PERSISTENCE DESIGN

Chapter 6

Software Engineering
Computer Science School
DSIC – UPV

Goals

- Understand the need of maintaining the persistence in the development of software
- Know the Data Access Patterns to be used in implementation to achieve a layer abstraction
- Understand the DB object model vs. the relational model and know its advantages
- *Note:* The relational logical design of the DB from an OO model (class diagram) will be covered in another course (Databases).

Contents

1. Introduction
2. DAO, Repository and UoW patterns
3. Persistence in ORDB and OODB
4. Conclusions

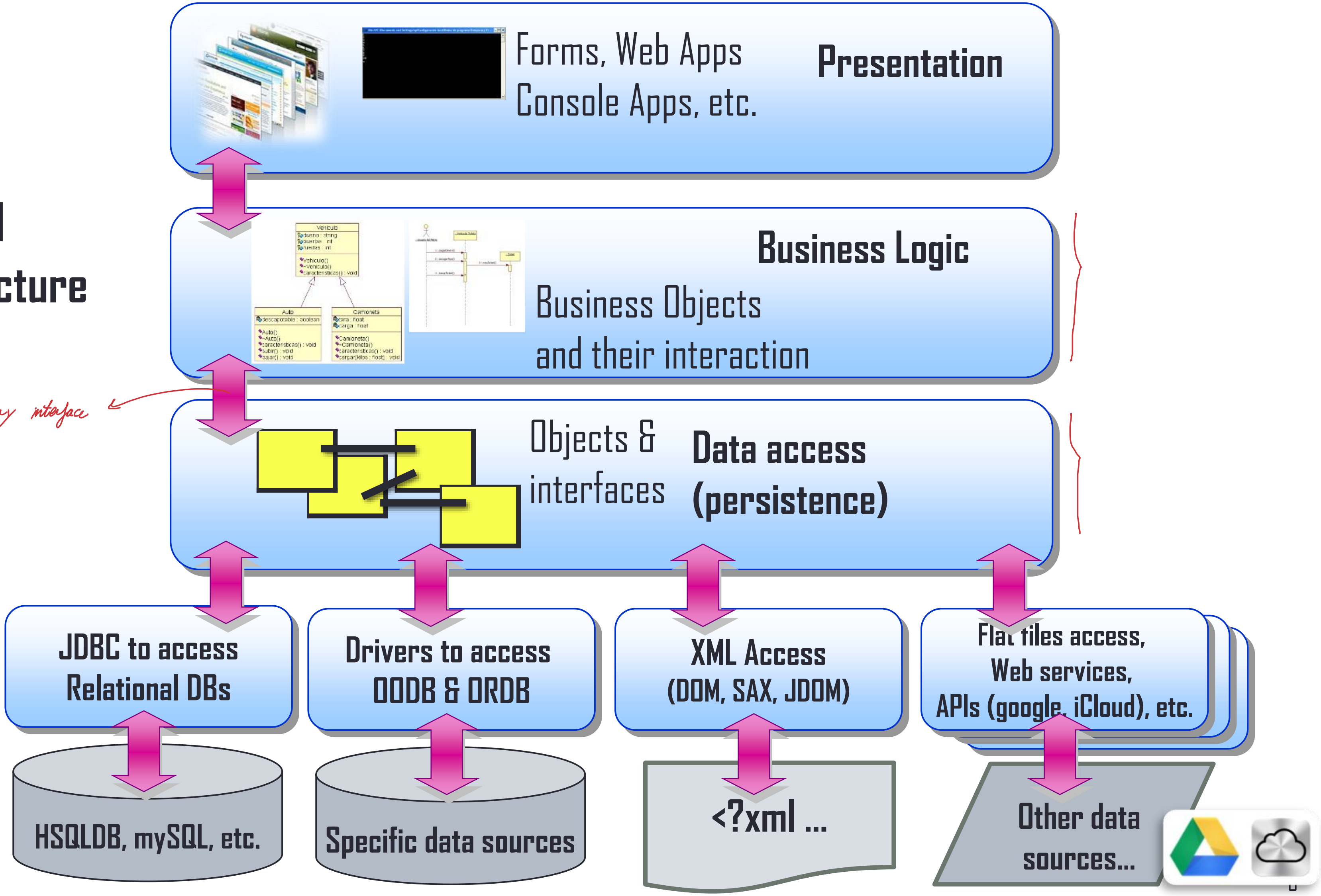
INTRODUCTION

Introduction

- In most applications the storage of non volatile information is essential
  - A specific format may be used for each application (limited compatibility)
  - A structured or relational format based on DB may be used (greater compatibility – based on standards such as SQL)
- The use of DBs results in using libraries to manage the access to data (JDBC, ADO, ODBC, etc.)

