Tema 4. Diseño de la capa de persistencia

BDSW – Bases de datos en sistemas web

Wladimiro Díaz

Departament d'Informàtica
Escola Técnica Superior d'Enginyeria
Universitat de València

Máster ISAW 2014

Indice

Indice

Indice

Indice

Indice

Indice

Objetivo

- En este capítulo veremos cómo escribir una buena capa de persistencia que nos permita separar de forma limpia el dominio de los objetos de su almacenamiento.
- Es importante separarla de la capa de negocio porque:
  - Permite reusar el código del objeto en otras aplicaciones.
  - La implementación es mucho más fácil de leer y por tanto de mantener.
  - Una implementación más fácil implica menos código y menos *bugs*.
  - Permite cambiar el repositorio de almacenamiento con mínimos cambios en la capa de persistencia.

Indice

¿Qué es la *Data Access Object*?

Cosas que ya sabemos:

- La mayoría de las aplicaciones tienen que manejar, en algún momento, datos persistentes.
- Los datos se pueden hacer persistentes por varias vías: serialización, mediante una base de datos relacional, etc...
- En cualquier caso la aplicación debe interactuar con el sistema de persistencia.

Definición

DAO es un **patrón de diseño** utilizado para crear esta capa de persistencia.

¿Qué es la *Data Access Object*?

Cosas que ya sabemos:

- La mayoría de las aplicaciones tienen que manejar, en algún momento, datos persistentes.
- Los datos se pueden hacer persistentes por varias vías: serialización, mediante una base de datos relacional, etc...
- En cualquier caso la aplicación debe interactuar con el sistema de persistencia.

Definición

DAO es un **patrón de diseño** utilizado para crear esta capa de persistencia.

¿Qué es un patrón de diseño?

Según Christopher Alexander,

"cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema de modo que se pueda aplicar esa solución un millón de veces, sin hacer lo mismo dos veces"

- Alexander escribió esto en 1977 refiriéndose a patrones en ciudades y edificios...
- Sin embargo, lo que dice también es válido para patrones de diseño orientado a objetos.
  - Nuestras soluciones se expresan en términos de objetos e interfaces, en vez de paredes y puertas.
- En la esencia de ambos tipos de patrones se encuentra una solución a un problema dentro de un contexto.

Elementos de un patrón

- El **nombre del patrón** permite describir un problema de diseño junto con sus soluciones y consecuencias.
- El **problema** describe cuándo aplicar el patrón. Explica el problema y su contexto.
- La **solución** formada por un conjunto de elementos dispuestos específicamente.
- Las **consecuencias** son las ventajas e inconvenientes de aplicar el patrón.

¿Qué patrones nos interesan?

- **[DAO]**. *Data Access Object*.
- **[VO]**. *Value Object*. Este patrón se conoce también como:
  - **[DTO]**. *Data Transfer Object*.
  - **[TO]**. *Transfer Object*.

  Son utilizados por los DAOs para transportar los datos desde la base de datos a la capa de negocio.
- **[Singleton]**. Nos permite tener una única instancia asociada a cada clase.
- **[Factory]**. Permite delegar la creación de las instancias del DAO.
- También hablaremos del **[Business Object]**, que implementa la lógica de negocio de nuestro sistema.

¿Por qué el DAO?

Pongámonos en el siguiente supuesto:

- Hemos desarrollado una aplicación de gestión para un cliente.
  - La desarrollamos sobre el sistema de información del cliente con una base de datos Oracle.
- La aplicación es un éxito y pronto aparece un cliente que quiere adaptar la aplicación a su empresa.
  - Su sistema de información está basado en MySQL.
- Sin embargo...
  - No hemos dividido la capa de lógica de negocio de la capa de persistencia.
  - La interacción con la base de datos se hace directamente desde la capa de negocio
  - Nuestra aplicación está formada por muchas clases y la mayor parte de ellas se conectan directamente a la base de datos para leer y escribir datos.

¿Por qué el DAO?

Pongámonos en el siguiente supuesto:

- Hemos desarrollado una aplicación de gestión para un cliente.
  - La desarrollamos sobre el sistema de información del cliente con una base de datos Oracle.
- La aplicación es un éxito y pronto aparece un cliente que quiere adaptar la aplicación a su empresa.
  - Su sistema de información está basado en MySQL.
- Sin embargo...
  - No hemos dividido la capa de lógica de negocio de la capa de persistencia.
  - La interacción con la base de datos se hace directamente desde la capa de negocio
  - Nuestra aplicación está formada por muchas clases y la mayor parte de ellas se conectan directamente a la base de datos para leer y escribir datos.

¿Por qué el DAO?

Pongámonos en el siguiente supuesto:

- Hemos desarrollado una aplicación de gestión para un cliente.
  - La desarrollamos sobre el sistema de información del cliente con una base de datos Oracle.
- La aplicación es un éxito y pronto aparece un cliente que quiere adaptar la aplicación a su empresa.
  - Su sistema de información está basado en MySQL.
- Sin embargo...
  - No hemos dividido la capa de lógica de negocio de la capa de persistencia.
  - La interacción con la base de datos se hace directamente desde la capa de negocio
  - Nuestra aplicación está formada por muchas clases y la mayor parte de ellas se conectan directamente a la base de datos para leer y escribir datos.

¿Por qué el DAO?

Pongámonos en el siguiente supuesto:

- Hemos desarrollado una aplicación de gestión para un cliente.
  - La desarrollamos sobre el sistema de información del cliente con una base de datos Oracle.
- La aplicación es un éxito y pronto aparece un cliente que quiere adaptar la aplicación a su empresa.
  - Su sistema de información está basado en MySQL.
- Sin embargo...
  - No hemos dividido la capa de lógica de negocio de la capa de persistencia.
  - La interacción con la base de datos se hace directamente desde la capa de negocio
  - Nuestra aplicación está formada por muchas clases y la mayor parte de ellas se conectan directamente a la base de datos para leer y escribir datos.

