

Architecture Dilemmas: O/R Mapping Why/When

Arnon Rotem-Gal-Oz (www.rgoarchitect.com)

[based on a series of blog post that appeared on [Dr. Dobb's Portal](#)]

The purpose of this short paper is to examine the benefits and costs of object/relational (O/R) mapping and provide some recommendation on when O/R mapping should be used.

background

IT systems need persistent storage (unless, of course, the systems are really trivial). This persistent storage usually comes in the form of a relational database, such as Oracle, Microsoft SQL Server, IBM UDB, and the like. There are several alternatives to RDBMSs; for instance, object-oriented databases (OODBs) e.g. [db4o](#), persistent object stores ([PrevaYler](#), [XML databases](#) etc.). However, at least until now, all these alternatives haven't really caught and the fact is that RDBMSs remain pretty much ubiquitous.

Thus you are faced with the task of getting data from the DB into your business logic. Unfortunately the object model and the relational model are somewhat at odds (what Scott Ambler calls "[Impedance Mismatch](#)") which means that a properly designed object model will not have a direct automatic map between objects and tables.

The result of that mismatch means that you need to get the data by using one of the following approaches:

Direct data access; not highly recommended. Retrieve/Save by using SQL calls(either dynamic SQL or a stored procedure) whenever needed by the business logic. The reason this is not recommended is that there is no separation between.

Data Access Layer (DAL). Have an object (per table) that knows how the tables are constructed. The DAL objects retrieves the data. Sometimes this is combined with Data Transfer Objects (objects with only setters/getters to change the data) to transport data between tiers.

O/R Mapping. Have a layer that is responsible to bridge the gap between the "proper" object model and the "proper" database model.

ActiveRecord. A class that has the same structure of a table fields. A class instance represents a row in the table and static methods affect the whole table. The class

abstracts the SQL needed to actually get to the DB. ActiveRecord is a simplified form of O/R Mapping.

There are several variants for each approach and mixes of the approaches; for example, the [ActiveRecord implementation in the castle project](#) which builds on Nhibernate, an O/R mapper. Many implementations (for the various methods) use code generation (there are hundreds of generators you can find many of them ordered by platform at <http://www.codegeneration.net>). There are implementations that use stored procedures and there are ones that rely on dynamic SQL. Plus there are few middle-ground approaches like iBATis.

The case for O/R Mapping

The premise of O/R mapping is that you have a focused investment in creating the mapping and that you are free from worrying about the data structure.

This leads to quite a few benefits for using O/R mapping:

- Clean OO design. Hiding the relational model specifics lets the object model be more cleanly analyzed and applied.

- Productivity. Simpler code as the object model is free from persistence constraints. Developers can navigate object hierarchies, etc.

- Separation of concerns and specialization. Let the DB people worry about DB structure and the Object people worry about their OO models.

- Time savings. The O/R mapping layer saves you from writing the code to persist and retrieve objects. O/R mapping tool vendors claim 20-30% reduction in the code that needs to be written. Writing less code also means less testing.

O/R Mapping Costs

There benefits described above are very compelling reasons to use O/R mapping. However we need to remember that nothing comes without a cost.

One problem with O/R mappers is that they usually commit the "Needless Repetition" [deadly sin](#) (a.k.a. DRY—"Don't Repeat Yourself"). The table structure as well as their relations are stored both in the DB and in the mapping files used by the O/R mapper. This causes a maintenance problem as you need to ensure that both versions of the meta-data don't get

out of sync. Rails (as in Ruby on Rails) partly solves this problem by relying on naming conventions. (Note: You can also override the default mapping.) In other environments, you can use code generation for the mapping file to get a similar effect. Both of these approaches result in limited mapping--which means you are likely to switch to manual mapping so you can benefit from the DB capabilities.

Which brings us to another problem with O/R mappers. Writing these mapping files is a daunting task (unless we are talking about trivial mappings) not to mention that O/R mappers tend to have favorite mappings (e.g., class = table or class hierarchy = table etc.) and trying other mapping are more complicated (if at all possible). This needs to be updated every time you change the DB.