Introduction

Layered Architecture



I repository interface

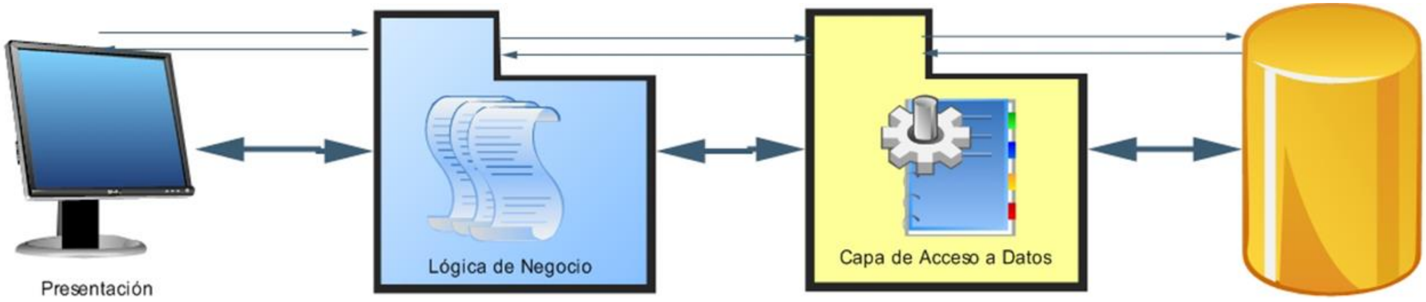
DAO DATA ACCESS PATTERN

- ✓ Structure
- ✓ Pros and Cons
- ✓ Implementation

Data access classes in the implementation

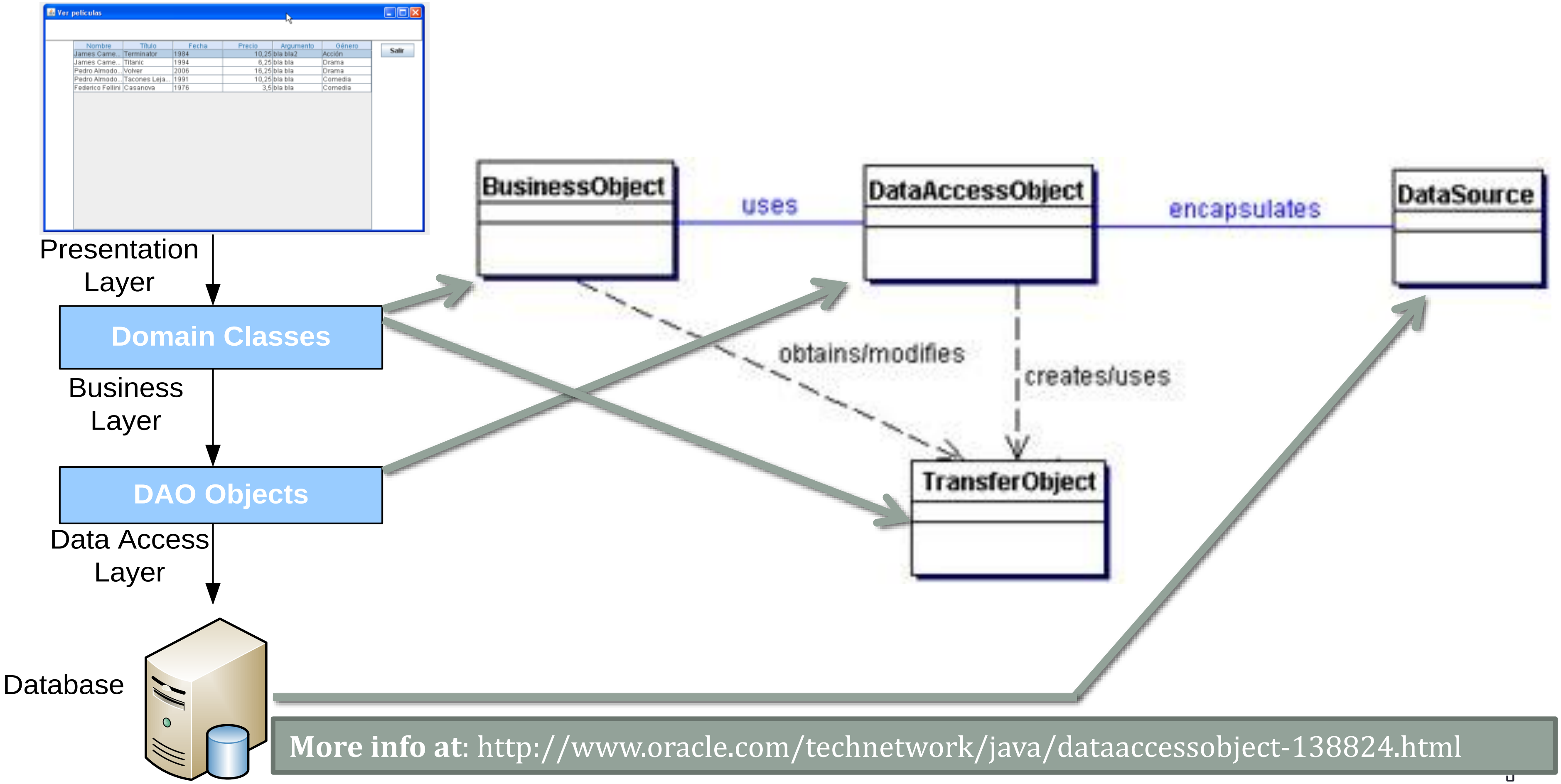
They are implementation bridges between:

- Data stored in objects
- Data stored in a relational DB
- Having methods to add, update, search and remove records
- Encapsulating the necessary logic to copy data values from classes of the problem domain (business logic layer) to the DB and viceversa



DAO Pattern Structure

Graphical structure of the **DAO pattern (Data Access Objects)**



Ver peliculas

Nombre	Título	Fecha	Precio	Argumento	Género
James Came...	Terminator	1984	10,25	bla bla2	Acción
James Came...	Titanic	1994	6,25	bla bla	Drama
Pedro Almodo...	Volver	2006	16,25	bla bla	Drama
Pedro Almodo...	Tacones Leja...	1991	10,25	bla bla	Comedia
Federico Fellini	Casanova	1976	3,5	bla bla	Comedia

Salir

More info at: http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

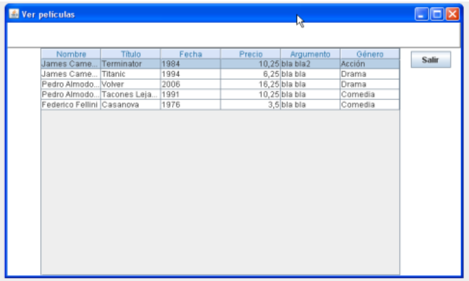
DAO Pattern

Structure of the DAO pattern. **Elements**

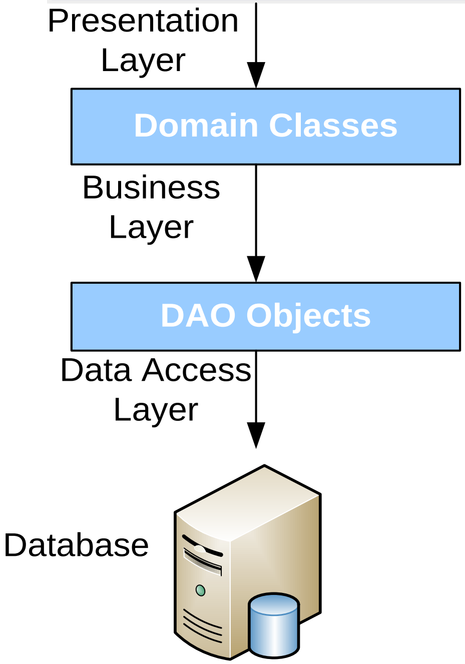
- **BusinessObject**: object of the business layer that needs access to the data storage to read or write information
- **DataAccessObject**: abstraction of the implementation of the data access layer. BusinessObject delegates the DAO all read/write operations
- **DataTransferObject** (DTO): represents an object holding data. DAO may return data to BusinessObject by means of a DTO. The DAO may receive data in a DTO to update the DB
- **DataSource**: implementation of the data source (RDBMS, OODBMS, XML repository, raw files, etc.)

