Wednesday, December 15, 2010

Historical Perspective of ORM and Alternatives

A couple of years ago I broke my basic rule of sticking to practical how-to and general programming philosophy and wrote Why I Do Not Use ORM. It sure got a lot of hits, and is read every day by people searching such things as "orm bad" or "why use orm". But I have never been satisfied with that post, and so I decided to take another stab from another angle. There are legitimate problems that led to ORM, and those problems need to be looked at even if we cannot quite agree on what they are or if ORM is the answer.

UPDATE: In response to comments below and on reddit.com, I have a new post that gives a detailed analysis of an algorithm implemented as a sproc, in app code with embedded SQL, and in ORM.

Here then, is one man's short history of commercial database application programming, from long before the ORM system, right up to the present.

This blog has two tables of contents, the Topical Table of Contents and the list of Database Skills.

The Way Back Machine

When I began my career the world was a different place. No Web, no Java, and Object Orientation had not yet entered the mainstream. My first application was written on a timeshare system (a microVAX) and writing LAN applications made me a good living for awhile before I graduated to client/server.

In those days there were three things a programmer (We were not "software engineers" yet, just programmers) had to know. Every programmer I knew wanted to master all of these skills. They were:

- How to design a database schema for correctness and efficiency.
- How to code an application that could process data from the database, correctly and efficiently.
- How to make a good UI, which came down to hotkeys and stuffing the screen with as much info as possible.

In this essay we are going to look at those first two.

My own experience may be somewhat peculiar in that I have never worked on a team where the programmers were separated from the database. *(OK, one exception, in my current assignment there is an iron curtain between the two, but happily it is not my problem from where I sit).* Coders made tables, and "tablers" wrote code. So this focus on being a good developer by developing both skills may be rare, enjoyed by those who have the same ecumenical background that I enjoyed.

Some Changes That Did Not Matter

Things changed rapidly, but most of those changes did not really affect application development.

When Windows 95 came out, being "almost as good as a Mac", we recoded our DOS apps into Windows apps without too much trouble and life went on as before.

Laser printers replaced dot-matrix for most office use, CPUs kept getting faster (and Windows kept getting slower), each year there were more colors on the screen, disks got bigger and RAM got cheaper.

Only the internet and the new *stateless programming* required any real adjustment, but it was easy for a database guy because good practice had always been to keep your transactions as short as possible. The stateless thing just kind of tuned that to a sharp edge.

Finally, with the internet, the RDBMS finally lost its place as sole king of the datastore realm, but those new datastores will have to wait for another day, lest we get bogged down.

Enter Object Orientation

Arguably nothing changed programming more than Object Orientation. Certainly not Windows 95, faster graphics or any of those other Moore's Law consequences. I would go so far as to say that even the explosion of the web just produced more programming, and of different kinds of apps, and even that did not come close to the impact of Object Orientation. Disagree if you like, but as it came in, it was new, it was strange, it was beautiful, and we were in love.

Now here is something you may not believe. The biggest question for those of us already successfully developing large applications was: What is it good for? What does it give me that I do not already have? Sure its beautiful, but *what does it do?*

User interfaces were for me the easiest first place to see the benefits. When the widgets became classes and objects, and we empolyed encapsulation, inheritance and composition, the world changed and I don't know anybody who ever looked back.

OOP, Data, and Data Structures

But in the matter of processing data, things were not so clear cut. The biggest reason may have been that all languages back then had *specialized data structures* that were highly tuned to handling relational data. These worked so well that nobody at first envisioned anything like ActiveRecord because we just did not need it.

With these structures you could write applications that ran processes involving dozens of tables, lasting hours, and never wonder, "Gosh, how do I map this data to my language of choice?" You chose the language you were using *precisely because it knew how to handle data!*

I would like to throw in just one example to show how OOP was not relevant to getting work done back then. I was once asked to optimize something called "ERP Allocation" that ran once/day, but was taking 26 hours at the largest customer site, obviously a big problem. It turned out there was a call to the database inside of a tightly nested loop, and when I moved the query outside of the loop the results were dramatic. The programmers got the idea and they took over from there. The main point of course is that it was all about how to efficiently use a database. The language was OOP, and the code was in a class, but that had nothing to do with the problem or the solution. Going further, coding a process so data intensive as this one using ActiveRecord was prima facia absurd to anybody who knew about data and code.

