DISEÑO DE LA PERSISTENCIA

Tema 6

Ingeniería del Software
ETS Ingeniería Informática
DSIC – UPV
Curso 2024-2025

Objetivos

- Comprender la necesidad de mantener la persistencia en el desarrollo de software
- Conocer los principales patrones de acceso a datos para conseguir una correcta abstracción y separación de capas
- Persistencia en BD relacionales vs. BD objetuales.

Nota: El diseño lógico relacional de la BD a partir de un modelo OO (diagrama de clases) se estudia en la ***asignatura BDA***.

Contenidos

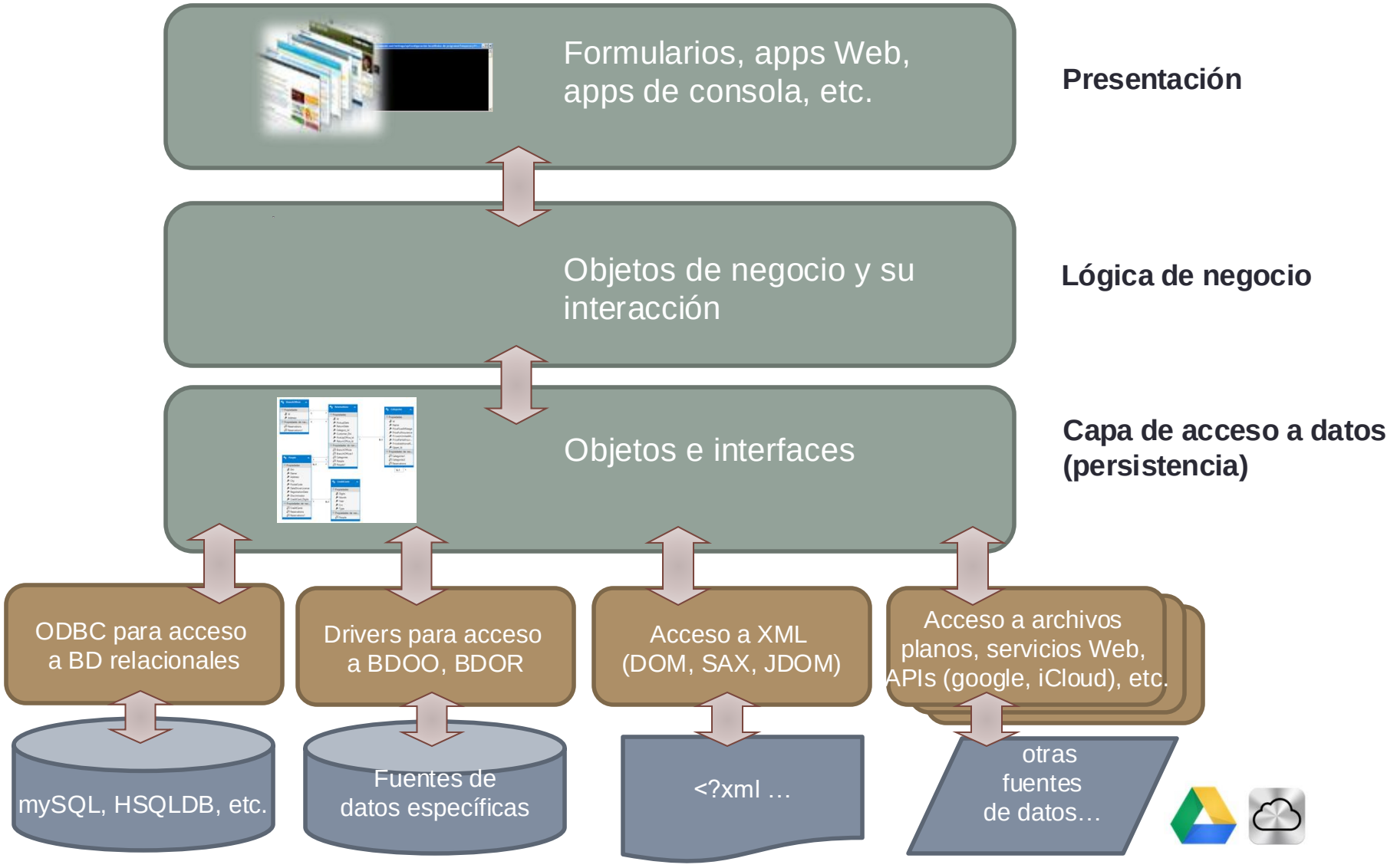
1. Introducción
2. Patrón de acceso a datos DAO
3. Persistencia en BDOR y BDOO
4. Conclusiones