Justo en ese instante descubrimos que:

- Vamos a tener que modificar todas las clases de nuestra aplicación.
- Será necesario reescribir todo el código de conexión/desconexión a la base de datos en cada una de las clases.
- Habrá que revisar el código buscando, revisando y adaptando todas las sentencias SQL a la nueva base de datos.

Conclusión

Si hubiéramos separado la lógica de negocio de la capa de persistencia sólo tendríamos que reescribir esta última para empezar a utilizar el nuevo motor de base de datos.

Justo en ese instante descubrimos que:

- Vamos a tener que modificar todas las clases de nuestra aplicación.
- Será necesario reescribir todo el código de conexión/desconexión a la base de datos en cada una de las clases.
- Habrá que revisar el código buscando, revisando y adaptando todas las sentencias SQL a la nueva base de datos.

Conclusión

Si hubiéramos separado la lógica de negocio de la capa de persistencia sólo tendríamos que reescribir esta última para empezar a utilizar el nuevo motor de base de datos.

¿Cómo funciona la capa DAO?

- El patrón **DAO** encapsula en acceso a la base de datos.
- Cuando la capa de lógica de negocio requiere un dato lo obtiene a través de la **API** que le ofrece el DAO.
- La API consiste generalmente en métodos **CRUD**: Create, Read, Update y Delete.
- Lo que hagan estos métodos es problema del DAO y depende de cómo el DAO implementa el método.
  - Por ejemplo, los datos se pueden estar almacenando en una base de datos, en ficheros planos mediante serialización, etc...
- La capa de negocio ni lo sabe ni tiene por qué saberlo.
  - Lo único importante es que cuando se invoca un método, éste realiza la acción para la que fue programado.

- En una aplicación hay tantos DAOs como datos persistentes.
- Para cada tabla en una base de datos relacional es necesario crear un DAO.

En nuestra aplicación empresa:

- `DepartamentoDAO`.
- `EmpleadoDAO`.
- `ProyectoDAO`.

- Cada una de estas clases interactúa con la base de datos.
- Los métodos de estas clases dependen de lo que hace la aplicación.

En general:

Al menos se implementan las cuatro operaciones básicas de una base de datos: CRUD.

- En una aplicación hay tantos DAOs como datos persistentes.
- Para cada tabla en una base de datos relacional es necesario crear un DAO.

En nuestra aplicación empresa:

- `DepartamentoDAO`.
- `EmpleadoDAO`.
- `ProyectoDAO`.

- Cada una de estas clases interactúa con la base de datos.
- Los métodos de estas clases dependen de lo que hace la aplicación.

En general:

Al menos se implementan las cuatro operaciones básicas de una base de datos: CRUD.

- En una aplicación hay tantos DAOs como datos persistentes.
- Para cada tabla en una base de datos relacional es necesario crear un DAO.

En nuestra aplicación empresa:

- `DepartamentoDAO`.
- `EmpleadoDAO`.
- `ProyectoDAO`.

- Cada una de estas clases interactúa con la base de datos.
- Los métodos de estas clases dependen de lo que hace la aplicación.

En general:

Al menos se implementan las cuatro operaciones básicas de una base de datos: CRUD.

- En una aplicación hay tantos DAOs como datos persistentes.
- Para cada tabla en una base de datos relacional es necesario crear un DAO.

En nuestra aplicación empresa:

- `DepartamentoDAO`.
- `EmpleadoDAO`.
- `ProyectoDAO`.

- Cada una de estas clases interactúa con la base de datos.
- Los métodos de estas clases dependen de lo que hace la aplicación.

En general:

Al menos se implementan las cuatro operaciones básicas de una base de datos: CRUD.

Los *Value Objects*

- Las clases DAO son transportadores de datos desde la base de datos a la capa de negocio y viceversa.
- Esos datos son los *Value Objects* (**VO**).
- Las clases VO contienen los atributos del modelo con sus correspondientes *getters* y *setters*.

En la aplicación empresa:

La clase VO `Empleado` tiene los mismos atributos que la tabla `Empleados` de la base de datos. Cada atributo cuenta con su correspondiente *getter* y *setter* denominados de la forma estándar, por ejemplo `getNombre()`.

Los *Value Objects*

- Las clases DAO son transportadores de datos desde la base de datos a la capa de negocio y viceversa.
- Esos datos son los *Value Objects* (**VO**).
- Las clases VO contienen los atributos del modelo con sus correspondientes *getters* y *setters*.

