

# **The Key to Minimizing Software Total Cost of Ownership.**

Tom Johnston

August, 2007.

## ***Metadata and Late Binding.***

In traditional software applications, requirements are hardcoded in both the codebase and the database, as procedural descriptions expressed in a programming language and as structural declarations expressed in a data definition language. But to hardcode requirements is to early-bind them. Such requirements are bound at compile-time when procedural source code is translated into machine-executable instructions, and when schema-defining source code is translated into entries in the catalog of the DBMS. When those requirements change, both code and schemas must be rewritten and recompiled. These changes can be very expensive to implement; *over the lifetime of an important system, their costs can easily exceed the combined costs of initial release and subsequent release new development.*

As for code, once requirements are expressed in code, they are difficult to change. This difficulty stems from two sources. On the one hand, the procedural definition of an algorithm can be quite complex because of the loops, branches and sequences of data transformation statements which intermingle to specify both what to do, and which rows of which tables to do it to.

As for schemas, once requirements are expressed in physical database schemas, and those schemas are populated with real data, a very heavy die is cast. The cost of changing a logical data model is minimal. The cost of changing a populated physical data model is usually significant, and can sometimes be prohibitively high. It includes the cost of recasting data, and the ripple effect costs of changing all the code and queries which reference the schemas that are being changed. If those changes cannot be carried out as a single atomic transaction, albeit one which might take days or weeks to

complete, then an additional cost is the cost of managing an inconsistent set of schemas during the transition period.

If requirements are late-bound, on the other hand, then neither code nor schemas need to change when those requirements change. Such requirements are bound to an event of their utilization at run-time, not at compile-time. This late-binding is accomplished by means of metadata and metadata-driven code. First, the requirements are expressed as metadata, i.e. as rows in metadata tables. Second, code reads this metadata, each time it executes.

One kind of code reads one kind of metadata to manage the translation between the standard relational description of the database which the programmer and end-user sees, and the internal form in which data is actually stored. This code late-binds the expression of data structures to the physical database.

Another kind of code reads another kind of metadata to manage the processes which create, transform and destroy instances of the standard data model, i.e. rows of tables. Just as the first kind of code late-binds the expression of data structures to the database, this kind of code late-binds the expression of algorithms to the codebase. This kind of code has its roots in what was known, several decades ago, as "table-driven code". Let us call the first kind of code "metadata-driven structure translation code", and the second kind "metadata-driven process definition code".

We shall deal with metadata-driven process definition code later. As for metadata-driven structure translation code, on the one hand it materializes rows of the tables of a standard data model from the physical database, based on requests to retrieve data from the database. On the other hand, it "deconstructs" rows of the tables of a standard data model which are created by the programmer or end-user, and presented for insertion or update in the database. It then stores the deconstructed data in the physical database, in a form which may bear very little resemblance to the rows of tables seen and manipulated by end-users and programmers.

In performing these translations, this metadata-driven structure-translation code completely isolates the structure of the data as stored from the structure of the data as seen. In doing so, it completely eliminates all early binding of requirements to physical data structures. Consequently, no matter what changes to a logical relational model must be implemented, no changes to the physical data model will be required, and all costs associated with such changes are automatically avoided.

Taken to an extreme, this should mean that one physical data model should suffice to express any logical data model. And in fact, it does mean that.

*{I wrote the above several years ago. In fact, it's the introduction to one of my series of articles published in datawarehouse.com. On coming across it this evening, I realized that it expresses the motivating force behind nearly all my publications. So I've tacked on the following material to explain how it has motivated me.}*

### ***Data Modeling and Software Development.***

Data modeling is at the heart of how I think about software applications. And at a very early point in my data modeling career, I observed that even among fully normalized logical data models, there was a wide range of them that would satisfy any given requirement. And just as clearly, some were much better than others. But aside from some kind of "elegance" which the best models seemed to have, I couldn't say what made the good ones good, the bad ones bad, and the ugly ones really bad.

Even today, managers and project leaders will often tell the modeler that they want a "third normal form" logical data model. If that's all they want, then data modeling should be paid on a par with a junior programming position, because there's so little to it. The rules to put a collection of data into first, second and third normal form can be taught in an afternoon. And there's nothing particularly difficult about them, or about applying them to data specified in a requirements document (when modeling from scratch) or found in a collection of files or non-normalized database tables (when reverse-engineering from existing data).

This line of thought has three branches.

### ***Lousy Data Models.***

The first branch was the realization that there were lots of lousy data designs around, and yet the systems that used them worked! Poorly-designed airplanes never get off the ground. But poorly designed data models are everywhere. I asked myself how this was possible.

The answer, of course, is that we somehow compensate for the lousy designs. Write complex-enough code, and you can maintain any database, no matter how badly it is designed. Write complex-enough queries, and you can usually get back the same result sets that simpler queries against better designed databases would return. This puts the cost of compensating for poorly designed data on software developers. And make no mistake: the cost is entirely unnecessary.

Those unnecessary costs are bad enough. But there is an even worse compensation mechanism. It comes into play when the personnel doing the compensating are not the software developers, but rather the end-user business community! The hallmark of this category of unnecessary cost is the database for which experienced end-users have a back pocket full of "tips and tricks" for how to write queries that Wow! more novice users.

I won't continue this discussion beyond this point, for now at least. But I did publish an article at datawarehouse.com a few years ago, with the title "Bad Data Models and How They Work".

### ***Techniques for Using Metadata for Late Binding Purposes.***

Several dozen of my articles have been concerned with this topic. I'll write more on it later.

### ***Ontologies and Taxonomies.***

At last! The philosopher can come out of the closet!

I am a philosopher by training, with a doctorate earned quite a few years ago. Metaphysics is the central discipline in philosophy (*not* the kind of stuff you find next to the astrology books at the bookstore!) Ontology and epistemology are the two branches of metaphysics.

What computer scientists have done is combine classical ontology with mathematical logic. It is this combination which makes possible software that can reason about data in more advanced ways than can relational DBMSs. This software is generally called an *inference engine*. And the better the ontology expressed in a database, the more extensive the knowledge that inference engines can extract from that database.

I'll probably be spending most of my time, over the next several years, explaining how all this works. It brings me full circle, however, back to my original concern, viz. what more than a sense of elegance distinguishes good data models from bad ones? For the answer is: a good data model is one with a good ontology.

More later. Much more.