INTRODUCCIÓN

Introducción

- En la mayoría de aplicaciones el almacenamiento **no volátil** de la información es esencial
  - Se puede utilizar un formato específico para cada aplicación (compatibilidad limitada)
  - Se puede usar un formato basado en XML, BD relacionales, OO o mixto (compatibilidad mucho mayor – ej. BD-SQL)
- La utilización de BD se traduce en el uso de bibliotecas para gestionar el acceso a los datos (ODBC, JDBC, ADO, etc.)

Arquitectura multicapa



PATRÓN DE ACCESO A DATOS (DAO)

Estructura

Ventajas e inconvenientes

Implementación

Referencias:

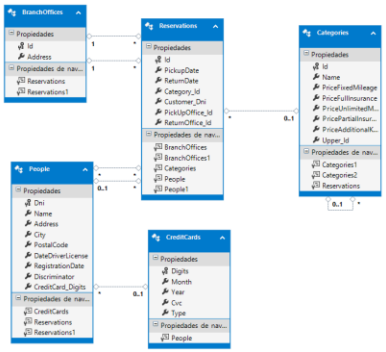
http://bit.ly/2J0JlIQ

http://bit.ly/2CPPib6

El patrón DAO (Data Access Object)

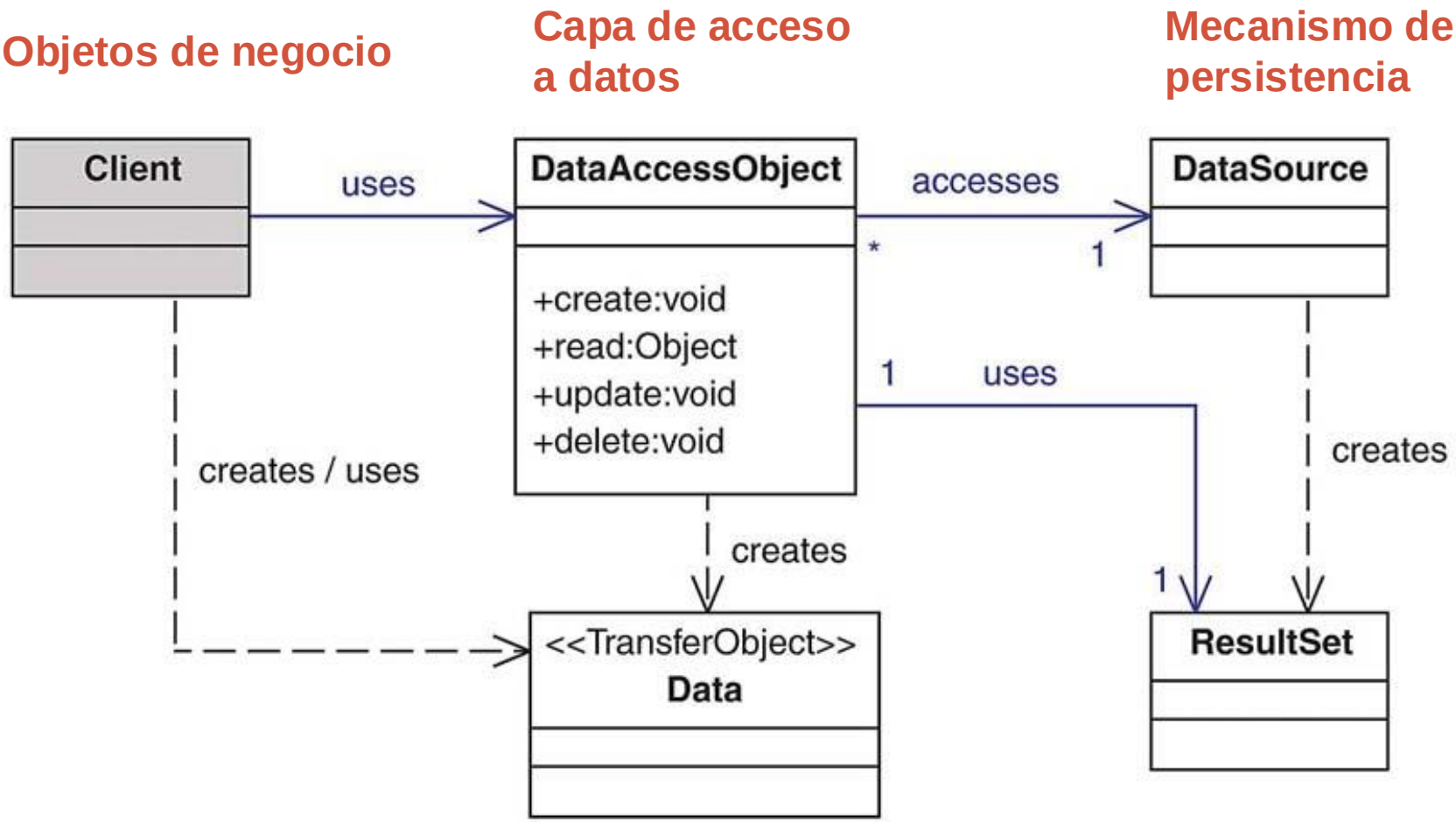
- Ofrece un puente de implementación entre:
  - Los objetos del dominio del problema (lógica de negocio)
  - Datos almacenados de forma persistente (ej. XML, BDR, BDOR, ficheros, etc.)
- Abstrae el mecanismo concreto de persistencia. Ofrece una interfaz de acceso a datos independiente de la implementación con métodos para añadir, actualizar, buscar y borrar registros