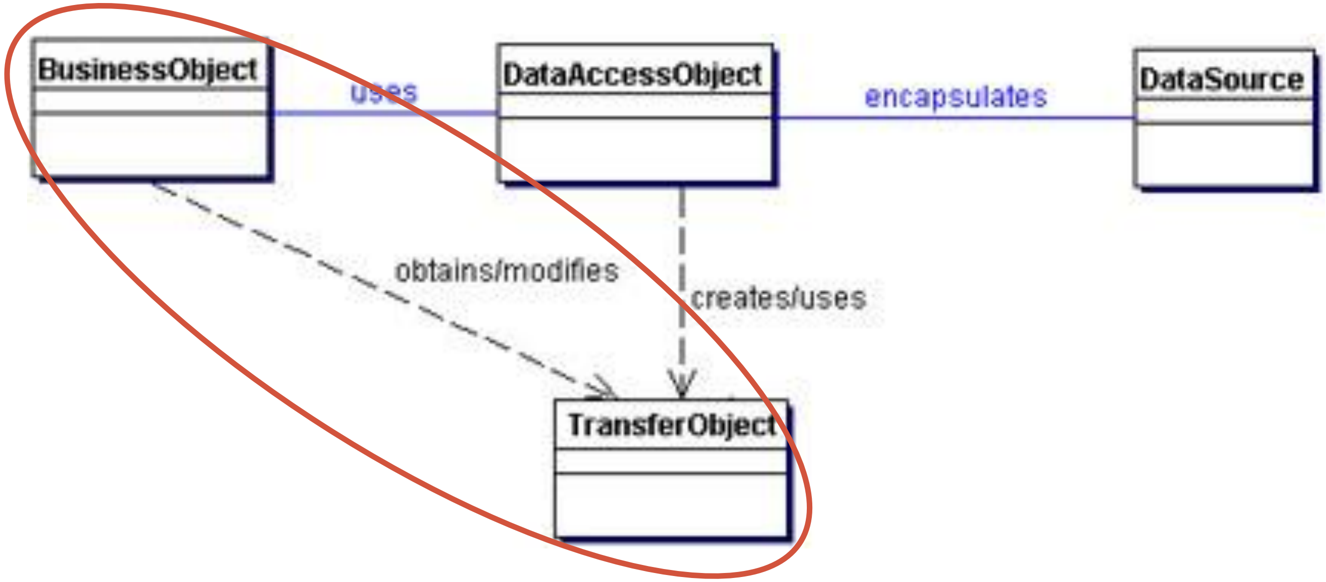
DAO Pattern Structure

Graphical structure of the **DAO pattern (Data Access Objects)**



Nombre	Título	Fecha	Precio	Argumento	Género
James Came...	Terminator	1984	10,25	bla bla2	Acción
James Came...	Titanic	1994	6,25	bla bla	Drama
Pedro Almodo...	Volver	2006	16,25	bla bla	Drama
Pedro Almodo...	Tacones Leja...	1991	10,25	bla bla	Comedia
Federico Fellini	Casanova	1976	3,5	bla bla	Comedia





For simplicity, these two classes may become just one which will be the BusinessObject (the TransferObject class is removed)