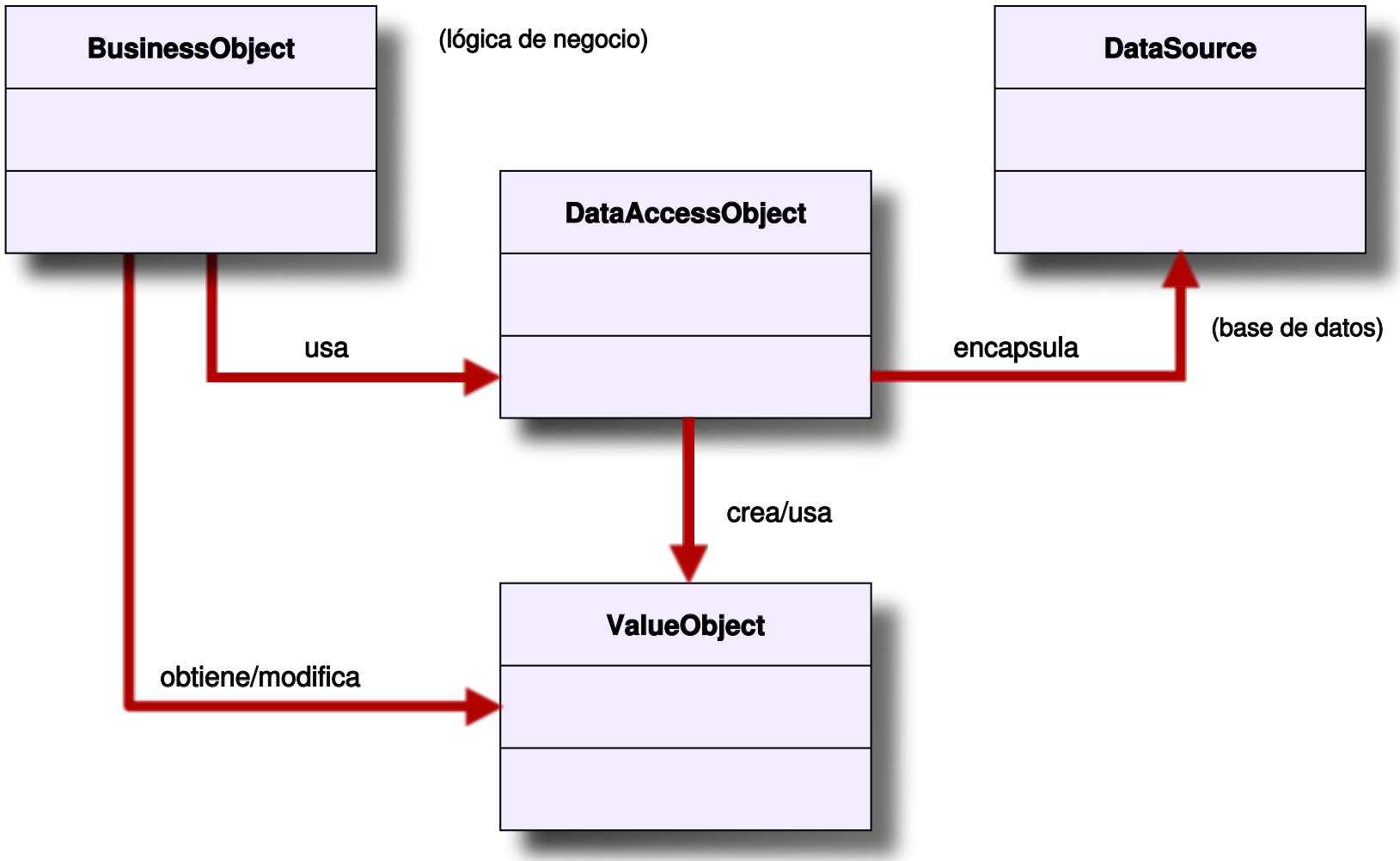
En la aplicación empresa:

La clase VO `Empleado` tiene los mismos atributos que la tabla `Empleados` de la base de datos. Cada atributo cuenta con su correspondiente *getter* y *setter* denominados de la forma estándar, por ejemplo `getNombre()`.

¿Cómo funciona?

- Supongamos que la capa de negocio desea guardar un dato en la base de datos.
  - Creará un objeto del tipo `Empleado`.
  - A través de los *accessors* va a modificar sus atributos.
  - Invocará al método `create` del DAO, quien leeará los atributos y los guardará en la base de datos.
- Para eliminar y actualizar datos, bastaría con pasar el `Id` del objeto.
- Para buscar datos, es posible pasar ciertos criterios de búsqueda a través de los atributos...

Esquema del funcionamiento de un DAO



La clase `EmpleadoDAO`

- Supongamos que debemos almacenar empleados en la base de datos.
- El **VO** de la clase `Empleado` ya lo vimos en el tema 1.
- Ahora vamos a crear una versión muy simple del DTO de la clase.

El código de la clase

```
1   class EmpleadoDAO {
2     public void create(Empleado e) {
3       // Codigo del metodo create
4     }
5     public Empleado read(Empleado e) {
6       // Codigo del metodo read
7     }
8     public void update(Empleado e) {
9       // Codigo del metodo update
10    }
11    public void delete(Empleado e) {
12      // Codigo del metodo delete
13    }
14  }
```

Más sobre el DAO

- El código de cada uno de los métodos depende de la base de datos que se esté utilizando.
- Si estamos utilizando una base de datos relacional, el método `create` utilizará una sentencia SQL para insertar una nueva tupla en la tabla `Empleados`.
- Este método puede generar una `Exception` si algo falla en el proceso.
- Por ejemplo:
  - El `idEmpleado` no es único.
  - No se ha asignado nombre al empleado (es `NULL`).
  - No se ha asignado apellidos al empleado (`NULL`).

Una nota importante:

¡En el código anterior no se ha creado ninguna conexión a la base de datos!

Más sobre el DAO

- El código de cada uno de los métodos depende de la base de datos que se esté utilizando.
- Si estamos utilizando una base de datos relacional, el método `create` utilizará una sentencia SQL para insertar una nueva tupla en la tabla `Empleados`.
- Este método puede generar una `Exception` si algo falla en el proceso.
- Por ejemplo:
  - El `idEmpleado` no es único.
  - No se ha asignado nombre al empleado (es `NULL`).
  - No se ha asignado apellidos al empleado (`NULL`).

Una nota importante:

¡En el código anterior no se ha creado ninguna conexión a la base de datos!

Más sobre el DAO

- El código de cada uno de los métodos depende de la base de datos que se esté utilizando.
- Si estamos utilizando una base de datos relacional, el método `create` utilizará una sentencia SQL para insertar una nueva tupla en la tabla `Empleados`.
- Este método puede generar una `Exception` si algo falla en el proceso.
- Por ejemplo:
  - El `idEmpleado` no es único.
  - No se ha asignado nombre al empleado (es `NULL`).
  - No se ha asignado apellidos al empleado (`NULL`).