lógica de negocio (objetos)



datos almacenados (eg. SQL)

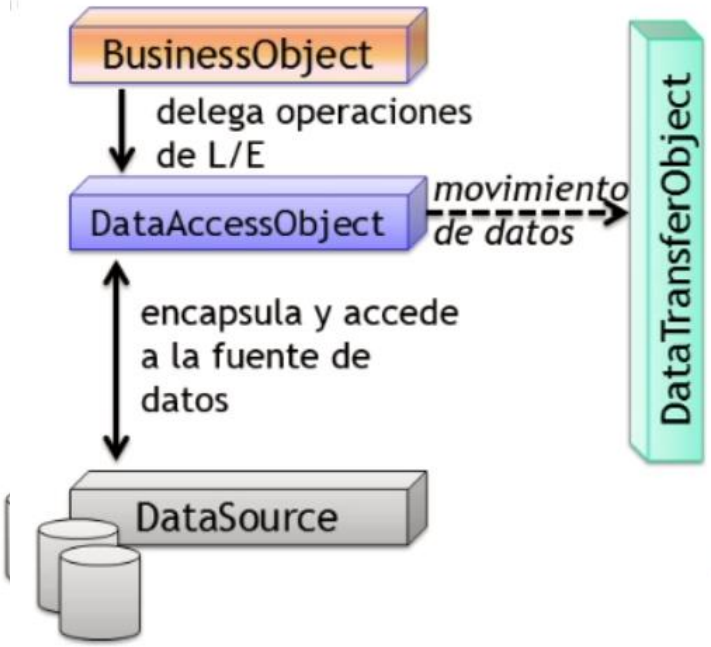
Estructura del patrón DAO



Más info en: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

Estructura del patrón DAO - Elementos

- **BusinessObject**: objeto de la capa de negocio que necesita acceder a la fuente de datos, para leer o almacenar
- **DataAccessObject (DAO)**: abstrae la implementación subyacente del acceso a datos a la capa de negocios para conseguir un acceso transparente a la fuente de datos. BusinessObject delega en el DAO las operaciones de lectura y escritura de datos
- **DataTransferObject (DTO)**: representa el objeto portador de los datos. DAO puede devolver los datos al BusinessObject en un DTO. El DAO puede recibir los datos para actualizar la BD en un DTO
- **DataSource**: implementación de la fuente de datos (SGBDR, SGBDOO, SGBDOR, repositorio XML, ficheros planos, etc.)