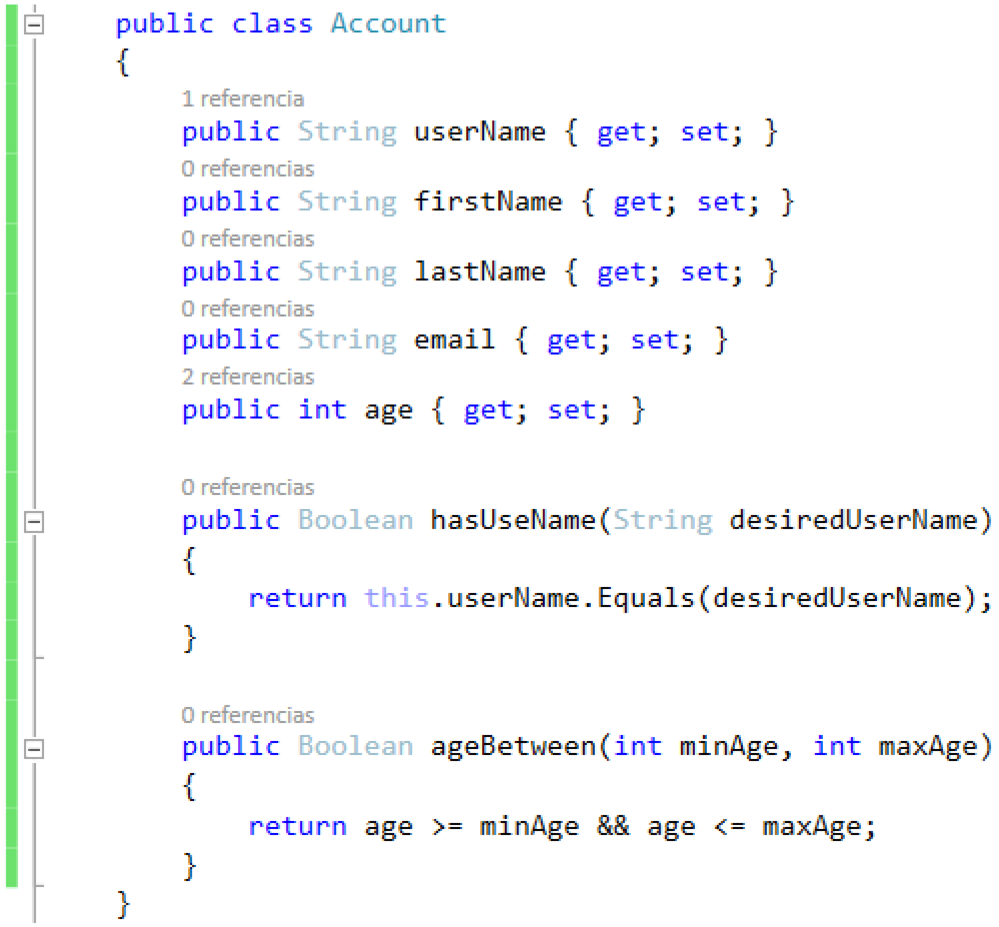
Implementation

Step by step DAO pattern implementation

1. Take as starting point the Business Logic classes
2. Define an interface for each DAO,
   - A DAO interface **for each domain class** with CRUD operations (Create, Read, Update, Delete) and any other needed operations
3. Define a class for each interface implementing its functionality
   - This class will know the details about how to access the data (e.g. SQL statements)

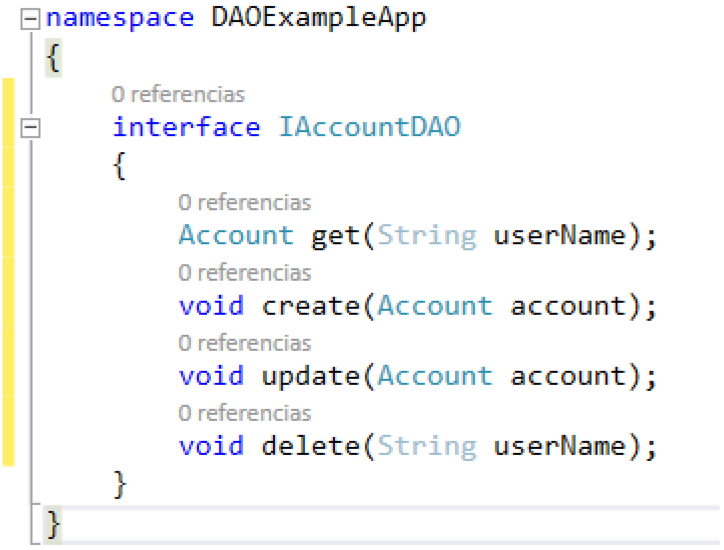
DAO Pattern: Example

Domain class



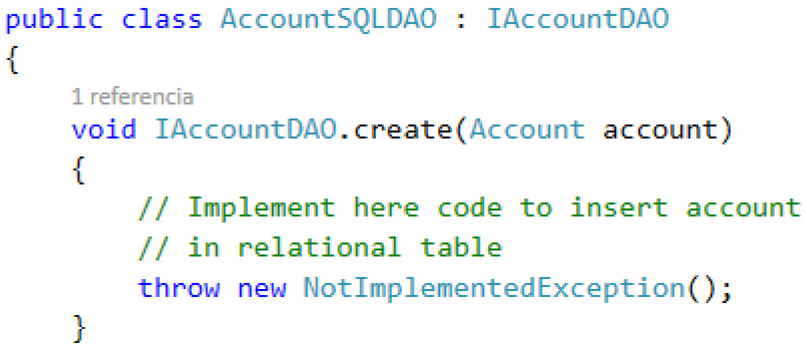
```
public class Account
{
    1 referencia
    public String userName { get; set; }
    0 referencias
    public String firstName { get; set; }
    0 referencias
    public String lastName { get; set; }
    0 referencias
    public String email { get; set; }
    2 referencias
    public int age { get; set; }

    0 referencias
    public Boolean hasUseName(String desiredUserName)
    {
        return this.userName.Equals(desiredUserName);
    }

    0 referencias
    public Boolean ageBetween(int minAge, int maxAge)
    {
        return age >= minAge && age <= maxAge;
    }
}
```

DAO Interface



```
namespace DAOExampleApp
{
    0 referencias
    interface IAccountDAO
    {
        0 referencias
        Account get(String userName);
        0 referencias
        void create(Account account);
        0 referencias
        void update(Account account);
        0 referencias
        void delete(String userName);
    }
}
```

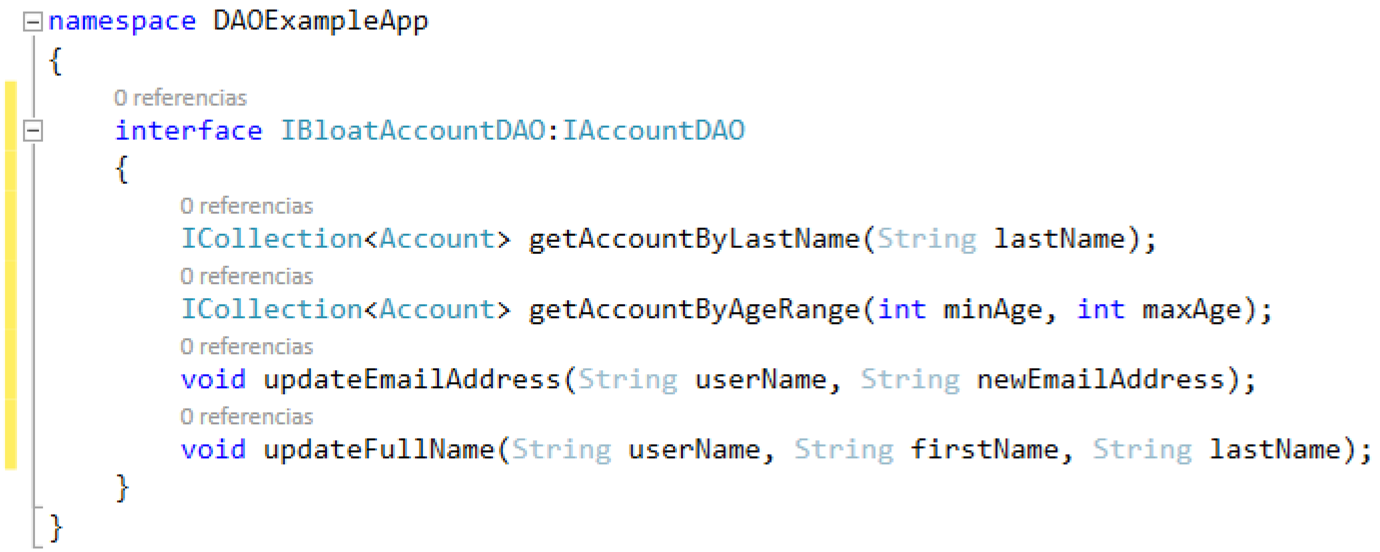
DAO Class



```
public class AccountSQLDAO : IAccountDAO
{
    1 referencia
    void IAccountDAO.create(Account account)
    {
        // Implement here code to insert account
        // in relational table
        throw new NotImplementedException();
    }
```