Una nota importante:

¡En el código anterior no se ha creado ninguna conexión a la base de datos!

Utilizando la capa de persistencia

- Ahora ya podemos utilizar la capa de persistencia desde la capa de negocio:

```
1   public static void main(String[] args) {
2     // Creamos el DAO
3     EmpleadoDAO edao = new EmpleadoDAO();
4     // Creamos el VO
5     Empleado empleado = new Empleado();
6     // Asignamos los datos
7     empleado.setIdEmpleado("000001");
8     empleado.setNombre("MIGUEL ANGEL");
9     empleado.setApellidos("LOPEZ CAMPOS");
10      ...
11    // Creamos el objeto
12    edao.create(empleado);
13  }
```

Indice

¿Por qué el patrón singletón?

- De lo dicho hasta ahora:
  - Los objetos VO sirven para transportar los objetos.
  - Los objetos DAO constituyen la capa de persistencia.
- Si nos fijamos en el código anterior, descubrimos que sólo necesitamos una instancia del DAO por VO.
  - Esto además redunda en la eficiencia de nuestro programa, que no necesita abrir una conexión a la base de datos para cada operación del DAO.
- Podemos adoptar dos soluciones:
  - Podemos crear los métodos del DAO estáticos, de manera que no sea necesario instanciar el DAO [No es una buena opción]
  - Implementamos el patrón Singletón, que nos permite contar con una sola instancia del DAO.

El código del singletón

```
1  class EmpleadoDAO {
2    private static EmpleadoDAO instancia = new EmpleadoDAO();
3
4    private EmpleadoDAO() {}
5    public static EmpleadoDAO getInstance() {
6      return instance;
7    }
8     ...
9  }
```

La lógica de negocio

- La lógica de negocio queda ahora:

```
1   public static void main(String[] args) {
2       // Obtenemos el DAO (que es unico)
3       EmpleadoDAO edao = EmpleadoDAO.getInstance();
4       // Creamos el VO
5       Empleado empleado = new Empleado();
6       // Asignamos los datos
7       empleado.setIdEmpleado("000001");
8       empleado.setNombre("MIGUEL ANGEL");
9       empleado.setApellidos("LOPEZ CAMPOS");
10          ...
11      // Creamos el objeto
12      edao.create(empleado);
13  }
```

Sin embargo...

¡Sorpresa!

No utilizaremos en lo que queda de ejercicio el patrón singletón

Pero...

Utilizaremos una técnica similar para disponer de una única conexión por DAO.

Sin embargo...

¡Sorpresa!

No utilizaremos en lo que queda de ejercicio el patrón singletón

Pero...

Utilizaremos una técnica similar para disponer de una única conexión por DAO.

Indice

¿Por qué el patrón Factory?

- Ahora ya disponemos de una capa de persistencia:
  - Independiza la lógica de negocio de la capa de acceso a los datos.
  - Funciona estupendamente con nuestro nuevo motor de base de datos MySQL.
- Ahora deberíamos reescribir el código del DAO para nuestro primer cliente, que recordemos que trabaja con Oracle.
  - Es necesario este proceso de *refractoring* ya que no podemos mantener tantas versiones del producto.
- Es posible que en el futuro debamos utilizar otros sistemas de persistencia e incluso mecanismos de persistencia híbridos.
- Sin embargo estamos contentos porque de momento sólo tenemos que reescribir las clases DAO para Oracle...

¿Seguro?

¡Es probable que se nos esté pasando por alto un pequeño detalle!

¿Por qué el patrón Factory?

- Ahora ya disponemos de una capa de persistencia:
  - Independiza la lógica de negocio de la capa de acceso a los datos.
  - Funciona estupendamente con nuestro nuevo motor de base de datos MySQL.
- Ahora deberíamos reescribir el código del DAO para nuestro primer cliente, que recordemos que trabaja con Oracle.
  - Es necesario este proceso de *refractoring* ya que no podemos mantener tantas versiones del producto.
- Es posible que en el futuro debamos utilizar otros sistemas de persistencia e incluso mecanismos de persistencia híbridos.
- Sin embargo estamos contentos porque de momento sólo tenemos que reescribir las clases DAO para Oracle...

¿Seguro?

¡Es probable que se nos esté pasando por alto un pequeño detalle!

El problema

- Toda el código de la lógica de negocio está lleno de inicializaciones del tipo:

```
1    EmpleadoDAO dao = new EmpleadoDAO();
```

- Ahora deberíamos utilizar algo así como:

```
1    OracleEmpleadoDAO dao = new OracleEmpleadoDAO();
```

- Y en un futuro tendremos cosas todavía más extrañas:

```
1    XmlEmpleadoDAO dao = new XmlEmpleadoDAO();
```

¿Cómo podemos abordar el problema?

- Ahora sabemos que instanciaciones como las anteriores no son una buena idea en nuestra lógica de negocio.
- Una solución consiste en delegar el trabajo de instanciación a una fábrica de DAOs.
- Nuesto código:

```
1    EmpleadoDAO = new MySQLEmpleadoDAO();
```

- Quedará como:

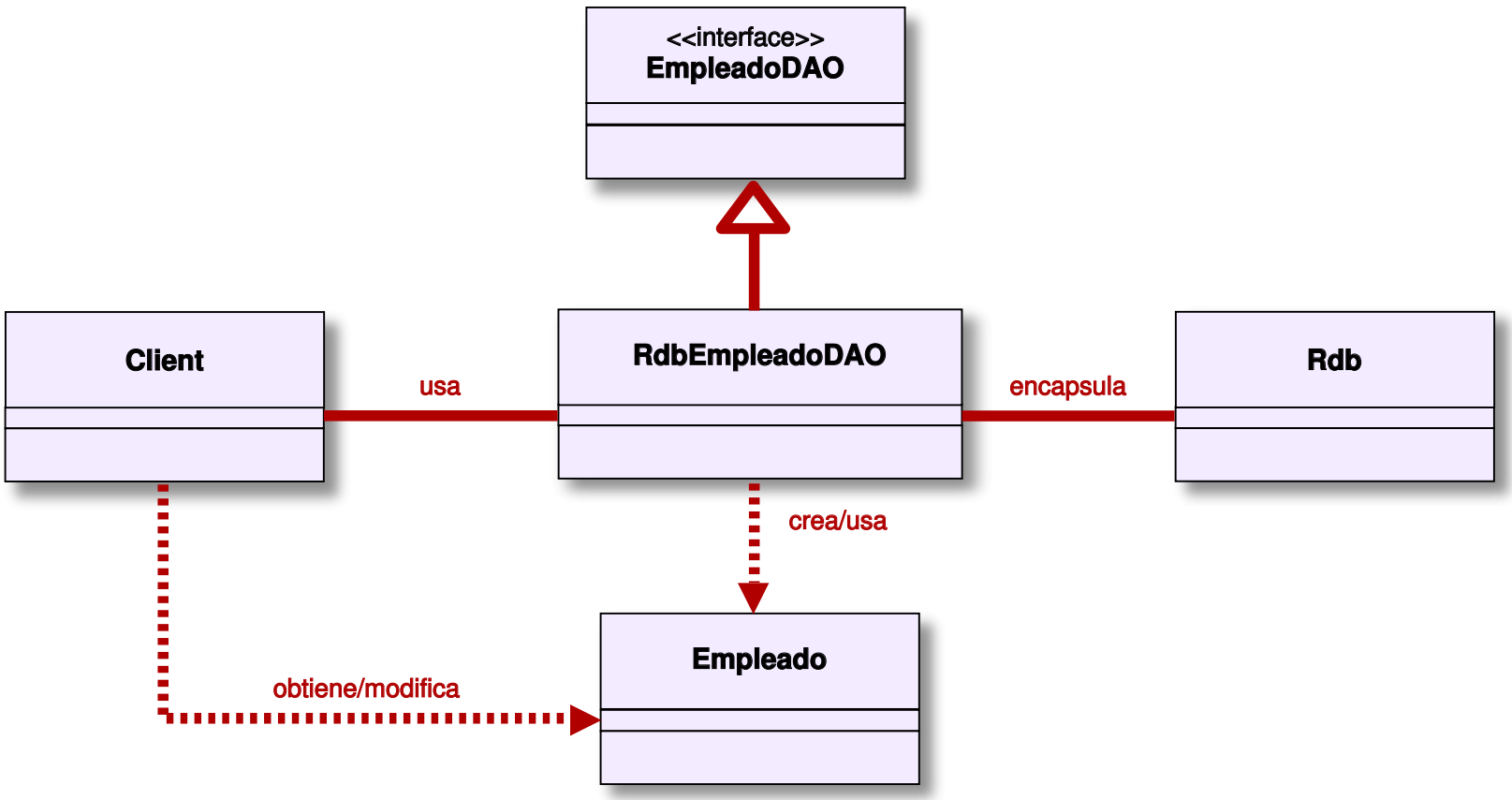
```
1    EmpleadoDAO = DAOFactory.getEmpleadoDAO();
```

- El método `getEmpleadoDAO()` devolverá un `EmpleadoDAO` del tipo adecuado para nuestro sistema.
- Utilizaremos esta misma fábrica de DAOs para obtener instancias adecuadas de los otros DAOS:

```
1    DepartamentoDAO = DAOFactory.getDepartamentoDAO();
2    ProyectoDAO = DAOFactory.getProyectoDAO();
```

¿Cómo abordar el problema?

- Primero implementaremos las clases DAO a través de una `interface`:



¿Por qué la interface `EmpleadoDAO`?

- La interface define los métodos DAO para los objetos `Empleado`.

```
1  // Interface de todos los EmpleadoDAO
2  public interface EmpleadoDAO {
3      public void insert(Empleado e);
4      public Empleado read(Empleado e);
5      public void update(Empleado e);
6      public void delete(Empleado e);
7  }
```

- Dispondremos de interfaces similares para `DepartamentoDAO` y `ProyectoDAO`.