Ventajas del patrón DAO

- **Encapsulación**. Los objetos de la capa de negocio no conocen detalles específicos de implementación del acceso a datos, ocultos en el DAO.
- **Migración más fácil**: migrar el sistema a un gestor de datos diferente supone cambiar la capa de DAO por otra.
- **Menor complejidad** en la capa de negocio al aislarse del acceso a datos.
- **Centraliza** el punto de acceso a datos.

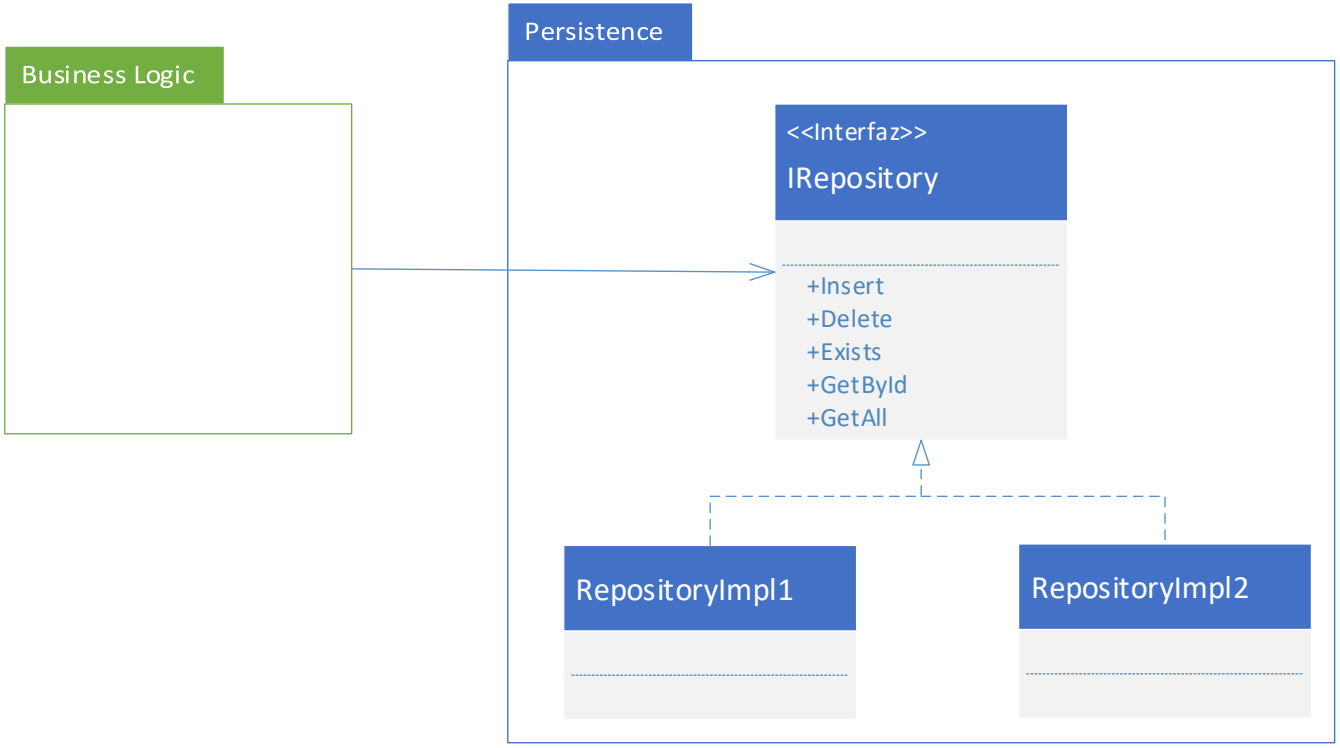
Inconvenientes del patrón DAO

- Arquitectura software ligeramente **más compleja**
- Hay que implementar **más código** para ofrecer ese nivel de indirección adicional
- Desde el punto de vista de la eficiencia se puede **ralentizar** el proceso