DAO Pattern: Example

- What if more specific queries are needed?
- What if more specific updates are needed?



```
namespace DAOExampleApp
{
    0 referencias
    interface IBloatAccountDAO:IAccountDAO
    {
        0 referencias
        ICollection<Account> getAccountByLastName(String lastName);
        0 referencias
        ICollection<Account> getAccountByAgeRange(int minAge, int maxAge);
        0 referencias
        void updateEmailAddress(String userName, String newEmailAddress);
        0 referencias
        void updateFullName(String userName, String firstName, String lastName);
    }
}
```

DAO Pattern: Example

- End up with a fat DAO encouraging to add even more methods to it in the future

- The DAO interface becomes more coupled to the fields of Account object. I have to change the interface and all its implementations if I change the type of fields stored in Account.

- Mocking the DAO interface becomes harder in unit test. I need to implement more methods in the DAO even my particular test scenario only uses one of them.

DAO Pattern

DAO pattern. **Advantages**:

- **Encapsulation**. Objects of the business layer do not know specific details of the implementation of the data access (hidden in the DAO).
- **Easier migration**: migrating to a different DBMS just involves changing the DAO layer.
- **Less complexity** in the business layer because the access to data is isolated.
- Data access **centralized** in a layer.

DAO Pattern

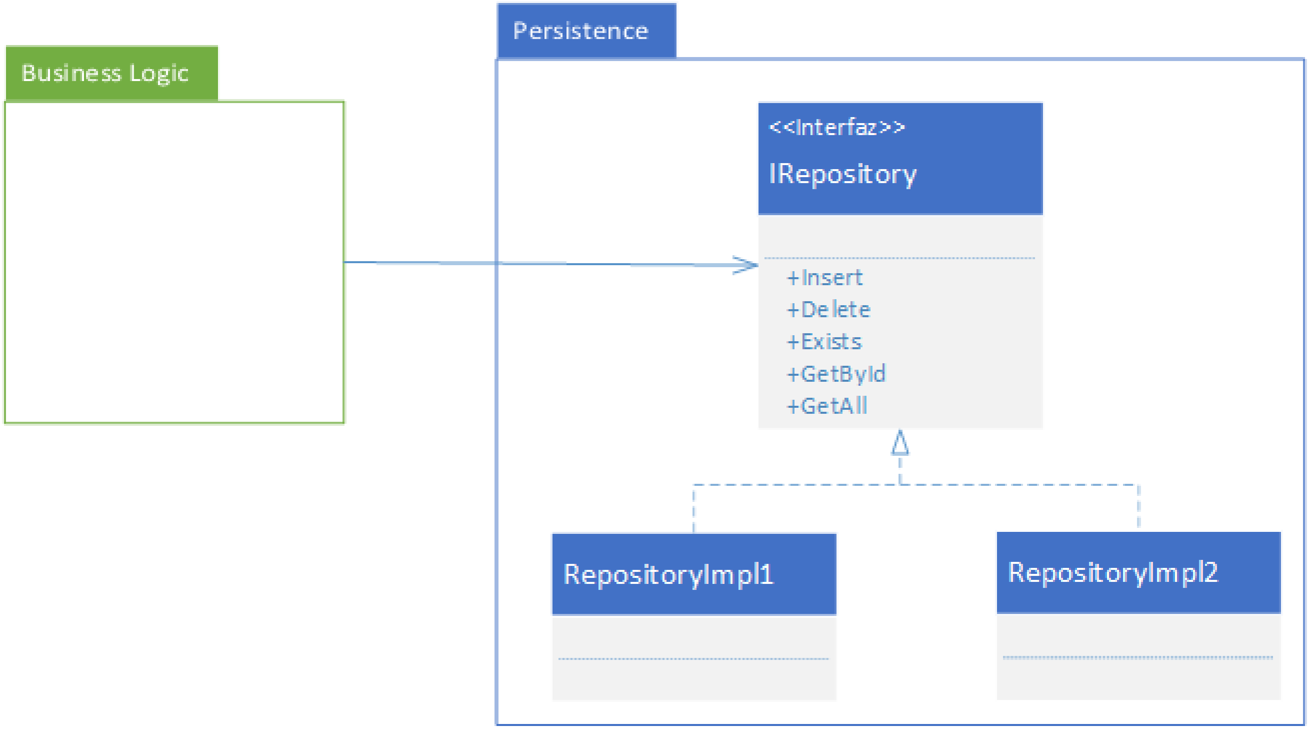
DAO pattern. **Disadvantages**:

- Sw architecture slightly more complex
- Additional code for the layer must be developed
- From an efficiency perspective the process may be slower
- Coupled to the fields of domain objets
- May affect maintainability of the code

REPOSITORY + UNIT OF WORK PATTERNS

Repository Pattern

- **Repository**:
  - Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.
  - Interface is fixed (well-defined contract) and independent of the fields of classes



The Repository Pattern

- The Repository Pattern has gained popularity since it was first introduced as a part of Domain-Driven Design (Evans, 2004)

- A Repository provides an abstraction so that data can be accessed as if it was an in-memory collection.
  - Adding, removing, updating, and selecting items from this collection is done through a series of straightforward methods, without the need to deal with database concerns
  - Using this pattern can help achieve loose coupling and make business objects persistence ignorant.