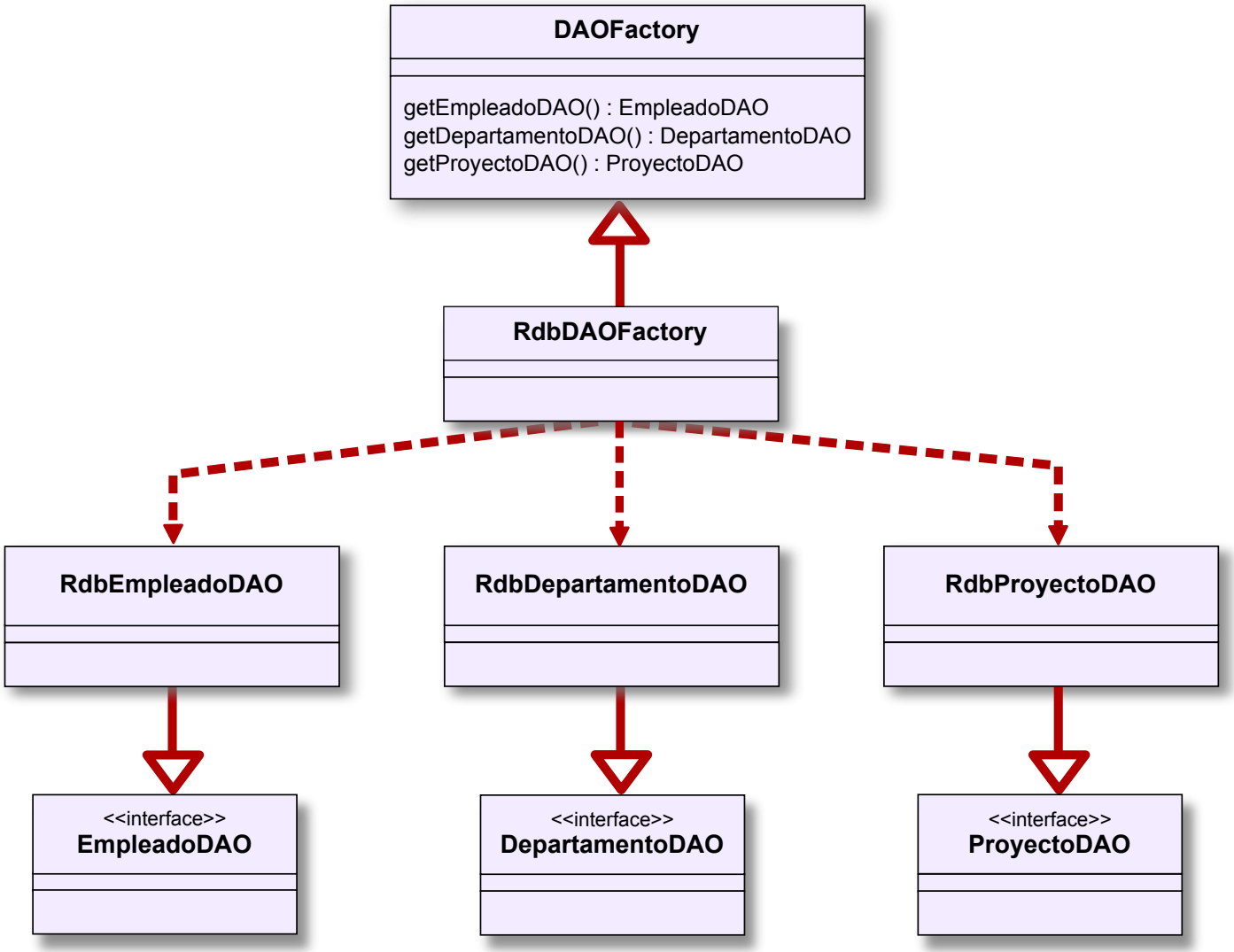
El nuevo MySQLEmpleadoDAO

```
1   class MySQLEmpleadoDAO implements EmpleadoDAO {
2
3     public MySQLEmpleadoDAO() {
4       // Inicializacion
5     }
6     public void create(Empleado e) {
7       // Codigo del metodo create
8     }
9     public Empleado read(Empleado e) {
10      // Codigo del metodo read
11    }
12    public void update(Empleado e) {
13      // Codigo del metodo update
14    }
15    public void delete(Empleado e) {
16      // Codigo del metodo delete
17    }
18  }
```

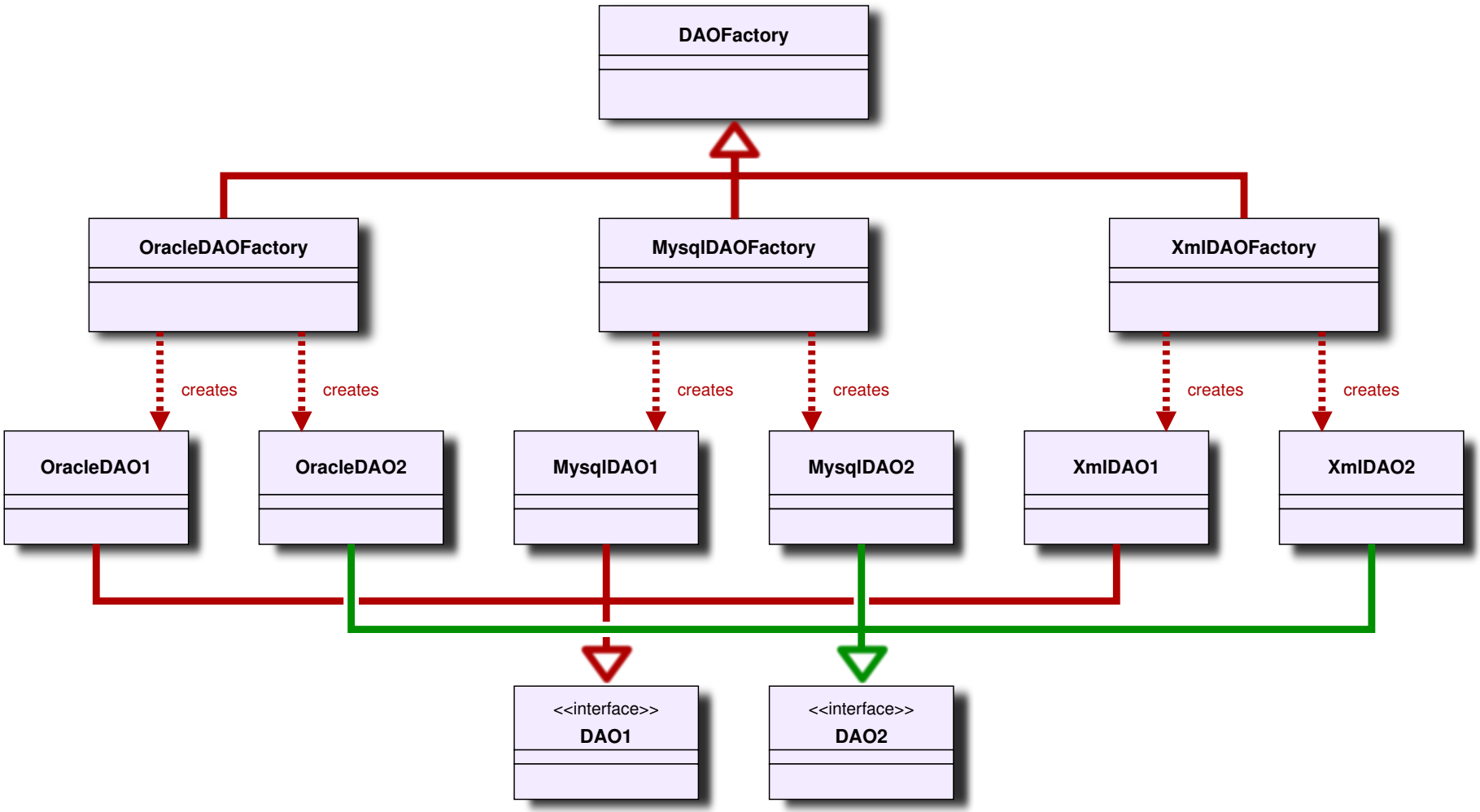
Estrategias para implementar la factoria

