Architecture for a business objects / database access layer

Asked 16 years, 3 months ago Modified 9 years, 5 months ago

Viewed 6k times

7

For various reasons, we are writing a new business objects/data storage library. One of the requirements of this layer is to separate the logic of the business rules, and the actual data storage layer.

It is possible to have multiple data storage layers that implement access to the same object - for example, a main "database" data storage source that implements most objects, and another "ldap" source that implements a User object. In this scenario, User can optionally come from an LDAP source, perhaps with slightly different functionality (eg, not possible to save/update the User object), but otherwise it is used by the application the same way. Another data storage type might be a web service, or an external database.

There are two main ways we are looking at implementing this, and me and a co-worker disagree on a fundamental level which is correct. I'd like some advice on which one is the best to use. I'll try to keep my descriptions of each as neutral as possible, as I'm looking for some objective view points here.

- Business objects are base classes, and data storage objects inherit business objects. Client code deals with data storage objects.

  In this case, common business rules are inherited by each data storage object, and it is the data storage objects that are directly used by the client code.

  This has the implication that client code determines which data storage method to use for a given object, because it has to explicitly declare an instance to that type of object. Client code needs to explicitly know connection information for each data storage type it is using.

  If a data storage layer implements different functionality for a given object, client code explicitly knows about it at compile time because the object looks different. If the data storage method is changed, client code has to be updated.

- Business objects encapsulate data storage objects.

  In this case, business objects are directly used by client application. Client application passes along base connection information to business layer. Decision about which data storage method a given object uses is made by business object code. Connection information would be a chunk of data taken from a config file (client app does not really know/care about details of it), which may be a single connection string for a database, or several pieces connection strings for various data storage types. Additional data storage connection types could also

be read from another spot - eg, a configuration table in a database that specifies URLs to various web services.

The benefit here is that if a new data storage method is added to an existing object, a configuration setting can be set at runtime to determine which method to use, and it is completely transparent to the client applications. Client apps do not need to be modified if data storage method for a given object changes.

- Business objects are base classes, data source objects inherit from business objects. Client code deals primarily with base classes.

  This is similar to the first method, but client code declares variables of the base business object types, and Load()/Create()/etc static methods on the business objects return the appropriate data source-typed objects.

  The architecture of this solution is similar to the first method, but the main difference is the decision about which data storage object to use for a given business object is made by the business layer, not the client code.

I know there are already existing ORM libraries that provide some of this functionality, but please discount those for now (there is the possibility that a data storage layer is implemented with one of these ORM libraries) - also note I'm deliberately not telling you what language is being used here, other than that it is strongly typed.

I'm looking for some general advice here on which method is better to use (or feel free to suggest something else), and why.

orm architecture

Share

Improve this question

Follow

edited Jul 3, 2015 at 21:37

JM.D

139 ● 1 ● 12

asked Sep 23, 2008 at 3:30

gregmac

25.2k ● 11 ● 90 ● 120

7 Answers

Sorted by: Highest score (default)

12

might i suggest another alternative, with possibly better decoupling: business objects *use* data objects, and data objects *implement* storage objects. This should keep the business rules in the business objects but without any dependence on the storage source or format, while allowing the data objects to support whatever manipulations are required, including changing the storage objects dynamically (e.g. for online/offline manipulation)

this falls into the second category above (business objects encapsulate data storage objects), but separates data semantics from storage mechanisms more clearly

Share Improve this answer

Follow

answered Sep 23, 2008 at 3:36

Steven A. Lowe

61.1k • 19 • 135 • 204

1

You can also have a facade to keep from your client to call the business directly. Also it creates common entry points to your business.

As said, your business should not be exposed to anything but your DTO and Facade.

Yes. Your client can deal with DTOs. It's the ideal way to pass data through your application.

Share Improve this answer

Follow

edited Sep 23, 2008 at 3:57

answered Sep 23, 2008 at 3:50

Pascal Paradis

4,295 • 5 • 38 • 50

1

I generally prefer the "business object encapsulates data object/storage" best. However, in the short you may find high redundancy with your data objects and your business objects that may seem not worthwhile. This is especially true if you opt for an ORM as the basis of your data-access layer (DAL). But, in the long term is where the real pay off is: application life cycle. As illustrated, it isn't uncommon for "data" to come from one or more

storage subsystems (not limited to RDBMS), especially with the advent of cloud computing, and as commonly the case in distributed systems. For example, you may have some data that comes from a Restful service, another chunk or object from a RDBMS, another from an XML file, LDAP, and so on. With this realization, this implies the importance of very good encapsulation of the data access from the business. Take care what dependencies you expose (DI) through your c-tors and properties, too.