One approach I've been taking recently is to isolate the DB structure from the DB side as well; that is, to add database views that expose the DB to the O/R mapper in simple-to-map structure. which means the mapping is simple and straightforward and the database is free to evolve and adapt as needed. For example, in one project we have set up "self-recording" where each update on the view resulting in an insert on the underlying table(s), while the view always reflects the latest version. Other views or queries can be used to go back in time to an earlier snapshot of the data. The downside of this approach is, of course, the double mapping layer (but at least O/R mapping is simple and relatively stable) hasn't been tested for high-data access load scenarios (as with the projects where it is used don't have such characteristics)

Then there are the queries, which is where a lot of O/R mappers stumble. You either get limited query capabilities or you get performance problems on complicated queries which means that you may need to resort to writing the queries manually (if your O/R mapper supports this)--which is what you were trying to avoid in the first place, not to mention that if your team is already familiar with DB techniques (SQL etc.) There's also the added overhead of learning that new query language. If you do feel comfortable with SQL and you want to take the middle ground you can use [iBATis](#) which lets you externalize the SQL into XML.

Note that you probably want to look at O/R mappers that support features like caches, lazy initialization , batch modes etc. to help avoid the performance problems mentioned above.

Conclusion and recommendations

O/R mapping is not a panacea, as was shown in the previous post. Using O/R mapping incurs several costs (which are sometimes hidden at first glance). Nevertheless, using O/R mapping provides a good balance between the need to bridge the gap between an OO model and a relational one versus the time and effort needed to provide that bridge. O/R mapping is especially useful if you are also following [Domain Driven Design](#) principles which support a rich and meaningful (domain) object model.

Several years ago (good) O/R mappers were hard to find. The first real O/R mapper I used was TopLink (now [Oracle Toplink](#))--indeed the Java world seems to be leading the adoption of O/R mapping into mainstream programming. Now there's [Hibernate](#) (which is rather popular), JDO (I think the first O/R mapping standard), and now the [Java Persistence API](#) as part of EJB 3.0.

On the .Net side there's a wide range of solutions, starting with NHibernate (the .Net version of Hibernate) and many commercial and open source solutions. Microsoft is following this trend and will be introducing two new O/R mapping frameworks: [LinQ for SQL](#) (formerly known as "DLinq") and LinQ [for Entities](#) (built above ADO.NET entity framework).

When a solution has mostly with data-entry screens or simple CRUD operations, a viable option is to use the [ActiveRecord pattern](#). ActiveRecord is basically a simplistic O/R mapping (actually it can be considered as an R/O mapping as it is the table row that is the mapped to an object). While ActiveRecord provides an [anemic domain model](#) it can still be useful in the scenario mentioned above. The added benefit of ActiveRecord mapping is that it is simple to generate this kind of mapping automatically (e.g. what Rails or MonoRail do).

DAL (Data Access Layer) or direct data access are a good options when the solution is very data centric and/or database intensive. A classic example for this would be a reporting application. iBAT is a variant on the DAL theme (I consider it an XML-based DAL). The good news is that the SQL is externalized from the code and can be tweaked and updated independently. The downside include lacking documentation as a lot of mapping files (mapping per query/SQL statement).

Lastly, on smaller projects which are not very data intensive, you can also consider using an object-oriented database (such as Versant FastObjects, Objectivity, or [db4O](#)). While I

wouldn't use them to create the next version of the NYSE data center, they can make life very easy where the data requirements are modest.

There are many ways to get to the data from the object model. Each approach has its place and sometimes it is worthwhile using more than one in a project. Whatever approach you take, I believe it is a good practice to consider utilizing the [Hexagonal architecture](#) principle and keep the objects clean from the data access code (POJO/POCO).

To summarize this paper O/R mapping is a good viable option for bridging the gap between the OO world and the relational one. However, you need to be aware of the costs of using such a tool and examine the context of use to determine the best data handling technique.