PATRÓN REPOSITORIO + UNIDAD DE TRABAJO

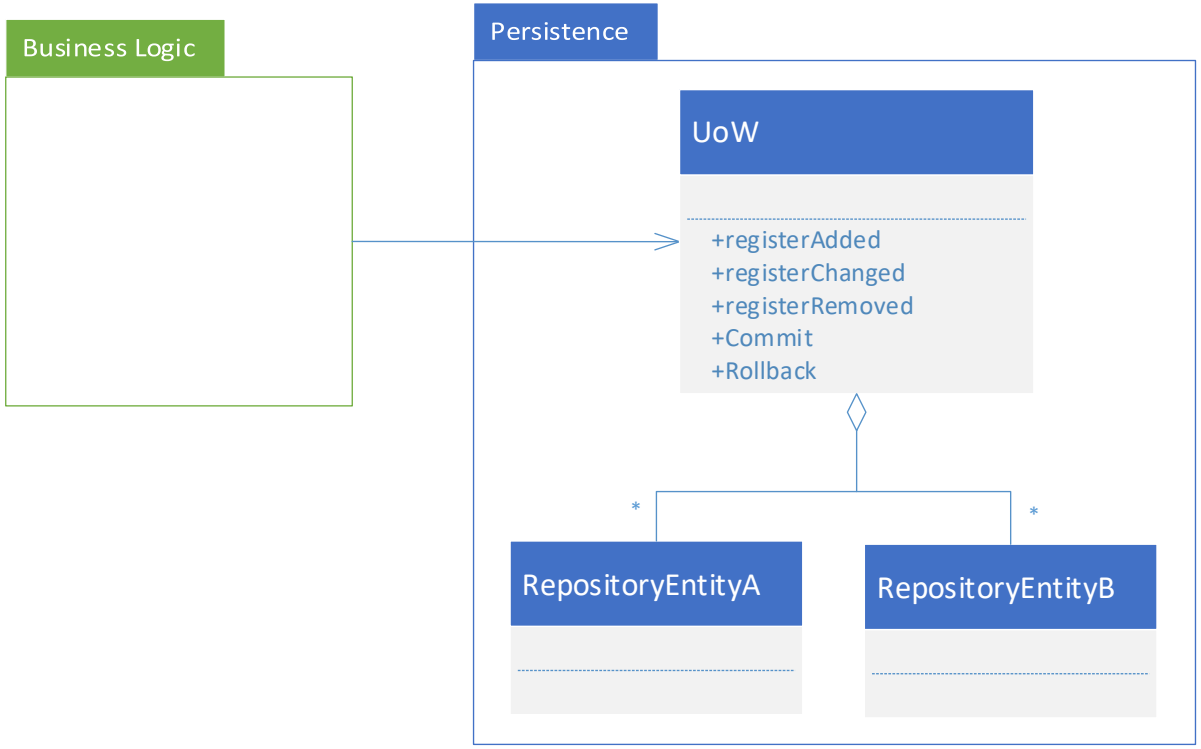
Patrón Repositorio

- **Repositorio** (Repository): puente entre la capa de dominio y la de persistencia (típicamente una BDOR). Ofrece una interfaz tipo colección para acceder a los objetos de dominio directamente (sin DTOs).



Patrones Repositorio + Unidad de Trabajo

- **Unidad de Trabajo** (Unit of Work, or UoW): mantiene un registro de los objetos afectados durante una transacción (operación atómica sobre objetos de negocio) y se encarga de persistir los cambios (o deshacerlos en caso de abortar la transacción) y resolver cualquier problema de concurrencia que pudiera surgir.



