

# Re: "Business Objects" and the DAL

---

*Source:*

<http://www.tech-archive.net/Archive/DotNet/microsoft.public.dotnet.framework.adonet/2006-02/msg00509.html>

---

- *From:* "Steve" <sss@xxxxxxx>
  - *Date:* Thu, 16 Feb 2006 10:48:30 -0800
- 

This is incredible! I need to review my original post and determine how I wrote that to catch 2 amazing responses like this!

Mike, thank you so much, I'm following right along with you and soaking all this knowledge up, this is a great post.

Thank you for being so thorough,  
Steve Klett

PS: Also printing yours :)

"Mike" <michael.lang@xxxxxxxxxxxx> wrote in message  
<news:1140110441.136567.66220@xx>

The most important thing about n-tier application design is that each layer talks to the layer next to it. The purpose for this is abstraction. If you have three layers, UI, Business, and Data, then the UI layer should never talk to the Data layer (and vice versa).

I've seen people create a BAL class and DAL class for every entity because they wanted to implement the "traditional three tier application design". The data entity returned a dataset, and the business entity returned that up to the UI for binding to a grid. But there is a problem. This does not include any abstraction between database entities and the UI. If a database table column names changes, then the entire application has to adjust, including the datagrid that is bound to the dataset.

I propose an alternative. Use custom entities. The business layer is responsible for returning the same consistent entities no matter how the database and data layer code changes. The "data layer" is for calling database stored procedures to update, insert, delete, or select data. Next you'll question how you organize these layers and classes into components. There are multiple schools of thought on that, and I'll point out a few below.

Start by defining what your entities are, and what properties those entities have. Also map out what the relationships between the

## Re: "Business Objects" and the DAL

entities are. You may want to do all this on paper or a UML tool like Visio before you write any code.

Next what database entities will you need to call to fill these entities? Are you using stored procedures, or are you using dynamic \*parameterized\* queries. I strongly discourage dynamically build unparameterized queries. What parameter values are required for the updates/inserts/deletes? What columns will be returned by the select queries or stored procedures? Have you accounted for all these required items in your entity designs?

Once you have all that information, it is time to design how you'll organize your components / layers. As I said before, I've seen multiple ways of doing this. There is no one "correct" way to do it. Each could be considered a "pattern", and most are documented somewhere. Many companies may have designed their own pattern that is similar to another published pattern. In rare cases, some companies have some great new pattern. I can't tell you which pattern you should use for your scenario, but I can point you in the direction for a few of them.

Pattern 1: "Business Layer Fully Functional Entities". This doesn't map to an exact pattern that I've seen anywhere, so I gave it my own name. Essentially there is a single business layer. It contains the entities, and each entity contains Save, Load, and Delete methods. These methods may either be static or instance. The static methods may return a collection of that entity type. An organization that uses this pattern typically has their own common data component that makes the actual calls to the database. Some of these components may be database neutral. An example that many use as a base is the Microsoft Enterprise Data Application Block. These business entity methods call the custom data component to make database calls. Then the UI calls these business classes to perform operations, and uses those entities to bind to their datagrid, or other controls.

Pattern 2: "Typical UI – BAL – DAL 3 tier design". This design has one extra tier than pattern 1. The BAL is identical, except that instead of calling the database component directly, it makes an identical call on a same named DAL class. The dal class handles the decision as to what database should be called. This design may or may not use a separate data operations component. This component may return a DataReader, a DataSet, or a special more lightweight data transport container to the BAL layer.

Pattern 2b: "UI – BAL – DAL factories" the only difference in this pattern is that the DAL classes are not the same name as the BAL classes. Instead functionality is grouped into data factories, which may not map as simply to the business entities. Instead it maps to the database tables. If the business entities are a direct map to the database tables, then pattern 2 and 2b are effectively the same, except that the DAL classes have the suffix "Factory".

## Re: "Business Objects" and the DAL

Pattern 3: "Provider Pattern". This pattern is used extensively in the .net framework 2.0. Have you seen the membership system? It uses the provider pattern. You can create your own MembershipProvider that determines how each call is made. In other words, you decide what database to call, what stored procedure, and what return columns are used to fill the base framework user class. For an example of the .Net 2.0 membership system, see the following which describes how to use this pattern in .net 1.1:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnaspnet/html/asp04212004.asp>

I use pattern 3 in development of my components. You can see a full API documentation with some examples on my web site here:

<http://www.xquisoft.com/xqsdn/documentation/index.html>

Here are a direct links to examples:

"An employee provider"

<http://www.xquisoft.com/xqsdn/documentation/xquisoft.data.datamanager.html>

"EmployeeFactory and EmployeeDbMap"

<http://www.xquisoft.com/xqsdn/documentation/XQuiSoft.Data.IDataFactory.html>

(The employee class itself is a simple class with a couple public properties only, not shown.)

For more API examples (no source) see the following:

From the table of contents see the XQuiSoft.Security component, User

class, UserManager class, and UserProvider class. This component is basically my "Business layer"

Also see XQuiSoft.Security.Data, DbUserProvider class, UserFactory class, and UserDbMap class. This component is basically my "data layer".

The user interface layer calls the business layer "manager" class to authenticate a user. The ui layer can then store that user in session for the next page request. It could also display that user in an edit page, update it, and then call the "manager" class to save the user.

The actual work in the provider pattern is done by the providers. In this case UserProvider is the abstract base class, and DbUserProvider makes the calls through the database via the UserFactory class. Application configuration determines the provider type that is used to fulfil requests to the manager. UserFactory uses UserDbMap just to get the name constants for the user database tables.

Note that the XQuiSoft.Data component is open source, and you can find it here: <http://sourceforge.net/projects/xqs-data/>

So is my version of the provider pattern in .net 1.1:

<http://sourceforge.net/projects/xqs-provider/>

Michael Lang

Re: "Business Objects" and the DAL

XQuiSoft LLC  
<http://www.xquisoft.com/>