- **Factory Method**. Se puede utilizar cuando el almacenamiento subyacente no está sujeto a cambios de implementación.
  - Se trata de una factoría concreta.
  - Producirá el número de DAOs que necesite la aplicación.
- **Abstract Factory**. Se debe utilizar cuando el almacenamiento subyacente está sujeto a cambios de implementación.
  - Este patrón a su vez puede construir y utilizar la implementación Factory Method.
  - Esta estrategia proporciona un objeto factoría abstracta que puede construir varios tipos de factorías concretas de DAOs.
  - Cada factoría soporta un tipo diferente de implementación del almacenamiento persistente.
  - Una vez que obtenemos la factoría concreta, la utilizamos para producir los DAOs de la implementación.

La factoría concreta de DAOs



La factoría abstracta de DAOs



El código de la `DAOFactory`

```
1   public abstract class DAOFactory {
2     // Lista de DAOs soportados
3     public static final int ORACLE = 1;
4     public static final int MYSQL = 2;
5     public static final int XML = 3;
6
7     // Metodos para cada DAO creado
8     public abstract EmpleadoDAO getEmpleadoDAO();
9     public abstract DepartamentoDAO getDepartamentoDAO();
10    public abstract ProyectoDAO getProyectoDAO();
11    ...
```

El código de `DAOFactory`

```
1       ...
2       public static DAOFactory getDAOFactory(int which) {
3         switch (witch) {
4           case ORACLE:
5             return new OracleDAOFactory();
6           case MYSQL:
7             return new MySQLDAOFactory();
8           case XML:
9             return new XMLDAOFactory();
10          default:
11            return null;
12        }
13      }
14    }
```

El código de `MySQLDAOFactory`

```
1   public class MySQLDAOFactory extends DAOFactory {
2     public static final String DRIVER="com.mysql.jdbc.Driver";
3     public static final String DBURL="jdbc:mysql://localhost/empresa";
4     public static final String USERNAME="usuario";
5     public static final String PASSWORD="pusuario";
6     // Metodo para crear conexiones
7     public static Connection createConnection() {
8       // Crea una conexion utilizando DRIVER, DBURL, USERNAME y
9       // PASSWORD
10    }
11    public EmpleadoDAO getEmpleadoDAO() {
12      return new MySQLEmpleadoDAO();
13    }
14    public DepartamentoDAO getDepartamentoDAO() {
15      return new MySQLEmpleadoDAO();
16    }
17    public ProyectoDAO getProyectoDAO() {
18      return new MySQLProyectoDAO();
19    }
20  }
```

La lógica de negocio

```
1   ...
2   // Crea la Factory
3   DAOFactory myFactory = DAOFactory.getDAOFactory(DAOFactory.MYSQL);
4   // Crea el DAO
5   EmpleadoDAO edao = myFactory.getEmpleadoDAO();
6   // Crea un nuevo empleado
7   Empleado e = new Empleado();
8   // Asignamos los datos
9   empleado.setIdEmpleado("000001");
10  empleado.setNombre("MIGUEL ANGEL");
11  empleado.setApellidos("LOPEZ CAMPOS");
12  ...
13  // Creamos el objeto
14  edao.create(empleado);
```

Indice

Ventajas del patrón DAO

- **Transpariencia**. Los objetos de negocio puede utilizar la fuente de datos sin conocer los detalles específicos de su implementación.
  - El acceso es transparente porque los detalles de la implementación se ocultan dentro del DAO.
- **Migración**. La migración implica cambios sólo en la capa DAO. Además:
  - La estrategia de factorías proporciona una implementación de factorías concretas por cada implementación del almacenamiento subyacente.
  - La migración a un almacenamiento diferente significa proporcionar a la aplicación una nueva implementación de la factoría.

Ventajas del patrón DAO

- **Transpariencia**. Los objetos de negocio puede utilizar la fuente de datos sin conocer los detalles específicos de su implementación.
  - El acceso es transparente porque los detalles de la implementación se ocultan dentro del DAO.
- **Migración**. La migración implica cambios sólo en la capa DAO. Además:
  - La estrategia de factorías proporciona una implementación de factorías concretas por cada implementación del almacenamiento subyacente.
  - La migración a un almacenamiento diferente significa proporcionar a la aplicación una nueva implementación de la factoría.

Ventajas del patrón DAO

- **Transpariencia**. Los objetos de negocio puede utilizar la fuente de datos sin conocer los detalles específicos de su implementación.
  - El acceso es transparente porque los detalles de la implementación se ocultan dentro del DAO.
- **Migración**. La migración implica cambios sólo en la capa DAO. Además:
  - La estrategia de factorías proporciona una implementación de factorías concretas por cada implementación del almacenamiento subyacente.
  - La migración a un almacenamiento diferente significa proporcionar a la aplicación una nueva implementación de la factoría.