Repository Per Entity or Business Object

- The simplest approach, especially with an existing system, is to create a new Repository implementation for each business object you need to store to or retrieve from your persistence layer.
- Further, you should only implement the specific methods you are calling in your application
- The biggest benefit of this approach is YAGNI (**You ain't gonna need it**)– you won't waste any time implementing methods that never get called

Repository = "bag of objects"

APPROACHES TO IMPLEMENT IT:

1) 1 repository per class → Disadvantage = high n°

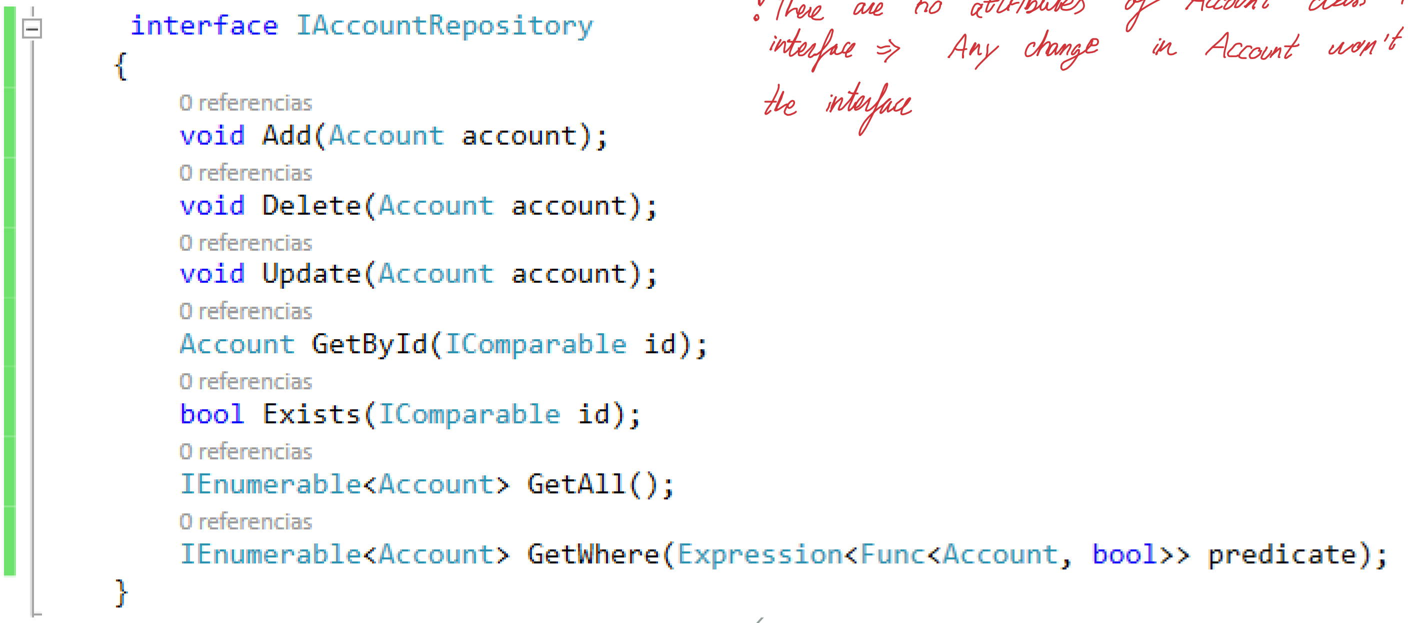
2) Create GENERIC INTERFACE

Repository: Example

- An interface that is independent of the fields of the domain class

FIRST APPROACH :

! There are no attributes of Account class in the interface ⇒ Any change in Account won't affect the interface



```
interface IAccountRepository
{
    0 referencias
    void Add(Account account);
    0 referencias
    void Delete(Account account);
    0 referencias
    void Update(Account account);
    0 referencias
    Account GetById(IComparable id);
    0 referencias
    bool Exists(IComparable id);
    0 referencias
    IEnumerable<Account> GetAll();
    0 referencias
    IEnumerable<Account> GetWhere(Expression<Func<Account, bool>> predicate);
}
```

Linq Expression, See our Linq seminar

Generic Repository

- Another approach is to go ahead and create a simple, generic interface for your Repository.
- An example of a generic C# repository interface might be:

SECOND APPROACH

!) The implementation depends on the technology used (Entity Framework)

```
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;
  IEnumerable<T> GetWhere<T>(Expression<Func<T, bool>> predicate) where
T : class;
    }
```

!) This will be used by Business Logic layer

One implementation for each specific persistence technology
instead of
One implementation per domain class

Repository vs DAO

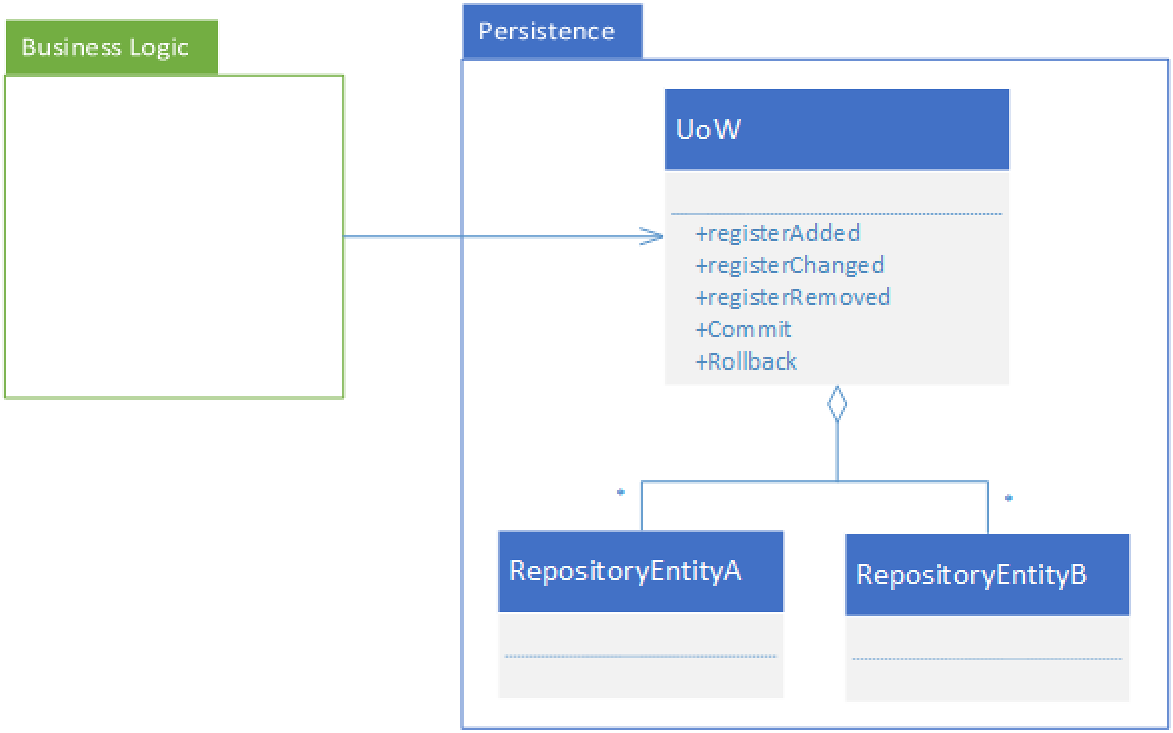
- DAO is much closer to the underlying storage , it’s really data centric. That’s why in many cases you’ll have DAOs matching db tables or views 1 on 1. A DAO allows for a simpler way to get data from a storage, hiding the ugly queries. But the important part is that they return data as in **object state**.