Java and the Languages of The Internet

But the web had another impact that was far more important than just switching to stateless programming. This was the introduction of an entirely new family of languages that took over the application space, listed here in no particular order: Perl, PHP, Python, Ruby, and the king of them all: Java.

All of these languages have one thing in common that positively jumps out at a veteran: *a complete lack of data structures specialized for handling relational data.* So as these languages exploded in popularity with their dismal offerings in data handling, the need to provide something better in that area became rapidly clear.

Java has a special role to play because it was pure OOP from the ground up. Even the whitespace is an object! The impact of Java is very important here because Object Orientation was now the One True Faith, and languages with a more flexible approach were gradually demoted to mere 'scripting' languages. *(Of course proponents will quickly point out that 1/12 of the world's population is now using a single application written in one of those 'scripting' languages).*

So the explosion of languages without decent data handling abilities, coupled with a rise in OOP-uber-alles thinking led us quite naturally to:

The First Premise of ORM: The Design Mismatch

The first premise of ORM is that there is a design mismatch between OOP and Relational, which must resolved before any meaningful work can be done.

This view is easy to sympathize with, even if you disagree, when you consider the points raised in the above sections, that the languages in play lack any real specialized data structures, and that a certain exclusive truthiness to OOP has arisen that is blind to entire classes of solutions.

So we must grant the ORM crowd their first premise, in modified form. It is not that there is a design mismatch, it is that there is something missing, something that was in older systems that is just not there in the newer languages. Granting that this missing feature is an actual mismatch requires a belief in the Exclusive Truth of OOP, which I do not grant. OOP is like the computer itself, of which Commander Spock said, "Computers make excellent servants, but I have no wish to be servant to a computer."

But anyway, getting back to the story, the race was on to replace what had been lost, and to do it in an OOPy way.

The Second Premise of ORM: Persistence

Fast forward and we soon have an entire family of tools known as Object-Relational-Mappers, or ORM. With them came an old idea: persistence.

The idea has always been around that databases exist to *persist* the work of the programmer. I thought that myself when I was, oh, about 25 or so. I learned fast that my view of reality was, *cough*, lacking, and that in fact there are two things that are truly real for a developer:

- The users, who create the paycheck, and
- The data, which those users seemed to think was supposed to be correct 100% of the

time.

From this perspective, the application code suddenly becomes a go-between, the necessary appliance that gets data from the db to the user (who creates the paycheck), and takes instructions back from the user and puts them in the database (correctly, thank you, and don't make the user wait). No matter how beautiful the code was, the user would only ever see the screen (or page nowadays) and you only heard about it if it was wrong. Nobody cares about my code, nobody cares about yours.

However, in the ORM world the idea of a database as the *persistence* layer now sits on a throne reserved for axiomatic truth. Those who disagree with me on this may say that I have the mistaken perspective of an outsider, to which I could say only that it is this very idea that keeps me an outsider.

But we should not paint the world with a broad brush. Chris Wong writes an excellent blog where he occassionally details how to respect the database while using Hibernate, in this post and this post.

An Alternative World View

There are plenty of alternatives to ORM, but I would contend that they begin with a different world view. Good business recognizes the infinite value of the users as the generators of the Almighty Paycheck, and the database as the permanent record of a job well done.

This worldview forces us into a humble position with respect to our own application code, which is that it is little more than a waiter, carrying orders to the kitchen and food back to the patrons. When we see it this way, the goal becomes to write code that can efficiently get data back and forth. A small handful of library routines can trap SQL injection, validate types, and ship data off to the database. Another set can generate HTML, or, can simply pass JSON data up to those nifty browser client libraries like ExtJS (now "Sencha" for some reason).

This covers a huge amount of what an application does, if you do not have much in the way of business logic.

But How Do You Handle Business Logic?

I have an entire essay on this about half-written, but in short, it comes down to understanding what business logic really is. **Update: This post is now available**

The tables themselves are the bottom layer of business logic. The table design itself implements the foundation for all of the business rules. This is why it is so important to get it right. The tables are organized using normalization to have a place for everything and everything in its place, and after that the application code mostly writes itself.

