de la capa de persistencia:

- ***Data Access Object***
- ***Abstract Factory***

Data Access Object:

Este patrón apunta a desacoplar (del resto de la aplicación) el acceso al mecanismo concreto de persistencia usado. Resuelve el establecimiento de un límite entre la capa lógica y la capa de persistencia.

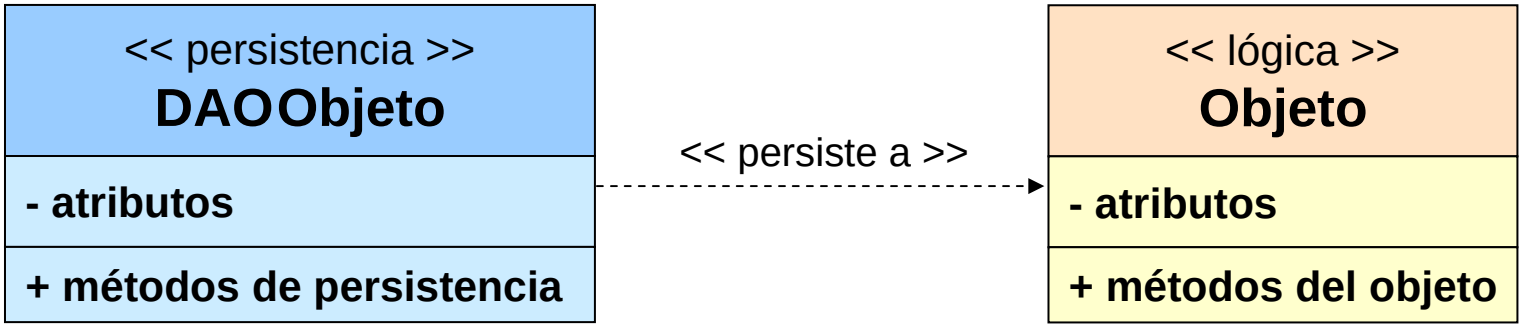
Problema:

- Se desea poder almacenar datos en forma persistente en algún mecanismo de almacenamiento (archivos planos, archivos XML, bases de datos, sistemas legados, etc.).
- Se desea ocultar dicho mecanismo de almacenamiento al resto de la aplicación (capa gráfica y capa lógica).

Data Access Object (continuación):

Solución:

- Para cada clase de objetos de la capa lógica que interese persistir, definir una clase denominada Data Access Object (DAO) que provea los métodos necesarios para acceder al mecanismo de persistencia de dichos objetos.



Observación: La flecha punteada en UML representa ***dependencia*** entre objetos. Significa que la clase origen hace uso de los objetos de la clase destino.

UNIVERSIDAD DE LA EMPRESA

Data Access Object (continuación):

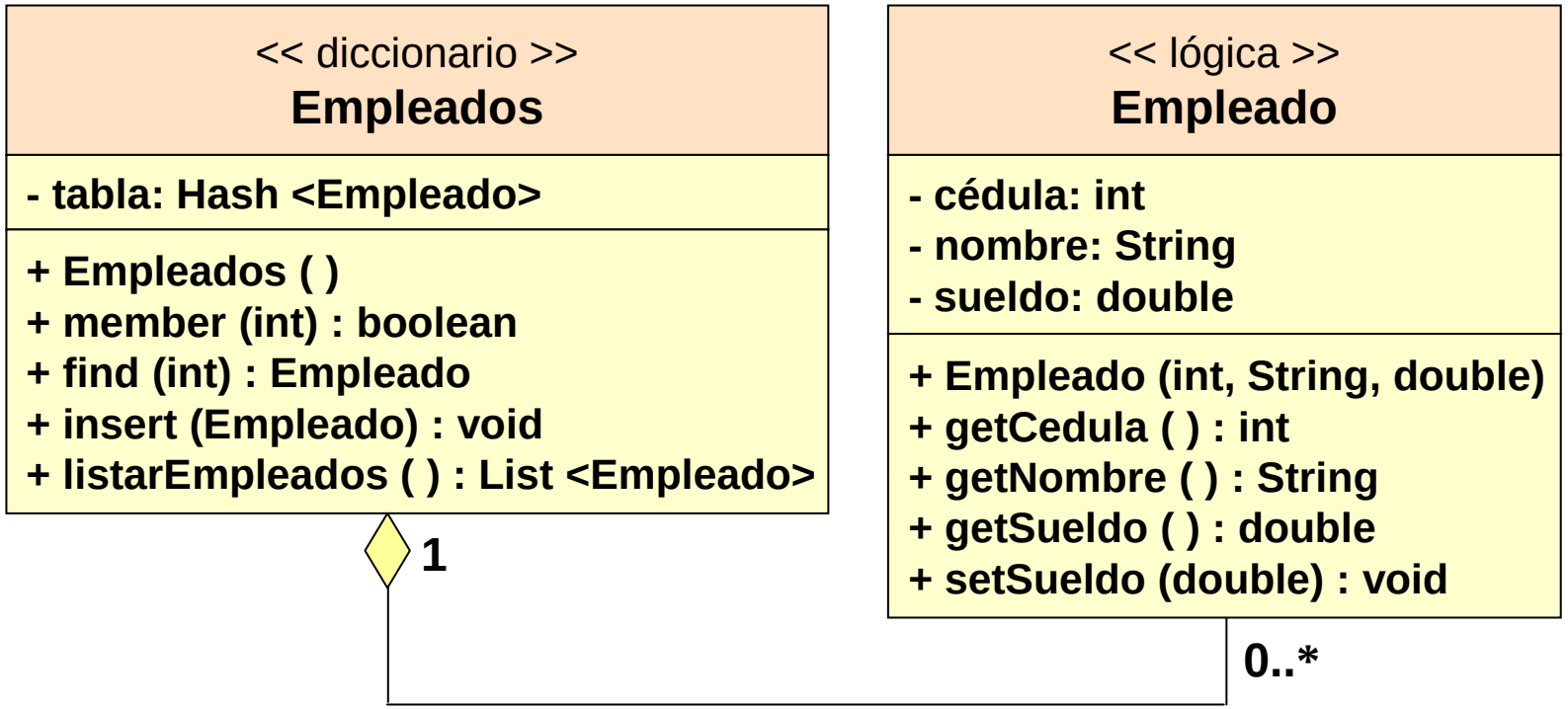
Consecuencias:

- El mecanismo concreto de almacenamiento utilizado queda ***encapsulado*** en forma interna a las clases DAO definidas.
- Los objetos pertenecientes a la capa lógica no conocen cuál es el mecanismo concreto utilizado para persistirlos.
- No obstante, para ***migrar*** de un mecanismo de persistencia a otro, es necesario ***modificar*** el código fuente de las clases DAO.

Observación: Sería deseable poder cambiar de mecanismo de persistencia ***sin*** tener que modificar el código de las clases DAO. Veremos que el patrón ***Abstract Factory*** resuelve este problema.