El patrón Repositorio

- Ha aumentado su popularidad desde que fue introducido como parte del enfoque denominado **Domain Driven Design** (Evans, 2004)
- Ofrece una **abstracción** para acceder a los datos como si fueran **colecciones en memoria**.
  - Interfaz con métodos para añadir, borrar, actualizar y buscar objetos del dominio de forma directa, sin necesidad de crear objetos de transferencia
  - Ayuda a reducir el acoplamiento entre la lógica de negocio y la capa de persistencia: objetos de negocio ignorantes respecto al mecanismo de persistencia

Un repositorio por objeto del dominio

- El enfoque más sencillo, especialmente cuando se introduce en un sistema antiguo, es crear un repositorio por cada objeto del dominio a persistir
- Solo se necesita implementar en cada repositorio los métodos que se vayan utilizar y nada más
- El mayor beneficio de este enfoque es que no se pierde tiempo implementando métodos que nunca se van a utilizar (YAGNI, *You Aren't Gonna Need It*)

Repositorio genérico

- Otro enfoque consiste en crear un repositorio genérico válido para cualquier objeto de dominio. Se necesita una sola interfaz y una única implementación por cada mecanismo de persistencia que queramos utilizar (y no una implementación por cada clase a persistir)
- Un ejemplo de interfaz para un repositorio genérico en C# sería:

```
public interface IRepository
{
  void Add<T>(T entity) where T : class;
  void Delete<T>(T entity) where T : class;
  T GetById<T>(IComparable id) where T : class;
  bool Exists<T>(IComparable id) where T : class;
  IEnumerable<T> GetAll<T>() where T : class;
}
```

Repositorio vs DAO

- El **concepto de DAO** está más cercano al mecanismo de persistencia subyacente; es un enfoque **centrado en los datos**. Por eso normalmente se crea un DAO por cada tabla o vista de una base de datos.
- El **concepto de repositorio** se encuentra más cercano a la capa de negocio, pues trabaja directamente con los objetos de negocio.
  - Internamente, un repositorio podría usar BDOR, pero también mecanismos de bajo nivel como DAO/DTO
- En *dominios anémicos* (sin verdadera lógica de negocio, solo get/set), repositorio y DAO son intercambiables

Patrón Unidad de Trabajo

- Actualizar la información en la base de datos cada vez que se modifica un objeto del dominio:
  - Es poco eficiente, incurriría en muchos cambios pequeños, y si algo sale mal durante una transacción hay que deshacer los cambios
  - Conduce a problemas de consistencia en el acceso concurrente (si otros clientes han obtenido los datos con cambios que ha habido que deshacer)
- Una Unidad de Trabajo mantiene un registro de cambios en los objetos de negocio durante una transacción completa. No realiza los cambios en el mecanismo de persistencia hasta que no se complete la transacción, y si hay que deshacer algo se soluciona en memoria
  - Es más eficiente y evita problemas de consistencia en el acceso concurrente

Patrón Unidad de Trabajo

- **No necesariamente hay que implementarlo ad-hoc para cada proyecto** ya que lo encontramos habitualmente en frameworks de persistencia, por ejemplo
  - La interfaz ITransaction en Nhibernate
  - La clase **DbContext** en **Entity Framework**
- Para una explicación más detallada sobre el patrón Repository + UoW en C# (EF5 & MVC4, con ejemplos) se recomienda este artículo. También hay una versión para EF6 & MVC5 : http://bit.ly/2OpGpvc.

PERSISTENCIA EN BDOR Y BDOO

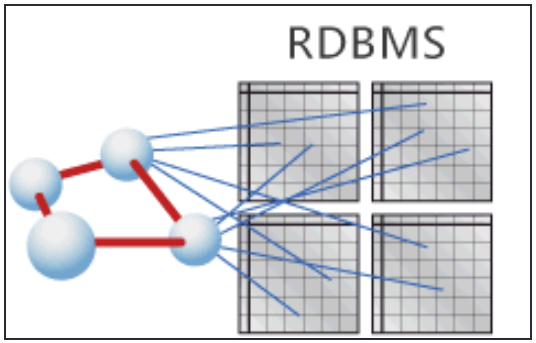
- Objetivos
- Ventajas
- Referencias adicionales