The application code then falls into two areas: value-add and no value-add. There is no value-add when the application simply ships data off to the user or executes a user request to update the database. Those kinds of things should be handled with the lightest possible library that gets the job done.

But the value-add stuff is different, where a user's request requires lookups, possibly computations and so forth. The problem here is that a naive analysis of requirements (particulary the transliteration error (Scroll down to "The Customer Does Not Design Tables) will tend to generate many cases of perceived need for value-add where a simpler design can reduce these cases to no value-add. But even when the database has been simplified to pristine perfection, there are jobs that require loops, multiple passes and so forth, which must be made idempotent and robust, which will always require some extra coding. But if you know what you are doing, these always turn out to be the ERP Allocation example given above: they are a lot more about the data than the classes.

Another huge factor is where you come down on the normalization debate, particularly on the inclusion of derived values. If you keep derived values out of the database, which is technically correct from a limited perspective, then suddenly the value-add code is much more important because *without it your data is incomplete*. If you elect to put derived values into your database than value-add code is only required *when writing to the database*, so huge abstractions meant to handle any read/write situation are unnecessary. (And of course, it is extremely important to Keep denormalized values correct).

And the Rest of It

This essay hardly covers the entirety of making code and data work together. You still have to synchronize schema changes to code, and I still think a data dictionary is the best D-R-Y way to do that.

I hope this essay shows something of why many programmers are so down on ORM, but much more importantly that there are coherent philosophies out there that begin with a different worldview and deliver what we were all doing before ORM and what we will all still be doing after ORM: delivering data back and forth between user and database.

Posted by KenDowns at
Email This BlogThis! Share to Twitter Share to Facebook Share to Google Buzz

Reactions:

21 comments:

Bram said...

It seems like the link to 'Why I Do Not Use ORM' in the first paragraph is pointing to the current post, rather than your old post.

(I will continue reading now, interesting stuff!)

December 16, 2010 4:19 AM

KenDowns said...

Bram, thanks, the link is fixed.

December 16, 2010 8:13 AM

Joe Yates said...

Ken, you say that "all languages back then had specialized data structures that were highly tuned to handling relational data".

Could you list some examples?

December 16, 2010 9:52 AM

KenDowns said...

Joe: I was thinking specifically of Visual Foxpro with its "local cursor", which went back to its roots as a desktop database.

Though I never used Delphi, I met a few users who were as devoted as we foxers were, and they said the same thing.

If you want a real enthusiastic sermon on the promised land, talk to any AS/400 guy, they will talk your ear off (and I think with some good reason) about how beautifully tabular data handling is built into everything.

Even COBOL (gads) had its basic RECORD definitions (with suitable disclaimers that it was not using a strict relational database).

December 16, 2010 10:42 AM

Dean said...

The central idea of object orientation is that objects have both information and behavior. It assumes that information and behavior are linked.

Databases come from a different paradigm, where data and instructions are separate things. It's possible to apply a given operation or transformation to a wide variety of data structures and types.

So there is a mismatch between the two approaches, but I wouldn't call it a *mapping* problem. It's a mental model problem.

December 16, 2010 11:07 AM

Harry Simons said...

Ken - thanx first off for your post, for sharing your insights.

1. > all languages back then had specialized data structures that were highly tuned to handling relational data.
For those who're from a later era, would you care to elaborate what those data structures were/are?

2. Can't the ORM impedance mismatch problem be (somewhat) solved by having the O-layer of the application deal with SETS of objects (which would correspond to tables) instead of one object at a time, or instead of a graph of objects that must be navigated through manually? You use a portable SQL via a JDBC-like API in your code, and the business logic gets too complex, you code that part in an OO-way. I have never really done enterprise programming, but in my head it feels as if one could do this.

3. > I have an entire essay on this about half-written
Will look forward to it! Would greatly appreciate if you could give some sneak-peak, either here or privately. I hope your essay will address the SQL portability problem, or won't assume the use of a big-iron DBMSes such as Oracle.

December 16, 2010 11:54 AM

JulesLt said...