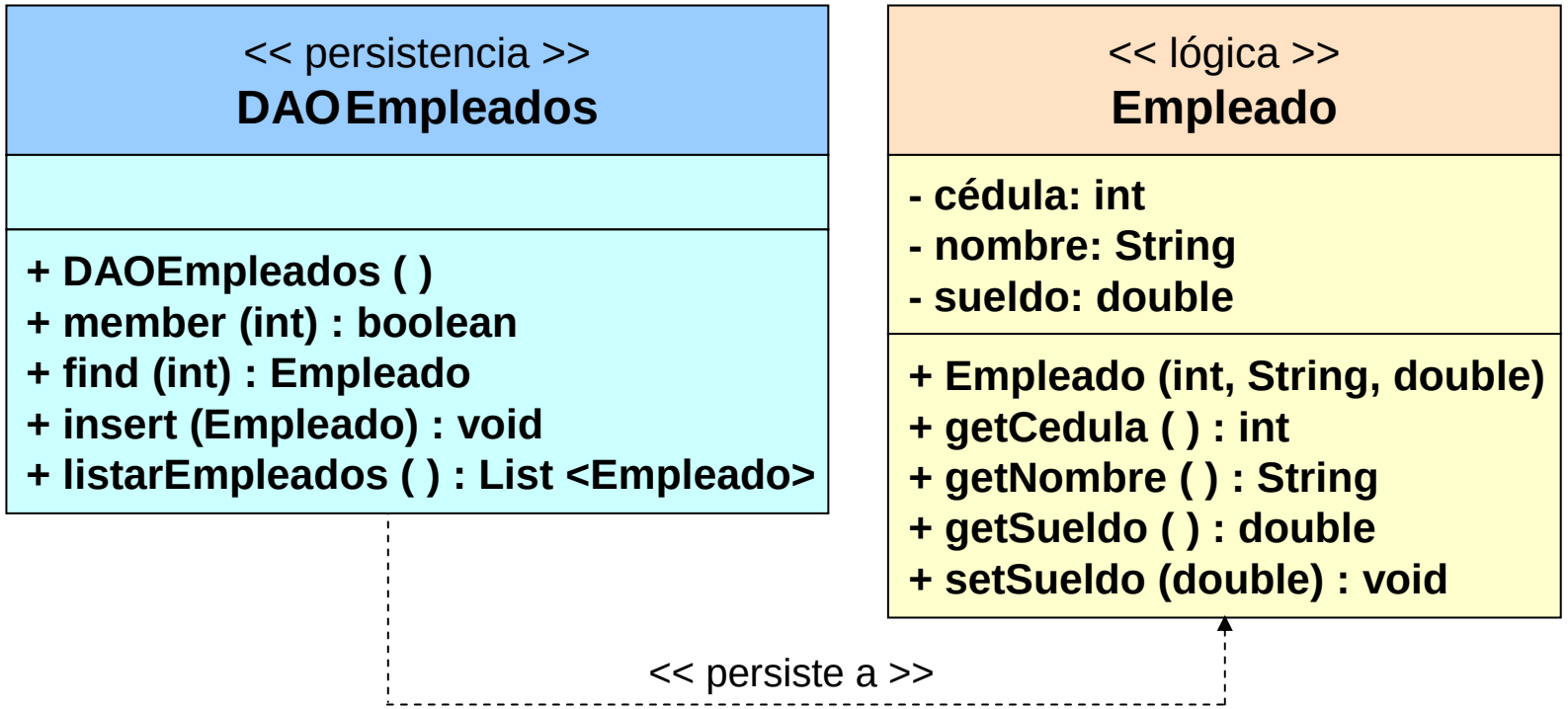
Ejemplo de aplicación de D.A.O:

Considere el siguiente diagrama de clases de implementación en **UML** para un sistema de gestión contable. Por el momento, los empleados se guardan en una colección de objetos en ***memoria***.



Ejemplo de aplicación de D.A.O (continuación):

Vamos a sustituir la colección de objetos en memoria por una clase DAO que mantenga las propiedades del diccionario pero guarde los empleados en una ***base de datos*** en vez de hacerlo en memoria.



UNIVERSIDAD DE LA EMPRESA

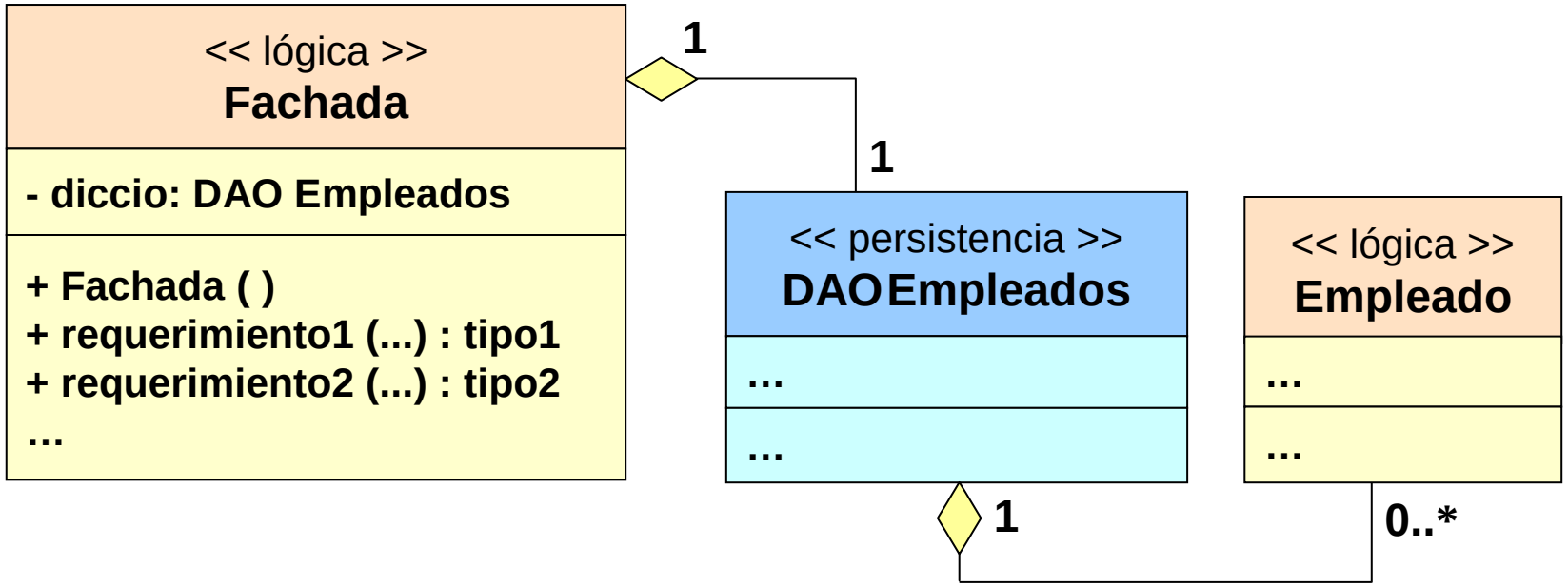
Ejemplo de aplicación de D.A.O (continuación):

- La clase D.A.O posee exactamente los ***mismos*** métodos que tenía el diccionario de objetos en memoria.
- Lo que cambia es la ***implementación*** de dichos métodos. En lugar de acceder a una estructura de ***hash*** en memoria, acceden al mecanismo de persistencia utilizado (la base de datos).
- Desde el punto de vista conceptual, la clase D.A.O ***mantiene*** las propiedades del diccionario. Las demás clases (incluida la clase Empleado) ***desconocen*** que ahora se trabaja sobre una BD en lugar de trabajarse en memoria.

Observación: Si bien el patrón marca una ***dependencia*** entre la clase D.A.O y la clase Empleado, en el diagrama igualmente se suele mantener la ***agregación*** para no perder la información de las ***multiplicidades***.

Ejemplo de aplicación de D.A.O (continuación):

Suponga que además se incorpora al diagrama una clase ***Fachada*** de acceso a la capa lógica. Posee los métodos para resolver los requerimientos funcionales de la aplicación y hace uso del DAO de modo ***idéntico*** a como haría uso del diccionario en memoria.



UNIVERSIDAD DE LA EMPRESA

Ejemplo de aplicación de D.A.O (continuación):

Vamos a implementar el requerimiento que realiza el alta de un nuevo empleado en el sistema. Por el momento, supondremos que se trata de una arquitectura física ***standalone*** y por lo tanto ***no*** es necesario realizar ***manejo de concurrencia***.

- Implementaremos una clase **Consultas** que define los textos de las consultas SQL a ejecutar (como en 1 y 2 capas).
- Implementaremos los métodos **member** e **insert** del **D.A.O** de Empleados.
- Implementaremos el método **altaEmpleado** de la **Fachada** que da solución al requerimiento.
- Asumiremos ya implementadas la clase **Empleado** y la clase **VOEmpleado** (Value Object) que trae los datos de capa gráfica.