DAO pattern allows to extend it as you like | Repository restricts the n° of things you can do ⇒
⇒ SIMPLIFY

- A repository sits at a higher level. It deals with data too and hides queries and all that but, a repository deals with **business/domain objects**. A repository will use a DAO to get the data from the storage and uses that data to restore a business object. Or it will take a business object and extract the data that will be persisted.
- Suggested Readings
  - https://medium.com/@jotauribe/data-access-objects-vs-repositories-b1497565a873
  - https://thinkinginobjects.com/2012/08/26/dont-use-dao-use-repository/

Repository + Unit of Work Patterns

- **Unit of Work** (UoW): Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.



Unit of Work Pattern

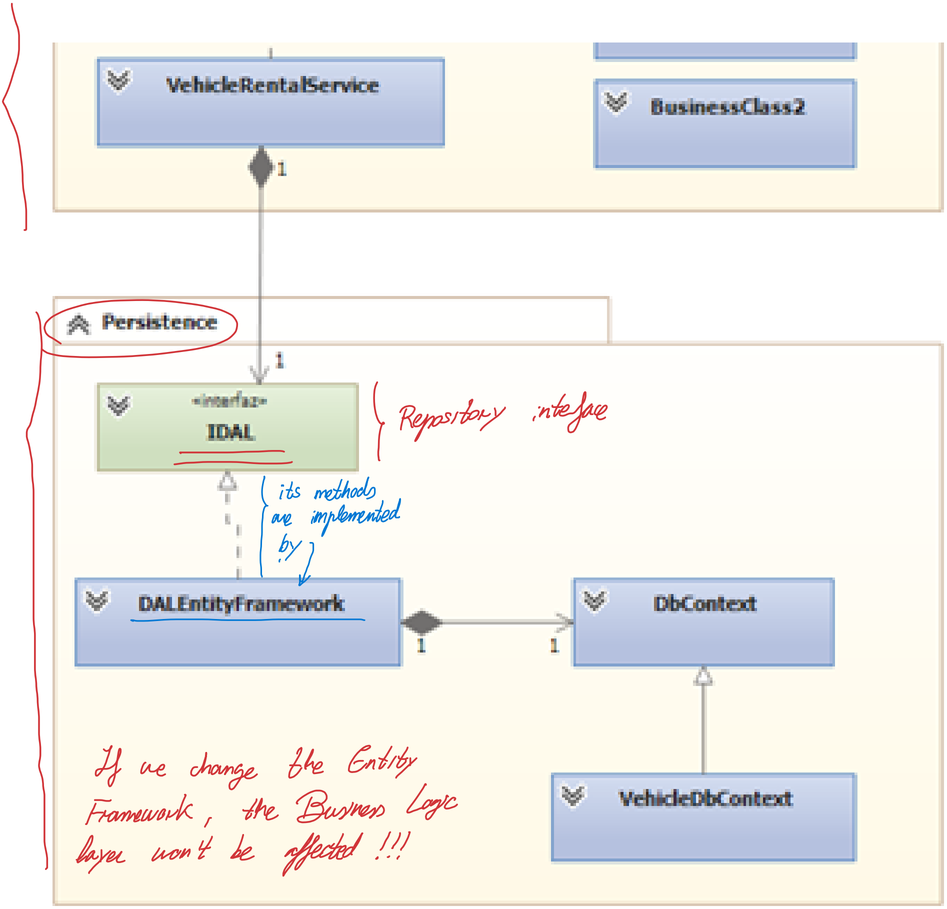
- **The Unit of Work pattern isn't necessarily something that you will explicitly build yourself**, but the pattern shows up in almost every modern persistence tool.
  - The ITransaction interface in Nhibernate
  - The DbContext class in the Entity Framework

- For a detailed tutorial on the Repository + UoW patterns in C# (EF5 & MVC4) see this Microsoft article. There is also a version for EF6 & MVC5

Layers Separation. Persistence

- It provides **access to a data source** (relational DB, OODB, XML files, etc.)
- The services provided by the persistence layer are specified again as an **interface** (e.g. IDAL)
- **Different implementations** of the interface may be given depeding on the concrete data source (e.g. DALEntityFramework, DALHibernate, DALXML, etc.)
    - By using an interface any change in the implementation of IDAL **does not affect** the business logic layer

BUSINESS LOGIC LAYER



Repository interface

its methods are implemented by

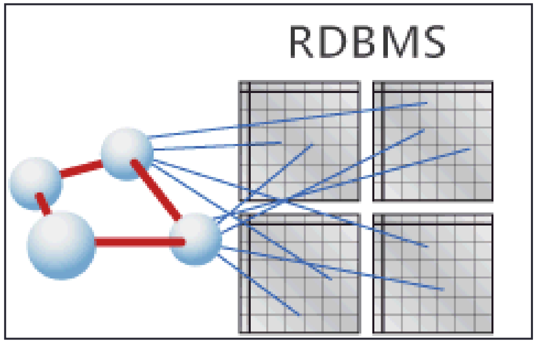
If we change the Entity Framework, the Business Logic layer won't be affected !!!