It's interesting to see Java described as a pure-OO language. I suppose in some senses it is, in that all code has to live on a class, but in other ways it's not (i.e. it has both C-style primitive numeric types and BigDecimal objects, and it lacks the dynamic flexibility of the more Smalltalk inspired languages like Ruby and Python).

Anyway, that's only slightly relevant. The one thing that struck me with learning Java was how absolutely awful it's collection handling was, being barely more than an array of pointers that you had to cast back into the relevant object type (until generics came along, and they are still controversial).

In more dynamic languages at least the objects in the collection remember what type they were.

Another revelation came from reading a paper on Higher Order Messaging (and considering LINQ in C#).

http://www.metaobject.com/papers/Higher_Order_Messaging_OOPSLA_2005.pdf

Basically, there seems a common pattern in the development of many OO languages around improving the syntax for performing the same operation across collections / sets of data.

Getting away from writing a loop to process all items in an array, towards a syntax where you can simply say 'do this action to all items in the collection that match these criteria'.

And to me that starts to look increasingly familiar.

I wonder if a lot of the mismatch starts with that - that a 'database' oriented developer starts from the point of view of the collection / set? Whereas OO is beginning to end up there

December 16, 2010 12:55 PM

KenDowns said...

Jules: nice comment. Groovy immediately comes to mind. Add fluent programming and you get some powerful stuff.

My own first real-world use of powerful set-oriented code outside of a database was actually jQuery, which IMHO suddenly made Javascript + DOM actually usabel.

December 16, 2010 1:15 PM

Marco Aurelio said...

Hi. When go to a D.B. related job interview, Its common that they ask me "How do you handle O.R.M. impedance mismatch". And I handle I don't have that problem beacause "E-R" is ALREADY OBJECT ORIENTED.

I have met developers that have "impedance mismatch", and the ones that not. I observe that the ones that have it, learn to use relational databases as "Flat Files", while does who doesn't, learn about the E-R Model...

December 16, 2010 2:43 PM

JulesLt said...

Forgot about jQuery, but yes, that's another one.

Anyone would think that it was a common programming language problem that needed addressing through the development of some kind of query language . . .

December 17, 2010 6:17 AM

Adam said...

Ken,

Great article! The one thing I'd like to point out is this:

Your two "truly real" ideas are missing one: maintainability. You're absolutely right that nobody (aside from other developers) actually cares about the content or structure of your code, but they very likely *do* care about how quickly you can change the application to make it do what they now want it to do, especially when they wanted it yesterday. Disregarding the idea of maintainability is what leads developers to do the most lightweight--and usually easiest--thing, such as embedding hard-coded SQL statements directly within business logic (or, heaven forbid, *user interface*) code.

But this approach is the posterchild for the extra work involved in abstraction, or even ORM; once you start fiddling around either with the underlying logic or the storage schema, you now have 37 different areas to address. Thank goodness you remembered 36 of them before you made that build on Friday afternoon before you left for vacation!

While it's important not to become too self-important and overstate the relevance of how your code is organized, it's also just as important to make sure that the extra work *has* gone into making it maintainable.

December 17, 2010 9:07 AM

Anthony Shaughnessy said...

It's long since seemed to me that the mismatch between OO and relational is that much of the work a typical business application does is manipulation of the relationships between entities rather than simply the entities themselves. In java we tend to model the entities (e.g. as POJOs) but get problems when we try and use ORM because the work we need to do is manipulating relationships. This can lead either to extremely inefficient code that queries far too much data or to bypassing the nice ORM framework in order to do the work in the database. Business applications are very well modelled by the relational model and the RDBMS was designed to support the business application. I think relegating the database to a second class citizen is a mistake.
Anthony

December 20, 2010 6:09 AM

Justis said...

If you haven't seen yet, from the other side of the spectrum.

http://bergie.iki.fi/blog/stop_using_sql-then/

which links to

http://bergie.iki.fi/blog/why_you_should_use_a_content_repository_for_your_application/

From the 1st link:

"Sadly, web development often also involves SQL, and that is your granddad's programming language. SQL, and the stored procedure languages you are most likely to encounter, have deliberately not evolved much since the 70s and 80s."

Hmmm, sql is useless and Oracle has put a "no further enhancements" code freeze on pl/sql for the last 20 years.