UDE
UNIVERSIDAD DE LA EMPRESA

Ejemplo de aplicación de D.A.O (continuación):

Implementación de la clase **Consultas**:

```
public class Consultas {
    public String existsCedula() {
        String query = "SELECT cedula FROM " +
                "Empleados WHERE cedula = ?";
        return query;
    }
    public String insertEmpleado() {
        String insert = "INSERT INTO Empleados " +
         "(cedula, nombre, sueldo) VALUES (?,?,?)";
        return insert;
    }
}
```

Ejemplo de aplicación de D.A.O (continuación):

Implementación del método **member** de la clase **DAOEmpleados**:

```
public boolean member (int ced) throws ...
{
   boolean existeCedula = false;
   try
   {
      Connection con =
        DriverManager.getConnection(url,usr,pwd);
      Consultas consultas = new Consultas();
      String query = consultas.existsCedula();
      PreparedStatement pstmt1 =
        con.prepareStatement(query);
      ...
```

Ejemplo de aplicación de D.A.O (continuación):

```
            ...
            pstmt1.setInt(1, ced);
            ResultSet rs = pstmt1.executeQuery();
            if (rs.next())
               existeCedula = true;
            rs.close();
            pstmt1.close();
            con.close();
        }
        catch (SQLException e)
               throw new PersistenciaException(...);
        return existeCedula;
}
```

Ejemplo de aplicación de D.A.O (continuación):

Implementación del método **insert** de la clase **DAOEmpleados**:

```
public boolean insert (Empleado emp) throws ...
{
   boolean existeCedula = false;
   try
   {
      Connection con =
        DriverManager.getConnection(url,usr,pwd);
      Consultas consultas = new Consultas();
      String insert = consultas.insertEmpleado();
      PreparedStatement pstmt2 =
        con.prepareStatement(insert);
      ...
```

Ejemplo de aplicación de D.A.O (continuación):

```
            ...
            pstmt2.setInt (1, emp.getCedula());
            pstmt2.setString (2, emp.getNombre());
            pstmt2.setDouble (3, emp.getSueldo());
            pstmt2.executeUpdate ();
            pstmt2.close ();
            con.close ();
        }
        catch (SQLException e)
                throw new PersistenciaException(...);
}
```

Ejemplo de aplicación de D.A.O (continuación):

Implementación del método **altaEmpleado** de la clase **Fachada**:

```
public void altaEmpleado (VOEmpleado vo) throws
            EmpleadoException, PersistenciaException
{
   int ced = vo.getCedula();
   if (!diccio.member(ced))
   {
      String nom = vo.getNombre();
      double sue = vo.getSueldo();
      Empleado emp = new Empleado(ced,nom,sue);
      diccio.insert(emp);
   }
   else
      throw new EmpleadoException(...);
}
```

UNIVERSIDAD DE LA EMPRESA

Ejemplo de aplicación de D.A.O (continuación):

Observaciones:

- La clase Fachada hace uso del D.A.O de Empleados de modo ***idéntico*** a como haría uso del diccionario en memoria. El acceso a la BD queda totalmente ***encapsulado*** en el D.A.O.

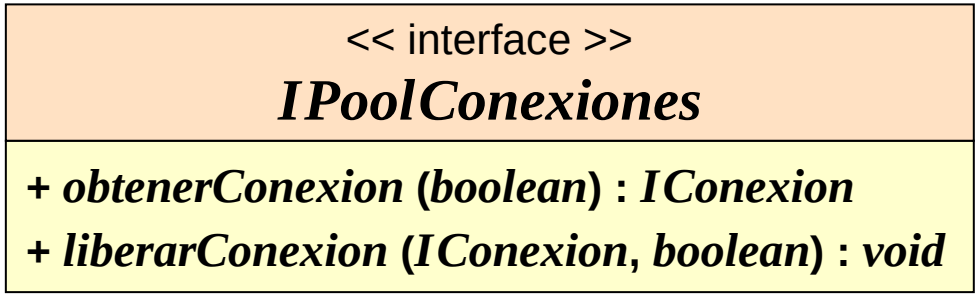
- La fachada es responsable de convertir los ***Value Objects*** (en el ejemplo: **VOEmpleado**) a clases internas a la capa lógica (en el ejemplo: **Empleado**).

- Cuando un Value Object resulta ***idéntico*** a su correspondiente de la capa lógica (como sucede en el ejemplo), en ocasiones se suele prescindir del Value Object. Esto ahorra código fuente, pero como contraparte ***perjudica*** a la separación en capas, ya que la capa gráfica ahora ***conoce*** un objeto de la capa lógica.

Manejo de concurrencia en conjunto con D.A.O:

Supongamos ahora que el ejemplo anterior posee una arquitectura física ***distribuida***, esto obliga a realizar ***manejo de concurrencia***, dado que varios clientes accederán a la ***fachada*** en paralelo.

Para ello, incorporaremos a la fachada del ejemplo anterior el uso de un ***pool de conexiones*** (introducido en 2 capas). Incluiremos en la fachada el atributo: `private IPoolConexiones pool;`



Observación: Nótese que, al igual que en 2 capas, la interface ***IPoolConexiones*** define un pool de conexiones ***abstracto***, que puede poseer diversas implementaciones posibles.

UNIVERSIDAD DE LA EMPRESA

Manejo de concurrencia en conjunto con D.A.O (cont.):

En el ejemplo anterior, el pedido de conexiones se realizaba en forma ***interna*** a cada método del D.A.O, dado que no era necesario manejar ***transacciones*** (pues la arquitectura era ***standalone***).

Dado que ahora se tiene un ambiente ***concurrente***, las conexiones ***no*** serán solicitadas dentro de cada método, sino que ahora serán obtenidas del ***pool de conexiones*** dentro de la ***fachada***.

El método **altaEmpleado** de la clase **Fachada** ahora obtendrá una conexión del pool y se la pasará como parámetro a los métodos del D.A.O invocados durante la ejecución del requerimiento. Una vez finalizado el mismo, la conexión será devuelta al pool.

Nótese que ahora todos los métodos del D.A.O harán uso de la ***misma*** conexión durante la ejecución del requerimiento.

UNIVERSIDAD DE LA EMPRESA

Manejo de concurrencia en conjunto con D.A.O (cont.):

```
public void altaEmpleado (VOEmpleado vo) throws
            EmpleadoException, PersistenciaException
{
   try
   {
     IConexion icon = pool.obtenerConexion(true);
     int ced = vo.getCedula();
     boolean existCed = diccio.member(icon, ced);
     if (!existCed)
     {
        String nom = vo.getNombre();
        double sue = vo.getSueldo();
        ...
```

Manejo de concurrencia en conjunto con D.A.O (cont.):

```
            ...
            Empleado emp = new Empleado(ced,nom,sue);
            diccio.insert(icon, emp);
        }
        else
            msgError = "empleado ya registrado";
        pool.liberarConexion(icon, true);
    }
    catch (Exception e)
    {
        pool.liberarConexion(icon, false);
        errorPersistencia = true;
        msgError = "error de acceso a los datos";
    }
    ...
```

Manejo de concurrencia en conjunto con D.A.O (cont.):

```
    ...
    finally
    {
      if (existCed)
         throw new EmpleadoException (msgError);
      if (errorPersistencia)
         throw new PersistenciaException (msgError);
    }
}
```

Observaciones:

- La lógica correspondiente a la resolución del requerimiento sigue siendo la ***misma*** que antes, con el agregado de la ***concurrencia***

UDE
UNIVERSIDAD DE LA EMPRESA

Manejo de concurrencia en conjunto con D.A.O (cont.):

Observaciones:

- El uso del pool se hace de modo ***idéntico*** a como en 2 capas, solicitando la conexión al comienzo y liberándola al final.
- Al tratarse de un pool ***abstracto*** de conexiones, ***no*** es visible en el código fuente el tipo de conexión ***concreta*** que se utiliza, dado que se trabaja con una conexión ***abstracta*** (***IConexion***)
- Dependiendo de la implementación concreta, puede tratarse de una conexión a una ***base de datos***, pero también podría ser una conexión a algún ***otro*** mecanismo de persistencia.
- Sea cual sea, el mecanismo concreto utilizado queda ***oculto*** detrás de los métodos del D.A.O y del pool de conexiones ***abstracto*** (***separación*** entre la capa lógica y la de persistencia)

UNIVERSIDAD DE LA EMPRESA

Mapeo entre OO y BD:

Cuando se trabaja con un DBMS ***relacional*** como mecanismo de persistencia, es necesario traducir (mapear) el diseño ***orientado a objetos*** a tablas de la BD relacional.

Veremos en esta sección algunas técnicas habitualmente utilizadas para establecer una correspondencia (mapeo) entre una base de datos relacional y un diseño OO en aplicaciones en ***3 capas***.

Las técnicas que presentaremos ***no*** son infalibles y tampoco son las únicas que existen. La correspondencia entre lenguajes OO y BD relacionales es un ***problema abierto*** que ***no*** cuenta con una solución perfecta hasta la fecha. Existen múltiples técnicas posibles cada una presenta pros y contras.

UNIVERSIDAD DE LA EMPRESA

Mapeo entre OO y BD (continuación):

En el modelo relacional cada ***entidad*** se representa usualmente mediante una tabla de la BD. En el modelo OO, cada ***clase*** de objetos se almacena usualmente en alguna colección de objetos.