Ventajas del patrón DAO

- **Transpariencia**. Los objetos de negocio puede utilizar la fuente de datos sin conocer los detalles específicos de su implementación.
  - El acceso es transparente porque los detalles de la implementación se ocultan dentro del DAO.
- **Migración**. La migración implica cambios sólo en la capa DAO. Además:
  - La estrategia de factorías proporciona una implementación de factorías concretas por cada implementación del almacenamiento subyacente.
  - La migración a un almacenamiento diferente significa proporcionar a la aplicación una nueva implementación de la factoría.

- **Reducción de la complejidad del código de los Objetos de Negocio**.
  - Se simplifica el código de los objetos de negocio y del resto de clientes que utilizan los DAOs.
  - Todo el código relacionado con la implementación (por ejemplo las sentencias SQL) están dentro del DAO.
  - Mejora la lectura del código y la productividad del desarrollo.
- **Centraliza Todos los Accesos a Datos**.
  - El DAO es una capa que aisla el resto de la aplicación de la implementación.
  - La La aplicación sea más sencilla de mantener y de manejar.

- **Reducción de la complejidad del código de los Objetos de Negocio**.
  - Se simplifica el código de los objetos de negocio y del resto de clientes que utilizan los DAOs.
  - Todo el código relacionado con la implementación (por ejemplo las sentencias SQL) están dentro del DAO.
  - Mejora la lectura del código y la productividad del desarrollo.
- **Centraliza Todos los Accesos a Datos**.
  - El DAO es una capa que aisla el resto de la aplicación de la implementación.
  - La La aplicación sea más sencilla de mantener y de manejar.

- **Reducción de la complejidad del código de los Objetos de Negocio**.
  - Se simplifica el código de los objetos de negocio y del resto de clientes que utilizan los DAOs.
  - Todo el código relacionado con la implementación (por ejemplo las sentencias SQL) están dentro del DAO.
  - Mejora la lectura del código y la productividad del desarrollo.
- **Centraliza Todos los Accesos a Datos**.
  - El DAO es una capa que aisla el resto de la aplicación de la implementación.
  - La La aplicación sea más sencilla de mantener y de manejar.

- **Reducción de la complejidad del código de los Objetos de Negocio**.
  - Se simplifica el código de los objetos de negocio y del resto de clientes que utilizan los DAOs.
  - Todo el código relacionado con la implementación (por ejemplo las sentencias SQL) están dentro del DAO.
  - Mejora la lectura del código y la productividad del desarrollo.
- **Centraliza Todos los Accesos a Datos**.
  - El DAO es una capa que aisla el resto de la aplicación de la implementación.
  - La La aplicación sea más sencilla de mantener y de manejar.

Inconvenientes

- **Añade una capa extra**. Los DAOs crean un capa de objetos adicional entre el cliente y la fuente de datos que:
  - Es necesario diseñar e implementar.
  - Implica un esfuerzo de desarrollo adicional.
- **Diseño de un árbol de clases**. Implica diseñar e implementar el árbol de factorías concretas y el árbol de productos concretos.
  - ¿Se justifica el esfuerzo adicional que requiere la flexibilidad?
  - Incrementa la complejidad del diseño.
  - Podemos reducir el esfuerzo empezando primero con el patrón *Factory Method* y sólo avanzar hasta el patrón *Abstract Factory* si es necesario.

Inconvenientes

- **Añade una capa extra**. Los DAOs crean un capa de objetos adicional entre el cliente y la fuente de datos que:
  - Es necesario diseñar e implementar.
  - Implica un esfuerzo de desarrollo adicional.
- **Diseño de un árbol de clases**. Implica diseñar e implementar el árbol de factorías concretas y el árbol de productos concretos.
  - ¿Se justifica el esfuerzo adicional que requiere la flexibilidad?
  - Incrementa la complejidad del diseño.
  - Podemos reducir el esfuerzo empezando primero con el patrón *Factory Method* y sólo avanzar hasta el patrón *Abstract Factory* si es necesario.

Inconvenientes

- **Añade una capa extra**. Los DAOs crean un capa de objetos adicional entre el cliente y la fuente de datos que:
  - Es necesario diseñar e implementar.
  - Implica un esfuerzo de desarrollo adicional.
- **Diseño de un árbol de clases**. Implica diseñar e implementar el árbol de factorías concretas y el árbol de productos concretos.
  - ¿Se justifica el esfuerzo adicional que requiere la flexibilidad?
  - Incrementa la complejidad del diseño.
  - Podemos reducir el esfuerzo empezando primero con el patrón *Factory Method* y sólo avanzar hasta el patrón *Abstract Factory* si es necesario.

Inconvenientes

- **Añade una capa extra**. Los DAOs crean un capa de objetos adicional entre el cliente y la fuente de datos que:
  - Es necesario diseñar e implementar.
  - Implica un esfuerzo de desarrollo adicional.
- **Diseño de un árbol de clases**. Implica diseñar e implementar el árbol de factorías concretas y el árbol de productos concretos.
  - ¿Se justifica el esfuerzo adicional que requiere la flexibilidad?
  - Incrementa la complejidad del diseño.
  - Podemos reducir el esfuerzo empezando primero con el patrón *Factory Method* y sólo avanzar hasta el patrón *Abstract Factory* si es necesario.

Relación de patrones

- **Transfer Object**. Un DAO utiliza Transfer Objects para transportar los datos desde y hacia sus clientes.
- **Factory Method**. La Factoría para implementar las factorías concretas y sus productos (DAOs).
- **Abstract Factory**. Para añadir flexibilidad.
- **Broker**. El patrón DAO está relacionado con el patrón Broker, que describe aproximaciones para desacoplar clientes y servidores en sistemas distribuidos.
  - El patrón DAO aplica este patrón específicamente para desacoplar la capa de recursos de los clientes en otra capa, como las capas de negocio o presentación.