Objetivos

- En lugar de implementar una BD relacional se proporciona una BDOO (no relacional)
  - El almacenamiento interno representa a los objetos como tal (no disgregados en tablas)
  - Ya no es necesario un middleware objetual-relacional
    - No obstante, las BDOR son un modelo mixto (extensión de una BDR para manejar el paradigma OO) y también soportan SQL
  - La mayoría de operaciones se realizan de forma más eficiente, pues no es necesario manejar datos en distintas tablas
    - Ej. cuando se accede a una relación no se dispone de la clave ajena para recuperar el registro de otra tabla sino del propio objeto relacionado

BDOR y BDOO

- En sistemas complejos resulta tedioso tener que realizar la conversión entre datos de un modelo OO y los de un modelo relacional
  - Paso de las características del lenguaje de programación a SQL y viceversa



- Existen diversas herramientas que hacen esta correspondencia automáticamente para varios lenguajes (C#, Java, VB...)
  - Ej. Entity Framework, Hibernate, etc.

ENTITY FRAMEWORK

HIBERNATE

Algunos ejemplos

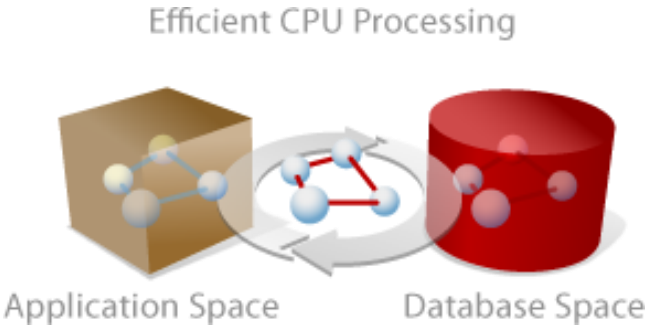
```
1  public void store(Car car){
2      ObjectContainer db = Db4o.openFile("car.yap");
3      car.engine(new TurboEngine());
4      db.set(car);
5      db.commit();
6      db.close();
7  }
```

```
// OPEN THE DATABASE
d_Database db;
db.open( "business" );

d_Transaction tx;
tx.begin();

d_Ref obj;
obj = new( &db, "Customer" ) Customer();

obj->name = "Luke";
obj->surname = "Skywalker";

// INSERT THE OBJECT AS "MYFRIEND"
db.set_object_name( obj, "MyFriend" );

tx.commit();
```

```
// OPEN THE DATABASE
d_Database db;
db.open( "business" );

d_Transaction tx;
tx.begin();

d_Ref obj;

// RETRIEVE THE ENTRY CALLED "MYFRIEND"
obj = db.lookup_object( "MyFriend" );

// DISPLAY THE CUSTOMER NAME
cout << "MyFriend is: " << obj->name;

tx.commit();
```

Ventajas

- se une la potencia de los objetos con la flexibilidad de SQL (en algunos casos)
- se simplifica en gran medida el desarrollo de aplicaciones en capas
  - no es necesario escribir consultas SQL
  - solo requiere un modelo del dominio (y no 2)
  - la comunicación entre todas las capas es en forma de objetos, sin conversión/mapping



Resumen

- Los patrones de acceso a datos (DAO/Repository) permiten abstraer el acceso a la capa de persistencia de su implementación
- Es posible aplicar una sencilla correspondencia para derivar un modelo relacional a partir de un modelo OO
- Las BD objetuales simplifican el desarrollo de aplicaciones porque:
  - el modelo de datos se puede proyectar (hacia o desde la aplicación) sin necesidad de establecer correspondencias
  - las operaciones son simples operaciones sobre objetos
  - en general, también son más simples de usar y más eficientes

Bibliografía básica

- Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004
- Feddema H.B., DAO object model: the definitive reference. O'Reilly, 2000.
- Fowler, M., Patterns of Enterprise Application Architecture. Addison-Wesley, 2002