En principio, cada entidad del modelo relacional corresponde a una clase del modelo OO. En un primer mapeo, los objetos de una misma colección se almacenarían en una misma tabla de la BD.

Lo anterior constituye un ***punto de arranque*** para el mapeo entre ambos modelos, pero ***no*** es de ninguna forma una regla universal. Dependiendo del diseño de la aplicación, otros mapeos pueden resultar convenientes. No existe una receta universalmente válida. Existen múltiples técnicas de mapeo OO – BD aplicables.

UNIVERSIDAD DE LA EMPRESA

Mapeo entre OO y BD (continuación):

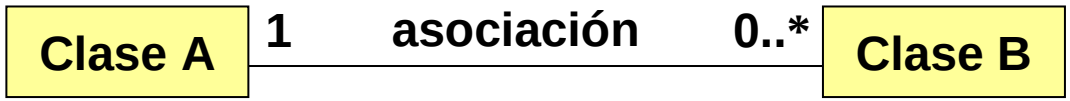
Presentaremos cinco de las técnicas más comunes para realizar el mapeo entre ambos modelos (relacional y orientado a objetos). Cada una toma un tipo de asociación surgido del ***análisis*** y le hace corresponder un ***diseño*** OO y un ***esquema de tablas*** de BD que sean compatibles con el tipo de asociación.

Las cinco técnicas a estudiar son las siguientes:

1) Mapeo de asociaciones "***1 a muchos***".
2) Mapeo de asociaciones "***muchos a 1***".
3) Mapeo de asociaciones "***muchos a muchos***".
4) Mapeo de asociaciones con ***clases de asociación***.
5) Mapeo de ***herencia***.

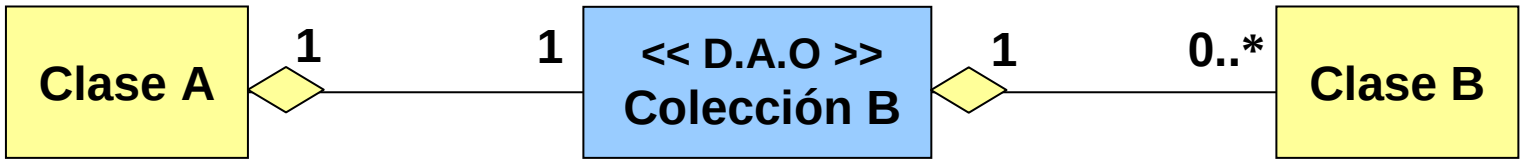
1) Mapeo de asociaciones "*1 a muchos*":

Durante el análisis se propuso la siguiente asociación:



Supongamos que los requerimientos indican que la ***navegabilidad*** va de ***izquierda*** a ***derecha***. Esto es, que cada objeto de clase A debe poder acceder a los objetos de clase B que le corresponden.

Un diseño orientado a objetos comúnmente usado para representar esta asociación es el siguiente:



1) Mapeo de asociaciones "*1 a muchos*" (continuación):

El esquema de tablas de la BD que mejor representa a esta asociación sería el siguiente:

Tabla A (clave de A, atributos de A)

Tabla B (atributos de B, clave de A)

Cada tupla de la tabla A contiene los datos de un objeto de clase A. Cada tupla de la tabla B contiene los datos de un objeto de clase B junto con la clave (foránea) del objeto de clase A correspondiente.

Tabla A

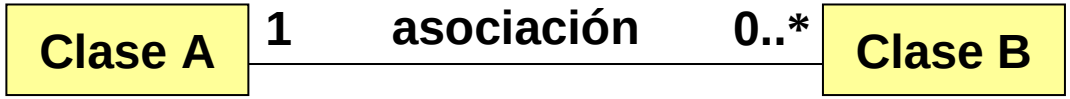
clave A1	atributos A1
clave A2	atributos A2

Tabla B

atributos B11	clave A1
atributos B12	clave A1
atributos B21	clave A2
atributos B22	clave A2

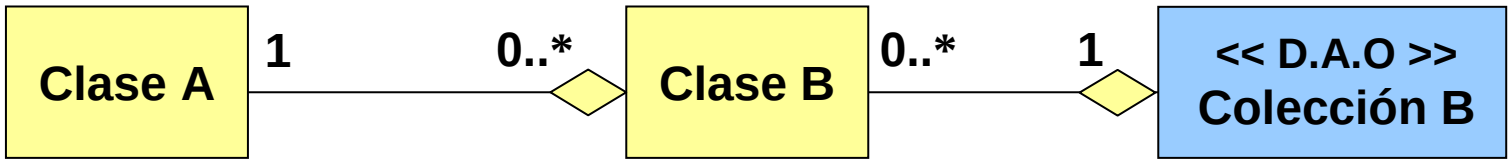
2) Mapeo de asociaciones "*muchos a 1*":

Durante el análisis se propuso la siguiente asociación:



Supongamos que los requerimientos indican que la ***navegabilidad*** va de ***derecha*** a ***izquierda***. Esto es, que cada objeto de clase B debe poder acceder al (único) objeto de clase A que le corresponde

Un diseño orientado a objetos comúnmente usado para representar esta asociación es el siguiente:



2) Mapeo de asociaciones "*muchos a 1*" (continuación):

El esquema de tablas de la BD que mejor representa a esta asociación sería ***idéntico*** que para el caso "*1 a muchos*":

Tabla A (clave de A, atributos de A)

Tabla B (atributos de B, clave de A)

Cada tupla de la tabla A contiene los datos de un objeto de clase A. Cada tupla de la tabla B contiene los datos de un objeto de clase B junto con la clave (foránea) del objeto de clase A correspondiente.

Tabla A

clave A1	atributos A1
clave A2	atributos A2

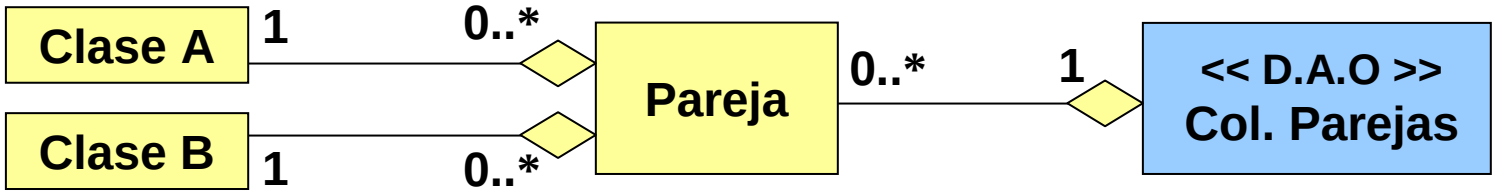
Tabla B

atributos B11	clave A1
atributos B12	clave A1
atributos B21	clave A2
atributos B22	clave A2

3) Mapeo de asociaciones "*muchos a muchos*":

Durante el análisis se propuso la siguiente asociación:

En forma ***independiente*** de la navegabilidad indicada por los requerimientos de la aplicación, un diseño OO comúnmente usado para representar esta asociación es el siguiente:



La clase **Pareja** se incluye en el diseño para vincular a cada pareja de objetos de clase A y clase B asociados en el análisis.

3) Mapeo de asociaciones "*muchos a muchos*" (continuación):

El esquema de tablas de la BD que mejor representa a esta asociación sería el siguiente:

Tabla A (clave de A, atributos de A)

Tabla B (clave de B, atributos de B)

Tabla Parejas (clave de A, clave de B)

Cada tupla de las tablas A y B contiene los datos de un objeto de clase A y clase B respectivamente. Cada tupla de la tabla Parejas contiene las claves de los objetos de clase A y clase B relacionados

Tabla A

clave A1	atributos A1
clave A2	atributos A2
clave A3	atributos A3

Tabla B

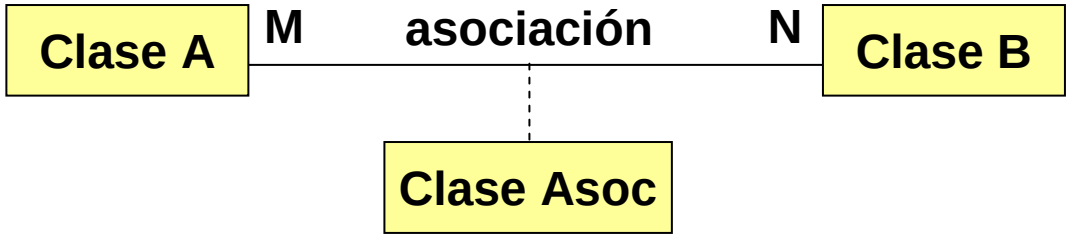
clave B1	atributos B1
clave B2	atributos B2
clave B3	atributos B3

