

## Re: Having difficulty refactoring a DB application

**Source:** <http://coding.derkeiler.com/Archive/General/comp.object/2004-10/0343.html>

---

**From:** Cristiano Sadun (*cristianoTAKESadun\_at\_THIShotmailOUT.com*)

**Date:** 10/08/04

Date: Fri, 8 Oct 2004 08:58:35 +0000 (UTC)

"U-CDK\_CHARLES\Charles" <"Charles Krug"@cdksystems.com> wrote in news:ake9d.1181\$Ua.196@trndny06:

- > *List:*
- >
- > *Apologies if this is a repeat.*
- >
- > *My current project is a mess of close-coupling. I have sensible*
- > *classes that work for my "business model objects"*
- >
- > *I have a sensible database definition that more or less works for the*
- > *application. We're still refining certain aspects of it, but it's*
- > *more or less set.*
- >
- > *The problem as I see it is that I've SQL statements scattered*
- > *willy-nilly throughout the application.*
- >
- > *Seems to me an object should be responsible for its own creation,*
- > *destruction and assignment.*

This stuff is tricky – and there's not a general way of handling it. This said, something can be done: what to do is a matter of costs vs. benefits.

OO is all about minimizing cost of certain future changes. So the usual first questions are: is the application going to change a lot? Is its context of applicability (if it's a component) change a lot and require extension/customization? How much confidence you have in your predictions about future change? How much cost do you want to put up to minimize the cost of change and/or minimize the risk of having got the change predictions wrong?

Answering these questions gives you a rough idea of which properties you desire from your software, and how much effort are you (or the stakeholders) prepared to pay. Within these boundaries, you'll have some possibilities.

comp.object: Re: Having difficulty refactoring a DB application

In general, it's a good idea to encapsulate storage access somewhere – if it stays within your cost limits; for every abstraction, you need to decide if you need either a live object – with a separate lifecycle from the database data and and explicit load() and save() operations; or a more or less stateless bridge towards the data storage.

- >
- > *I've come up with two possible solutions to this.*
- >
- > *The first is to make the business objects aware of the associated*
- > *stored methods. The problem with this is that should the underlying*
- > *methods change, the classes will have to change to account for that.*

Well, that's going to happen anyway. Your system will be One :-) with its database. It's the cost of such changes that is important – from the cost to detect the change, which may depend on your organization (for example, if the guys running the db are different than the guys running the app you need to automate the change process as much as possible to avoid to rely on human memory ;), to the cost of modifying the app itself – which is, reliably locate the package/code to be changed and of course change it.

This first solution, for example, tends to make it more difficult to ensure that in the face of a db change all the code is changed accordingly – unless you have a test suite which exercises all the database-associated methods and/or exercise a strict type policy for which store procedure input/output changes are picked up by the type system (difficult to do) and so on.

- > *The second is to put in an adapter class that knows how to access the*
- > *database.*

This makes it easier to locate what is need of change, but harder to execute the change itself: outside an object, there's less immediately visible context and "meaning" that helps the programmer doing the right thing. He'll probably have to spend more time deciphering the code to understand how the db change affects the overall architecture.

A third possibility (which is, alas, more expensive in terms of coding and possibly performance) is to allocate responsibilities precisely between the objects that do depend on the storage, a middle layer, and the adapter itself – for example ensuring that the middle layer does explicitly checks on the assumed invariants of the stored procedures – basically a semantic check), while the object itself receives the results only if everything's as expected.

- >
- > *Much of the difficulty for me is reconciling SQL's idea of a "record*
- > *set" with well-formed well-behaved objects. Some queries returns more*
- > *than one record, and others return records in different form,*
- > *ill-formed or partial objects. Some of this could be massaged, but*

> *I'm not certain about proven methods for doing this.*

Try counting. Say the total of queries is 100. If 98 return structurally different results and 2 the same structure, and there's no obvious cheap way of using inheritance and/or delegation to classify the 98 in a reasonable amount of abstractions, you'd be left with 98 + 1 classes – too many. You'll probably be somewhere in the middle – so you'll have to strike a balance between generality of a given class and its semantic consistency. For example, if certain properties are valued only in certain cases, you can use explicit Undefined object as results, or you can raise checked exceptions, or unchecked exceptions (depending on what your language allows). The baseline is to ensure that in failure conditions you code fails quickly and reliably.

>

> *Any other ideas about how to structure this aside from the two I've stated?*

>

> *Any relevant patterns, either in GoF or other references that apply to this situation?*

For every subproblem, u'll have to micro–design and use any of the relevant patterns. For example, for certain objects it'll make sense to use a flyweight. For others, u'll want a factory that gets the result of the query from the adapter, sees which columns are returned and creates the best suited object as a result. Rule–based logic could be useful there, if the complexity warrants it. It's very difficult to say in general, I'm afraid.