Interfaces on different logic layers

Asked 16 years, 4 months ago Modified 10 years, 9 months ago

Viewed 6k times

12

Say you have an application divided into 3-tiers: GUI, business logic, and data access. In your business logic layer you have described your business objects: getters, setters, accessors, and so on... you get the idea. The interface to the business logic layer guarantees safe usage of the business logic, so all the methods and accessors you call will validate input.

This great when you first write the UI code, because you have a neatly defined interface that you can trust.

But here comes the tricky part, when you start writing the data access layer, the interface to the business logic does not accommodate your needs. You need to have more accessors and getters to set fields which are/used to be hidden. Now you are forced to erode the interface of your business logic; now it is possible set fields from the UI layer, which the UI layer has no business setting.

Because of the changes needed for the data access layer, the interface to the business logic has eroded to the point where it is possible to even set the business logic with invalid data. Thus, the interface does not guarantee safe usage anymore.

I hope I explained the problem clearly enough. How do you prevent interface eroding, maintain information hiding and encapsulation, and yet still accommodate different interface needs among different layers?

architecture

Share

Improve this question

Follow

edited Mar 8, 2014 at 23:39

alvonellos

1,062 ● 1 ● 9 ● 28

asked Aug 12, 2008 at 21:06

user1144

This is a good question. I met the same problem. I will post my solution soon! – Darren Jan 21, 2023 at 9:14

9 Answers

Sorted by: Highest score (default)

7

If I understand the question correctly, you've created a domain model and you would like to write an object-relational mapper to map between records in your database and your domain objects. However, you're concerned about polluting your domain model with the 'plumbing' code that would be necessary to read and write to your object's fields.

Taking a step back, you essentially have two choices of where to put your data mapping code - within the domain class itself or in an external mapping class. The first option is often called the Active Record pattern and has the advantage that each object knows how to persist itself and has sufficient access to its internal structure to allow it to perform the mapping without needing to expose non-business related fields.

E.g

```
public class User
{
    private string name;
    private AccountStatus status;

    private User()
    {
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public AccountStatus Status
    {
        get { return status; }
    }

    public void Activate()
    {
        status = AccountStatus.Active;
    }

    public void Suspend()
    {
        status = AccountStatus.Suspended;
    }

    public static User GetById(int id)
    {
        User fetchedUser = new User();

        // Lots of database and error-checking
code
        // omitted for clarity
```

In this example, we have an object that represents a User with a Name and an AccountStatus. We don't want to allow the Status to be set directly, perhaps because we want to check that the change is a valid status transition, so we don't have a setter. Fortunately, the mapping code in the GetById and Save static methods have full access to the object's name and status fields.

The second option is to have a second class that is responsible for the mapping. This has the advantage of seperating out the different concerns of business logic and persistence which can allow your design to be more testable and flexible. The challenge with this method is how to expose the name and status fields to the external class. Some options are: 1. Use reflection (which has no qualms about digging deep into your object's private parts) 2. Provide specially-named, public setters (e.g. prefix them with the word 'Private') and hope no one uses them accidentally 3. If your language suports it, make the setters internal but grant your data mapper module access. E.g. use the InternalsVisibleToAttribute in .NET 2.0 onwards or friend functions in C++

For more information, I'd recommend Martin Fowler's classic book 'Patterns of Enterprise Architecture'

However, as a word of warning, before going down the path of writing your own mappers I'd strongly recommend looking at using a 3rd-party object relational mapper (ORM) tool such as nHibernate or Microsoft's Entity Framework. I've worked on four different projects where, for various reasons, we wrote our own mapper and it is very easy to waste a lot of time maintaining and extending the mapper instead of writing code that provides end user value. I've used nHibernate on one project so far and, although it has quite a steep learning curve initially, the investment you put in early on pays off considerably.

Share Improve this answer Follow

answered Aug 13, 2008 at 6:08

Wheelie
3,906 ● 2 ● 35 ● 39

Any simplified version of this answer please – Darren Jan 21, 2023 at 10:53

5

This is a classic problem - separating your domain model from your database model. There are several ways to attack it, it really depends on the size of your project in my opinion. You could use the repository pattern as others have said. If you are using .net or java you could use NHibernate or Hibernate.

What I do is use Test Driven Development so I write my UI and Model layers first and the Data layer is mocked, so the UI and model is build around domain specific objects, then later I map these object to what ever technology I'm using the the Data Layer. Is a very bad idea to let the database determine the design of your app, write the app first and think about the data later.

ps the title of the question is a little mis-leading

Share Improve this answer Follow

edited Aug 17, 2008 at 15:08

answered Aug 12, 2008 at 21:28

Dan
29.4k ● 44 ● 151 ● 209

1

@Ice^^Heat:

> What do you mean by that the data tier should not be aware of the business logic tier? How would you fill an business object with data?

The UI asks the ServiceClass in the business tier for a service, namely getting a list of objects filtered by an object with the needed parameter data.
Then the ServiceClass creates an instance of one of the repository classes in the data tier, and calls the GetList(ParameterType filters).
Then the data tier accesses the database, pulls up the

data, and maps it to the common format defined in the "domain" assembly.
The BL has no more work to do with this data, so it outputs it to the UI.

Then the UI wants to edit Item X. It sends the item (or business object) to the service in the Business Tier. The business tier validates the object, and if it is OK, it sends it to the data tier for storage.

The UI knows the service in the business tier which again knows about the data tier.