Tabla Parejas

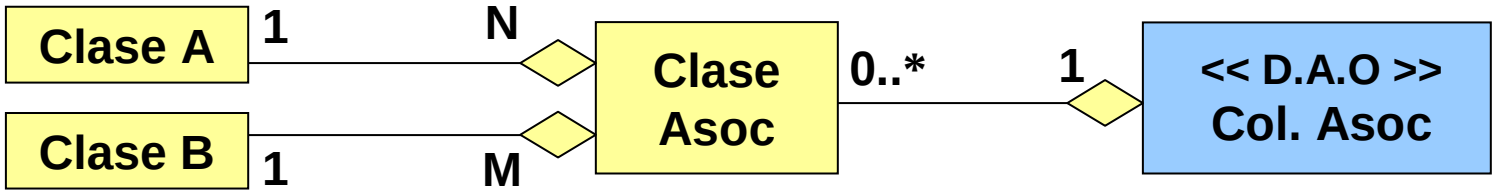
clave A1	clave B1
clave A1	clave B2
clave A3	clave B2

4) Mapeo de asociaciones con *clases de asociación*:

Durante el análisis se propuso la siguiente asociación:



En forma ***independiente*** de la navegabilidad indicada por los requerimientos y de las multiplicidades **M** y **N**, un diseño OO usado comúnmente para representar esta asociación es el siguiente:



Nótese que se trata del ***mismo*** diseño que el caso anterior, con la diferencia que la clase de asociación posee además datos propios

4) Mapeo de asociaciones con *clases de asociación* (cont.):

El esquema de tablas de la BD que mejor representa a esta asociación sería el siguiente:

Tabla A (clave de A, atributos de A)

Tabla B (clave de B, atributos de B)

Tabla Asoc (clave de A, clave de B, atributos de Asoc)

Se trata del ***mismo*** esquema de tablas del caso anterior, con el único agregado de los datos propios de la clase de asociación.

Tabla A

clave A1	atribs. A1
clave A2	atribs. A2
clave A3	atribs. A3

Tabla B

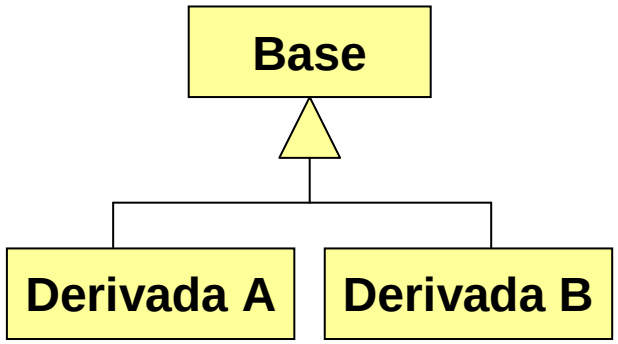
clave B1	atribs. B1
clave B2	atribs. B2
clave B3	atribs. B3

Tabla Asoc

clave A1	clave B1	datos A1-B1
clave A1	clave B2	datos A1-B2
clave A3	clave B2	datos A3-B2

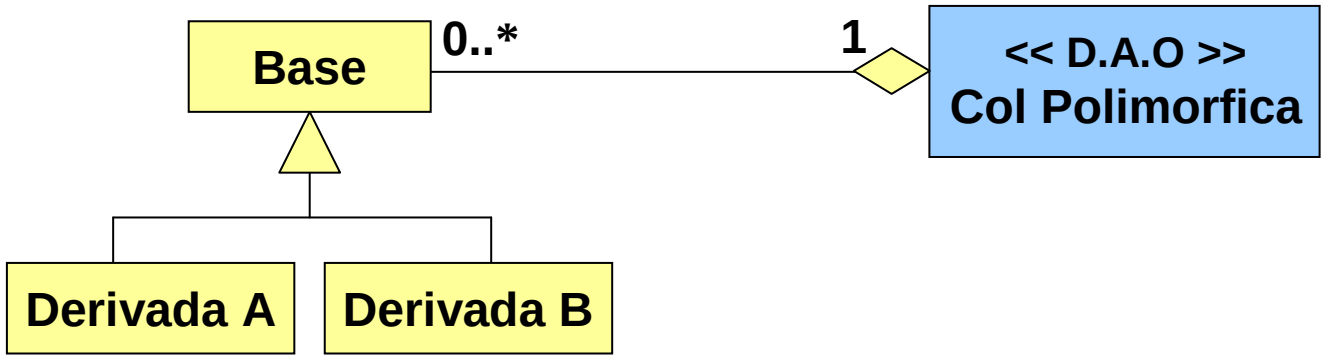
5) Mapeo de *herencia*:

Durante el análisis se propuso la siguiente herencia:



La clase base eventualmente puede ser ***abstracta***. También puede haber cualquier cantidad de clases ***derivadas*** (en la figura se muestran solamente dos).

Un diseño orientado a objetos comúnmente usado para representar esta jerarquía de herencia es el siguiente:



5) Mapeo de *herencia* (continuación):

Cada una de las cuatro técnicas vistas antes proponía un ***único*** esquema de tablas que representa el diseño correspondiente. Sin embargo, para el mapeo de herencia suelen proponerse ***tres*** esquemas posibles, que enumeramos a continuación:

A. Una tabla ***única*** para todas las clases de la jerarquía.

B. Una tabla para la ***clase base*** (atributos comunes) más una tabla por cada ***clase derivada*** (atributos específicos).

C. Una tabla ***separada*** para cada clase (base y derivadas), cada una conteniendo ***todos*** los atributos necesarios.

Ninguno de los tres esquemas es infalible. Tampoco hay uno que predomine sobre los otros dos. Dependiendo del problema, se elige aquel que mejor se ajuste a la realidad de la aplicación.

5) Mapeo de *herencia* (continuación):

Esquema **A**: Una tabla ***única*** para todas las clases de la jerarquía.

Tabla única (atributos base, atributos der.A, atributos der.B)

Cada tupla almacena únicamente los atributos que necesita, todos los demás quedan con ***valores nulos***. Los atributos ***base*** siempre contienen valores ***no nulos*** en ***todas*** las tuplas.

Cuando una tupla posee valores ***no nulos*** en los atributos de una clase derivada, ***necesariamente*** debe poseer valores nulos en los atributos de las demás clases derivadas.

Tabla única

datos base 1	**NULL**	**NULL**
datos base 2	**datos derivada A2**	**NULL**
datos base 3	**NULL**	**datos derivada B3**

5) Mapeo de *herencia* (continuación):

Esquema **B**: Una tabla para la ***clase base*** (atributos comunes) más una tabla por cada ***clase derivada*** (atributos específicos).

Tabla base (clave base, atributos base)
Tabla derivada A (atributos der.A, clave base)
Tabla derivada B (atributos der.B, clave base)

Cada tupla de tabla base guarda datos de un objeto de clase base. Puede ser propio de la clase base o de una clase derivada. Los atributos específicos se guardan en la tabla de dicha clase derivada junto con la clave (foránea) que referencia a la tupla de tabla base.

Tabla base

clave base 1	datos base 1
clave base 2	datos base 2
clave base 3	datos base 3

Tabla derivada A

datos der. A2	clave base 2

Tabla derivada B

datos der. B3	clave base 3

5) Mapeo de *herencia* (continuación):

Esquema **C**: Una tabla ***separada*** por cada clase (base y derivadas) cada una conteniendo ***todos*** los atributos necesarios.

Tabla base (atributos base)
Tabla derivada A (atributos base, atributos der.A)
Tabla derivada B (atributos base, atributos der.B)

Solamente los datos de los objetos ***propios*** de la clase base están en tabla base. ***Todos*** los atributos de los objetos de cada clase derivada están en la tabla correspondiente a dicha clase derivada.

Tabla base	Tabla derivada A		Tabla derivada B	
datos base 1	datos base 2	datos der. A2	datos base 3	datos der. A3

Nótese la ***ausencia*** de valores nulos tanto en este esquema como en el esquema anterior (esquema B).

5) Mapeo de *herencia* (continuación):

Cada uno de los tres esquemas presenta ventajas y desventajas, algunas de los cuales enumeramos a continuación:

Esquema **A**: Una tabla ***única*** para todas las clases de la jerarquía.