That said, an approach I've been toying with is to put the "meat" of the architecture in a business controller. Thinking of contemporary data-access more as a resource than traditional thinking, the controller then accepts in a URI or other form of metadata that can be used to know what data resources it must manage for the business objects. Then, the business objects DO NOT themselves encapsulate the data access; rather the controller does. This keeps your business objects lightweight and specific and allows your controller to provide optimization, composability, transaction ambiance, and so forth. Note that your controller would then "host" your business object collections, much like the controller piece of many ORMs do.

Additionally, also consider business rule management. If you squint hard at your UML (or the model in your head like I do :D), you will notice that your business rules model are actually another model, sometimes even persistent (if you are using a business rules engine, for example). I'd consider letting the business controller also

actually control your rules subsystem too, and let your business object reference the rules through the controller. The reason is because, inevitably, rule implementations often need to perform lookups and cross-checking, in order to determine validity. Often, it might require both hydrated business object lookups, as well as back-end database lookups. Consider detecting duplicate entities, for example, where only the "new" one is hydrated. Leaving your rules to be managed by your business controller, you can then do most anything you need without sacrificing that nice clean abstraction in your "domain model."

In pseudo-code:

```
using(MyConcreteBusinessContext ctx = new
MyConcreteBusinessContext("datares://model1?
DataSource=myserver;Catalog=mydatabase;Trusted_Conne
ruleres://someruleresource?
type=StaticRules&handler=My.Org.Business.Model.RuleM
{

User user = ctx.GetUserById("SZE543");
user.IsLogonActive = false;
ctx.Save();
}

//a business object
class User : BusinessBase {
  public User(BusinessContext ctx) : base(ctx) {}

  public bool Validate() {
    IValidator v = ctx.GetValidator(this);
    return v.Validate();
  }
}

// a validator
```

```
class UserValidator : BaseValidator, IValidator {
 User userInstance;
 public UserValidator(User user) {
  userInstance = user;
 }

 public bool Validate() {
   // actual validation code here
   return true;
 }
}
```

Share Improve this answer

Follow

answered May 22, 2010 at 0:03

Timex

121 ● 1 ● 4

0

Clients should never deal with storage objects directly. They can deal with DTO's directly, but any object that has any logic for storage that is not wrapped in your business object should not be called by the client directly.

Share Improve this answer

Follow

answered Sep 23, 2008 at 3:44

Charles Graham

24.8k ● 14 ● 47 ● 56

0

Check out CSLA.net by Rocky Lhotka.

Share Improve this answer

Follow

answered Sep 23, 2008 at 3:58

DancesWithBamboo

4,156 ● 1 ● 20 ● 20

0

Well, here I am, the co-worker Greg mentioned.

Greg described the alternatives we have been considering with great accuracy. I just want to add some additional considerations to the situation description.

Client code can be unaware about datastorage where business objects are stored, but it is possible either in case when there is only one datastorage, or there are multiple datastorages for the same business object type (users stored in local database and in external LDAP) but the client does not create these business objects. In terms of system analysis, it means that there should be no use cases in which existence of two datastorages of objects of the same type can affect use case flow.

As soon as the need in distinguishing objects created in different data storages arise, the client component must become aware about multiplicity of data storages in its universe, and it will inevitably become responsible for the decision which data storage to use on the moment of object creation (and, I think, object loading from a data storage). Business layer can pretend it is making this decisions, but the algorithm of decision making will be based on type and content of the information coming from the Client component, making the client effectively responsible for the decision.

This responsibility can be implemented in numerous ways: it can be a connection object of specific type for each data storage; it can be segregared methods to call to create new BO instances etc.

Regards,

Michael

Share Improve this answer

Follow

answered Sep 23, 2008 at 13:50

Michael

0

CLSA has been around a long time. However I like the approach that is discussed in Eric Evans book http://dddcommunity.org/

Share Improve this answer

Follow

edited Apr 16, 2011 at 9:03

Tristan

9,101 ● 7 ● 54 ● 108

answered Sep 23, 2008 at 5:09

Daniel Honig

4,418 ● 6 ● 28 ● 24