

# Re: Relational-to-OOP Tax

---

*Source:* <http://coding.derkeiler.com/Archive/General/comp.object/2007-03/msg00023.html>

---

- *From:* "topmind" <[topmind@xxxxxxxxxxxxxxxxxxxx](mailto:topmind@xxxxxxxxxxxxxxxxxxxx)>
  - *Date:* 1 Mar 2007 09:55:18 -0800
- 

frebe wrote:

With procedural, you don't have classes as middle-men between the app code and the schema. The schemas *\*are\** the "classes". The "noun model" is in the database, not classes. With OO, you have the class noun structure/model and the database noun structure/model which are fairly similar (we are assuming existence of RDBMS), but different enough that you spend time translating between the two. If you embrace the DB, you don't have this verbose, code-wasting middle man.

In the procedural languages I've seen I still have to create structures of some kind to work with the data in the GUI.

Many do, but it is not necessary.

Can you post a link to code that edits a record in a GUI that doesn't use structures of any kind in a procedural language?

<http://www.oscommerce.com/solutions/downloads>

PHP doesn't have the construct SELECT INTO, so arrays has to be used to represent one row in a query result. osCommerce also use classes as data structures in some scenarios, but mostly not.

I would like to add some detail to this, if you don't mind. Procedural modules that use query results often have a commonly-used structure: the "result set". It is basically a single temporary or virtual table.

## Re: Relational-to-OOP Tax

It differs from OO structures in that it is "FLAT". It is "normalized" for that particular task and only that task most of the time. There are no one-to-many or many-to-many relationships in it. OO structures often require "pointer hopping" to deal with nested or linked objects that point to other objects. The result set is thus a simpler thing.

For example, an OO program may create a "paystub" structure by having paystub object(s) with lineItem (paystub lines) objects contained within each paystub object. This is called "composition" if I am not mistaken.

However, the procedural version will usually join the payStub record(s) with the lineItem records for the particular task at hand. The result set is thus "flat". One is not dealing with a one-to-many structure, and this usually simplifies the software. One does \*not\* build a domain "noun model" in RAM. It's left to the DB for that job. The result set is a local temporary abstraction that only has to fit the task at hand. Thus, one can focus on the task at hand rather than dealing with complex data structures. However, OO design does encourage a "noun model in RAM". Wrapping them in set/gets or iterator methods does not change the fact that you have to deal with complex structures that wouldn't likely exist in the procedural version. Flat is where it's at.

OO'ers often talk about "separation of concerns". Well, this flattening is an application of that. In the task we don't really worry much about the complexity of the domain noun model. We created a simple temporary structure (result set) that is used for the task at hand. There are no data structures in the task (module) that model the composition (one-to-many, etc.) of the real world. We are shutting out some of the complexities of the real world to focus on the task. Queries create that abstraction for us.

A little caveat note: For a large result set this is sometimes memory-hogging since the payStub item would be repeated, but that is mostly an issue for "batch" processing, which may require a bit different approach, such as processing one or few pay stubs at a time or use cursors. But, most interactive systems process one or a few items at a time such that such duplication is not a problem. Some relational purists frown on cursors, but that is another topic. Further, a query itself is often able to do much of the processing in a declarative way, such as an UPDATE statement so that we don't have to loop through individual result set records. A well-designed schema often facilitates this so that we can farm off much of the processing to relational algebra to avoid loops and IF statements. The need for lots of loops and IF's is sometimes a sign of bad schema design. Thinking and designing declaratively and thinking in sets instead of trees or graphs takes a bit of skill, but can pay off handsomely.

## Re: Relational-to-OOP Tax

And you are less likely to have to build and manage complex structures in RAM. (I used to sometimes use tables for local, temporary stuff also, before they removed such features due to OO hype.)

Are you thinking that you just use a database driver to connect to the database and use the database itself to store the entire state of the relational model without any structures at all and just always query the database to get the current state?

This is the basic idea in the relational model. The application asks for data using relational algebra, when it needs it.

Are you thinking business logic is implemented as triggers on tables?

Business logic may be implemented in tables (base relations), constraints, views (derived relations), triggers, stored procedures or application code – in that preferred order.

All edits on a screen would just send updates as SQL to the database immediately. The GUI would commit when the user presses a save button, or rollback if the user presses cancel?

The database doesn't need to be remote. The host programming language may have relational algebra built-in and an light-weight embedded database engine.

I would seriously question the application style because it requires constant database queries which will negatively impact performance.

Only if the application and database are different processes.

/Fredrik

-T-

.