Ventajas:

- La inserción de un nuevo objeto implica la inserción de datos en una ***única*** tabla.
- Las consultas y listados se realizan sobre una ***única*** tabla.

Desventajas:

- Si se agrega una nueva clase derivada, hay que ***agregar columnas*** a la tabla.
- Se trabaja con ***valores nulos***.

5) Mapeo de *herencia* (continuación):

Esquema **B**: Una tabla para la ***clase base*** (atributos comunes) más una tabla por cada ***clase derivada*** (atributos específicos).

Ventajas:

- Si se agrega una nueva clase derivada, ***no*** es necesario modificar las tablas existentes.
- ***No*** se trabaja con valores nulos.

Desventajas:

- La inserción de un nuevo objeto de una clase derivada implica la inserción de datos en ***dos tablas***.
- Las consultas y listados se realizan sobre ***varias tablas***.

5) Mapeo de *herencia* (continuación):

Esquema **C**: Una tabla ***separada*** por cada clase (base y derivadas) cada una conteniendo ***todos*** los atributos necesarios.

Ventajas:

- Si se agrega una nueva clase derivada, ***no*** es necesario modificar las tablas existentes.
- La inserción de un nuevo objeto implica la inserción de datos en una ***única*** tabla.
- ***No*** se trabaja con valores nulos.

Desventajas:

- Si se agregan atributos correspondientes a la clase base, es necesario modificar ***todas las tablas***.
- Las consultas y listados se realizan sobre ***varias tablas***.

UDE
UNIVERSIDAD DE LA EMPRESA

Mapeo entre OO y BD (continuación):

Las técnicas anteriores constituyen prácticas habitualmente usadas para hacer el mapeo entre ambos modelos. Pueden tomarse como referencia, pero de ninguna manera abarcan todas las posibilidades de diseño. ***No*** son soluciones universales.

El diseño de toda aplicación es una tarea ***compleja***, que admite múltiples posibilidades. Así como en cada sistema OO pueden existir muchas opciones de diseño, en consecuencia pueden existir muchas alternativas de mapeo entre los dos modelos.

*"Diseñar es un **arte** más que una **ciencia**"*

Esta frase dice mucho acerca de lo que implica diseñar. Se trata de combinar creatividad con técnicas metódicas. Ninguna técnica es infalible y siempre se pueden aplicar variantes o técnicas nuevas.

UNIVERSIDAD DE LA EMPRESA

Migración a un nuevo mecanismo de persistencia:

Vimos que el patrón ***D.A.O*** permite desacoplar (del resto de la aplicación) el acceso al mecanismo concreto de persistencia usado. Establece un límite entre la capa lógica y la capa de persistencia.

Sin embargo, en caso de querer ***migrar*** a un nuevo mecanismo de persistencia (por ejemplo a otro motor de BD, o a una estructura de archivos XML) es necesario ***modificar*** el código de las clases DAO.

El patrón ***Abstract Factory*** resuelve este problema. Posibilita el cambio de mecanismo de persistencia ***sin*** tener que modificar el código de las clases DAO (ni ninguna otra línea de código).

Presentaremos primero este patrón en su ***forma pura***, y luego lo aplicaremos en forma conjunta con ***D.A.O***.

Abstract Factory:

Este patrón apunta a manipular familias de objetos relacionados entre sí (en adelante, ***familias de productos***) sin necesidad de conocer a qué clases en concreto pertenecen.

Problema:

- Se desea poder instanciar productos de una determinada familia sin conocer las clases concretas a las que pertenecen.
- Se desea poder sustituir una familia de productos por otra familia sin necesidad de modificar ninguna línea de código fuente.

UNIVERSIDAD DE LA EMPRESA

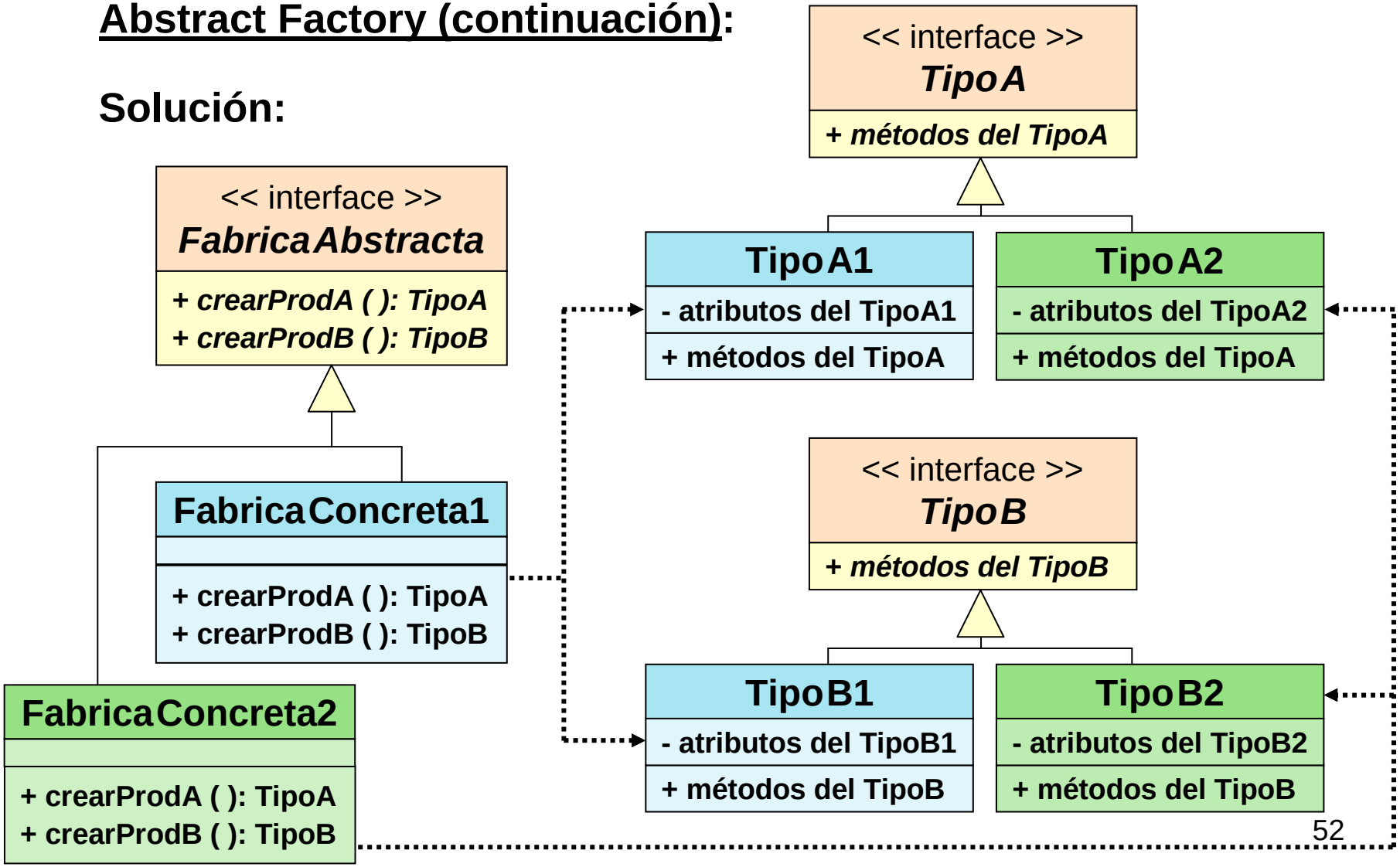
Abstract Factory (continuación):

Solución:

- Definir una interface (o una clase abstracta) por cada tipo de producto que se vaya a manejar.
- Definir una clase que implemente la interface de cada producto. El conjunto de dichas clases corresponde a una ***familia***.
- Definir una interface (o una clase abstracta) con métodos abstractos que permitan instanciar cada tipo de producto (la ***fábrica abstracta***).
- Por cada familia, definir una clase que implemente los métodos de la fábrica abstracta (la ***fábrica concreta***) de modo que creen instancias concretas de los tipos de productos de la familia.

Abstract Factory (continuación):

Solución:



UNIVERSIDAD DE LA EMPRESA

Abstract Factory (continuación):

Consecuencias:

- El cliente (porción de código que manipula a los productos) ***no conoce*** cuál es la implementación concreta de los productos manipulados.
- Promueve la ***consistencia*** en el manejo de productos (evita que se manejen en simultáneo productos de familias distintas).
- Agregar una nueva familia resulta ***sencillo*** (se crea una nueva fábrica concreta y nuevos productos concretos, sin modificar nada de código fuente).
- Agregar un nuevo tipo de producto resulta ***complejo*** (se debe crear una nueva interface y agregarle un nuevo método tanto a la fábrica abstracta como a las fábricas concretas).