OBJECT DESIGN

- ✓ Goals
- ✓ An example. Caché Intersystems

Object Design

- In complex systems it is tedious to convert data between OO and Relational models
  - Mapping features from programming language to SQL and viceversa



- There are several tools that perform automatically this mapping for several languages (Java, C#, VB...)
  - Ex. Hibernate, **Entity Framework**

Object Design. Goals

Instead of implementing a relational DB an OO DB is provided

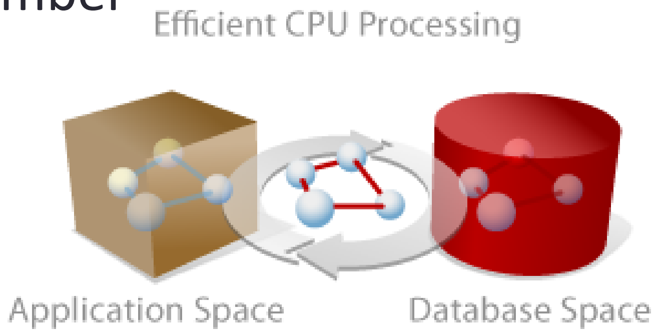
- The internal storage represents objects as such (no dispersion in tables)
- No object-relational or object-SQL middleware
- Most operations are implemented in a more efficient way, no need to manage data in different relational tables
    - Ex., when a relationship is accesed there is no foreign key to retrieve the record of another table but we have the referenced object itself

Object Design. Advantages

There are important associated advantages:

- The power of objects with the flexibility of query languages all together

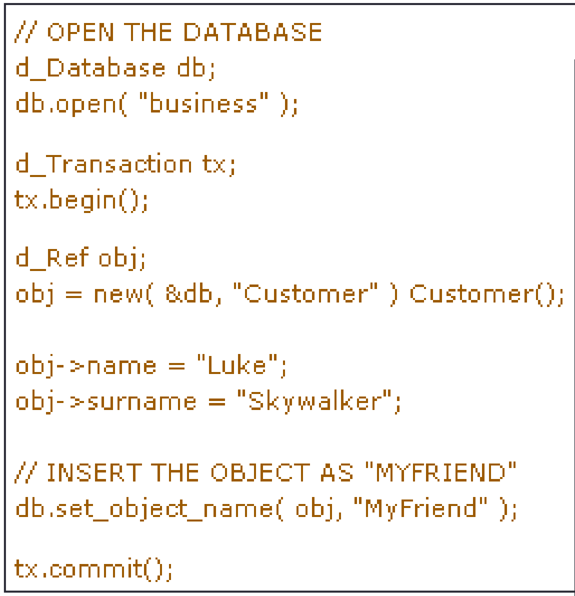
- The development of layered applications is simplified
  - No need for a persistence layer based on SQL and a business logic based on objects (no need for constructing the model twice)
  - The logic layer communicates with the persistence layer by means of objects without conversion/mapping
  - Costant access in terms of “object.member”



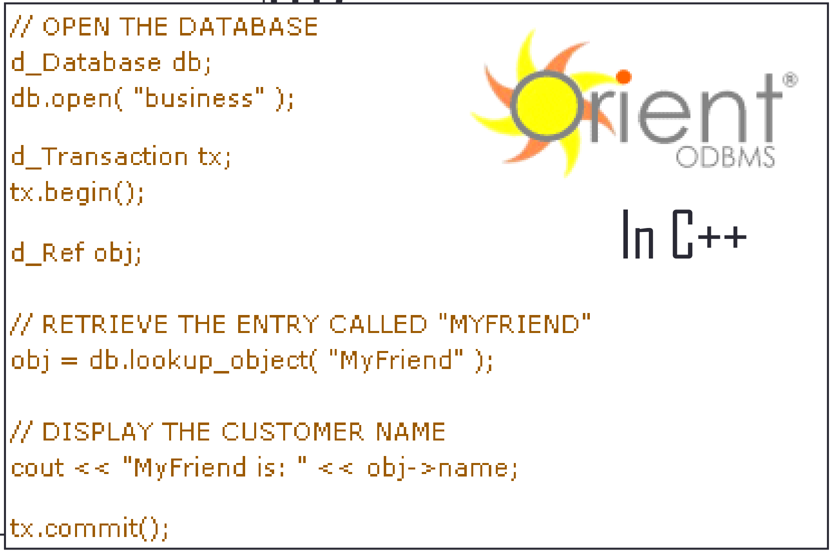
Object Design. Additional systems

Open Source Object Database (OODB):

- db4objects (www.db4o.com)
- ObjectStore (www.progress.com/es-es/objectstore)
- Objetivity (www.objectivity.com)
- Orient (m)



```
// 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();
```

In C++

Object Design. Additional free OODBMS

- **Goods**, Generic OO Database System
  www.garret.ru/goods.html

- **JDOInstruments**, embedded OO database for Java Data Objects
  sourceforge.net/projects/jdoinstr

- **Ozone**, Java based OO database management system java-source.net/
  source/database-engines/ozone

CONCLUSIONS

To sum up

- The data access patterns (DAO/Repository) provide an abstraction to the persistence layer

- It is possible to apply a simple mapping to derive the relational model from a class diagram
  - In some cases several models are posible

- OODBs simplify the development of applications because:
  - The data model is built once and may be projected from/to the application without mappings
  - Operations and data management is simple, just operations on objects and their relationships
  - In general they are simple and efficient

References

- Fowler, M., *Patterns of Enterprise Application Architecture,* Addison-Wesley, 2002
- Evans: *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.
- Core J2EE Patterns: Best Practices and Design Strategies, 2nd Edition (chapter on Data Access Object Pattern)
- Gamma et al., *Design patterns: elements of reusable object-oriented software, 1994,* Addison Wesley