The UI is responsible for mapping the users data input to and from the objects, and the data tier is responsible for mapping the data in the db to and from the objects. The Business tier stays purely business. :)

Share Improve this answer Follow

answered Aug 12, 2008 at 21:34

Lars Mæhlum
6,102 ● 3 ● 29 ● 32

0

It could be a solution, as it would not erode the interface. I guess you could have a class like this:

```
public class BusinessObjectRecord : BusinessObject
{
}
```

Share Improve this answer Follow

answered Aug 12, 2008 at 21:18

user1144

0

I always create a separate assembly that contains:

- A lot of small Interfaces (think ICreateRepository, IReadRepository, IReadListRepsitory.. the list goes on and most of them relies heavily on generics)
- A lot of concrete Interfaces, like an IPersonRepository, that inherits from IReadRepository, you get the point..
  Anything you cannot describe with just the smaller interfaces, you put into the concrete interface.
  As long as you use the IPersonRepository to declare your object, you get a clean, consistent interface to work with. But the kicker is, you can also make a class that takes f.x. a ICreateRepository in its constructor, so the code will end up being very easy to do some really funky stuff with. There are also interfaces for the Services in the business tier here.
- At last i stick all the domain objects into the extra assembly, just to make the code base itself a bit cleaner and more loosely coupled. These objects dont have any logic, they are just a common way to describe the data for all 3+ layers.

Btw. Why would you define methods in the business logic tier to accommodate the data tier?
The data tier should have no reason to even know there is a business tier..

Share Improve this answer Follow

edited Aug 12, 2008 at 21:21

answered Aug 12, 2008 at 21:16

Lars Mæhlum
6,102 ● 3 ● 29 ● 32

0

What do you mean by that the data tier should not be aware of the business logic tier? How would you fill an business object with data?

I often do this:

```
namespace Data
{
    public class BusinessObjectDataManager
    {
         public void SaveObject(BusinessObject
object)
         {
                // Exec stored procedure
         {
    }
}
```

Share Improve this answer Follow

answered Aug 12, 2008 at 21:28

user1144

So the problem is that the business layer needs to expose more functionality to the data layer, and adding this functionality means exposing too much to the UI

0

layer? If I'm understanding your problem correctly, it sounds like you're trying to satisfy too much with a single interface, and that's just causing it to become cluttered. Why not have two interfaces into the business layer? One would be a simple, safe interface for the UI layer. The other would be a lower-level interface for the data layer.

You can apply this two-interface approach to any objects which need to be passed to both the UI and the data layers, too.

```
public class BusinessLayer : ISimpleBusiness
{}

public class Some3LayerObject :
ISimpleSome3LayerObject
{}
```

Share Improve this answer Follow

answered Aug 12, 2008 at 21:52

Derek Park
46.8k ● 16 ● 59 ● 76

0

You may want to split your interfaces into two types, namely:

- View interfaces -- which are interfaces that specify your interactions with your UI, and
- Data interfaces -- which are interfaces that will allow you to specify interactions with your data

It is possible to inherit and implement both set of interfaces such that:

```
public class BusinessObject : IView, IData
```

This way, in your data layer you only need to see the interface implementation of IData, while in your UI you only need to see the interface implementation of IView.

Another strategy you might want to use is to compose your objects in the UI or Data layers such that they are merely consumed by these layers, e.g.,

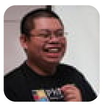
```
public class BusinessObject : DomainObject

public class ViewManager<T> where T : DomainObject

public class DataManager<T> where T : DomainObject
```

This in turn allows your business object to remain ignorant of both the UI/View layer and the data layer.

Share Improve this answer Follow

edited Aug 13, 2008 at 1:12

answered Aug 13, 2008 at 1:00



Jon Limjap

95.3k ● 15 ● 103 ● 153

0

I'm going to continue my habit of going against the grain and say that you should question why you are building all these horribly complex object layers.

I think many developers think of the database as a simple persistence layer for their objects, and are only concerned with the CRUD operations that those objects need. Too much effort is being put into the "impedence mismatch" between object and relational models. Here's an idea: stop trying.

Write stored procedures to encapsulate your data. Use results sets, DataSet, DataTable, SqlCommand (or the java/php/whatever equivalent) as needed from code to interact with the database. You don't need those objects. An excellent example is embedding a SqlDataSource into a .ASPX page.

You shouldn't try to hide your data from anyone. Developers need to understand exactly how and when they are interacting with the physical data store.

Object-relational mappers are the devil. Stop using them.

Building enterprise applications is often an exercise in managing complexity. You have to keep things as simple as possible, or you will have an absolutely un-maintainable system. If you are willing to allow some coupling (which is inherent in any application anyway), you can do away with both your business logic layer and your data access layer (replacing them with stored procedures), and you won't need any of those interfaces.

Share Improve this answer Follow

answered Aug 13, 2008 at 22:45

Eric Z Beard
38.4k ● 27 ● 101 ● 147

I suppose this might work if all you are doing is putting a screen on top of a database table. – JC. Oct 1, 2008 at 20:45

No, I'm not talking about apps that are that simple. I'm talking about writing coarse-grained procs to handle all of your business logic in complex enterprise apps, and keeping the code that calls the procs as simple as possible.
– Eric Z Beard Oct 1, 2008 at 21:08

1 I don't see how moving business logic into stored procedures helps matters. I'd prefer to keep them in a nice business layer in a language that supports good unit testing. – Ben Fulton Oct 16, 2008 at 19:50