UNIVERSIDAD DE LA EMPRESA

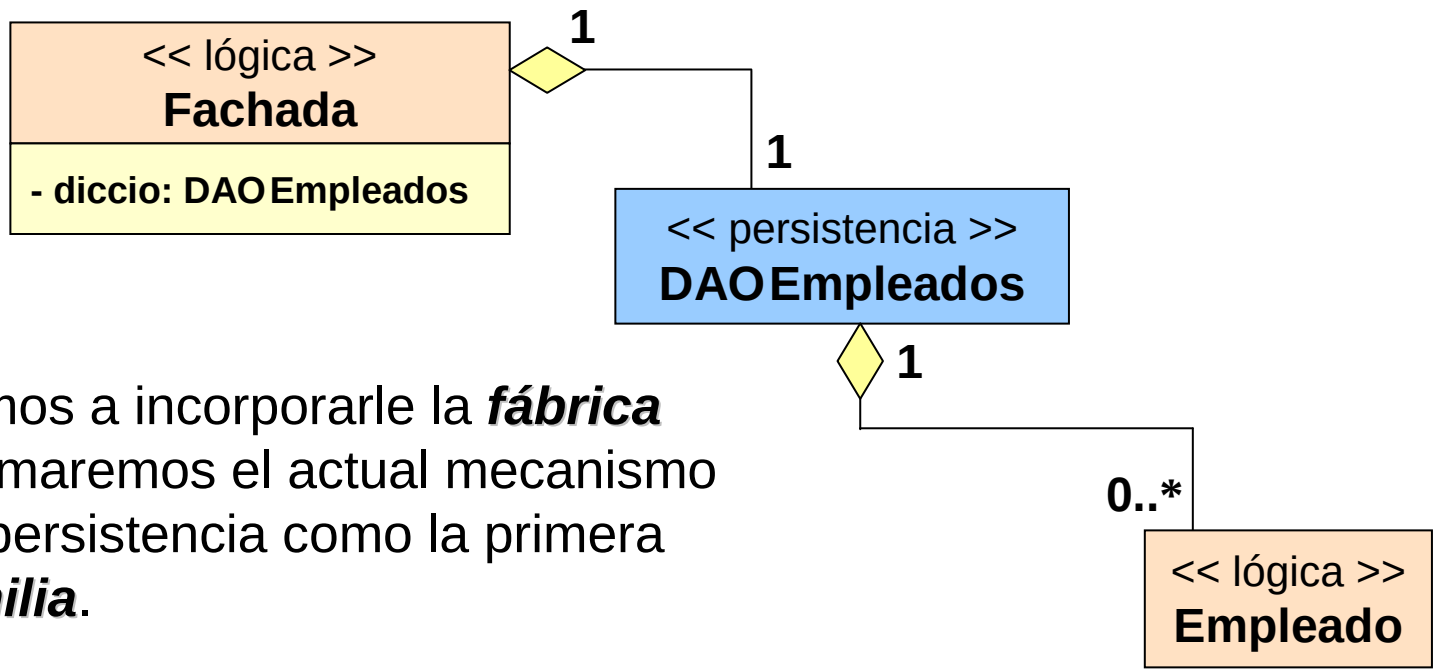
Migración a un nuevo mecanismo de persistencia (cont.):

El patrón ***D.A.O*** combinado con ***Abstract Factory*** permite migrar de un mecanismo de persistencia a otro sin necesidad de modificar código fuente. La combinación se realiza según las siguientes consideraciones:

- Cada mecanismo de persistencia constituye una ***familia*** del patrón Abstract Factory.
- Cada tipo de D.A.O concreto constituye un ***producto*** del patrón Abstract Factory.
- Se crea una ***fábrica concreta*** por cada mecanismo de persistencia a manipular.
- Se incorpora un método a cada fábrica por cada tipo de ***D.A.O*** concreto que se vaya a manipular.

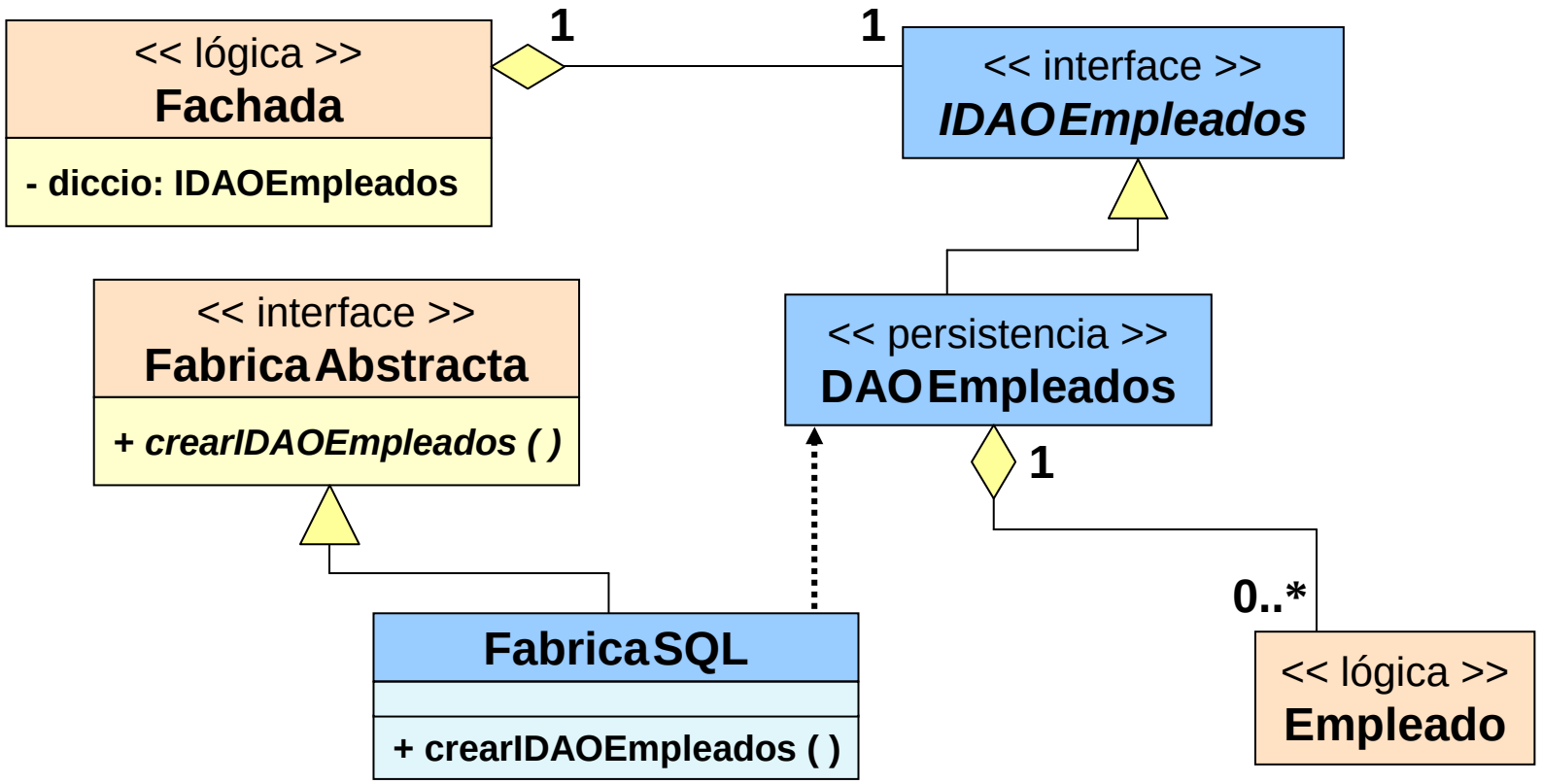
Migración a un nuevo mecanismo de persistencia (cont.):

A continuación, modificamos el primer ejemplo de aplicación de ***D.A.O*** a efectos de poder migrar a un segundo mecanismo de persistencia. Consideremos la siguiente versión (simplificada, sin los métodos) del diagrama de clases original:



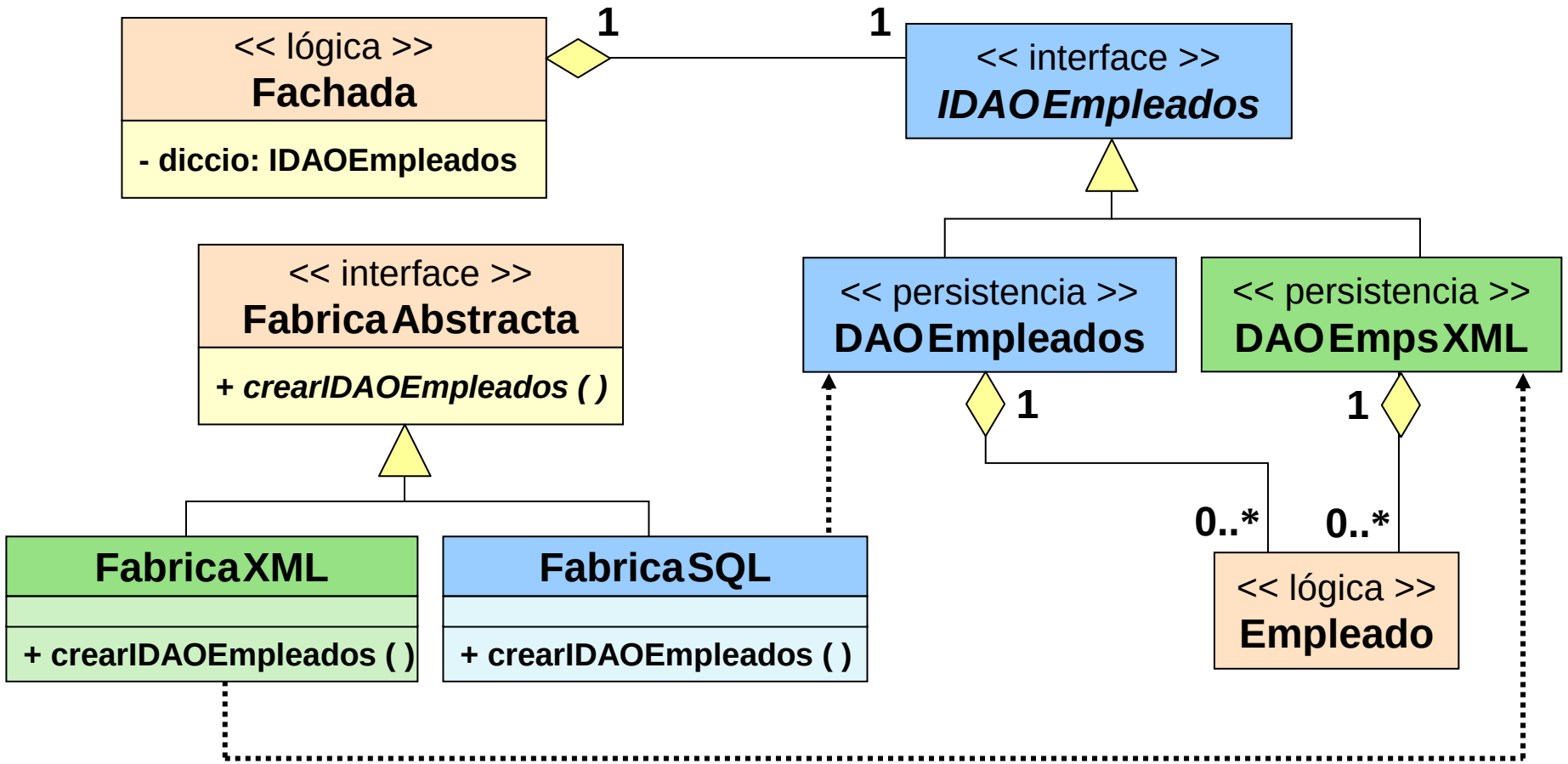
Vamos a incorporarle la ***fábrica*** y tomaremos el actual mecanismo de persistencia como la primera ***familia***.

Migración a un nuevo mecanismo de persistencia (cont.):



Ahora incorporaremos al diagrama la fábrica y el DAO concretos correspondientes al 2° mecanismo de persistencia (la 2° ***familia***).

Migración a un nuevo mecanismo de persistencia (cont.):



UDE
UNIVERSIDAD DE LA EMPRESA

Migración a un nuevo mecanismo de persistencia (cont.):

Observaciones:

- La fachada sigue invocando exactamente los ***mismos*** métodos del DAO de empleados que invocaba antes de la migración.
- Dichos métodos están ahora definidos en la interface ***IDAOEmpleados***. En la clase ***DAOEmpleados*** mantienen la misma implementación que tenían antes de la migración.
- La clase ***DAOEmpsXML*** implementa los ***mismos*** métodos, sólo que persiste a los empleados en ***XML*** en vez de hacerlo en ***SQL***
- La instanciación del DAO ahora corre por cuenta de la ***fábrica***. Para cambiar de mecanismo de persistencia, alcanza con instanciar su correspondiente fábrica concreta. Será ella quien a su vez instancie el DAO correspondiente.

UNIVERSIDAD DE LA EMPRESA

Migración a un nuevo mecanismo de persistencia (cont.):

Presentamos ahora el codigo fuente correspondiente a:

- La fábrica ***abstracta*** y las dos fábricas **concretas**.
- La interface ***IDAOEmpleados*** y sus dos implementaciones concretas (**DAOEmpleados** y **DAOEmpsXML**).
- El constructor de la clase **Fachada**.

Definición de la interface correspondiente a la fábrica abstracta:

```
public interface FabricaAbstracta
{
   public IDAOEmpleados crearIDAOEmpleados();
}
```

UNIVERSIDAD DE LA EMPRESA

Migración a un nuevo mecanismo de persistencia (cont.):

Implementación de las fábricas concretas:

```
public class FabricaSQL implements
                                FabricaAbstracta {
   public IDAOEmpleados crearIDAOEmpleados() {
      return new DAOEmpleados();
   }
}
public class FabricaXML implements
                              FabricaAbstracta {
public IDAOEmpleados crearIDAOEmpleados() {
      return new DAOEmpsXML();
}
}
```

Migración a un nuevo mecanismo de persistencia (cont.):

Definición de la interface correspondiente al DAO. Los cabezales siguen correspondiendo a los métodos de la clase DAO de empleados del ejemplo original:

```
public interface IDAOEmpleados {
    public boolean member(int ced);
    public void insert(Empleado emp);
    public Empleado find(int ced);
    public List<Empleado> listarEmpleados();
}
```

UNIVERSIDAD DE LA EMPRESA

Migración a un nuevo mecanismo de persistencia (cont.):

Las clases **DAOEmpleados** y **DAOEmpsXML** implementan ambas la interface ***IDAOEmpleados***. La primera mantiene ***exactamente*** la implementación del ejemplo original. La segunda implementa los mismos métodos de modo que accedan a la persistencia en XML.

```
public class DAOEmpleados implements
                                    IDAOEmpleados {
   /* accede a la persistencia en SQL */
}

public class DAOEmpsXML implements
                                    IDAOEmpleados {
   /* accede a la persistencia en XML */
}
```

Migración a un nuevo mecanismo de persistencia (cont.):

La clase Fachada ya ***no*** instancia al DAO directamente, sino que delega dicha tarea a la fábrica abstracta. La fábrica concreta es instanciada ***dinámicamente***.

```
public class Fachada {

   private IDAOEmpleados diccio;

   public Fachada() {
      String nomFab = /* cargo el nombre de la
       fábrica de un archivo de configuración */
      FabricaAbstracta fabrica =
        Class.forName(nomFab).newInstance();
      diccio = fabrica.crearIDAOEmpleados();
   }
/* resto de la fachada igual que antes */
```

UNIVERSIDAD DE LA EMPRESA

Migración a un nuevo mecanismo de persistencia (cont.):

Conclusiones:

- El cambio de mecanismo de persistencia permanece totalmente ***oculto*** para la fachada. Ella invoca los métodos definidos en la ***interface*** del DAO pero desconoce cuál es la clase concreta que se está ejecutando.
- Todos los métodos de la fachada permanecen ***incambiados***. La lógica permanece totalmente ajena al cambio de persistencia.
- ***No*** se modifica la implementación de los métodos del DAO original. En su lugar, se incorpora una ***nueva*** clase DAO que accede al nuevo mecanismo de persistencia.
- Se puede cambiar muy fácilmente de un mecanismo de persistencia a otro simplemente modificando el nombre de la fábrica concreta en un ***archivo de configuración***.