December 20, 2010 8:07 PM

KenDowns said...

Justis:

1) Sometimes they get it right the first time

2) Check out Common Table Expressions, window functions, others.

December 20, 2010 10:55 PM

KenDowns said...

Adam: Your point on maintainability is well taken. I'll have to consider working that into a separate post.

December 20, 2010 10:56 PM

Lukas Eder said...

Thanks for this great article, Ken! Somehow it never occurred to me to think about the ways programmers handled data back in the days before ORM. Your point of view really shows why people like me feel slightly awkward today when it comes to "mapping", "fine-tuning", and "persisting" data between the different worlds. I always thought that RDBMS (including vendor-specific extensions, such as stored procedures, etc) are the perfect way to model and handle data. On the other hand, all the "standard" solutions provided by the J2EE/JSR folks seemed to somehow ignore my love for RDMBS, obscuring more and more the data's heritage. To name a few: EJB1/2, Hibernate, JPA, Criteria Query...

I wanted to go back to SQL, to the relational and support all the features that modern RDBMS offer today, without giving up on what Java can do. That is why I created jOOQ, a new database abstraction tool handling all the drawbacks of interfacing with an RDBMS from Java/JDBC while giving full credit to SQL (via its DSL, similar to Linq) and proprietary features (such as UDTs, stored procedures, etc).

I have written an article on that topic: http://java.dzone.com/announcements/simple-and-intuitive-approach. Any feedback is welcome!

Cheers
Lukas

December 30, 2010 4:52 PM

KenDowns said...

Lukas:

jOOQ looks fascinating. I will be interested in how it develops.

I am working on a db building tool http://http://code.google.com/p/triangulum-db/ (Triangulum), and one of the goals is to write out files in various languages that might feed into something like jOOQ.

December 30, 2010 5:56 PM

Lukas Eder said...

Ken, that looks very interesting as well! One of the major problems I'm facing with jOOQ is the various databases' understanding of similar concepts, such as primary key, foreign key, relations etc.

Unfortunately, the JDBC's standard DatabaseMetaData object is not reliable, as implementations are often incomplete. Instead I have to browse the meta-schema myself, which is a major source of errors. jOOQ might be able to build on top of a tool like triangulum-db (or Hibernate), to discover the meta-schema...

Cheers and a happy New Year!

January 2, 2011 6:39 AM

Kyle Hailey said...

>>The main point of course is that
>>it was all about how to efficiently
>>use a database. The language was
>>OOP, and the code was in a class,
>>but that had nothing to do with the
>>problem or the solution.
I think part of the problem trying to "protect" programmers, OO or not, from the database, like the database is an after thought. The example you gave happens over and over again. Happens with procedural programmers as well as OO though OO seems to exacerbate the disconnect. The common analogy is "if I was going to the grocery store with long list of items, would I buy one thing and go home, then go back and buy one other thing, then come home?" of course not, I go to the grocery and buy everything on my list. Seems obvious, but alas but coders make the same mistake over and over again.
As, I think Stephane Faroult put it, databases operate on row sets, not objects/variables. I don't know of any ORM or any other layer way to fix this. It's how people think and what people are taught that has to be addressed.
Of course, I do think about automated ways of addressing the situation. I think of ways of finding the existing inefficient patterns and flagging them.
- Kyle
http://dboptimizer.com

January 6, 2011 6:07 PM

Shawn Owston said...

one factor not mentioned is performance based on what "tier" the business logic is being executed. Stored Procedures to execute business logic and execution on the data tier is much more efficient in comparison to loading into entities at an application level, performing the business logic "work" and then persisting the changes back to the datasource.

I categorically refute the statement that SQL has not evolved...especially as an Oracle Database user where PL/SQL is extremely powerful, and much more efficient than using any object oriented programming language to perform the same tasks.

I would argue that ORM's are extremely "inflexible" when the data model schema is constantly evolving throughout time and requires too much "code level" changes to update to a schema change.

This fact alone makes ORM's cumbersome and difficult to work with in comparison to building out the business logic in the db itself.

in summary:

1. ORM's are slow...all of them
2. ORM's are not designed to adapt to "change" very efficiently

June 3, 2011 5:01 PM